NIR-Tree: A Non-Intersecting R-Tree

Kyle Langendoen, Brad Glasbergen, Khuzaima Daudjee
{kjlangen,bjglasbe,kdaudjee}@uwaterloo.ca
Cheriton School of Computer Science
University of Waterloo

ABSTRACT

Indexes for multidimensional data based on the R-Tree are popularly used by databases for a wide range of applications. Such index trees support point and range queries but are costly to construct over datasets of millions of points. We present the Non-Intersecting R-Tree (NIR-Tree), a novel insert-efficient, in-memory, multidimensional index that uses bounding polygons to provide efficient point and range query performance while indexing data at least an order of magnitude faster. The NIR-Tree leverages non-intersecting bounding polygons to reduce the number of nodes accessed during queries, compared to existing R-family indexes. Our experiments demonstrate that inserting into a NIR-Tree is 27× faster than the ubiquitous R*-Tree, with point queries completing 2× faster and range queries executing just as quickly.

CCS CONCEPTS

• **Information systems** → **Multidimensional range search**.

KEYWORDS

Multidimensional indexing, Tree, Bounding polygon

ACM Reference Format:
Kyle Langendoen, Brad Glasbergen, Khuzaima Daudjee. 2021. NIR-Tree: A Non-Intersecting R-Tree. In *33rd International Conference on Scientific and Statistical Database Management (SSDBM 2021), July 6–7, 2021, Tampa, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468791.3468818

1 INTRODUCTION

Multidimensional indexes are an important part of modern databases [10, 12, 20]. Correlated data, such as points in space or the RGB values of a pixel, can be stored and retrieved together in multidimensional indexes. These indexes need to support efficient querying and retrieval of data for many popular application domains, such as infectious disease tracking, continental road networks, video game states, and scientific simulations of entire galaxies. The volume and variety of such multidimensional data demand indexes that can deliver high performance through low latency querying of data.

Solutions to indexing multidimensional data [3, 14, 22] are based largely on the conceptual structure that originates from the R-Tree [11]. R-Trees recursively group multidimensional data into *bounding rectangles* that represent an approximation of the data group's local region of space. When executing search queries, bounding rectangles are consulted to direct the search into continually smaller, more specific rectangular regions that meet the search criteria. When data groups are poorly represented by large or intersecting bounding rectangles, search is slowed by accessing regions whose data does not meet the search criteria.

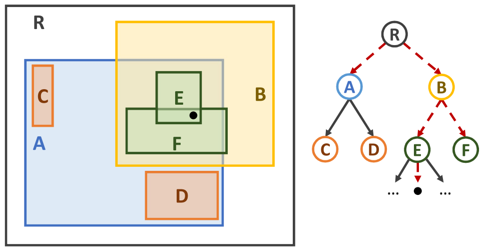

Figure 1: An R-Tree exhibiting scatter during search.

To illustrate, consider Figure 1 depicting an R-Tree with nodes and associated bounding rectangles shown in the same colour. R-Trees require parent bounding rectangles to enclose their children's bounding rectangles, and so R's black bounding rectangle encloses children A and B with blue and yellow bounding rectangles respectively. The search for the black point in Figure 1, indicated by the dashed red line, is slowed by *scattering* into logical nodes A and F. This search spuriously accesses A and F which do *not* contain the desired point because their associated bounding rectangles undesirably enclose the point. Bounding rectangle pairs A, B and E, F create scatter and thus slow search with their *intersection*.

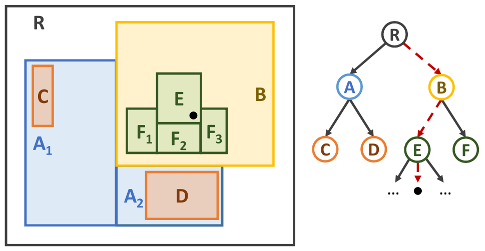

Figure 2: NIR-Tree exhibiting no scatter during search.

A desirable, efficient multidimensional index would support search by minimizing intersection between bounding rectangles. As a running example, consider the same point from Figure 1 indexed by the reconfigured geometry in Figure 2. This improved R-Tree design, which we call the **NIR-Tree**, removes the intersection area between *bounding polygon* pairs A, B and E, F. By doing so, search for the same black data point no longer scatters to nodes A and F. This 40% reduction in accessed nodes, again highlighted by a dashed red path in Figure 2, translates into a faster search.

Prior proposals improved ways to organize bounding rectangles [3, 14, 22] while other approaches [13, 15] abandoned rectangles entirely in favour of more complex geometric objects. Although these approaches reduce intersection area they, unlike our proposed NIR-Tree, cannot eliminate it entirely. Proposals which use arbitrarily shaped bounding objects [13, 15] suffer from slow, complicated geometric tests to determine if a point is enclosed by a bounding object.

In this paper, we present the NIR-Tree, a new, entirely in-memory tree that adaptively replaces bounding rectangles with bounding polygons. The NIR-Tree *guarantees* zero-area intersection between bounding polygons which accelerates point queries by reducing the number of nodes accessed compared to existing R-family indexes. Our experiments demonstrate that inserting into a NIR-Tree is an order of magnitude faster than the ubiquitous R*-Tree, with point queries completing in half the time and range queries executing just as quickly.

Concretely, our contributions to multidimensional indexing are:

(1) The design and implementation of the NIR-Tree, a novel in-memory technique to create axis-aligned bounding polygons from bounding rectangles.
(2) A zero-area intersection guarantee among bounding polygons in the NIR-Tree, including an analysis and proof of this guarantee.
(3) Extensive evaluation on both real and synthetic datasets, demonstrating that the NIR-Tree is 27.8× faster to construct, 2.2× faster to point query, and up to 8% faster to range query than the R*-Tree.

Section 2 discusses how bounding polygons can eliminate intersection area, Section 3 presents the design of the NIR-Tree, Section 4 analyzes the zero-area intersection guarantee, and Section 5 provides an experimental evaluation of the NIR-Tree compared with prior approaches. Related work is covered in Section 6, and Section 7 concludes our work. Without loss of generality, we discuss the two dimensional versions of trees throughout the paper (with the exception of Section 4) to simplify the presentation.

2 BACKGROUND

Scatter, which degrades search performance in an R-Tree, is caused by positive intersection area. Insertions induce intersections when bounding rectangles are *expanded* to enclose a new point.

When expansion is required, a bounding rectangle that will contain the new point is *selected* for expansion based on a cost function called a *metric*. Bounding rectangles within the R-Tree are selected to minimize the amount of additional area required to enclose a new point. Figure 3a depicts the R-Tree's metric in action. A new black point must be enclosed either by the blue or the yellow rectangle. Since the yellow rectangle requires less additional area than the blue rectangle to enclose the new point the yellow rectangle is selected and expanded.

Bounding rectangle intersection is not limited to the R-Tree's metric. For example, a simple alternative metric that selects bounding rectangles based on minimum distance to the new point is depicted in Figure 3b, yet intersection may still occur. For another example, the R*-Tree [3] considers perimeter of the expanded bounding rectangle (Figure 3a) and additional intersection area (Figure 3c). However, all of these alternatives may cause intersection. The yellow rectangle in Figure 3 always minimizes the given metric in each case, yet its selection and expansion invariably causes positive intersection area. Note that this happens *even when the metric optimizes for intersection area directly.*

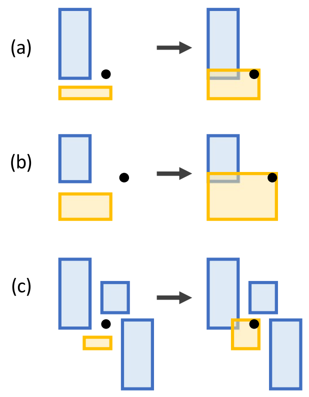

Figure 3: Expansion causing intersection in three examples.

In contrast with existing R-family trees, when the NIR-Tree's bounding rectangles cause intersection, they are replaced with bounding polygons that do not intersect. NIR-Tree bounding polygons are sets of rectangles that, when treated as a logical whole, form an axis-aligned polygon like the collection of blue $\{A_1, A_2\}$ and green $\{F_1, F_2, F_3\}$ rectangles illustrated in Figure 2. By forming bounding polygons from bounding rectangles during `insert`, the NIR-Tree achieves zero-area intersections between bounding polygons (Figure 2) where the R-Tree fails to do so.

3 THE NIR-TREE

In this section, we outline the logical structure of the NIR-Tree and then describe in detail how the NIR-Tree creates, expands, and splits bounding polygons (and associated nodes) during `insert`. Afterwards, the deletion operation `remove` is detailed, and the section concludes with an explanation of point and range `search` operations.

3.1 Structure and Data Layout

The NIR-Tree is structured as a tree of nodes, each associated with a bounding polygon. Nodes may be one of two types: routing or leaf. Routing nodes contain a set of branches, where branches are a pointer to a child node $child_i$ and a bounding polygon $\mathcal{P}_i$ representing the geometric region of that child.

Definition 1. *A **routing node** is a tuple*

$$\langle parent, \{\langle child_0, \mathcal{P}_0\rangle, \ldots, \langle child_n, \mathcal{P}_n\rangle\}\rangle$$

Leaf nodes contain a set of points, which are optionally associated with some value. Both types of nodes contain a pointer to their parent to enable upwards tree traversal.

Definition 2. *A **leaf node** is a tuple* $\langle parent, \{p_0, \ldots, p_n\}\rangle$

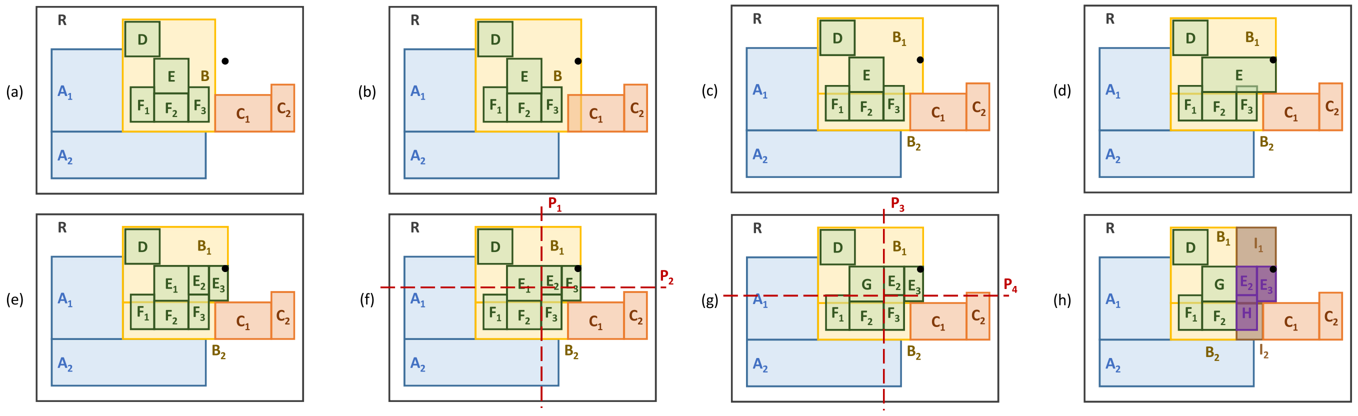

Figure 4: Insertion is shown with polygon expansion, fragmentation, refinement, and splitting. Maximum Fanout = 3.

As in other multidimensional indexes, nodes must contain no more branches or data points than some maximum fanout m assigned at tree construction time. Nodes that contain more than m branches or data points are said to *overflow* and must be split. In contrast to nodes, bounding polygons may be composed of as many —or as few— rectangles as desired.

For notational convenience, we will use $N.poly$ to denote the bounding polygon associated with node N. Concretely, if N is a child of parent P, denoted $N.parent$, then $N.poly$ is the bounding polygon associated with N's branch in P.

3.2 Updating

Updates define the geometric structure of the NIR-Tree and greatly influence the performance of other operations. The primary update operation `insert` is carried out in two stages: (i) a downward root-to-leaf sweep expanding bounding polygons along the insertion path, carried out by `chooseLeaf` (Algorithm 1), and then (ii) an upward leaf-to-root sweep along the same path, splitting nodes whenever they overflow, carried out by `adjustTree` (Algorithm 2). We will use Figure 4 as a running example for the steps executed during `insert`, described next.

3.2.1 Insert Downward Sweep. Insertion starts with the NIR-Tree illustrated in Figure 4a. Note that all bounding polygons have zero-area intersection and every bounding polygon is completely enclosed by its parent's bounding polygon. During the downward sweep, `chooseLeaf` (Algorithm 1) executes the following process at every level, starting at the root. First, the stopping condition must be checked (line 2). If the current node is a leaf, then execution stops because a leaf has been successfully chosen. Otherwise, `chooseLeaf` uses the NIR-Tree's area minimization metric to select bounding polygons for expansion on each level (line 6). Bounding polygons within the NIR-Tree are selected to minimize the amount of additional area required to enclose a new point (Figure 3b). Every rectangle within each bounding polygon is evaluated using this metric, and the bounding polygon with the constituent rectangle that needs the least additional area is selected. Our metric minimizes additional area because future steps force intersection area to be zero.

Algorithm 1 chooseLeaf(N, p) $\longrightarrow L$

```
Require: $N$ is the root, $p$ a new point, $L$ is a leaf
1:  $L = N$
2:  while $L$ is not a leaf do
3:    if $\exists B \in L.branches$ such that $p \in B.\mathcal{P}$ then
4:      $L = B.child$
5:    else
6:      Let $B$ be the branch of $L$ for which $B.\mathcal{P}$ requires least
        additional area to enclose $p$.
7:      Expand $B.\mathcal{P}$ to contain $p$
8:      for $\forall B' \in L.branches$ where $B'.\mathcal{P}$ not disjoint from $B.\mathcal{P}$
        do
9:        $B.\mathcal{P}$ = fragment($B.\mathcal{P}, B'.\mathcal{P}$)
10:     end for
11:     $B.\mathcal{P} = B.\mathcal{P} \cap L.\mathcal{P}$
12:     refine($B.\mathcal{P}$)
13:     $L = B.child$
14:   end if
15: end while
16: return $L$
```

In the best case, the new point already lies within an existing bounding polygon and no expansion is necessary (line 3). Unfortunately, the new point in Figure 4a is not enclosed by any existing bounding polygon. Instead, bounding polygon B is selected for expansion because it requires the least additional area to enclose the new point. Specifically, the rectangle within bounding polygon B requiring the least additional area is expanded to enclose the new point. Since B contains only one rectangle, it is trivially selected for expansion. Now, as concretely observed between bounding polygons B and C in Figure 4b, expansion may cause sibling bounding polygons to have positive intersection area.

If expanding a bounding polygon (line 7) causes intersection area between sibling bounding polygons to become positive, then the NIR-Tree replaces the selected rectangle within the selected bounding polygon with a set of new rectangles that do not intersect its sibling bounding polygons (line 9). We call this process *fragmentation* because the offending rectangle is replaced with fragments of

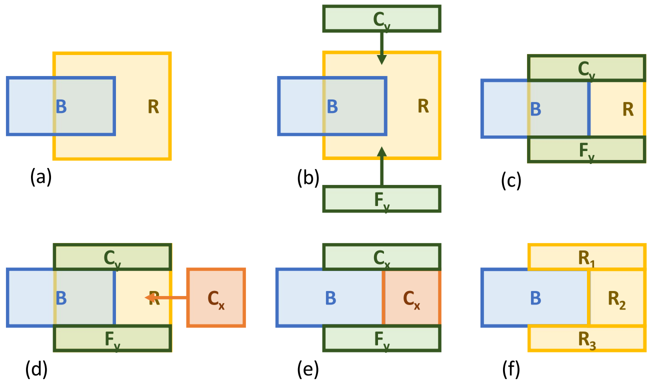

Figure 5: Polygon fragmentation in two dimensions.

itself. We start by discussing fragmentation generally using Figure 5 before applying the process to our running example in Figure 4c.

Since the fragmentation process must generalize to any number of dimensions, one dimension is processed at a time. In Figure 5a, bounding polygon $\mathcal{P} = \{R\}$ has expanded and intersects bounding polygon $\mathcal{P}' = \{B\}$. To aid our analogy, the y dimension is considered first, where a ceiling C_y and floor F_y are created by bounding copies of R with the "top" and "bottom" of B in y (Figures 5b and 5c). The process is then repeated for the x dimension where a ceiling C_x is created; bounded now not only by the "top" and "bottom" of B in x, but also all previously created ceilings and floors. The fully bounded ceiling is depicted in orange in Figures 5d and 5e. Notice that a floor F_x is not created for the x dimension because the "bottom" of B in x lies outside of R. After all ceilings and floors in all dimensions have been computed this way, $\mathcal{P}$ is set to be these resulting fragments. That is, $\mathcal{P} = \{R_1, R_2, R_3\}$ as depicted in Figure 5f. Each rectangle is fragmented into at most $2 \times d$ pieces in $\mathbb{R}^d$. If the bounding polygon $\mathcal{P}'$ had been a set of multiple rectangles $\{B_1, B_2, B_3\}$ instead of a set consisting of a single rectangle $\{B\}$, then the process just described would be executed for each of B_1, B_2, B_3 with any fragments of R created during previous iterations.

Applying the above process to our running example, the single rectangle within B is replaced with two rectangles B_1, B_2 (Figure 4c), achieving zero-area intersection with bounding polygon C. To maintain a valid NIR-Tree, we prune away any area of the produced fragments outside the selected node's parent bounding polygon (line 11). Since no area of B_1 or B_2 in Figure 4c is outside of the area of R, B_1 and B_2 remain the same.

If as a result of fragmentation or expansion B's geometric region could be enclosed using fewer rectangles, then `refine` (line 12, Figure 6) will reduce the number of rectangles in B if B matches one or more of the following patterns. First, when a rectangle is also a line and lies on the perimeter of another rectangle (Figure 6a), the line rectangle is removed. Second, when a rectangle is enclosed by another rectangle (Figure 6b), the enclosed rectangle is removed. Finally, when rectangles are organized into a column or row of constant width or height and have positive intersection area (Figure 6c), the row or column is replaced with a single rectangle. In Figure 4c, B_1 and B_2 do not exhibit any of the three patterns so B is left unaltered.

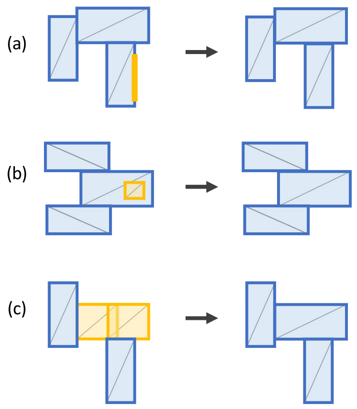

Figure 6: Refinement reducing polygon size in rectangles.

After the execution of rectangle fragmentation and refinement are complete for the current level, insertion moves down the tree (line 13) into the selected child node (B in our running example). Selection, expansion, fragmentation, refinement, and descent are repeated until there exist no more children to be selected. As a result, we see in Figure 4 that E is expanded to enclose the new point, and then E is fragmented and refined into E_1, E_2, E_3 in Figure 4e so as to eliminate intersection with F. Since E is a leaf, we place the new point in E, and we pass E to the second stage of insert, `adjustTree`.

Algorithm 2 adjustTree(N)

```
Require: $N$ is a leaf, $\exists m$ a maximum fanout
1: while $N$ not root do
2:     if $|N.branches| > m$ or $|N.data| > m$ then
3:         $\langle N_L, \mathcal{P}_L \rangle, \langle N_R, \mathcal{P}_R \rangle$ = splitNode($N$, partitionNode($N$))
4:         Add $\langle N_L, \mathcal{P}_L \rangle$ to $N.parent.branches$
5:         Add $\langle N_R, \mathcal{P}_R \rangle$ to $N.parent.branches$
6:         Remove $\langle N, \mathcal{P} \rangle$ from $N.parent.branches$
7:     end if
8:     $N = N.parent$
9: end while
```

3.2.2 *Insert Upward Sweep.*

During execution of the upward sweep, `adjustTree` (Algorithm 2) splits overflowing nodes (line 3). If placing the resulting nodes into the parent (line 4) causes the parent to overflow, then the split *propagates* upwards by splitting the parent. If eventually the root overflows, then a new node will be allocated and set to be the root. Each of the two nodes created from the old root become the new root's children. For example, assume that E already contains three points before being selected to contain the new point in Figure 4d. E will then contain 4 points, exceeding the maximum fanout of $m = 3$, and thus must be split.

The splitting of nodes always begins at the leaf level. To split a node, data and bounding polygons are divided along some line determined by `partitionNode` (Algorithm 4). This dividing line is called a *partition line*. NIR-Tree partition lines are determined by computing a d-dimensional average point called the *geometric median* (lines 4, 8). For leaf nodes, the median is the average of data

Algorithm 3 splitNode(N, l, d) $\rightarrow \langle N_L, \mathcal{P}_L \rangle, \langle N_R, \mathcal{P}_R \rangle$

```
Require: $N$ is a node, $l \in \mathbb{R}$, $d$ a dimension
1: if $N$ not root then
2:     $\mathcal{P}_{ref} = N.parent.poly$
3: else if $N$ is leaf then
4:     $\mathcal{P}_{ref} = \{R\}$ where $R$ is the smallest rectangle enclosing all points in $N.data$
5: else
6:     $\mathcal{P}_{ref} = \{R\}$ where $R$ is the smallest rectangle enclosing all polygons in $N.branches$
7: end if
8: $\mathcal{P}_L, \mathcal{P}_R$ = left and right side of $\mathcal{P}_{ref}$ sliced along $l$ in dimension $d$
9: $N_L = N_R = \langle N.parent, \emptyset \rangle$
10: Branch $left = \langle N_L, \mathcal{P}_L \rangle$
11: Branch $right = \langle N_R, \mathcal{P}_R \rangle$
12: for $p \in N.data$ do
13:     if $p \in \mathcal{P}_L$ and $p \in \mathcal{P}_R$ then
14:         $N_{tie} = N$ with smaller of $|N_L.data|$ and $|N_R.data|$
15:         $N_{tie} = \{p\} \cup N_{tie}.data$
16:     else if $p \in \mathcal{P}_L$ then
17:         $N_L.data = \{p\} \cup N_L.data$
18:     else if $p \in \mathcal{P}_r$ then
19:         $N_R.data = \{p\} \cup N_R.data$
20:     end if
21: end for
22: for $B \in N.branches$ do
23:     if $B.\mathcal{P} \cap \mathcal{P}_L = \emptyset$ then
24:         $N_R.branches = \langle B.child, B.\mathcal{P} \rangle \cup N_R.branches$
25:         $B.child.parent = N_R$
26:     else if $B.\mathcal{P} \cap \mathcal{P}_R = \emptyset$ then
27:         $N_L.branches = \langle B.child, B.\mathcal{P} \rangle \cup N_L.branches$
28:         $B.child.parent = N_L$
29:     else
30:         Branches $b_L, b_R$ = splitNode($B.child$, $l$, $d$)
31:         $N_L.branches = \langle b_L.child, b_L.\mathcal{P} \rangle \cup N_L.branches$
32:         $b_L.child.parent = N_L$
33:         $N_R.branches = \langle b_R.child, b_R.\mathcal{P} \rangle \cup N_R.branches$
34:         $b_R.child.parent = N_R$
35:     end if
36: end for
37: refine($\mathcal{P}_L$)
38: refine($\mathcal{P}_R$)
39: return $b_L, b_R$
```

points. For routing nodes, the median is the average of bounding polygon rectangle corners. These medians determine a possible partition line in each dimension. For leaf nodes the dimension with highest sample variance is chosen (line 3). For routing nodes the dimension whose line divides the fewest bounding polygons is chosen (line 9). Tangibly, Figure 4f shows the two possible partition lines $P1$ and $P2$ of the leaf E. We will assume that the most variate dimension is x, and thus $P1$ is chosen to be the partition line.

With a partition line computed, `splitNode` (Algorithm 3) divides the current node along it (line 8). In Figure 4g, E_1 lies entirely to the

Algorithm 4 partitionNode(N) $\rightarrow l, d$

```
Require: $N$ is a node, $l \in \mathbb{R}$, $d$ a dimension
1: if $N$ is a leaf then
2:     $S = \{(v, d) \mid v = \sigma^2(N.data_d)\}$
3:     $G$ =geometricMedian($N.data$)
4:     $d' = s.d$ where $s \in S$ such that $s.v$ is maximum
5: else if $N$ is a routing node then
6:     $R = \bigcup_{B \in N.branches} B.\mathcal{P}.rectangles$
7:     $corners = \bigcup_{r \in R}\{r.ll, r.ur\}$
8:     $G$ =geometricMedian($corners$)
9:     $d' = d$ such that the line $a = G_d$ passes through minimum number of $r \in R$
10: end if
11: return $G_{d'}, d'$
```

left of the partition line, so it is converted into the bounding polygon for a new left-hand node G. E_2, E_3 are entirely to the right of the partition line so they remain the bounding polygon for the now right-hand node E. Data points are divided based on containment in left- and right- hand nodes (line 13), E and G respectively in our example. If a point lies on the perimeter of both the left- and right-hand bounding polygons, then the tie is broken by choosing the node containing fewest data points.

Continuing our example, at the level above E, B receives the new node G (Figure 4f) and overflows, requiring the split of E to be propagated by splitting B. New partition lines $P3$ and $P4$ are computed as described above; because $P3$ splits one bounding polygon and $P4$ splits two, $P3$ is used. Both partition lines divide B's child F's bounding polygon, and F is therefore split by a recursive call to `splitNode` (line 30). The current partition line is passed to the recursive call so F is split along the same line as its parent, $P3$ in Figure 4g. Importantly, using the partition line computed for B means F is split into F and H even though it does *not* exceed the maximum fanout m. F and H (Figure 4h) lie entirely on opposite sides of $P3$ and are placed into B, and B's new sibling I, respectively. An identical process follows to split the root.

3.2.3 Deletion. Similarly to `insert`, the deletion operation `remove` also has a forward root-to-leaf and a backward leaf-to-root stage. In the forward root-to-leaf stage, `remove` searches for the point to delete in exactly the same manner as `search`, which is described in the next section (Section 3.3). Once the leaf node containing the requested point is located, deletion proceeds through the second leaf-to-root stage lazily. In a process almost identical to `adjustTree`, deletion walks the tree backward, replacing calls to `splitNode` with branch deletions when $|N.branches| = 0$ or $|N.data| = 0$. No further reorganization of the tree is required.

3.3 Searching

Finally, we describe the NIR-Tree `search` (Algorithm 5) operation that follows from the other operations described above. Point searches are equivalent to range searches executed with a query rectangle R whose two defining corner points are equal. `search` begins at the root of the NIR-Tree and proceeds downward towards the leaves. At each tree level the query rectangle is tested for intersection with the bounding polygon in each of the current node's branches. We determine intersection between the query rectangle and a bounding polygon (line 3) by computing rectangle-rectangle intersection between the query and each rectangle comprising the polygon. Any node whose associated bounding polygon intersects the query rectangle is recursively searched. Points in leaf nodes reached by a search are filtered based on containment in the query rectangle (line 7). Points may be independent or associated with data items as keys. This choice determines if points themselves are placed into the output accumulator A (line 9) or if the data associated with the point-keys are placed into A.

Algorithm 5 search(N, R) $\rightarrow A$

```
Require: $N$ is a node, $R$ is a rectangle
 1: $A = \emptyset$
 2: for $B \in N.branches$ do
 3:     if $R \cap B.\mathcal{P} \neq \emptyset$ then
 4:         $A = A \cup$ search($B.child$, $R$)
 5:     end if
 6: end for
 7: for $p \in N.data$ do
 8:     if $p \in R$ then
 9:         $A = A \cup \{p\}$
10:     end if
11: end for
12: return $A$
```

4 NIR-TREE ANALYSIS

In this section, we provide definitions of the relevant terms used throughout the paper. We then formally analyze and prove the zero-area intersection guarantee among bounding polygons in the NIR-Tree.

4.1 Geometric Primitives

It is assumed that the underlying space discussed is $\mathbb{R}^d$, thus, we define a point in the obvious way (see Figure 7a for an example):

Definition 3. *A **point** $p = (p_0, \ldots, p_d) \in \mathbb{R}^d$.*

While points have no obvious total sort order we use the related concept called dominance:

Definition 4. *A point p' **dominates** a point p, denoted $p \leq p'$, if and only if $\forall i$, $p_i \leq p'_i$.*

Together, points and the dominance relation enable the specification of rectangles. Note that rectangles, as defined here, are always axis-aligned. Further, observe that a rectangle is defined with two characteristic points, ll and ur, representing the lower left and upper right respectively (see Figure 7b for an example):

Definition 5. *A **rectangle** $R = \{ll, ur \in \mathbb{R}^d \mid ll \leq ur\}$ with corner points ll and ur.*

Definition 6. *A point p is contained by a rectangle R, denoted $p \in R$, if and only if $R.ll \leq p \leq R.ur$.*

From a collection of rectangles, polygons, and what it means for a point to be in a polygon, are defined. Again observe that since rectangles are axis-aligned, polygons will also be axis-aligned (Figure 7c).

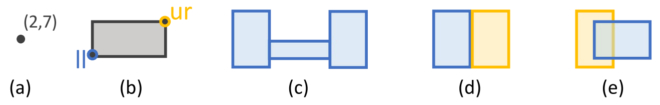

Figure 7: Definitional examples.

Definition 7. *A **polygon** is a set of n rectangles $\mathcal{P} = \{R_0, \ldots, R_n\}$.*

Definition 8. *A point p is contained by a polygon $\mathcal{P}$, denoted $p \in \mathcal{P}$, if and only if $\exists R \in \mathcal{P}$ such that $p \in R$.*

4.2 Geometric Relationships

We now explicitly define how geometric primitives interact through intersection, perimeter, and disjointness.

Definition 9. *The **intersection** of rectangles R and R' is defined as $R \cap R' = \{\min(R.ll, R'.ll), \max(R.ur, R'.ur)\}$. Where max is the point that result from choosing the coordinate-by-coordinate maximum of the inputs. Similarly for min.*

Since it is useful to talk about points on the "edge", or "border", or "perimeter" of a rectangle, the concept of perimeter is defined next.

Definition 10. *A point p is said to be on the **perimeter** of a rectangle R if and only if $\exists d$ such that $p_d = R.ll_d$ or $p_d = R.ur_d$.*

Polygons that intersect may have area (volume, hyper-volume) in common that is zero or greater. To distinguish between zero-area and positive-area intersection, we introduce the idea of disjoint polygons. If polygons are disjoint then their area of intersection is zero. Visual examples of disjointness and intersection are found in Figures 7d and 7e respectively.

Definition 11. *The **intersection** of polygons*

$$\mathcal{P} \cap \mathcal{P}' = \bigcup_{R \in \mathcal{P}, R' \in \mathcal{P}'} \{R \cap R'\}$$

Definition 12. *Rectangles R and R' are **disjoint** if and only if:*

(1) *$R \cap R' = \emptyset$ or*
(2) *$\forall r \in R \cap R'$, r on the perimeter of R and R'*

Definition 13. *Two polygons $\mathcal{P}$ and $\mathcal{P}'$ are **disjoint** if and only if $\forall R \in \mathcal{P}$ and $\forall R' \in \mathcal{P}'$, R and R' are disjoint.*

Rectangle fragmentation is central to insertion (Algorithm 1). Described in Section 3.2 and illustrated in Figure 5, we now give a precise definition for the concept of rectangle-rectangle fragmentation and polygon-polygon fragmentation.

Definition 14. *A rectangle **fragmented** by a rectangle, $fragment(R, R') = \{C_0, F_0, \ldots, C_d, F_d\}$ where ceilings C_i and floors F_i are defined, with $j \in [0, d]$, as follows:*

$$(1)\ C_i.ll_j = \begin{cases} \max(C_0.ur_0, \ldots, C_{i-1}.ur_0) & j < i \\ R'.ur_i & j = i \\ R.ll_j & j > i \end{cases}$$

$$(2)\ C_i.ur_j = \begin{cases} \min(C_0.ll_0, \ldots, C_{i-1}.ll_0) & j < i \\ R.ur_j & j \geq i \end{cases}$$

(3) $F_i.ll$ *obtained similarly by replacing* $ur \rightarrow ll$ *in (2).*
(4) $F_i.ur$ *obtained similarly by replacing* $ll \rightarrow ur$ *in (1).*
(5) $\forall i, C_i, F_i$ *are rectangles.*

Polygon fragmentation is the pairwise generalization of rectangle fragmentation.

Definition 15. *A polygon* ***fragmented*** *by a polygon,*

$$fragment(\mathcal{P}, \mathcal{P}') = \bigcup_{R \in \mathcal{P}, R' \in \mathcal{P}'} fragment(R, R')$$

4.3 Zero-Area Intersection Guarantee

Having defined geometric primitives and the relationships between them, we define the invariant properties for the NIR-Tree, followed by a proof that the NIR-Tree respects these properties before and after the execution of any operation. The NIR-Tree's zero-area intersection guarantee is exactly Definition 16 property (2).

Definition 16. *The* ***invariant****, denoted* IV*, for a NIR-Tree* T *and* $\forall N \in T$ *consists of:*

(1) $N.poly \subseteq N.parent.poly$ *or* $\forall p \in N.data, p \in N.poly$
(2) $\forall N' \neq N$ *at height* h*,* $N'.poly$ *and* $N.poly$ *are disjoint*

These invariant properties of the NIR-Tree ensure that children are entirely enclosed by their parent, and that sibling nodes have zero-area intersections. These properties are shown to hold by the following theorem.

Theorem 1. *For any NIR-Tree* T*, if* T *satisfies* IV *before the execution of* `insert`*,* `search`*, or* `delete` *then* T *also satisfies* IV *afterwards.*

Proof. Observe that as described above, neither `search` nor `delete` may affect IV since neither alters the bounding polygon of any node. Further, note that Algorithm 1 lines 7-11 and Algorithm 3 are the only operations in `insert` which alter bounding polygons. A discussion of `refine` is omitted since it simply removes redundant rectangles and hence does not alter the logic of this proof.

Observe that Algorithm 1 is executed first and then Algorithm 3 is executed second as part of Algorithm 2. Thus it will suffice to show that, at the conclusion of Algorithm 1, T satisfies $IV(1)$ and $IV(2)$ and subsequently that at the conclusion of Algorithm 3, T satisfies $IV(1)$ and $IV(2)$.

First, let us consider $IV(1)$. On inspection, Algorithm 1 explicitly satisfies $IV(1)$ by intersecting $B.\mathcal{P}$ with $B.parent.poly$ and the end of execution on line 11.

Second, let us consider $IV(2)$. Let $N = B.child$ from Algorithm 1 line 6. Line 7 possibly causes $N.poly$ at height h to violate $IV(2)$. However, suppose S is any sibling of N. By construction of the NIR-Tree, S is at the same height h as N. If, after execution of line 7, $S.poly$ is not disjoint from $N.poly$ then by definition of rectangle fragmentation $S.poly$ is disjoint from $N.poly$ after execution of line 9. Now suppose S is *any* node at height h in T. Since Algorithm 1 lines 7-11 do not alter $S.parent$ or $N.parent$, we know that $S.poly \subseteq S.parent.poly$ and that $S.parent.poly$ is disjoint from $N.parent.poly$. It therefore follows that after line 11, since $N.poly \subseteq N.parent.poly$, $N.poly$ is disjoint from $S.poly$ by definition of subset and disjointness. Thus $IV(1)$ and $IV(2)$ are satisfied for any N in T after Algorithm 1 is executed.

To complete the proof, we show that $IV(1)$ and $IV(2)$ hold for Algorithm 3 if $IV(1)$ and $IV(2)$ hold before its execution.

First, let us consider $IV(1)$. By definition of $\mathcal{P}_L$ and $\mathcal{P}_R$ at Algorithm 3 line 8, $\forall R_L \in \mathcal{P}_L$, $R_L.ll_d \leq l$ and $R_L.ur_d \leq l$. Moreover, $\forall R_R \in \mathcal{P}_R$, $R_R.ll_d \geq l$ and $R_R.ur_d \geq l$. Additionally, $\mathcal{P}_L \subseteq \mathcal{P}_{ref}$ and $\mathcal{P}_R \subseteq \mathcal{P}_{ref}$. Therefore, $N_L.poly = \mathcal{P}_L \subseteq N_L.parent.poly = \mathcal{P}_{ref}$ and $N_R.poly = \mathcal{P}_R \subseteq N_R.parent.poly = \mathcal{P}_{ref}$ since $N_L.parent = N_R.parent = N.parent$.

If N is a leaf then we must show that $\forall p \in N_L.data, p \in N_L.poly$. When N is leaf it suffices to consider only Algorithm 3 lines $12 - 21$. Observe that $N_L.data = \emptyset$ by construction at line 9. In conjunction with the definition at line 9, lines 16 and 18 yield $p \in N_L \leftrightharpoons p \in \mathcal{P}_L = N_L.poly$. Without loss of generality, the same argument may be applied to $p \in N_R.data$ and $N_R.poly = \mathcal{P}_R$. $IV(1)$ is therefore satisfied when N is a leaf.

If N is not a leaf then we must show that $\forall B_L \in N_L.branches$, it is the case that $B_L.\mathcal{P} \subseteq \mathcal{P}_L$ upon conclusion of the execution of lines $22 - 36$. If B_L was placed in N_L at line 27 it is clear that $B_L.\mathcal{P} \subseteq \mathcal{P}_L$ since $B_L.\mathcal{P} \cap \mathcal{P}_R = \emptyset$ and it is certainly the case that $B_L.\mathcal{P} \subseteq \mathcal{P}_{ref}$ by our hypothesis that $IV(1)$ is satisfied before execution. Otherwise, B placed in N_L at line 31 implies $B_L.\mathcal{P}$ was sliced during the recursive call at line 30 along l in d since l and d are passed to the recursive call unchanged. Thus, $\forall R \in B_L.\mathcal{P}$, $R.ll_d \leq l \Rightarrow B_L.\mathcal{P} \subseteq \mathcal{P}_L$ as desired. Without loss of generality, the same argument made here may be applied $\forall B \in N_R.branches$ with respect to $\mathcal{P}_R$. $IV(1)$ is therefore satisfied at the conclusion of Algorithm 3 for all nodes N.

Second, let us consider $IV(2)$. Again by construction of $\mathcal{P}_L$ and $\mathcal{P}_R$ at Algorithm 3 line 8, we have that $\forall p \in \mathcal{P}_L \cap \mathcal{P}_R, p_d = l$ and therefore $\mathcal{P}_L$ and $\mathcal{P}_R$ are disjoint. Since $\mathcal{P}_L \subseteq \mathcal{P}_{ref}$ and $\mathcal{P}_R \subseteq \mathcal{P}_{ref}$ by construction, we have that $N_L.poly = \mathcal{P}_L$ and $N_R.poly = \mathcal{P}_R$ are disjoint from all $S.poly$ at height h because $N.poly = \mathcal{P}_{ref}$ is disjoint from all $S.poly$ at height h by our hypothesis that $IV(2)$ is satisfied before execution. The same argument may be applied to the recursive call at line 30 since l and d are passed to $B.child$ unchanged. The transitivity of subset implies $b_L.\mathcal{P} \subseteq \mathcal{P}_L \subseteq N.parent.poly$ and $b_R.\mathcal{P} \subseteq \mathcal{P}_R \subseteq N.parent.poly$ as desired. The transitivity of subset is not limited and so this argument may be applied recursively as many times as necessary to satisfy $IV(2)$. With the execution of Algorithm 3 thus complete, $IV(2)$ is satisfied in all cases.

Therefore we have shown that all operations on a valid NIR-Tree T satisfying IV result in a tree satisfying $IV(1)$ and $IV(2)$, thus satisfying IV. □

5 PERFORMANCE EVALUATION

In this section, we experimentally demonstrate the performance advantage of the NIR-Tree over the R-Tree [11], R$^+$-Tree [22], and R*-Tree [3], in terms of search and insertion efficiency using both real and synthetic datasets.

5.1 Experimental Setup

We first describe our experimental setup and methodology, including machine configuration.

5.1.1 Software. We compared the NIR-Tree against our implementations of the R-Tree [11], R$^+$-Tree [22], and R*-Tree [3]. The R$^+$-Tree was selected for comparison because of its similar partition line and recursive downward split mechanism. The R*-Tree was selected for comparison due to its popularity and widely accepted baseline status. All trees compared against resided entirely in main memory during experiments. We empirically determined the maximum fanout m of all trees so as to provide the best point and range query performance ($m = 50$ for the NIR-Tree, $m = 100$ for all others). Trees that required a minimum fanout were configured to use half the maximum fanout.

5.1.2 Hardware. We evaluated trees on a machine with 4× Intel E5-4620v2 2.60GHz CPUs (32 physical cores and 20MB L3 cache), 256GB of physical memory, and a 400GB Intel S3700 SSD.

5.1.3 Methodology. We constructed each tree by sequentially inserting all of the points in each dataset. Point and range queries were executed on the constructed tree. Statistics such as tree memory usage and tree height were gathered at the end of each experiment. Every point in each dataset was queried in the tree exactly once, and 1000 or more range queries were executed depending on the dataset. Operation (e.g., search) times reported are averages over five independent runs. 95% confidence intervals are shown as error bars (at times barely visible) around each averaged operation time.

5.2 Datasets and Queries

We describe next the six datasets and associated queries used to evaluate the trees.

5.2.1 Datasets. Each of the four trees were constructed using each of the following datasets:

- **California**: A subset of the larger TIGER/Line dataset mapping the United States [4, 5]. The data is a mixture of points and rectangles expressed as doublets or quartets of GPS coordinates. Due to the mixed nature of the data, we represented rectangles by their centroids. Dimensions = 2. Size = 1.8M points.
- **Biological**: A large real collection of points representing biological features [4, 5]. Points cluster in a pyramid-shaped region between the X and Y axes with a distinct break before clustering in two sheets along the X and Y axes. Dimensions = 3. Size = 11.9M points.
- **Forest**: Real data describing a 30×30m section of forest, collected by the United States Forestry Service [4, 5]. Attributes describing elevation, distance to water, fire, and roadways are indexed. Data is skewed towards water features and directional stream patterns are present. Dimensions = 5. Size = 581K points.
- **Canada Roads**: A large collection of polygons representing the road network of Canada created by Statistics Canada [7]. All the corners of each polygon were indexed due to their axis-unaligned nature. Data is densest around population centers in the South West and South East, and sparser in the Mid-West and North. Dimensions = 2. Size = 19.4M points.
- **Gaia**: A subset of the stars observed by the European Space Agency's Gaia mission to map our galaxy [6, 8]. All the stars within a square portion of the sky, anchored by the star Proxima Centauri, are indexed. Points describing each star consist of two coordinates placing each star in the sky, using degrees, and a third attribute measuring distance from the Sun in parsecs. Data is dense along the galactic plane, becoming sparser above and below the plane. Dimensions = 3. Size = 18M points.
- **Uniform**: A synthetic collection of points distributed in the unit square. Points are generated by choosing each coordinate from the range $[0.0, 1.0]$ with uniform probability. Dimensions = 2. Size = 10M points.

5.2.2 Queries. All datasets were queried using rectangles specified in advance of the experiment. Each query rectangle contained about 1000 points. Query rectangles for California, Biological, and Forest data were provided by the benchmark [4, 5]. Query rectangles for Canada Roads, Gaia, and Uniform data were generated prior to experimentation and in a manner analogous to query rectangles provided by the California, Biological, and Forest benchmarks to ensure uniform comparison. In particular, Uniform query rectangles were generated by selecting a random point in the unit square to be ll of the query rectangle (see Definition 5) and then adding a value α to each coordinate of ll to get ur. α was selected so that the resulting query rectangle contained 1000 points in expectation. Query rectangles for Canada Roads and Gaia data were generated by iteratively increasing 5000 small rectangles centered around 5000 randomly selected points until each rectangle contained about 1000 points.

5.3 Results

We present and analyze the point query and range query times as well as insertion times for the NIR-Tree and its counterparts. We find that among the trees that provide consistently competitive range queries, the NIR-Tree is superior given its categorically faster point queries and insert operations.

5.3.1 Point Query Performance. Recall from Section 1 that the NIR-Tree's scatter reduction is highly efficient and occurs only when a point lies on the perimeter of two bounding polygons. Scatter in the NIR-Tree was extremely rare; no point query in the NIR-Tree ever accessed more than 7 nodes, in contrast to the R$^+$-Tree where, for example, hundreds of thousands of point queries accessed 14 or more nodes on the Canada Roads dataset. On all datasets, the NIR-Tree executed 99% or more point queries optimally, clearly outperforming the second place R$^+$-Tree which executed on average 65% of point queries optimally.

Efficient point queries allow the NIR-Tree to out-compete every tree on every dataset with the sole exception of the R$^+$-Tree on the Biological dataset. The R*-Tree, popular for its strong range query performance, is outperformed by the NIR-Tree by an average factor of 2.2×. Additionally, the R*-Tree is usually also outperformed by the R$^+$-Tree. The NIR-Tree saw its closest competition from the R$^+$-Tree instead of the R*-Tree, because the R$^+$-Tree uses a similar partition-line oriented split to reduce intersection and thus scatter. The R$^+$-Tree enjoyed an advantage over the NIR-Tree on the Biological dataset by being one level shorter. R$^+$-Tree height was not due to better node utilization, but rather to a maximum fanout twice that of the NIR-Tree. Maximum fanout is set relatively low for the NIR-Tree to keep bounding polygons simple. In general, larger fanouts in the NIR-Tree require more complex bounding because they must remain disjoint, yet enclose more children. In aggregate, the R$^+$-Tree was slightly faster than the NIR-Tree on the

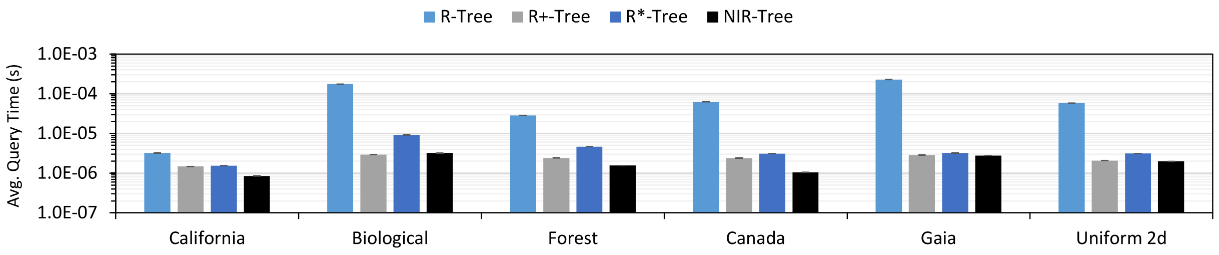

Figure 8: Point query times (log scale).

Biological dataset through a combination of mild scatter reduction, and consistently requiring one less random memory access for point queries. Despite this, the NIR-Tree used 24% less time than the R$^+$-Tree to perform all point containment operations, and executed 20% more queries optimally. These advantages for the NIR-Tree are evident in all five other datasets.

On all datasets the NIR-Tree's competitors were ordered in the same fashion: the R$^+$-Tree was closest, followed by the R*-Tree, and then by the R-Tree. The difference between R$^+$-Tree and R*-Tree point search times decreased to its lowest point on the California dataset.

5.3.2 Update Performance. Existing multidimensional indexes trade off insertion time for query performance [1, 3, 14, 16, 17]. Breaking from this norm, the NIR-Tree improves insertion time without compromising query times. For example, the NIR-Tree is significantly faster to construct than the R*-Tree while having better point query performance (Section 5.3.1), and equivalent range query performance (Section 5.3.3). For insertions, the R*-Tree consumes, on average, a substantial 27.8× the insertion time of the NIR-Tree.

By allowing leaf or routing nodes to be re-inserted once on each level, the R*-Tree induces a significant amount of tree restructuring during insertion, resulting in a denser tree (Section 5.3.4) and very high insertion times. The NIR-Tree delivers a large advantage because it avoids expensive re-insertions. In fact, the R*-Tree runs in $O(n^2)$ time if there are re-insertions [3]. By leveraging bounding polygons instead of re-insertions to reduce query scatter, the cost of NIR-Tree insertions are kept small. These insertion strategies give rise to the differences between the NIR-Tree and the R*-Tree in Figure 9. Other trees use neither re-insertions nor bounding polygons, thereby trading-off insertion time for query performance.

Among all trees, the R*-Tree requires the most amount of time to insert, followed by the R-Tree, followed by the NIR-Tree, and then the R$^+$-Tree. This ordering is prevalent on all datasets. As described above, the NIR-Tree is much faster than the R*-Tree due to quadratic operations present within re-inserts. Next, the R-Tree, using an $O(n^2)$ split algorithm but no re-insertions, still underperforms the NIR-Tree whenever the average size of a bounding polygon is below 3. The NIR-Tree's superior insertion performance is reflected by its average insertion time speedup of 1.3× the R-Tree. The R$^+$-Tree is 81× faster than the R*-Tree and fastest of all the trees for two reasons. First, the R$^+$-Tree's split algorithm cost metric is computed using bounding rectangles instead of the more flexible bounding polygons of the NIR-Tree, yielding faster splits. Second, by creating stripes and repeatedly splitting within them along the same dimension, the R$^+$-Tree executes fewer downward splits than the NIR-Tree (see Algorithm 3). Stripes, though computed quickly, are generally a poor bounding shape and impose a significant cost during range searches. Long, thin rectangles provide unsatisfactory range query performance (Section 5.3.3) because range queries extend across many stripes and waste time filtering data at opposite ends of the stripes. For example, in the California dataset, the NIR-Tree has a rectangular side length ratio 7:29, while the R$^+$-Tree stripes have an extreme side length ratio of 3:1226. As expected, its range query performance is the least desirable among all trees tested. Overall, no tree with faster construction time than the NIR-Tree outperformed the NIR-Tree on range queries, as we discuss in the next section.

Table 1: Tree size by dataset measured in nodes.

Dataset	R-Tree	R$^+$-Tree	R*-Tree	NIR-Tree
California	30,526	30,386	29,843	75,298
Biological	187,744	198,944	167,306	553,831
Forest	9,222	12,599	8,334	31,945
Canada Roads	310,016	310,219	281,370	699,203
Gaia	286,303	267,943	258,304	709,576
Uniform 2d	155,943	146,131	143,639	325,449

5.3.3 Range Query Performance. Across different datasets and dimensions, range queries within the NIR-Tree performed at parity with its fastest competitor, the R*-Tree (Figure 10). On skewed datasets, the R*-Tree and R$^+$-Tree were the NIR-Tree's closest competitors. On all other datasets, the R-Tree and R*-Tree were the NIR-Tree's closest competitors. The NIR-Tree and R*-Tree range queries were essentially equivalent, the NIR-Tree being within 8% of the R*-Tree on average, and outperforming the R*-Tree on the Canada Roads dataset. Examining range query performance through the lens of dataset skew, we see that the gap between the R*-Tree and the NIR-Tree is largest on the highly skewed Forest and Biological datasets, and smallest on map-style datasets (California, Canada Roads, Gaia), which exhibit less skew.

Range queries within the NIR-Tree and R*-Tree accessed a similar number of nodes on map-style datasets as well as executing in a similar amount of time. For example, California dataset range queries accessed an average of 61 nodes in the NIR-Tree while accessing an average of 52 nodes in the R*-Tree. Despite indexing

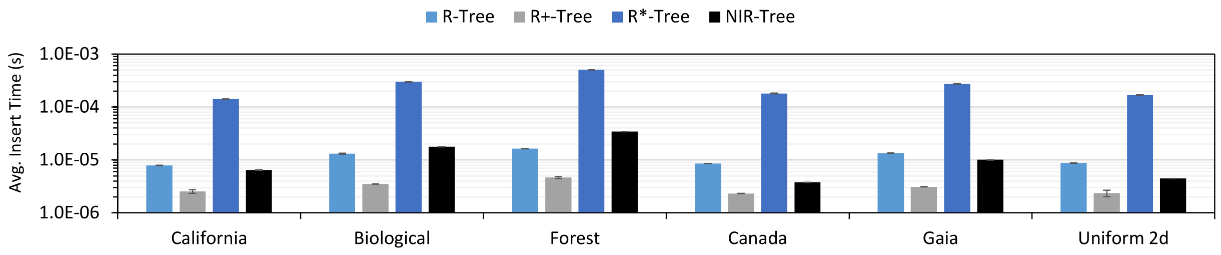

Figure 9: Insertion times (log scale).

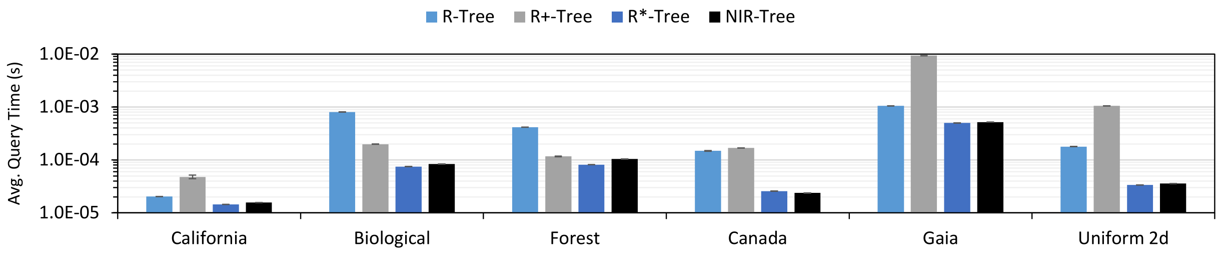

Figure 10: Range query times (log scale).

with more complicated bounding polygons, the NIR-Tree spent 14% less time than the R*-Tree testing intersection between bounding objects and the query rectangle, since the dense nodes of the R*-Tree required many more intersection tests. The similar intersection test time, and similar node access time meant the NIR-Tree and R*-Tree performed similarly across these datasets.

In contrast to map-style datasets, the NIR-Tree accessed more nodes than the R*-Tree on the highly skewed Forest and Biological datasets. For example, range queries on the NIR-Tree indexing Biological data accessed an average of 124 nodes, while accessing an average of 73 nodes in the R*-Tree. Higher node accesses are expected since the NIR-Tree indexed using half the maximum fanout of the R*-Tree (Section 5.3.4). However, with skew increasing the average number of rectangles per bounding polygon, 4.78 vs. 2.68 for the Gaia dataset, the NIR-Tree and R*-Tree spent an identical total time of 2.13s performing intersection tests, resulting in the NIR-Tree not making up for higher node accesses.

Among other trees, dataset distribution and higher dimensionality explain the R-Tree and R$^+$-Tree's relative performance inversion on Biological and Forest datasets. Higher dimension datasets ameliorated the R$^+$-Tree's bounding rectangles' extreme side length ratio because they became long and thin in only one dimension while other dimensions became more rectangular. Consider the Forest dataset: R$^+$-Tree bounding rectangles had an average side length ratio of 30:112:31:113:413, which while extreme in the 30:413 case, exhibit much better relative ratios e.g., 30:31 and 30:112, for other cases. Additionally, the roughly pyramid-shaped data distributions of Biological and Forest meant striping did not always cause range queries to filter points on the opposite extreme of the space.

5.3.4 Tree Structure. The unbounded complexity of NIR-Tree polygons may mislead one to assume that the memory usage of the NIR-Tree is untenable. Our analysis of tree structures show these concerns to be unfounded. Across all experiments, the average NIR-Tree polygon was formed using only 3.3 rectangles. Even in extremely skewed datasets, such as Biological and Forest, the average polygon was still formed using an average of 5.4 rectangles (Table 2). As the number of dimensions rise, it is only reasonable that more rectangles are required to index any given node, because there are effectively more neighbours which must be avoided.

Table 2: Polygon sizes in the NIR-Tree.

Dataset	Total Size of All Polygons	Avg. Polygon Size
California	215,256	2.86
Biological	2,649,392	4.78
Forest	194,158	6.08
Canada Roads	1,254,812	1.79
Gaia	1,902,288	2.68
Uniform 2d	446,566	1.37

To index without intersection, the NIR-Tree uses only about 20% more memory compared to the space-efficient R*-Tree (Table 3). Memory overhead in the NIR-Tree is largely accounted for by the storing of bounding polygons. For example, the index for California stores 3.28MB of bounding polygons, Biological stores 40.42MB, and Gaia stores 43.5MB. Competitor trees contained no more than 1.2× the number of nodes of any other (Table 1), aligning with their memory consumption pattern (Table 3). The NIR-Tree's nodes had a maximum of 50 children while competitor tree nodes had a maximum of 100 children. While the NIR-Tree is expected to use about 2× the number of nodes used by any of its competitor trees, the slightly higher observed factor of 2.6× for the NIR-Tree is

Table 3: Memory usage (MB).

Dataset	R-Tree	R^+-Tree	R^*-Tree	NIR-Tree
California	32	31	31	38
Biological	300	296	290	349
Forest	23	24	23	27
Canada Roads	335	328	319	386
Gaia	455	446	439	516
Uniform 2d	172	168	164	192

accounted for by the downward splits during Algorithm 3, which keep the NIR-Tree intersection free. All trees had similar heights, indexing datasets in 4 levels with the NIR-Tree occasionally using 5 levels for the larger datasets.

6 RELATED WORK

The field of multidimensional indexing is well studied [21]. We divide prior approaches into two categories: data partitioning and space partitioning, and contrast each with the NIR-Tree.

6.1 Data Partitioning

Data partitioning indexes cluster data and normally recursively form trees from these groups of data. Data partitioning indexes are a popular type of multidimensional index implemented in mainstream databases such as PostgreSQL [10] and MySQL [20]. The NIR-Tree is closely related to the works in this class of partitioning.

The R-Tree [11] is a multidimensional extension of the B-Tree [2]. Rectangular data objects are recursively grouped into rectangular regions to create a balanced tree. Unlike one dimensional B-Tree ranges, R-Tree regions may intersect because their *volumetric* data objects take up space and may intersect. Unlike an R-Tree, the NIR-Tree considers points rather than volumetric objects, exploiting the infinitesimal nature of points to eliminate positive area intersection.

The R^*-Tree [3] improves upon the R-Tree by re-inserting the data or children of overflowing nodes. Re-insertion creates a restructuring effect larger than node splits alone, usually resulting in denser, better organized trees. The R^*-Tree also identifies four axes of optimization along which data partitioning trees may move. These axes are: *area* covered by bounding objects, area of *intersection* between bounding objects, and *perimeter* of bounding objects should all be minimized while at the same time nodes should be dense and have high *utilization*. The RR^*-Tree [5] matches the performance of the R^*-Tree by considering one additional factor during splits and chooseSubtree. When selecting a leaf in chooseSubtree, the RR^*-Tree optimizes for minimum area, and minimum perimeter when the dimensionality of the data is high. An extra factor, considered during node splits, is the logical number of children on each side of the split. The RR^*-Tree seeks to balance splits logically while also balancing the splits geometrically. The NIR-Tree balances leaf splits logically and geometrically while totally eliminating intersection.

Optimizing for intersection area, the NIR-Tree and R^+-Tree [22] both seek to eliminate intersection. The R^+-Tree lessens intersection by splitting nodes along a partition line. Any bounding objects lying across the partition line are recursively split. Volumetric data lying across the partition line cannot be split and are instead duplicated to both sides of the split. While the two resulting nodes from a split are disjoint in an R^+-Tree, the tree still creates intersection when expanding bounding rectangles during insertion. By contrast, the NIR-Tree uses rectangle fragmentation in addition to partition lines to guarantee disjoint regions on insert.

The Hilbert R-Tree [14] uses a metric predicated on the Hilbert space-filling curve. Instead of selecting bounding rectangles with least additional area, or least intersection area (see Figure 3), the Hilbert R-Tree selects bounding rectangles whose centroids have a Hilbert number closest to the point of interest's Hilbert number. However, computing such numbers requires prior knowledge of the extreme values in a dataset and does not consider intersection area between bounding rectangles. The result is a clustering effect, because Hilbert numbers exhibit an order that approximates spatial nearness.

Bulk loading data into an R-Tree has been investigated by the Priority R-Tree [1] that requires the dataset to be known entirely in advance. Then, the Priority R-Tree creates a maximally dense tree with optimal range query disk accesses. However, these properties do not hold if further insertions are executed after bulk loading.

Some trees discard axis-alignment constraints by using arbitrarily oriented bounding rectangles. SICC indexes [24] remove axis-alignment constrains. They group points by temporal locality of insertion, computing a new axis through each group using incremental principal component analysis (roughly analogous to a line of best fit), and fitting a bounding rectangle around the points oriented along this new axis. While incremental analysis and construction is suitable for observational data, it cannot be applied to the general point data indexed by other indexes such as the NIR-Tree.

k-discrete oriented polygon (k-DOP) hierarchies [15], P-Trees [13], and clipped bounding rectangles [23] all illustrate arbitrarily-shaped bounding objects. Complex polygons achieve low coverage and intersection area, but quickly become computationally unwieldy for dimensions greater than three. k-DOP hierarchies define new shapes by having up to k axes in d dimensional space where $k >= d$. However, the complexity of shapes is limited since k is fixed in advance, whereas NIR-Tree polygons are as complex as required by the data. P-Trees on the other hand allow for any number of additional dimensions to be introduced. As a result, polygons in the P-Tree may become arbitrarily complex leading to slowdowns as a result of excessive range checks. By contrast, NIR-Tree polygons are constantly culled during insertion (Algorithm 3), resulting in polygons of small size (Table 2).

Recent work on clipping bounding rectangles [23] removes dead space from the corners of bounding rectangles. Using a close analog of skylines, called stairlines, clipped bounding rectangles create bounding polygons (Definition 7) which tightly enclose their children. Different from clipped bounding rectangles, NIR-Tree bounding polygons are created based on siblings.

6.2 Space Partitioning

An alternative multidimensional indexing approach partitions space rather than data to implicitly form groups. Grid files [18] are a popular space partitioning option [10, 12, 20]. These files divide space into disjoint variable size cells on a grid. Each cell represents a page on disk. Cells are split on overflow, and merged on underflow. Grid files maintain a directory mapping cells to pages. The most significant drawbacks of this approach are the rapid growth of the directory size when the data is sparse and the difficulty of choosing grid granularity appropriately.

Recent approaches build on the grid file by replacing the mapping directory with machine learning models. In addition to replacing the mapping directory, Flood [17] uses its models to optimize attributes of the grid such as resolution. Although Flood is not a static index, it requires that its models be retrained and its data rearranged when a data distribution shift occurs. LISA [16] by contrast, is fully dynamic and uses its models in a hierarchy. LISA is a disk-based index that has a prediction function locating the cell wherein a point lies, and a cell-local prediction function selecting which pages to access. While LISA reduces I/O cost, its CPU cost, the dominant in-memory factor [19], is higher than that of the R*-Tree's, which is equivalent to that of the NIR-Tree's.

Breaking from the grid file mold, Quad-Trees [9] recursively divide space into quadrants until a set number of objects are within each quadrant. However, regular divisions of space lead to deep trees when the data is skewed in some region(s). Deep trees lead to long search paths, exacerbated by dense areas that are likely to be accessed simply because those regions hold more of the data. As shown in our experiments (Section 5), the NIR-Tree is very short, needing < 6 levels for even very large or very skewed datasets. Finally, the Quad-Tree family scales poorly due to its 2^d fanout requirement. Conversely, the NIR-Tree can maintain short trees with fixed fanout independent of the dimension.

7 CONCLUSION

Traditional multidimensional indexes support efficient point and range searches through expensive insertion-time optimization techniques. By limiting the basic bounding shape to a rectangle, existing R-Tree family indexes cannot achieve zero-area intersection. The NIR-Tree introduces flexible bounding polygons as a new type of bounding shape which provably guarantee optimal (zero) intersection between bounding shapes to support efficient insertion and searches. We compared the NIR-Tree with the popular R*-Tree and observed that constructing a NIR-Tree is on average 27× faster than constructing an R*-Tree without trading off for query times. The NIR-Tree achieved equivalent range searches and 2× faster point searches. With simple yet flexible bounding polygons, the NIR-Tree is an efficient, state-of-the-art multidimensional index.

ACKNOWLEDGMENTS

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Canada Foundation for Innovation and Ontario Research Fund. We would like to thank Gladimir Baranoski and Therese Biedl for insightful discussion, and Pei Lin Li, James Lu, and Douglas Hicks for implementation assistance.

REFERENCES

[1] Lars Arge, Mark De Berg, Herman Haverkort, and Ke Yi. 2008. The priority R-tree: A practically efficient and worst-case optimal R-tree. *ACM Transactions on Algorithms (TALG)* 4, 1 (2008), 9.

[2] R. Bayer and E. McCreight. 1970. Organization and Maintenance of Large Ordered Indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control (SIGFIDET '70)*. Association for Computing Machinery, New York, NY, USA, 107–141. https://doi.org/10.1145/1734663.1734671

[3] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD '90)*. Association for Computing Machinery, New York, NY, USA, 322–331. https://doi.org/10.1145/93597.98741

[4] Norbert Beckmann and Bernhard Seeger. 2008. A Benchmark for Multidimensional Index Structures. https://www.mathematik.uni-marburg.de/~rstar/benchmark/

[5] Norbert Beckmann and Bernhard Seeger. 2009. A Revised R*-Tree in Comparison with Related Index Structures. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 799–812. https://doi.org/10.1145/1559845.1559929

[6] A. G. A. Brown, A. Vallenari, T. Prusti, J. H.J. de Bruijne, F. Mignard, R. Drimmel, C. Babusiaux, C. A.L. Bailer-Jones, U. Bastian, and et al. 2016. GaiaData Release 1. *Astronomy and Astrophysics* 595 (Nov 2016), A2. https://doi.org/10.1051/0004-6361/201629512

[7] Statistics Canada. 2016. Census – Road Network Files. https://www12.statcan.gc.ca/census-recensement/2011/geo/RNF-FRR/index-2011-eng.cfm?year=16

[8] Gaia Collaboration, A. G. A. Brown, A. Vallenari, T. Prusti, J. H. J. de Bruijne, C. Babusiaux, and M. Biermann. 2020. Gaia Early Data Release 3: Summary of the contents and survey properties. arXiv:2012.01533

[9] Raphael A. Finkel and Jon Louis Bentley. 1974. Quad trees a data structure for retrieval on composite keys. *Acta informatica* 4, 1 (1974), 1–9.

[10] PostgreSQL Global Development Group. 1996-2021. PostgreSQL. https://www.postgresql.org/

[11] Antonin Guttman. 1984. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD '84)*. Association for Computing Machinery, New York, NY, USA, 47–57. https://doi.org/10.1145/602259.602266

[12] IBM. 2020. IBM Db2. https://www.ibm.com/analytics/db2

[13] H. V. Jagadish. 1990. Spatial Search with Polyhedra. In *Proceedings of the Sixth International Conference on Data Engineering, February 5-9, 1990, Los Angeles, California, USA*. IEEE, 311–319. https://doi.org/10.1109/ICDE.1990.113483

[14] Ibrahim Kamel and Christos Faloutsos. 1994. Hilbert R-Tree: An Improved R-Tree Using Fractals. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 500–509.

[15] James T Klosowski, Martin Held, Joseph SB Mitchell, Henry Sowizral, and Karel Zikan. 1998. Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE transactions on Visualization and Computer Graphics* 4, 1 (1998), 21–36.

[16] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A Learned Index Structure for Spatial Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2119–2133. https://doi.org/10.1145/3318464.3389703

[17] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 985–1000. https://doi.org/10.1145/3318464.3380579

[18] J. Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. 1984. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Trans. Database Syst.* 9, 1 (1984), 38–71. https://doi.org/10.1145/348.318586

[19] Matthaios Olma, Farhan Tauheed, Thomas Heinis, and Anastasia Ailamaki. 2017. BLOCK: Efficient Execution of Spatial Range Queries in Main-Memory. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management (SSDBM '17)*. Association for Computing Machinery, New York, NY, USA, Article 15, 12 pages. https://doi.org/10.1145/3085504.3085519

[20] Oracle. 2021. MySql. https://www.mysql.com/

[21] Hanan Samet. 2004. Object-based and image-based object representations. *ACM Computing Surveys (CSUR)* 36, 2 (2004), 159–217.

[22] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. 1987. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB '87)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 507–518.

[23] Darius Sidlauskas, Sean Chester, Eleni Tzirita Zacharatou, and Anastasia Ailamaki. 2018. Improving Spatial Data Processing by Clipping Minimum Bounding Boxes. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 425–436. https://doi.org/10.1109/ICDE.2018.00046

[24] Sheng Wang, David Maier, and Beng Chin Ooi. 2016. Fast and Adaptive Indexing of Multi-Dimensional Observational Data. *Proc. VLDB Endow.* 9, 14 (2016), 1683–1694. https://doi.org/10.14778/3007328.3007334