2010

Threshold interval indexing techniques for complicated uncertain data

Andrew Knight

Threshold Interval Indexing Techniques for Complicated Uncertain Data

Andrew Knight

Computer Science

B. Thomas Golisano College of Computing and Information Sciences

August 31, 2010

Andrew Knight, student

Mangeet Rege, chair

Qi Yu, reader

Zack Butler, observer

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 incur high computational cost in addition to the I/O cost. In this paper, I present a novel indexing strategy focusing on one-dimensional uncertain continuous data, called *threshold interval indexing*. Threshold interval indexing is able to balance 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. This interval tree optimizes pruning by storing *x-bounds*, or pre-calculated probability boundaries, at each node. In addition to the basic threshold interval index, I present two variants, called the *strong threshold interval index* and the *hyper threshold interval index*, which leverage x-bounds not only for pruning but also for accepting results. Furthermore, I present a more efficient *memory-loaded* versions of these indexes, which reduce the storage size so the primary interval tree can be loaded into memory. Each index description includes methods for querying, parallelizing, updating, bulk loading, and externalizing. I perform an extensive set of experiments to demonstrate the effectiveness and efficiency of the proposed indexing strategies.

I. Introduction

The term *uncertain data* defines data collected with an inherent and distinctly quantifiable level of uncertainty [2]. Whereas values for *certain data* are given as exact constants, values for uncertain data are instead given by probability measures, most notably by probability distribution functions (PDFs). Uncertain data can be generated from many circumstances:

- Scientific measurements include margins of error due to limited instruments [2].
- Sensor networks are imprecise due to hardware limits [2].
- GPS is only accurate to within a few meters [12].
- Uncertainty in mobile object tracking can magnify errors in predictive queries [2], [52].
- Forecasting weather or economic data is based heavily upon statistics and probabilities [3], [38], [47].
- Aggregated demographic data only represent summaries, not actual data [4].
- Privacy-preserving data mining often introduces jitter to protect individuals [2].

- Lost information creates incompleteness in data [38].

Uncertain data is different from *erroneous data* in that uncertain data, though perhaps not precise, encapsulates a margin of error or uncertainty as part of its data model, whereas erroneous data is reported as-is and is assumed to be correct. Just as for certain data, uncertain data values may be continuous or categorical (discrete), based on the domain of possible values.

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 [11], [40], the U-tree [47], and 2D mapping techniques [1], [11]. 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 involves complicated computations. For example, computer simulations of cell signaling dynamics for biology research generate probability distributions with multiple peaks [31]. An example of such functions is given in Figure 1. Multimodal probability models have also been adopted in many real-world applications, such as cluster analysis [51].

Computing a PDF, such as multimodal probability, may involve high computational cost. Numerical approaches, such as Monte Carlo integration, have been exploited to improve the performance. Monte Carlo integration offers a practical technique for multidimensional integrals because regions can be difficult to define. Tao et al. proposed to use Monte Carlo integration for efficiently calculating probabilities given arbitrary PDFs [47]. First, the desired region to integrate is identified. A larger finite region, whose integral is much easier to calculate, is then defined around the desired region. N points are selected at random within the larger region. The integral is estimated by the fraction of points within the desired region times the integral of the larger region [21].

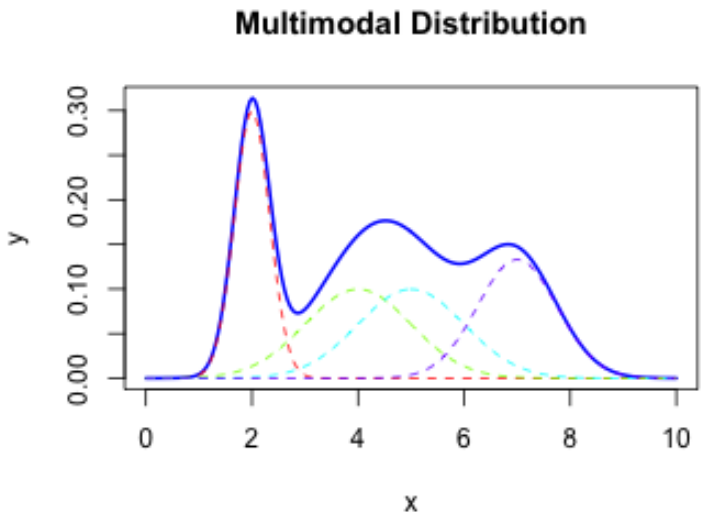


Fig. 1: A mixture of four Gaussian distributions, similar to ones in [31]. The four constituent distributions, shown by the dotted lines, are $(\mu = 2, \sigma = \frac{1}{3}), (\mu = 4, \sigma = 1), (\mu = 5, \sigma = 1)$, and $(\mu = 7, \sigma = \frac{3}{4})$.

Riemann sum, on the other hand, provides a better strategy for one-dimensional cases. Riemann sums numerically approximate integrals by calculating the combined area of N rectangles under a function's curve [46]. Larger numbers of rectangles mean a smaller error margin. As shown in Figure 2, Riemann sums are both faster and more accurate than Monte Carlo integrations for one-dimensional functions. It is also worth to note *accuracy is very important for indexing uncertain data.* If probabilities are not calculated accurately, then range queries may include invalid objects or exclude valid objects. Indeed, using Riemann sums is very similar to the argument presented by Agarwal et al. for using histograms with 2D mapping techniques [1]: histograms, as a series of rectangles, can represent any function to arbitrary accuracy and are more practical for collecting data.

Even though Riemann sums are faster than Monte Carlo integrations in one-dimensional cases, they still incur high computational cost. As shown in Figure 3, 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

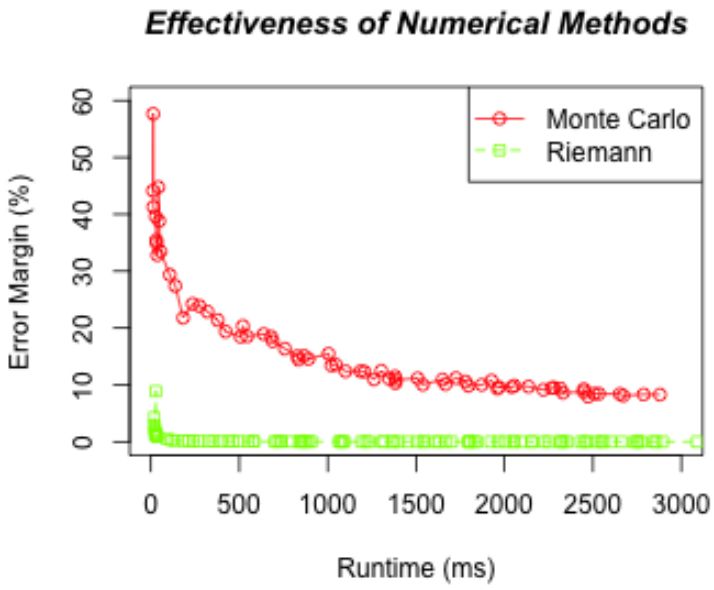


Fig. 2: Comparison between Riemann sums and Monte Carlo integration for 1000 random probability calculations on the multimodal distribution given in Figure 1.

disk I/Os and the CPU cost to achieve an optimal overall query performance.

In this paper, I present a new strategy for indexing one-dimensional uncertain continuous data called *threshold interval indexing*. It addresses the weaknesses of previous indexing structures, particularly for handling complicated PDFs, by treating uncertain objects as intervals and thereby leveraging interval tree techniques. I borrow 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 [11] but disregarded in favor of an R-tree with extra probability limits called x-bounds. I assert that x-bounds can just as easily be applied to interval trees to index uncertain data with special benefits.

The following list summarizes my contributions:

- I present *threshold interval indexing*, a new indexing approach for complicated one-dimensional uncertain continuous data.
- I provide *three structures* which apply threshold interval strategies:

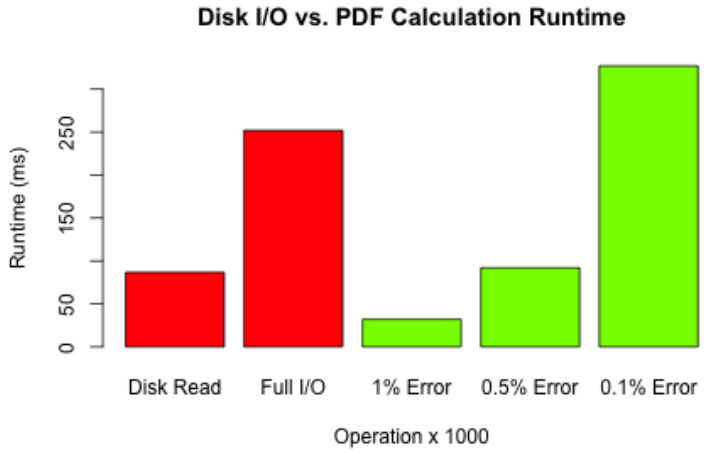


Fig. 3: Runtime comparison between disk I/Os and Riemann sums on the distribution from Figure 1. A disk read is the time taken to physically read one block of data (4096 bytes) from the disk. The full I/O is the time to both read and parse the data. The right three bars represent Riemann sums with different error margins.

  - The *threshold interval index* (TII), which applies x-bounds to nodes.
  - The *strong threshold interval index* (STII), which applies x-bounds to each object.
  - The *hyper threshold interval index* (HTII), which stores x-bounds as intervals.
- I provide efficient methods for *querying, building, and maintaining* the structures.
- I present a *memory-loaded* strategy to reduce storage size of each structure.
- I *experimentally prove the success of my indexing strategies* by comparing their performance results to results from existing strategies.
- I *identify appropriate scenarios* in which to use different threshold interval indexes based on experimental performance results.

The different indexes address unique optimizations. The TII presents a universal structure for threshold interval indexing techniques. The strong TII allows for fewer probability calculations by storing x-bounds for each object, meaning faster runtime. However, it requires more storage space. The hyper TII eliminates all probability calculations, but it puts stricter preconditions on query parameters. The memory-loaded TII loads the entire primary tree into memory because of its reduced size. This reduces the number of disk I/Os, but the primary tree is not as flexible

for updates.

The rest of the paper is organized as follows. Section II explains the motivation for developing new indexing strategies for complicated uncertain data. Section III presents the basic threshold interval index. Section IV presents the strong TII based on the design of the TII. Section V presents the hyper TII, a specialized TII which does not perform any probability calculation during a query. Section VI describes the memory-loaded strategy threshold interval indexing. Section VII gives a thorough analysis of performance test results for range queries. Section VIII discusses related work with uncertain data. Section IX concludes the paper and offers 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* [1], [11], [12], [38], [44].

A. Data Model

Definition 1: An *uncertain attribute* e is an attribute whose value is determined by a probability distribution function.

Definition 2: An *uncertain object* u is an object containing an uncertain attribute. Denote this attribute as $u.e$. If u has only one uncertain attribute, then it can be called one-dimensional. If $u.e$ is a continuous variable, then u can be called continuous.

Definition 3: The *uncertainty domain* for u is the domain for $u.e$'s PDF. If u is one-dimensional, then the uncertainty domain can be called an *uncertainty interval*. Although theoretically an uncertainty domain can be infinite, it should be made finite for practicality.

Definition 4: In the *uncertain object model*, a database table T may hold uncertain objects. Each object is stored in standard relational database tables. Table columns must be homogeneous: all attribute values in a column must be certain or uncertain.

Indexing continuous uncertain objects improves efficiency of *range queries* [1], also called *probabilistic threshold queries* [11]. Range queries are significantly different for uncertain data.

For certain data, an object is either within the query interval or not. For uncertain data, each object has a probability of being within the query interval, based on its uncertainty domain.

Definition 5: Given a database table T, a *query interval* $[a, b]$ for an attribute e, and a *threshold probability* τ, a *range query* returns all uncertain objects u_i from T for which $Pr(u_i.e \in [a, b]) \geq \tau$.

The range query definition above forms the problem statement. With no index, a query must calculate a probability for each object to determine if the object falls within the query interval. Naturally, an efficient index prunes many uncertain objects from a search to avoid unnecessary probability calculations, which, given the complexity of the PDFs, could save a lot of time.

B. External Interval Tree Index

Interval trees [13] 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. The first index's primary tree is a balanced tree over a set of fixed endpoints with a branching factor of $\sqrt{B}$ as the base tree, where B is the block size. The second index replaces the static interval tree with a *weight-balanced B-tree* [7] storing interval endpoints to achieve dynamic interval management. Note that its primary tree does *not* store the intervals, it only stores endpoints to control tree spread. In both indexes, each internal node v represents an interval I_v containing all of its child nodes' endpoints. Each interval I_v is divided into subintervals called *slabs* by the endpoint boundaries on v's immediate child nodes. When using this tree to index a set of interval objects I, an interval $i \in I$ is stored at the lowest node v in the tree such that i is not split across slab boundaries. Each node v stores these intervals in secondary structures for each slab boundary: B-trees normally or in an underflow structure if the number of segments is less than $B/2$ [7], [23]. These lists hold all intervals that cross the boundary on the left side, on the right side, and as a multislab. *Stabbing queries* are used to return results.

Since the endpoints in the first index are fixed, it can become unbalanced and therefore inefficient due to spread and skew in the input interval set. The second index, although much more complicated, adapts well to skew and to new inputs. However, the downfall of both interval indexes, as mentioned in [11], 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. Furthermore, although this external index is theoretically optimal, it is not always practical [35].

C. Probability Threshold Index

The *probability threshold index* (PTI) [11] allows range queries to prune more branches from searching than interval indexes allow. The PTI uses a one-dimensional R-tree as a base tree. Only leaves store uncertain objects. Each internal node has a *minimum bounding rectangle* (MBR) that encloses the narrowest boundaries $[L, R]$ for all child PDFs. Tighter bounds, called *x-bounds*, are also calculated for each node. X-bounds are the pair of boundaries (L_x, R_x) such that the probability an object attribute's value exists in $[L, L_x]$ or $[R_x, R]$ is equal to x [11]. Thus, when performing a range query for objects in $[a, b]$ with probability threshold τ, if, for a certain node, $\tau \geq x$ and $[a, b]$ does not overlap L_x or R_x at its right or left ends, then the node and its children may be pruned from the search.

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. Note that the U-tree is very much like a multi-dimensional PTI with additional pruning techniques [47].

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 [11]. Methods involving *variance-based clustering* are provided in [11] to solve this problem; however, they only work for PDFs that are *variance monotonic*. Furthermore, Cheng et al. 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, exacerbate overlap, as shown in Figure 4. Simply put, sloppy R-trees are inefficient, but optimal R-trees are very difficult to maintain. Strategies such as segment indexes and the SR-tree [25] address different interval lengths, and interval indexes handle skew very well.

When rectangles overlap, not all objects which fall completely within the query interval can

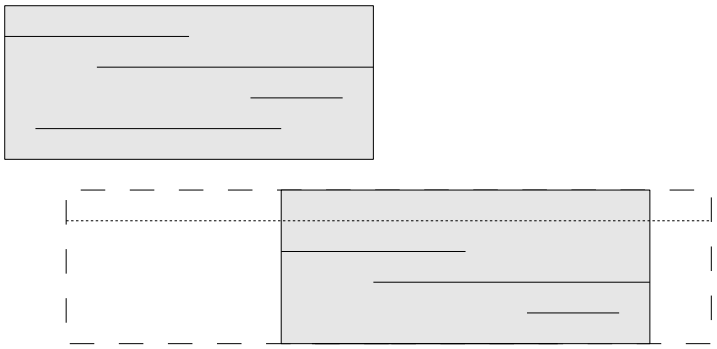


Fig. 4: MBRs can easily become skewed. The dotted rectangle shows how the bottom MBR must expand to accommodate the uncertain object denoted by the dotted line. These two MBRs now severely overlap.

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 [11]. Agarwal et al. then expanded 2D mapping techniques to histogram PDFs [1]. Histogram PDFs can easily be transformed into linear piecewise cumulative distribution functions (CDFs). A CDF F can then be transformed into a linear piecewise threshold function g, for which $g(x)$ gives the minimum value y such that $F(y) - F(x) \geq \tau$ for a preset probability threshold τ. Threshold functions are calculated for each uncertain object and turned into a set of line segments. A range query for an interval $[a, b]$ graphs the point (a, b) and returns all objects whose line segment threshold functions are below it.

The structures of the indexes presented in [1] manipulate the line segments. The half-plane range reporting technique partitions the line segments into sets of layers. Queries visit each layer in order until a layer surpasses the query interval. Fractional cascading improves visit time. The

segment tree, interval tree, and hybrid tree use the same notions presented in [7] about interval management and slabs to form optimized index structures.

2D mapping indexes 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. However, the application of interval tree techniques presented in [1] is a novel enhancement over techniques presented in [7].

III. Threshold Interval Index

The *threshold interval index* (TII) addresses the shortcomings of the indexes described in Section II. It is like a dynamic external interval tree, but 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. These advantages allow the TII to handle complicated uncertain data more aptly.

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. The weight-balanced B-tree must hold the following properties [7]:

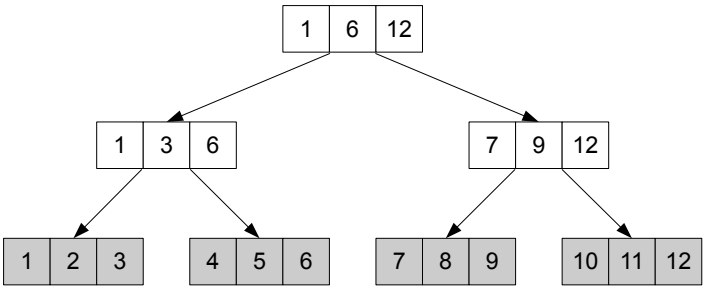


Fig. 5: A weight-balanced B-tree where $r = 2$ and $k = 3$. Leaves, shown in gray, store all uncertain object endpoints. The left and right end values at each node denote the node's interval. Parent nodes span their children's intervals. Node intervals never overlap.

- All leaves have the same depth and weight between k and $2k - 1$.
- An internal node on level l has weight less than $2r^l k$.
- An internal node on level l except for the root has weight greater than $\frac{1}{2}r^l k$.
- The root has more than one child.

These specifications guarantee that each internal node in the tree has a minimum of $(\frac{1}{2}r^l k)/(2r^{l-1}k) = \frac{1}{4}r$ child nodes and a maximum of $(2r^l k)/(\frac{1}{2}r^{l-1}k) = 4r$ child nodes [7]. The height of the tree is $O(\log_r(N/k))$ [7]. Adjusting r and k control the fan-out of the tree and, consequently, influence the size of the secondary structures stored at each internal node. Thus, the weight-balanced B-tree provides an effective way to dynamically manage intervals and spread. Figure 5 illustrates a weight-balanced B-tree.

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_i} (for $1 \leq i \leq c$), and a contiguous region of slabs, such as $I_{v_2}I_{v_3}I_{v_4}$, 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 and that child nodes are ordered.

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

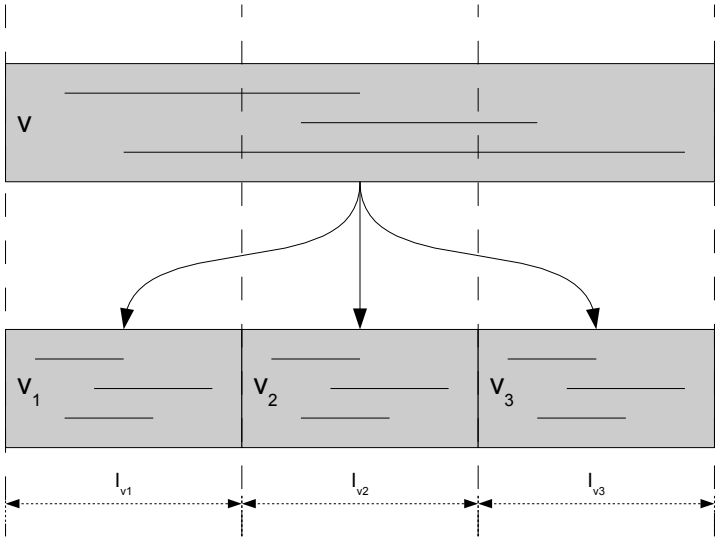


Fig. 6: A node v with three child nodes. The dotted lines denote slab boundaries. Note how objects are only stored within intervals which can completely contain them.

at exactly one node in the tree, as shown in Figure 6. 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 [11] and can easily be applied to the TII.

Definition 6: An *x-bound* is a pair of values (L_x, R_x) for a PDF $f(x)$ with domain $[L, R]$

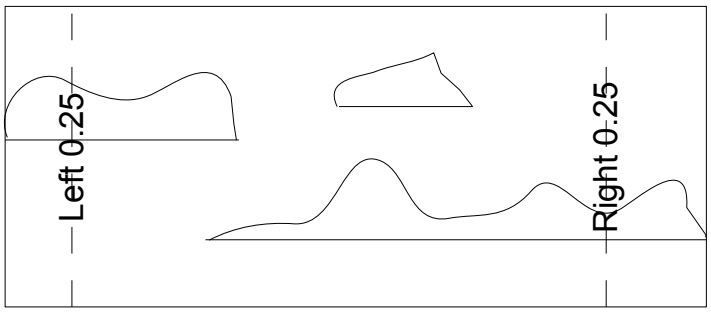


Fig. 7: X-bounds are calculated for each node based on uncertain objects' PDFs. The left and right 0.25-bounds are tighter than the MBR.

such that

$$x = \int_{L}^{L_x} f(x)dx = \int_{R_x}^{R} f(x)dx \tag{1}$$

L_x is the *left x-bound*, and R_x is the *right x-bound*. Note that x is a threshold 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 7. 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 [11]. 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 τ 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. Objects are checked at nodes along the stab to see if they belong to the result set.

Theorem 1: Let v be an internal node with child nodes v_i for $1 \leq i \leq c$. For each child node, let $[v_i.L, v_i.R]$ be the interval, and let $(v_i.L_x, v_i.R_x)$ be the left and right x-bounds. Furthermore, let Q be a range query executed with a query interval $[a, b]$ and a threshold $\tau \geq x$. If $v_i.R_x < a$ or $v_i.L_x > b$, then v_i and its children can be pruned from the search.

Proof: The maximum probability that any object stored in v_i has a value in $[v_i.R_x, v_i.R]$ is x, by the definition of an x-bound. If $v_i.R_x < a$, then the probability that any object stored in v_i falls in the query interval must be less than x. Since $x \leq \tau$, there is no way any object stored in v_i could meet the threshold probability. The same argument could be applied for the interval $[v_i.L, v_i.L_x]$ when $v_i.L_x > b$. ■

The appropriate child to choose for each step from parent to child has the minimum i for which v_i cannot be pruned from the search, as shown in Figure 8. Simply put, the leftmost unpruned child is picked. Remember, child nodes are ordered and do not overlap. A data structure holds references to all visited nodes so that nodes revisited during the right stab are not reevaluated.

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. The method for finding valid results relies on two theorems.

Theorem 2: Let Q be a range query with query interval $[a, b]$, and let u be an uncertain object with uncertainty interval $[u.L, u.R]$. If $u.R \leq a$ or $u.L \geq b$, then u is not a valid query result.

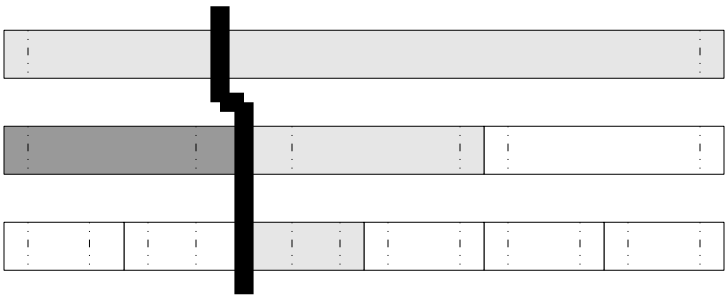


Fig. 8: A left stab is denoted by the thick black line. The stab visits the light gray nodes. The white nodes are not visited. The dark gray node is pruned based on x-bounds.

Proof: If $u.R \leq a$, then u is completely to the left of the query interval. If $u.L \geq b$, then u is completely to the right of the query interval. Either way, there is no overlap with the query interval and therefore a 0% chance of being within it. ■

Theorem 3: Let Q be a range query with query interval $[a, b]$, and let u be an uncertain object with uncertainty interval $[u.L, u.R]$. If $a \leq u.L$ and $u.R \leq b$, then u is a valid query result.

Proof: Since the uncertainty interval for u falls completely within the query interval, there is no chance that u could appear anywhere outside the query interval. ■

Between the secondary structures, only the right endpoint list is needed. The search on this list begins by finding the first object whose right endpoint is greater than a. Remember, this list is sorted, so a binary search can be performed. Any object whose right endpoint is less than a can be disregarded because of Theorem 2. Each object whose right endpoint is greater than a must be investigated. Any object whose endpoints are within the query interval is added to the result set because of Theorem 3. 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 9. The same strategy applies for the left boundary leaf. All valid objects are added to the result set.

It is interesting to note that the left stab may visit nodes whose intervals do not contain a. There could be a situation where $a \in I_{v_i}$ yet v_i is pruned. In that case, the path of the stab goes to the next leftmost interval after v_i which cannot be pruned, and a will be less than every uncertainty interval in the new node. (Remember, intervals represented by tree nodes of the same level do not overlap.) Thus, the starting point in the right endpoint list of every subsequent node

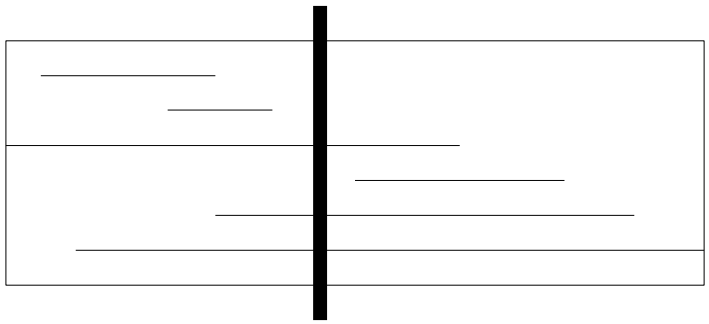


Fig. 9: Selecting objects from one node during a left stab. All objects to the left of the thick line can be disregarded. All objects intersecting the thick line require a probability calculation.

in the stab will always be the first element.

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.

Theorem 4: Let Q be a range query with query interval $[a, b]$. Let v_L and v_R be the boundary leaves returned by the left and right stabs. Let N be the set of nodes visited by the left and right stabs. Let S be the set of all nodes v such that v is between v_L and v_R and $v \notin N$. All uncertain objects stored at all nodes in S are valid query results.

Proof: Every node in the tree between the two boundary leaves, by definition, has an interval that overlaps $[a, b]$. There are two classes for these nodes: those which overlap the query interval's endpoints, and those which fall entirely within the query interval. The nodes in N overlap the query interval endpoints, but all of their valid results have already been discovered

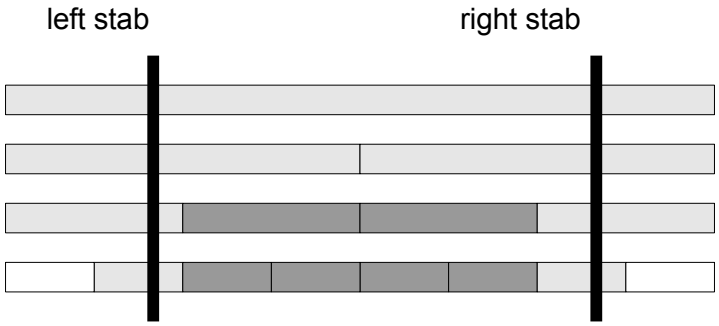


Fig. 10: A stab 'n grab query. The light gray nodes are visited during the stabs, and the dark gray nodes are visited during the grabs. Note how grabbed nodes fall completely within the query interval.

by the stabs, so they can be ignored outright. All other nodes (internal nodes as well as leaves) between the boundary leaves therefore fall within $[a, b]$. Remember, nodes at the same level in the tree have non-overlapping intervals. The nodes in S are either leaves which directly fall between the boundary leaves or internal nodes which fall between nodes of the same level in N. By Theorem 3, the uncertain objects stored at all nodes in S must be in the query interval. ■

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 must be done on the secondary structures. Figure 10 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 5: 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 $p \geq \tau$ in $O(kd \log_r(N/k) + T/k)$ 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 $O(r)$, then the total number of nodes in the tree is $O(\Sigma_{i=0}^{\log_r(N/k)} r^i) =$

$O(r^{\log_r(N/k)}) = O(N/k)$. 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 $O(kd \log_r(N/k))$ 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 $O(kd \log_r(N/k) + T/k)$. ■

5) Parallelization: Range queries on the TII can easily be parallelized because each object is stored at exactly one node and branches eventually diverge from each other. Define the *diverging node* as the first node in the tree for which the left and right stabs proceed to different children. In the worst case, the diverging node is the root; in the best case, there is no diverging node, meaning the left and right boundary leaves are the same leaf. The left and right stabs can start at the same time as one stab. After the diverging node is reached, the left and right stabs can be executed on separate threads, because they traverse nodes from that point forward. Furthermore, each branch in between the left and right stabs must be "grabbed," so another thread can be spawned for each branch. Each thread only visits nodes going from top to bottom.

C. Maintenance

When performing updates to the TII, both the primary tree and the secondary structures must be changed.

1) Insertion: When an uncertain object is inserted, both of its uncertainty interval endpoints are first added to the primary tree. The process for maintaining a weight-balanced B-tree is fully described in [8]. Let r be the branching parameter and k be the leaf parameter. An element is added to the appropriate leaf v. If v now has $2k$ elements, it is split across a boundary b into two leaves v and v' holding k elements each, and a reference to v' is stored in the parent node. Each time a split occurs, the parent may exceed its weight limit of $2r^l k$, so splits are cascaded up the tree as necessary. Splits for parent nodes may not always create equal halves, but the weight restrictions can always be met.

When a node v is split across a boundary b, its two endpoint lists must also be split. The set of objects stored at v is partitioned into three sets: (1) objects which are entirely left of

the boundary (right endpoint $\leq b$), (2) objects which are entirely to the right of the boundary (left endpoint $\geq b$), and (3) objects which intersect the boundary (left endpoint $\leq b \leq$ right endpoint). Objects in the first set remain at v. Objects in the second set are moved to the new node v'. Objects in the third set must be moved to the parent node. Since the lists are sorted, binary searches and hash tables can optimize the partition. This process is much easier than the process described in [8] because the TII only uses two lists per internal node.

Once the uncertain object's endpoints are inserted into the primary tree and all nodes are split, the object itself is inserted into the left and right endpoint lists of the appropriate node. A binary search on each list can optimize these insertions.

Finally, the x-bounds $(u.L_x, u.R_x)$ are calculated for the object. Let $(v.L_x, v.R_x)$ be the x-bounds for the node v into which the object is being stored. If $u.R_x > v.R_x$, then $u.R_x$ becomes the new right x-bound for v. Likewise, if $u.L_x < v.L_x$, then $u.L_x$ becomes the new left x-bound for v. The new object's x-bounds "tighten" the node's x-bounds in these two cases. Changes in x-bounds should cascade up to parent nodes as necessary.

Theorem 6: 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. An uncertain object can be added to I with an average time of $O(\log_r(N/k) + \log_2 k + d) = O(\log N + d)$ and a worst-case time of $O(kd \log_2 k \log_r(N/k) + d)$.

Proof: For the average case, a leaf will not need to be split. Finding the leaf into which to insert the object takes $O(\log_r(N/k))$ (the height of the tree), and inserting the object into the secondary structures takes $O(\log_2 k)$. Adjusting slab boundaries and x-bounds takes $O(d)$. Therefore, the average insertion time is $O(\log_r(N/k)) + O(\log_2 k) + O(d) = O(\log(N/k) + \log k + d) = O(\log N + d)$. For the worst case, nodes must be split. Since each node stores $O(k)$ objects, partitioning the list of objects takes $O(k)$. The parent node's list must be resorted because of its new elements, which takes $O(k \log_2 k)$. Thus, every split takes $O(k \log_2 k)$. Since parents might need to be split as well, the total number of splits is $O(k \log_2 k \log_r(N/k))$. X-bounds must be recalculated for every split node. After inserting the object into the right node as in the average case, the worst case time becomes $O(log_r(N/k)) + O(kd \log_2 k \log_r(N/k)) + O(\log_r(N/k) + \log_2 k + d) = O(kd \log_2 k \log_r(N/k) + d)$. ■

2) Deletion: Deleting an object is done the same way as for the dynamic external interval tree using global rebuilding [8], [37]. The object is deleted from the secondary structures, and

its endpoints in the primary tree are flagged. Once half of the objects have been deleted from the index, the whole index is reconstructed.

Changing the x-bounds stored at the node from which the object is deleted is difficult for the TII because x-bounds are not stored for each uncertain object. When an object is deleted, there are two alternative strategies for handling x-bound updates. The first strategy is very simple: do not change the x-bounds. Although the x-bounds will not be as tight as possible, they will still prune away nodes. No extra work must be done for this strategy. If the index is reconstructed, then x-bounds will automatically be updated for the new structure. The second strategy would be to replace the x-bounds if necessary. This means the x-bounds for every uncertain object stored at the node must be calculated and the tightest left and right from this set must be identified. If these new x-bounds are looser than the node's previous x-bounds, meaning the object removed had the tightest x-bounds in the node, then the node's x-bounds are set to these new values. Then, the x-bounds of all parents of the node must be checked as well. Certainly, the second strategy involves a lot of extra work. For most implementations of the TII, the first strategy is recommended.

Theorem 7: 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. An uncertain object can be deleted from I in $O(\log_r(N/k) + \log_2 k) = O(\log N)$ time.

Proof: Finding the object to delete is analogous to the process for insertion proved in Theorem 6, minus the $O(d)$ used to update x-bounds. ■

Corollary 1: A full deletion involving updated x-bounds can be performed in $O(d \log_r(N/k) + kd)$ time.

Proof: After the object is found and deleted from the secondary structures (which takes $O(\log_r(N/k) + \log_2 k)$), x-bounds for all objects at that node must be calculated, which takes $O(kd)$ time. Then, x-bounds of all parents must be checked against potentially new x-bounds, which takes $O(d)$ per parent for up to $O(\log_r(N/k))$ parents. The total deletion time is then $O(\log_r(N/k) + \log_2 k) + O(kd) + O(d \log_r(N/k)) = O(d \log_r(N/k) + kd)$. ■

D. Bulk Loading

Bulk loading the TII can be performed for the primary tree just like bulk loading can be performed on any B-tree [41]. The trick is to maintain proper weight and balance when building the primary tree. Remember, for branching parameter r and leaf parameter k, leaf weight must be between k and $2k-1$, and internal node weight for level l must be between $\frac{1}{2}r^l k$ and $2r^l k$.

Before bulk loading runs, the list of uncertain objects' uncertainty interval endpoints is sorted. Each endpoint is handled as an independent value; it is not associated with its other endpoint.

Bulk loading runs in levels, starting at level 0 (the leaves) and ending when there is one node at the top (the root). At level 0, the list of endpoints is divided into contiguous sublists of $2k-1$ elements each. If the last list has less than k elements, an element from previous lists can be forwarded until each list has at least $k+1$ elements. A leaf node structure is created for each sublist, using its minimum and maximum elements as slab boundaries. This leaf should have two empty lists for its secondary structures as well as fields for its interval boundary $[L, R]$ and x-bounds (L_x, R_x). Initially, $L_x = R$ and $R_x = L$ because the leaf does not yet hold any uncertain objects. (With these trivial x-bounds, the leaf will always be pruned. As soon as the first object is added, these x-bounds will be overwritten.)

For each subsequent level l, the list of slab boundaries is used to build internal nodes and create intervals for children. This list contains the slabs for each leaf in order. Whenever two adjacent slabs don't share the same right/left boundary, either boundary may be chosen to preserve continuous intervals. The list is divided into contiguous sublists of r elements each. Again, if the last list has fewer than r elements, elements can be added until it has at least $\frac{1}{4}r$. The sublists are turned into internal nodes, constructed much like the leaves mentioned for level 0. However, nodes built in the previous level are added as child nodes, in order, to these internal nodes. Each internal node receives r children. Because there is one slab in the remainder list for each child node, each parent node has exactly enough pointers for r children, and all children are guaranteed to have a parent. Ordering is important because the elements from each remainder list divide the child nodes into slabs. Furthermore, the fan-out of r preserves weight balance.

Theorem 8: When bulk loading a B-tree T, grouping elements by $2k$ at level 0 and by r at any level $l > 0$ makes T a weight-balanced B-tree.

Proof: Each leaf node appears at level 0, which means all leaves have the same depth. The

weight of each leaf is $2k-1$. Even if extra elements must be added to the last leaf in the list, each leaf will have at least k elements. This meets the boundary condition for leaves. For an internal node at level 1, there are r child nodes, meaning the weight is $r(2k-1) = 2rk - r < 2r^1k$. Even if extra elements must be added to the last internal node, there will be at the very least $\frac{1}{4}r + 1$ elements per node, and the ratio of children to parents remains one to one. For an internal node at level l, the weight becomes $r(2r^{l-1}k - r^{l-1}) = 2r^lk - r^l < 2r^lk$. Furthermore, $2r^lk - r^l = (r^l)(2k-1) \geq \frac{1}{2}r^lk$ because $2k-1 \geq \frac{1}{2}k$. (This lower bound also holds for $\frac{1}{4}r+1$). Both boundary conditions are met for internal nodes. Bulk loading stops when only one node is left as the root. All these meet the qualifications for a weight-balanced B-tree. ■

After the primary tree is constructed, each uncertain object is inserted into the secondary structures of its appropriate node as described in the bottom part of Section III-C1. Unfortunately, these objects cannot be efficiently added while bulk loading the tree because every uncertainty interval would need investigation every time a node is constructed. X-bounds are calculated from the bottom-up after all objects have been added. Calculating them each time an object is added results in a massive amount of recalculation, since x-bounds for child nodes may influence x-bounds for their parents.

Theorem 9: A set of N uncertain objects can be bulk loaded into a TII in $O(N \log N + Nd)$ time.

Proof: Sorting the list of uncertain objects takes $O(N \log N)$ time. For building the primary tree, let $T(n)$ represent the time to build the tree starting at the leaves, and let $S(n, i)$ represent the time to build the levels of nodes for all levels $l \geq i$. $T(n) = S(n/(2k), 1) + \Theta(n/(2k))$ because the level of leaves has $n/(2k)$ nodes. $S(n, i) = S(n/r, i+1) + \Theta(n)$ because n/r new nodes are created at level i, but n children from level $i-1$ must be attached to the new nodes. According to case 3 of the master method [13], $S(n) = \Theta(n)$ and $T(n) = \Theta(n)$. Once the primary tree is constructed, inserting an object into its appropriate secondary structure and calculating x-bounds is $O(\log N) + O(d) = O(\log N + d)$. Note that node splitting will not happen during bulk loading. For N objects, insertion takes $O(N \log N + Nd)$. Therefore, the total time to bulk load a TII is $O(N \log N) + \Theta(N) + O(N \log N + Nd) = O(N \log N + Nd)$. ■

E. 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 the primary tree, an uncertain object is represented only by its uncertainty interval endpoints, each of which is one unit of data. Each child node needs two units for endpoints, one unit for its block pointer, and two units for each set of x-bounds, meaning each node requires $3+2n$ units, where n is the number of x-bounds stored for the tree. The number of children per node still ranges from $\frac{1}{4}r$ to $4r$. Thus, to fit each primary tree node into one block, $k = \frac{1}{2}B$ and $r = \frac{1}{4(3+2n)}B$. The external weight-balanced B-tree then maintains the following properties:

- All leaves have the same depth and weight between $\frac{1}{2}B$ and $B-1$.
- An internal node on level l has weight less than $B(\frac{B}{4(3+2n)})^l$.
- An internal node on level l except for the root has weight greater than $\frac{1}{4}B(\frac{B}{4(3+2n)})^l$.
- The root has more than one child.

Each node uses one block to store intervals, x-bounds, and child pointers, and each secondary structure uses $O(1)$ blocks to store its objects. In [7], $k = \frac{1}{2}B$ and $r = \frac{1}{4}\sqrt{B}$ to guarantee that each internal node has $\Theta(\sqrt{B})$ children. The reasons and proofs for using these parameters are explained in [7]. The primary reason for selecting a fan-out of $\sqrt{B}$ for the external interval tree is to ensure the number of blocks is limited to $O(1)$ for each leaf, each slab list, and each internal node's pointers. If the uncertain objects stored in the TII are relatively small in size, then the same upper bounds apply for the number of disk I/Os. However, if the objects are large in size, which may be the case given how the PDFs are stored, then more blocks will be needed for secondary structures. Tight space bounds cannot be placed on the secondary blocks. It would be best to let the DBMS hold default PDF types and store arguments for these PDFs in the databases to reduce storage size; for example, the index would store three values for type, mean, and standard deviation for a normal PDF. Nicely, though, the number of blocks needed for the primary tree does not change, because it only stores interval endpoints. The value of r given for the TII is appropriate given the necessity of x-bounds. Naturally, extra metadata might force the values of k and r to be slightly smaller.

Theorem 10: The external TII can be stored using $O(N/B)$ blocks. Range queries can be

answered using $O(\log_B N + T/B)$ disk I/Os and $O(B \log_B N)$ probability calculations, and updates can be performed using $O(\log_B N)$ disk I/Os and $O(1)$ probability calculations.

Proof: Substitute $r = \frac{1}{4(3+2n)}B$ and $k = \frac{1}{2}B$. The bounds from [7] still remain for disk I/Os. Each node uses $O(1)$ blocks to store bounds and pointers to child nodes, and each endpoint list uses $O(1)$ blocks as well. Since there are $O(N/k)$ nodes, the tree is stored in $O(N/B)$ blocks. For range queries, $O(\log_B (N/B)) = O(\log_B N)$ I/Os are used for each stab and $O(T/B)$ I/Os are used for the grabs for a total of $O(\log_B N + T/B)$ I/Os. Probability calculations must only be performed on stabs, not grabs. The number of calculations for each block is $O(k) = O(B)$, so the total number of calculations is $O(B \log_B N)$. Update bounds for disk I/Os follow from the same analysis for stabs, and $O(1)$ probabilities must be calculated for the node into which the object is inserted. Splits do not require more probability calculations. ■

IV. Strong Threshold Interval Index

The threshold interval index applies x-bounds to primary tree nodes, which can only be used to prune nodes from a query during a stab. Uncertain objects can only be accepted as valid results without extra calculation during the grabs. If an object visited during a stab overlaps the query interval boundaries, an expensive probability calculation must be performed. The *strong threshold interval index* (STII) presented in this section allows uncertain objects to be accepted during stabs without probability calculations by storing x-bounds for each object. These x-bounds can be stored by making minor changes to the TII.

A. Structure

The structure for the STII is almost identical to the TII. The primary tree is the same. X-bounds are applied to nodes in the same fashion. Each node stores the objects contained exclusively within its interval. The difference for the STII is in the secondary structures.

For the STII, all objects for a node are stored in one list sorted by id. This list, called the *object map*, should store id, PDF, and any other data associated with the object, but not x-bound values. Then, for each x-bound threshold value, two additional lists are needed. These lists store the x-bound pair (L_x, R_x) and id for each object in the object map. One list stores x-bound pairs ordered from least to greatest using L_x, and the other list orders x-bound pairs using R_x, much like how the TII's object lists store objects based on endpoint values. In total, the STII needs

$1 + 2n$ lists per node, where n is the number of x-bound threshold probabilities for the index. It is mandatory that $x = 0.0$ and $x = 1.0$ (for 0% and 100% respectively) be included among the x thresholds for the correctness of the range query algorithm.

X-bound pairs are stored in separate lists from the objects because each x-bound list must be sorted for range queries. Since PDFs are arbitrary, object orderings might be different based on various x values. For example, a skewed-left distribution and a skewed-right distribution will have different x-bounds even if they share the same endpoints. The object map maps ids to objects, so objects can be quickly fetched based on any x threshold probability ordering. Separating x-bounds from objects also allows for fewer disk I/Os during a query, which will be explained in later subsections.

B. Range Query Evaluation

Again, range queries over the STII are nearly identical to range queries over the TII. The stab 'n grab search is the same: perform two stabs to find left and right boundary leaves, and grab all nodes in between. Pruning during the stab is also the same, since it uses the nodes' x-bounds. The query can be parallelized by the same method as for the TII, too, since parallelization relies on paths through the tree. The primary change for the STII is in how the query stabs an individual node.

A stab can automatically accept many objects as valid results when visiting a node in the STII. When a stab visits a node, it must fetch the object map. It must also fetch the appropriate x-bound lists. For a left/right stab, the x-bound list to fetch must be sorted by right/left endpoint, respectively. This is similar to the left and right endpoint lists for TII nodes. The x values to use depend on the query threshold τ. Two lists are needed, one for which $x \leq \tau$, called the *pruning list*, and one for which $x \geq \tau$, called the *accepting list*. Denote these x values as x_p and x_a, respectively. When pruning, $x_p \leq \tau$ ensures that all objects ignored do not qualify for the threshold probability. When accepting, $x_a \geq \tau$ ensures that all objects accepted have a probability of at least τ for being a valid result. This is why $x = 0.0$ and $x = 1.0$ must be included as threshold probabilities in the index, so that no matter what value τ the query uses, there will always be an x such that $x \leq \tau$ for pruning and $x \geq \tau$ for accepting. Figure 11 illustrates the difference between pruning and accepting thresholds. If the index stores an x for which $x = x_p = x_a = \tau$, then these two lists are the same, and only one list is needed.

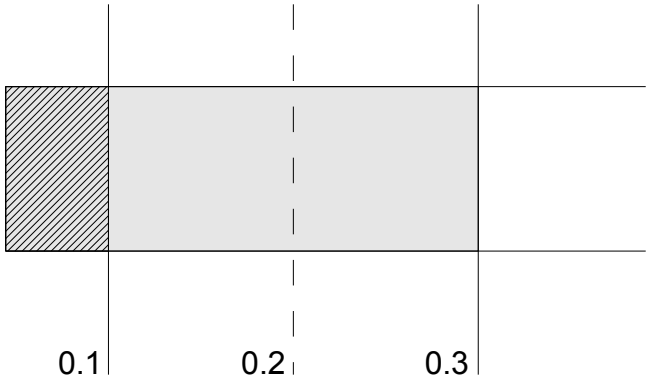


Fig. 11: Pruning vs. accepting x values. This uncertain object stores x-bounds for $x = 0.1$ and $x = 0.3$, but suppose $\tau = 0.2$. The left-0.1-bound cannot be used to accept this object because the diagonally-shaded region does not cover enough area, and the left-0.3-bound cannot be used to prune this object because the gray region holds enough area to satisfy τ at unknown points.

Otherwise, two different x-bound lists are needed, and the two values for x should be as close to τ as possible.

Once the appropriate x-bound lists are fetched, the objects are sorted in order of the pruning list. The first object whose x-bound falls within the query interval is found using a binary search. For a left/right stab, this is a right/left x-bound, respectively. This prunes all objects which cannot fall within the query interval. Every subsequent object in the list is then visited. Theorem 11 provides conditions for accepting objects as valid results:

Theorem 11: Let u be an uncertain object with uncertainty interval $[u.L, u.R]$ and x-bounds $(u.L_x, u.R_x)$. Let Q be a range query with query interval $[a, b]$ and probability threshold τ. If $x \geq \tau$, and if any one of the following four conditions is true, then u is a valid result for Q:

1) $[u.L, u.L_x]$ is entirely within $[a, b]$
2) $[u.R_x, u.R]$ is entirely within $[a, b]$
3) $1 - 2x \geq \tau$ and $[u.L_x, u.R_x]$ is entirely within $[a, b]$
4) $\int_a^b u.PDF(x)\,dx \geq \tau$

Proof: (1) (2) By definition of x-bounds, $\int_{u.L}^{u.L_x} u.PDF(x)\,dx = \int_{u.R_x}^{u.R} u.PDF(x)\,dx = x \geq$

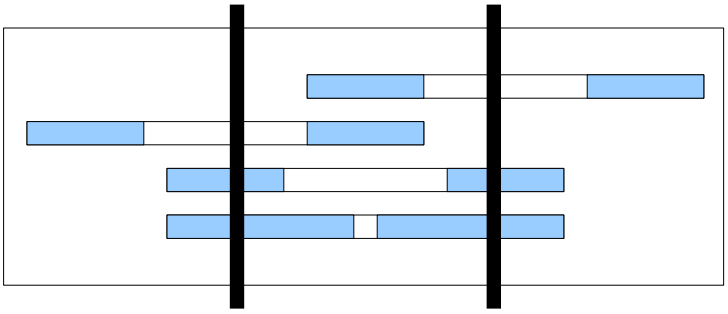


Fig. 12: STII node stab conditions. The two thick vertical bars represent the query interval, and shaded regions represent x-bound regions. The first object's left x-bound region is within the query interval (1), as is the second object's right x-bound region (2). The third and fourth objects' middle regions are entirely within the query interval. Suppose $x = 0.3$, which is likely for the third object: it can be accepted (3). Suppose however $x = 0.45$, which is likely for the fourth object: a probability calculation will be required (4). Note that in a real object list, these objects would be sorted by x-bounds, and objects would not have mixed x values.

τ. Therefore, both regions satisfy τ, and u is a valid result of Q. (3) The area under all of $u.PDF$ is 1.0. The sum of area under $u.PDF$ for the two intervals mentioned in (1) and (2) is $2x$. Therefore, the remaining area for the interval $[u.L_x, u.R_x]$ is $1 - 2x$. If $1 - 2x \geq \tau$ and $[u.L_x, u.R_x]$ is entirely within $[a, b]$, then u still qualifies as a result of Q. (If $x \geq 0.5$ and the x-bound interval is backwards, this condition is voided anyway because $1 - 2x \leq 0 \leq \tau$.) (4) $\int_a^b u.PDF(x)\,dx = \Pr(x \in [a, b])$. This is the problem statement from Section II. Figure 12 illustrates each of the four conditions. ■

Note that conditions (1), (2), and (3) use x-bounds stored in the accepting list (maintaining the precondition that $x_a \geq \tau$). This means that a probability calculation is not performed unless condition (4) is needed. This is why both the pruning list and accepting list are needed: the pruning list for the binary search, and the accepting list for the conditions.

The time bounds for a range query using the STII are the same as for the TII, given in Theorem 5 as $O(kd \log_r(N/k) + T/k)$. Despite optimizations with the pruning and accepting lists, the worst case runtime still involves some probability calculations. The proof is the same.

C. Maintenance, Bulk Loading, and Externalization

Methods for maintenance, bulk loading, and externalization for the STII are nearly identical to the methods used for the TII. Again, the difference is handling x-bounds for each object.

a) Insertion: When each object is added, its x-bounds for all x values must be calculated. The object is inserted into the object map of the appropriate node, and its x-bounds are inserted into the other lists. Each list insertion uses a binary search. Since these insertions all have the same time bounds, their summation is simply a constant factor. Therefore, the time bounds presented in Theorem 6 for the TII also hold for the STII.

b) Deletion: When an object is deleted, its x-bounds must also be removed from the secondary structures. They could be difficult to find for each x-bound list, so it would be best to calculate all x-bounds first and then use binary searches to find all values to delete. Theorem 7 can be applied to the STII for time bounds by similar arguments in its proof. Furthermore, a full deletion would be more feasible, since x-bounds are already calculated for each object.

c) Bulk Loading: The TII's bulk loading algorithm disregards individual objects' x-bounds after storing x-bounds for the nodes. The STII's bulk loading algorithm must store them instead. The best time to calculate x-bounds for each object is when the object is inserted into the index, after the primary tree is built. This way, each object and its x-bounds are inserted in order. When applying x-bounds to tree nodes, all values are already calculated. Again, time bounds are the same as given in Theorem 9.

d) Externalization: Secondary structures must be formatted differently for effective externalization. Each object map and list of x-bounds should have its own set of blocks. This way, during a query, only the blocks for the object map and appropriate x-bound lists must be read, saving time instead of reading all blocks. The bounds for index storage size and number of disk I/Os for a query remain the same as for a TII, given in Theorem 10.

V. Hyper Threshold Interval Index

The threshold interval index uses x-bounds to prune branches from a query search. The strong version uses x-bounds both to prune and to accept results at the object level. For both, however, probability calculations are always required when results can neither be pruned nor accepted, which occurs frequently. The *hyper threshold interval index* (HTII) presented in this section uses

x-bounds purely for accepting valid results and never performs any probability calculations. X-bounds for each object are stored as intervals in an interval tree. This structure is much different from the TII and STII. While the promise of no probability calculations is desirable, the HTII is limited in query interval size and can only hold one x-bound threshold probability.

A. Structure

As with other threshold interval indexes, the HTII has a primary tree with secondary structures. However, the key difference in the HTII is that it does not store uncertain object intervals in its primary tree. Instead, it stores *threshold satisfier intervals*.

Definition 7: Let u be an uncertain object with an uncertainty interval $[u.L, u.R]$ and x-bounds $(u.L_x, u.R_x)$. Given a probability threshold value x, a *threshold satisfier interval* (or *threshold satisfier* for short) is an interval $[L_s, R_s]$ within $[u.L, u.R]$ such that $\int_{L_s}^{R_s} u.PDF(x)\,dx = x$. The *left threshold satisfier* is the interval $[u.L, u.L_x]$, and the *right threshold satisfier* is the interval $[u.R_x, u.R]$.

A threshold satisfier for an uncertain object *satisfies* a query probability threshold. Thus, if a threshold satisfier falls within a query interval, and the probability thresholds are equal, then the object can be accepted as a valid result without a probability calculation at query time. Indeed, the intervals built using the accepting list during STII queries are, by definition, threshold satisfiers. Only two probabilities are calculated for x-bounds each time a new object is added to the index. Figure 13 depicts an uncertain object with two pairs of threshold satisfiers.

The HTII is built using one threshold probability value x. The HTII's primary tree stores both left and right threshold satisfier intervals for each uncertain object using x. This means four endpoints are inserted into the weight-balanced B-tree for each uncertain object. The secondary structures hold both threshold satisfiers and uncertain objects. If a tree node v covers an interval I_v, then all threshold satisfiers which fall entirely within I_v and overlap child node intervals are stored in the secondary structures for v. For each node, there are three secondary structures. A *left endpoint list* stores threshold satisfiers sorted by left endpoint. A *right endpoint list* stores threshold satisfiers sorted by right endpoint. (The "left" and "right" specifications for these endpoint lists refer to ordering of satisfiers. It does *not* divide left and right threshold satisfiers; both left and right satisfiers are stored together in both lists.) Finally, all uncertain objects which have threshold satisfiers stored in this node are stored in a third list sorted by id, called the *object*

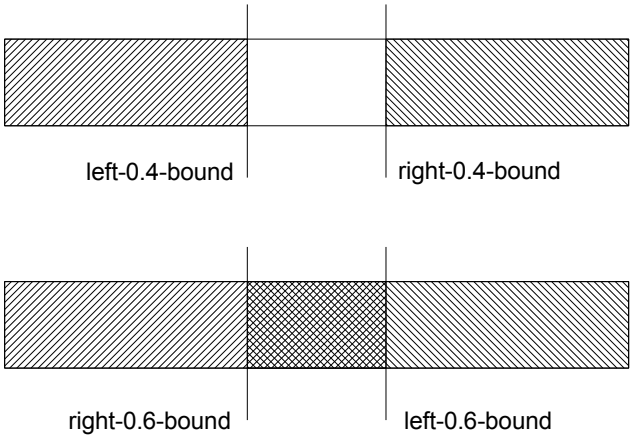


Fig. 13: Threshold satisfiers on an uncertain object for $x = 0.4$ (top) and $x = 0.6$ (bottom). Threshold satisfiers overlap when $x > 0.5$.

map. Each threshold satisfier must store the id of its uncertain object so that when threshold satisfiers are accepted during a query, the object to which it refers can quickly be retrieved from the object map.

Note that an object will be stored in two different nodes if its left and right satisfiers fall into different nodes. Thus, x-bounds are not stored for primary tree nodes, because x-bounds cannot be calculated when objects span two nodes. This distinguishes the HTII from the TII and STII and more closely resembles the dynamic interval index presented in [7]. Also note that only one probability threshold value can be stored using this index, since threshold satisfiers are the primary intervals stored, not the uncertain objects themselves.

B. Range Query Evaluation

As with all threshold interval indexes, the HTII uses a stab 'n grab approach for range queries. Although the conditions for finding results are simpler, queries must meet two preconditions in order to guarantee validity:

1) The query's probability threshold must equal the index's probability threshold.

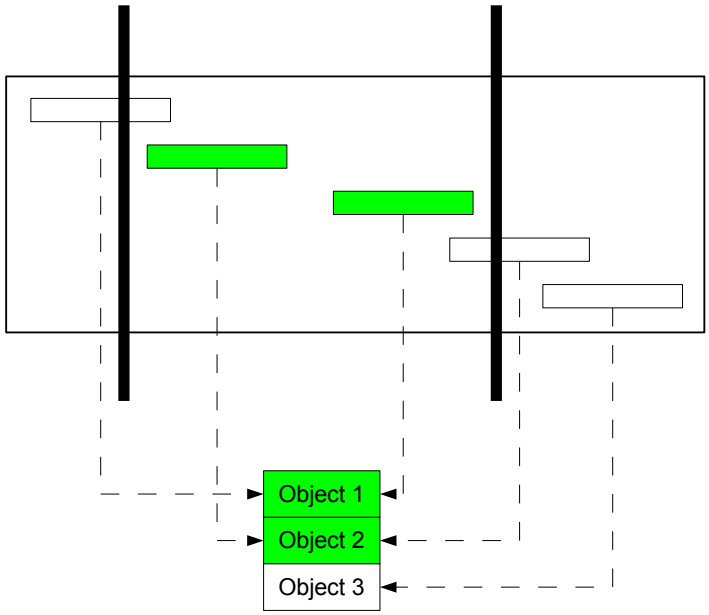


Fig. 14: A left stab for an HTII node. Two threshold satisfiers fall entirely within the query interval, so their objects are valid results. Note how only one of the threshold satisfiers for an object must be within the query interval to meet the query's probability threshold.

2) The query's interval length must be greater than or equal to the largest object interval length.

The query algorithm must be explained before the preconditions can be fully understood.

1) Stab 'n Grab Alterations: Range queries use a simpler stab 'n grab search than the version used for the TII. Since the HTII nodes do not store x-bounds, there is no x-bound pruning for branches. The procedure for finding boundary leaves and grabbing nodes in between remains the same. Again, the difference is in retrieving results from an individual node stab.

Stabbing an individual node is fairly straightforward, shown in Figure 14. The appropriate list of threshold satisfiers is needed: sorted by left endpoint for a left stab, and sorted by right endpoint for a right stab. Note how this is different than for the TII, in which any object interval that overlapped the query interval needed investigation. For the HTII, only threshold satisfiers which fall completely within the query interval are of interest, because, otherwise, they do not satisfy the query probability threshold. Thus, a left stab performs a binary search to find the first threshold satisfier whose left endpoint is greater than or equal to the query interval's left

endpoint (likewise with right endpoints for a right stab). Every subsequent threshold satisfier from this point forward must meet only one condition: it must fall entirely within the query interval. The ids for qualifying threshold intervals fetch the valid objects from the object map. No probability calculations are needed.

Duplicate results are an unfortunate side effect when using the HTII. If both left and right threshold satisfiers for an object fall within a query interval, an object might be reported twice by two different nodes. Simple hashing storage techniques or a results scan after performing the query can eliminate duplicate objects from the result set.

2) Preconditions: The first precondition is that the query's probability threshold must equal the probability threshold used to build the index. As mentioned earlier, the HTII can only be built upon one threshold value, in contrast to the TII and STII which can store multiple thresholds. If the query's threshold is larger than the index's threshold, then the threshold satisfiers are not large enough to give acceptable results. If the query's threshold is smaller than the index's threshold, then some objects can be accepted, but other objects may be overlooked since their threshold satisfiers would be too large and marked as invalid. Threshold satisfiers must have equal probability to the query threshold.

The second precondition for the HTII is that query interval length must be at least as large as the largest object interval length in the index. Theorem 12 provides the basis for this precondition.

Theorem 12: Let Q be a range query with interval $[a, b]$ and probability threshold τ. Let u be an uncertain object with uncertainty interval $[u.L, u.R]$. For u to be accepted or rejected as a valid result of Q using only its left and right threshold satisfiers, the smallest value possible for the query interval length $(b - a)$ must be u's uncertainty interval length $(u.R - u.L)$.

Proof: The left and right threshold satisfiers for u are $[u.L, u.L_\tau]$ and $[u.R_\tau, u.R]$, respectively. Each covers a probability of exactly τ. Suppose a query interval $[a, b]$ has a length less than the object interval length. Then, there is a chance that the query interval falls entirely within the object interval. The threshold satisfiers would be of no use, since they would both partially fall outside of the query interval. The only way to accept or reject u would be to perform a probability calculation. However, if the query interval length is equal to or greater than the object interval length, then at least one threshold satisfier must always fall within the query interval if u is a valid result. If u is valid and $a = u.L$, then both left and right threshold satisfiers for u fall within $[a, b]$, since $(b - a) \geq (u.R - u.L)$. If the query interval were then moved to the

right such that $u.L < a < u.R_\tau$, then the left satisfier would no longer fall within $[a, b]$, but the right satisfier would. An analogous argument can be made for the right side of u. Furthermore, if u is valid, then the left-most position for $[a, b]$ would be where $b = u.L_\tau$, and the right-most position for $[a, b]$ would be where $a = u.R_\tau$. Threshold satisfiers denote these extremes. ■

The HTII's primary tree is built upon threshold satisfiers. There is no effective way to retrieve objects for probability calculations if a query interval were to fall entirely within an object's interval. Both of the object's threshold satisfiers would either be pruned or rejected during a stab. The query must then check all objects, which would completely nullify the unique advantages of using threshold satisfiers. Therefore, query interval lengths must be larger than or equal to the size of the largest object interval length stored in the index. The size can be stored with the index, updated every time an object is added, and checked every time a query is run. This precondition is not very detrimental if most objects are roughly the same size and if queries are expected to be greater in interval length than the object size.

3) Time Bounds: The time bounds for the HTII are slightly lower than the time bounds for the TII and STII presented in Theorem 5, since no probability calculations are performed during a query.

Theorem 13: Let I be an HTII storing N uncertain objects, whose primary tree has branching parameter r and leaf parameter k. A range query Q with query interval $[a, b]$ and threshold τ can return all T uncertain objects stored in I which fall in the query interval with probability $p \geq \tau$ in $O(k \log_r(N/k) + T/k)$ time.

Proof: For N objects, $2N$ threshold satisfiers are stored. A stab visits $O(\log_r(2N/k))$ nodes from root to boundary leaf and must visit all objects stored at a node in the worst case. No probability calculations are performed. Hence, the stabs take $O(k \log_r(2N/k))$ time. In the worst case, the grabs take $O(2T/k)$ time, since both satisfiers might be visited for each valid object. Therefore, two stabs and all grabs can be performed in a combined time of $O(2k \log_r(2N/k) + 2T/k) = O(k \log_r(N/k) + T/k)$. ■

C. Maintenance, Bulk Loading, and Externalization

Just like for the STII, operations on the HTII are merely variations of those for the TII. Minor tweaks are needed for otherwise identical procedures.

a) Insertion: When each object is added, left and right threshold satisfier intervals must be calculated and constructed. They are inserted into the primary tree, and the object is inserted into the same node(s). Nodes are split if necessary. The object size must also be checked against the maximum object size. Time bounds presented in Theorem 6 still hold, since inserting two intervals only multiplies a constant factor of 2.

b) Deletion: When an object is deleted, both threshold satisfiers must be found and removed. They can easily be found by searching first for the object's endpoints and then by matching ids. Once the satisfiers are found, the copy of the object stored at that node is also deleted. There are no x-bounds to delete on the nodes. Revising the maximum object size would be difficult, so this can be left unchanged, just like x-bounds can be unchanged for the TII and STII. Theorem 7 can be applied to the HTII for time bounds.

c) Bulk Loading: The first thing to do when bulk loading a HTII is to calculate left and right threshold satisfiers for all objects. These endpoints are then used to form the nodes. Once nodes are built, objects are inserted as described in Section V-C0a. Again, there is no need to calculate x-bounds for nodes. The time bounds are the same as given in Theorem 9. Even though no x-bounds are calculated for nodes, they are calculated at the beginning for threshold satisfiers. Forming and sorting endpoints thus takes $O(Nd + N \log N)$, not $O(N \log N)$, and the other time bounds are the same.

d) Externalization: A different set of blocks should be used for each node's secondary structure lists. This layout mirrors the STII. As such, the bounds for index storage size and number of disk I/Os for a query remain the same as for a TII, given in Theorem 10. However, $O(0)$ probability calculations are performed for queries.

VI. Memory-Loaded Threshold Interval Indexing

Although the TII and its variants aptly balance 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. Manolopoulos et al. also suggest that the theoretically optimal external interval index is not always practically efficient [35]. This section introduces the *memory-loaded threshold interval index* (MTII) as an alternative TII to reduce the number of disk I/Os during range queries. The MTII is meant to be an external index. Since the primary tree is significantly smaller, all nodes can be preloaded into memory before

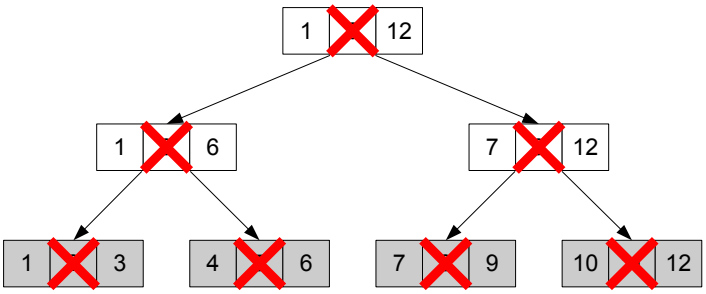


Fig. 15: The MTII's primary tree is like the TII's primary tree from Figure 6, except all object endpoints between a node's interval boundaries are removed after construction.

the query runs to improve runtime. The memory-loaded strategy presented here can also apply to the STII and the HTII.

A. *Updated Structure*

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, as shown in Figure 15. 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, $r = 2$ and $k = \frac{1}{2}B$. There is no reason to change the leaf parameter k from the value suggested in [7], since k controls the spread of objects at the bottom level of the tree. The branching parameter should be set to $r = 2$ to make the primary tree a binary tree. Since the whole primary tree will reside in memory, increased fanout is unnecessary.

Storing the primary tree is trickier for the MTII, since one block will hold more than one node. 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

TABLE I: Storage Size for MTII Primary Tree

Objects	Nodes	Node Storage	Blocks	With X-Bounds
10000	80	1.92 kB	1	2
100000	800	19.2 kB	5	10
1000000	8000	192 kB	47	94

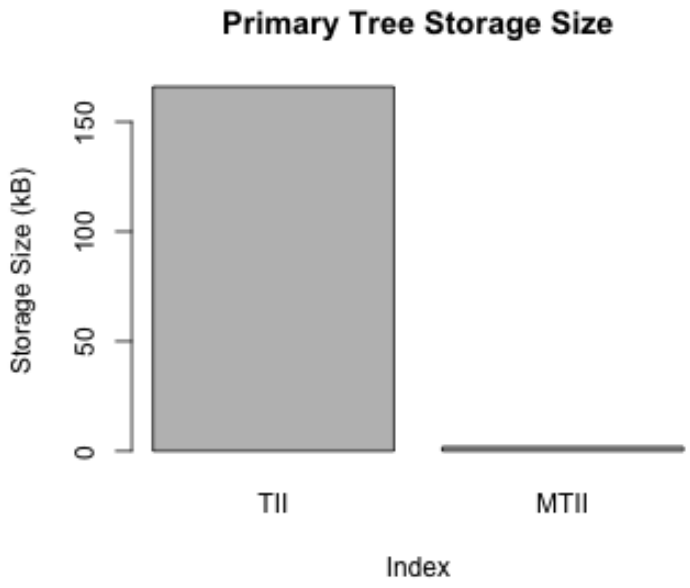


Fig. 16: Size Comparison for 10000 Objects and 1 X-Bound

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. Table I gives reduced storage sizes for the MTII primary tree. Figure 16 illustrates the drastic reduction in storage size between the TII and the MTII.

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 x 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.

B. Queries, Maintenance, and Bulk Loading

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.

Update methods are simpler for the MTII. Inserting an object is the same: find the appropriate node based on intervals and store the object in the secondary structures. The primary tree is not updated because extra endpoints are not stored in the nodes. Once the secondary structures are too large, meaning they use more than one or two blocks of data, then the node can be split using standard binary tree procedures. Intervals and x-bounds must also be updated. Deletion follows similar guidelines as for the TII.

The bulk loading algorithm is also similar. The difference for the MTII is that after the object endpoints are partitioned for forming leaves, only the minimum and maximum values are stored for each leaf. The rest of the process is identical. Nothing needs to change for constructing the internal nodes because each one's size depends on r, the branching parameter.

Time bounds for the MTII are the same as for the TII. Although primary tree nodes are far smaller, secondary structures are unchanged. Thus, the bounds for disk I/Os are unchanged.

The MTII is slightly more limited than the TII, however. Although the MTII can handle updates, it is not a fully dynamic interval index. The primary tree does not store all uncertain objects' endpoints and therefore cannot automatically adjust tree node intervals as objects are added. Instead, nodes split more liberally when secondary structures become too large. Loss of full dynamization makes the MTII less suitable for updates which skew balance away from its existing uncertain objects. In this way, it is somewhat similar to the fixed-endpoint tree interval index [7].

C. Memory Loading for Strong and Hyper TIIs

The principles for loading the primary tree into memory can easily be applied to the strong and hyper threshold interval indexes. These new indexes are called the *strong memory-loaded threshold interval index* (SMTII) and the *hyper memory-loaded threshold interval index* (HMTII). For both indexes, $r = 2$ in the primary tree. There is no x-bound tree for the HMTII. Secondary structures are not changed.

VII. Experimental Results

This section presents an analysis of experimental results for the proposed indexes' range query performances. The experiments focus on range query performance because range queries are the primary point of interest for threshold interval indexes. The probability threshold index [11] serves as the benchmark against which threshold interval indexes are tested. All indexes are externalized with a constant block size of 4096 bytes.

Range query performance is measured by three primary metrics:

- number of disk I/Os
- number of probability calculations
- runtime (in milliseconds)

The number of disk I/Os is the traditional metric for database performance, but the number of probability calculations must also be considered since PDFs can be complicated. Thus, total runtime is affected. All tests tabulate these three metrics.

A. Test Model

The experiment is divided into a set of tests. Each test evaluates range query performance for different scenarios. For example, the `Same` test generates a data set in which all objects have the same interval length. A test is given the following:

- a set of index types
- a set of probability thresholds
- a PDF type
- a set of uncertain object and query interval parameters

Each test randomly generates a synthetic set of uncertain objects based on the object parameters. These parameters specify endpoint boundaries, interval lengths, and number of objects.

Each object in the set has two attributes: a unique id and a PDF. One index of each type is constructed from this one common object set and the probability threshold set. 100 range queries are then randomly generated based on given parameters. Each query runs against each index for consistency. Performance is measured by number of disk I/Os, number of probability calculations, and total runtime.

For this experiment, each test is run twice for two types of PDFs. Just like for previous tests against the PTI, one PDF used is a uniform PDF [11]. The second PDF is the multimodal Gaussian distribution shown in Figure 1, 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 use Riemann sums. Each Riemann sum uses 1000 rectangles to keep the average error margin around 0.1%.

Tests are partitioned into two rounds: one including and one excluding the hyper threshold interval index. Partitioning is required in order to maintain consistency. The hyper TII has more restrictive query preconditions and therefore cannot handle arbitrary queries like the other indexes. The index types, probability thresholds, and query interval parameters between rounds must be different.

The experiment is implemented as a standalone simulation. All indexes are programmed in Java 6 and share a common code base. Performance tests are written using AspectJ, an aspect-oriented extension for Java, which allows performance data to be tabulated very easily without refactoring the original code. Tests are run on a Macbook Pro booting Mac OS 10.6.4 with a 2.4 GHz Intel Core 2 Duo processor and 4 GB of RAM.

B. Testing Round 1: Arbitrary Queries

The first round of tests investigate arbitrary query performance. Table II lists test names and descriptions, and Table III lists test parameters. Note that query intervals can be of any length or value within the object value interval. The indexes used for all tests are a perfect PTI, a practical PTI, the TII, the MTII, and the SMTII. Only the memory-loaded version of the strong TII is tested because, as it will be shown, memory-loaded indexes always outperform non-memory-loaded versions for static construction. Each index is constructed with the probability threshold values $\{0.1, 0.3, 0.5, 0.7, 0.9\}$.

TABLE II: Round 1 Tests

Test	Description
`Same`	object uncertainty intervals have the same length
`Different`	object uncertainty intervals have different length
`Dense`	many objects overlap
`Sparse`	objects are spaced out
`Threshold`	same as `Different`, but varies probability threshold
`Ratio`	varies query interval length versus object interval length

TABLE III: Round 1 Test Parameters

Parameter	Same	Different	Dense	Sparse	Threshold	Ratio
Num Objects	10000	10000	10000	10000	10000	10000
Min Object Value	0	0	0	0	0	0
Max Object Value	10000	10000	1000	1000000	10000	10000
Min PDF Length	100	50	1	1	50	10
Max PDF Length	100	500	100	10	500	10
Min Query Endpoint	0	0	0	0	0	0
Max Query Endpoint	10000	10000	1000	1000000	10000	10000
Min Query Length	1	1	1	1	1	varies
Max Query Length	10000	10000	1000	1000000	10000	varies

1) Perfect versus Practical PTI: Unfortunately, no optimal minimum bounding rectangle (MBR) strategy is proposed for the PTI [11]. The experimental PTI sorts all objects by their left endpoints and partitions them into leaves for bulk loading. This would most likely represent a "perfect" PTI. However, a more practical PTI goes through insertions and deletions, which will stretch MBRs. To create a more "practical" PTI, 10 objects are separated from the dataset and inserted into random leaves during bulk loading. Remember, since the TII and MTII are interval trees, they do not experience the same skew problems as the PTI. Figure 17 clearly

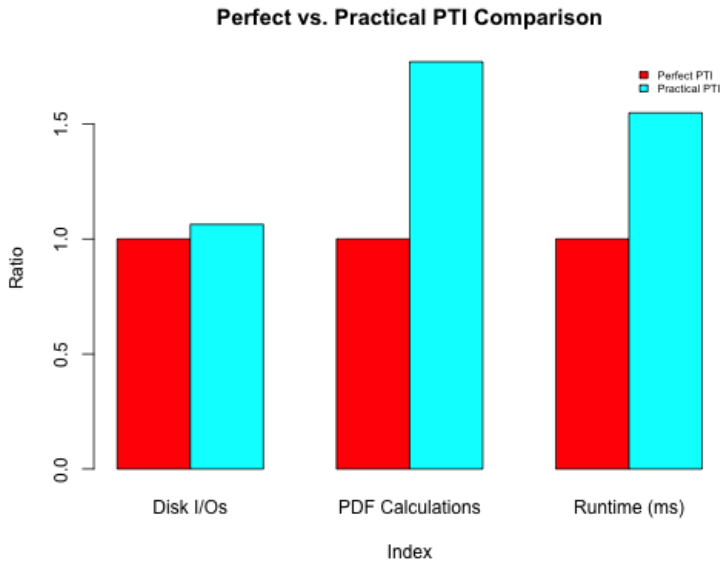
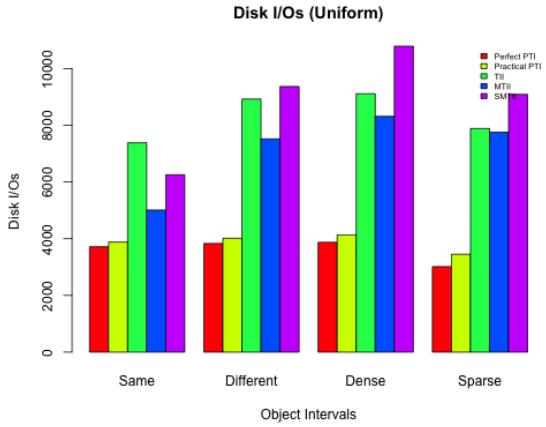


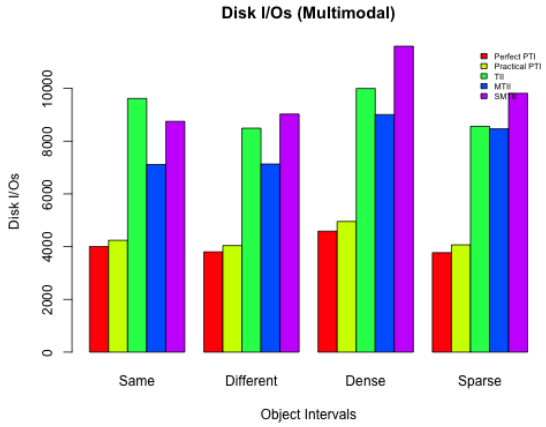
Fig. 17: Perfect vs. Practical PTI Performance

shows how a practical PTI does not perform as well as a perfect PTI. Although the number of disk I/Os is only slightly higher, the number of probability calculations is almost double, which sags runtime. Both practical and perfect PTIs are used in testing.

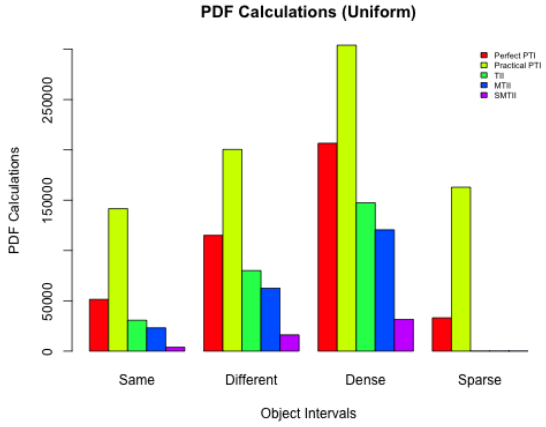
2) Effect of Object Spread: The `Same`, `Different`, `Dense`, and `Sparse` tests all test the spread of uncertain objects. Figures 18 gives results for these tests for the probability threshold $\tau = 0.3$. It is clear that object spread and size significantly affects performance. All indexes have worse performance for objects of different lengths and for densely clustered objects. What is interesting is the difference in performance metrics. The three threshold indexes generally use about 1.5 to 2 times as many disk I/Os as the two PTIs. Both PTIs are far surpassed, however, in regards to the number of probability calculations. The TII and MTII make roughly the same number of calculations, which are consistently about two-thirds as many as the perfect PTI and a third to half as many as the practical PTI. The SMTII makes hardly any calculations at all, which makes sense since it stores x-bounds for each object. This number is most staggering for sparse indexes: the threshold interval indexes make relatively no calculations. Overall, the total runtime favors threshold indexes for complicated probability functions, particularly the MTII and SMTII.



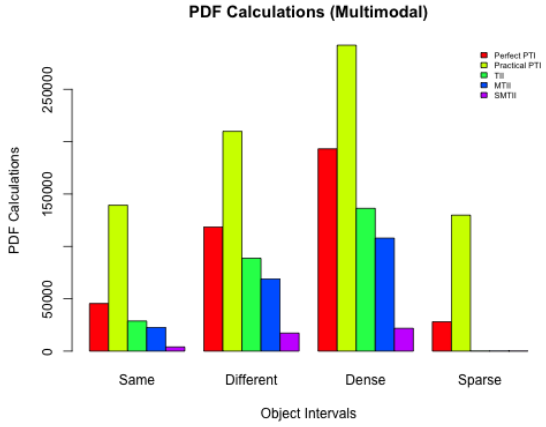
(a) Disk I/Os for Uniform PDFs



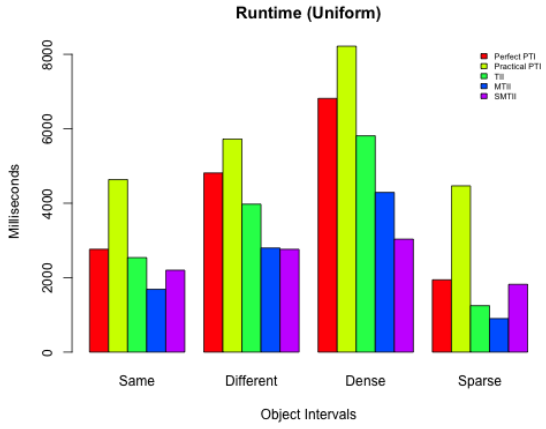
(b) Disk I/Os for Multimodal PDFs



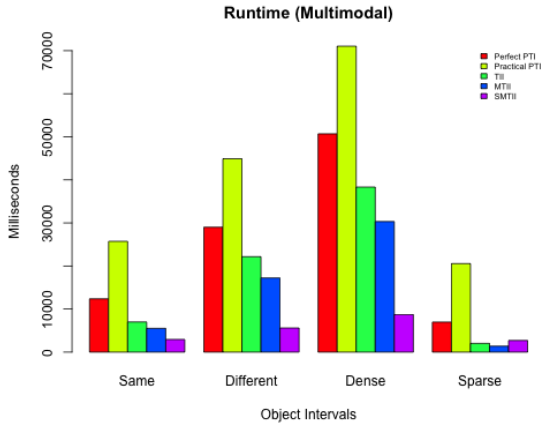
(c) Uniform PDF Calculations



(d) Multimodal PDF Calculations



(e) Runtime for Uniform PDFs



(f) Runtime for Multimodal PDFs

Fig. 18: Performance results for `Same`, `Different`, `Dense`, and `Sparse` tests.

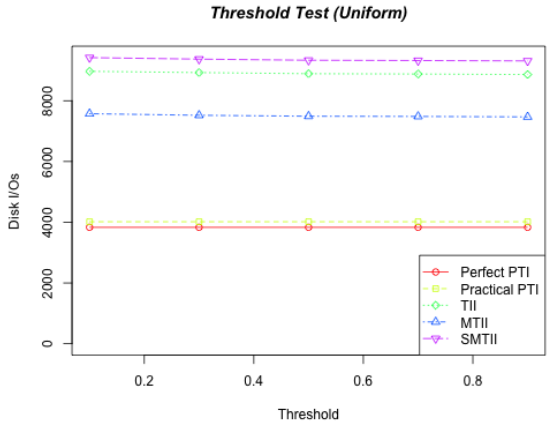
The trends between uniform and multimodal PDFs are generally the same for disk I/Os and probability calculations. This is not too surprising, because PDF shape has only a small affect on index structure. The major difference is in total runtime, as seen in Figures 18e and 18f. Since the multimodal PDF is more complicated, calculations take longer. Thus, the margin by which the threshold interval indexes outperform the PTI is much larger for multimodal PDFs than for uniform PDFs.

The SMTII's results are also significant. Relative to the other indexes, it consistently makes fewer PDF calculations. This means it can handle more complicated PDFs even better than the other threshold interval indexes. Comparing Figures 18e to 18f confirms this notion: while SMTII results are on par with (and sometimes worse than) the MTII and TII results for uniform PDFs, the SMTII performs staggeringly better on multimodal PDFs. Interestingly, its performance for the `Dense` and `Sparse` tests are opposite. The SMTII clearly handles densely clustered objects much better than other indexes, but it is slower for sparsely clustered objects. This makes sense because object-level x-bound pruning is most effective when similar objects are close together. However, for sparsely clustered objects, few probability calculations are needed anyway, so the toll for disk I/Os makes the SMTII slower than the TII and the MTII.

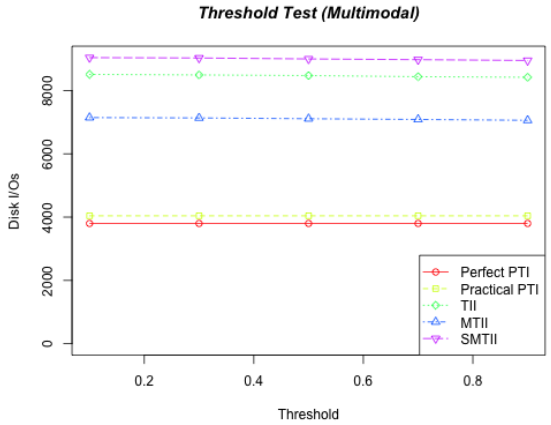
3) Effect of Probability Threshold: The `Threshold` test uses the same test parameters as the `Different` test, but it runs the query set on multiple probability thresholds. Figure 19 gives the results. Consistently, probability threshold does not have much effect on any of the indicators of performance, for either PDF. There are no significant differences in results for different threshold values, only minor variations

4) Effect of Query Interval Size: The `Ratio` test investigates the effect of query size relative to the objects' interval sizes. It runs random queries with a fixed query interval length for different ratios of query interval length to object interval length. For example, a ratio value of 3 means that the query interval length is three times as large as the object interval length. Each object has a constant interval length of 10 to make ratios consistent. Indexes are built only once, and 100 queries are constructed for each ratio value.

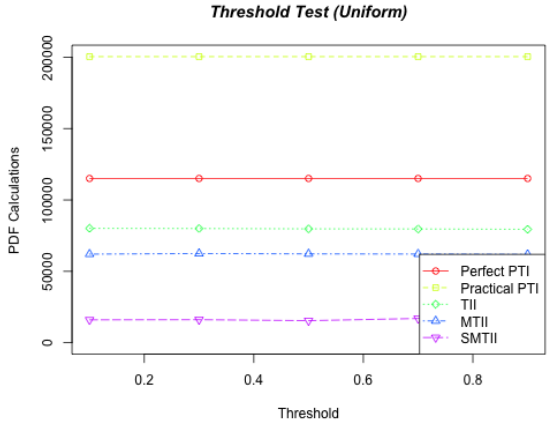
Figure 20 gives the results for the `Ratio` test. Results are consistent with the other tests' results. It is interesting to note what happens as the query interval ratio is increased. Naturally, the number of disk I/Os for each index increases linearly, since more objects are fetched for a larger query interval. However, the trend for probability calculations diverge between the PTIs and the



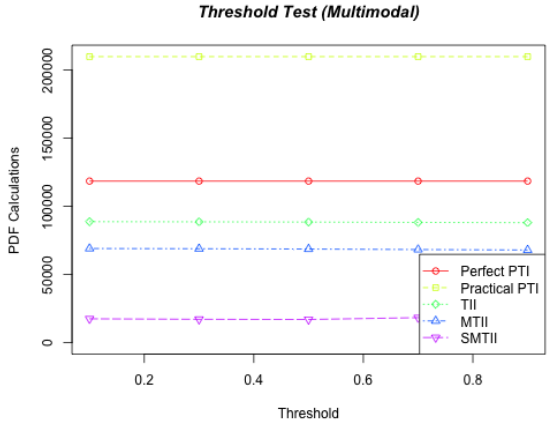
(a) Disk I/Os for Uniform PDFs



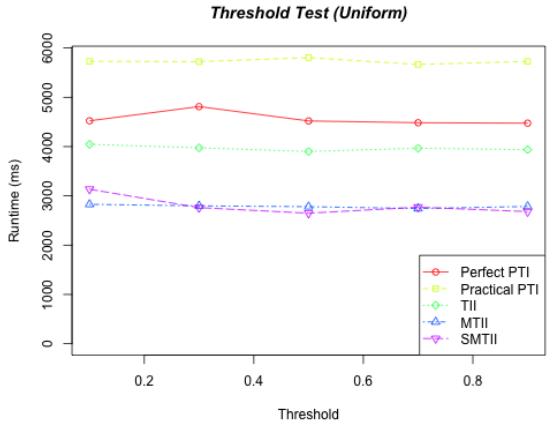
(b) Disk I/Os for Multimodal PDFs



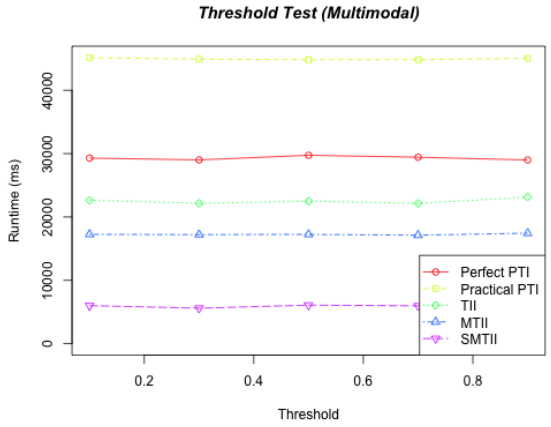
(c) Uniform PDF Calculations



(d) Multimodal PDF Calculations

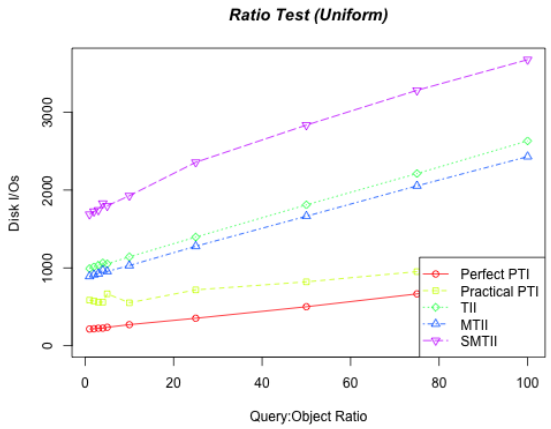


(e) Runtime for Uniform PDFs

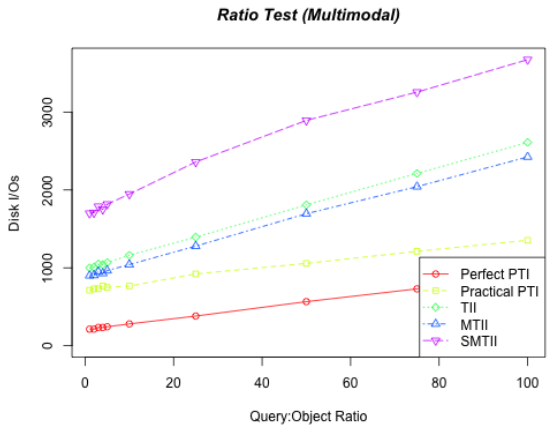


(f) Runtime for Multimodal PDFs

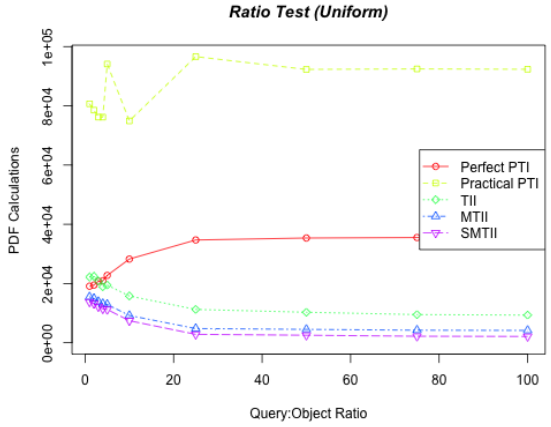
Fig. 19: Performance results for the `Threshold` test.



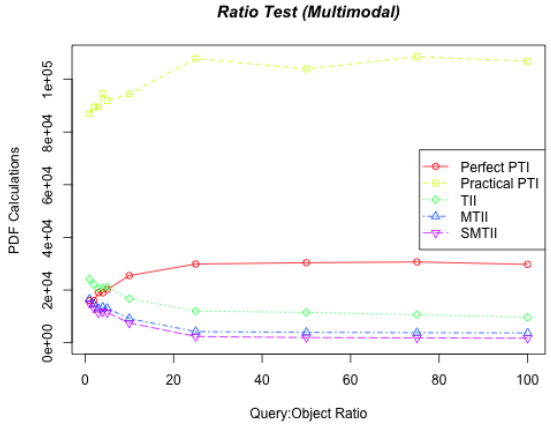
(a) Disk I/Os for Uniform PDFs



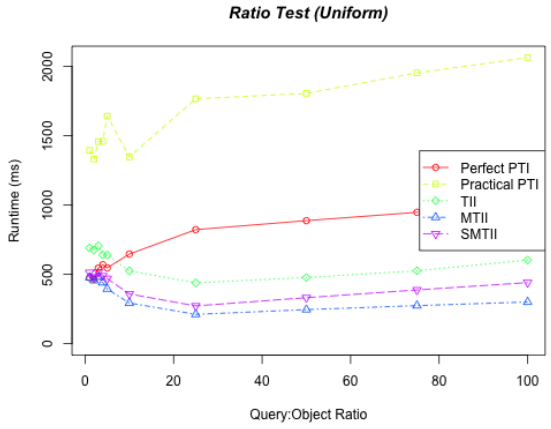
(b) Disk I/Os for Multimodal PDFs



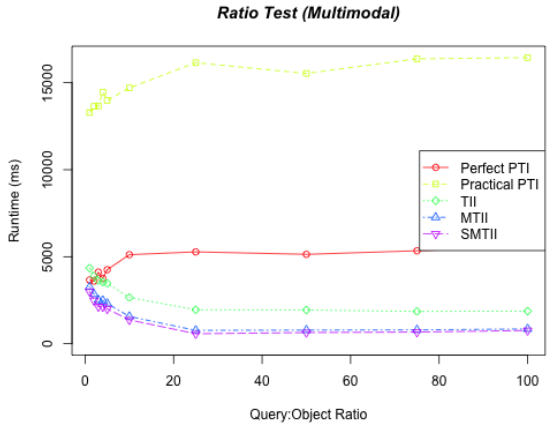
(c) Uniform PDF Calculations



(d) Multimodal PDF Calculations



(e) Runtime for Uniform PDFs



(f) Runtime for Multimodal PDFs

Fig. 20: Performance results for the `Ratio` test.

TABLE IV: Round 2 Test Parameters

Parameter	SameHyper	DifferentHyper	DenseHyper	SparseHyper	ThresholdHyper	RatioHyper
Num Objects	10000	10000	10000	10000	10000	10000
Min Object Value	0	0	0	0	0	0
Max Object Value	10000	10000	1000	1000000	10000	10000
Min PDF Length	100	50	1	1	50	10
Max PDF Length	100	500	100	10	500	10
Min Query Endpoint	0	0	0	0	0	0
Max Query Endpoint	10000	10000	1000	1000000	10000	10000
Min Query Length	100	500	100	10	500	varies
Max Query Length	10000	10000	1000	1000000	10000	varies

threshold interval indexes. For the PTI, the number of calculations increases asymptotically. For the TII and MTII, the number of probability calculations decreases asymptotically towards a very low constant value. The decrease in runtime with larger query intervals is a result of grabs: wider queries mean more nodes in between which do not need any probability calculations. The number levels off after a ratio value of about 25 for all indexes. Runtime levels off for multimodal PDFs, but for uniform PDFs, runtime continues to make small increases even after probability calculations stop increasing.

These results imply that threshold indexing is still superior when query sizes are large in respect to the average object size. This benefit is greatest after query sizes are about 25 times as large as object sizes. It is interesting to see that the perfect PTI and threshold interval indexes all have roughly the same performance for small ratio values (close to 1). It also shows again that the MTII and SMTII are more efficient than the TII.

C. Testing Round 2: Modified Queries for the Hyper TII

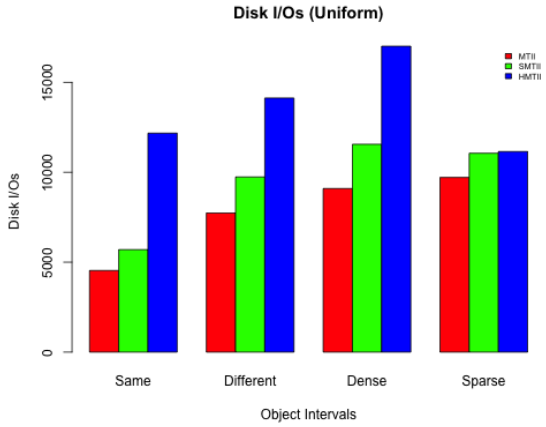
The second round of tests compares the hyper TII to the other threshold interval indexes. This round of tests only includes the MTII, SMTII, and HMTII. Only memory-loaded indexes are tested because they are more efficient. Including other indexes would also be redundant. Tests

are the same as given in Table II with the suffix `Hyper` appended to each name, but the new parameters are given in Table IV. The query interval parameters ensure that the smallest query size is larger than the largest object size. Except for the `ThresholdHyper` test, each index is built with only one probability threshold, $\tau = 0.3$. Naturally, this subsection will focus on analyzing the HMTII, since the MTII and SMTII were evaluated in the previous subsection.

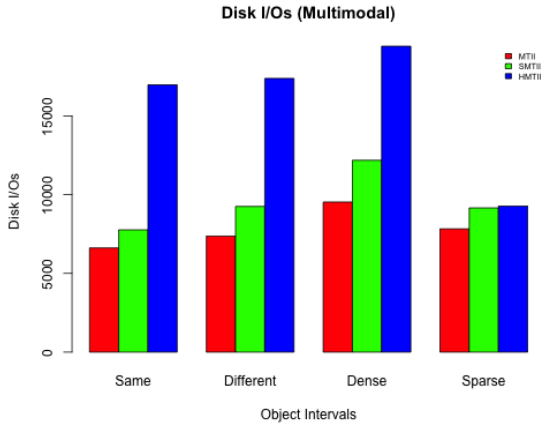
1) Effect of Object Spread: The `SameHyper`, `DifferentHyper`, `DenseHyper`, and `SparseHyper` tests again all test the spread of uncertain objects. Figures 21 gives results for these tests for the probability threshold $\tau = 0.3$. The HMTII's results are very interesting when compared to the MTII and SMTII. PDF type doesn't affect the relative number of disk I/Os and probability calculations, shown when comparing Figure 22a to Figure 22b and when comparing Figure 22c to Figure 22d. The HMTII uses roughly one and a half to two times as many disk I/Os as the MTII and the SMTII, except for the `SparseHyper` test in which the HMTII uses about as many disk I/Os as the SMTII. This not surprising since each object is stored with two intervals and might be copied twice. As expected, the HTMII does not perform any probability calculations.

The disk I/Os and probability calculations are reflected in the runtime. Obviously, HTMII runtime is dominated exclusively by the number of disk I/Os. Runtime is unaffected by PDF type, as shown in Figure 22e and Figure 22f, which both report nearly identical values for HMTII runtime. Thus, the HMTII only gains an advantage over other indexes when PDFs are significantly complicated. Figure 22e shows that the HMTII is sluggish for uniform PDFs, even when calculated using Riemann sums. However, HMTII performance is on par with SMTII performance for the more complicated multimodal PDF, as shown in Figure 22f.

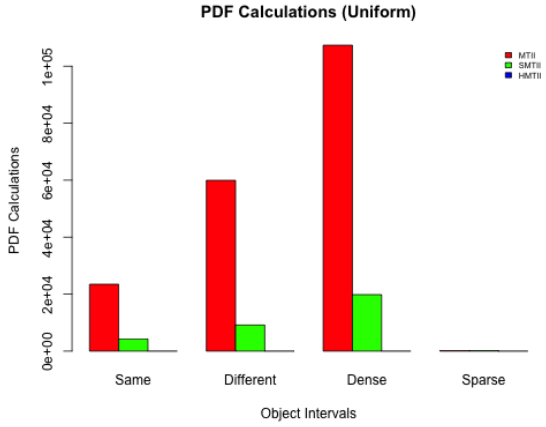
2) Effect of Probability Threshold: The `ThresholdHyper` test constructs a separate index for each probability threshold in $\{0.1, 0.3, 0.5, 0.7, 0.9\}$. The same uncertain object set and query set are used for each index. Figure 22 gives the results. Unlike the results for the `Threshold` test, results for the `ThresholdHyper` test are not constant. The MTII and SMTII again have roughly the same results regardless of threshold, but the HMTII's results vary based on the threshold. Disk I/Os and runtime are higher near middle threshold values (around 0.5, or 50%) and lower at extreme threshold values (closer to 0.0 or 1.0, or 0% and 100% respectively). For both uniform and multimodal PDFs, higher threshold values outperform lower threshold values. This suggests that the HMTII is better suited when range queries require a very high degree of



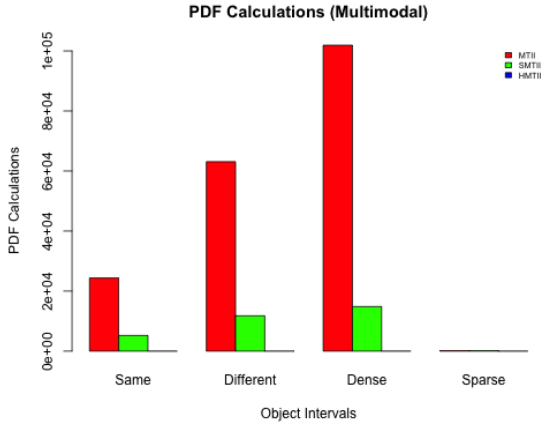
(a) Disk I/Os for Uniform PDFs



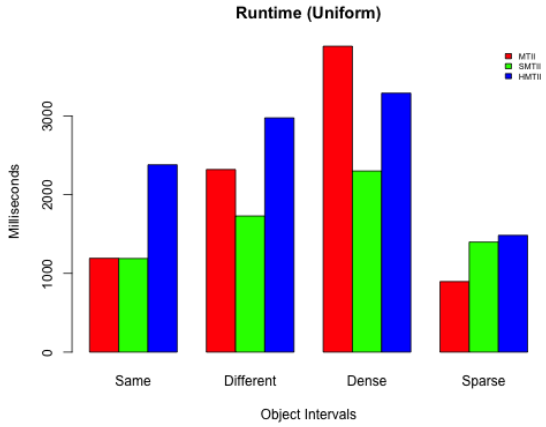
(b) Disk I/Os for Multimodal PDFs



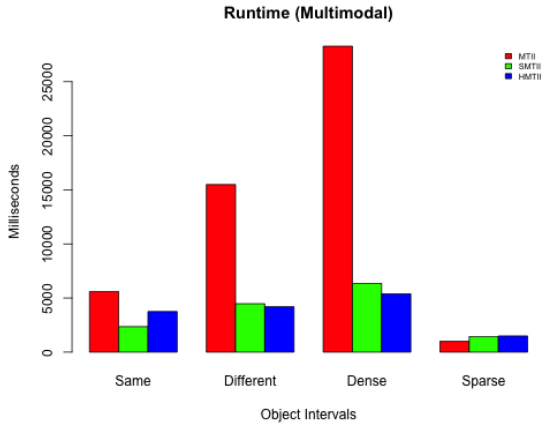
(c) Uniform PDF Calculations



(d) Multimodal PDF Calculations

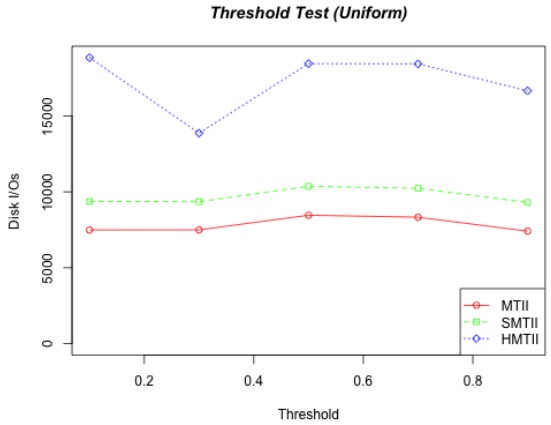


(e) Runtime for Uniform PDFs

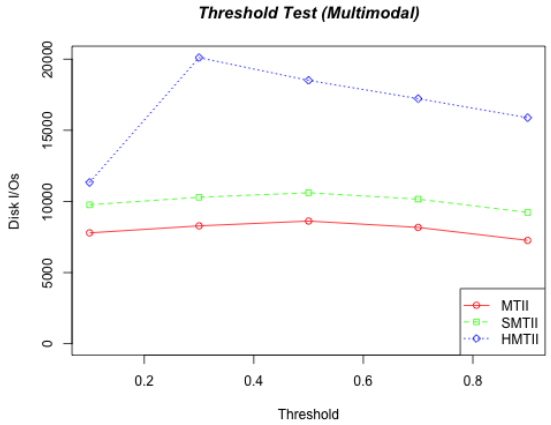


(f) Runtime for Multimodal PDFs

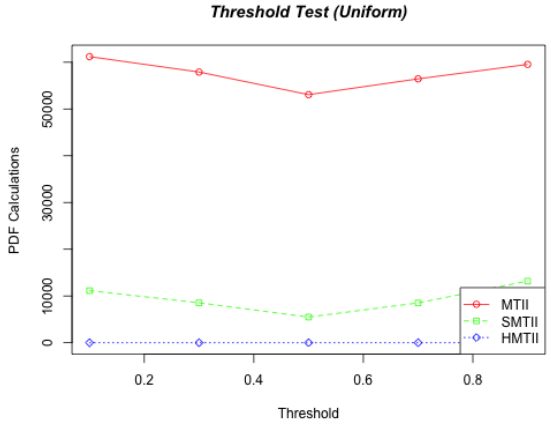
Fig. 21: Performance results for `SameHyper`, `DifferentHyper`, `DenseHyper`, and `SparseHyper` tests.



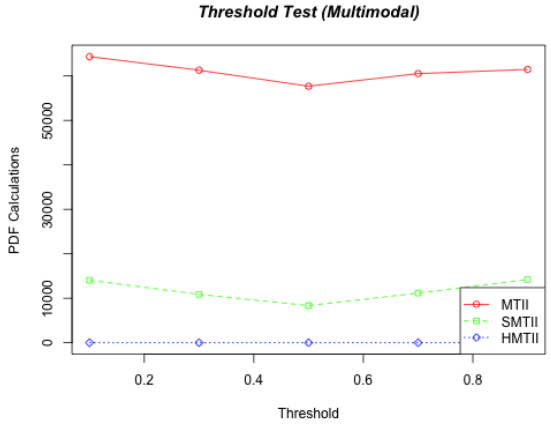
(a) Disk I/Os for Uniform PDFs



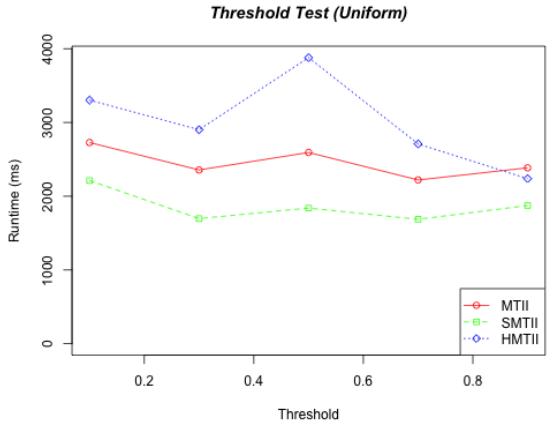
(b) Disk I/Os for Multimodal PDFs



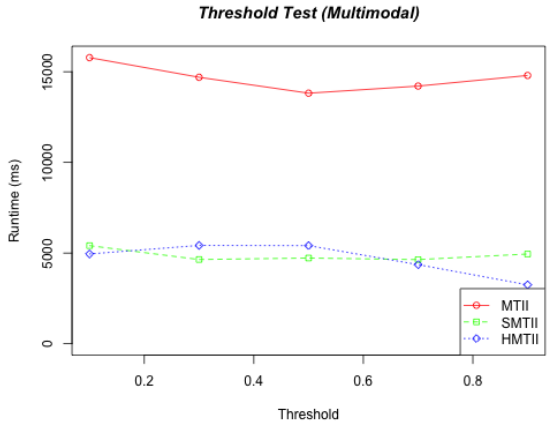
(c) Uniform PDF Calculations



(d) Multimodal PDF Calculations



(e) Runtime for Uniform PDFs



(f) Runtime for Multimodal PDFs

Fig. 22: Performance results for the `ThresholdHyper` test.

TABLE V: Suggested Indexes, in Order of Choice

Type of Complicated Uncertain Data	Index
Sparsely Clustered Objects	MTII
Extremely Complicated PDFs	HMTII
Very Complicated PDFs	SMTII
Densely Clustered Objects	SMTII
Otherwise or Unknown	MTII

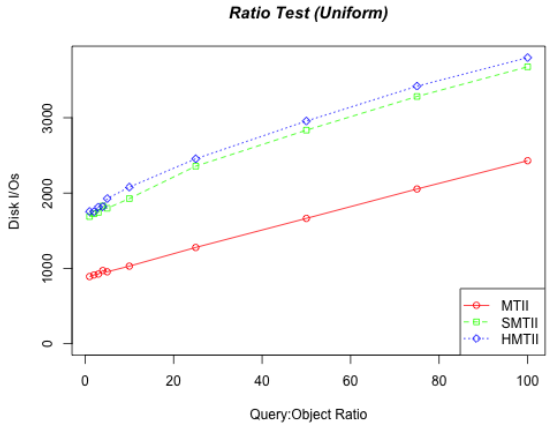
certainty, perhaps for when a probability threshold must be greater than 70%.

3) Effect of Query Interval Size: The `RatioHyper` test investigates the effect of query size relative to the objects' interval sizes. The ratio value system works the same as for the `Ratio` test. Figure 23 gives the results. Again, results are not surprising. Number of disk I/Os increases linearly, and number of probability calculations levels off after a ratio value of about 25. The HMTII has no probability calculations. Most interestingly is the runtime for the HMTII: it appears to increase linearly through all ratio values. There is no staggering decrease after lower ratio values are surpassed. This makes sense because runtime is tied exclusively to disk I/Os, not probability calculations.

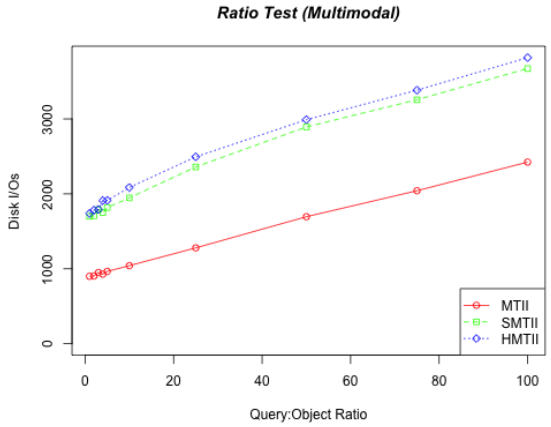
D. Evaluation Summary

Clearly, threshold interval indexing is superior to the PTI for one-dimensional uncertain objects with complicated PDFs. While a perfect PTI far outperforms a practical PTI, it still cannot perform as well as the TII. The TII uses more disk I/Os but far fewer probability calculations than the PTI, so runtime is faster. Furthermore, memory-loaded TIIs consistently outperform non-memory-loaded TIIs by a good margin for all three metrics. Memory loading is the more practical method. TIIs also perform significantly better when query intervals are large in respect to object intervals. The threshold probability used does not have much effect on performance when the thresholds are built into the index.

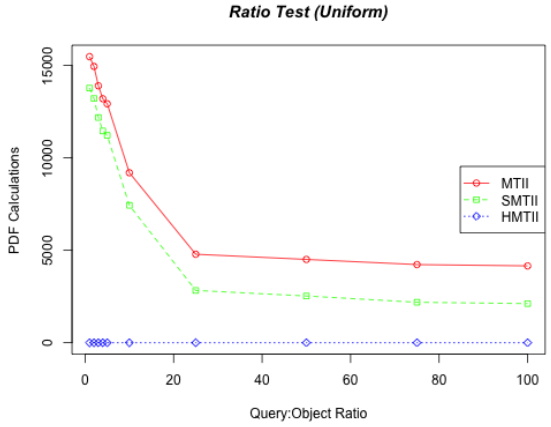
As for the variants on the TII, the strong TII is practical for indexing very complicated objects and densely clustered objects, but not for sparsely clustered objects. For less complicated



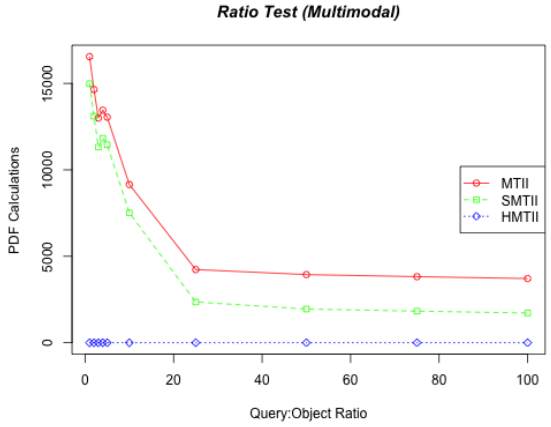
(a) Disk I/Os for Uniform PDFs



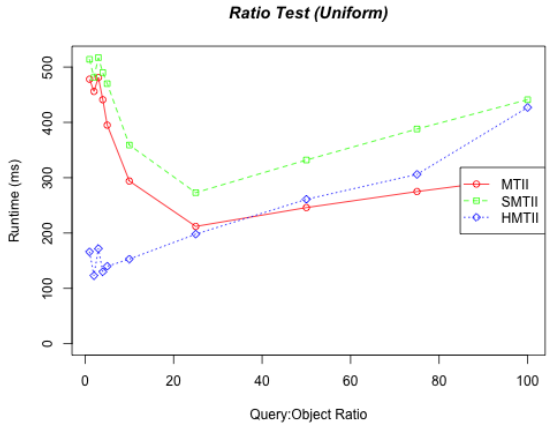
(b) Disk I/Os for Multimodal PDFs



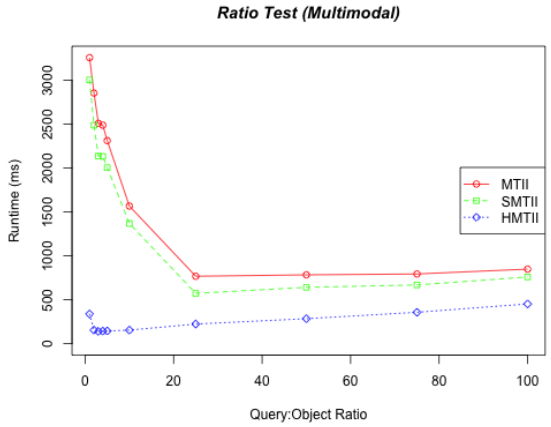
(c) Uniform PDF Calculations



(d) Multimodal PDF Calculations



(e) Runtime for Uniform PDFs



(f) Runtime for Multimodal PDFs

Fig. 23: Performance results for the `RatioHyper` test.

uncertain objects, the MTII would probably be better. Hyper TIIs are most comparable to strong TIIs, but their restrictions make them less worthwhile. Their only advantage is for low query interval to object interval length ratios. Overall, hyper TIIs should probably be disregarded in favor of MTIIs and HMTIIs unless probability calculations are extremely costly. Table V provides suggestions for the choice of uncertain index for given types of uncertain data, based on these experimental results.

VIII. Related Works

This section covers related research in the field of uncertain data. Most of this research has been performed in the past decade. Uncertain data techniques are still actively being developed.

A. Query Evaluation

1) Ranking Queries: A few different types of ranking queries have been proposed for uncertain data, which use the probabilistic database model. U-Top$k$ queries return the list of k tuples with the highest probability of being ranked as the top k [38], [45]. To efficiently perform a U-Top$k$ query, a search of possible states is performed, in which the search is extended only for the tuples of highest probability. Identifying independent tuples also allows for greater search tree pruning. The algorithm presented in [50] optimizes the U-Top$k$ query further from $O(nk)$ time and $O(k^2)$ space to $O(n \log k)$ time and $O(k)$ space.

A U-kRanks query return the tuple with the highest probability of being ranked at each position. This implies that the same tuple could appear in more than one position [38], [45]. When evaluating a U-kRanks query, only the most probable state for each rank so far needs to be stored. Independent tuples also exhibit the optimal substructure property, meaning a dynamic programming solution is possible. A more efficient dynamic programming solution is possible when considering special cases for which ordering of tuples does not matter [50]. Time can be optimized from $O(n^2k)$ to $O(nk)$, and space can be optimized from $O(nk)$ to $O(k)$.

The probabilistic threshold top-k (PT-k) query returns all tuples which have a probability greater than some threshold probability for being ranked in the top k positions [18], [38]. This captures tuples which might be missed by a U-Top$k$ or U-kRanks query. To evaluate, the dominant set property is leveraged, which states that whether or not a tuple is in the result set of the query depends on how many other tuples are ranked higher. Generation rule compressions

and pruning improve query time. Sampling methods and Poisson approximation methods can improve efficiency in exchange for accuracy [18].

2) Joins: A join between tables with uncertain data returns a cross product in which each paired tuple is associated with a probability $p \geq 0$. A *probabilistic join query* (PJQ) between two tables returns all pairs of tuples with a non-zero probability of meeting the join condition. Likewise, a *probabilistic threshold join query* (PTJQ) only returns tuples whose probability is greater than some threshold probability τ. A *confidence-based top-k join query* (PTop$k$JQ) returns the k tuples with the highest probability resulting from the join. It is possible to apply various existing join methods in addition to pruning techniques to optimize query time [12], [26].

Three primary techniques for joining uncertain continuous data are presented in [12]. In item-level pruning, an *interval join* or *overlap join* [16] is performed on the uncertain objects' uncertainty intervals. Simple comparisons to the probability threshold τ can be performed using CDF endpoint values to prune objects which cannot overlap. This saves calculation time. Between the two types of interval join algorithms, block nested loop join (BNLJ) and index nested loop join (INLJ), performance is roughly equivalent throughout different join conditions.

The second technique for continuous data presented in [12] is page-level pruning using uncertainty-based joins. Interval joins fail to utilize uncertainty PDFs in their pruning. In order to leverage uncertainty PDFs in pruning, *x-bounds* can be stored for each page. This requires only initial calculation and minor storage. X-bounds potentially allow for whole pages to be pruned. The page-level pruning algorithm presented in [12] is called *uncertainty-based block nested loop join* (U-BNLJ).

The third technique is very similar in that it uses a PTI for an *uncertainty-based index nested loop join* (U-INLJ). The strategy is the same as for a standard index nested loop join: use the index in a nested loop to more efficiently identify equal pairs. Performance-wise, the U-INLJ is superior, but it requires the most storage space, because it stores the index rather than repeating calculations for each query.

Joins on discrete data operate differently. For the *similarity join query*, [26], [27], each uncertain object is turned into a vector of its possible attributes. These vectors are then clustered into groups using the k-means clustering algorithm, and each group is approximated by calculating its minimum bounding hyper-rectangle.

The form of the similarity join query can be built into the *probabilistic distance range join* [26], [27]. This strategy uses an R-tree indexing structure to organize uncertain objects. Objects at the leaf level are filtered using object approximations and cluster approximations presented for the similarity join query.

3) Skyline Searches: Skyline analysis of a data set searches for the "best" objects by weighing tradeoffs between attributes. The most desirable objects constitute the *skyline*. Performing skyline searches on uncertain data can also be useful [39]. Pei et. al. give the example of data representing NBA players' statistics. Differences in performance from game to game introduce natural uncertainty into the data. Nevertheless, finding the best players is important.

The first challenge with skyline searches on uncertain data is how to model skylines. Objects are compared based on the probability that one object *dominates* another [39]. An object is dominant in regards to another object if all of its attributes are of lower value. Hence, the skyline is the set of all objects which do not dominate other objects. A *p-skyline* is the set of all objects with a probability of p of being in the skyline.

Two algorithms are presented in [39] to perform skyline searches. Both use *bounding-pruning-refining iteration*. Bounding finds the lower and upper bounds on skyline probability for each uncertain object. Pruning either adds an uncertain object to the p-skyline if its lower bound is greater than p or it removes an uncertain object from consideration if its upper bound is less than p. Refining repeats the process for objects left in the middle range. The iteration will eventually converge to assign all objects, because dominance is a total ordering. The bottom-up algorithm starts bounding and refining with subsets of instances at the bottom and works up to the skyline. The top-down algorithm starts the process with the whole set of uncertain objects and recursively partitions it into subsets for improvement. Experimentation showed both algorithms to have similar performance, although top-down performed better on sparser data sets and bottom-up performed better on denser data sets [39].

Reverse skyline searches can also be applied to uncertain data [33]. A reverse skyline obtains a dynamic skyline based on query parameters [14]. In a sense, it can find lowers layers of skylines instead of just the top layer. Reverse skylines are useful, particularly with uncertain data, when verifying faulty equipment or abnormal data [33]. A monochromatic probabilistic reverse skyline queries find reverse skylines over one data set, and bichromatic queries find reverse skylines between two data sets. Pruning can be performed spatially and probabilistically

for both approaches. Offline pre-computation of pruning spaces can optimize queries further.

B. Indexing

1) Categorical Data: Categorical (discrete) data has a finite data domain $D = \{d_1..d_n\}$. Each uncertain attribute within an object is called an *uncertain discrete attribute* (UDA) u. A UDA's value might be any value in the data domain. Therefore, it is represented by a vector of probabilities $u.P = \langle p_1..p_n \rangle$ such that $Pr(u = d_i) = u.p_i$. The probability that two UDAs are equal is given by the dot product of their probability vectors. Effective indexing strategies make equality queries and ranking queries more efficient [40], [44].

Two strategies are proposed by [44]. The first is the *probabilistic inverted index*. It is essentially a list of lists. Each element in the outer list represents a value in the data domain and holds a pointer to an inner list. This inner list holds ids of all UDAs which might be that data value, sorted in decreasing order of probability. Thus, the UDAs most likely to be the data value for the list appear first. Insertions and deletions must preserve the ordering of each inner list. Searching this structure given a threshold probability is relatively easy: entire rows and columns can be pruned from a search for values that have a probability less than the query threshold.

The second strategy proposed by [44] is the *probabilistic distribution R-tree* (PDR-tree). This is very similar to the probability threshold index presented in [11]. All UDAs are stored in an R-tree. Minimum bounding rectangles (MBRs) are placed around the elements in probability vectors. Methods for loading and insertion maintain non-overlapping MBRs as much as possible. Queries can then prune branches whose MBR values are outside of the query range. To achieve space optimization, UDA probabilities can be overestimated to rounded, discretized probability values and thereby stored in fewer bits. Unlike the PTI, however, x-bounds cannot be applied to the MBRs, because the boundaries for each MBR represent discrete probabilities of the UDA, not PDF domains.

Testing shows that there is no universal winner between the two strategies. Both indexes were tested against real and synthetic datasets [44], and the PDR-tree outperformed the probabilistic inverted index. With an increase in dataset size, the PDR-tree continued to outperform. However, the probabilistic inverted index became more efficient than the PDR-tree as the data domain size increased.

2) High Dimensionality: The strategies discussed for continuous and categorical uncertain data focus on indexing one dimension. Sometimes, however, it makes more sense to think of uncertainty in more than one dimension. For example, spacial data, like for GPS or for location-based services [49], has a region of uncertainty, not just values along one line. Multidimensional indexing requires more resources and computation, and there are different approaches.

The U-tree [47] is a natural extension of the probability threshold index [11]. Instead of using an R-tree as the base, the U-tree uses an R*-tree, which is a multidimensional R-tree that optimizes its structure by minimizing area, margin, overlap, and distance of minimum bounding rectangles [9].

Instead of using just MBRs, the U-tree uses *probabilistically constrained regions* (PCRs) to tighten regions, much like x-bounds for the probability threshold index. These PCRs are based on probability threshold values and can be used both to prune and to validate results without further PDF calculation. A *U-calatog* stores predetermined probability thresholds. Unfortunately, with many dimensions, PCRs are impractical for indexing because of their size. Luckily, they can be reduced to two functions called the *outer and inner conservative functional boxes* (CFBs). Experimentation shows that using CFBs is much more efficient than using PCRs.

A leaf in the U-tree contains an uncertain object, its PDF, its CFB, and its MBR. Internal nodes store pointers to child nodes as well as MBRs for its children's CFBs. By this setup, entire subtrees can be pruned from a query search. Queries start at the root and proceed to any nodes which cannot be pruned. The U-tree is also dynamic like the R*-tree, only the updating algorithm uses the summed counterparts of the optimization metrics. Sometimes, nodes must be split when adding new objects.

A major problem with multidimensional data is the *sparsity problem*: distances between pairs of points tend to be too similar to garner quality information using standard distance functions [4]. Aggarwal et. al. propose an expected distance function which uses a contrast ratio between means and standard deviations for a fraction of the uncertain objects [4].

Designing distance functions between objects is linked to indexing objects, too. The *UniGrid index* is a two-level inverted list [4]. All objects' attributes are first partitioned into sets of equi-depth ranges based on mean values. Then, those sets are partitioned again based on uncertainty level. Some metadata must also be stored for partition boundaries. Luckily, storage overhead required for the UniGrid is minimal in regards to the size of the original data set. Similarity

queries search for overlap in object spans between the inverted lists. Range queries identify intersections between the inverted lists and the query interval.

3) Probabilistic Database: The probabilistic database model holds possible worlds for what its data could represent. Indexing such a database is significantly different from indexing data in the uncertain object models. Kanagal et. al. show how correlated probabilistic databases can be represented using *probabilistic graphical models* (PGMs) [22]. These PGMs can be transformed into *junction trees* [20], also called *clique trees*, which have clique nodes of multiple objects and separator nodes of single objects. Queries run along the junction tree to find results. Given probabilities, some cliques can have *shortcut potentials*, meaning a query can skip them.

The INDSEP data structure recursively partitions the junction tree for a probabilistic database into a set of hierarchical subtrees [22]. A linear tree partitioning algorithm [30] is used at each level of partitioning to ensure each partition fits into one data block. This gives the index a top-down view of the data through the junction tree. Variable renaming is performed once the hierarchy is built to sort variables effectively.

The INDSEP data structure allows for query optimizations. Subtrees can be replaced with shortcut potentials to reduce search space.

From initial construction, the tree is balanced. The tree is also dynamic, but adding or removing uncertain objects might make it unbalanced. Whenever a new object is added, cliques and separator nodes must be adjusted.

4) Moving Objects: Indexing moving objects has been well researched [19], [52]. An example of indexing moving objects would be for a city bus route schedule: busses drive along the streets, and they can be tracked to give real-time feedback on their arrival times.

For certain moving data, a robust indexing structure is the B^x-tree [19]. At its core, a B^x-tree is a standard B+-tree. It stores moving objects. A *moving object* has three components: a position (x, y), a velocity (v_x, v_y), and an update time t_i. Objects are first partitioned by intervals of their update times. Next, in order to store it in a one-dimensional fashion (thus using less storage space), a linearization process, such as a Peano curve or Hilbert curve, indexes the position over a spatial domain. This gives the B^x-tree an advantage over the R-tree, which is two-dimensional. Finally, the key for which to store the object into the B+-tree is given by the concatenation of the timestamp and the linearization index as bit strings. Over time, updates arrive for new positions of objects.

A major problem with moving objects is that most of the time, their data is inherently uncertain. Since they are moving, it is harder to track their exact locations. Adding uncertainty to the data model for moving objects allows for more accurate models [52]. Specifically, rather than storing position and velocity as concrete values, position and velocity can be stored as probability distribution functions over spatial domains. To be pragmatic, the two dimensional domains for location and velocity can be discretized into a grid of values, and each cell in the grid can hold a probability for the existence of the value in that cell. Naturally, the sum of probabilities for all cells should be 100%. Two strategies for movement inference, *rectangle inference* and *Monte-Carlo simulation* [52], approximate object positions between time updates and predict future object movements. With this uncertainty model in place, uncertain moving objects can be indexed with a B^x-tree.

A *probabilistic range query* for uncertain moving objects return all objects which fall into a spatial query range R with a minimum threshold probability τ [52]. To answer the query, first a basic spatial range query is performed on the index to find all candidate objects whose locations might be in range. Then, probability verification is performed on each object.

A *top-k probabilistic NN query* (k-PNN) returns the k objects with highest probability of being nearest to a location q at a time t [52]. It is solved by performing a series of range queries for circular spatial regions and analyzing positions of objects in different rings.

Handling uncertainty in the management of moving objects is still a very new topic of research. The above information is only the beginnings of research. The model seems solid, but better techniques will probably be developed for indexing and querying in forthcoming years.

C. Database Systems

Although support for uncertainty features has not yet been implemented by mainstream commercial DBMSs, systems such as Trio [48], MayBMS [24], and Orion [43] have been proposed and implemented by researchers. The systems are similar, but each is built in a unique way.

1) Trio: Trio uses a scheme called the *Uncertainty-Lineage Database*, which is an extension to the standard relational model [5], [10], [48]. Uncertainty is encapsulated in the form of *maybes*, *alternatives*, and *confidences*. A tuple has a "maybe" if it may or may not exist in the table.

Tuple attributes have alternative values for all possibilities, for example, a set of possible colors for a car. Both maybes and alternatives can also be associated with confidence percentages.

Lineage encapsulates how alternatives are derived. For example, suppose Alice and Bob each drive a red car. Charlie's car is hit by a car in the parking lot. The low-resolution security camera can only determine that the car which hit Charlie's car is red. In table T_1, owners and car colors are stored. Say T_1 stores Alice with red, Bob with red, Charlie with green, Dave with blue, and Ed with purple. In table T_2, suspects for hitting Charlie's car are stored. T_2 holds Alice and Bob. Each element in T_2 has a lineage function λ containing references to T_1 for red color alternatives. Lineage can also trace multiple attributes and various alternatives.

Trio has been implemented as a three-layer architecture [5], [48]. At the bottom is a Postgres database. This means all metadata for managing uncertainty and lineage is stored in relational tables. In the middle is a Trio API written in Python to translate ULDB features into a purely relational model. This layer accepts queries written in a language called *TriQL* (pronounced "treacle") [5], [48]. TriQL adds syntactic extensions to SQL to manage extra features for the ULDB. At the top layer are applications such as GUIs and command line interfaces.

Trio is a good prototype for a probabilistic database. The middle layer is fairly portable to any relational database, not just Postgres. Lineage may or may not be important for managing uncertain data, but its availability as a feature is nice. TriQL is very much like SQL and therefore easy to learn. It also includes SQL-like queries for aggregation and ordering. The biggest weakness Trio has is that it can only handle uncertain categorical data, not uncertain continuous data. Furthermore, indexing and query optimization are not discussed in [5], [10], [48].

2) MayBMS: MayBMS is an open-source project built as an extension to Postgres [24]. Using the probabilistic database model, it relies heavily upon its own *probabilistic world-set algebra* [24], which adds both operations for computing tuple confidence and repair-keys for introducing uncertainty to relational algebra. This algebra is probably the most sophisticated of any presently developed for uncertain data. Data is represented by a purely relational system called *U-relational databases*. MayBMS also adds syntax to SQL to access its features.

The creators of MayBMS wish for the system to support continuous distributions [24]. They also wish to develop a standardized query language for uncertain data [24]. However, MayBMS suffers from the same weakness as Trio in that it cannot handle uncertain continuous data in the

uncertain object model.

3) Orion: Orion 2.0 is the first database system to support both continuous and categorical uncertain data simultaneously and natively [43]. It is the successor to U-DBMS and built upon PostgreSQL, written mostly in C. Whereas Trio is written as an extra two layers on top of a relational database, Orion incorporates support for uncertain data from within the database engine. Storing uncertain categorical data is similar to methods described for Trio and MayBMS, but Orion includes a set of pre-defined probability distributions for uncertain continuous data. Furthermore, users can specify their own histogram PDFs. In addition, Orion features lineage (called *history* [43]), indexing, join support, missing data support, and query optimization.

D. Clustering

Data mining, much like data management, is a rich research field for uncertain data. Presently, most research efforts have focused on clustering techniques [3]. Many existing clustering algorithms can easily be modified to handle uncertain data. As for data management, the state-of-the-art is still actively developing. Three major techniques for clustering include UK-means, a density-based method, and an evolutionary data stream method.

1) UK-means: One of the oldest and most standard algorithms for clustering data is the k-means algorithm [34]. Initially, k centroids are chosen at random. All data points are then assigned to the closest centroid based on a distance function. The centroids are then recalculated based on the points assigned to each cluster. Points are then reassigned again to their closest clusters. This process repeats until the centroids converge; that is, until the centroid points only have minimal changes.

The UK-means algorithm extends the k-means algorithm for use with uncertain data [3], [36]. Since exact data point locations are unknown, the UK-means algorithm uses the expected distance function when assigning data points to clusters. Repeatedly calculating the expected distance is computationally expensive, however. Some calculations can be pruned by using *min-max-dist pruning*. The minimum bounding rectangle is calculated for each object. When an object is being assigned to a cluster, the minimum and maximum distances from each centroid to the MBR is calculated, which takes very simple mathematics. Then, the smallest maximum distance is determined. Any centroids whose minimum distance is greater than the smallest maximum

distance can be pruned from the search, since that cluster would be guaranteed to be farther from the object than other clusters. Thus, many expected distance calculations can be avoided.

An alternative to the UK-means algorithm, called CK-means, optimizes expected distance function calculations instead of pruning centroids [3], [32]. The novelty noted in [32] is that expected distance calculations are very similar to moment of inertia calculations from the study of mechanics. Just like the parallel axis theorem allows for moment of inertia calculations to be calculated from any reference point, a single reference point can be calculated for expected distance. Then, the expected distance between any object o and an axis y is equal to the expected distance from a centroid to o plus the square of the distance from y to k. Furthermore, the full calculation of the expected distance between a centroid and an object becomes unnecessary, and the problem reduces a k-means algorithm with a new distance function. This reduction is CK-means.

2) Density-Based: Density-based clustering on uncertain data is based on the DBSCAN algorithm for certain data [3], [17]. In the DBSCAN algorithm, points are clustered together based on their proximity to other points. A parameter is given for the minimum number of points a cluster must hold. Two points are *directly density-reachable* if they are within a given distance and surrounded by the minimum number of points required to form a cluster. Two points are *density-reachable* if there exists a chain of points between them for which each link in the chain is directly-density reachable. Clusters form around points which are *density-connected*, meaning all points in the cluster are density-reachable. Thus, clusters form around points which are densely congregated, not necessarily around points which are close together.

In order to adapt DBSCAN for use with uncertain data, objects are treated as fuzzy objects [28]. Fuzzy objects have a spatial domain where the object can exist. A standard distance between two fuzzy objects cannot be calculated, so instead, an expected distance function (also called a fuzzy distance function) is used. The FDBSCAN algorithm adopts this expected distance function [3], [28]. Furthermore, every two objects now have a *reachability probability*. If the reachability probability is less than 50%, then the two points are disregarded. Experimentation proved how the FDBSCAN was effective and efficient in clustering uncertain data [28].

Hierarchical density-based clustering can be performed on uncertain data with the FOPTICS algorithm, which extends OPTICS much like FDBSCAN extends DBSCAN [3], [29]. FOPTICS is similar to FDBSCAN except that the density parameters can be adjusted for each level of

clustering. This gives greater insight into separating smaller clusters from larger clusters.

3) Micro-Clustering: UMicro is an evolutionary clustering algorithm for uncertain data [3]. The difference in UMicro from standard micro-clustering is that UMicro uses error-based micro-clusters. Data points arrive from a data stream at different times. Each data point includes a vector of values and a vector of error values. Calculating expected similarity between data points uses error values to account for uncertainty in cluster membership.

IX. Conclusion

Threshold interval indexing is a new strategy for indexing complicated uncertain continuous data of one dimension. Three structures are presented. The threshold interval index applies x-bounds to the nodes of an interval tree to improve pruning. The strong threshold interval index applies x-bounds to each object to further increase pruning and also to enable some results to be accepted without probability calculations. The hyper threshold interval index stores threshold satisfier intervals to accept results without any probability calculations at all, but it is limited in functionality. Methods for updating, bulk loading, and externalizing are provided for each structure. Memory loading strategies are also described for practical efficiency. The key advantage of threshold interval indexing over existing strategies, such as the probability threshold index, is that it handles complicated PDFs much more efficiently because it handles object layout better with its intervals. Experimental results prove this assertion.

The field of uncertain data has many more topics to explore. Specifically, future research should explore 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. Testing should mix PDF types since most experiments test indexes with homogeneous PDF types. Perhaps different coordinate systems for probability functions might be useful, such as polar coordinates for spatial data.

For threshold interval indexing, the experiments performed for this paper only tested range queries. Testing update times for adding and deleting uncertain objects would yield valuable information. Furthermore, query performance should be tested after performing several updates to ensure balance is maintained and efficiency is preserved. Changes to the TII's structure should also be tested. The primary tree parameters k and r could be altered to prove or disprove optimal values for each. Secondary structures could be subdivided further into slab lists like the dynamic

external interval tree [7]. A quicker bulk loading strategy might be possible for the MTII, since it discards most of its objects' endpoints.

Topics from related works could also be incorporated into indexing uncertain data. Perhaps certain rectangle management strategies could improve PTI performance, such as the priority R-tree [6] or the segment R-tree [25]. Although this paper focuses on range queries, threshold interval indexes could be used for joins [12], [16]. Database management systems should incorporate more uncertain data strategies [5], [43]. Parallelization should be explored further.

REFERENCES

[1] P. K. Agarwal, S.-W. Cheng, Y. Tao, and K. Yi. Indexing uncertain data. In *Symposium on Principles of Database Systems (PODS)*, pages 137–146, 2009.

[2] C. C. Aggarwal. *Managing and Mining Uncertain Data*, chapter An Introduction to Uncertain Data Algorithms and Applications, pages 2–8. Springer, 2009.

[3] C. C. Aggarwal. *Managing and Mining Uncertain Data*, chapter On Clustering Algorithms for Uncertain Data, pages 389–406. Springer, 2009.

[4] C. C. Aggarwal and P. S. Yu. On indexing high dimensional data with uncertainty. In *IEEE International Conference on Data Engineering (ICDE)*, 2008.

[5] P. Agrawal, O. Benjelloun, A. D. Sarma, C. Hayworth, S. Nabar, T. Sugihara, and J. Widom. Trio: A system for data, uncertainty, and lineage. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 1151–1154, 2006. Demonstration Description.

[6] L. Arge, M. de Berg, H. J. Haverkort, and K. Yi. The priority r-tree: A practically efficient and worst-case optimal r-tree. In *SIGMOD Conference*, pages 347–358, 2004.

[7] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *Proc. IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 560–569, 1996. Extended Abstract.

[8] L. Arge and J. S. Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 32(6):1488–1508, 2003.

[9] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD Conference*, pages 322–331, 1990.

[10] O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom. Uldbs: Databases with uncertainty and lineage. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 953–964, 2006.

[11] R. Cheng, Y. Xia, S. Prabhakar, R. Shah, and J. S. Vitter. Efficient indexing methods for probabilistic threshold queries over uncertain data. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 876–887, 2004.

[12] R. Cheng, Y. Xia, S. Prabhakar, R. Shah, and J. S. Vitter. Efficient join processing over uncertain data. In *Proc. of the 15th ACM International Conference on Information and Knowledge Management*, pages 738–747, 2006.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. Massachusetts Institute of Technology, 2nd edition, 2001.

[14] E. Dellis and B. Seeger. Efficient computation of reverse skyline queries. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 291–302, 2007.

[15] A. Deshpande, L. Getoor, and P. Sen. *Managing and Mining Uncertain Data*, chapter Graphical Models for Uncertain Data, pages 77–112. Springer, 2009.

[16] J. Enderle, M. Hampel, and T. Seidl. Joining interval data in relational databases. In *SIGMOD Conference*, pages 683–694, 2004.

[17] M. Ester, H. peter Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD Conference*, pages 226–231, 1996.

[18] M. Hua, J. Pei, W. Zhang, and X. Lin. Ranking queries on uncertain data: A probabilistic threshold approach. In *SIGMOD Conference*, pages 673–686, 2008.

[19] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient b+-tree based indexing of moving objects. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 768–779, 2004.

[20] F. V. Jensen and F. Jensen. Optimal junction trees. In *UAI*, pages 360–366. Morgan Kaufmann, 1994.

[21] M. H. Kalos and P. A. Whitlock. *Monte Carlo methods. Vol. 1: basics*. Wiley-Interscience, New York, NY, USA, 1986.

[22] B. Kanagal and A. Deshpande. Indexing correlated probabilistic databases. In *SIGMOD Conference*, 2009.

[23] P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter. Indexing for data models with constraints and classes. In *Symposium on Principles of Database Systems (PODS)*, 1993.

[24] C. Koch. *Managing and Mining Uncertain Data*, chapter MayBMS: A System for Managing Large Uncertain and Probabilistic Databases, pages 149–183. Springer, 2009.

[25] C. P. Kolovson and M. Stonebraker. Segment indexes: dynamic indexing techniques for multi-dimensional interval data. In *SIGMOD Conference*, pages 138–147, 1991.

[26] H.-P. Kriegel, T. Bernecker, M. Renz, and A. Zuefle. *Managing and Mining Uncertain Data*, chapter Probabilistic Join Queries in Uncertain Databases, pages 257–298. Springer, 2009.

[27] H.-P. Kriegel, P. Kunath, M. Pfeifle, and M. Renz. Probabilistic similarity join on uncertain data. In *Proc. 11th International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 295–309, 2006.

[28] H.-P. Kriegel and M. Pfeifle. Density-based clustering of uncertain data. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 672–677, 2005.

[29] H.-P. Kriegel and M. Pfeifle. Hierarchical density based clustering of uncertain data. In *Proceedings of the Fifth IEEE International Conference on Data Mining (ICDE)*, pages 689–692, 2005.

[30] S. Kundu and J. Misra. A linear tree partitioning algorithm. In *SIAM J. Comput.*, pages 151–154, 1977.

[31] Y. Lan and G. A. Papoian. Evolution of complex probability distributions in enzyme cascades. *Journal of Theoretical Biology*, 248:537–545, 2007.

[32] S. D. Lee, B. Kao, and R. Cheng. Reducing uk-means to k-means. In *Proceedings of the Seventh IEEE International Conference on Data Mining Workshops*, pages 483–488, 2007.

[33] X. Lian and L. Chen. Monochromatic and bichromatic reverse skyline search over uncertain databases. In *SIGMOD Conference*, 2008.

[34] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297, 1967.

[35] Y. Manolopoulos, Y. Theodoridis, and V. J. Tsotras. *Advanced Database Indexing*, chapter 4: Access Methods for Intervals. Kluwer, 2000.

[36] W. K. Ngai, B. Kao, C. K. Chui, R. Cheng, M. Chau, and K. Y. Yip. Efficient clustering of uncertain data. In *Proceedings of the Sixth International Conference on Data Mining (ICDM)*, pages 436–445, 2006.

[37] M. H. Overmars. *The Design of Dynamic Data Structures*, chapter Lecture Notes in Computer Science 156. Springer-Verlag, 1983.

[38] J. Pei, M. Hua, Y. Tao, and X. Lin. Query answering techniques on uncertain and probabilistic data. In *SIGMOD Conference*, pages 1357–1364, 2008. Tutorial Summary.

[39] J. Pei, B. Jiang, X. Lin, and Y. Yuan. Probabilistic skylines on uncertain data. In *Proc. International Conference on Very Large Data Bases (VLDB)*, 2007.

[40] S. Prabhakar, R. Shah, and S. Singh. *Managing and Mining Uncertain Data*, chapter Indexing Uncertain Data, pages 299–325. Springer, 2009.

[41] R. Ramakrishnan and J. Gehrke. *Database Management Systems*, chapter Tree-Structured Indexing, pages 338–369. McGraw Hill, 3rd edition, 2003.

[42] S. Sarkar and D. Dey. *Managing and Mining Uncertain Data*, chapter Relational Models and Algebra for Uncertain Data, pages 45–76. Springer, 2009.

[43] S. Singh, C. Mayfield, S. Mittal, S. Prabhakar, S. Hambrusch, and R. Shah. Orion 2.0: Native support for uncertain data. In *SIGMOD Conference*, pages 1239–1242, 2008. Demonstration Description.

[44] S. Singh, C. Mayfield, S. Prabhakar, R. Shah, and S. Hambrusch. Indexing uncertain categorical data. In *IEEE International Conference on Data Engineering (ICDE)*, pages 616–625, 2007.

[45] M. A. Soliman and I. F. I. an Kevin Chen-Chuan Chang. Top-k query processing in uncertain databases. In *IEEE International Conference on Data Engineering (ICDE)*, 2007.

[46] J. Stewart. *Calculus: Early Transcendentals*. Brooks Cole, 6th edition, 2007.

[47] Y. Tao, R. Cheng, X. Xiao, W. K. Ngai, B. Kao, and S. Prabhakar. Indexing multi-dimensional uncertain data with arbitrary probability density functions. In *Proc. International Conference on Very Large Data Bases (VLDB)*, 2005.

[48] J. Widom. *Managing and Mining Uncertain Data*, chapter Trio: A System for Data, Uncertainty, and Lineage, pages 113–148. Springer, 2009.

[49] O. Wolfson, A. P. Sistla, S. Chamberlain, and Y. Yesha. Updating and querying databases that track mobile units. In *Distributed and Parallel Databases*, 1999.

[50] K. Yi, F. Li, G. Kollios, and D. Srivastava. Efficient processing of top-k queries in uncertain databases. In *IEEE International Conference on Data Engineering (ICDE)*, 2008.

[51] J. Yu, M.-S. Yang, and P. Hao. A novel multimodal probability model for cluster analysis. In *RSKT '09: Proceedings of the 4th International Conference on Rough Sets and Knowledge Technology*, pages 397–404, Berlin, Heidelberg, 2009. Springer-Verlag.

[52] M. Zhang, S. Chen, C. S. Jensen, B. C. Ooi, and Z. Zhang. Effectively indexing uncertain moving objects for predictive queries. In *Proc. International Conference on Very Large Data Bases (VLDB)*, 2009.