A Non-Intersecting R-Tree

by

Kyle Jacob Langendoen

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2021

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

I would like to acknowledge the names of my co-authors who contributed to the research described in this thesis, these include:

- Khuzaima Daudjee
- Brad Glasbergen

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 twice as efficient point queries, and equivalent range query performance while indexing data up to 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, and 1.2× faster than the Revised R*-Tree. Furthermore, point queries in the NIR-Tree complete 2.2× faster than in the R*-Tree, and 3.1× faster than in the Revised R*-Tree, while range queries execute just as quickly.

Acknowledgments

For continued direction, advice, and editorial guidance throughout the entire research process, I would like to thank my supervisor Professor Khuzaima Daudjee. For their constructive comments which contributed to the quality of this work I would like to thank the readers Professors Tamer Oszu and Galdimir Baranoski. For her repeated insights into the geometry problems encountered during this research I would like to thank Professor Therese Biedl. For his technical advice, editorial guidance, and implementation of the R*-Tree I would like to thank Brad Glasbergen. For their initial implementations of the R*-Tree and R^+-Tree I would like to thank Pei Lin Li and James Lu respectively. Finally, for support both technical and personal during the process of creating this thesis I would like to thank the community here at the University of Waterloo.

This work has made use of data from the European Space Agency (ESA) mission *Gaia* (`https://www.cosmos.esa.int/gaia`), processed by the *Gaia* Data Processing and Analysis Consortium (DPAC, `https://www.cosmos.esa.int/web/gaia/dpac/consortium`). Funding for the DPAC has been provided by national institutions, in particular the institutions participating in the *Gaia* Multilateral Agreement.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Multidimensional indexes are an important part of modern databases [25, 14, 12]. 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.

Indexing multidimensional data is a challenging task due to the fundamental lack of a total data order. Data existing along a single dimension may be placed into a totally ordered state, for example integers from smallest to largest, and that order may be exploited by index structures such as the B-Tree [2]. However, data existing in multiple dimensions has no exact analog, since two multidimensional data points may have different orderings in different dimensions. The challenge for multidimensional indexes is therefore to efficiently impose or discover some ordering of the data that, while not total, is nonetheless exploitable for high performance, low latency querying of data.

Solutions to indexing multidimensional data [3, 29, 17, 5, 21, 6, 30] are based largely on the conceptual structure that originates from the R-Tree [13]. 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.

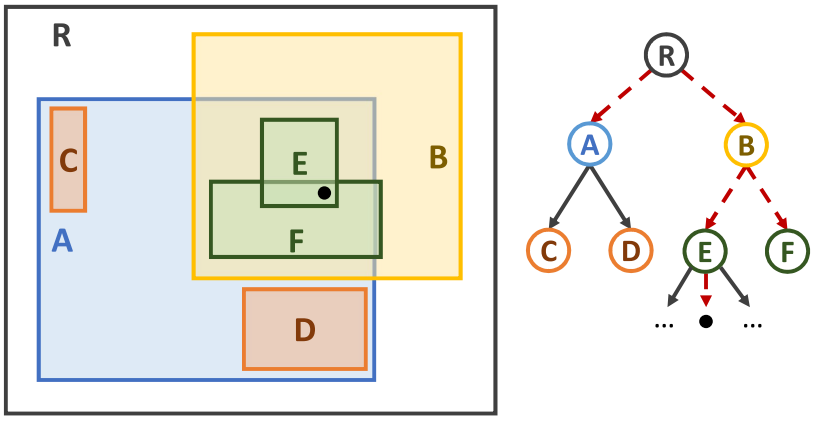

Figure 1.1: An R-Tree exhibiting scatter during search.

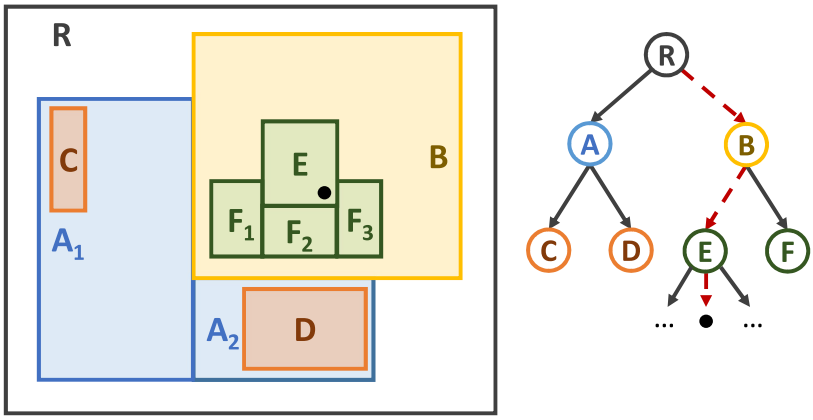

Figure 1.2: NIR-Tree exhibiting no scatter during search.

To illustrate, consider Figure 1.1 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 1.1, 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*.

A desirable, efficient multidimensional index would support search by minimizing in-

tersection between bounding rectangles. As a running example, consider the same point from Figure 1.1 indexed by the reconfigured geometry in Figure 1.2. Instead of the limited geometry of bounding rectangles, Figure 1.2 illustrates flexible sets of bounding rectangles which compose *bounding polygons*. 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 33% reduction in accessed nodes, again highlighted by a dashed red path in Figure 1.2, translates into a faster search.

Prior proposals improved ways to organize bounding rectangles [3, 29, 17] while other approaches [19, 16] 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 [19, 16] suffer from slow, complicated geometric tests to determine if a point is enclosed by a bounding object.

From a hardware perspective, current and prior work shows a clear trend toward fully memory resident multidimensional indexes; our proposed NIR-Tree is in this vein. Established multidimensional indexes now regularly use hybrid memory-disk layouts [5], while new works assume the index is entirely in memory [24, 22]. Moreover, entire databases in both the academic domain, such as Hyper [18], and in the commercial domain, such as VoltDB [10], SingleStore (formerly MemSQL) [15], and SAP Hana [28], are fully in-memory.

1.1 Thesis Contributions

In this thesis, we present the NIR-Tree, a new in-memory tree that adaptively replaces bounding rectangles with bounding polygons. The NIR-Tree *guarantees* zero-area intersection between bounding polygons to reduce the number of nodes accessed during queries compared to existing R-family indexes, thereby eliminating slowdowns in search caused by scattering. 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, the contributions of this thesis to multidimensional indexing are as follows:

1. The design and implementation of the NIR-Tree, a novel in-memory structure that creates axis-aligned bounding polygons from bounding rectangles.

2. Complete implementations of the R-Tree, R*-Tree, Quad-Tree, NIR-Tree and a modern implementation of the R^+-Tree.

3. A zero-area intersection guarantee among bounding polygons in the NIR-Tree, including an analysis and proof of this guarantee.

4. Creation of two easily accessible point datasets and accompanying query workloads representing both the road network of Canada and a section of the Milky Way as observed by the Gaia telescope.

5. Extensive evaluation on both real and synthetic datasets, demonstrating that the NIR-Tree is 27.8× faster to construct and 2.2× faster to point query than the R*-Tree, and 1.2× faster to construct and 3.1× faster to point query than the Revised R*-Tree.

1.2 Thesis Organization

The remaining chapters of this thesis are organized as follows:

1. Chapter 2 discusses the fundamentals of the R-Tree and how bounding polygons can eliminate intersection area.

2. Chapter 3 covers work with related approaches to the problem of multidimensional indexing.

3. Chapter 4 presents the design of the NIR-Tree, describing in detail the three basic data structure operations of insertion, searching, and deletion.

4. Chapter 5 analyzes the zero-area intersection guarantee, proving it holds before and after any of the three basic data structure operations.

5. Chapter 6 provides an experimental evaluation of the NIR-Tree compared with related prior approaches.

6. Chapter 7 concludes our work and provides future directions for building upon the NIR-Tree.

Without loss of generality, we reference and discuss the two-dimensional versions of trees throughout the thesis (with the exception of Chapter 5) to simplify the presentation.

Chapter 2

Background

In this chapter we describe the basic R-Tree and explain how and why scatter arises in multidimensional indexes based on the R-Tree [13].

2.1 The R-Tree

The R-Tree partitions data by recursively grouping data. To illustrate, consider Figure 2.1. Every R-Tree's lowest level is the collection of data points being indexed. Figure 2.1a shows the geometric structure of this lowest level alongside the logical structure in Figure 2.1b. Imposing structure on the raw data, the next level of every R-Tree groups data points together, approximating them by a bounding rectangle. Each bounding rectangle entirely encloses all the points in the group it represents. Notice that in Figure 2.1c there are no points outside the bounding rectangles A, B, C. The R-Tree represents each geometric grouping as exactly one logical node. Logical nodes and their associated bounding rectangles are coloured and labeled the same in Figure 2.1c (geometric) and Figure 2.1d (logical). The top level of the example tree depicted in Figure 2.1 groups bounding rectangles A, B, and C together in a larger bounding rectangle D (Figure 2.1f). Thus, the logical children of node D are A, B, C as shown in Figure 2.1e.

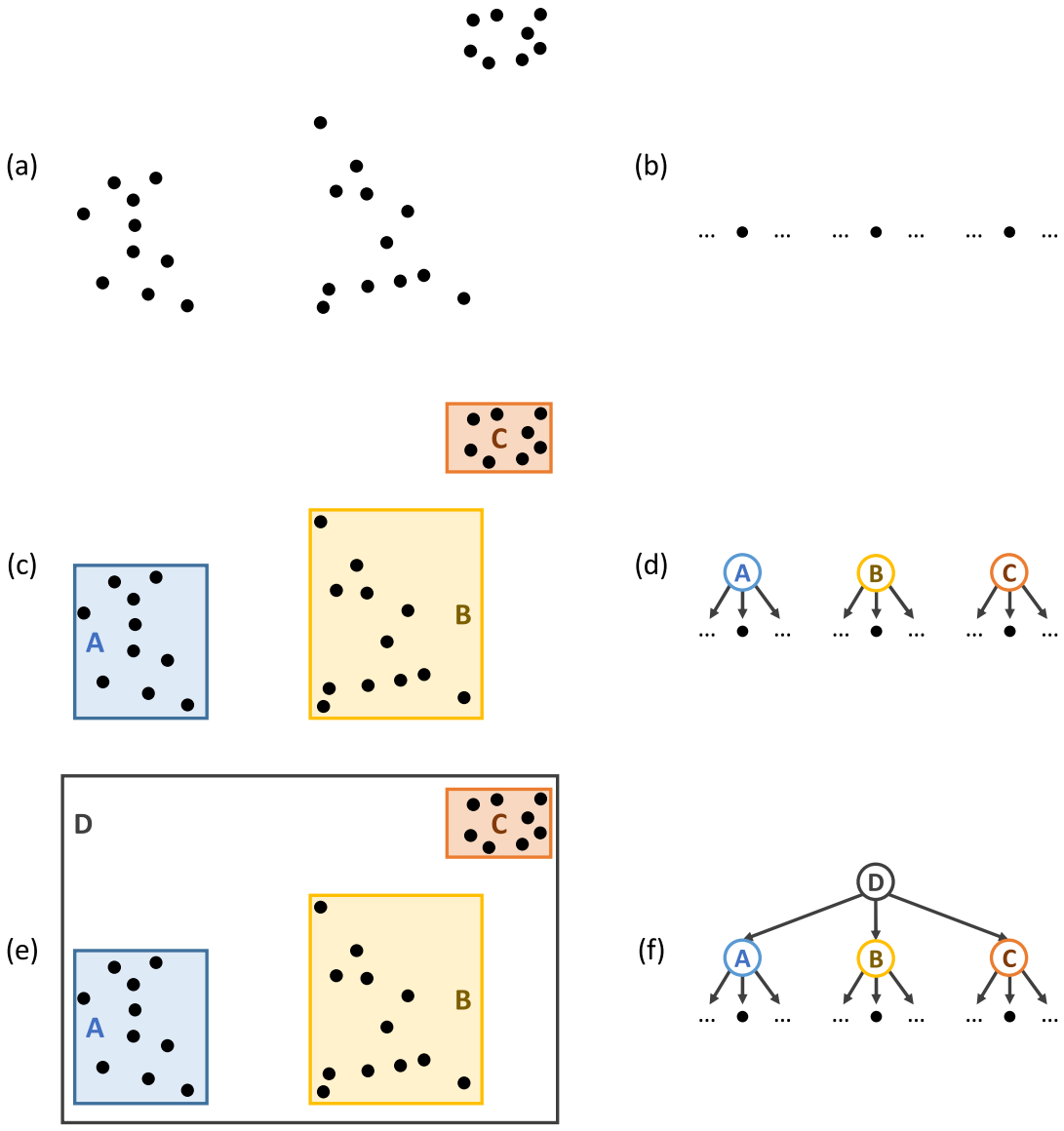

Figure 2.1: Recursive grouping forming an R-Tree. Geometric formation is shown in (a), (c), and (e) while corresponding logical formation is shown in (b), (d), and (f).

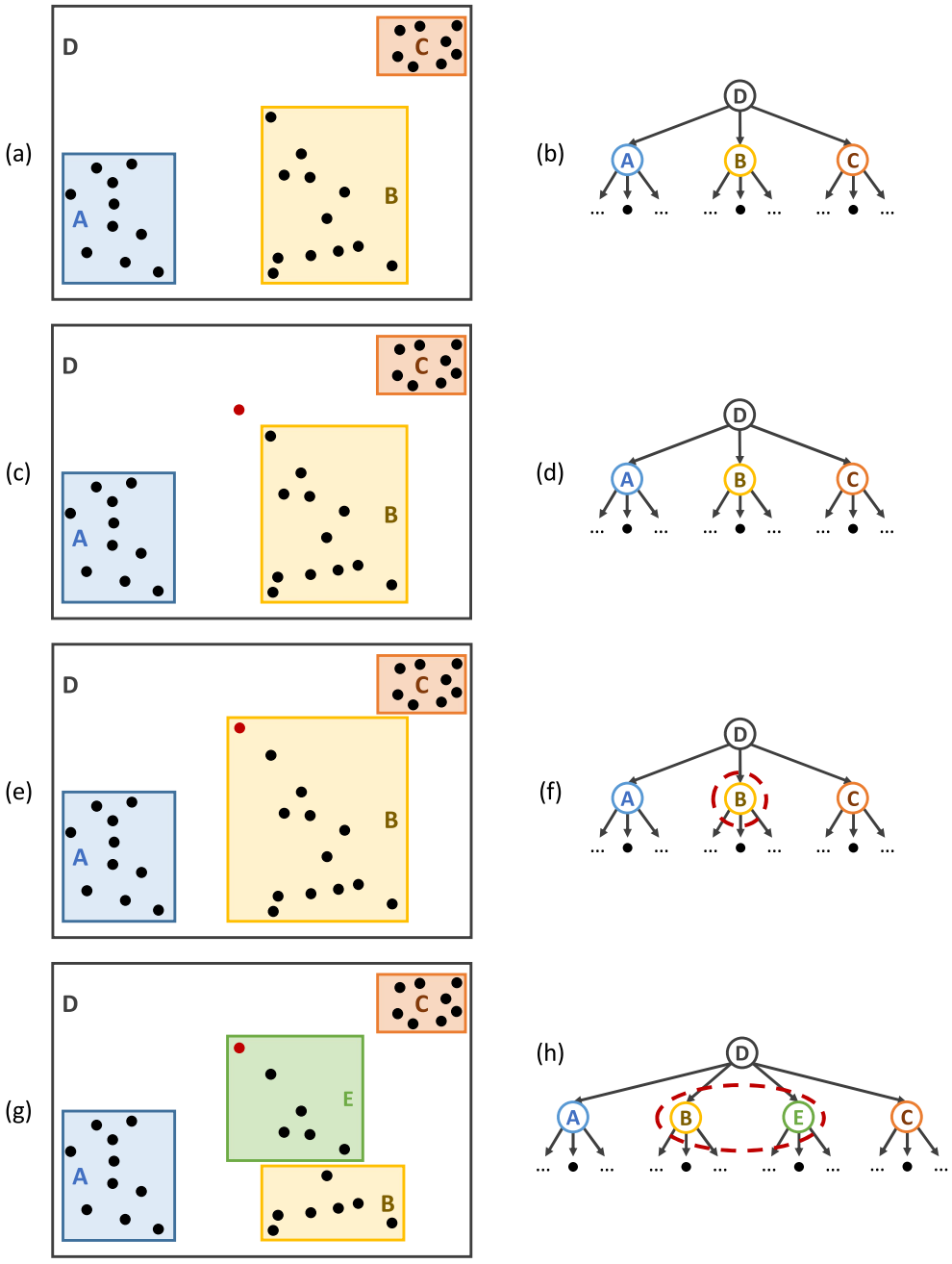

Figure 2.2: Splitting a node in an R-Tree. Geometric alterations to the tree are shown in (a), (c), (e), and (g) while corresponding logical alterations to the tree are shown in (b), (d), (f), and (h).

R-Trees are a dynamic data structure and we will describe the insertion process next using Figure 2.2. The R-Tree of Figure 2.2 begins as the same R-Tree just discussed in Figure 2.1. Suppose that a new red point is to be inserted (Figure 2.2c). The process of choosing a leaf into which the new point will be placed begins at the root of the tree, in this case D (Figure 2.2c). Since the new point is contained in a bounding rectangle at this level, namely D, the children of D are considered next. When a new point is not contained in an existing rectangle, as is the case for the red point among nodes A, B, C, a bounding rectangle is *expanded* to enclose the new point. The *selection* of a bounding rectangle is based on a cost function called a *metric*. Bounding rectangle B is selected to expand in Figure 2.2e, enclosing the new point. However, R-Tree nodes cannot contain an arbitrarily large number of logical children. Upon construction, a maximum fanout for each node is set, nodes may not have more children than the maximum fanout. To address a node that is *overflowing* with children, the R-Tree splits the overflowing logical node into two new nodes positioned at the same height of the tree and having the same parent as the overflowing node. Figure 2.2f and Figure 2.2h show B as an overflowing node before and after the application of a split. The logical and geometric split are performed at the same time. Splits place children into two groups (nodes E and B in Figure 2.2h), deciding which group a child belongs to by computing a cost metric involving the child and each group's bounding rectangle (rectangles E and B in Figure 2.2g). Each group is then made a child of the overflowing node's parent. If by adding nodes to the parent it would overflow, then the splitting process is recursively applied to the parent until the root is reached. The R-Tree grows taller by one level if the root itself overflows and must be split.

2.2 Scatter

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. Bounding rectangles within the R-Tree are selected to minimize the amount of additional area required to enclose a new point. Figure 2.3a depicts the specifics of the R-Tree's metric. 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 2.3b, yet intersection may still occur. For another example, the R*-Tree [3] and Revised R*-Tree both consider the perimeter of the expanded

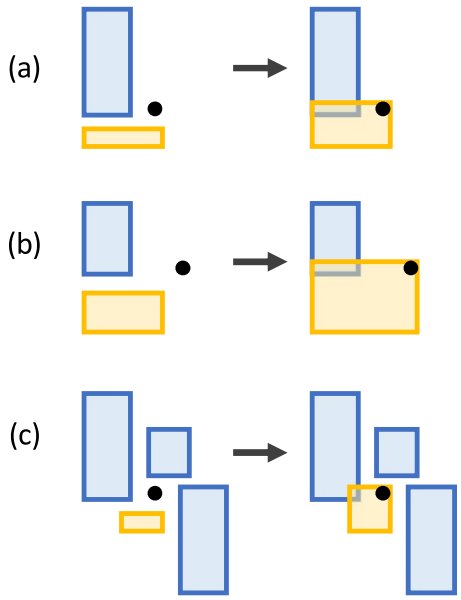

Figure 2.3: Expansion causing intersection in three examples. (a) Expansion using change in area. (b) Expansion using distance. (c) Expansion using intersection area.

bounding rectangle (Figure 2.3a) and additional intersection area (Figure 2.3c). However, all of these alternatives may cause intersection. The yellow rectangle in Figure 2.3 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.*

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 1.2. By forming bounding polygons from bounding rectangles during `insert`, the NIR-Tree achieves zero-area intersections between bounding polygons (Figure 1.2) where the R-Tree fails to do so. Zero-area intersection in turn produces point queries in the NIR-Tree which execute $2 - 4\times$ faster than in any other state-of-the-art R-family tree.

Chapter 3

Related Work

The field of multidimensional indexing is well studied with proposals for multidimensional data management [27]. In this chapter, we contrast the NIR-Tree with prior approaches to multidimensional indexing. We divide these approaches into two categories: data partitioning and space partitioning.

3.1 Data Partitioning

Data partitioning indexes cluster data and typically 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 [12] and MySQL [25]. The NIR-Tree is closely related to the works in this class of partitioning.

The R-Tree [13] is a multidimensional extension of the B-Tree [2]. Rectangular data objects are recursively grouped into rectangular regions to create a balanced tree. When the root has too many children, exceeding some set maximum, the overflow is addressed by building a new root on top and breaking the original into two nodes. In opposite fashion, when the root has too few children and falls below some set minimum, the underflow is addressed by replacing the root of the tree with a child node. Unlike one dimensional B-Tree ranges, R-Tree regions may intersect because their *volumetric* data objects take up space and may intersect. R-Tree performance degrades as the number of dimensions increases because the volumetric data objects become more likely to intersect in some dimension. The effect is known as the *curse of dimensionality*.

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 have many children, thus having high *utilization* of the maximum fanout.

The Revised R*-Tree [5] matches the performance of the R*-Tree without using reinsertions 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 Revised R*-Tree seeks to balance splits logically while also balancing the splits geometrically.

Optimizing for intersection area, the NIR-Tree and R^+-Tree [29] 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.

The Hilbert R-Tree [17] 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 2.3), the Hilbert R-Tree selects bounding rectangles whose centroids have a Hilbert number closest to the point of interest's Hilbert number. The result is a clustering effect, because Hilbert numbers exhibit an order that approximates spatial nearness. The focus on clustering excludes consideration of intersection area from Hilbert R-Trees, and they may have sub-optimal intersection area.

Some trees discard axis-alignment constraints by using arbitrarily oriented bounding rectangles. SICC indexes [31] 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.

P-Trees [16] illustrate arbitrarily-shaped bounding objects. Since a rectangle is defined by a maximum and minimum value in each of two dimensions, additional pairs along new axes can be added to create more complex shapes. P-Trees 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.

Similarly, k-discrete oriented polygon (k-DOP) hierarchies [19] use additional axes but fix the number of axes to be k in a d dimensional space such that $k >= d$. For example, such over-description of the space allows 4-DOP's to form octagons in two dimensional space. However, the complexity of shapes is limited since k is fixed in advance. Complex polygons achieve low coverage and intersection area, but quickly become computationally unwieldy for dimensions greater than three. k-DOP hierarchies are therefore generally used for collision detection in low dimensional spaces.

Recent work on clipping bounding rectangles [30] 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.

To address the curse of dimensionality, trees like the TV-Tree [21] or X-Tree [6] specifically target high-dimensional data and the degradation it causes in R family trees. The TV-Tree dynamically "telescopes" the dimensions under consideration to artificially reduce dimensionality. Reduction is possible since often only a small number of dimensions are discriminant. The X-Tree outperforms the TV-Tree by simultaneously allowing overfull nodes to exist and choosing splits in which the resulting bounding rectangles cause the least overlap.

Unlike an R-Tree, the NIR-Tree begins by considering points rather than volumetric objects, exploiting the infinitesimal nature of points to eliminate positive area intersection. As we demonstrate, the area of intersection in the NIR-Tree can be pushed to zero, an area smaller than any other data partitioning index can attain. To achieve zero-area intersection, the NIR-Tree uses rectangle fragmentation and partition lines to create bounding polygons based on siblings. Partition lines balance leaf splits in the NIR-Tree logically and geometrically while totally eliminating intersection during node splits. Left unchecked, rectangle fragmentation and partition lines may cause NIR-Tree polygons to contain redundant rectangles. To keep polygons small in size (Table 6.1), NIR-Tree polygons are constantly culled during insertion (Algorithm 4). Small dynamic bounding polygons allow the NIR-Tree to gracefully accommodate an arbitrary amount of additional insertions, which some space partitioning indexes cannot guarantee, as discussed in the next section.

3.2 Space Partitioning

An alternative multidimensional indexing approach partitions space rather than data to implicitly form groups. Grid files [23] are a popular space partitioning option [25, 14, 12]. 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.

Inspired by grid files and seeking to bring multidimensional indexes into main memory, BLOCK [24] defines a number, L, of uniform static grids at multiple resolution levels. Intersection tests dominate range search times in memory and multi-resolution grids drastically reduce the number of intersection tests required. However, selection of the L appropriate grids is only possible because BLOCK assumes the indexed data is static.

Recent approaches build on the grid file by replacing the mapping directory with machine learning models. In addition to replacing the mapping directory, Flood [22] uses its models to optimize attributes of the grid such as resolution. Although Flood is not static like BLOCK, it requires that its models be retrained and its data rearranged when a data distribution shift occurs. Too many additional insertions of points not predicted by Flood's models require a complete rebuild of the index.

LISA [20] 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 [24], is higher than that of the R*-Tree's, which is equivalent to that of the NIR-Tree's.

The *curse of dimensionality* is not exclusive to data partitioning approaches. Space partitioning indexes also suffer from, and address, the curse of dimensionality. The VA-File [32] is a grid file with a novel heuristic that determines cells based on **V**ector **A**pproximations of high-dimensional data points. While not fully static, VA-Files can only tolerate minor shifts in data distribution before reconstruction is required.

Breaking from the grid file mold, Quad-Trees [11] 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. Quad-Tree fanout, as well as height, scales poorly due to its 2^d fanout requirement. Conversely, the NIR-Tree can maintain short trees

Table 3.1: Categorization of Multidimensional Indexes On Core Features.

Structure	Data or Space Partitioning	Memory or Disk Based	Bounding Object	Dynamic Bounding Objects	Bounding Objects May Intersect
R-Tree [13]	Data	Disk	Rectangle	No	Yes
Hilbert R-Tree [17]	Data	Disk	Rectangle	No	Yes
R^+-Tree [29]	Data	Disk	Rectangle	No	Yes
R*-Tree [3]	Data	Disk	Rectangle	No	Yes
Revised R*-Tree [5]	Data	Disk	Rectangle	No	Yes
CBB [30]	Data	Disk	Axis-aligned Polygon	Yes	Yes
SICC Index [31]	Data	Disk	Axis-unaligned Rectangle	No	Yes
P-Tree [16]	Data	Disk	Min and max values for each axis	No	Yes
k-DOP Tree [19]	Data	Memory	Discrete Oriented Polygon	No	Yes
TV-Tree [21]	Data	Disk	Rectangle	No	Yes
X-Tree [6]	Data	Disk	Rectangle	No	Yes
PR-Tree [1]	Data	Disk	Rectangles	No	Yes
BLOCK [24]	Space	Memory	Grid	No	No
LISA [20]	Space	Disk	Grid	No	No
FLOOD [22]	Space	Memory	Grid	No	No
VA-File [32]	Space	Disk	Grid	No	No
Quad-Tree [11]	Space	Memory	Quadrants	No	No
NIR-Tree	Data	Memory	Axis-aligned Polygon	Yes	No

with fixed fanout independent of the dimension. Finally, while Quad-Trees do not require reconstruction upon insertion, deletes require reconstruction of the entire subtree rooted at the deleted element.

3.3 Index Structure Categorization

To highlight the differences between the NIR-Tree and the related work discussed above, every multidimensional indexing approach was categorized along five dimensions. These five dimensions are i) indexing partitioning style: either data or space, ii) index memory model: either disk or memory based, iii) index bounding object type, iv) whether the index's bounding object representations are dynamic or static, and v) the index's bounding objects ability to prevent intersection with one another.

First, Table 3.1 divides indexes into two groups wherein the index either creates groups implicitly by subdivisions of the space, or makes groups explicitly by choosing particular data for particular groups. These two styles are known as space partitioning and data partitioning respectively. Our categorization includes indexes of both partitioning types. The NIR-Tree is a data partitioning index and therefore most closely related to other data partitioning indexes. However, as can be seen in the fifth category, the NIR-Tree brings the non-intersection of space partitioning bounding objects into a data partitioning index style.

Second, research into multidimensional indexes spans multiple eras of computer memory development. With main memory sizes increasing in modern systems, some multidimensional indexes like the BLOCK [24] index, and the k-DOP Tree [19], assume data will

fit entirely into main memory. While the NIR-Tree is listed as an in-memory index, and our experiments are performed in-memory (Chapter 6), the NIR-Tree's balanced tree-like structure can be extended to operate on disk pages in a direct analog to previous R-family trees' disk operations.

Third, while bounding rectangles are the most popular bounding object, data partitioning multidimensional indexes do contain a variety of bounding objects. Space partitioning indexes, on the other hand, uniformly use regular rectangular partitions of space. The data partitioning SICC index [31] moves away from the rectangle paradigm slightly by permitting rotated bounding rectangles. The Clipped Bounding Box technique [30] and the NIR-Tree both use bounding rectangles in a more flexible way, constructing polygons out of rectangles. Finally, multidimensional indexes like the P-Tree [16] or the k-DOP Tree [19] add additional axes to the space to build irregular polygons. As seen in our classification, multidimensional indexes with more complex bounding objects are rarer, since there exists a trade off between precise representations of the underlying data and the cost of computing intersection or containment with those representations.

Fourth, the ability of bounding objects to change the degree to which they represent the underlying data is categorized. The overwhelming majority of multidimensional indexes select a single bounding object to approximate all data groupings, using it for the entire lifetime of the index. If data could be better approximated by a more complex bounding object, the multidimensional indexes without dynamic bounding objects cannot take advantage of this fact. Instead, only the NIR-Tree, TV-Tree [21], Clipped Bounding Box technique [30], and the P-Tree [16] can dynamically adjust the quality of approximation given by their bounding objects. The TV-Tree adjusts quality of approximation by adjusting the number of dimensions used by a given bounding object. For example, if all data points in a group are equal in the fourth dimension, then the TV-Tree will create a three dimensional bounding object and not consider the fourth dimension. The P-Tree acts like a fully dynamic version of the k-DOP Tree. While k-DOP's must have k specified in advance, the P-Tree creates as many additional axes as it needs to approximate the data to some desired level. Lastly, the NIR-Tree and Clipped Bounding Box technique increase the quality of their approximations by removing regions of bounding rectangles whenever they are causing intersection area or are empty, respectively.

Fifth and finally, we categorize multidimensional indexes based on the amount of intersection area allowed between bounding objects. By definition space partitioning multidimensional indexes do not have positive intersection area between their bounding objects. Exactly opposite to space partitioning indexes, data partitioning indexes all allow intersection area, with the sole exception of the NIR-Tree. This is the advantage the NIR-Tree brings to data partitioning multidimensional indexes. Without positive intersection area,

the benefits of no-scatter searching (Chapter 1) are available to data partitioning indexes through the NIR-Tree's techniques.

Chapter 4

The NIR-Tree

In this chapter, 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 point and range `search` operations are detailed, and the chapter concludes with an explanation of the deletion operation `remove`. A visualization of the NIR-Tree, along with visualizations of other trees for comparison, appear in Appendix B.

4.1 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* $\langle parent, \{p_0, \ldots, p_n\}\rangle$

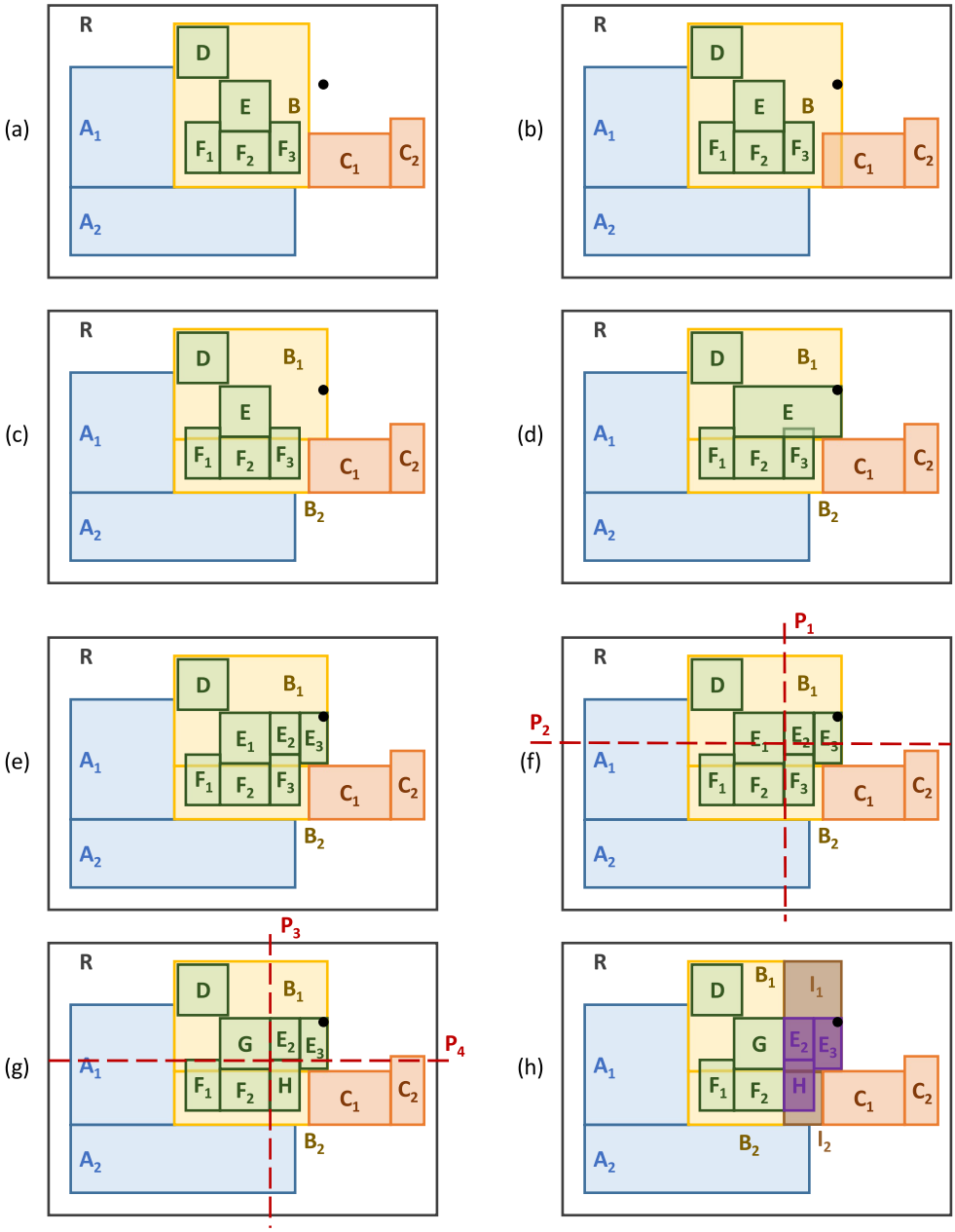

Figure 4.1: Insertion is shown with polygon expansion, fragmentation, refinement, and splitting. Maximum Fanout = 3.

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 $N.branches$ and $N.data$ to denote the set of tuples and the set of data points within node N respectively. Additionally, 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.

4.2 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 4.1 as a running example for the steps executed during `insert`, described next.

4.2.1 Insert Downward Sweep

Insertion starts with the NIR-Tree illustrated in Figure 4.1a. 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 2.3b). 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) $\rightarrow L$

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: $B.\mathcal{P} = \text{fragment}(B.\mathcal{P}, B'.\mathcal{P})$
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

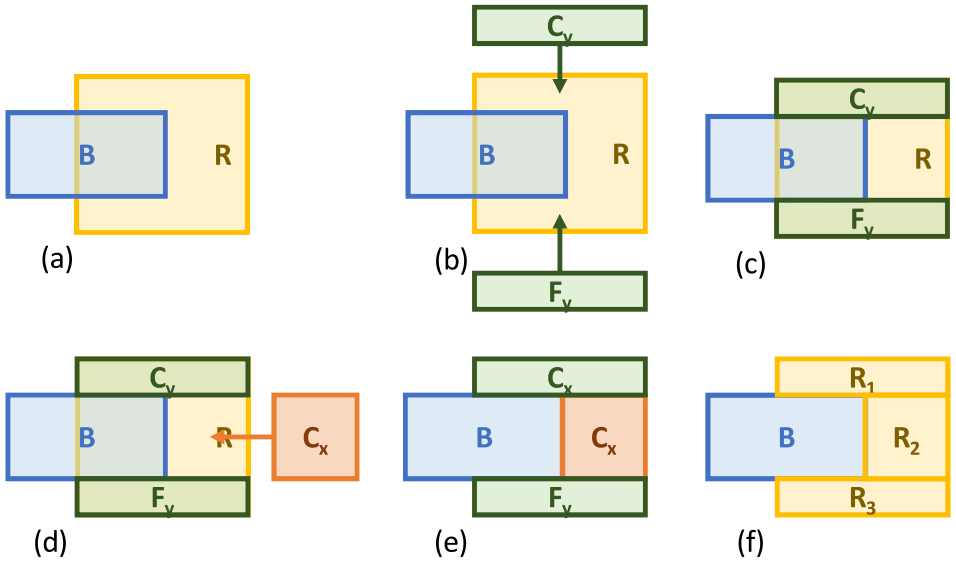

Figure 4.2: Polygon fragmentation in two dimensions. The original state of B (a) is fragmented in the y dimension (b), (c) and is fragmented in the x dimension (d), (e), resulting in a new polygon composed of fragments (f).

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 4.1a 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 4.1b, 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 itself. We start by discussing fragmentation generally using Figure 4.2 before applying the process to our running example in Figure 4.1c.

Since the fragmentation process must generalize to any number of dimensions, one dimension is processed at a time. In Figure 4.2a, 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 4.2b and 4.2c). 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 4.2d and 4.2e. 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 4.2f. 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 4.1c), 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 4.1c 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 4.3) 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 4.3a), the line rectangle is removed. Second, when a rectangle is enclosed by another rectangle (Figure 4.3b), 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 4.3c), the row or column is replaced with a single rectangle. In Figure 4.1c, B_1 and B_2 do not exhibit any of the three patterns so B is left unaltered.

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 4.1 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 4.1e 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`.

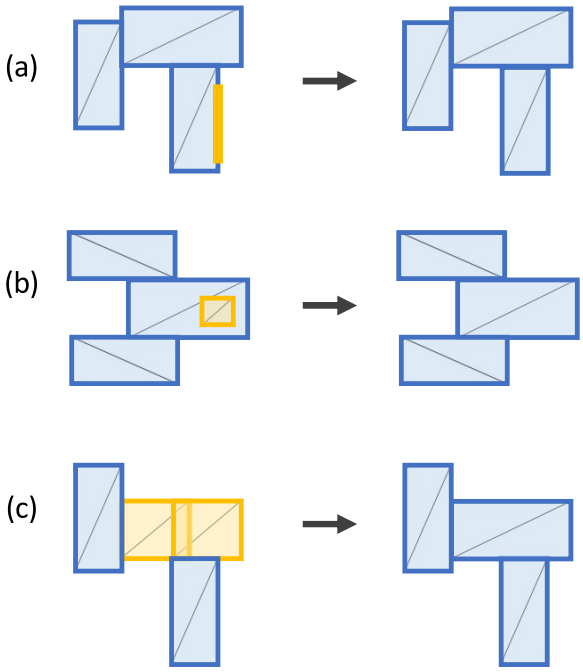

Figure 4.3: Refinement reducing polygon size in rectangles. (a) Redundant rectangles on the perimeter. (b) Redundant rectangles within other rectangles. (c) A mergeable row of rectangles.

Algorithm 2 adjustTree(N)

Require: N is a leaf, $\exists m$ a maximum fanout

while N not root **do**
  if $|N.branches| > m$ or $|N.data| > m$ **then**
    $\langle N_L, \mathcal{P}_L \rangle, \langle N_R, \mathcal{P}_R \rangle$ = splitNode(N, partitionNode(N))
    Add $\langle N_L, \mathcal{P}_L \rangle$ to $N.parent.branches$
    Add $\langle N_R, \mathcal{P}_R \rangle$ to $N.parent.branches$
    Remove $\langle N, \mathcal{P} \rangle$ from $N.parent.branches$
  end if
  $N = N.parent$
end while

Algorithm 3 getReferencePolygon(N) $\rightarrow \mathcal{P}_{ref}$

Require: N is a node

if N not root **then**
  $\mathcal{P}_{ref} = N.parent.poly$
else if N is leaf **then**
  $\mathcal{P}_{ref} = \{R\}$ where R is the smallest rectangle enclosing all points in $N.data$
else
  $\mathcal{P}_{ref} = \{R\}$ where R is the smallest rectangle enclosing all polygons in $N.branches$
end if
return $\mathcal{P}_{ref}$

4.2.2 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 4.1d. 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 5). 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 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 4.1f 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 4) divides the current node along it (line 2). In Figure 4.1g, E_1 lies entirely to the 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 7), 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 4.1f) 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 24). The current partition line is passed to the recursive call so F is split along the same line as its parent, $P3$ in Figure 4.1g. 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 4.1h) lie entirely on opposite sides of $P3$ and are placed into B, and B's new sibling I, respectively.

Algorithm 4 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

$\mathcal{P}_{ref}$ = getReferencePolygon(N)
$\mathcal{P}_L, \mathcal{P}_R$ = left and right side of $\mathcal{P}_{ref}$ sliced along l in dimension D
$N_L = N_R = \langle N.parent, \emptyset\rangle$
Branch $left = \langle N_L, \mathcal{P}_L\rangle$
Branch $right = \langle N_R, \mathcal{P}_R\rangle$
for $p \in N.data$ **do**
  if $p \in \mathcal{P}_L$ and $p \in \mathcal{P}_R$ **then**
    $N_{tie} = N$ with smaller of $|N_L.data|$ and $|N_R.data|$
    $N_{tie} = \{p\} \cup N_{tie}.data$
  else if $p \in \mathcal{P}_L$ **then**
    $N_L.data = \{p\} \cup N_L.data$
  else if $p \in \mathcal{P}_R$ **then**
    $N_R.data = \{p\} \cup N_R.data$
  end if
end for
for $B \in N.branches$ **do**
  if $B.\mathcal{P} \cap \mathcal{P}_L = \emptyset$ **then**
    $N_R.branches = \langle B.child, B.\mathcal{P}\rangle \cup N_R.branches$
    $B.child.parent = N_R$
  else if $B.\mathcal{P} \cap \mathcal{P}_R = \emptyset$ **then**
    $N_L.branches = \langle B.child, B.\mathcal{P}\rangle \cup N_L.branches$
    $B.child.parent = N_L$
  else
    Branches b_L, b_R = splitNode($B.child$, l, D)
    $N_L.branches = \langle b_L.child, b_L.\mathcal{P}\rangle \cup N_L.branches$
    $b_L.child.parent = N_L$
    $N_R.branches = \langle b_R.child, b_R.\mathcal{P}\rangle \cup N_R.branches$
    $b_R.child.parent = N_R$
  end if
end for
refine($\mathcal{P}_L$)
refine($\mathcal{P}_R$)
return b_L, b_R

Algorithm 5 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'

An identical process follows to split the root.

4.3 Searching

Finally, we describe the NIR-Tree `search` (Algorithm 7) 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 6 search(N, p) $\rightarrow D$

Require: N is a node, p is a point, D the data associated with p

1: **for** $B \in N.branches$ **do**
2: **if** $p \in B.\mathcal{P}$ **then**
3: $D =$ search($B.child$, p)
4: **end if**
5: **end for**
6: **for** $p' \in N.data$ **do**
7: **if** $p = p'$ **then**
8: $D =$ data associated with p'.
9: **end if**
10: **end for**
11: **return** D

Algorithm 7 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

4.4 Deletion

Generally, the deletion operation is not a focus of multidimensional indexes. Some indexes such as the Quad-Tree [11] do not support deletions at all, others such as the RR*-Tree [5] do not apply their techniques to deletion. However, for the sake of completeness, deletion is implemented in the NIR-Tree and described briefly here.

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 previous section (Section 4.3). 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.

Chapter 5

NIR-Tree Analysis

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

5.1 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 5.1a 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 5.1b 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 $R.ll \leq p \leq R.ur$.*

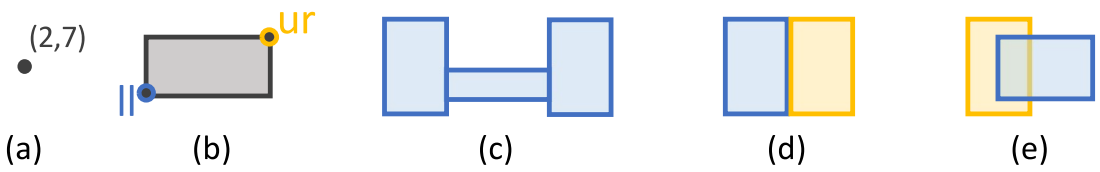

Figure 5.1: Definitional examples. (a) Point. (b) Rectangle. (c) Polygon. (d) Disjoint rectangles. (e) Intersecting rectangles.

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 5.1c).

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 $p \in R$.*

5.2 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 results 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 5.1d and 5.1e 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 the `insert` algorithm (Algorithm 1). Described in Section 4.2 and illustrated in Figure 4.2, 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 $ur \to ll$ in (2).*
4. $F_i.ur$ *obtained similarly by replacing $ll \to ur$ 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,*

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

5.3 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 4 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 4 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 4, 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 4 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 4 line 2, $\forall R_L \in \mathcal{P}_L$, $R_L.ll_d \leq l$ 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 4 lines 6 — 15. Observe that $N_L.data = \emptyset$ by construction at line 3. In conjunction with the definition at line 3, lines 10 and 12 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 16 — 30. If B_L was placed in N_L at line 21 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 35 implies $B_L.\mathcal{P}$ was sliced during the recursive call at line 24 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 4 for all nodes N.

Second, let us consider $IV(2)$. Again by construction of $\mathcal{P}_L$ and $\mathcal{P}_R$ at Algorithm 4 line 2, 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 24 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 4 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. □

Chapter 6

Performance Evaluation

In this chapter, we experimentally demonstrate the performance advantage of the NIR-Tree over the R-Tree [13], Quad-Tree [11], R^+-Tree [29], R*-Tree [3], and Revised R*-Tree [5] in terms of search and insertion efficiency using both real and synthetic datasets.

6.1 Experimental Setup

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

6.1.1 Software

We compared the NIR-Tree against our implementations of the R-Tree [13], Quad-Tree [11], R^+-Tree [29], R*-Tree [3], and Revised R*-Tree [5]. The Quad-Tree was selected for comparison for two reasons. First the Quad-Tree, like the NIR-Tree, is a multidimensional index designed for memory and structured as a tree. Secondly, the Quad-Tree serves as a reference point to space partitioning style multidimensional indexes. Next, the R^+-Tree was selected for comparison because a split mechanism similar to its partition line and recursive downward split is present in the NIR-Tree. The R*-Tree was selected for comparison due to its popularity and widely accepted baseline status. The Revised R*-Tree is a less popular version of the R*-Tree achieving similar query performance with lower insertion times, and was therefore selected as the state-of-the-art competitor. All trees resided entirely in main memory during experiments. We set the maximum fanout m of all trees so as to provide

the best point and range query performance. For the R-Tree, R^+-Tree, and R*-Tree, a maximum fanout of 100 and minimum fanout of 50 produced the best query times. For the Revised R*-Tree, the authors' suggested maximum fanout of 100 and minimum fanout of 20 produced the best query times. For the NIR-Tree, a maximum fanout of 50 resulted in the best query times by keeping bounding polygons small. Finally, the Quad-Tree's fanout at each node was set by definition to 2^d.

6.1.2 Hardware

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

6.1.3 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, 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.

6.2 Datasets and Queries

We describe next the six datasets and associated queries used to evaluate the trees. Extended descriptions of the datasets, including visualizations, are found in Appendix A.

6.2.1 Datasets

Each of the six indexes 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 [8]. 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 [7, 9]. 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.

6.2.2 Queries

All datasets were queried using rectangles that contained about 1000 points. Query rectangles for California, Biological, and Forest data are specified in advance by the benchmark [4, 5]. Query rectangles for Uniform data 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.

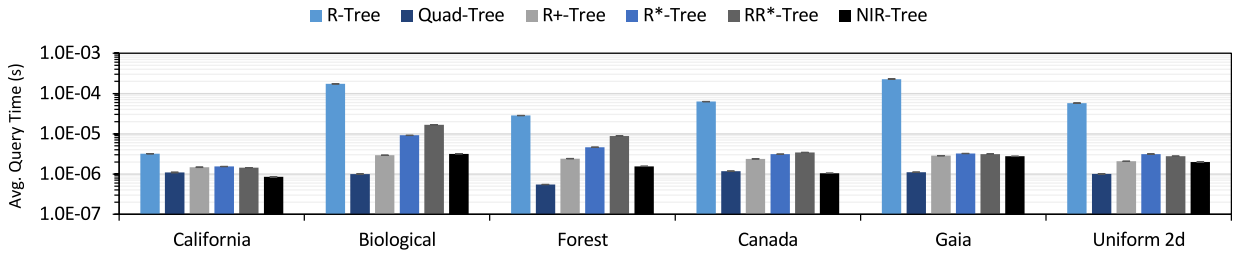

Figure 6.1: Point query times (log scale).

6.3 Results

We present and analyze the point query and range query times as well as insertion times for the NIR-Tree and its competitors.

6.3.1 Point Query Performance

Recall from Chapter 1 that the NIR-Tree's scatter reduction is highly efficient. On the California, Biological, and Gaia datasets, the NIR-Tree executed 99.8% of queries optimally with no scatter. On the Forest dataset, the NIR-Tree provided optimal point query execution for 98.6% of queries, and on the Uniform and Canada datasets the NIR-Tree executed optimally 99.9% of the point queries. The rare cases where the NIR-Tree did not provide no-scatter searches were when a point lay on the perimeter of two bounding polygons. No point query in the NIR-Tree ever needed to access more than two extra nodes.

Efficient point queries allow the NIR-Tree to out-compete every R-family tree on every dataset with the sole exception of the R$^+$-Tree on the Biological dataset. Moreover, the NIR-Tree out-competes the space partitioning Quad-Tree on the California and Canada Roads datasets. The popular R*-Tree 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. Lastly, the Revised R*-Tree is outperformed by the NIR-Tree by an average factor of 3.1×.

The NIR-Tree saw its closest competition among R-family trees from the R$^+$-Tree instead of the R*-Tree or Revised 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 so as to keep bounding polygons simple. In general, larger fanouts in the NIR-Tree require more complex bounding polygons because they must remain disjoint, yet enclose more children. In aggregate, the R$^+$-Tree was slightly faster than the NIR-Tree on the 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.

Between the data partitioning NIR-Tree and space partitioning Quad-Tree, the NIR-Tree outperformed the Quad-Tree on California and Canada Roads by an average factor of 1.2×. By the Quad-Tree's nature, point queries exhibit no scatter and only d less-than comparisons must be made at each node. Point queries are therefore slowed only by tall, unbalanced trees created on skewed datasets. When indexing in the presence of mild skew in California and Canada Roads, the NIR-Tree's advantage in tree balance overcame the cost of point in polygon tests, retrieving points faster than the Quad-Tree. For example, the Quad-Tree accessed an average of 64.4 nodes during each point query on the Canada Roads dataset. This is 13× more nodes per query than the NIR-Tree accessed. By contrast, the Uniform dataset created a more balanced Quad-Tree due to its distribution. On the Uniform dataset the Quad-Tree accessed an average of 16.1 nodes during point queries, which by comparison is only 3.2× the number of nodes the NIR-Tree accessed during point queries. As a result, Quad-Tree point queries consumed less time than NIR-Tree point queries. On all datasets the Quad-Tree achieved good point query times at the cost of range query performance (Section 6.3.3) and intensive memory usage (Section 6.3.4).

The NIR-Tree's competitors were ordered in the same fashion across all datasets: the Quad-Tree was closest, followed by the R$^+$-Tree, then the R*-Tree or Revised R*-Tree, then the R-Tree. The differences among the indexes' point search times decreased to their lowest point on the California dataset.

6.3.2 Update Performance

Existing multidimensional indexes trade off insertion time for query performance [3, 20, 22, 1, 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 and Revised R*-Tree while having better point query performance (Section 6.3.1), and equivalent range query performance (Section 6.3.3). For insertions, the R*-Tree consumes, on average, a substantial 27.8× the insertion time of the NIR-Tree, and the Revised

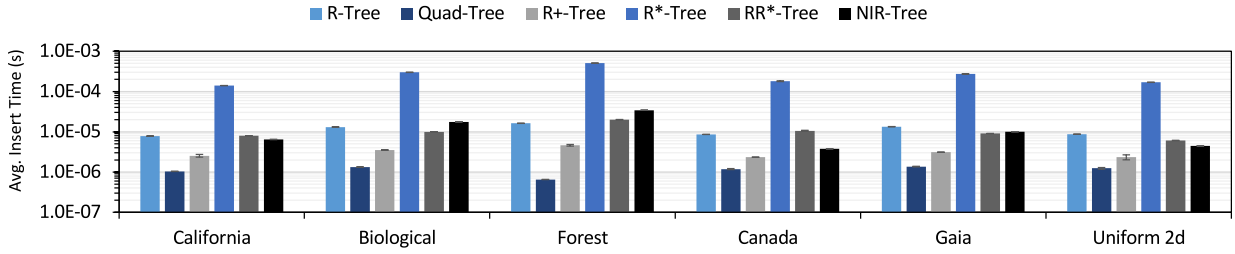

Figure 6.2: Insertion times (log scale).

R*-Tree consumes, on average, 1.2× the insertion time of the NIR-Tree. The Quad-Tree trades off insertion time for query performance in the opposite direction, providing the lowest insertion times and the highest range query times of all trees.

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 6.3.4) 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 6.2. Other trees use neither re-insertions nor bounding polygons, thereby trading-off insertion time for query performance.

Point queries in the Revised R*-Tree perform uniformly slower than in the NIR-Tree. Significantly, NIR-Tree insertion performance is not generally traded-off to achieve this result. On the Biological and Forest datasets the NIR-Tree builds more complex polygons in the presence of both high skew and high dimensionality, thus requiring more time to insert on average. The 1.7× slowdown present on these two datasets is compensated for by a 5.4× speedup of point queries and a 1.2× speedup of range queries over the Revised R*-Tree. More importantly, the majority of datasets show the NIR-Tree on average both inserting points 1.6× faster, and querying for points 1.9× faster than the Revised R*-Tree.

Among all trees, the R*-Tree requires the most amount of time to insert, followed by the R-Tree or Revised R*-Tree, followed by the NIR-Tree, then the R$^+$-Tree, and then the Quad-Tree. This ordering is prevalent on all datasets except Biological and Forest where the NIR-Tree and Revised R*-Tree swap places. As described above, the NIR-Tree is much faster than the R*-Tree due to quadratic operations present within R*-Tree re-insertions. Next, the R-Tree, using an $O(n^2)$ split algorithm but no re-insertions [13], still under-performs the NIR-Tree whenever the average size of a bounding polygon is below 3.

The Revised R*-Tree consumes a similar amount of time to insert as the R-Tree, having a worst case runtime of $O(n^2)$ [3], but improved average case performance as seen in our experiments. The NIR-Tree's superior insertion performance is reflected by its average insertion time speedup of 1.3× over the R-Tree and 1.2× over the Revised R*-Tree.

The R$^+$-Tree is 81× faster than the R*-Tree and faster than the NIR-Tree for two reasons. First, the R$^+$-Tree's split algorithm runs in $O(dn\log(n))$ time [29], and the 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 4). 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 6.3.3) 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, R$^+$-Tree striping results in range query performance least desirable among the R-family trees tested.

The Quad-Tree is 271× faster than the R*-Tree and faster than all the trees for the simple reason that it does not have to compute cost metrics or split nodes during insertion. Insertions in a Quad-Tree are equivalent to a search plus one memory allocation for a new tree node that is always placed in an empty fanout slot at the bottom of the tree. Dissimilarly, R-family trees perform extra computations at insertion time to maintain the balanced property necessary for efficient range queries. Accordingly, the Quad-Tree's straightforward insertion strategy impairs range query performance to the extent that range queries on the Quad-Tree are slowest among all trees tested (Section 6.3.3).

Overall, among trees providing the best range query performance, no tree was faster to construct than the NIR-Tree.

6.3.3 Range Query Performance

Across different datasets and dimensions, range queries within the NIR-Tree performed well (Figure 6.3). On all datasets, the R*-Tree and Revised 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. Similarly, the NIR-Tree and Revised R*-Tree range queries were also equivalent, the NIR-Tree being faster by an average of 3.7%. Examining range

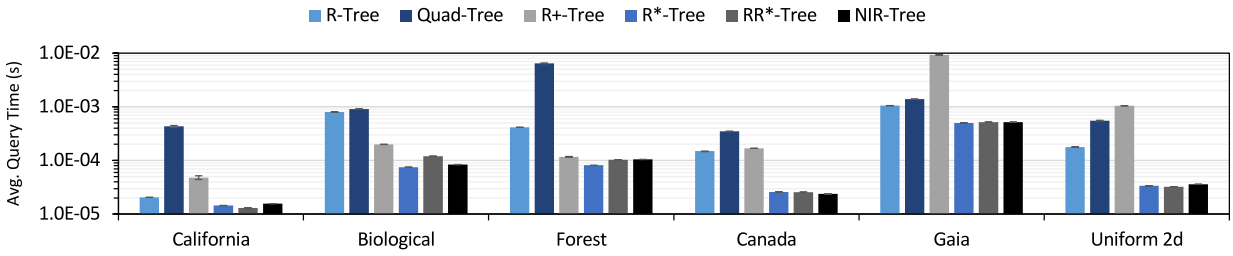

Figure 6.3: Range query times (log scale).

query performance, we see that the gap between the R*-Tree and the NIR-Tree is largest on the Forest and Biological datasets. In contrast with those highly skewed datasets, map-style datasets with less skew (California, Canada Roads, Gaia) demonstrated the NIR-Tree is very competitive with the R*-Tree. Distinct from the R*-Tree trend, the NIR-Tree equaled or exceeded the performance of the Revised R*-Tree on all datasets except California and Uniform 2d. Lastly the Quad-Tree, while a close competitor of the NIR-Tree for point query and insertion times, was consistently an order of magnitude slower to range search than the NIR-Tree.

Range queries within the NIR-Tree and R*-Tree accessed a similar number of nodes on map-style datasets. 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 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 per node. 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 6.3.4). However, with skew increasing the average number of rectangles per bounding polygon to 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.

Range queries on the Revised R*-Tree and NIR-Tree were identical for the Forest and Gaia datasets, performing within 1% of each other. Next, on the Canada Roads and Biological datasets, the NIR-Tree performed range queries 25% more efficiently on average

than the Revised R*-Tree. Without re-insertions or flexible bounding objects, the Revised R*-Tree's ability to eliminate skew was dampened on these datasets. By contrast, the two datasets where range queries performed 13.5% more efficiently on the Revised R*-Tree were exactly those datasets with limited skew, namely California and Uniform. Overall, the NIR-Tree was competitive with the Revised R*-Tree.

Range queries executed exceptionally poorly on the Quad-Tree. For all datasets except Gaia, range queries were over 10× faster when performed on the NIR-Tree. The best range query performance for the Quad-Tree was on the Gaia dataset where the NIR-Tree still maintained a significant 2× speedup. Commensurate with its order of magnitude advantage, the NIR-Tree accessed over an order of magnitude fewer nodes during range queries. Again taking the Biological dataset as an example, the Quad-Tree accessed an average of 1798 nodes per range search compared to the 124 nodes of the NIR-Tree. Compounding the scatter effect, traversing the Quad-Tree for range queries required much more computation than traversing the Quad-Tree for point queries. Instead of exactly d less-than comparisons per node, range queries required a point in rectangle test in addition to 2^d quadrant-query rectangle intersection tests per node to direct further searching. Pitfalls of Quad-Tree range queries are made most obvious by the Forest dataset. There the data is highly skew, creating a deep tree, and the number of dimensions forces the fanout of each node of the Quad-Tree to approach the fanout of the NIR-Tree and other R-family trees. Quad-Tree nodes account for $2^5 = 32$ possible branches, and must test each of these branches when range querying, yet many of these branches are short and represent dead ends of the tree. As a result, the Quad-Tree spends an excessive amount of time testing the query rectangle against quadrants, inhibiting range queries.

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.

6.3.4 Memory Overhead

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 sizes, polygon size,

and tree heights show these concerns to be unfounded. Across all experiments, the average 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 6.1). 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.

To index without intersection, the NIR-Tree uses only about 20% more memory compared to the space-efficient R*-Tree (Table 6.3). 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 (Table 6.2) of any other with the exception of the Quad-Tree (Table 6.2), aligning with their memory consumption pattern (Table 6.3). The NIR-Tree used on average 2.6× the number of nodes competitor trees used, but had half the maximum fanout of R-family competitor trees. Thus, a 2× size in nodes is expected. Still, the NIR-Tree shows 1.3× above the expected number of nodes. These extraneous nodes in the NIR-Tree make up the last portion of memory overhead, and result from downward splits during Algorithm 4, which keep the NIR-Tree intersection free.

Lastly, the Quad-Tree used an average of 2.3× the memory footprint of the NIR-Tree. For example, the Quad-Tree used 1.07GB to index the Gaia dataset while the NIR-Tree used 516MB and the most compact tree, the R*-Tree, used 439MB (Table 6.3). Since each node with a Quad-Tree contains exactly one data point and space for pointers to 2^d descendants, the Quad-Tree allocates space for pointers it does not use, inflating its memory overhead into the gigabyte range, while R-family trees require only on the order of megabytes. The Quad-Tree's point per node paradigm also inflates the height of tree. At it's deepest, the Quad-Tree had a height of 30 on the most skew dataset Forest, and 2086 on the largest dataset Canada Roads. Extreme heights and a large memory overhead illustrate the considerable trade off the Quad-Tree makes to achieve efficient insertions and point queries.

Table 6.1: 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

Table 6.2: Tree size by dataset measured in nodes.

Dataset	R-Tree	Quad-Tree	R^+-Tree	R*-Tree	RR*-Tree	NIR-Tree
California	30,526	1,888,012	30,386	29,843	49,622	75,298
Biological	187,744	11,958,999	198,944	167,306	261,392	553,831
Forest	9,222	581,012	12,599	8,334	14,850	31,945
Canada Roads	310,016	19,371,405	310,219	281,370	386,475	699,203
Gaia	286,303	18,084,053	267,943	258,304	378,921	709,576
Uniform 2d	155,943	10,000,000	146,131	143,639	207,124	325,449

Table 6.3: Memory usage (MB).

Dataset	R-Tree	Quad-Tree	R^+-Tree	R*-Tree	RR*-Tree	NIR-Tree
California	32	100	31	31	34	38
Biological	300	729	296	290	309	349
Forest	23	44	24	23	24	27
Canada Roads	335	1030	328	319	339	386
Gaia	455	1100	446	439	465	516
Uniform 2d	172	534	168	164	176	192

6.3.5 Node Utilization

Excluding the Quad-Tree, the height of competitor trees was 4 across all experiments. Due to a smaller maximum fanout, the NIR-Tree used one level more than competitor trees, for a tree height of 5 (with the exception of Forest where it used 4). Concretely, the NIR-Tree's competitors index the largest dataset, Canada Roads, in 4 levels which requires a node utilization of at least 60% given their maximum fanout of $m = 100$. Theoretically, the NIR-Tree could index 9.76M points in a 5 level tree with 50% utilization of $m = 50$. With a higher utilization of 60%, the NIR-Tree could theoretically index up to 24.3M points in a 5 level tree. Comparing these theoretical utilizations with measured NIR-Tree heights and competitor tree heights, we see that the NIR-Tree achieves good node utilization of well over 50%, indexing the 19M points of Canada Roads in 5 levels.

6.3.6 Summary

We compared the NIR-Tree with five spatial indexes and found that only two, the popular R*-Tree and sate-of-the-art Revised R*-Tree, provided similar range queries. Among these three front-running trees, we observed that constructing a NIR-Tree is on average 27× faster than constructing an R*-Tree, and on average 1.2× faster than constructing a Revised R*-Tree without trading off for query times in either instance. Moreover, the NIR-Tree achieved range searches equivalent with both trees and point searches 2.2× and 3.1× faster respectively.

Chapter 7

Conclusion and Future Work

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. With simple yet flexible bounding polygons, the NIR-Tree is an efficient, state-of-the-art multidimensional index.

The NIR-Tree offers at least three future directions for multidimensional indexing. To benefit datasets that cannot be indexed in-memory, it would be interesting to extend the NIR-Tree to disk. With a tree-based paradigm, the NIR-Tree is already appropriately structured for paging. Furthermore, disk access dominating search times means the NIR-Tree could afford more complex polygons at each level, in turn speeding range searches. Next, multithreading within the NIR-Tree can offer greater concurency, and therefore preformance. The NIR-Tree is suited to such an extension due to a split strategy that requires access only to descendant nodes and not the entire insertion path within the tree. Finally, the problem of covering or re-covering a bounding polygon's area with a different configuration of constituent bounding rectangles during the insertion operation is an interesting theoretical geometry problem.

References

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

[2] R. Bayer and E. McCreight. 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, page 107141, New York, NY, USA, 1970. Association for Computing Machinery.

[3] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 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, pages 322–331, New York, NY, USA, 1990. Association for Computing Machinery.

[4] Norbert Beckmann and Bernhard Seeger. A benchmark for multidimensional index structures, 2008.

[5] Norbert Beckmann and Bernhard Seeger. 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, page 799812, New York, NY, USA, 2009. Association for Computing Machinery.

[6] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The x-tree: An index structure for high-dimensional data. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 28–39, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[7] 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. Gaiadata release 1. *Astronomy and Astrophysics*, 595:A2, Nov 2016.

[8] Statistics Canada. Census road network files, 2016.

[9] Gaia Collaboration, A. G. A. Brown, A. Vallenari, T. Prusti, J. H. J. de Bruijne, C. Babusiaux, and M. Biermann. Gaia early data release 3: Summary of the contents and survey properties, 2020.

[10] Volt DB. Volt db, 2010-05-25.

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

[12] PostgreSQL Global Development Group. Postgresql, 1996-2021.

[13] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. Association for Computing Machinery.

[14] IBM. Ibm db2, 2020.

[15] SingleStore Inc. Singlestore, 2013-05-23.

[16] H. V. Jagadish. Spatial search with polyhedra. In *Proceedings of the Sixth International Conference on Data Engineering, February 5-9, 1990, Los Angeles, California, USA*, pages 311–319. IEEE, 1990.

[17] Ibrahim Kamel and Christos Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, page 500509, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[18] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp olap main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*, pages 195–206, 2011.

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

[20] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. Lisa: A learned index structure for spatial data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 21192133, New York, NY, USA, 2020. Association for Computing Machinery.

[21] King-Ip Lin, Hosagrahar V Jagadish, and Christos Faloutsos. The tv-tree: An index structure for high-dimensional data. *The VLDB Journal*, 3(4):517–542, 1994.

[22] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. Learning multi-dimensional indexes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 9851000, New York, NY, USA, 2020. Association for Computing Machinery.

[23] J. Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst.*, 9(1):3871, 1984.

[24] Matthaios Olma, Farhan Tauheed, Thomas Heinis, and Anastasia Ailamaki. 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, New York, NY, USA, 2017. Association for Computing Machinery.

[25] Oracle. Mysql, 2021.

[26] Robbi Bishop-Taylor Robson Fletcher. A visualization of the roadways across canada, 2018.

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

[28] SAP SE. Sap hana — in-memory database, 2021.

[29] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th International Conference on Very Large Data Bases*, VLDB '87, page 507518, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc.

[30] Darius Sidlauskas, Sean Chester, Eleni Tzirita Zacharatou, and Anastasia Ailamaki. Improving spatial data processing by clipping minimum bounding boxes. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 425–436, 2018.

[31] Sheng Wang, David Maier, and Beng Chin Ooi. Fast and adaptive indexing of multi-dimensional observational data. *Proc. VLDB Endow.*, 9(14):16831694, 2016.

[32] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings*

of the 24rd International Conference on Very Large Data Bases, VLDB '98, page 194205, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

APPENDICES

Appendix A

Dataset Detail

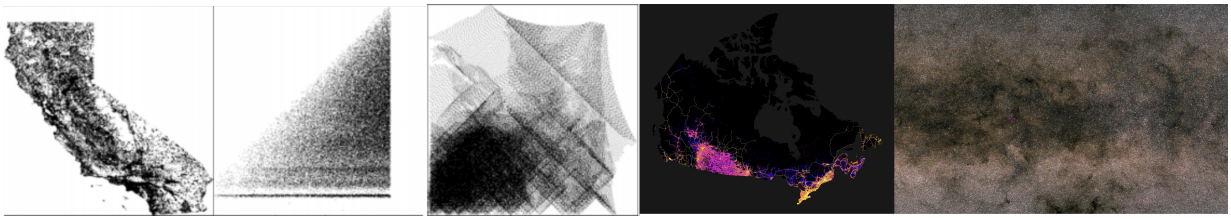

Figure A.1: Left to right: California, Biological, Forest, Canada Roads, Gaia.

The following dataset descriptions contain extended discussion including more precise characterizations of the skewness of each dataset. Descriptions use the Fisher-Pearson coefficient to measure the skewness of each dataset. For reference, the Normal distribution has a coefficient of 0, while the Exponential distribution has a coefficient of 2. Negative coefficients describe a skew towards larger values while positive coefficients describe a skew towards smaller values. In addition to measuring the Fischer-Pearson coefficient, we measured the kurtosis of each dataset. Kurtosis measures the difference between the tails of a Normal distribution and the given distribution. A kurtosis value of 0 means the tails are equivalently shaped. High kurtosis implies that either outliers are extreme and the majority of data is concentrated around the mean, or that outliers are less extreme and the majority of data is spread out into the tails of the distribution. Low kurtosis implies that the majority of the data is concentrated around the mean with few outliers. The visualizations in Figure A.1 aid in interpreting the meaning of skew coefficients and kurtosis values.

- **California**: A subset of the larger TIGER/Line dataset mapping the United States [4, 5]. Visualization of the data in Figure A.1 is sourced from the author of the benchmark [4]. The data is a mixture of points and rectangles expressed as doublets or quartets of GPS coordinates. No false Northing or false Easting is present, the coordinate data corresponds exactly to locations in California. Due to the mixed nature of the data, we represented rectangles by their centroids. The overall distribution of points corresponds to the location of cities. Roads are present throughout the state but densest around the cities on the coast and in the South. Expressed concretely, the Fisher-Pearson coefficients of skewness is 0.0149 for the East-West dimension and 0.4021 for the North-South dimension. Thus, data points in the California dataset are distributed mostly uniformly East to West while being significantly skewed to the South, validating our qualitative observation that cites are located generally on the coast and to the South. Extreme outliers are not present in this dataset as can be observed from Figure A.1 and the kurtosis value confirms this; both dimensions have a kurtosis value of approximately -1.0. Dimensions = 2. Size = 1.8M points.

- **Biological**: A large real collection of points representing biological features [4, 5]. Visualization of the data in Figure A.1 is sourced from the author of the benchmark [4]. Points cluster in a pyramid-shaped region between the X and Y axes with a distinct break before clustering in a sheet parallel to the X axis. The high coefficient of skew, -1.1180, for the X axis confirms this, being more than halfway to resembling an exponential distribution. The remaining dimensions exhibit a lesser but still high level of skew at -0.4323 and 0.3319. Similar to the California dataset, few outliers exist and the kurtosis values are accordingly small negative values. For the X dimension where skew is extreme, a high kurtosis of 0.9078 aligns with the long tail observed in Figure A.1. 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]. Visualization of the data in Figure A.1 is sourced from the author of the benchmark [4]. Attributes describing elevation, distance to surface water, distance to wildfire ignition points, and distance to nearest roadways are indexed. All measures are recorded in meters. Data is skewed towards water features and directional stream patterns are present. The third dimension is so skewed with skew coefficient 1.7902, it almost resembles an exponential distribution. Similarly, the second and fifth dimensions are also extremely skewed with coefficients 1.1404 and 1.2886 respectively. All but the first dimension exhibit positive skew, favoring small values and making the area around the origin dense with data. This dataset is therefore particularly challenging for multidimensional indexes because of

its relatively high dimensionality and extreme skew properties. Dimensions = 5. Size = 581K points.

- **Canada Roads**: A large collection of polygons representing the road network of Canada created by Statistics Canada [8]. Visualization of the data in Figure A.1 is sourced from the CBC [26]. All the corners of each polygon were indexed due to their axis-unaligned nature. False Northing and false Easting is present, forcing the sign of every coordinate to be positive. Data is densest around population centers in the South West and South East, and sparser in the Mid-West and North. The visualization highlights longer more major roads in brighter colors and illustrates an almost bimodal East-West distribution with skew coefficient -0.5123. The kurtosis value of the East-West dimension is accordingly -1.2335. With a few roads extending into the Northern region of Canada, extreme outliers from the mean in the South create another high kurtosis value of 1.7443 for the North-South dimension. The North-South dimension is about as skewed as the East-West dimension with a coefficient of 0.8316, the change in sign indicating skew towards smaller values instead of larger values. 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 [7, 9]. Visualization of the data in Figure A.1 was created via the Gaia Visualization tool [7, 9]. 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, which is visualized as two bright bands in the center of the image with a gap in between. Moving up from the galactic plane, data is slightly sparser, visualized as the less bright blue regions at the top and bottom. Although the density of the data does vary, skewness coefficients of -0.0420 and 0.0811 for the first two dimensions confirm the visual intuition that the data is relatively uniform. In contrast with the relative uniformity of the first two dimensions, distance from the Sun is skewed towards smaller values with a coefficient of 1.6009. Combining an anchor point of Proxima Centauri with the intra-galaxy mission of Gaia resulted in a selection of stars close to the Sun. In keeping with the relatively uniform nature of the data, kurtosis values for this dataset were low for the first two dimensions at -1.2086 and -1.3234 respectively. The third dimension on the other other hand, contains outlier distances as a few stars are far away while the majority are close. The very high kurtosis value of 5.9444 confirms this intuition. Distance from the Sun on the other hand, contains 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. By construction the data is not skew and concentrates uniformly from the mean. Therefore, we did not compute Fischer-Pearson coefficients or kurtosis values for this dataset. Dimensions = 2. Size = 10M points.

Appendix B

Visualizations

Below are visualizations of all six multidimensional indexes discussed throughout this thesis. We produced the visualizations of each tree on a two dimensional uniform dataset containing $10,000$ points and with maximum and minimum fanouts set as in Chapter 6. Trees consisted of exactly three levels: root, middle, and leaf. The root level visualizations of each tree contained at most three rectangles and are therefore not shown. Leaf level visualizations consist of an uninteresting uniform field of points and are not shown. Therefore, for all indexes except the Quad-Tree, indexes were visualized using only their middle level bounding boxes. Each bounding box is shown as a different colored rectangle, or in the case of the NIR-Tree, collection of rectangles. The Quad-Tree is slightly different due to its space partitioning nature. Instead of displaying bounding rectangles which group data together, the Quad-Tree's visualization displays all of its space partitioning lines, each in a different colour. The purpose of these visualizations is to further illustrate the geometric shape of each tree discussed. In particular note the intersection area of bounding rectangles of the R-Tree in Figure B.1, the striping of the R^+-Tree in Figure B.1, the improved but not eliminated intersection area of bounding rectangles of the R*-Tree and RR*-Tree in Figure B.1, the volume of space-partitioning lines in the Quad-Tree in Figure B.2, and finally the zero-area intersection of bounding polygons within the NIR-Tree in Figure B.3.

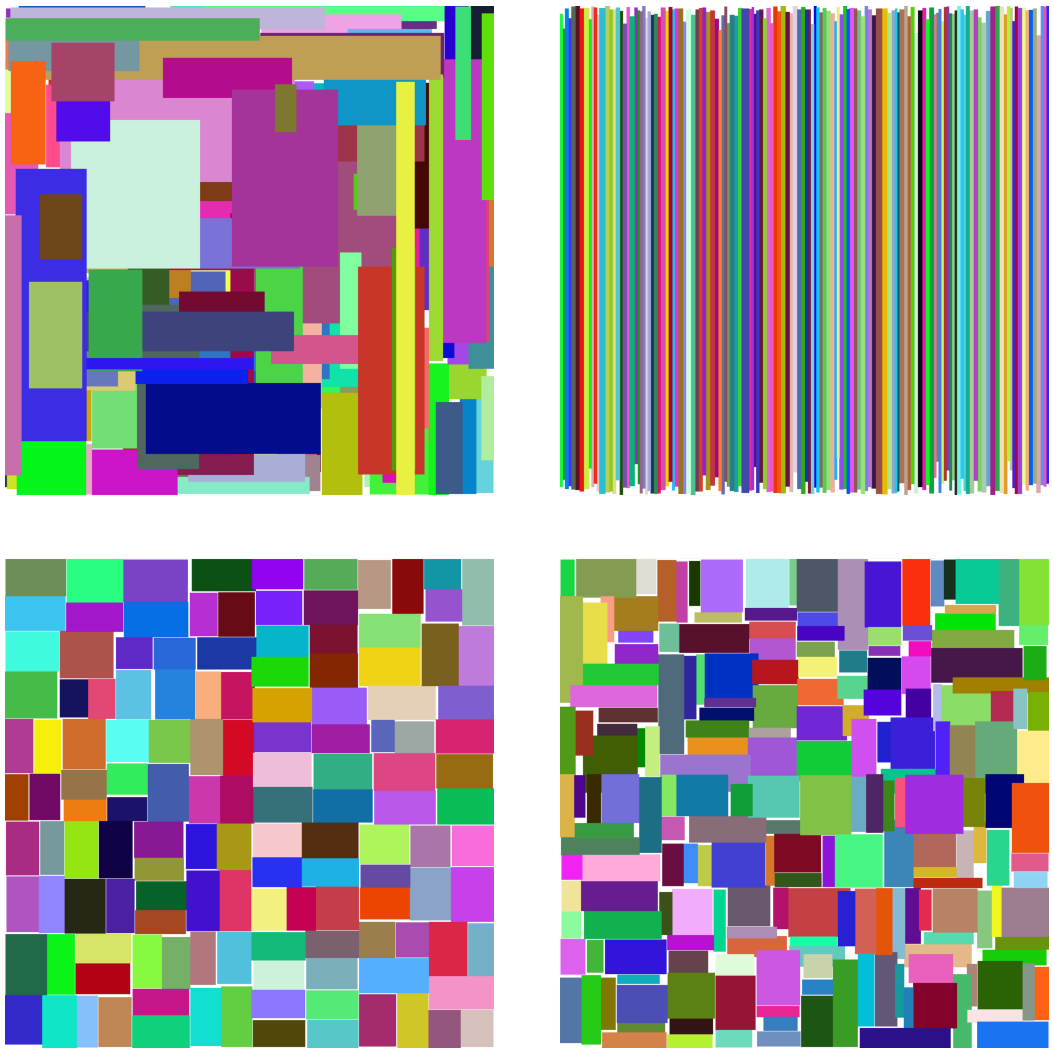

Figure B.1: Clockwise from the top: R-Tree, R^+-Tree, RR*-Tree, R*-Tree.

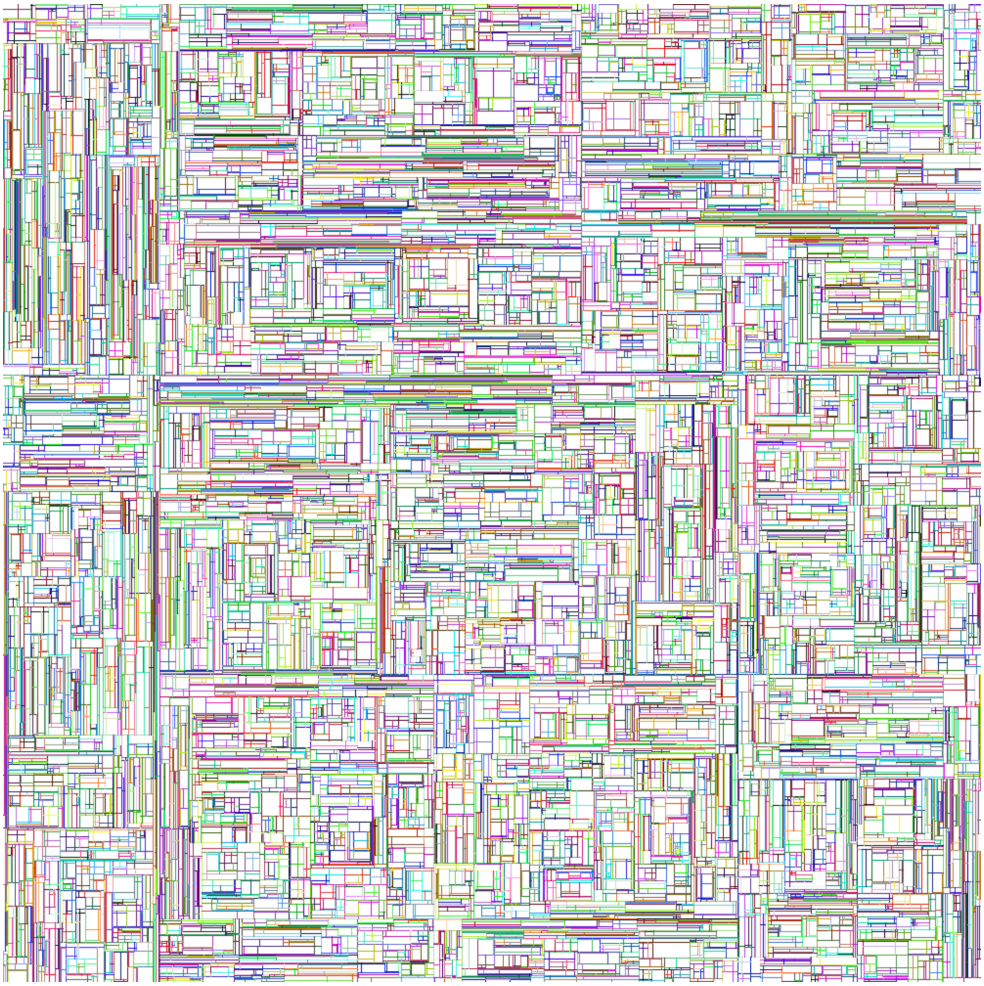

Figure B.2: A Quad-Tree. Displayed large to preserve detail.

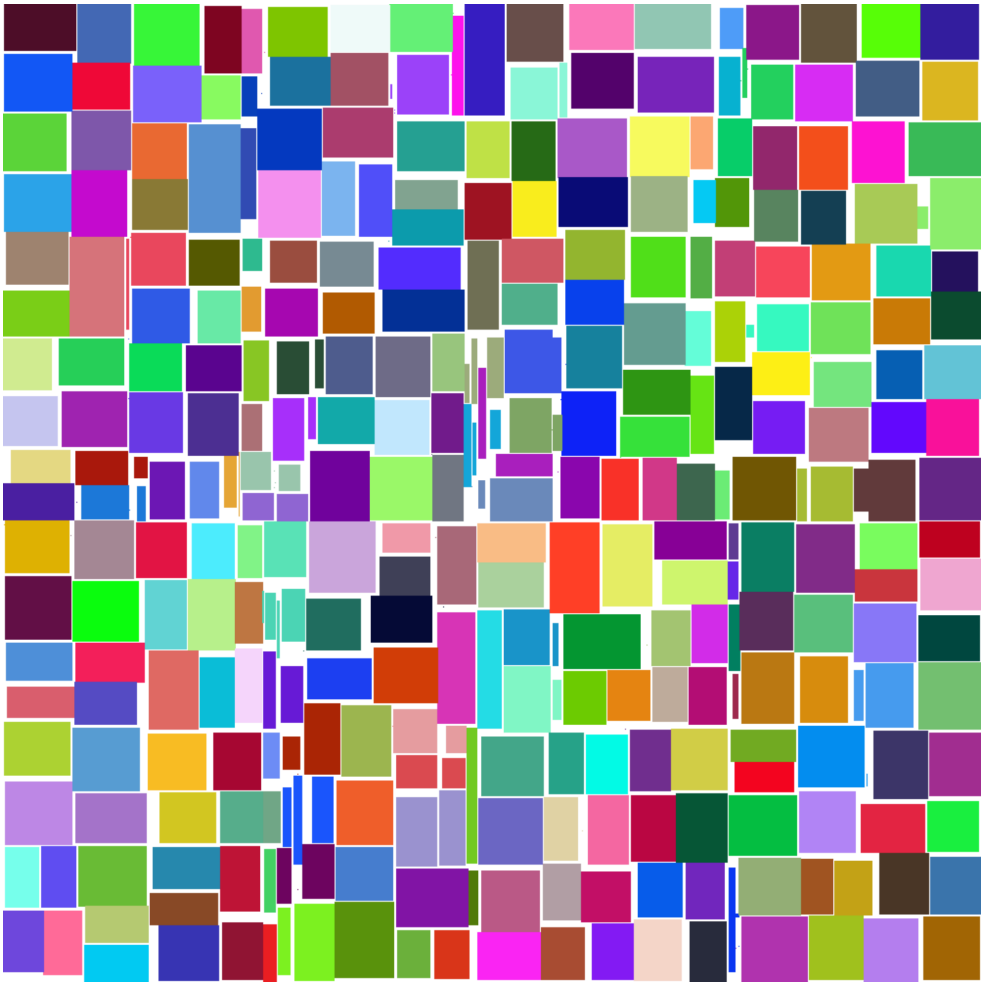

Figure B.3: A NIR-Tree. Displayed large to preserve detail.