Disk-based Indexing for NIR-Trees using Polygon Overlays

by

Fadhil Abubaker

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, 2024

Author's Declaration

I hereby declare that I am the sole author of this 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.

Abstract

This thesis presents the NIR+-Tree, a disk-resident R-Tree variant that eliminates overlap among its minimum bounding rectangles (MBRs). The NIR+-Tree is an extension of the main-memory NIR-Tree [22], adopting techniques for efficient storage and retrieval on disk. By employing non-intersecting polygons instead of rectangles for data partitioning, the NIR+-Tree minimizes the number of spurious disk accesses incurred due to MBR overlap. To stabilize the height of the NIR+-Tree, the dynamically-sized polygons are stored in main-memory using an efficient encoding. Experimental results show that the NIR+-Tree is efficient at point queries and selective range queries, using $2\times$ to $5\times$ fewer disk accesses than its closest competitors, the R+-Tree and the R*-Tree.

Additionally, this thesis investigates bulk-loading algorithms for the NIR+-Tree. Bulk-loading can be used to efficiently construct an index from a pre-defined set of data. Bulk-loading algorithms that generate MBRs with significant overlap create NIR+-Trees with undesirable, complex polygons. This thesis shows that top-down bulk-loading algorithms are better suited for the NIR+-Tree than bottom-up algorithms, due to their overlap minimizing properties. These techniques enable the NIR+-Tree to be a complete, disk-based indexing solution for spatial data.

Acknowledgements

I would like to thank my advisor, Prof. Khuzaima Daudjee, for his support and supervision during my master's. From our multiple conversations in the year before I came to Waterloo, to the many discussions we had during my time here, I am grateful for his guidance and mentorship.

I would like to thank Prof. Trevor Brown and Prof. Grant Weddell for serving on my committee and for their valuable feedback on this thesis.

I owe much thanks to Brad Glasbergen for introducing me to the NIR-Tree project and for providing the initial code on which this work is based on. I want to express my gratitude to Shirley Chen, who is an amazing collaborator and proved to me that good research is often a team effort.

My friends at Waterloo and the Data Systems Group have made the past two years fun and enjoyable. In particular, I would like to thank Benson Guo and Amine Mhedhbi for their support and advice, both personal and professional.

Before coming to Waterloo, I was fortunate to have a mentor in Hossam Hammady during my time at QCRI. Under his watchful eye, I got the opportunity to work on many technical challenges within Rayyan, which in turn sparked my interest in data systems research and motivated me to pursue a master's.

Lastly, I would like to thank to my family for their never-ending support through this journey. I would like to thank Umma and Uppa for their love and encouragement during the crucial years of my education, starting from MUWCI, to CMU-Q and now Waterloo. Tutu has always been a constant source of inspiration and I hope to support her on her own journey in the years to come. Finally, I would like to thank Nana, whose wonderful presence has brought so much warmth, joy, colour and beauty into my life...

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The use of spatial data has grown at an unprecedented scale over the past decade due to the increasing popularity of location-based technologies. Applications relying on spatial data are ubiquitous in daily life, ranging from ride-hailing services to navigation systems. As a result, there is a need to develop algorithms and data structures for efficient spatial data processing.

R-Trees [17] are a popular class of data structures for indexing spatial data. R-Trees group spatial data into rectangles and then further organize these rectangles into a hierarchy. This results in a tree-like structure, where the larger rectangles at higher levels of the tree encapsulate the smaller rectangles at lower levels of the tree. Since the rectangles encompass spatial objects within the smallest extent possible, they are also called minimum bounding rectangles (MBRs). Point or range queries on an R-Tree involve inspecting MBRs that satisfy the search criteria at each level and then recursively traversing the tree from top to bottom.

The search performance of an R-Tree depends heavily on the area, perimeter and degree of overlap between its constituent MBRs [3]. Many variants of R-Trees have been developed over the years [37, 3, 5], each of which use different strategies to optimize the size and shape of their MBRs. Numerous augmentations to MBRs themselves have been proposed [9, 38], which can be adapted for any tree that uses MBRs for spatial data partitioning.

In particular, overlap between rectangles in an R-Tree can significantly affect performance, such that the search complexity increases from $O(\log n)$ to $O(n)$ in the worst case. For example, Figure 1.1 shows the geometric layout of an R-Tree on the left and the structure of its nodes on the right. Consider the range query executed on this R-Tree in the same figure. Since the query lands on the overlapping region between node A and B, the

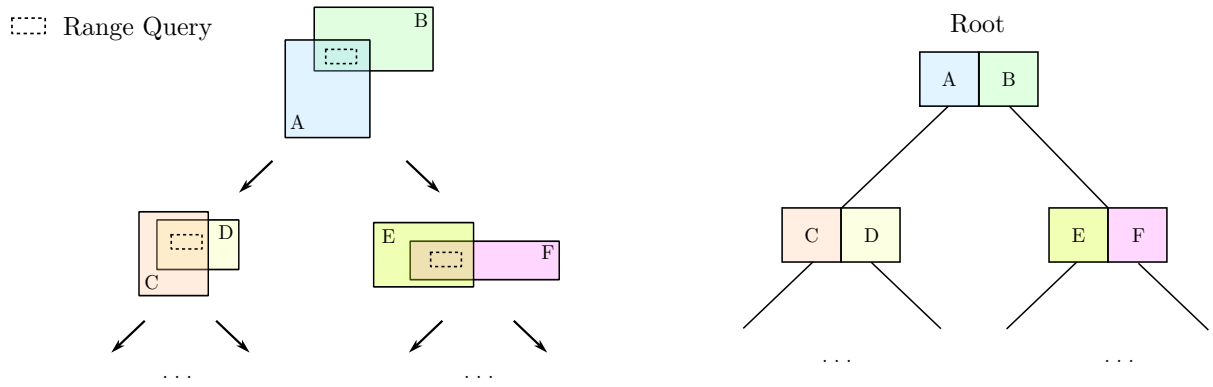


Figure 1.1: A range query executed on an R-Tree landing on multiple overlapping regions.

search algorithm has to traverse both nodes to find relevant data. The same effect is seen in the next level of the tree, where the query lands on the overlapping region between nodes C and D as well as nodes E and F. This in turn causes the query to descend all four nodes, potentially traversing branches that do not yield any results. By obfuscating which node the region belongs to, overlap can cause many spurious searches that degrade query performance. Consequently, overlap at higher levels of an R-Tree can induce many such searches, as they force queries to explore a larger portion of the tree to find relevant data.

Recently, Langendoen et al. introduced the NIR-Tree [22], an in-memory R-Tree variant that completely eliminates overlap. It does so by fragmenting a single overlapping rectangle into multiple smaller ones and forming them into an axis-aligned polygon that no longer intersects. Experimental results show that the NIR-Tree is very efficient at point queries while providing comparable range search performance with other popular R-Tree variants.

However, being an in-memory index, the NIR-Tree is limited to handling small datasets that can fit entirely in RAM. Furthermore, the price-to-performance ratio of main-memory continues to be significantly higher than secondary storage [26]. As a result, data systems popularly use secondary storage to handle large volumes of spatial data that grow beyond the capacity of main-memory. This is especially true for applications that rely on location-based technologies, as they often deal with large amounts of spatial data. Therefore, there is significant motivation to adapt the NIR-Tree for disk-based indexing.

The I/O (Input/Output) cost model is popularly used for analyzing the performance of secondary storage indexes [12]. Operating systems read and write to disks in fixed-size pages, with each such disk access counting as one I/O. Therefore, disk-based indexes are typically decomposed into pages and laid out on the storage medium. Due to the high cost

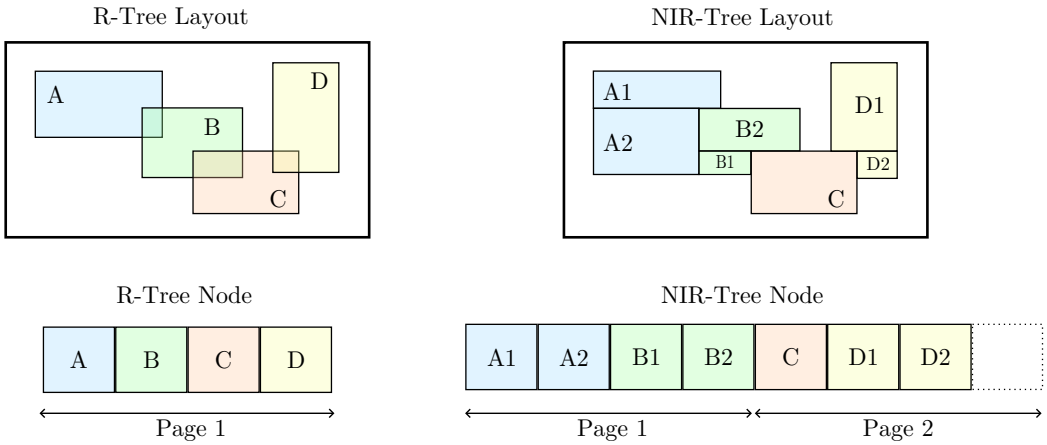


Figure 1.2: Rectangle layout and branch node structure of an R-Tree and a NIR-Tree.

of disk accesses, these indexes must aim to reduce the I/O cost of operations. The key strategy is to maximize the data stored in a single page, so that a single I/O returns as much useful information as possible. This is done by setting the fanout — the number of children in each node — such that a single node occupies an entire disk page. This in turn reduces the height of the R-tree and minimizes the number of I/Os required to traverse it.

Conventional disk-based layouts pose two challenges for the NIR-Tree in comparison to other R-Trees. First, each child in a NIR-Tree is associated with a polygon rather than a rectangle. This introduces increased storage overhead as a single polygon is composed of multiple rectangles. Second, the number of rectangles that make up the polygons in a NIR-Tree are unbounded and cannot be predicted ahead of time. Figure 1.2 shows the branch node and geometric layout of an R-Tree and a NIR-Tree with fanout = 4. Assuming a page can hold 4 rectangles, the R-Tree can fit its branch node within a page, while the NIR-Tree has to spill over to a second page. Thus, extensive overlap can produce large polygons that cause a node to overflow a disk page in a NIR-Tree. As a result, the NIR-Tree uses more I/Os than other R-Tree variants to execute the same query, making it less suitable for disk-based indexing.

This thesis proposes the NIR+-Tree, an improved variant of the NIR-Tree that can efficiently index spatial data on disk. The key insight is that the original rectangles the NIR+-Tree uses to derive polygons can be kept on disk, while the polygons themselves can be stored separately in main-memory. This stabilizes the fanout of the NIR+-Tree and keeps its I/O costs competitive with other R-Trees. During query execution, the polygons in main-memory can be used to disambiguate overlapping regions, reducing the number of

spurious I/Os. The term *polygon overlay* is used to refer to this design, since the geometric shape of the rectangles on disk are refined by the polygons in memory.

Additionally, various bulk-loading strategies have been proposed to efficiently construct indexes from the R-Tree family given a pre-defined set of input data [24, 13, 35]. By exploiting a global ordering on the input dataset, bulk-loading produces R-Trees with better layout and space utilization in less time than sequential inserts. While these algorithms can be adapted for the NIR+-Tree, they do not take into account the polygons generated by it. This work also investigates optimal bulk-loading algorithms for the NIR+-Tree. Specifically, an optimal bulk-loading algorithm must start with minimal overlap so that the NIR+-Tree produces small polygons. This ensures that subsequent insertions do not quickly generate complex polygon overlays.

The rest of this thesis is organized as follows. Chapter 2 discusses background on R-Trees and NIR-Trees. Chapter 3 outlines the design of the NIR+-Tree and the concept of polygon overlays. Chapter 4 explores suitable bulk-loading algorithms for the NIR+-Tree. Chapter 5 presents experimental evaluations on the NIR+-Tree against other R-Tree variants. Chapter 6 covers related work on spatial indexing. Chapter 7 concludes this thesis and proposes future work on the NIR+-Tree.

Chapter 2

Background

This chapter introduces the R-Tree data structure, describes its insert and search algorithms and illustrates how R-Trees generate MBRs that can overlap. Two different R-Tree variants are also presented, the R+-Tree [37] and the R*-Tree [3]. Finally, this chapter describes the NIR-Tree, an in-memory variant of the R-Tree which fragments overlapping rectangles into polygons to eliminate overlap.

2.1 R-Tree

R-Trees are the multi-dimensional equivalent of the ubiquitous B-Tree [2], designed for indexing spatial data in any number of dimensions and optimized for disk accesses. R-Trees group together spatial data that are close to each other into leaf nodes and represent them with their minimum bounding rectangles (MBRs). The MBRs of leaf nodes are then further grouped together into branch nodes, whose MBRs are then recursively grouped into more branch nodes, and so on, thereby constructing a hierarchial data structure. MBRs can be seen as an approximation of the spatial extent of the objects they enclose. They can be used to quickly answer spatial queries by examining only those MBRs that intersect with the query region.

Table 2.1 summarizes the functions used by the insert algorithm which are common across all R-Tree variants. Spatial data is inserted into the R-Tree using a two-step process. In the first step, a leaf node needs to be selected to contain the new data entry. This is handled by the `chooseLeaf` function, which traverses the tree starting from the root and selects the child node with the MBR that requires the least expansion area to enclose the

new data entry. It may also be possible that no expansion is needed if the data entry to be inserted already lies within an existing MBR. The algorithm then expands the chosen MBR, descends into the selected node and repeats the process of selecting a new child node. This continues until the algorithm reaches the leaf level, at which point a leaf node is chosen to contain the new data entry.

In the second step, the `adjustTree` method is called to split nodes in the tree that have exceeded the maximum fanout M. Starting from the leaf node chosen to contain the new data entry, the algorithm traverses the tree upwards, splitting any overfull nodes and propagating new split nodes upwards. The algorithm terminates when it encounters a node that does not need to be split. When encountering a node that needs to be split, the `splitNode` function is called to decide how to partition the entries in the node into two new nodes. There are three different variants of the `splitNode` function for the original R-Tree, each with varying complexity: the linear split, the quadratic split and the exponential split. Out of these, the quadratic split is the most commonly used, as it offers a good balance between performance and partitioning quality. Table 2.2 lists the computational complexity of the `chooseLeaf` and `splitNode` methods at a single level of the tree for all R-Tree variants.

Function name	**Description**
`chooseLeaf`	Traverses down the tree and expands MBRs of nodes to contain a new data entry.
`adjustTree`	Traverses up the tree and splits overfull nodes, propagating new split nodes upwards.
`splitNode`	Called by `adjustTree` to split an overfull node into two new nodes.

Table 2.1: Functions used in the insert procedure of R-Trees.

	`chooseLeaf`	`splitNode`
R-Tree	$O(M)$	$O(M^2)$
R*-Tree	$O(M^2)$	$O(M \log M)$
R+-Tree	$O(M)$	$O(M \log M)$
NIR-Tree	$O(M)$	$O(M)$

Table 2.2: Complexity of `chooseLeaf` and `splitNode` methods at a single level of the tree across R-Tree variants. M is the number of entries in a node.

The following is a brief description of the quadratic split of the R-Tree: The split algorithm iterates through all possible $\binom{M+1}{2}$ pairs of entries in the overfull node and

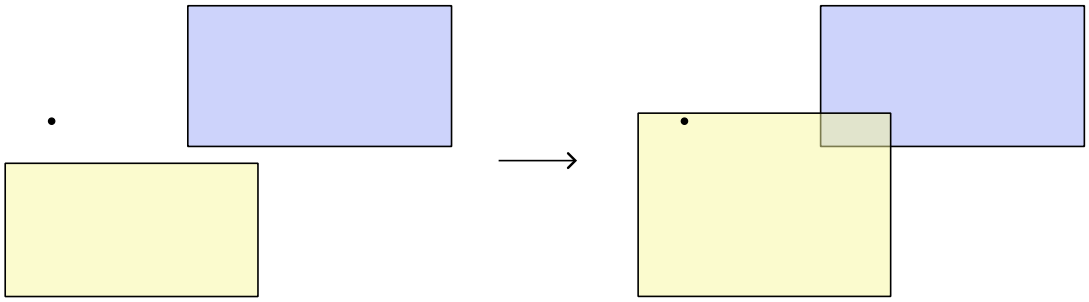


Figure 2.1: MBR expansion during the `chooseLeaf` operation causing overlap.

computes the minimum bounding rectangle (MBR) comprised of each pair. Following this, the algorithm then selects the pair of entries whose MBR has the largest area. These two entries are then assigned to be the starting entries of the two new nodes. The intuition is that grouping these two entries together will result in the worst possible MBR with a large area and hence they must be separated into different nodes. Once the starting entries have been selected, the algorithm then iterates through the remaining entries and assigns each entry to the node whose MBR requires the least expansion area to cover the entry. The algorithm terminates when each entry has been assigned to one of the two new nodes.

Searches in R-Trees are performed by traversing the tree starting from the root node: each child in the root node is examined to determine if its MBR intersects with the query point or region. If so, the algorithm descends into the child node and recursively searches its children. On encountering a leaf node, the algorithm then examines each entry in the node to determine if it intersects with the query. If so, the entry is added to the result set. In the case of point searches, the algorithm can immediately terminate once it has found the first matching entry. Range queries, on the other hand, require the algorithm to examine all leaf nodes that intersect with the query region to collect data objects that satisfy the query.

Overlap in the MBRs of R-Trees happen during execution of the `chooseLeaf` and the `splitNode` functions. In the case of `chooseLeaf`, overlap can occur when the algorithm selects a node and expands its MBR to contain the new data entry. Even when optimizing for the smallest expansion area, it is still possible for the expanded MBR to overlap with other MBRs in the node. Figure 2.1 illustrates this, where expanding the yellow MBR is optimal with respect to the expansion area but this still causes overlap with its sibling MBR. For `splitNode`, overlap can occur due to the partitioning of entries into two new nodes. In particular, after dividing the entries into two groups, it is possible for the MBRs

of the two groups to overlap. For example, Figure 2.2 shows a node with four entries being split into two new nodes. The `splitNode` algorithm selects A and B to form the first node and C and D to form the second node. However, this grouping results in overlap between the two MBRs produced by the split. As such, it is often not possible to expand MBRs or partition nodes without introducing overlap. Consequently, queries that intersect with overlapping regions force searches to examine more nodes (by reading more disk pages) than necessary.

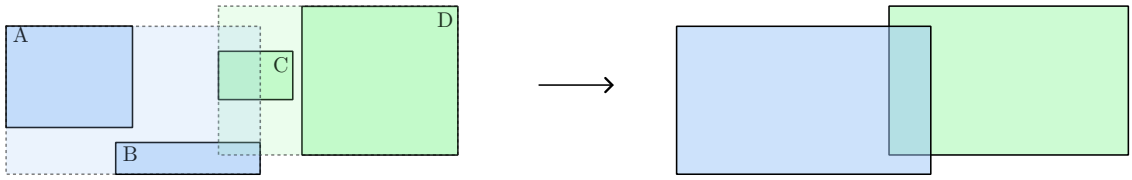


Figure 2.2: Overlap introduced by grouping in the `splitNode` operation.

2.2 R-Tree Variants

2.2.1 R*-Tree

The R*-Tree [3] is an R-Tree variant with modified versions of the `chooseLeaf` and `splitNode` functions. The distinguishing feature of the R*-Tree is that it performs re-insertions of the entries in a node to improve the spatial quality of its MBRs. The R*-Tree is the most popular R-Tree variant and is considered to be state-of-the-art along with its successor, the revised R*-Tree [5].

The `chooseLeaf` function of the R*-Tree is mostly the same as the original R-Tree, except that it uses a different heuristic to select a leaf node to contain the new data entry. Instead of selecting the MBR that has the least expansion area, the R*-Tree selects the MBR that produces the least overlap with other MBRs in the node when expanded. This optimization helps minimize disk I/O for point queries and small range queries [3].

The `splitNode` function of the R*-Tree differs significantly from the quadratic split of the R-Tree by incorporating MBR overlap and perimeter in addition to the area into the splitting heuristic. `splitNode` first selects a dimension to split by sorting the $M + 1$ entries in the node along a dimension and then iterating through all possible M splits that produce two groups each. The algorithm computes the MBRs of the two groups produced

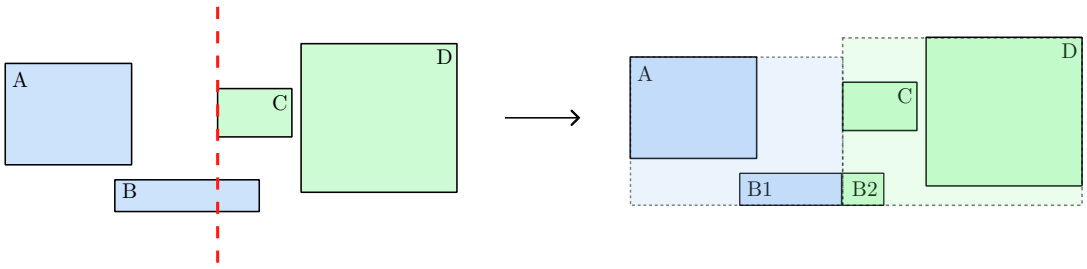


Figure 2.3: `splitNode` in the R+-Tree. The dashed red line is the split line. The split causes node B to be partitioned into B1 and B2 in the resulting grouping.

by each such split and sums up the perimeter values of the MBRs across all M splits; it then selects the dimension with the least perimeter sum as the split axis. Then, within the selected split axis, the algorithm selects the split that minimizes the overlap of the two MBRs. Ties are broken by selecting the split with the least MBR area. Note that R*-Trees have to maintain a minimum fanout m for each node, and so in practice only $M - 2m$ splits are considered to accommodate at least m entries in each new node.

The R*-Tree also performs re-insertions of the entries in an overfull node during the insert operation. When inserting a new data entry causes a node to exceed the maximum fanout M, instead of splitting the node immediately, the R*-Tree first selects some of the entries in the node to be re-inserted. For all the $M + 1$ entries in the node, the algorithm computes the distance between the entries and the center of the MBR of the node. It then sorts the entries in descending order of this distance and removes the first p entries. These p entries are then re-inserted into the tree using the original insert procedure. Intuitively, this process moves entries that are far away from the center of one node into another node that is more suited to contain them. Therefore, re-insertions can be considered as a form of reorganization of the entries in a node given the existing state of the tree. Experimental results show that re-insertions can significantly improve the quality of MBRs in the tree, reducing I/O costs by up to 50% [3].

2.2.2 R+-Tree

The R+-Tree [37] is an R-Tree variant that aims to reduce overlap in its constituent MBRs. In particular, the R+-Tree uses a modified node splitting technique ensuring that the MBRs produced by `splitNode` do not overlap.

For each dimension d, the `splitNode` function of the R+-Tree sorts entries in its node along that dimension and computes splits using each entry. Each such split determines a

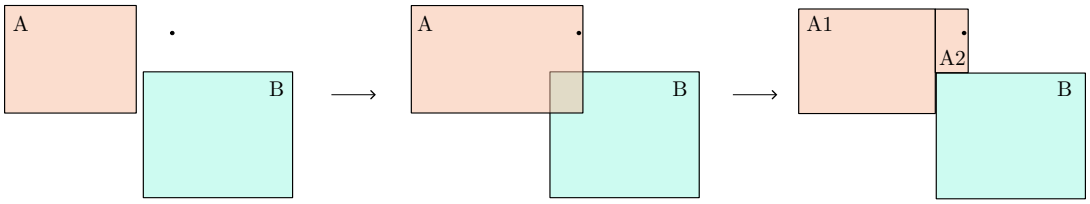


Figure 2.4: Expansion and fragmentation of an MBR into a polygon in the NIR-Tree.

partition line that can be used to divide the entries of a node into two groups. For points, the split is the d-th coordinate of the entry, while for rectangles it is the d-th coordinate of the lower left corner of the rectangle. While the R+-Tree does not have a minimum branch factor, it does have a fill factor f that can be configured when splitting a node, which ensures that one of the two new nodes will have at least f entries. Unlike other R-Tree variants, the R+-Tree propagates its split downwards, splitting nodes further down the tree if necessary using the same split. Figure 2.3 shows an example of a split in the R+-Tree recursively splitting its children. Note that downward propagated splits can partition nodes that are not overfull, and thus can grow the tree by a few extra nodes for every such split. However, these splits are useful for ensuring that the MBRs of nodes produced by `splitNode` do not overlap, which can make up for the increased tree size during query execution.

R+-Trees are similar to the K-D-B-Tree [34] introduced by Robinson et al. which is very efficient at indexing point data. The R+-Tree differs from the K-D-B-Tree by using MBRs that tightly enclose the points they contain, while the K-D-B-Tree's nodes simply partition space without bounding the constituent points — as a result, the MBRs of an R+-Tree cover the spatial data they index more precisely than a K-D-B-Tree. Moreover, R+-Trees (like other R-Trees) are capable of indexing rectangle data objects, while K-D-B-Trees are restricted to point data. Note that the R+-Tree [37] does not modify the `chooseLeaf` function, and hence it uses the same area minimization heuristic as the original R-Tree. Consequently, overlap can still occur when MBRs are expanded in the `chooseLeaf` function.

2.2.3 NIR-Tree

The NIR-Tree [22] is an in-memory R-Tree variant based on the R+-Tree that completely eliminates overlap in its constituent MBRs. It does so by fragmenting overlapping rectangles into polygons that are comprised of smaller, non-overlapping rectangles. As a result,

any overlapping region is fully disambiguated, which minimizes the number of spurious searches. Unlike the other R-Tree variants presented in this chapter, the NIR-Tree substitutes minimum bounding rectangles with minimum bounding polygons, which provide a more accurate representation of the data objects they enclose in addition to eliminating overlap.

Polygons in the NIR-Tree are formed out of the overlapping rectangles produced by the `chooseLeaf` function. Once an MBR has been chosen and expanded to include the new data point, the algorithm checks if the expanded MBR overlaps with any other sibling MBRs in the node. If so, the `fragment` method is called to break down the expanded MBR into smaller rectangles, constituting a bounding polygon that no longer overlaps with its siblings. Fig 2.4 shows an example MBR that gets expanded and then subsequently fragmented into a polygon during the `chooseLeaf` operation. Details on how the `fragment` algorithm produces polygons can be found in [22].

The `splitNode` function of the NIR-Tree uses the same downward split mechanism as the R+-Tree, ensuring that MBRs produced by splitting overfull nodes do not overlap. However, the choice of the split is different, as instead of using the entries of a node for candidate splits, the NIR-Tree computes a d-dimensional average point called the *geometric median* and uses it as the split line. For leaf nodes, the geometric median is computed by averaging all d-th coordinates of the points in the node. For branch nodes, the geometric median is computed by averaging all the d-th coordinates of the corners of each polygon in the node. The choice of the dimension d also varies: for leaf nodes, d is chosen to be the dimension along which the MBR of the node extends the most, while for branch nodes, d is chosen to be the dimension whose geometric median results in the least number of downward splits at the current level of the tree.

Thus, the NIR-Tree eliminates overlap produced by the `chooseLeaf` and `splitNode` functions by using non-overlapping polygons and downward splits, respectively. The NIR-Tree's polygons offer good search performance for point and highly selective range queries that are more likely to land on overlapping regions in other R-Tree variants. However, adapting the NIR-Tree's polygons to work on disk is challenging since it is difficult to maintain a static fanout on a disk page with dynamically-sized polygons. Subsequent chapters propose techniques to convert the NIR-Tree to an efficient disk-based index.

Chapter 3

NIR+-Tree Design

This chapter describes the design of the NIR+-Tree augmented with polygon overlays. The logical definition of the NIR+-Tree on disk is presented, followed by its concrete implementation. The polygon overlay data structure is then introduced. Finally, this chapter presents an encoding that exploits the geometric layout of polygons to reduce the memory usage of the polygon overlay.

3.1 Logical Layout

A brief description of the logical structure of the original NIR-Tree is necessary to outline the design of the NIR+-Tree in comparison.

The original NIR-Tree consists of branch and leaf nodes organized into a hierarchical structure. A NIR-Tree is configured with a fanout M, which is the number of branches or points contained in a branch or leaf node, respectively. A branch in turn contains a polygon representing the area covered by that branch and a child pointer pointing to the children of that branch. A branch node contains a pointer to its parent node and M branches. The polygons belonging to each branch in a branch node are disjoint, preventing redundant branch traversals during searches. Fig 3.1 illustrates an example branch node in the NIR-Tree. Similarly, a leaf node contains a pointer to its parent node and M data points.

The overall structure of the NIR+-Tree remains mostly the same as described above except for two modifications. These changes are required to efficiently map the NIR+-Tree to pages on disk.

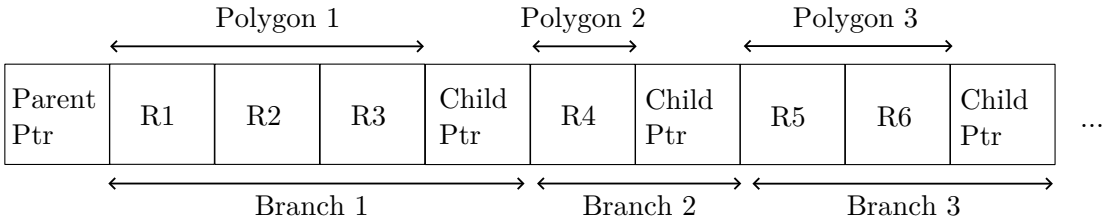


Figure 3.1: NIR-Tree branch node with three branches. Branches contain polygons with three, one and two rectangles respectively.

First, to maximize the data that can be stored in a page, parent pointers are removed from both branch and leaf nodes. Parent pointers are used in operations that traverse upwards, such as the `adjustTree` algorithm. Instead of pointers, the path of branches taken is tracked using a stack and used for such traversals. Second, instead of storing polygons, *reference rectangles* are stored in nodes. Reference rectangles are the smallest rectangle enclosing all the polygons of a branch node or all the points in a leaf node. The polygons themselves are moved out of the nodes and stored separately in the polygon overlay.

These terms are formally defined below:

Definition 1 *A branch B_i is a pair $(child_i, R_i)$, where $child_i$ denotes the pointer to the child of the branch and R_i denotes the reference rectangle of the branch. A branch B_i is also associated with a polygon P_i which is used to derive the reference rectangle R_i.*

Definition 2 *A branch node is a set of branches $\{B_1, \cdots, B_n\}$, where $1 \leq n \leq M$ and M is the configured fanout.*

Definition 3 *A leaf node is a set of points $\{p_1, \cdots, p_n\}$, where each $p_i \in \mathbb{R}^d$ for some dimension d, $1 \leq n \leq M$ and M is the configured fanout.*

A unique identifier is also defined for each branch in a NIR+-Tree, which is required by the polygon overlay data structure.

Definition 4 *Let B be the set of all branches in a NIR+-Tree. We define a function $f_{id} : B \mapsto \mathbb{N}$ that assigns each branch a unique integer value. Thus, each branch B_i in a NIR+-Tree has a unique integer id represented by $f_{id}(B_i)$.*

Definition 5 *A polygon overlay is a set of tuples $\{(f_{id}(B_1), P_1) \cdots (f_{id}(B_n), P_n)\}$ where $n \leq$ the total number of branches.*

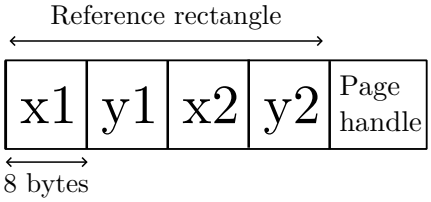


Figure 3.2: NIR+-Tree branch containing a reference rectangle with coordinates (x1, y1) and (x2, y2) and its page handle.

3.2 Physical Layout

This section describes the concrete implementation details of the NIR+-Tree. As previously mentioned, branch and leaf nodes in a NIR+-Tree must have their fanout M configured such that each node occupies an entire disk page. The internal details of NIR+-Tree nodes are presented and used to calculate a value for M. The size of a disk page is assumed to be 4KB in the discussion below.

Since the original NIR-Tree is in-memory, the child pointers in a branch node are virtual memory addresses. Most disk-backed indexes instead use page handles, which are integers that uniquely identify the pages stored on disk to refer to other nodes [12]. A lookup table is maintained that maps page handles to addresses in virtual memory. When a page is loaded from disk, this lookup table is updated with the address of the page in memory. Subsequently, the lookup table is consulted to translate the page handle into a memory address during insert or search operations.

Figure 3.2 illustrates the structure of a NIR+-Tree branch that uses page handles to refer to child nodes. A 2-dimensional rectangle can be represented using its lower left and upper right coordinates. Thus, a reference rectangle R consists of 4 double-precision floating-point values, using 32 bytes in total. Page handles in this implementation use 8 bytes. This results in a branch size of 40 bytes. Additionally, the first 20 bytes of a page are used to store metadata, giving a usable space of 4076 bytes in a 4KB page.

Thus, the maximum fanout M for a 4KB page is

$$M = \left\lfloor \frac{\text{Usable Space in Page}}{\text{Branch Size}} \right\rfloor$$

$$M = \left\lfloor \frac{4076}{40} \right\rfloor = \left\lfloor 101.9 \right\rfloor = 101$$

A leaf node in the NIR+-Tree is simply an array of points, each consuming 16 bytes in the 2-dimensional case. Note that since leaf nodes store points and not rectangles, it is possible to increase the fanout of leaf nodes beyond that of branch nodes. For example, the fanout for a leaf node can be set to $\lfloor 4076/16 \rfloor = 254$. However, in practice both branch and leaf nodes are configured to have the same fanout to keep implementations simple.

3.3 Polygon Overlay

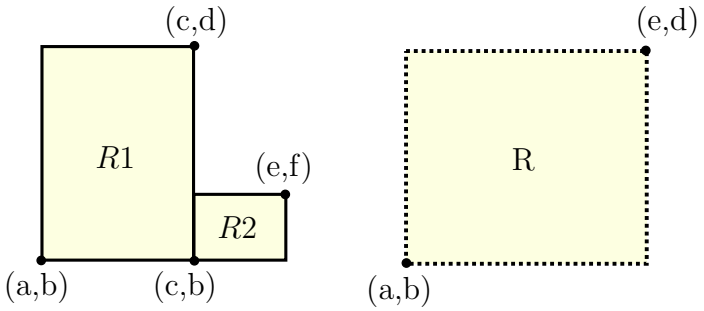
The polygon overlay is a data structure that stores the polygons associated with each branch in a NIR+-Tree. When executing a search, it is possible to use the reference rectangle in each branch to determine whether the search area overlaps with the branch. This however can lead to false positives, as the reference rectangle can overlap with the reference rectangles in other branches. To disambiguate the overlapping region, the polygon associated with each branch is consulted to determine whether the branch actually contains the search area or not.

Hash tables are used to concretely implement polygon overlays such that branches are mapped to their associated polygons. Since each branch has a unique integer in the form of its page handle, this is used as the key for the hash table. Therefore, for a given branch B_i, $f_{id}(B_i) =$ the page handle of B_i. This allows the NIR+-Tree to perform an $O(1)$ expected lookup to fetch the polygon associated with a branch during a search.

3.3.1 Polygon Encoding

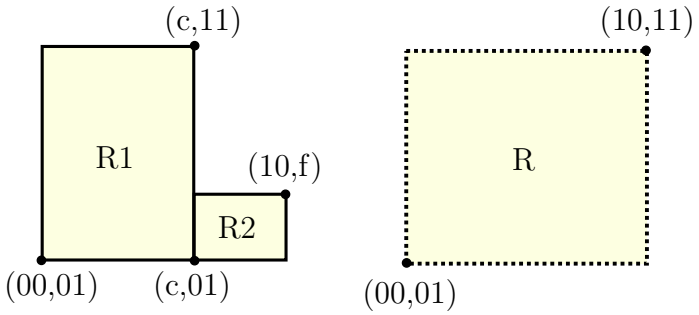
In datasets that exhibit high overlap, most branches of the NIR+-Tree can have polygons stored in the overlay. Furthermore, there is no limit to the number of rectangles that can make up a polygon, and a high degree of overlap can induce complex polygons. As a result, the overlay can consume a significant amount of memory.

To address this issue, an encoding for efficiently storing polygons in the overlay is proposed, similar to the clip points representation in Clipped Bounding Boxes [38]. The encoding exploits the fact that the coordinates of a polygon and its reference rectangle can have the same values depending on their geometric layout. For example, consider the coordinates of polygon P consisting of two rectangles $R1$ and $R2$ in Figure 3.3a. $R1$'s lower left coordinate is (a, b) while its upper right coordinate is (c, d). Similarly, $R2$ is represented by (c, b) and (e, f). Now consider the reference rectangle R in Figure 3.3b. Its lower left coordinate is (a, b) and its upper right coordinate is (e, d). Since the reference rectangle R



(a) Polygon $P = \{R1, R2\}$ (b) Reference rectangle R

Figure 3.3: Example of coordinate sharing between a polygon and its reference rectangle.



(a) Encoded polygon P (b) 2-bit encoding scheme

Figure 3.4: Encoding scheme for polygons using reference rectangles.

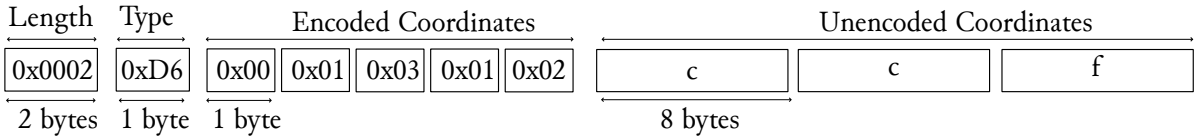


Figure 3.5: Serialized byte array for the polygon in Figure 3.4a.

is stored in the branch node, it is possible to encode the coordinates of the polygon P by referring to matching coordinates of R. This gives us a more compact representation of P that can be stored in the overlay.

To represent the four points of a reference rectangle R, only a 2-bit encoding scheme is required. For example, suppose the lower left coordinates of R are encoded using (00, 01) and the upper right coordinates of R are encoded using (10, 11), as shown in 3.4b. Matching coordinates in the polygon P can now be replaced using this encoding. In this implementation, double-precision floating-point values are used to represent coordinates on a plane. By applying the encoding scheme described above, it is possible to substitute an 8-byte double with a 2-bit representation. In practice, a full byte is used to store the 2-bit representation to keep implementations simple. Thus, before applying the encoding, storing the 8 coordinates of P using doubles takes 64 bytes. After applying the encoding, 5 of the coordinates can be represented using the 2-bit scheme, while the remaining 3 coordinates are stored as doubles, as shown in Figure 3.4a. This gives us a representation that uses only $5 \cdot 1$ byte $+$ $3 \cdot 8$ bytes $= 29$ bytes, reducing memory usage by 54%.

Given a polygon P and its reference rectangle R, P is serialized into a byte array containing metadata along with encoded and unencoded coordinates. To keep track of how many coordinates are contained in the byte array, a 16-bit integer is prepended, indicating the number of rectangles in P. This allows the NIR+-Tree to use polygons containing up to $2^{16} = 65{,}536$ rectangles, which is more than sufficient for most datasets. Then, to identify which coordinates are encoded and which are not, a bit array is used, representing type information. For each coordinate, one bit is used to indicate whether the coordinate is encoded or not. A set bit indicates that the coordinate is encoded, while an unset bit indicates that the coordinate is unencoded. The type bits first represent the x and y coordinates of the lower left corner and then the x and y coordinates of the upper right corner of each rectangle. Thus, for $R1$ and $R2$ in Figure 3.4a, the type bits are 1101 0110 (0xD6). Following the type information, the encoded and unencoded coordinates are appended to the byte array. Fig 3.5 shows the serialized byte array for the polygon in Figure 3.4a.

Let E and U be the number of encoded and unencoded coordinates in a polygon P

respectively; this gives us the number of bits in the serialized byte array as:

$$\begin{aligned} &= \text{Length Bits} + \text{Type Bits} + \text{Coordinates} \\ &= 16 + (E + U) + (8 \cdot E + 64 \cdot U) \\ &= 16 + 9 \cdot E + 65 \cdot U \end{aligned}$$

For the polygon in Figure 3.4a, $E = 5$ and $U = 3$, giving us a total of $16+9\cdot5+65\cdot3 = 256$ bits or 32 bytes as the final size of the byte array.

Interpreting the serialized byte array requires us to first parse the number of rectangles in P from the first 16 bits. The deserialization then iterates through each bit in the type information to parse the encoded and unencoded coordinates. Two pointers are maintained, one to the start of the encoded coordinates and one to the start of the unencoded coordinates. For a set bit in the type information, 1 byte is parsed from the encoded coordinates and substituted with the appropriate coordinate from the reference rectangle of P. For an unset bit, 8 bytes are parsed from the unencoded coordinates, using the coordinate as is in the deserialized representation. This process continues until all the coordinates have been parsed, reconstructing the original polygon P from the serialized byte array.

Chapter 4

NIR+-Tree Algorithms

This chapter describes the algorithms used for inserting and searching data in the NIR+-Tree. In particular, update and search algorithms must reference NIR+-Tree nodes using page handles and utilize the overlay for polygon-related operations.

4.1 Inserts

The algorithms used for inserting data into the NIR+-Tree must also include the additional step of manipulating the polygon overlay. Methods are first presented for retrieving and updating polygons in the overlay.

The `getPolygonFromOverlay` (Algorithm 1) method first checks if the branch has a polygon associated with it in the overlay. If so, it returns the polygon. Otherwise, it returns the reference rectangle of the branch. The `insertPolygonInOverlay` (Algorithm 2) method inserts the polygon associated with a branch only if it constitutes two or more rectangles. This is because a polygon with only one rectangle is equivalent to the reference rectangle of the branch, and does not need to be stored in the overlay.

The method `getNodeUsingHandle` is used to retrieve a node from disk using a page handle. To generate reference rectangles from polygons, the `getReferenceRectangle` method is used as is from the original NIR-Tree [23].

Algorithm 1 getPolygonFromOverlay(NIR+-Tree T, Branch B) $\rightarrow$ Polygon P

1: $P \leftarrow T.Overlay[B.handle]$
2: **if** $P \neq \emptyset$ **then**
3: **return** P
4: **else**
5: **return** $B.R$
6: **end if**

Algorithm 2 insertPolygonInOverlay(NIR+-Tree T, Branch B, Polygon P)

1: **if** $|P.rectangles| \geq 2$ **then**
2: $T.Overlay[B.handle] \leftarrow P$
3: **end if**

When the insert function is called on the NIR+-Tree, the first step is to select a leaf node to store the new point, handled by the `chooseLeaf` method (Algorithm 3). Starting from the root node handle, each node to be examined must be retrieved from disk using `getNodeUsingHandle`. Subsequently, if the node is a leaf node, the algorithm returns immediately. Otherwise, each branch in the branch node is checked to see if the point is contained in the polygon associated with the branch (line 4). The polygons themselves are retrieved from the overlay using `getPolygonFromOverlay` (line 4). If the point is contained in the polygon, the algorithm continues its search on the selected branch node (lines 5-6). Otherwise, the algorithm selects the branch that has the polygon with the smallest area increase for the point and expands it. Finally, the algorithm checks if the expanded polygon overlaps with any other polygon in the branch node. If so, the expanded polygon is fragmented into non-overlapping rectangles using the original NIR-Tree's `fragment` method (line 14). The new polygon is inserted in the overlay and the branch is updated with a new reference rectangle (lines 15-16). The algorithm then recurses on the selected branch node, repeating the process until a leaf node is reached. Furthermore, the path of branches taken from the root node to the leaf node is stored in a stack S for each level traversed in the NIR+-Tree.

Once a point has been inserted into the appropriate leaf node, the `adjustTree` method (Algorithm 4) is called to ensure that the NIR+-Tree satisfies the maximum fanout M. It uses the stack S returned by `chooseLeaf` to traverse upwards, instead of parent pointers (lines 4). If the fanout of a node exceeds M, the node is split using `splitNode` to create two new nodes with branches B_L and B_R, each of which have fanout $< M$ (line 6). The new branches are inserted into the parent node and their polygons are inserted into the

Algorithm 3 chooseLeaf(NIR+-Tree T, Point p) $\rightarrow$ Node N, Stack S

```
1: Node N ← getNodeUsingHandle(T.root.handle)
2: Stack S = {T.root.handle}
3: while N is not a leaf node do
4:     if ∃B ∈ N.branches such that p ∈ P ← getPolygonFromOverlay(T, B) then
5:         N ← getNodeUsingHandle(B.handle)
6:         S.push(B.handle)
7:     else
8:         Find B such that P ← getPolygonFromOverlay(T, B) has the smallest
9:         area increase for p among all of N.branches.
10:        Expand P to include p.
11:        Retrieve all polygons P′ from the overlay for each branch B′ ∈ N.branches
12:        such that P and P′ overlap.
13:        for ∀P′ do
14:            P = fragment(P′, P)
15:            insertPolygonInOverlay(T, B, P)
16:            B.R ← getReferenceRectangle(P)
17:        end for
18:        N ← getNodeUsingHandle(B.handle)
19:        S.push(B.handle)
20:    end if
21: end while
22: return N, S
```

Algorithm 4 adjustTree(NIR+-Tree T, Stack S)

Require: $\exists M$ a maximum fanout

```
1: Branch B ← S.pop()
2: while B ≠ ∅ do
3:     Node N ← getNodeUsingHandle(B)
4:     Branch B_P ← S.pop()
5:     if |N.branches| > M or |N.points| > M then
6:         (B_L, P_L), (B_R, P_R) ← splitNode(N, partitionNode(N))
7:         B_L.R ← getReferenceRectangle(P_L)
8:         B_R.R ← getReferenceRectangle(P_R)
9:         if B_P ≠ ∅ then
10:            Node N_P ← getNodeUsingHandle(B_P)
11:            N_P.branches ← N_P.branches \ {B}
12:            T.Overlay[B.handle] = ∅
13:            N_P.branches ← N_P.branches ∪ {B_L, B_R}
14:            insertPolygonInOverlay(T, B_L, P_L)
15:            insertPolygonInOverlay(T, B_R, P_R)
16:        else
17:            Node N' ← (B_L, B_R)
18:            T.root ← N'
19:        end if
20:    end if
21:    B ← B_P
22: end while
```

overlay (lines 10-15). If the root node was split, a node containing the two new branches is created and set as the new root of the NIR+-Tree (lines 17-18). The algorithm then traverses up the tree (line 21) until all nodes in the path with fanout $> M$ have been split.

The `splitNode` method takes a node N whose fanout exceeds M and splits it into two new nodes N_L and N_R, each of which have fanout $< M$. The line to split the node is obtained using the `partitionNode` method, which computes the geometric median of the points or branches in the node. The `splitNode` and `partitionNode` methods are identical to the original NIR-Tree implementation and can be found in [23]

4.2 Searches

The section presents algorithms used for executing point or range searches in the NIR+-Tree. The `search` method is initially called on the root node of the NIR+-Tree and takes as argument the point or rectangle being queried.

For point searches (Algorithm 5), if the current node being searched is a branch node, the algorithm iterates over each branch in the node (line 3) and checks if the point is contained in the polygon associated with the branch (line 5). If so, the algorithm recurses on the selected branch node (line 7). Since the NIR+-Tree guarantees that the polygons in a branch node are disjoint, the point being searched is assured to be contained in at most one polygon. As a result, the algorithm can stop iterating over branches once it finds a polygon that contains the point being queried (line 8). If the current node being searched is a leaf node, the algorithm iterates over each point in the node and returns the data associated with a point if it matches the point being queried (lines 12-19).

Range searches (Algorithm 6) are executed similarly, except that an accumulator is used to store the points that are contained in the range (line 1). If the current node being searched is a branch node, the algorithm iterates over each branch in the node (line 3) and checks if the range intersects with the polygon associated with the branch (line 5). Unlike point searches, a range search must iterate over all branches in a branch node, since the range being queried can overlap with multiple polygons, even if they are disjoint. If the current node being searched is a leaf node, the range is checked against each point in the node (line 12). Points contained in the range are added to the accumulator (line 14) which is returned after the search is complete.

Algorithm 5 search(NIR+-Tree T, Node N, Point p) $\rightarrow$ Data D

```
1: D = ∅
2: if N is a branch node then
3:     for B ∈ N.branches do
4:         P ← getPolygonFromOverlay(T, B)
5:         if p ∈ P then
6:             N ← getNodeUsingHandle(B.handle)
7:             search(T, N, p)
8:             break
9:         end if
10:    end for
11: end if
12: if N is a leaf node then
13:    for p′ ∈ N.points do
14:        if p = p′ then
15:            D = data associated with p′
16:            break
17:        end if
18:    end for
19: end if
20: return D
```

Algorithm 6 search(NIR+-Tree T, Node N, Rectangle R) $\rightarrow$ Accumulator A

```
1: A = ∅
2: if N is a branch node then
3:     for B ∈ N.branches do
4:         P ← getPolygonFromOverlay(T, B)
5:         if R ∩ P ≠ ∅ then
6:             N ← getNodeUsingHandle(B.handle)
7:             A = A ∪ search(T, N, R)
8:         end if
9:     end for
10: end if
11: if N is a leaf node then
12:     for p ∈ N.points do
13:         if p ∈ R then
14:             A = A ∪ {p}
15:         end if
16:     end for
17: end if
18: return A
```

Chapter 5

Bulk Loading

Previous chapters presented the design as well as the search and update algorithms of the NIR+-Tree. This chapter examines bulk-loading algorithms for the NIR+-Tree. Bulk-loading algorithms allow the NIR+-Tree to quickly and efficiently index large datasets. An overview of bulk-loading techniques for R-Trees is first presented. Subsequently, the two existing classes of bulk-loading algorithms in the literature are described and evaluated on their suitability for bulk-loading the NIR+-Tree.

5.1 Overview

In most applications involving spatial data, it is common to have a large dataset that is known a priori. For example, map applications utilize a large database of roads and landmarks which needs to be indexed for efficient retrieval. Such datasets are typically static and are infrequently updated. In such cases, it is more efficient to *bulk-load* the index from the entire dataset at once rather than inserting each record individually. By leveraging the knowledge of the entire dataset before construction, bulk-loading can create indexes with better space utilization and geometric layout than indexes created by sequential inserts.

Bulk-loading algorithms for R-Trees follow a common blueprint: they typically sort the given dataset using an ordering and then create branch or leaf nodes incrementally at each level by grouping together entries. Since each level of the tree is fully constructed before the next level is created, the nodes of a bulk-loaded R-Tree have close to 100% occupancy.

Bulk-loading algorithms for R-Trees differ in the ordering used to sort the dataset as well as the grouping strategy used to create nodes. Unlike single-dimensional data, multi-

dimensional data can be ordered in multiple ways. For example, a dataset of 2D points can be sorted by their x-coordinate, y-coordinate, or by their distance to another point, such as the centroid value of the input dataset. Sorting the input dataset using a particular ordering places entries that are close to each other in the same node. This is useful for spatial data retrieval, as it is likely that neighboring data entries will need to be accessed together during range searches.

Grouping strategies for bulk-loading algorithms can be classified into two categories: bottom-up and top-down.

A bottom-up bulk-loading algorithm constructs an index by gathering the N sorted input data points and grouping them into leaf nodes containing M points each, where M is the fanout of the tree. This results in $\lceil N/M \rceil$ leaf nodes created at the first level. It then recursively gathers the $\lceil N/M \rceil$ leaf nodes and groups them into branch nodes containing M nodes each, producing $\lceil N/M^2 \rceil$ branch nodes at that level. This process then recursively continues upwards until the root node is formed. Let the height of the bulk-loaded tree be $h = \lceil \log_M N \rceil$; we then have that the number of children in the root node to be $\lceil N/M^{h-1} \rceil$.

Top-down algorithms take the opposite approach: They start by partitioning the N data points into nodes containing M^{h-1} points each. These $\lceil N/M^{h-1} \rceil$ nodes are then designated to be the children of the root node of the R-Tree. Each of these child nodes are then recursively split to create nodes containing M^{h-2} points each. The recursive split continues until $\lceil N/M \rceil$ leaf nodes containing M points each are created, at which point the bulk-loading process is complete.

While bulk-loading is a one-time step used to index a given dataset, it is also not uncommon to update the index with new data after it has been bulk-loaded. A bulk-loaded index exhibits good space utilization and geometric layout, however, subsequent insertions into the index can weaken these properties. Therefore, while bulk-loading algorithms for R-Trees can be used for constructing NIR+-Trees, they must be modified to take into account the non-intersecting property of the MBRs in the NIR+-Tree. In particular, a bulk-loading algorithm suitable for the NIR+-Tree must strive to produce MBRs with reduced overlap, so that polygons need not be extensively created in the initial bulk-loading step. This is crucial if the index is to be updated with new data in the future, as insertions can worsen the overlap between MBRs and cause the polygons to grow large in size.

To find a suitable bulk-loading algorithm for the NIR+-Tree, we discovered that grouping strategies can have a significant impact on the degree of MBR overlap. In particular, top-down approaches avoid overlap among its MBRs better than bottom-up approaches. In the next sections, bottom-up and top-down bulk-loading algorithms are described in detail using examples of both. These algorithms are then examined on their effectiveness

at producing MBRs with reduced overlap.

5.2 Bottom-Up Bulk-Loading

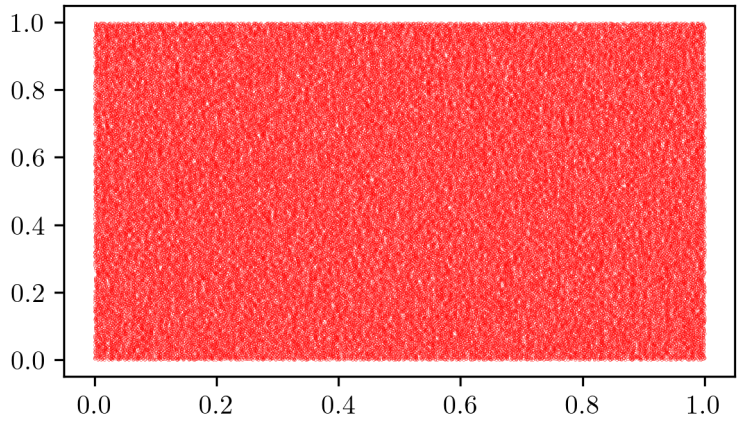


Figure 5.1: One million uniformly distributed points on a 1×1 grid.

Leutenegger et al. [24] provide a general framework for constructing disk-based R-Trees using bottom-up bulk-loading algorithms:

1. Sort the input point dataset using a particular ordering, such that the N points are grouped into $\lceil N/M \rceil$ groups of M points each, where each group of M points is to be placed into the same leaf node.

2. Place each $\lceil N/M \rceil$ group of points into a page and output the (MBR, page handle) pair for each group. The page handles will be used as child pointers by the nodes in the next level of the tree.

3. Recursively process all such MBRs at the leaf level into nodes at the next level. Continue processing each level until the root node is formed.

Examples of bottom-up bulk-loading algorithms include Nearest-X [35], Hilbert-Sort [19] and Sort-Tile-Recursive [24]. Each of these algorithms follow the steps described above, only differing in the ordering used to initially sort the input dataset. When handling point data, the bulk-loading algorithm must specify how to order points at the leaf level as well

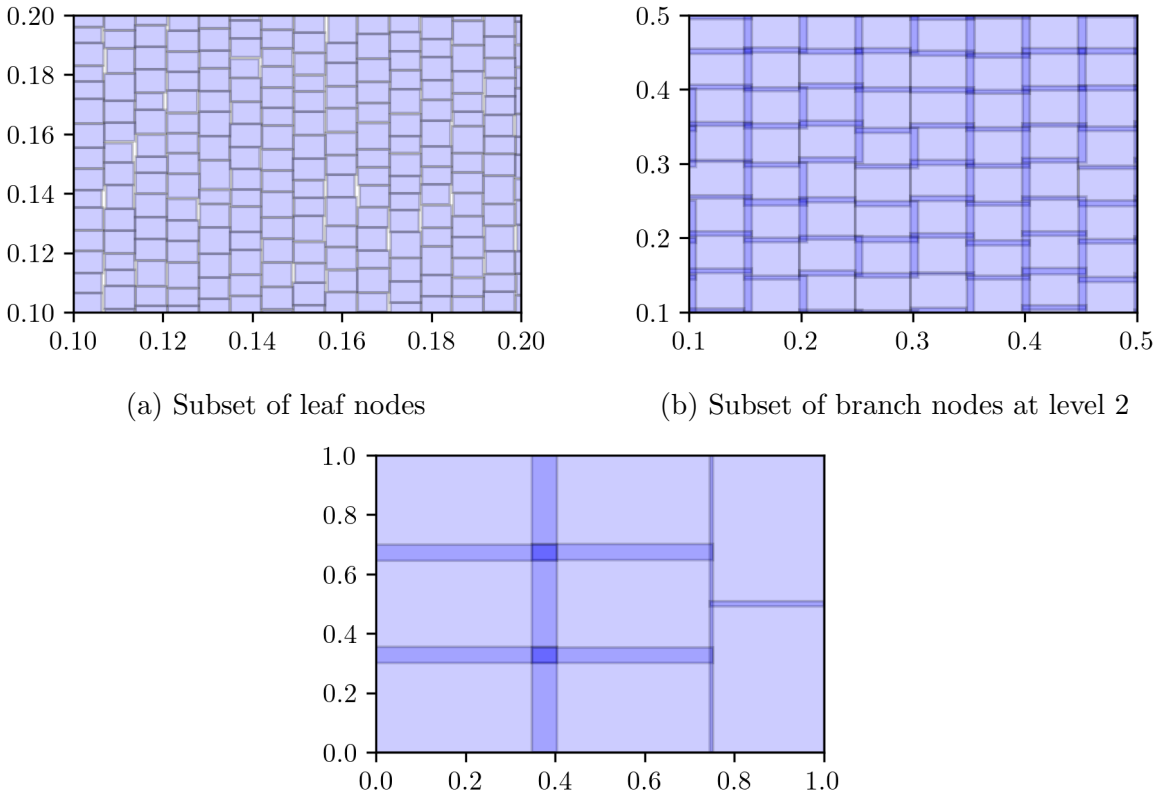
as the MBRs at the branch levels. The orderings used in these algorithms are described below:

- Nearest-X (NX): Data points are sorted by their x-coordinate while MBRs are sorted by the x-coordinate of their center.
- Hilbert-Sort (HS): Data points are sorted by their Hilbert value while MBRs are sorted by the Hilbert value of their center. Note that MBRs can be sorted in other ways, such as mapping the MBR into four dimensional space using its lower left and upper right corner points and then sorting by the Hilbert value of this four dimensional point.
- Sort-Tile-Recursive (STR): Data points are initially sorted by their x-coordinate, then tiled into $S = \lceil \sqrt{N/M} \rceil$ vertical slices each. The points in each slice are then sorted by their y-coordinate and grouped into nodes of M points each. The process is executed recursively for MBRs of branch nodes using their center point for sorting and tiling.

STR is the most widely used bulk-loading algorithm, due to its superior performance over NX and ease-of-implementation over HS. Hence, STR is used as the representative bottom-up bulk-loading algorithm for experiments. To evaluate the viability of STR for bulk-loading the NIR+-Tree, a dataset of one million uniformly distributed points is generated on a 1×1 grid, as shown in Figure 5.1. STR is then used to bulk-load this dataset and examine the geometric layout of the resulting index. In the experiment, the fanout is configured to be $M = 50$, which results in a tree height $h = 4$. The leaf nodes are denoted to be at level 3, the branch nodes above to be at level 2, and so on with the root at level 0.

Figure 5.2a shows a subset of the leaf nodes generated by STR when bulk-loading one million uniformly distributed points. Due to the tiling mechanism of STR, the leaf nodes are well-packed into square-like MBRs. In general, spatial indexes prefer to structure their MBRs into square-like shapes due to numerous benefits [3]. Given a fixed area, a square has the smallest perimeter, and hence is easier to compactly organize into a given space. Furthermore, most range queries tend to be square-like in shape, and hence square-like MBRs satisfy a query more naturally than MBRs with irregular shapes. Finally, square-like MBRs in one level of the tree allow for smaller, square-like MBRs in the level above, which in turn produces MBRs with minimal dead space across the entire tree.

Observe that the leaf nodes in Figure 5.2a also have no overlap among them. Roussopoulos et al. [35] proved that given a finite set of points N, it is possible to generate



(a) Subset of leaf nodes

(b) Subset of branch nodes at level 2

(c) All branch nodes at level 1

Figure 5.2: MBRs of an R-Tree bulk-loaded using Sort-Tile-Recursive.

MBRs that group the points such that all MBRs are disjoint from each other. To understand why the leaf node MBRs in the STR-generated tree are disjoint, a brief sketch of this proof is provided. The proof first demonstrates that it is possible to transform each of the N points such that they all have unique x-coordinates by rotating the points around the origin by an angle α. After the rotation, sorting the points by their unique x-coordinate and grouping them in the same order will always produce disjoint MBRs. This is because each MBR consists of points that have a smaller x-coordinate than any successively created MBR, and hence the extent of each MBR cannot overlap with any other MBR.

Applying this proof to the one million uniformly distributed points in Figure 5.1, observe that these points already have unique x-coordinates, and hence STR automatically produces disjoint MBRs at the leaf level without the need for rotations.This is illustrated

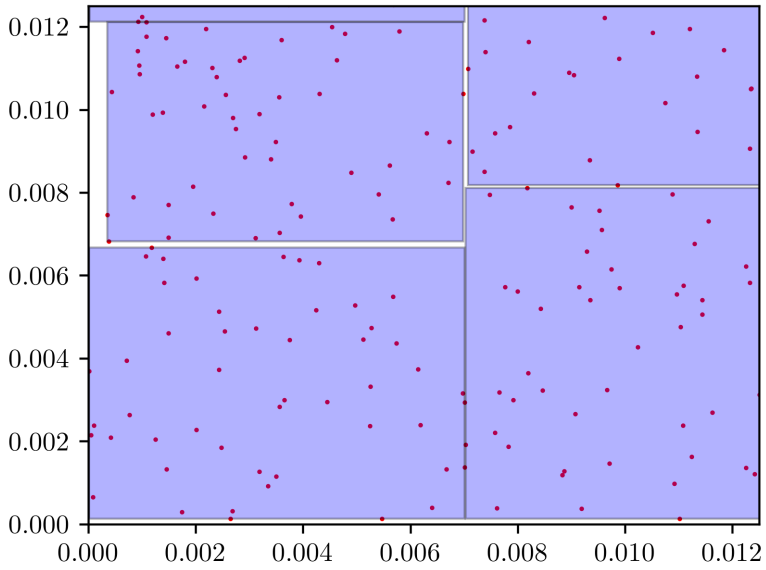


Figure 5.3: Close-up of the MBRs in Figure 5.2a, along with their constituent points.

in Figure 5.3, which shows a zoomed-in view of the leaf node MBRs produced by STR and the points they contain. Observe that by sorting and grouping points, each MBR encloses points in a way that prevents overlap with other MBRs. Therefore any bulk-loading algorithm that sorts the input dataset by the x-coordinate (or y-coordinate) of its points and groups them will produce leaf node MBRs that are disjoint, assuming that the points have unique x-coordinates (or y-coordinates). In practice, the number of unique x-coordinates (or y-coordinates) depends on the dataset itself, and this translates to the number of disjoint MBRs at the leaf level.

However, Roussopoulos et al. also showed that the zero overlap proof only applies to point objects and not rectangles [35]. Figure 5.2b shows a subset of the MBRs of branch nodes formed by grouping the MBRs of the leaf nodes. Observe that the branch node MBRs retain the square-like shape of their children. However, the MBRs of these branch nodes overlap with each other (indicated by the shaded region), despite the MBRs of the leaf nodes being disjoint. Unlike points, rectangles have non-zero area and sometimes cannot be grouped in a way that prevents overlap. Observe the same effect at the next level of the tree, as shown in Figure 5.2c.

Thus, while bottom-up algorithms ensure that leaf node MBRs are disjoint, they cannot guarantee that the MBRs of branch nodes at higher levels will be disjoint. We are able to

confirm the same by bulk-loading other datasets using STR and observing the resulting MBRs. In particular, skewed datasets with points clustered in specific regions can cause significant overlap in the MBRs of STR-generated trees.

However, the square-like shape of the MBRs generated by STR are highly desirable and can be useful for range searches. Furthermore, in theory, the overlap between MBRs could be eliminated by fragmenting the rectangles into polygons. Unfortunately, using STR to bulk-load the NIR+-Tree results in large polygon sizes and construction times. This is because STR produces MBRs with too much overlap, which causes the polygon fragmentation algorithm to dominate the execution time of the bulk-loading process. Moreover, the intricate overlap between MBRs produces complex polygon shapes that are substantial in size. Therefore, bottom-up algorithms are not suitable for bulk-loading the NIR+-Tree due to their inability to minimize overlap during index construction.

5.3 Top-Down Bulk-Loading

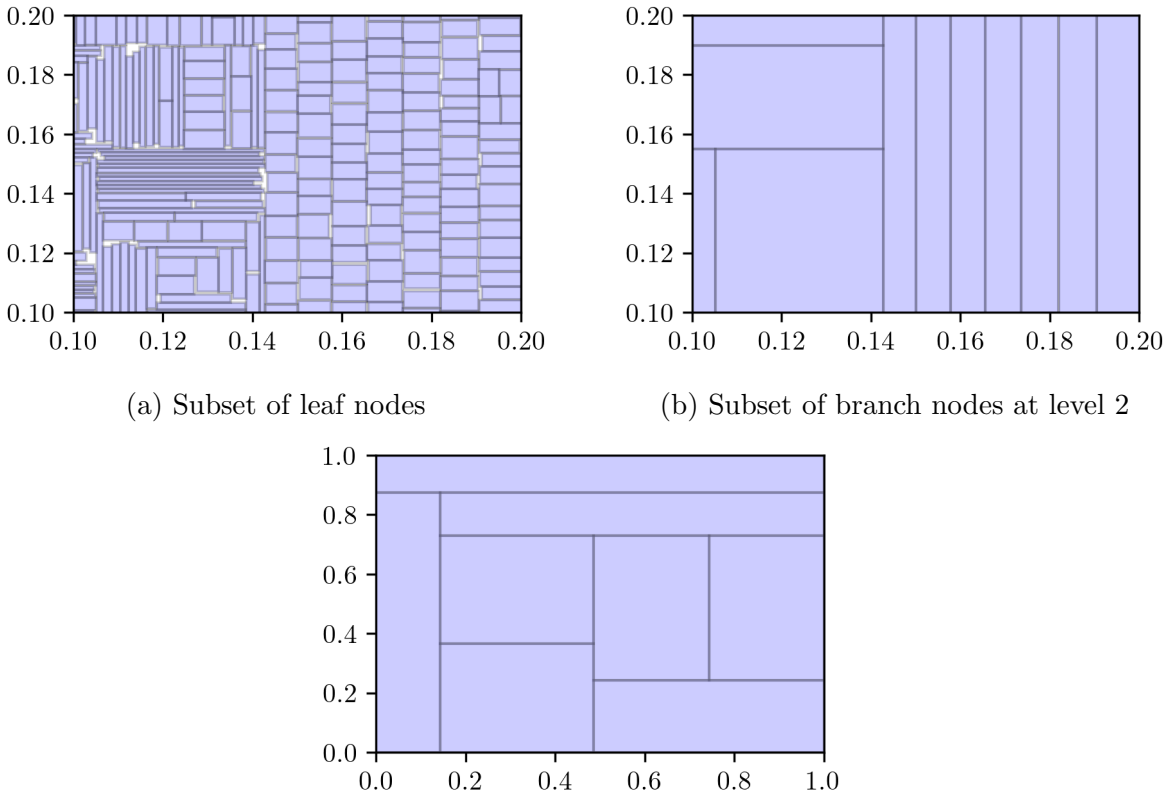
Top-down bulk-loading algorithms begin index construction starting from the root node and then recursively create nodes at each level until the leaf nodes are formed. In this section, we examine the Top-Down Greedy Splitting (TGS) algorithm [13], which is the most widely used top-down bulk-loading algorithm, and evaluate how well it avoids overlap.

TGS was created with the insight that constructing MBRs bottom-up leads to less control of the shape of MBRs at higher levels. MBRs at higher levels of the tree tend to be large and influence the search path of queries more than the lower levels. Therefore, optimizing the shape of MBRs at higher levels minimizes search cost better than MBRs at lower levels. Thus, TGS takes a top-down approach to constructing MBRs during bulk-loading. Furthermore, it uses a greedy strategy to partition data points into MBRs by evaluating multiple partitions across all dimensions and choosing one with the lowest cost. This is in contrast to most bulk-loading algorithms such as STR which uses simple partitioning strategies instead of incorporating cost-based partitioning. Moreover, the cost function is user-specified, which can be tuned to generate MBRs that optimize for area, perimeter, overlap or any other metric of choice.

A description of the TGS algorithm is provided below:

1. The algorithm starts by constructing the M children of the root node. First, compute C, the total number of points that the subtree of each child node will contain.

2. Then, for each dimension d:



(a) Subset of leaf nodes

(b) Subset of branch nodes at level 2

(c) All branch nodes at level 1

Figure 5.4: MBRs of an R-Tree bulk-loaded using Top-Down Greedy Splitting.

(a) Sort the points by their d-th coordinate.

(b) For i from 1 to $M - 1$:

i. Consider a cut at $C \cdot i$ that partitions the points into two groups $G1$ and $G2$, where $|G1| = C \cdot i$ and $|G2| = C \cdot (M - i)$.

ii. Group points in $G1$ and $G2$ into MBRs $B1$ and $B2$ respectively.

iii. Let f be the user-specified cost-function. Compute the cost of the cut $cost_i = f(B1, B2)$.

iv. If $cost_i$ is the lowest cost seen so far, record the cut as the choice of cut for dimension d.

3. Let c be the cut with the lowest cost across all dimensions. Partition the points into

two groups using c. Recursively compute cuts for each of the two groups by repeating step 2.

4. Proceed to compute cuts that minimizes the cost function until M partitions with C points are formed. Compute MBRs for each of the M partitions. At this point, the root node is fully constructed with M children.

5. Recursively partition the C points in each of the M children to construct the next level of the tree by repeating steps 1 to 4.

6. Continue constructing each level using cuts until the leaf nodes are formed.

TGS is used to bulk-load the same dataset of one million uniformly distributed points used in the previous section with fanout $M = 50$. The cost function is set to be $f(B1, B2) = perimeter(B1) + perimeter(B2)$, which selects the cut that produces MBRs with the smallest perimeter.

The MBRs produced by TGS are examined starting from the root of the tree. Figure 5.4c shows the MBRs of the branch nodes at level 1 of the bulk-loaded tree. Observe that unlike STR, the MBRs of level 1 branch nodes are disjoint. We see the same for the MBRs at level 2 and the leaf nodes, as shown in Figures 5.4b and 5.4a respectively. Thus, the top-down approach of TGS is quite effective at producing MBRs that do not overlap.

To understand why, we revisit the zero overlap proof [35] that showed that it is possible to generate disjoint MBRs by sorting points by their x-coordinate (or y-coordinate) and grouping them in the same order. Bottom-up algorithms like STR only apply this principle when constructing MBRs of leaf nodes. Consequently, in STR, the MBRs of branch nodes are shaped by the MBRs of their children, which offer no guarantee on the degree of overlap produced. In contrast, TGS always groups points into MBRs at every level of the tree in a top-down fashion. This allows TGS to leverage the zero overlap technique across the entire tree, producing MBRs that are disjoint.

Thus, TGS is a good candidate to use for bulk-loading the NIR+-Tree, since it produces MBRs with reduced overlap. As a result, there is little need for fragmenting overlapping rectangles into polygons, which can be computationally expensive. Moreover, this reduced overlap among MBRs translates to small polygon overlays. This is a desirable property since subsequent insertions into the bulk-loaded NIR+-Tree tree can increase the overlap between MBRs and produce large polygons. We empirically verify that TGS produces small polygon overlays by bulk-loading multiple datasets and applying polygon fragmentation on overlapping MBRs; the results show that very few polygons are needed to keep MBRs disjoint on average for TGS.

Chapter 6

Experimental Evaluation

This chapter evaluates the performance of the NIR+-Tree with the three competing spatial indexes introduced in Chapter 2: the R-Tree, R+-Tree and R*-tree. The evaluation benchmarks the insert and search performance of each index on four real-world 2D datasets. The STR and TGS bulk-loading algorithms mentioned in Chapter 5 are also evaluated on their search performance and their ability to minimize MBR overlap.

6.1 Experimental Setup

For all indexes, the maximum fanout is configured to be $M = 80$ and for indexes that require setting a minimum fanout (R-Tree and R*-Tree), $m = 40$, as suggested in prior work [4]. For the R*-Tree, the percentage of entries to be re-inserted is configured to be $s = 30\%$, which is the optimal setting mentioned in [3]. For the R+-Tree, the fill factor, which is the number of entries to be retained in a node during a split, is configured to be 50%. This effectively results in the R+-Tree always splitting on the median point of either the x-axis or y-axis. As discussed in Section 4.1, the NIR+-Tree uses the R+-Tree's downward-propagating split technique, except that it splits on the geometric median of a node. For each dataset, each R-Tree variant is built by inserting all points sequentially, after which a series of point or range queries are executed.

The search performance of the STR and TGS bulk-loading algorithms are also evaluated. Each dataset is bulk-loaded using STR and TGS, after which the same point and range queries are executed. TGS is configured to use the same cost function that optimizes for MBR perimeter mentioned in Section 5.3. Additionally, we examine the polygon overlay produced by applying polygon fragmentation to the MBRs produced by TGS.

Since the experiments in this section evaluate disk-based indexes, page access counts are used as the primary measure of performance. The number of pages accessed during an operation is a robust metric for evaluating the performance of disk-based indexes, as it is independent of buffer pool sizes, data layout, workload patterns and the underlying storage hardware. In the case of inserts, the total time taken to build the indexes completely in-memory is also reported, as a way of comparing the overhead of the NIR+-Tree's polygon construction algorithms.

6.1.1 Datasets

Four 2D spatial datasets from the UCR Spatio-Temporal Active Repository [15] are used for the evaluation. A description of each dataset follows:

- **TDrive**: A dataset containing one-week trajectory samples of 10,357 taxis in Beijing, China. The combined trajectory data consists of points mostly concentrated in the city center, with a subset of points scattered around the city outskirts.
  · Total number of points: 11,317,142
  · Average number of points per 0.1^2 grid: 2304.92

- **Tweets**: A collection of geo-tagged tweets across the United States. Points are concentrated in major urban areas and largely towards the eastern half of the country. A subset of points are located farther away from the mainland in Alaska and Hawaii.
  · Total number of points: 15,598,403
  · Average number of points per 0.1^2 grid: 321.03

- **GeoLife**: GPS trajectory data collected from 178 users over a period of four years from the GeoLife social network. Points are largely distributed across 30 cities in China, with a subset of points lying across a few cities in Europe and the United States.
  · Total number of points: 22,033,507
  · Average number of points per 0.1^2 grid: 2653.68

- **NYCTaxi**: A dataset representing all yellow cab pickup and drop-off points in New York City from January to December 2013. Due to the year-long sampling time-frame and small sampling area, points are heavily concentrated in the city, making this the largest and densest dataset used in experiments.
  · Total number of points: 81,616,580
  · Average number of points per 0.1^2 grid: 9301.03

6.1.2 Queries

The search performance of each index is evaluated using both point and range queries. For point queries, 100,000 random points are sampled from each dataset and then queried using each index. For range queries, rectangles are generated as described in [4]. First, 1000 random points are chosen from the input dataset and used as the center of a range query. For each such center point, a k-nearest neighbor query is run to find the k nearest neighbors to that point. A rectangle is then generated by bounding all the k points, giving us a range query that returns a fixed number of output. k is set to 10, 100, 1000 and 10000, generating four range query profiles with varying selectivity for evaluation.

6.2 NIR+-Tree Experiments

6.2.1 Inserts

Figure 6.1 shows the total number of pages accessed for building each index. Across all four datasets, the R*-Tree uses the most I/Os because of its re-insertion technique. Whenever a node is full, the R*-Tree re-inserts 30% of the entries in the node assuming this will result in a better reorganization of the entries. This results in the R*-Tree using roughly 2× to 3× as much I/O as its competitors. The R-Tree uses slightly more I/Os for inserts than the NIR+-Tree and R+-Tree because its splits have to satisfy a minimum fanout. Consequently, the R-Tree has better node utilization – after a split, each node is guaranteed to have at least $m = 40$ entries. However, this comes at the cost of poor insert performance, since it is likely that its nodes will have to be split again in the future. While the R+-Tree does not have a minimum fanout, it has a fill factor set to 50%, which creates split nodes with 50% utilization. However, the R+-Tree also propagates its splits downwards, splitting child nodes that are not full. This produces nodes with lower utilization at lower levels of the tree, but provides better insertion performance, since new inserts are less likely to cause splits. The NIR+-Tree and R+-Tree use roughly the same number of I/Os for inserts, since they use the same split technique, only differing in which point to split.

Figure 6.2 shows the total time taken to build each index in-memory. Similar to the I/O results, the R*-Tree takes the longest to construct, since it has to re-insert 30% of the entries in each node. Note that during an insertion, the R*-Tree uses an $O(n^2)$ algorithm for deciding which branch to insert a entry at each level of the tree. This algorithm is executed for each re-inserted entry, resulting in the R*-Tree taking significantly longer to

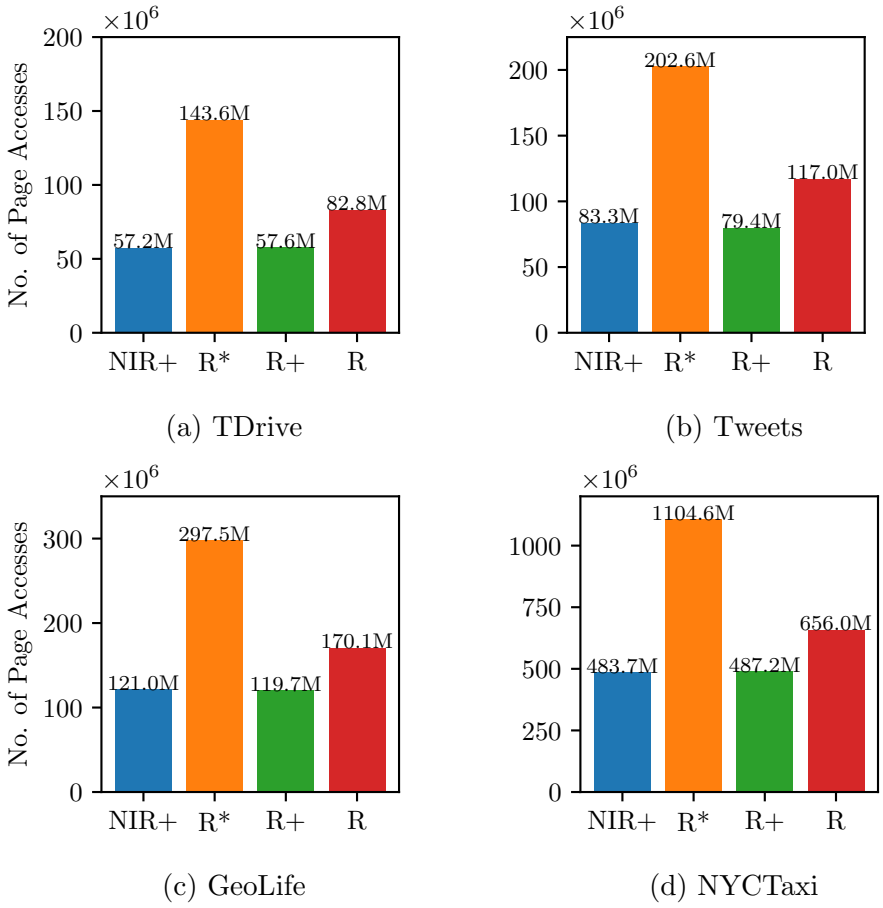
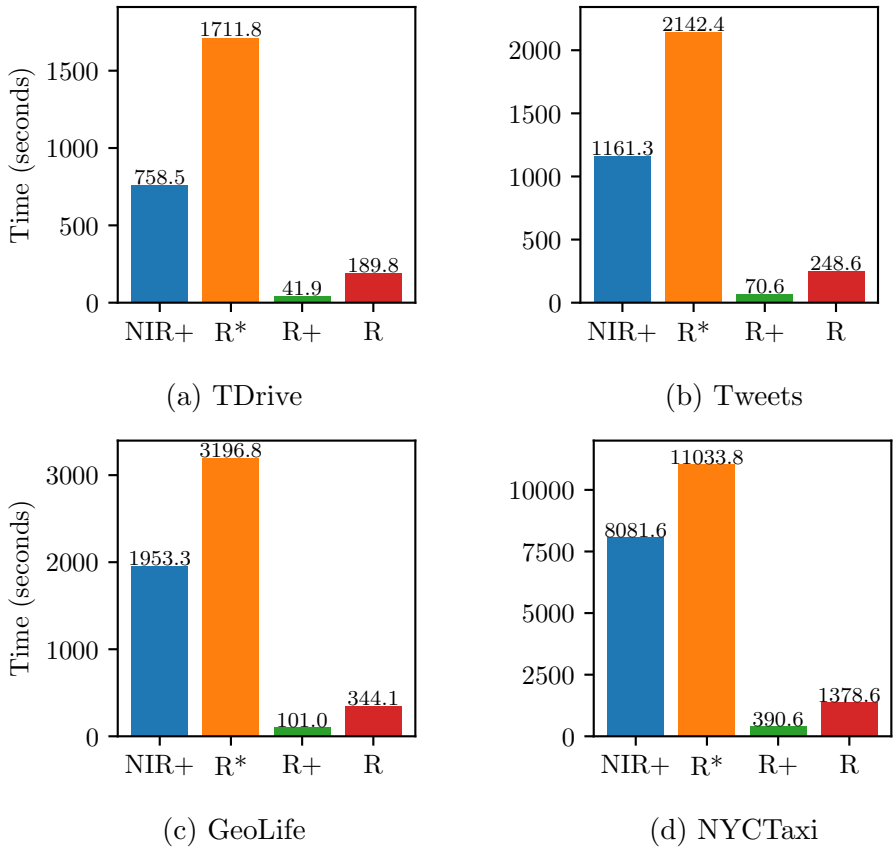


Figure 6.1: Total pages accessed for index construction

build than its competitors. The NIR+-Tree comes in second, since instead of using re-insertions, it constructs polygons for each overlapping node, which is CPU-intensive by nature. The R-tree and R+-Tree take the least amount of time to build, since they do not have to perform any additional computations during insertions.



(a) TDrive (b) Tweets

(c) GeoLife (d) NYCTaxi

Figure 6.2: Total time taken for index construction (seconds)

Dataset	NIR+	R*	R+	R
TDrive	4	4	4	5
Tweets	5	4	4	5
Geolife	5	5	5	5
NYCTaxi	5	5	5	5

Table 6.1: Tree height for all datasets

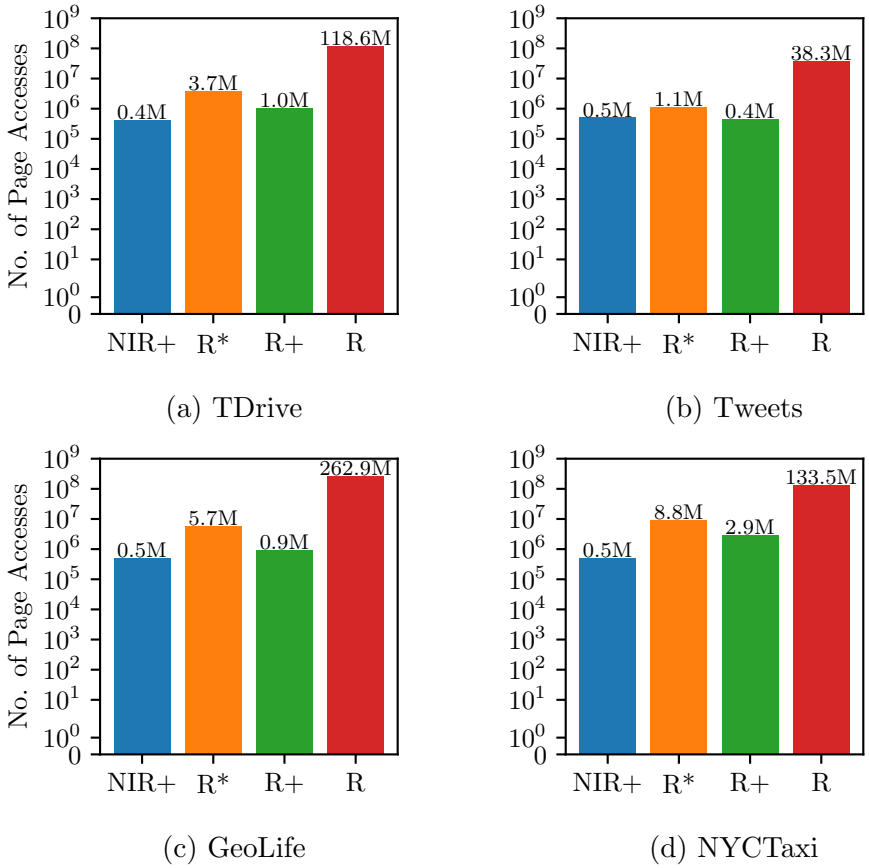


Figure 6.3: Total pages accessed for 100,000 point queries (log scale)

6.2.2 Searches

Point Searches

Figure 6.3 shows the number of pages accessed for executing 100,000 point queries on each index. Note that a point query is executed optimally when it does not intersect with any overlapping region in the tree and therefore uses I/Os exactly equal to the height of the tree. Table 6.1 shows the height of each tree for each dataset. Across all four datasets, the NIR+-Tree executed all 100,000 point queries optimally, using exactly 400,000 I/Os on the TDrive dataset and exactly 500,000 I/Os on Tweets, GeoLife and NYCTaxi. In all datasets, the NIR+-Tree's closest competitor was the R+-Tree. The NIR+-Tree uses roughly 2× less I/O than the R+-Tree on TDrive and GeoLife and roughly 6× less I/O on

NYCTaxi. However, on the Tweets dataset, the R+-Tree was able to beat the NIR+-Tree by a small margin of roughly 65,000 I/Os. This is because the median split of the R+-Tree works well on the Tweets dataset, resulting in the R+-Tree having a height of 4, while the NIR+-Tree has a height of 5. Therefore, even though the NIR+-Tree executed each point query optimally, the geometric median split of the NIR+-Tree resulted in a taller tree, causing it to use one more I/O per point query. Furthermore, Tweets is the least dense of all datasets, resulting in not as much overlap between MBRs for each index. As a result, polygon construction is not as effective on the Tweets dataset as it is on the other datasets. On the other hand, the NIR+-Tree was most effective on the NYCTaxi dataset against its competitors, which is the most dense of all four datasets.

Range Searches

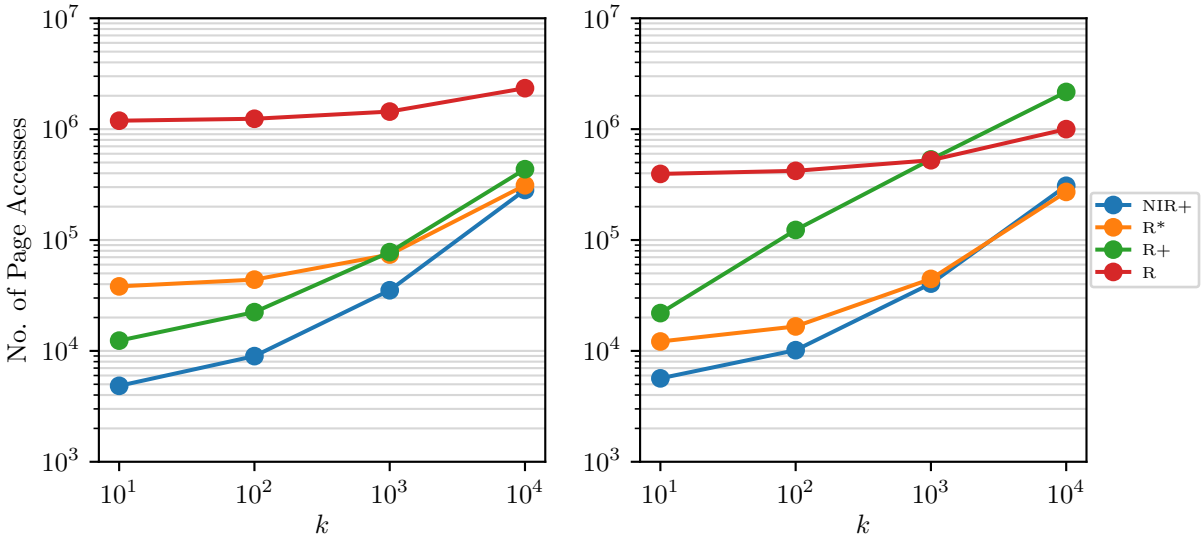
Figure 6.4 shows the I/Os used for executing range queries. Across all values of k, the NIR+-Tree was able to outperform or match the performance of its closest competitor, either the R+-Tree or the R*-Tree. For range queries with selectivity $k = 10^1$, the NIR+-Tree uses roughly 2.5× less I/O on TDrive and GeoLife and roughly 5× less I/O on NYCTaxi compared to the R+-Tree. On the Tweets dataset, the NIR+-Tree uses 2× less I/O than the R*-Tree.

As query selectivity decreases, the NIR+-Tree's performance advantage declines. This is because larger range queries are less impacted by overlap, since they are more likely to anyway intersect with multiple branches in a node. In particular, on the Tweets dataset, the performance of the NIR+-Tree closely matches that of the R*-Tree for $k \geq 10^2$. However, on the remaining datasets, the NIR+-Tree continues to outperform the R+-Tree by using 2× to 5× less I/O. Finally, the NIR+-Tree's performance advantage plateaus for range queries with $k = 10^4$, where it uses roughly the same number of I/Os as its nearest competitor. Table 6.2 summarizes the I/O improvements of the NIR+-Tree for range queries across all values of k.

6.2.3 Memory Usage

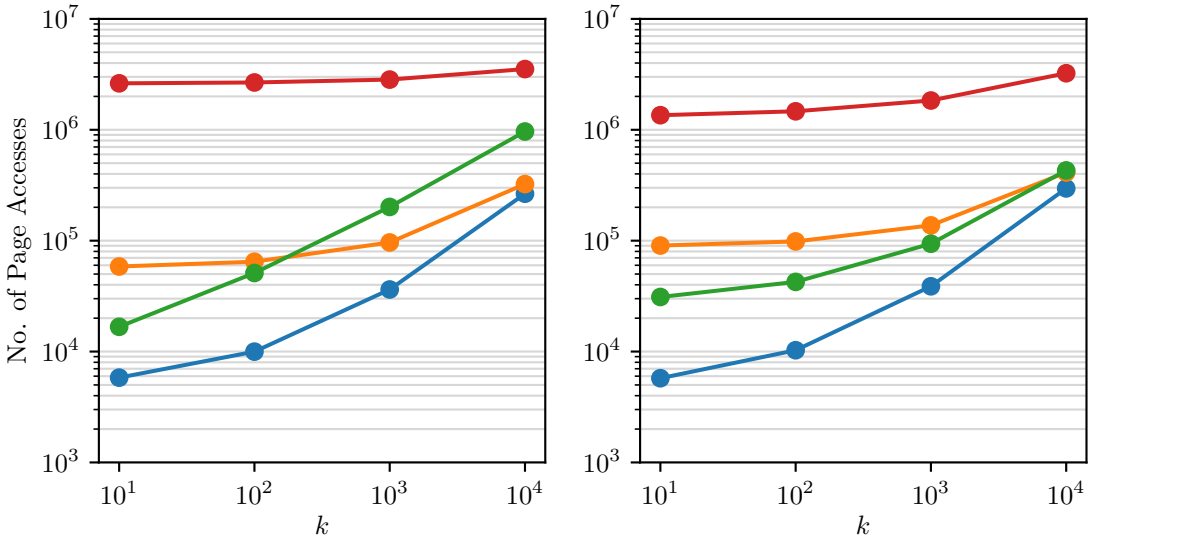
In this section, the memory usage of the NIR+-Tree is compared with its competitors. In particular, this section examines the overhead of the geometric median split of the NIR+-Tree and the polygon overlay.

Table 6.3 shows the size of each R-Tree variant after inserting all points in the four datasets. The R*-Tree uses the least amount of memory since it has to satisfy the minimum



(a) TDrive

(b) Tweets



(c) GeoLife

(d) NYCTaxi

Figure 6.4: Range query results (log-log scale)

fanout $m = 40$ and it performs re-insertions. Both allow for better organization of entries across the tree and packs nodes efficiently. The R-Tree comes in second, since it also has to

TDrive			
k	R*	R+	R
10^1	7.89×	**2.56**×	246.08×
10^2	4.90×	**2.49**×	138.24×
10^3	2.10×	**2.21**×	40.88×
10^4	**1.10**×	1.54×	8.28×

Tweets			
k	R*	R+	R
10^1	**2.15**×	3.88×	69.80×
10^2	**1.64**×	12.14×	41.45×
10^3	**1.10**×	13.18×	12.96×
10^4	**0.88**×	7.02×	3.24×

Geolife			
k	R*	R+	R
10^1	10.05×	**2.87**×	451.07×
10^2	6.47×	**5.12**×	267.72×
10^3	**2.66**×	5.55×	78.19×
10^4	**1.22**×	3.64×	13.31×

NYCTaxi			
k	R*	R+	R
10^1	15.72×	**5.40**×	235.59×
10^2	9.58×	**4.13**×	142.82×
10^3	3.54×	**2.42**×	47.38×
10^4	**1.39**×	1.45×	10.93×

Table 6.2: Range search I/O reduction of the NIR+-Tree against other indexes for each value of k. Nearest competitor is in bold.

satisfy a minimum fanout but does not perform re-insertions. The NIR+-Tree and R+-Tree have the largest tree sizes, due to their downward-propagating split technique. However, the geometric median split of the NIR+-Tree produces smaller tree sizes than the R+-Tree on all datasets, except for Tweets.

Figure 6.5 shows the memory usage of the polygon overlay with and without the encoding technique presented in Section 3.3.1. Across all datasets the size of the encoded polygon overlay never exceeds more than 3% of the total tree size. Therefore, the polygon overlay can be kept pinned in memory to refine searches. As the size of the dataset grows, so does the absolute amount of memory saved by the encoding technique. The encoding is most effective on the NYCTaxi dataset, where it reduces memory usage by $\approx 30\%$. For all other datasets, the encoding shrinks the size of the overlay by $\approx 25\%$. Observe that the effectiveness of the encoding decreases as the number of rectangles that make up a polygon increases. This is because the encoding only removes duplicate points for rectangles that share a common edge with the reference rectangle of the polygon. As the constituent rectangles in a polygon increases, there are fewer rectangles that share such a common edge, minimizing the benefits of the encoding.

Dataset	NIR+	R*	R+	R
TDrive	302	259	310	288
Tweets	442	357	372	398
Geolife	576	504	601	562
NYCTaxi	2108	1882	2189	2062

Table 6.3: Index sizes (MB)

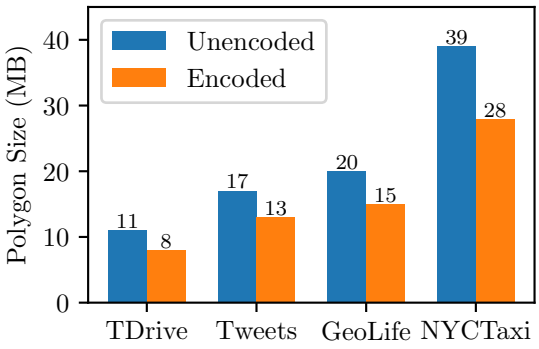


Figure 6.5: Memory usage of unencoded vs encoded polygons (MB)

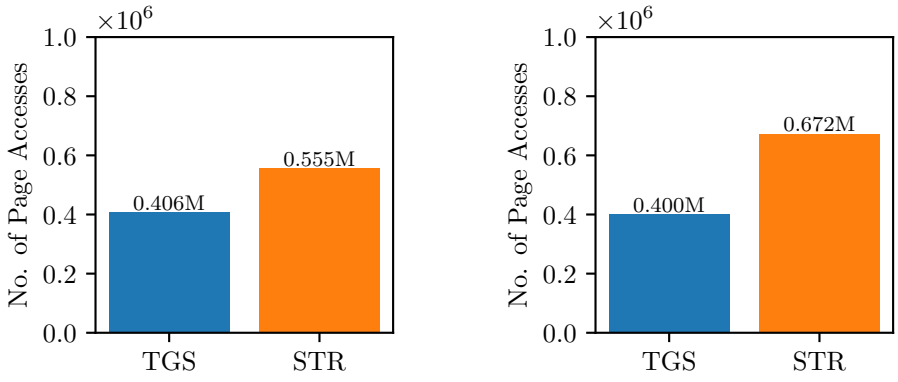
6.3 Bulk-Loading Experiments

This section evaluates the search performance of the TGS and STR bulk-loading algorithms. We empirically demonstrate that TGS produces indexes with less overlap than STR, resulting in better search performance for point and selective range queries. Note that index construction results are omitted, since TGS always takes roughly 4× longer than STR due to TGS's greedy split selection compared to the static splits of STR.

6.3.1 Searches

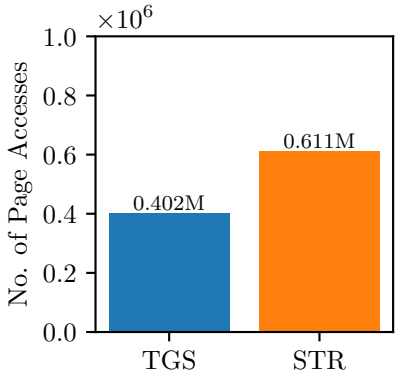
Point Searches

Figure 6.6 shows the I/O cost of executing 100,000 point queries for each bulk-loaded dataset. Bulk-loading algorithms strive to maximize the number of entries in each node to produce indexes with the lowest possible height and size. Thus, a bulk-loaded index with no overlap should execute point queries optimally, requiring I/Os equal to the height of the

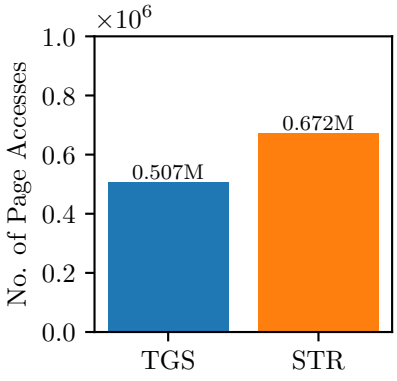


(a) TDrive

(b) Tweets



(c) GeoLife



(d) NYCTaxi

Figure 6.6: Total pages accessed for 100,000 point queries

Dataset	TGS/STR Height
TDrive	4
Tweets	4
Geolife	4
NYCTaxi	5

Table 6.4: Bulk-loaded index height for TGS and STR across all datasets

index. Table 6.4 shows the height for each bulk-loaded dataset. Across all four datasets, the TGS algorithm constructs indexes with little to no overlap among its MBRs compared

to STR. As a result, TGS executes each point query close to optimal, using 4.06, 4.00, 4.01 and 5.06 I/Os per query on TDrive, Tweets, GeoLife and NYCTaxi respectively. On the other hand, the STR algorithm does not eliminate overlap as effectively as TGS, and thus uses 5.55, 6.72, 6.11 and 6.71 I/Os per query on the same datasets.

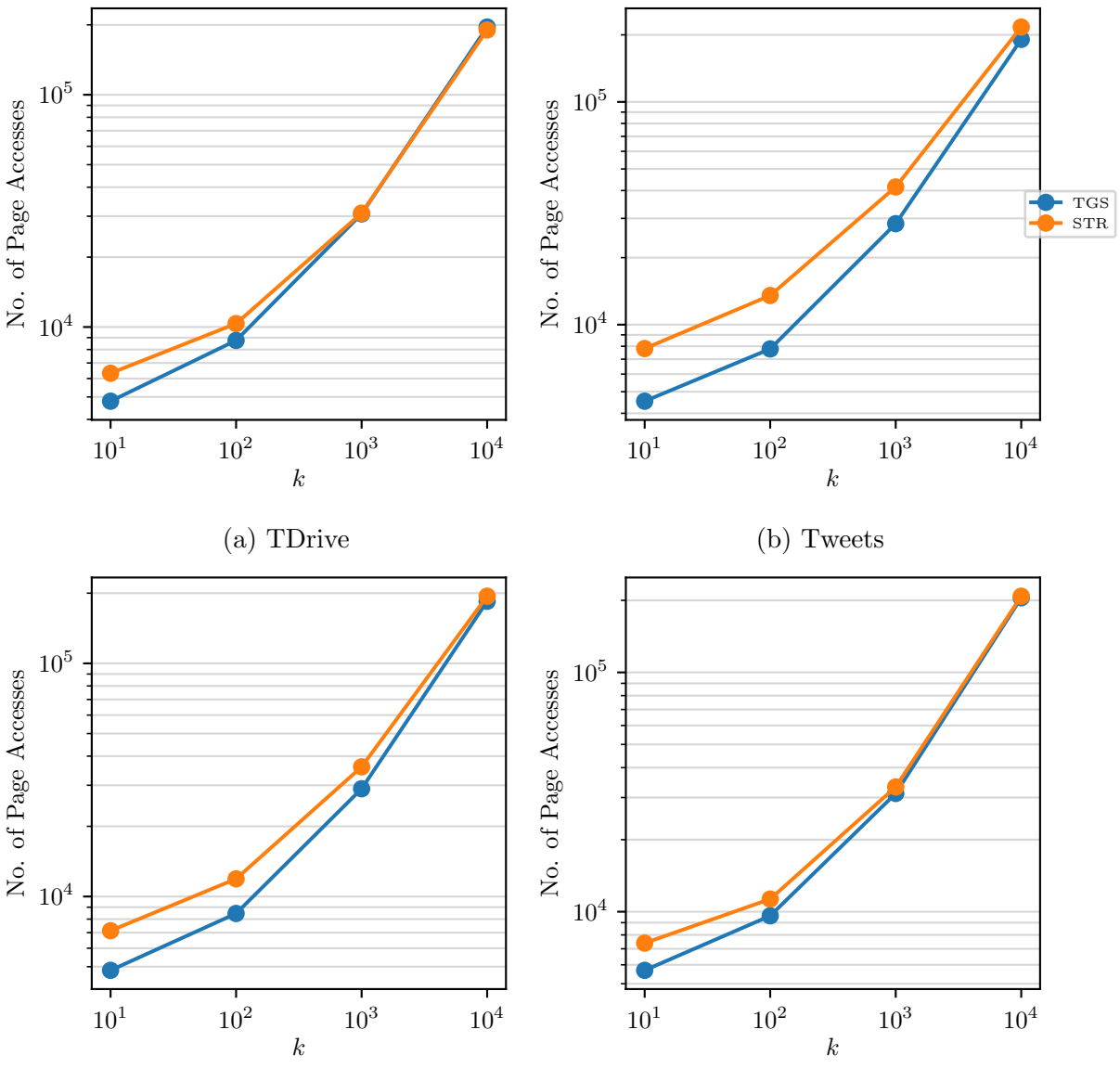
TGS outperforms STR the most on the sparse Tweets dataset, using 40% less I/O to perform point queries. On the other hand, its advantage is the least on the TDrive and NYCTaxi dataset, where it uses 25% less I/O than STR. Recall from Chapter 5 that the top-down approach of TGS works well to eliminate overlap when the points are not aligned on the same axis. Sparse datasets like Tweets do not have as many points that share an axis, and hence TGS produces MBRs with little to no overlap. On the other hand, it is more likely for points to share the same axis on dense datasets like TDrive and NYCTaxi, forcing TGS to generate MBRs that overlap each other. Thus, queries on dense datasets induce more spurious I/O compared to sparser datasets on TGS due to MBR overlap.

Range Searches

Figure 6.7 shows the I/O cost for 1000 range searches for each selectivity value k. TGS exhibits better range search performance compared to STR across all datasets when the selectivity is high, which increases the chance that the query lands on an overlapping region. In the case of STR, which does not eliminate overlap as effectively as TGS, this results in queries accessing more pages than necessary. Similar to the point query results, observe that TGS has the largest advantage on the Tweets dataset, using 40% to 10% less I/O than STR as k increases. The performance gap between TGS and STR is the smallest on the NYCTaxi and TDrive dataset, where TGS uses 23% to $< 1\%$ less I/O compared to STR with increasing k. STR even outperforms TGS on the TDrive dataset for $k = 10^4$ by a margin of 3%. As the selectivity decreases, the number of pages accessed by STR and TGS converge. This is because range queries with low selectivity are more affected by the area and margin of MBRs than the degree of overlap. Moreover, beyond a certain selectivity, the differences between the bulk-loading algorithms become negligible, since it is likely that most pages in the tree will be accessed anyway.

6.3.2 Memory Usage

Previously, Table 6.3 showed index sizes after inserting each data point individually into the various R-Tree variants. Due to the differences in how they choose to insert each individual data point into a leaf node, each R-Tree variant produces indexes of different



(a) TDrive

(b) Tweets

(c) GeoLife

(d) NYCTaxi

Figure 6.7: Range query results (log-log scale)

sizes across all four datasets. However, bulk-loading algorithms pack their nodes with as many entries as possible, and so bulk-loaded indexes tend to have the same size irrespective of the algorithm used. Table 6.5 displays the size of the indexes produced by TGS and

Dataset	Number of Points	TGS/STR Size
TDrive	11,317,142	182
Tweets	15,598,403	251
Geolife	22,033,507	354
NYCTaxi	81,616,580	1314

Table 6.5: Dataset size and bulk-loaded index sizes for TGS and STR (MB)

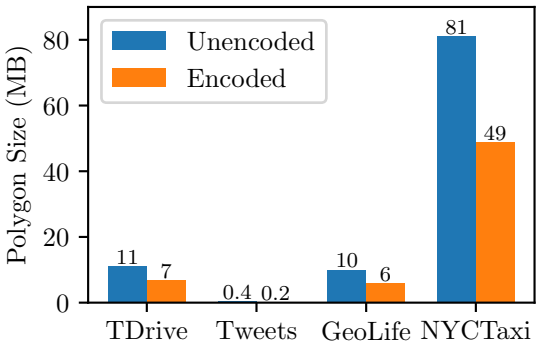


Figure 6.8: Memory usage of unencoded vs encoded polygons for TGS (MB)

STR. As expected, index sizes increase in a perfectly linear trend with dataset sizes.

To confirm the viability of using TGS as a bulk-loading algorithm for the NIR+-Tree, the `fragment` method is applied on overlapping MBRs to convert them into polygons in the indexes produced by TGS. Figure 6.8 shows the memory overhead of the polygon overlay across all four datasets indexed by TGS. The size of the polygon overlay can be used as a proxy for the degree of overlap present in the constructed index. On a sparse dataset such as Tweets, which has little overlap to begin with, TGS produces a minimal encoded polygon overlay that is $< 0.1\%$ of the tree. Conversely, TGS fragments a significant amount of MBRs into polygons on NYCTaxi, producing a polygon overlay comprising 3.7% of the tree. Surprisingly, using TGS results in a bigger polygon overlay compared to using the NIR+-Tree to index NYCTaxi — 49 MB vs 28 MB respectively. This is explained by the fact that the nodes in TGS are more packed than the nodes in the NIR+-Tree. This packing results in more MBRs in the nodes created by TGS compared to the NIR+-Tree, increasing the likelihood of overlap. We note the same for the TDrive dataset whose polygon overlay size on TGS is approximately the same as the NIR+-Tree. The opposite trend holds for Tweets and GeoLife, where TGS has smaller overlays than the NIR+-Tree.

6.4 Summary

By employing non-overlapping polygons to refine searches, the NIR+-Tree is quite effective at reducing the number of I/Os for point searches and selective range searches. As the selectivity of the range query decreases, minimizing overlap has little impact. The NIR+-Tree's closest competitor is the R+-Tree, which the NIR+-Tree outperforms by using 2× to 5× less I/O on range queries. The NIR+-Tree also outperforms the popular R*-Tree by as much as 15×. Furthermore, the NIR+-Tree's polygon overlay has a tenable memory footprint as it only uses a small percentage of the total tree size on disk. The main weakness of the NIR+-Tree is its computationally heavy polygon construction algorithms. However, this is preferred to the R*-Tree's re-insertion technique, which expends a significant amount of I/O in an attempt to find a better layout for its MBRs. This makes the NIR+-Tree a viable alternative to the R*-Tree for applications that execute point and selective range queries.

When the dataset is known in advance, bulk-loading algorithms can be used to construct indexes with optimal node utilization and structure. Experimentally, we show that the top-down TGS algorithm outperforms STR on point and selective range queries due to its ability to generate MBRs with less overlap. Consequently, the small number of overlapping MBRs in TGS results in small polygon overlays, making TGS an ideal candidate to use as a starting point for the NIR+-Tree.

Chapter 7

Related Work

This chapter presents a survey of related work in the area of spatial indexes. Spatial indexes are typically classified into two categories: space-partitioning and data-partitioning indexes, with R-Trees belonging to the latter. This chapter describes both main-memory and disk-resident indexes for each category. The various enhancements to MBRs that have been proposed in the literature are then presented. Finally, the chapter concludes with a discussion of bulk-loading algorithms for R-Tree variants.

7.1 Space-partitioning Indexes

Space-partitioning indexes recursively partition multi-dimensional space into multiple regions, where each region hosts a collection of data objects. Note that the regions in space-partitioning indexes are disjoint and do not overlap. Consequently, such indexes are typically designed only for point data, since volumetric data (such as rectangles) across multiple regions will need to be replicated in each region. Additionally, the nodes of space-partitioning indexes can cover regions that do not hold any data and thus can contain significant dead space.

The Quad-Tree [11], Oct-Tree [28] and KD-Tree [6] are hierarchical space-partitioning indexes designed for main-memory use. Quad-Trees and Oct-Trees partition 2D and 3D space into quadrants and octants respectively. Each partition is then further recursively subdivided into quadrants or octants until the partition holds a single data object. Quad-Trees and Oct-Trees do not take into account the distribution of data when computing partitions and hence can have nodes that contain no data. Quad-Trees and Oct-Trees

are not balanced trees, since objects that are densely concentrated in a particular area can cause the tree to construct more nodes in that region. KD-Trees [6] take a different approach by partitioning space into two along the median of the data objects, alternating the partitioning dimension at each level of the tree. Consequently, each partition maintains an equal number of data objects and thus unlike Quad-Trees and Oct-Trees, KD-Trees are balanced search trees.

The KDB-Tree [34] is the disk-based equivalent of the in-memory KD-Tree. Unlike a KD-Tree, where each partitioned region is its own node, each KDB-Tree node contains multiple partitioned regions stored on a single disk page. The subsequent split of a node that happens after inserting a point into a KDB-Tree can cause downward-propagating splits, making insertion expensive. To mitigate this issue, hB-Trees [27] were proposed, which represents the partitioned regions in a node using a KD-Tree. This allows it to perform splits in more than one dimension, restricting downward splits to a single root-to-leaf path. This minimizes the number of downward-propagating splits, making insertions more efficient. BKD-Trees [32] further improve the performance of inserts by maintaining a structure of multiple KDB-Trees and only applying updates to a subset of them.

The UB-Tree [33] is a multi-dimensional, disk-based index built on top of the B-Tree [2]. The UB-Tree maps each multi-dimensional data object to a single-dimension using a space-filling curve such as the Morton order. The key selling point of the UB-Tree is that it can be easily integrated into existing database systems that already use B-Trees.

The Grid File [30] is a disk-backed, non-hierarchical space-partitioning index. Each cell in a Grid File contains a handle to a disk page that holds data objects. Selecting the appropriate granularity of the cell size is critical to the performance of a Grid File. The granularity at which each dimension is divided into cells is maintained using a linear scale. Various extensions to the Grid File have been proposed, such as the Twin-Grid File [18] which attempts to improve the space utilization of the cells in a grid.

Non-hierarchical space-partitioning indexes for main-memory have also been proposed, such as BLOCK [31], which aims to reduce the CPU cost associated with searching in a spatial index. BLOCK uses a grid-based approach to partition space into a grid of cells, where each grid contains a pointer to data objects. Similar to how the NIR-Tree and NIR+-Tree enhances searches using polygons, BLOCK refines a search using multiple grids each with different cell sizes.

7.2 Data-partitioning Indexes

Data-partitioning indexes group together data objects that are spatially close into a tight enclosing structure. Since the geometry of the enclosure is derived from the data objects, data-partitioning indexes are suitable for storing volumetric data without replicating them. Since these indexes tightly enclose data objects, they exhibit minimal dead space in their nodes. However, the main drawback of data-partitioning indexes is that their enclosing structures can overlap, forcing searches to visit multiple nodes.

R-Trees [17] are the most widely used data-partitioning indexes, designed for disk-based indexing. The R-Tree's enclosing structure is the Minimum Bounding Rectangle (MBR), which is the smallest rectangle that can contain a collection of point or volumetric objects. Improvements to R-Trees involve techniques to improve the spatial quality of its MBRs. The R*-Tree [3] performs re-insertions of the entries in an overfull node to find better placements for them. The Revised R*-Tree [5] uses modified heuristics when inserting data and splitting nodes instead of re-insertions to achieve the same or better performance as the R*-Tree [3]. To prevent overlap between MBRs when handling volumetric data, the R+-Tree [37] replicates data objects when they lie across multiple nodes. Similar to the space-partitioning KDB-Tree [34], the R+-Tree also uses a downward-propagating split strategy when splitting overfull nodes. Unlike the KDB-Tree, the R+-Tree bounds its data objects using MBRs, eliminating a significant amount of dead space. The NIR-Tree and NIR+-Tree extend the R+-Tree by eliminating the overlap that can happen between its MBRs during the `chooseLeaf` operation.

CR-Trees [20] are R-Tree variants optimized for main-memory indexing by utilizing compression for the MBRs in a node. Each MBR is compressed using a relative representation that stores the difference between a child MBR and a parent MBR. By compressing MBRs, the CR-Tree packs more MBRs into a single node, increasing the fanout and cache utilization of the tree. Similarly, the NIR-Tree [22] reduces cache misses by preventing unnecessary pointer chasing incurred during searches due to overlapping MBRs. The NIR-Tree achieves this by fragmenting overlapping MBRs into polygons comprised of smaller rectangles that precisely cover the area owned by the original MBR. This enables the NIR-Tree to fetch only those nodes that are relevant to the query. The NIR+-Tree adapts this technique for disk-based indexing by using non-overlapping polygons to prevent spurious disk accesses during queries.

While R-Trees are suitable for indexing data in low-dimensional spaces such as 2D and 3D, they do not scale well to higher dimensions. The TV-Tree [25] and X-Tree [7] are disk-based hierarchical indexes designed for efficiently indexing high-dimensional data. The

TV-Tree stores data using only the dimensions necessary to distinguish between objects at each level of the tree. The X-Tree reduces the significant overlap that can happen between MBRs in high-dimensional spaces by using a split algorithm that minimizes overlap. In case the X-Tree cannot find a suitable split, it merges together the entries in a node into supernodes spanning multiple pages.

Waffle [29] is a disk-resident spatial index that combines ideas from both space and data-partitioning indexes to produce MBRs that are disjoint. Waffle partitions a collection of points into MBRs using axis-aligned splits and stores splits in the order of their creation in its branch nodes. This ordering allows Waffle to insert new points into the index without having to expand MBRs. However, Waffle can only handle point data and is not suitable for volumetric data since splits can replicate data objects across nodes.

7.3 MBR Augmentations

Minimum Bound Rectangles (MBRs) are the basic building blocks of data-partitioning indexes. Therefore, techniques that improve the spatial quality of MBRs will also improve the performance of any index that uses them.

Clipped Bounding Boxes (CBBs) [38] refine MBRs by clipping away dead space from their corners. CBBs re-use the underlying geometry of the data objects an MBR encloses to produce clip points that prune away dead space. Clip points are stored in main-memory and are used to minimize page accesses during searches. Each clip point is associated with a point inside the MBR and a particular corner of the MBR. Instead of storing the full coordinates of the corners, CBBs represent the four corners of an MBR using a 2-bit flag. Unlike CBBs, the NIR+-Tree only uses the lower left and upper right corners of the MBR for its encoding. Moreover, the NIR+-Tree encodes at the granularity of coordinates instead of corners, allowing it to substitute the coordinates of a polygon using matching coordinates of its reference rectangle. Finally, the NIR+-Tree refines searches similar to CBBs by using polygons, but unlike CBBs, polygons target overlap between MBRs instead of dead space.

MaMBo [9] aims to eliminate dead space from MBRs by tessellating an MBR into a uniform grid of cells. Grid cells of an MBR in MaMBo are represented using a bitmap, with a bit set to 1 if the cell is occupied and 0 if it contains dead space. MaMBo uses these bitmaps to decide whether a query intersects dead space in an MBR or not, thereby avoiding disk accesses to MBRs that do not intersect the query.

Raster Intervals [14] augment MBRs using grids in order to accelerate spatial joins.

The goal in a spatial join is to find all pairs of intersecting spatial objects from two sets. MBRs are used as an initial filter to prune away pairs of objects that do not intersect. In the subsequent refinement step, the remaining pairs are tested for intersection using the actual geometry of the objects contained in the MBRs. Raster Intervals apply a grid on each pair of MBRs to quickly determine intersection and avoid a computationally expensive refinement phase.

Like the techniques presented above, polygons used by the NIR-Tree and NIR+-Tree can be considered as MBR enhancements since they disambiguate overlapping sections of MBRs, minimizing spurious searches.

7.4 Bulk-Loading Algorithms

This section discusses bulk-loading algorithms for R-Tree variants. Bulk-loading algorithms can be used to construct an R-Tree with optimal node utilization from a pre-defined set of input data. As presented in Chapter 5, bulk-loading algorithms can be classified into two categories: bottom-up and top-down.

Bottom-up algorithms construct leaf nodes first and then recursively build the branch nodes of the tree. Nearest-X [35] sorts objects on the x-coordinate and then groups objects in this order into nodes. Hilbert-Sort [19] uses the space-filling Hilbert curve to sort objects for grouping. Sort-Tile-Recursive [24] sorts objects on the x-coordinate, groups them into vertical tiles, sorts objects in each tile on the y-coordinate and then groups them into nodes. The PR-Tree [1] is a worst-case optimal bulk-loaded R-Tree that provides a theoretical bound on the number of I/Os incurred during a range query. FLAT [39] is an indexing scheme that uses STR to generate leaf node pages and then stores pointers to neighboring leaf node pages. This allows FLAT to perform range queries without having to traverse a hierarchy of branch nodes.

Top-down algorithms start by partitioning the input data into the highest-level branch nodes and then recursively build lower-level branch nodes, constructing the leaf nodes last. Top-Down Greedy Split (TGS) [13] evaluates all binary groupings of objects when constructing nodes at each level of the tree and selects the best grouping using a cost metric. Waffle [29] also introduces a top-down bulk-loading algorithm for point data, where it computes and stores axis-aligned splits to generate non-overlapping partitions. As demonstrated in Chapter 5, top-down algorithms such as TGS generate MBRs with little to no overlap, making them suitable for producing bulk-loaded NIR+-Trees.

Chapter 8

Conclusion

This thesis presented techniques to convert the main-memory NIR-Tree for disk-based indexing. By moving complex polygons out of branch and leaf nodes and into a memory-resident hash table, the fanout of the NIR-Tree is kept steady. Doing so enables efficient disk-based indexing while retaining the non-intersecting property of the original NIR-Tree. Furthermore, it is possible to represent polygon coordinates in memory by their relative position to the MBR on disk. Using this representation, polygons are encoded such that they use less than 5% of the total index size for storage. The resulting index, the NIR+-Tree, uses up to 5× fewer disk accesses to execute searches compared to state-of-the-art R-Tree variants on real-world point datasets.

Additionally, this thesis examines algorithms that can be used to create bulk-loaded NIR+-Trees. Chapter 5 shows that bottom-up bulk-loading algorithms create MBRs that overlap significantly, making them a poor choice for bulk-loading the NIR+-Tree. Instead, top-down bulk-loading algorithms create minimally overlapping MBRs on point datasets by grouping points into MBRs at each level of the tree. The resulting bulk-loaded NIR+-Tree needs very few polygons to disambiguate overlapping MBRs, making it a good starting index for subsequent workloads that insert more data.

The NIR+-Tree can be improved and extended in several ways:

- The techniques and experiments presented so far only allow the NIR+-Tree to index point data. More work is required to support indexing volumetric data, such as rectangles. Specifically, the `fragment` algorithm used to convert overlapping MBRs into polygons must be extended to work for volumetric data.

- The number of rectangles constituting a polygon in the NIR+-Tree are unbounded and can grow arbitrarily large. This can lead to poor performance when the NIR+-Tree is entirely in memory, due to an unbounded number of intersection tests. One approach to taming polygon complexity is to simply discard overly large polygons from the overlay, thereby reducing the number of intersection tests required as well as the size of the overlay.

- The concept of converting overlapping rectangles into non-overlapping polygons need not be limited to the NIR+-Tree. An interesting avenue of future work is to apply the polygon fragmentation to other R-Tree variants to see how much performance can be gained.

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 Trans. Algorithms*, 4(1), mar 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, pages 107–141, 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, pages 799–812, New York, NY, USA, 2009. Association for Computing Machinery.

[6] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, sep 1975.

[7] Stefan Berchtold, Daniel Keim, and Peer Kröger. The x-tree: An index structure for high-dimensional data. 02 1970.

[8] H. Blanken, A. Ijbema, P. Meek, and B. van den Akker. The generalized grid file: description and performance aspects. pages 380–388, 1990.

[9] Giannis Evagorou and Thomas Heinis. Mambo - indexing dead space to accelerate spatial queries. In *Proceedings of the 33rd International Conference on Scientific and Statistical Database Management*, SSDBM '21, pages 73–84, New York, NY, USA, 2021. Association for Computing Machinery.

[10] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '89, pages 247–252, New York, NY, USA, 1989. Association for Computing Machinery.

[11] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Inf.*, 4(1):1–9, mar 1974.

[12] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems: The complete book*, page 568. Pearson, 2014.

[13] Yván J. García R, Mario A. López, and Scott T. Leutenegger. A greedy algorithm for bulk loading r-trees. In *Proceedings of the 6th ACM International Symposium on Advances in Geographic Information Systems*, GIS '98, pages 163–164, New York, NY, USA, 1998. Association for Computing Machinery.

[14] Thanasis Georgiadis and Nikos Mamoulis. Raster intervals: An approximation technique for polygon intersection joins. *Proc. ACM Manag. Data*, 1(1), may 2023.

[15] Saheli Ghosh, Tin Vu, Mehrad Amin Eskandari, and Ahmed Eldawy. UCR-STAR: The UCR Spatio-Temporal Active Repository. *SIGSPATIAL Special*, 11(2):34–40, December 2019.

[16] D. Greene. An implementation and performance analysis of spatial data access methods. pages 606–615, 1989.

[17] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2):47–57, jun 1984.

[18] Andreas Hutflesz, Hans-Werner Six, and Peter Widmayer. Twin grid files: Space optimizing access schemes. *SIGMOD Rec.*, 17(3):183–190, jun 1988.

[19] Ibrahim Kamel and Christos Faloutsos. On packing r-trees. In *Proceedings of the Second International Conference on Information and Knowledge Management*, CIKM '93, pages 490–499, New York, NY, USA, 1993. Association for Computing Machinery.

[20] Kihong Kim, Sang K. Cha, and Keunjoo Kwon. Optimizing multidimensional index trees for main memory access. *SIGMOD Rec.*, 30(2):139–150, may 2001.

[21] Kihong Kim, Sang K. Cha, and Keunjoo Kwon. Optimizing multidimensional index trees for main memory access. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 139–150, New York, NY, USA, 2001. Association for Computing Machinery.

[22] Kyle Langendoen, Brad Glasbergen, and Khuzaima Daudjee. Nir-tree: A non-intersecting r-tree. In *Proceedings of the 33rd International Conference on Scientific and Statistical Database Management*, SSDBM '21, pages 157–168, New York, NY, USA, 2021. Association for Computing Machinery.

[23] Langendoen, Kyle Jacob. A non-intersecting r-tree. Master's thesis, 2021.

[24] S.T. Leutenegger, M.A. Lopez, and J. Edgington. Str: a simple and efficient algorithm for r-tree packing, 1997.

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

[26] David Lomet. Cost/performance in modern data stores: How data caching systems succeed. In *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*, pages 140–140, 2019.

[27] David B. Lomet and Betty Salzberg. The hb-tree: A multiattribute indexing method with good guaranteed performance. *ACM Trans. Database Syst.*, 15(4):625–658, dec 1990.

[28] Donald Meagher. Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-d objects by computer. 10 1980.

[29] Moin Hussain Moti, Panagiotis Simatis, and Dimitris Papadias. Waffle: A workload-aware and query-sensitive framework for disk-based spatial indexing. *Proc. VLDB Endow.*, 16(4):670–683, dec 2022.

[30] J. Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst.*, 9(1):38–71, mar 1984.

[31] 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.

[32] Octavian Procopiuc, Pankaj K. Agarwal, Lars Arge, and Jeffrey Scott Vitter. Bkd-tree: A dynamic scalable kd-tree. pages 46–65, 2003.

[33] Frank Ramsak, Volker Markl, Robert Fenk, Martin Zirkel, Klaus Elhardt, and Rudolf Bayer. Integrating the ub-tree into a database system kernel. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 263–272, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[34] John T. Robinson. The k-d-b-tree: A search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, SIGMOD '81, pages 10–18, New York, NY, USA, 1981. Association for Computing Machinery.

[35] Nick Roussopoulos and Daniel Leifker. Direct spatial search on pictorial databases using packed r-trees. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, SIGMOD '85, pages 17–31, New York, NY, USA, 1985. Association for Computing Machinery.

[36] Nick Roussopoulos and Daniel Leifker. Direct spatial search on pictorial databases using packed r-trees. *SIGMOD Rec.*, 14(4):17–31, may 1985.

[37] 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, pages 507–518, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc.

[38] 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.

[39] Farhan Tauheed, Laurynas Biveinis, Thomas Heinis, Felix Schurmann, Henry Markram, and Anastasia Ailamaki. Accelerating range queries for brain simulations. pages 941–952, 2012.