Enhancing Spatial Query Efficiency Through Dead Space Indexing in Minimum Bounding Boxes

by

Ying Chen

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

© Ying Chen 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 introduces the Grid Minimum Bounding Box (GMB) as an enhancement to traditional Minimum Bounding Box (MBB), designed to mitigate the negative impact of dead space and improve query efficiency. The GMB achieves this by utilizing a low-cost grid bitmap to index dead space within the MBB, enabling the filtering of queries that intersect only with dead space. This filtering reduces false positives in intersection tests and minimizes unnecessary disk I/O to leaf nodes. A key advantage of GMB is that it is developed as an augmentation technique, enabling seamless integration with any R-tree variant without altering the core indexing architecture. The effectiveness of this technique is validated through a comprehensive set of experiments on both real-world and synthetic datasets, demonstrating significant improvements in query performance across various datasets and query types.

The key contributions of this thesis include the development of a data-driven algorithm for constructing GMBs, the design of a grid bitmap compression technique, the implementation of an efficient maintenance system for dynamic GMBs, and the enhancement of search operations through GMB-based intersection tests. These contributions collectively establish GMB as a robust solution to the challenges presented by dead space within MBBs across different R-tree variants.

Acknowledgements

I would like to thank my supervisor, Prof. Khuzaima Daudjee, for his continuous support, insightful guidance, and constant encouragement during my master's studies. His knowledge and dedication have greatly influenced this thesis and my academic growth. Without his mentorship, this work would not have been possible.

I am also very thankful to Prof. Tamer Özsu and Prof. Grant Weddell for being on my committee. Their valuable feedback, thoughtful comments, and encouragement have greatly improved this research. Their dedication to excellence has inspired me to aim for the same in my own work.

I would like to express my gratitude to Fadhil Abubaker for showing me how to be a good collaborator. His enthusiasm and dedication brought me the joy of working together on a project that laid the foundational groundwork for this thesis.

I am also grateful to my friends in the Data Systems Group (DSG) lab and at the University of Waterloo. The interesting discussions and friendships have made this journey enriching and unforgettable. Thank you for the great memories and all the support you've given me, both in my studies and personally. In particular, I would like to thank Ali Ghadirli, Gaurav Sehgal, and Anurag Chakraborty.

Looking back, the two memorable years I spent at SAP were a key time when I first got into databases. I am thankful to the mentors I met there, who kindly shared their knowledge and sparked my interest in this field. Their guidance laid the foundation for my later studies and research, and I value the professional relationships and friendships formed during that time.

I also want to sincerely thank Gursharan Kahlon, whose constant support during the hardest times in my life has been a source of hope. Her kindness and guidance have helped me find inner peace, and I am deeply grateful to have her in my life. Her encouragement has helped me through personal challenges and has also positively influenced my studies.

Finally, I deeply thank my family for their unconditional love and support throughout this journey. A special thank you goes to my dog, Bai, who has been the true savior of my life. Bai's companionship and constant loyalty have given me comfort and motivation during the hardest times. Their belief in me has been the driving force behind my accomplishments.

Dedication

This work is dedicated to my grandpa LiangCai Chen, who is no longer with us. Rest in peace.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements iv

Dedication v

List of Figures viii

List of Tables x

1 Introduction 1

2 Background 6

2.1 R-tree 6

2.2 Minimum Bounding Box 7

2.3 Formalization and Notations 11

3 Grid Minimum Bounding Box Design 13

3.1 Motivation of Grid Minimum Bounding Box 13

3.2 Logical Layout 16

3.3 Physical Layout 17

4 Grid Minimum Bounding Box Algorithms **21**
4.1 Construction . . . 21
4.2 Query . . . 22
4.3 Update . . . 26
4.3.1 Insert . . . 28
4.3.2 Delete . . . 28

5 Experiment Evaluation **29**
5.1 Setups . . . 29
5.1.1 Datasets . . . 30
5.1.2 Queries . . . 31
5.2 Dead Space Elimination . . . 32
5.3 Query Performance . . . 36
5.3.1 Query Performance for 2 Dimensional Point Data . . . 37
5.3.2 Query Performance for 2 Dimensional Rectangle Data . . . 37
5.3.3 Query Performance for 3 Dimensional Point Data . . . 43
5.3.4 Query Performance for 3 Dimensional Rectangle Data . . . 43
5.4 Overheads . . . 45
5.4.1 Storage Overheads . . . 48
5.4.2 Construction Overheads . . . 49
5.5 Summary . . . 51

6 Related Work **55**
6.1 Space-partitioned Indexes . . . 55
6.2 Data-partitioned Indexes . . . 57
6.3 Bounding Shapes . . . 59
6.4 Augmentation Techniques . . . 60

7 Conclusion **62**

References **63**

List of Figures

1.1 Existing Room for Performance Improvement 3

2.1 Unique Identification for Axis-aligned Hyper-rectangle 7
2.2 Use Cases for Minimum Bounding Box 8
2.3 Three Different Cases for Intersection Test of MBB 9
2.4 Visualization of Characteristics of MBBs 9
2.5 Comparison between Dead Space and Overlap 10

3.1 An Example of two-level R-tree with 8 two-dimensional objects 14
3.2 An Example of Range Query on R-tree based on MBB 14
3.3 An Example of Grid Bitmap on R-tree 15
3.4 An Example of Range Query on R-tree based on GMB 15
3.5 Logical Layout of R-tree with GMB in 3.1 17
3.6 Physical Layout of R-tree with GMB in bitset 18
3.7 Physical Layout of R-tree with GMB in list of grid location 20

5.1 Elimination of Dead Space on R-tree variants for 2 dimensional point data 34
5.2 Elimination of Dead Space on R-tree variants for 2 dimensional rectangle data 34
5.3 Elimination of Dead Space on R-tree variants for 3 dimensional point data 35
5.4 Elimination of Dead Space on R-tree variants for 3 dimensional rectangle data 35
5.5 Leaf Node Accesses Distribution for 2-dim Point Data on Query 0 38
5.6 Leaf Node Accesses Distribution for 2-dim Point Data on Query 2 38

5.7 Leaf Node Accesses Distribution for 2-dim Point Data on Query 3 39
5.8 Leaf Node Accesses Distribution for 2-dim Rectangle Data on Query 0 . . 40
5.9 Leaf Node Wasted Accesses Reduction for 2-dim Rectangle Data on Query 0 40
5.10 Leaf Node Accesses Distribution for 2-dim Rectangle Data on Query 2 . . 41
5.11 Leaf Node Wasted Accesses Reduction for 2-dim Rectangle Data on Query 2 41
5.12 Leaf Node Accesses Distribution for 2-dim Rectangle Data on Query 3 . . 42
5.13 Leaf Node Wasted Accesses Reduction for 2-dim Rectangle Data on Query 3 42
5.14 Leaf Node Accesses Distribution for 3-dim Point Data on Query 0 43
5.15 Leaf Node Accesses Distribution for 3-dim Point Data on Query 2 44
5.16 Leaf Node Accesses Distribution for 3-dim Point Data on Query 3 44
5.17 Leaf Node Accesses Distribution for 3-dim Rectangle Data on Query 0 . . 45
5.18 Leaf Node Wasted Accesses Reduction for 3-dim Rectangle Data on Query 0 46
5.19 Leaf Node Accesses Distribution for 3-dim Rectangle Data on Query 2 . . 46
5.20 Leaf Node Wasted Accesses Reduction for 3-dim Rectangle Data on Query 2 47
5.21 Leaf Node Accesses Distribution for 3-dim Rectangle Data on Query 3 . . 47
5.22 Leaf Node Wasted Accesses Reduction for 3-dim Rectangle Data on Query 3 48
5.23 Storage Distribution for R-tree variants on 2-dim Point Data 49
5.24 Storage Distribution for R-tree variants on 2-dim Rectangle Data 50
5.25 Storage Distribution for R-tree variants on 3-dim Point Data 50
5.26 Storage Distribution for R-tree variants on 3-dim Rectangle Data 51
5.27 CPU time distribution for R-tree variants on 2-dim Point Data 52
5.28 CPU time distribution for R-tree variants on 2-dim Rectangle Data 52
5.29 CPU time distribution for R-tree variants on 3-dim Point Data 53
5.30 CPU time distribution for R-tree variants on 3-dim Rectangle Data 53

List of Tables

3.1 Bitmap Size Example 19

5.1 R-tree Variants Fan-out 30
5.2 Dataset Characteristics 31
5.3 Query Setups 32
5.4 Average Performance Impact Summary on R-tree 54

Chapter 1

Introduction

Over the last three decades, there has been significant growth in the generation and utilization of spatial data across various applications. This surge has necessitated the development of spatial databases, a distinct subset of data management systems tailored to the unique characteristics of spatial data. Systems such as PostGIS [3], PyGIS [4], ArcGIS [2], QGIS [5], and Apache Sedona [1] underscore the importance and complexity of managing spatial datasets. Given the critical role of spatial data, numerous academic and industrial research efforts have focused on spatial data indexing, with variants of the R-tree [23, 9, 11, 44, 7, 29] family receiving substantial attention.

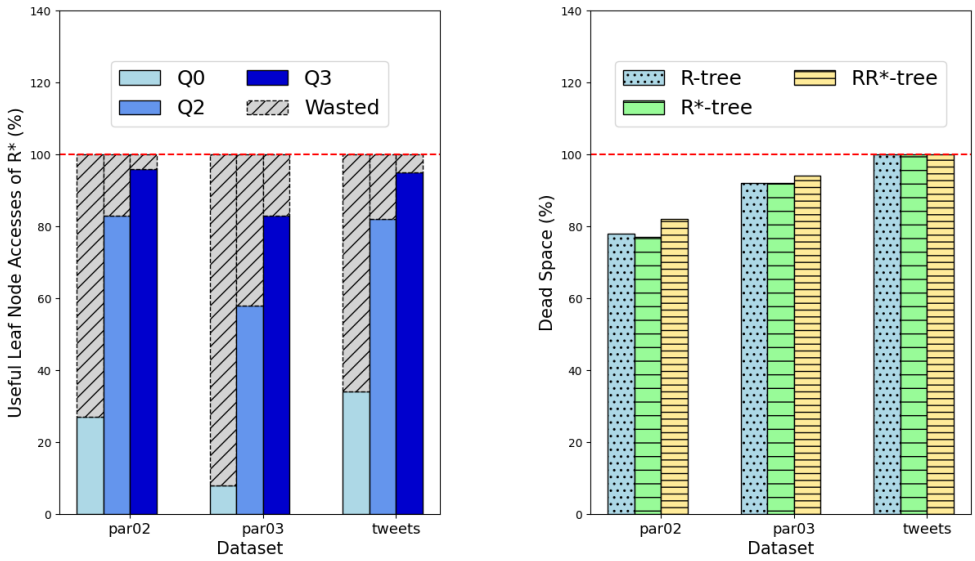
The R-tree [23], inspired by the B-tree [8], organizes data by grouping closely related objects into subgroups. However, spatial object attributes are independent and frequently demonstrate high co-occurrence in queries, exemplified by the latitude and longitude in geospatial data from GPS systems, or the coordinate axes in mechanical Computer-Aided Design (CAD) systems. The co-occurence of multiple attributes in queries makes it less obvious on the optimal clustering for data retrieval. Instead of utilizing one-dimensional bucket as the B-tree, the core component of the R-tree and many other spatial indexing techniques is based on Minimum Bounding Box (MBB). A tree structure based index is composed of nodes, where R-tree is essentially a hierarchy of nodes represented by MBBs. Each leaf node of the tree represents a subgroup of closely related data objects, and each branch node of the tree represents a subgroup of closely related sub-level nodes where nodes can be either branch nodes or leaf nodes. Each node has a corresponding MBB where the MBB is the smallest axis-aligned hyper-rectangle that contains all the objects of interest. MBBs can be considered as an efficient high-level approximation of all obejcts within the subgroups or nodes. Another characteristic of MBB is that it requires only two multi-dimensional points for representation, such as $< LowerLeftPoint, UpperRightPoint >$.

This simplicity of representation ensures storage efficiency, while intersection tests between MBBs and queries can be conducted in linear time relative to the number of dimensions.

The primary advantage of utilizing Minimum Bounding Box (MBB) as building block for R-tree is its ability to serve as an effective pre-filter for spatial queries, ensuring only true negative outcomes. The true negative property here means that all negative results of pre-filter by MBB are guaranteed to be correct. If an MBB of a node does not intersect with a query, it guarantees that none of the objects within the node intersect with the query. It allows the corresponding node to be skipped without checking any individual objects with the search query. However, if an MBB of a node does intersect with the query, each object within the node must be individually examined to determine which objects are desired by the query. Despite its usefulness, the approximate nature of MBB can lead to false positives, meaning that positive outcomes of the MBB's intersection test can be false. When an MBB intersects with a search query, it does not guarantee that some of the enclosed objects will also intersect. It is possible that none of the objects on the node are desired by query. This can only be determined by checking all the objects within the node, resulting in wasted node accesses.

As mentioned in the previous paragraph, the false positive outcomes of intersection tests directly lead to wasted node accesses, which significantly impact query performance. To quantify the impact of MBB's false positives on intersection tests, we introduce two concepts: useful node accesses and wasted node accesses. If a node contains no objects that answer the query, it is categorized as a wasted node access; otherwise, it is a useful node access. Figure 1.1(a) shows the distribution of useful and wasted leaf node accesses for various datasets across three common query types. Q0 is point query type, and Q2 and Q3 are range queries. Q2 is range query where 100 objects are expected in the output and Q3 has 1000 objects as expected in the output. Surprisingly, for the most selective query type Q0, 70% to 90% of leaf node accesses are actually wasted, indicating that while the MBB intersects with the query, the node contains no desired objects. This test is conducted on the state-of-the-art R*-tree which suggests that the performance of other R-tree variants may be even less optimal.

When an MBB gives a false positive output for a query, it means the MBB intersects with the query, but none of the enclosed objects answer the query. In other words, the query only intersects with the dead space of the MBB. Dead space is the portion of the MBB that does not actually contain any data objects; it is the empty or unused space within the boundary of the MBB. Dead space is one of the key characteristics of MBBs, as it can lead to inefficiencies in query processing by increasing the likelihood of false positives in intersection tests. In Figure 1.1(b), we quantify the dead space within MBBs at the leaf node level for different R-tree variants. For datasets with rectangle data types, par02



(a) Useful and Wasted Leaf Node Accesses on R*-tree

(b) Dead Space at Leaf Node Level

Figure 1.1: Existing Room for Performance Improvement

and par03, we observe around 80% dead space in two dimensions and even higher, 90%, in three dimensions. For dataset with point entry types Tweets, since there is no volume to the data, the entire enclosed space can be considered as dead space.

The wasted leaf node access test in Figure 1.1(a) indicates that there is significant room for performance improvement even in the state-of-the-art R*-tree. Our dead space quantification test in Figure 1.1(b) further demonstrates that dead space can be a major issue contributing to false positive query outputs, which in turn lead to wasted leaf node accesses. This dead space arises from the approximative nature of MBBs, where their simplicity provides efficiency in both storage and intersection test, but at the cost of precision.

Given these observations, an important insight is to rebalance the trade-off between simplicity and precision of MBBs. One approach to achieve this is by enhancing MBBs with additional information about dead space. Therefore, we propose the Grid Minimum Bounding Box (GMB) as an enhanced MBB that includes an auxiliary data structure to provide detailed information on the location of dead space at a low storage cost. At a high level, the GMB is constructed by analyzing the characteristics of MBBs and using a low-cost bitmap to index dead space within the MBB. By indexing dead space, queries that intersect only with dead space can be filtered out using the dead space bitmap, thereby reducing the false positive rate of MBB intersection tests and improving query performance as a desired result. The advantage of having GMB as an augmentation technique is that GMB does not impose any assumptions on how the indexing or tree structure is built, making it a versatile technique that can be applied to any indexing structure relying on MBBs. Consequently, the benefits of this technique can be harnessed by any R-tree variant.

In summary, the contributions of this thesis work are as follows:

1. Development of a data-driven, cost-based algorithm to construct Grid Minimum Bounding Boxes (GMBs).
2. Design and implementation of a grid bitmap compression technique to minimize the construction and storage costs associated with GMBs.
3. Implementation of an efficient maintenance system for GMBs with lazy updates.
4. Enhancement of search operations through the implementation of an intersection test for GMBs.
5. Comprehensive experimental validation to demonstrate the effectiveness and robustness of the proposed GMB methodology.

The remainder of this thesis is organized as follows. Section 2 describes the background knowledge including R-tree, Minimum Bounding Boxes (MBBs) and some defined concepts used for the rest of the thesis. Section 3 introduces the motivation behind the Grid Minimum Bounding Boxes (GMBs) technique as well as logical and physical designs of GMB. Section 4 follows with the algorithms for constructing GMBs, querying with GMBs, and updating GMBs. Section 5 presents a comprehensive set of experiments and analyses, evaluating the performance and effectiveness of the proposed method. Section 6 provides a thorough review of related work, highlighting key developments and methodologies in the field of spatial indexing. Finally, Section 7 concludes the thesis by summarizing the findings and discussing potential directions for future research works.

Chapter 2

Background

This chapter provides the necessary background knowledge for the rest of the thesis. In the first subchapter, we introduce the R-tree data structure. Then, we look into the concept of Minimum Bounding Box (MBB) with applications. Following this, we list definitions and notations which will be used in subsequent chapters.

2.1 R-tree

The R-tree [23] is a dynamic index structure designed to efficiently manage and query spatial data on disk. Structurally, the R-tree is a balanced tree similar to a B-tree, but it is specifically adapted for spatial data. Each node in an R-tree represents a subgroup of closely related entries and associates with a Minimum Bounding Box (MBB) which encloses all objects. As data is inserted or removed, the R-tree dynamically adjusts by splitting or merging nodes to maintain balance and to optimize query performance for disk accesses. The R-tree supports various spatial queries, common query types include point queries, range queries, nearest neighbor searches, and spatial joins.

Since R-tree being proposed by Guttman in 1984, many variants of R-tree had been introduced in the following decades. All members of R-tree family keep the traditions of constructing a hierarchical data structure consists of recursive subgroups of objects, but they proposed different methodologies for local clustering to form nodes during insertions.

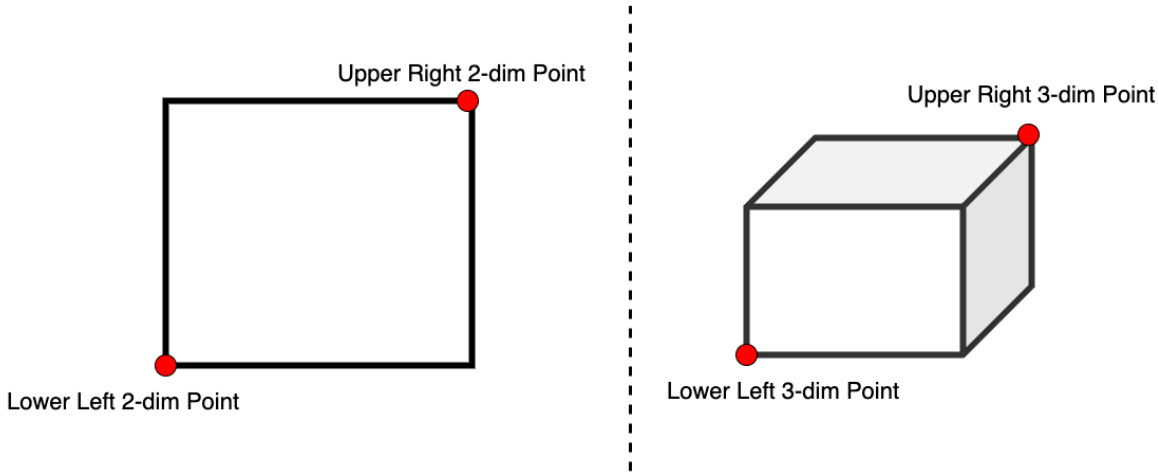
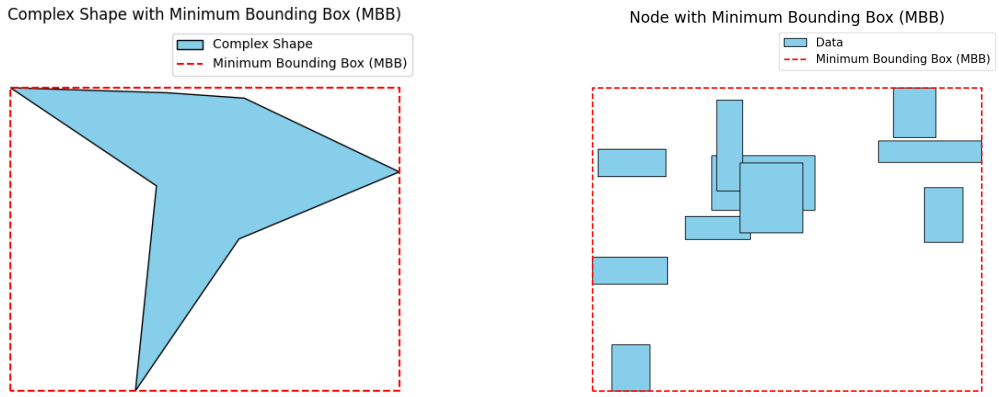


Figure 2.1: Unique Identification for Axis-aligned Hyper-rectangle

2.2 Minimum Bounding Box

Minimum Bounding Box (MBB), also known as Minimum Bounding Rectangle (MBR), is defined as the smallest box that can completely contain a given set of points or geometric objects. The sides of the MBB are parallel to the coordinate axes, making it axis-aligned which allows more compact storage. MBB representation requires the fixing of only two high-dimensional points as shown in Figure 2.1. Only lower left and upper right points are required to uniquely identify the MBB in space, such as $< LowerLeftPoint, UpperRightPoint >$. There exists two common use cases for MBB in the context of multi-dimensional spatial data. One of such use case is that we use MBB to approximate complex spatial objects. In Figure 2.2(a), the red dotted line is the Minimum Bounding Box of the complex shape where precise representation would require 6 hyper-planes. Varied complex shapes would require representation with varied sizes. MBB here can help as a size regulator which provides an approximation with fixed size. Another use case of MBB is to represent a group of all subgroups or spatial objects contained within it. In Figure 2.2(b), the red dotted line represents the MBB of ten rectangular data objects. In this case, the MBB serves as an approximation of all enclosed objects, offering the advantage of lower storage cost.

In the context of R-trees, the Minimum Bounding Box (MBB) plays a crucial role as a building block. An R-tree is a hierarchical data structure that utilizes MBBs efficiently to manage and query multi-dimensional spatial objects. Each node in the R-tree, whether a branch node or a leaf node, is associated with an MBB that contain all the child nodes or spatial objects within it. This bounding box serves as a spatial pre-filter, allowing the



(a) MBB for Complex Shape

(b) MBB for Subgroups of Objects

Figure 2.2: Use Cases for Minimum Bounding Box

R-tree to quickly eliminate irrelevant nodes during search operations. When performing queries, such as point searches or range searches, the R-tree leverages the MBBs to determine potential intersections with the query region. If an MBB does not intersect with the query region as shown in True Negative Result in Figure 2.3, the entire subtree associated with that node can be safely skipped, thereby reducing the number of nodes that need to be accessed and improving query performance. However, while MBBs are effective as a pre-filter for the search query, they can also lead to false positives. As shown in False Positive Result in Figure 2.3 where an MBB intersects with the query region but none of the enclosed objects do. This necessitates further examination of individual objects within the node, which results in additional unnecessary leaf node access. Despite this, the use of MBBs in R-trees remains a powerful technique for managing and querying spatial data efficiently

As discussed earlier, poorly chosen MBBs can result in a high false positive rate during intersection tests. High false positive rate can lead to high performance cost of disk accesses. R-tree variants address this issue by clustering objects while considering various characteristics of MBBs. The most commonly considered characteristics are coverage, dead space, overlap, and perimeter, as illustrated in Figure 2.4. From this figure, we can observe that dead space is actually a subset of coverage, similar to overlap. Overlap can

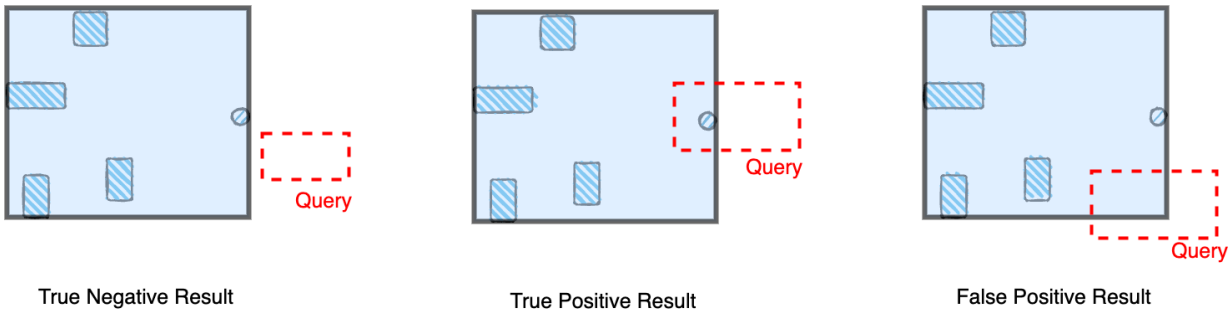


Figure 2.3: Three Different Cases for Intersection Test of MBB

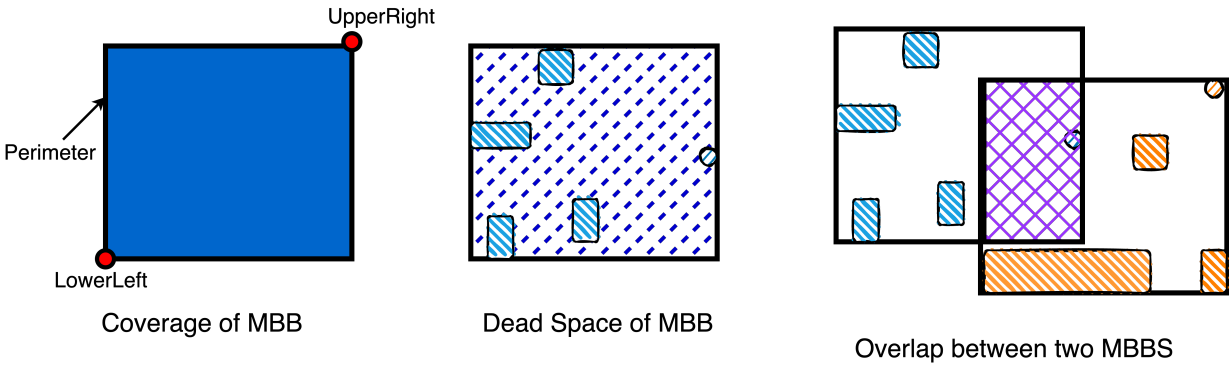


Figure 2.4: Visualization of Characteristics of MBBs

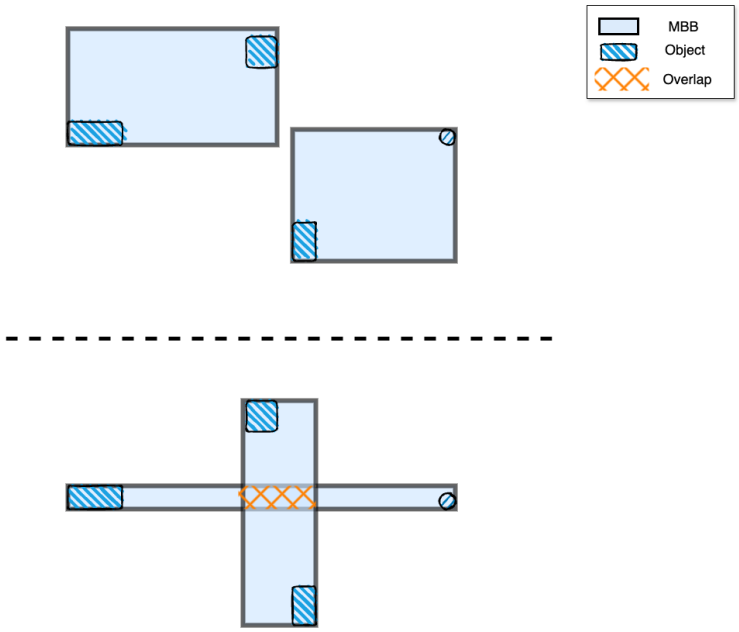


Figure 2.5: Comparison between Dead Space and Overlap

include portions of dead space as well. With a uniform query distribution, coverage, dead space, and overlap can all be expected to positively correlate with the false positive rate. This means that as the coverage area of the MBBs increases, or as the amount of dead space and overlap between MBBs grows, the likelihood of false positives also rises. Dead space refers to the portion of an MBB that does not actually contain any data objects but still may intersect with the query, leading to unnecessary node accesses. Similarly, overlap occurs when multiple MBBs cover the same region of space, which can cause the query to intersect with multiple nodes even though only one or none of them might contain relevant objects. This overlapping also increases the chances that a query will intersect with empty areas or irrelevant objects, thereby triggering false positives. Consequently, different R-tree variants employ various techniques to efficiently construct MBBs that reduce these properties. However, balancing these factors is not always straightforward. For example, in Figure 2.5, one clustering candidate exhibits zero overlap but more dead space, while another candidate shows much less dead space at the cost of some overlaps.

2.3 Formalization and Notations

In this subsection, we provide formal definitions and notations to support our discussions in the subsequent sections of this thesis.

Definition 1 (Minimum Bounding Box - MBB). *For a set of objects $O = \{o_1, o_2, \ldots, o_m\}$, the minimum bounding box of O, denoted by MBB(O), is the tightest axis-aligned hyper-rectangular boundary that contains all the objects.*

An axis-aligned hyper-rectangle can also be represented by two points such that $\text{MBB}(O) = \langle LL, UR \rangle$, where LL (lower left) and UR (upper right) are the minimized and maximized corners with the following definitions:

$$LL[d] = \min_{o \in O} o[d] \quad \text{for all dimensions } d$$
$$UR[d] = \max_{o \in O} o[d] \quad \text{for all dimensions } d$$

Definition 2 (Split). *For a given n-dimensional hyper-rectangle R, the split of R is denoted as $S(R) = \{s_0, s_1, \ldots, s_{n-1}\}$, where s_i is the number of splits along the i-th dimension, and i ranges over all dimensions. R is equally split along each dimension.*

Definition 3 (Grid). *Let S be the split for an n-dimensional hyper-rectangle R. The smallest sub-region of R resulting from this division is referred to as a grid.*

Definition 4 (Grid Location). *For a given n-dimensional hyper-rectangle R divided by split S, the grid location is the unique representation of each grid in the space, represented as an n-sized tuple $(i_0, i_1, \ldots, i_{n-1})$. i_j is the index of the grid along the j-th dimension, ranging from 0 to $s_j - 1$.*

Definition 5 (Grid ID). *For a given n-dimensional hyper-rectangle R divided by split S, each grid within R is assigned a unique grid ID (GID). GID is defined using a one-to-one mapping from grid location to an integer ID ranging from 0 to the number of divided grids. Let $(i_0, \ldots, i_{n-1})$ be the grid location of interest; the GID can be calculated with the formula:*

$$\text{GID}(i_0, i_1, \ldots, i_{n-1}) = \sum_{j=0}^{n-1} \left(i_j \times \prod_{k=j+1}^{n-1} s_k \right)$$

Definition 6 (Grid Bitmap). *For a given set of objects O, the grid bitmap, denoted by*

GBitmap, is constructed on top of MBB(O). Each bit in the bitmap indicates whether a grid intersects with any objects from O. The GID of each grid is used as the key for the grid bitmap. If a grid does not intersect with any objects, it is referred to as a dead grid.

$$\text{GBitmap}[\text{GID}(G)] = \begin{cases} 0 & \text{if} \quad \forall o_i \in O,\ o_i \cap G = \emptyset \\ 1 & \text{if} \quad \exists o_i \in O,\ o_i \cap G \neq \emptyset \end{cases}$$

Definition 7 (Grid Minimum Bounding Box). *For a given set of objects O, the grid minimum bounding box of O, denoted by GMB(O), is MBB(O) augmented with a grid bitmap, such that GMB(O) = $\langle$MBB(O), GBitmap(O)$\rangle$.*

Definition 8 (Useful Leaf Node Access). *Given a leaf node (LN) containing a set of objects O and a query Q, if the LN is accessed to answer Q and contains at least one object that answers the query, this leaf node access is considered useful.*

Definition 9 (Wasted Leaf Node Access). *Given a leaf node (LN) containing a set of objects O and a query Q, if the LN is accessed to answer Q but does not contain any objects that answer the query, this leaf node access is considered wasted. In other words, accessing this LN for the given Q is wasteful.*

Definition 10 (Dead Space). *For a minimum bounding box B of a set of objects O, the dead space of MBB, denoted by DS(B), is the union of all subregions in B that do not contain any objects from O. It can be formalized as:*

$$\text{DS}(B) = \{p \mid p \in B \text{ and } p \notin o_i \text{ for any } o_i \in O\}$$

Definition 11 (Overlap). *For a set of minimum bounding boxes $BSet = \{B_0, B_1, \ldots, B_m\}$, the overlap of a chosen B_i, denoted by Over(B_i), is the union of all sub-regions within B_i that intersect with any other B_j within BSet. It can be formalized as:*

$$\text{Over}(B_i) = \{p \mid p \in B_i \text{ and } \exists B_j, p \in B_j \text{ where } i \neq j\}$$

Definition 12 (Coverage). *For a minimum bounding box B, coverage refers to the hyper-volume of B, which is the total space that B encompasses in its respective dimensional space.*

Chapter 3

Grid Minimum Bounding Box Design

This chapter outlines the design of the Grid Minimum Bounding Box (GMB) integrated with the R-tree. We begin with a discussion of the motivation behind the GMB, emphasizing how the grid bitmap is used to index dead space within a leaf node. Following this, we specify both the logical and physical layouts of the GMB, functioning as an auxiliary structure stored in main memory. To optimize storage and performance, two different data structures based on the size of the grid bitmap are employed in the design of layouts.

3.1 Motivation of Grid Minimum Bounding Box

In this section, we explain the motivation behind the Grid Minimum Bounding Box (GMB) with a visualized example. First, let us consider how an R-tree can be used to answer a range query. Suppose we want to build an R-tree to index eight objects in a 2-dimensional space, with a maximum fan-out of five and a minimum fan-out of three. This means that, except for the root node, every node should contain at least three objects and at most five objects. As shown in Figure 3.1, the eight objects on the left are partitioned into two subgroups, resulting in leaf nodes A and B, which together form a two-level R-tree on the right. Now, consider a range query, depicted by the red dotted line on the left side of Figure 3.2. The query intersects with the MBB of root node and also with both MBBs of Node A and Node B. Therefore, both leaf nodes must be accessed to check if any objects within the nodes match the query. As shown in the Figure 3.2, only objects 6 and 7 in Node B intersect with the query, while none of the objects in Node A do. As a result, accessing Node B is useful, whereas accessing Node A results in a wasted leaf node access.

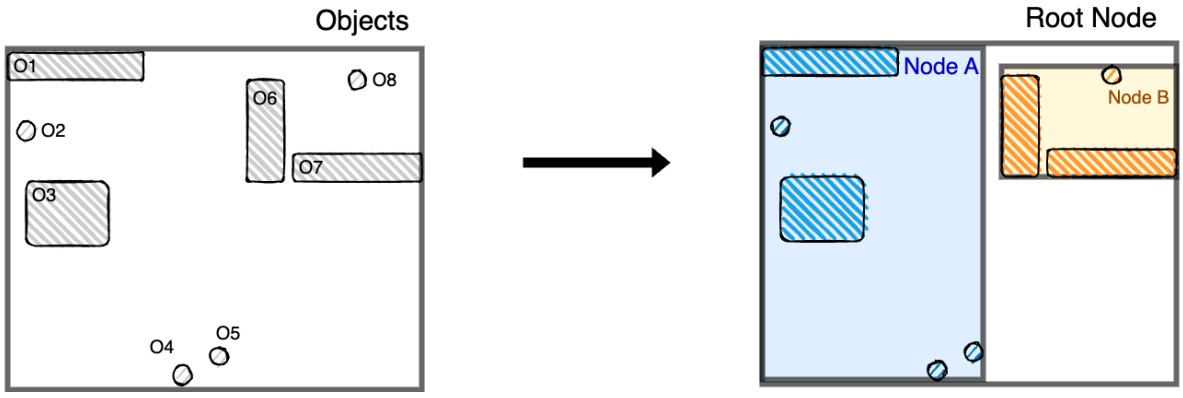


Figure 3.1: An Example of two-level R-tree with 8 two-dimensional objects

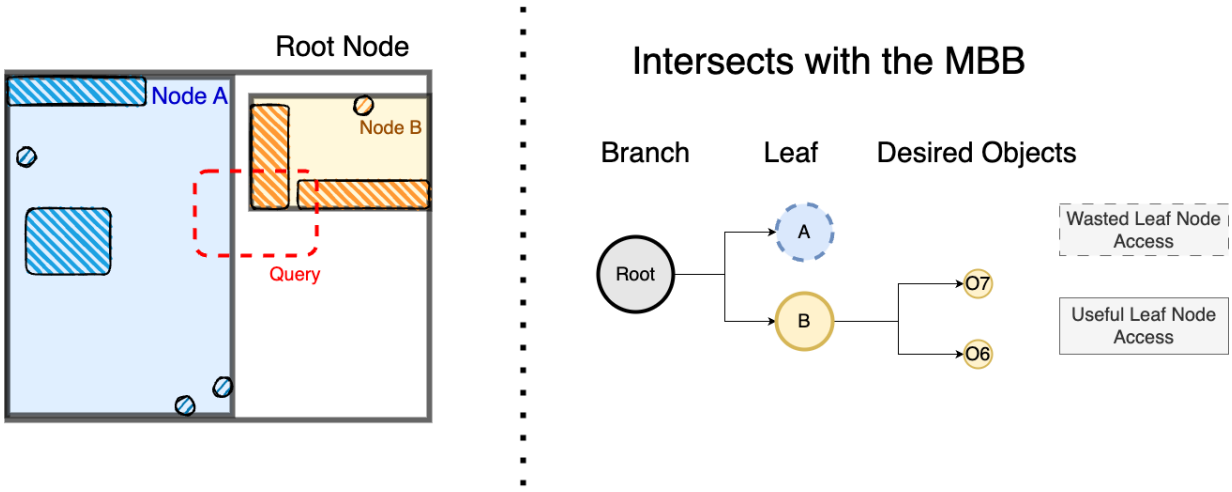


Figure 3.2: An Example of Range Query on R-tree based on MBB

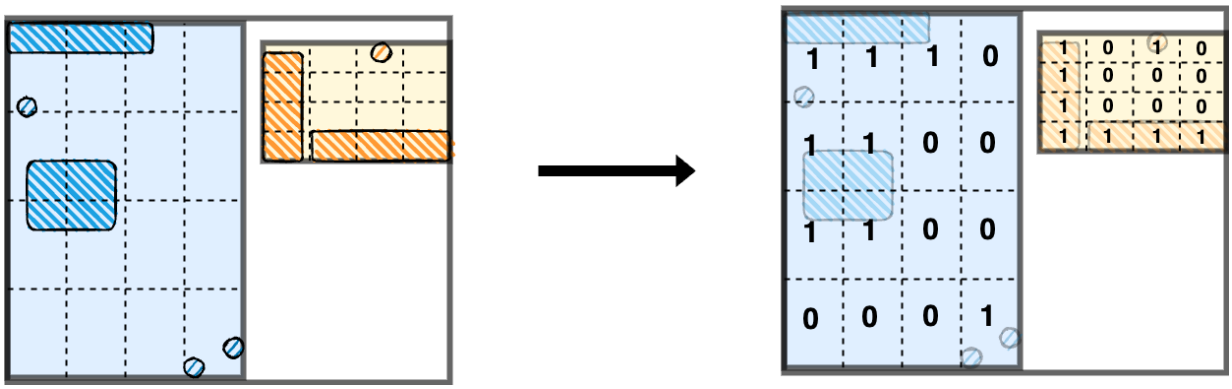


Figure 3.3: An Example of Grid Bitmap on R-tree

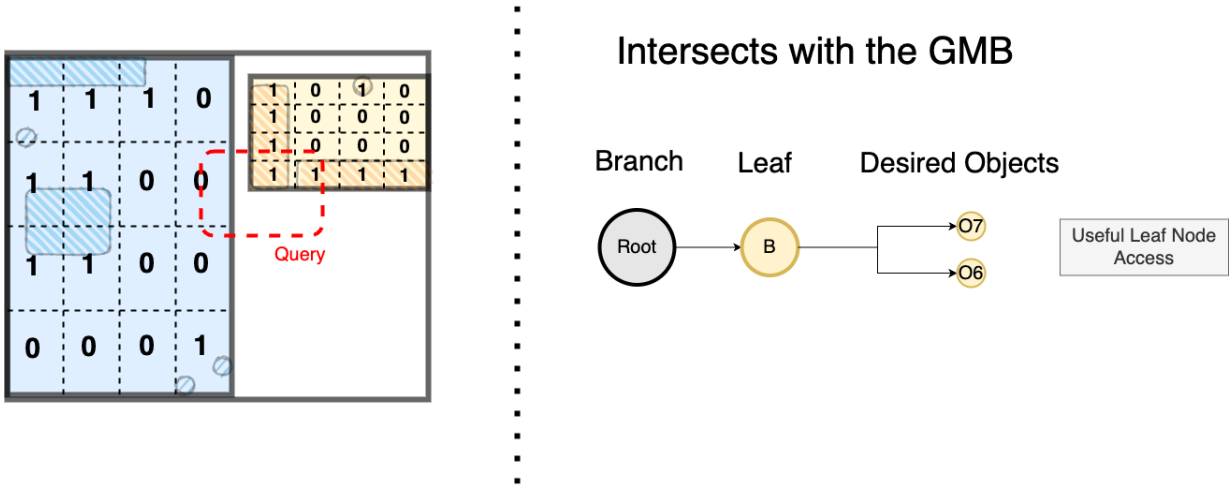


Figure 3.4: An Example of Range Query on R-tree based on GMB

The wasted access occurs because the intersection test between the MBB of Node A and the query yields a false positive, as the query intersects only the dead space within Node A. Intuitively, the presence of significant dead space within an MBB can increase the likelihood of false positives during queries, subsequently degrading query performance, as discussed in the Background chapter. At a high level, we can envision the Grid Minimum Bounding Box (GMB) as placing a grid over the MBB and indexing whether each grid cell contains only dead space. This approach provides a high-level summary of the spatial distribution of objects using a memory-efficient data structure, where only one bit is needed to store information for each grid cell. This process is visualized in Figure 3.3. The left diagram in this figure shows how the MBB is divided into grids by equal splits, and the right diagram uses one bit per grid cell to label the presence of objects.

When the MBB fails to prevent false positive searches due to dead space, the grid bitmap augmented on the MBB serves as a secondary check to eliminate unnecessary accesses. Therefore, it helps to reduce costly leaf node accesses and disk I/O operations. The intersection test on the GMB begins with the standard MBB intersection check. If this check returns true, the GMB then performs an additional step to verify whether all intersected grid cells are dead space. If all intersected grids are dead, the intersection test returns false, filtering out some of the false positives produced by the MBB alone. As shown in Figure 3.4, the query intersects with the MBBs of both Nodes A and B. Since the two intersected grids in Node A are marked as 0 (indicating dead space), the GMB intersection test for Node A returns false. This allows us to save the leaf node access to Node A. Conversely, since some (3) of the four intersected grids in Node B are marked with 1, the GMB intersection test for Node B returns true.

Essentially, the grid bitmap captures the distribution information of objects on the leaf node in a multi-dimensional manner while the node is still in memory. It then efficiently stores this additional distribution data as a bitmap in memory. When a leaf node is accessed by a query, this auxiliary main memory-based information helps avoid unnecessary disk I/O on the leaf node. The grid bitmap provides more precise information than just the MBB. The primary benefit arises because the bottlenecked disk I/O, a dominant query performance factor in disk-based R-trees, is efficiently trade-off with cheap CPU computation by checking the grid bitmap in our proposed method.

3.2 Logical Layout

To provide an explanation of the logical design of an R-tree built on GMBs, we use the dataset from Figure 3.1 and the GMBs constructed in Figure 3.3 as examples. The logical

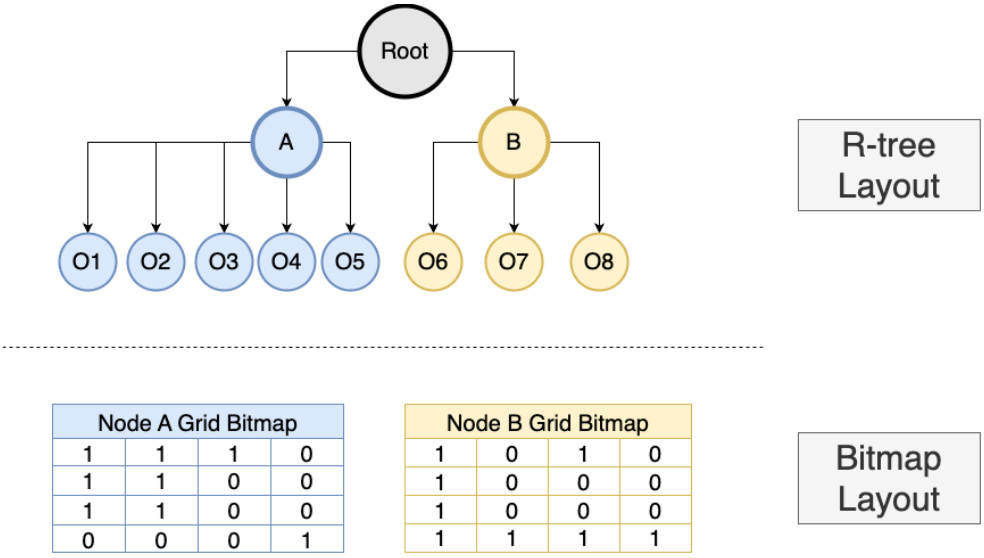


Figure 3.5: Logical Layout of R-tree with GMB in 3.1

layout of the R-tree with GMBs is illustrated in Figure 3.5. As shown in the figure, the layout comprises two distinct parts: a standard R-tree structure and a collection of grid bitmaps each paired with one leaf node. An R-tree is fundamentally a hierarchy of branch nodes and leaf nodes, with leaf nodes located exclusively at the lowest level as shown in the top subgraph. The grid bitmaps are stored separately as an auxiliary structure, independent of the R-tree. This separation preserves the original layout of the R-tree, allowing for seamless integration of GMB with any R-tree variants without altering the core architecture. Specifically, the grid bitmap associated with each leaf node is maintained in a separate data structure, facilitating efficient construction and updates at the node level.

3.3 Physical Layout

In this section, we continue with the same example to describe how the GMB-based R-tree is stored. In Figure 3.6, the top subgraph illustrates the storage structure of a standard R-tree using MBBs. At the storage level, a leaf node consists of an array of $\langle \text{MBB(data)}, \text{Pointer(data)} \rangle$ for each data object it contains, while a branch node consists of an array of $\langle \text{MBB(node)}, \text{Pointer(node)} \rangle$ for each node it contains. For a disk-based R-tree, each node is stored on a single disk page, where node can be either leaf node or

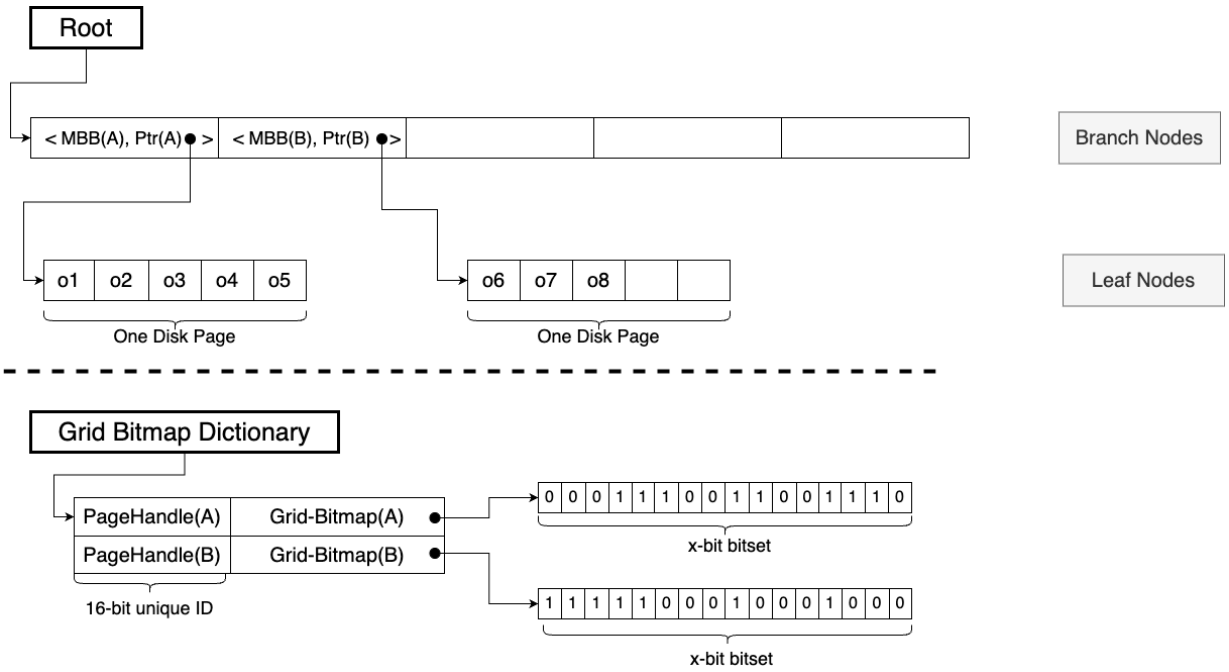


Figure 3.6: Physical Layout of R-tree with GMB in bitset

	Split at each dim	Required Bitset Size (bits)	Required Bitset Size (bytes)
2 dimensions	2^3	2^6	8
	2^4	2^8	32
	2^5	2^{10}	128
	2^8	2^{16}	8192
3 dimensions	2^2	2^6	8
	2^3	2^9	64
	2^4	2^{12}	512
	2^8	2^{24}	2097152

Table 3.1: Bitmap Size Example

branch node. All nodes are visualized with a maximum capacity of 5 objects per page in this example. The bottom subgraph shows a dictionary of grid bitmaps, where one grid bitmap is paired with one leaf node of interest. This dictionary is stored in main memory to allow for easy updates and queries. We use the Page Handler, which is unique to each leaf node, as the key in the dictionary, and we store the bitset corresponding to the grid bitmap as the value. The size of the bitset is chosen adaptively, based on either default configuration or node characteristics. In this project, we limit the construction of grid bitmaps to leaf nodes only; the reasons for this limitation will be discussed in the later chapter on experiments and analysis.

With more splits along each dimension, the grid bitmap can capture more precise information about dead space. However, this comes at the cost of increased storage requirements in main memory. As shown in Table 3.1, for two-dimensional dataset with 2^8 splits per dimension, a bitset of 8192 bytes is required for each leaf node which is too much overhead. When dealing with datasets of point objects, there is an upper bound on the number of grids occupied in the grid bitmap. Since points are non-volumetric data, each point can intersect with only one grid in the grid bitmap. Therefore, the upper bound of occupied grids corresponds to the maximum fan-out of the R-tree. Knowing that most grids will be dead grids, we can optimize storage by using a vector to store only the locations of occupied grids instead of indexing every grid in a large bitset. For example, with 2^8 splits per dimension, 8 bits can store the grid index for each dimension. For a two-dimensional point dataset with a maximum fan-out of 80, the upper bound of storage requirement would be $80 \times 8 \times 2$ bits (160 bytes). This is significantly less than the 8192 bytes required by the bitset for a large number of splits. Additionally, this structure is independent of the number of splits per dimension, depending only on the size of the data structure used to store grid locations. The physical storage is shown with details in Figure 3.7 for comparison.

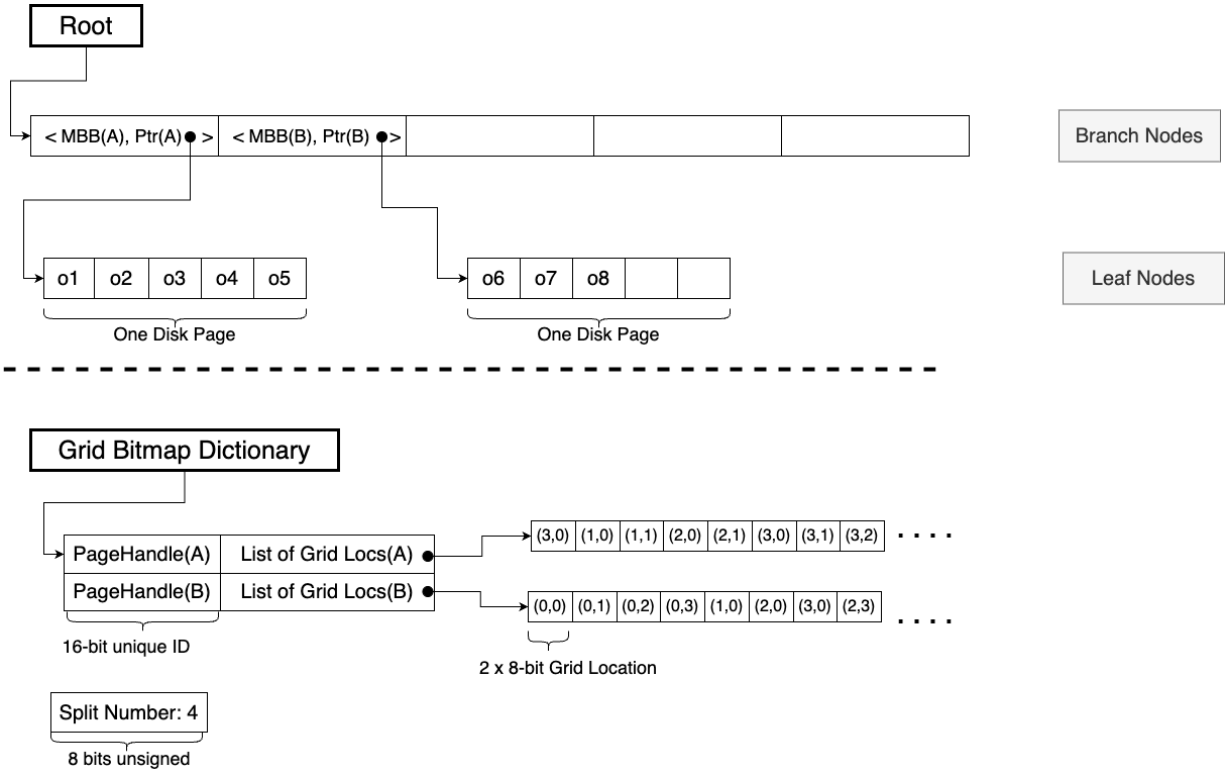


Figure 3.7: Physical Layout of R-tree with GMB in list of grid location

Chapter 4

Grid Minimum Bounding Box Algorithms

This chapter provides a detailed description of the algorithms for constructing, querying, and updating a Grid Minimum Bounding Box (GMB). As discussed in the Design chapter, there are two physical representations for grid bitmaps: the dynamic bitset and the list of occupied grid locations. Depending on the available storage budget, one representation may outperform the other in terms of storage efficiency. Construction algorithms are provided for both representations. Additionally, we explain how the bitmap can be built adaptively based on the observed overlap and dead space. The adaptive algorithm is designed to strike a balance between space efficiency and query processing performance. We then describe how intersection tests are conducted using GMBs and how queries are processed for both bitmap representations. Finally, we outline the methods for updating GMBs as the underlying spatial data changes, ensuring that the grid bitmap remains accurate and continues to enhance query performance over time.

4.1 Construction

With a chosen bitmap size, constructing a grid bitmap is straightforward. The process for constructing a grid bitmap using a dynamic bitset is detailed in Algorithm 1. First, we determine the appropriate *Split* for each dimension based on the given bitmap storage budget. We then iterate through all the objects contained in the leaf node, calculate each object's *GridLocation* using Algorithm 5, which is based on the MBB and *Split*, and

convert this *GridLocation* into a $GridID$ using Algorithm 4. The corresponding $GridID$ is then set to true in the grid bitmap. After constructing the grid bitmap, we assess the percentage of grid cells that contain dead space to evaluate the effectiveness of the *GridBitmap*. Specific for Line 5, calculating the $GridID$ for a point is straightforward, as a point can only intersect with a single grid. Similarly, determining the $GridIDList$ for a rectangle is uncomplicated. As shown in Algorithm 5, we simply calculate the grid locations for the $UpperRight$ and $LowerLeft$ points, and include all grid locations within these boundary grids.

The construction of a grid bitmap using a list of *GridLocation* is described in Algorithm 2. This process is very similar to Algorithm 1, except that the grid bitmap is now represented as a set of *GridLocation* at Line 3, and there is no need to convert *GridLocation* into $GridID$.

In this thesis, we also explore the adaptive construction of grid bitmaps based on observed overlap and dead space, as shown in Algorithm 3. We begin by constructing a small grid bitmap to gather information about dead space at Line 5. Based on collected information on dead space percentage from Line 6 and overlap percentage with sibling MBBs from Line 3, we evaluate if a larger grid bitmap could be more effective at indexing data objects. If we determine that a larger bitmap would be beneficial, we recreate the bitmap with a larger size as shown at Line 11-14. Consequently, different leaf nodes may have grid bitmaps with varying sizes, depending on their specific characteristics. This adaptive approach allows us to achieve a relatively balanced trade-off between storage cost and query efficiency.

4.2 Query

As the R-tree structure remains intact, queries can traverse the hierarchy of branch nodes in the same manner as the original structure with MBBs. This ensures that the application of the GMB introduces minimal changes. The code modification for the search function is limited to a single function call at the leaf node level, replacing the intersection test on MBB with the intersection test on GMB.

The intersection test with the GMB is straightforward and efficient, as demonstrated in Algorithm 6. At Lines 2-3, a standard MBB intersection test is conducted as a basic case. Since MBB has a true negative guarantee, returning False is appropriate. If the *GridBitmap* does not exist, True is returned at Lines 4-5. We consider both representation in this algorithm, where Line 11-17 describes the logic of examing bitset based *GridBitmap*

Algorithm 1 Construct Grid-Bitmap with dynamic bitset

```
1: procedure GridBitmap(O, MBB, BitmapSize, dimension)
2:     Split[d] ← BitmapSize^(1/dimension) for d = 0 ... dimension − 1
3:     GridBitmap ← initialized bitset as false with size BitmapSize
4:     for i ← 1 to |O| do
5:         GLocList ← GridLocationHelper(o_i, MBB, Split)
6:         for all gridLocc ∈ GLocList do
7:             GID ← CalculateGIDHelper(gridLoc, Split)
8:             GridBitmap.set(GID, true)
9:         end for
10:    end for
11:    if DeadGridPct(GridBitmap) ≤ β then
12:        return
13:    else
14:        return GridBitmap
15:    end if
16: end procedure
```

Algorithm 2 Construct Grid-Bitmap with grid location

```
1: procedure GridBitmap(O, MBB, BitmapSize, dimension)
2:     Split[d] ← BitmapSize^(1/dimension) for d = 0 ... dimension − 1
3:     GridBitmap ← initialized as empty set of GridLocation
4:     for i ← 1 to |O| do
5:         GLoc ← GridLocationHelper(o_i, MBB, Split)
6:         Add GLoc to GridBitmap
7:     end for
8:     if DeadGridPct(GridBitmap) ≤ β then
9:         return
10:    else
11:        return GridBitmap
12:    end if
13: end procedure
```

Algorithm 3 Construct Grid-Bitmap Adaptively

```
1: procedure AdaptiveGridBitmap(O, MBB, dimension)
2:     TotalSiblingOverlap ← Overlap(MBB, SiblingMBBs)
3:     OverlapPct ← TotalSiblingOverlap/Coverage(MBB)
4:     BitmapSize ← DefaultBitmapSize
5:     GridBitmapCandidate ← GridBitmap(O, MBB, BitmapSize, dimension)
6:     DeadGridPct ← DeadGridPct(GridBitmapCandidate)
7:     if OverlapPct ≤ α1 or DeadGridPct ≤ β1 then
8:         return
9:     else if OverlapPct ≤ α2 and DeadGridPct ≤ β2 then
10:        return GridBitmapCandidate
11:    else if OverlapPct ≤ α3 and DeadGridPct ≤ β3 then
12:        return GridBitmap(O, MBB, BitmapSize * factor, dimension)
13:    else
14:        return GridBitmap(O, MBB, BitmapSize * factor * factor, dimension)
15:    end if
16: end procedure
```

Algorithm 4 Convert Grid Location to GID

```
1: procedure CalculateGIDHelper(GridLoc, S)
2:     GID ← 0
3:     for i ← 1 to dimension n do
4:         product ← 1
5:         for j ← i + 1 to dimension n do
6:             product ← product × S[j]
7:         end for
8:         GID ← GID + GridLoc[i] × product
9:     end for
10:    return GID
11: end procedure
```

Algorithm 5 Calculate Grid Location of Rectangle-type Entry within a given MBB

procedure GridLocationHelper(R, MBB, S)
    $LowerGrid \leftarrow empty$
    $UpperGrid \leftarrow empty$
    $R_{LL} \leftarrow max(R.lowerLeft, MBB.lowerLeft)$
    $R_{UR} \leftarrow min(R.upperRight, MBB.upperRight)$
    for $i \leftarrow 1$ to dimension n **do**
        $gridLength \leftarrow (MBB.UR[i] - MBB.LL[i])/(S[i])$
        $LowerGrid[i] \leftarrow \left\lfloor \frac{R_{LL}[i] - MBB.LL[i]}{gridLength} \right\rfloor$
        $UpperGrid[i] \leftarrow \left\lfloor \frac{R_{UR}[i] - MBB.LL[i]}{gridLength} \right\rfloor$
    end for

    ▷ The following is for including all grids between Lower and Upper Grid Indices
    $GridLocationList \leftarrow \{LowerGrid\}$
    for $i \leftarrow 1$ to dimension n **do**
        for all $gridLoc \in GridLocationList$ **do**
            $gridLocCopy \leftarrow gridLoc$
            for $k \leftarrow LowerGrid[i]$ to $UpperGrid[i]$ **do**
                $gridLocCopy[i] \leftarrow k$
                $GridLocationList.append(gridLocCopy)$
            end for
        end for
    end for
    return $GridLocationList$
end procedure

and Line 19-24 describes the logic for checking grid location based *GridBitmap*. At either section, the *GridBitmap* is utilized as an attempt to eliminate the false positive case by checking if Query Q intersects only with dead grids. At Line 9, the *GridLocation* procedure is called to find the locations of all grids in the MBB that intersect with query Q. For the bitset case, if all of the query-intersected grids are labeled as False in the *GridBitmap*, then False is confidently returned at Line 17, indicating that Query Q does not intersect with any occupied grids. For the case of grid location, if none of query-intersected grids exists in the set, then we know query doesn't intersect with any occupied grids and False can be returned at Line 24. Generally, utilizing *GridBitmap* can help avoiding false positive results given by intersection tests of traditional MBBs.

At Line 11 or 19, the loop size is determined by the number of grids intersecting with Query Q, which is positively correlated with the volume of $Q \cap MBB$ relative to the volume of the MBB. When the volume of the query is large relative to the MBB (close to 1.0), less performance benefit is gained as it is less likely for all grids to be dead. Therefore, a computation optimization with a volume ratio threshold is introduced at Line 6. The threshold is set to 0.85.

For point queries, the size of *gridLocList* at Line 9 is always 1. For range queries, the loop size is upper-bounded by the size of the bitmap. Therefore, the largest performance improvement from the grid bitmap is expected for point or small range queries, which are very common query types in spatial data management [24, 6]. The experiments also indicate that for large range searches, the differences among various clustering heuristics or techniques gradually diminish.

4.3 Update

When a data entry is inserted or deleted in an R-tree, the update is reflected in all accessed branch nodes and at least one leaf node containing the target data entry. Multiple leaf nodes may need to be updated to satisfy the fan-out constraint. An overflowed node needs to be split when it exceeds the maximum fan-out size, and a small node needs to be redistributed to other nodes or deleted if it falls below the minimum fan-out size.

The MBB may change during these updates. An expansion of the MBB for insertion or a contraction of the MBB for deletion is possible. When an MBB is updated due to changes in the content of the leaf node, a reconstruction of the grid bitmap with one of Algorithms 1 or 2 is necessary to reflect the new distribution of dead space on the leaf node. Compared to updates on leaf node content which involve disk I/O to retrieve leaf node, updating grid bitmap, a memory-resident data structure, is relatively inexpensive.

Algorithm 6 Intersection Test between GMB and Query

```
1: procedure IntersectionTest(MBB, GridBitmap, S, Q)
2:     if MBB ∩ Q = ∅ then
3:         return False
4:     else if GridBitmap == empty then
5:         return True
6:     else if Vol(Q)/Vol(MBB) ≥ RatioThreshold then
7:         return True
8:     else
9:         gridLocList ← GridLocation(Q, MBB, S)
10:        if GridBitmap is bitset then
11:            for all gridLoc ∈ gridLocList do
12:                GID ← CalculateGID(gridLoc, S)
13:                if GridBitmap.at(GID) == True then
14:                    return True
15:                end if
16:            end for
17:            return False
18:        else if GridBitmap is GridLocationSet then
19:            for all gridLoc ∈ gridLocList do
20:                if gridLoc ∈ GridBitmap then
21:                    return True
22:                end if
23:            end for
24:            return False
25:        end if
26:    end if
27: end procedure
```

There are two scenarios in which the MBB does not change after an update query: (1). When the inserted entry lies entirely within the existing MBB and the fan-out constraint is still satisfied after insertion. No change in the MBB is expected, as no expansion is required. (2). When the deleted entry is not the only object defining the boundary of the MBB. In other words, if the entry being removed does not determine the MBB's boundary, the MBB does not need to be recalculated. In these simpler cases, where the MBB remains unchanged after a leaf node content update, there is no need to reprocess all objects within the leaf node to update the grid bitmap; only the object being inserted or deleted needs to be processed.

4.3.1 Insert

Insertion increases the number of occupied grids, but it only affects the grids intersected with the inserted entry. Lines 5-9 of Algorithm 1 or Lines 5-6 of Algorithm 2 can be used to update the existing grid bitmap. With the information of object to be inserted, we just need to find all grid locations of the inserted entry and mark them as occupied grids.

4.3.2 Delete

When a data entry is removed, its occupied grids cannot be confidently turned into dead grids, as other objects may intersect with the same grids. One approach is to process grid bitmap updates caused by deletions lazily. Deletion only turns occupied grids into possibly dead grids. The knowledge of occupied grids remains accurate, and the current bitmap will still yield accurate query results. Therefore, updating such grid bitmaps can be delayed until a reconstruction caused by an expansion or contraction of the MBB.

Chapter 5

Experiment Evaluation

This chapter evaluates the performance of Grid Minimum Bounding Box (GMB) for multiple factors through detailed experiments with variations that includes datasets, data types, R-tree variants, variable sizes of grid bitmaps, different construction methods, and storage methods. We study the performance of GMB through multiple angles as well, including dead space elimination, query performance improvement on three different types of query, overheads brought by storage and construction. We evaluated the GMB technique on several popular R-tree variants using a well-established multi-dimensional index benchmark [10]. To emphasize the advantages of the proposed technique, we also conduct experiment on larger scale real-life datasets, including Tweets and Geolife [19]. This analysis focuses on assessing the effectiveness of the GMB in eliminating dead space, enhancing query performance, and evaluating the overheads compared to R-tree variants utilizing standard MBBs.

5.1 Setups

We implemented three widely-used variants of the R-tree for our benchmark in C++: the standard R-tree, the R*-tree, and the Revised R*-tree (RR*-tree). We maintained a universal abstraction across all R-tree variants, with variant-specific implementations of the *SplitNode* and *ChooseSubtree* methods. The configuration for each tree's fan-out is detailed in Table 5.1, in accordance with recommendations from previous studies based on 4KB disk page [6]. Notably, the RR*-tree is designed with one fewer on maximum fan-out to accommodate an additional counter in its structure.

In our experiment setup, we varied the splits per dimension: 2^3, 2^4, 2^5 for two-dimensional datasets and 2^2, 2^3, 2^4 for three-dimensional datasets. This variation resulted in grid bitmap of sizes 64, 256, and 1024 for two dimensions and 64, 512, and 4096 for three dimensions. Additionally, we introduced a grid bitmap with 2^8 splits per dimension for only point datasets to evaluate the performance of grid bitmap with representation of grid locations. Other than constructing grid bitmap with pre-configured size as mentioned above, we also tested the performance of building grid bitmap adaptively based on characteristics of MBBs.

Given that the R-tree is a disk-based index, we opted to use the leaf node access count as our primary performance metric. This measure is considered more equitable and robust as it is unaffected by variations in buffer pool size, storage layout, bandwidth, and underlying storage hardware. We did not count branch node accesses, reasoning that branch nodes, being hierarchically higher, are more likely to remain resident in memory. Although we did not assess potential enhancements from applying the grid bitmap to branch nodes (aligning with methodologies from prior studies [45]), we anticipate further performance improvements could be realized, given the frequent access of branch nodes.

		R-tree	R*-tree	Revised R*-tree
2 dimensions	Node Max Fan-out	83	83	82
	Node Min Fan-out	40	40	40
3 dimensions	Node Max Fan-out	61	61	60
	Node Min Fan-out	30	30	30

Table 5.1: R-tree Variants Fan-out

5.1.1 Datasets

To examine the effectiveness of the Grid Minimum Bounding Box (GMB), we conducted benchmarks using both synthetic and real-world datasets. The characteristics of these datasets are summarized in Table 5.2. Given the extensive variety of datasets, we categorized them into four subgroups for clarity and structured analysis: (1) two-dimensional point datasets, (2) two-dimensional mixed datasets, (3) three-dimensional point datasets, and (4) three-dimensional rectangle datasets. We performed tests across all dataset subgroups to evaluate the impact of the GMB, with results presented accordingly.

Data Set	Dimension	Data Type	Data Source	Size	Dead Space on R*	Input Order
abs02	2	Rectangle	Artificial	1,000,000	0.41	row ordered
abs03	3	Rectangle	Artificial	1,000,000	0.58	row ordered
bit02	2	Point	Artificial	1,000,000	1.0	random
bit03	3	Point	Artificial	1,000,000	1.0	random
dia02	2	Rectangle	Artificial	1,000,000	0.96	dithering ordered
dia03	3	Rectangle	Artificial	1,000,000	0.99	dithering ordered
par02	2	Rectangle	Artificial	1,048,576	0.77	z-order
par03	3	Rectangle	Artificial	1,048,576	0.92	z-order
ped02	2	Point	Artificial	1,034,621	1.0	random
ped03	3	Point	Artificial	1,034,621	1.0	random
pha02	2	Point	Artificial	1,025,397	1.0	multi-center ordered
pha03	3	Point	Artificial	1,025,397	1.0	multi-center ordered
uni02	2	Point	Artificial	1,000,000	1.0	random
uni03	3	Point	Artificial	1,000,000	1.0	random
rea02	2	Mixed	Real	1,888,012	0.76	random in subregion
rea03	2	Point	Real	11,958,999	1.0	random
Tweets	2	Point	Real	15,598,403	1.0	random
GeoLife	2	Point	Real	22,033,507	1.0	random

Table 5.2: Dataset Characteristics

5.1.2 Queries

For our analysis, we utilized three distinct types of queries with different selectivities, designated as Query 0, Query 2, and Query 3. Generally, Query 0 is a point query designed to return a single object from the dataset. Query 2, a more selective range query, is expected to return 10^2 (100) objects on average, while Query 3, the least selective range query, aims to retrieve 10^3 (1000) objects. The source of search queries on multi-dimensional datasets was retrieved from benchmarks [10], whereas queries applied to Tweets and Geolife datasets were synthesized as following. For Query 0, we randomly selected 100,000 objects from the dataset to serve as point search queries. For Queries 2 and 3, we began by randomly selecting 10,000 objects from the datasets. For each selected object, we conducted a k-nearest neighbor (KNN) search to identify the 100 and 1000 spatially closest objects, respectively. The results of these searches were then enclosed within a Minimum Bounding Box (MBB), which defined the range for Queries 2 and 3 in the experiments. For most of the datasets, queries are generated based on selectivity upon desired size of output. The exception is dataset $ped0x$ where queries are generated based on volume. Query 0 for $ped0x$ is generated with a volume as $1/totalObjects$ of the entire data space, where Query 2 is $100/totalObjects$ and Query 3 is $1000/totalObjects$ of the entire data space. The differences between datasets are summarized in Table 5.3.

		Tweets / GeoLife	ped0x	other datasets
Query 0	desired output size	1	volume-based	1
	query count	100,000	1,000,000	10% of data size
Query 2	desired output size	100	volume-based	100
	query count	10,000	1,000,000	1% of data size
Query 3	desired output size	1000	volume-based	1000
	query count	10,000	1,000,000	0.316% of data size

Table 5.3: Query Setups

5.2 Dead Space Elimination

In this section, we evaluate the effectiveness of the Grid Minimum Bounding Box (GMB) in reducing the dead space of leaf nodes. While the elimination of dead space does not directly represent improved query performance, it serves as a valuable indicator of clustering efficacy, as dead space is a critical metric in assessing object clustering effectiveness as explained in the previous Background chapter.

The baseline dead space for a leaf node is calculated using Equation 5.1. The equation's first component measures the coverage of the Minimum Bounding Box (MBB), calculated as shown in Equation 5.3. The latter part of Equation 5.1 computes the volume of the union of all overlapping objects, which is a more complex task. We employ the Line Sweep technique to calculate the union volume of a set of objects. Briefly, this technique involves maintaining an active set of objects intersecting with a sweep line, where the product of the intersected line length and the swept distance yields the swept volume. When the GMB is integrated into a leaf node, its purpose is to pinpoint and index the dead space within the node, effectively considering it removed. The revised dead space with GMB applied is defined in Equation 5.2, which subtracts the total volume of dead grids identified by GMB from the original dead space calculation.

$$\text{DeadSpace}(LeafNode(O)) = \text{Vol}(\text{MBB(O)}) - \text{Vol}(\bigcup_{o_i \in O} o_i) \tag{5.1}$$

$$\text{DeadSpace}(LeafNode(O)_{GMB}) = \text{DeadSpace}(LN(O)) - \sum_{dg_i \in GMB} (\text{Vol}(deadgrid_i)) \tag{5.2}$$

$$\text{Hyper-Volume}(R) = \text{Vol}(R) = \prod_{i=1}^{d} \text{Length}(R_i) \tag{5.3}$$

We evaluate the average dead space percentage for each combination of dataset and tree variant by assessing the dead space as a percentage of the total coverage of the leaf

node. In all of Figures 5.1 5.2 5.3 and 5.4 , the patterned bar heights represent the average original dead space, while the colored gray sections depict the average remaining dead space after applying GMB. Therefore, the non-colored patterned bar segments illustrate the dead space successfully captured and removed by GMB.

It is important to note some observations that are consistent across all datasets. A notable initial observation is that datasets comprised solely of point data originally exhibit 100% dead space shown in Figures 5.1 and 5.3, as points, lacking volume, equate their leaf node dead space directly to their coverage according to Equation 5.1. Consequently, the elimination of dead space is particularly effective in point datasets; only a 64-bit bitmap can achieve a reduction of the original dead space by 40% to 80%, underscoring potential enhancements in query performance, as will be explored in the subsequent section. With the implementation of larger grid bitmap with size of 2^{16} or 2^{32} , it is possible to nearly eliminate all dead space in point datasets. Significant reductions in dead space are also observed in datasets with rectangular data types. For example, in the par03 dataset, where initial dead space is approximately 80%, the application of GMB reduces this to between 50% and 20%. Similarly, in the read02 dataset, with an initial around 85% dead space, reductions of dead space from 40% to 75% are achieved. Even in the more densely populated abs02 dataset, which originally exhibits only 40% to 50% dead space, the implementation of GMB facilitates a reduction of 10% to 25%, achieving dead space levels of 25% to 40%. With less original dead space present in the abs0x' dataset, there is limited potential for dead space reduction, suggesting that the improvements in query performance for abs0x' may be less significant compared to other datasets as we will observe in next section.

When using the adaptive bitmap, as shown in Figures 5.2 and 5.4, we generally observe a dead space reduction comparable to that achieved with a 64-bit or 256-bit grid bitmap, with the exception of the dia0x dataset. The adaptive bitmap construction method analyzes the observed overlap and dead space from MBB to determine the optimal size for the grid bitmap. Due to the characteristics of the dia0x dataset, where all objects are aligned along the diagonal, there is minimal overlap in MBBs. As a result, the adaptive construction method omits some grid bitmaps entirely, as its benefit is negligible in this case. As we will later observe, the dia0x dataset has a very low rate of wasted leaf node accesses, reflecting its unique data distribution characteristics. This confirms that grid bitmaps are unnecessary for dataset dia0x.

Overall, these results demonstrate the effectiveness of GMB in mitigating dead space. The next section will delve into the tangible benefits of this reduction, specifically focusing on the corresponding improvements in query performance.

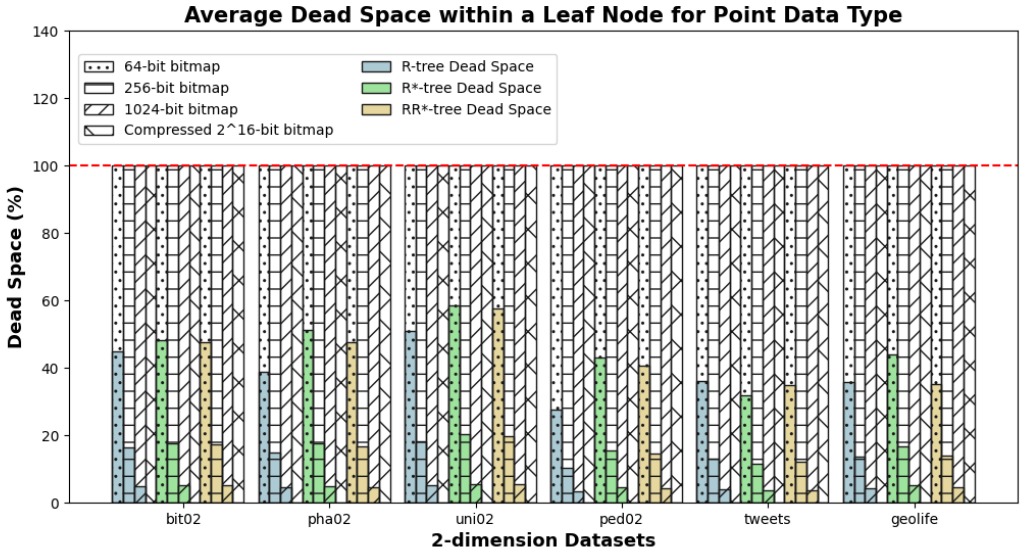


Figure 5.1: Elimination of Dead Space on R-tree variants for 2 dimensional point data

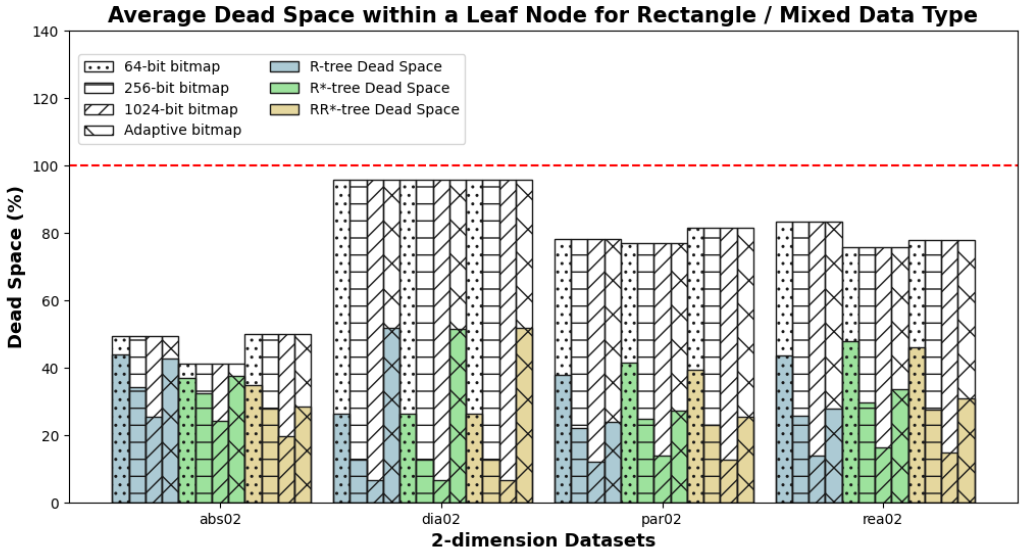


Figure 5.2: Elimination of Dead Space on R-tree variants for 2 dimensional rectangle data

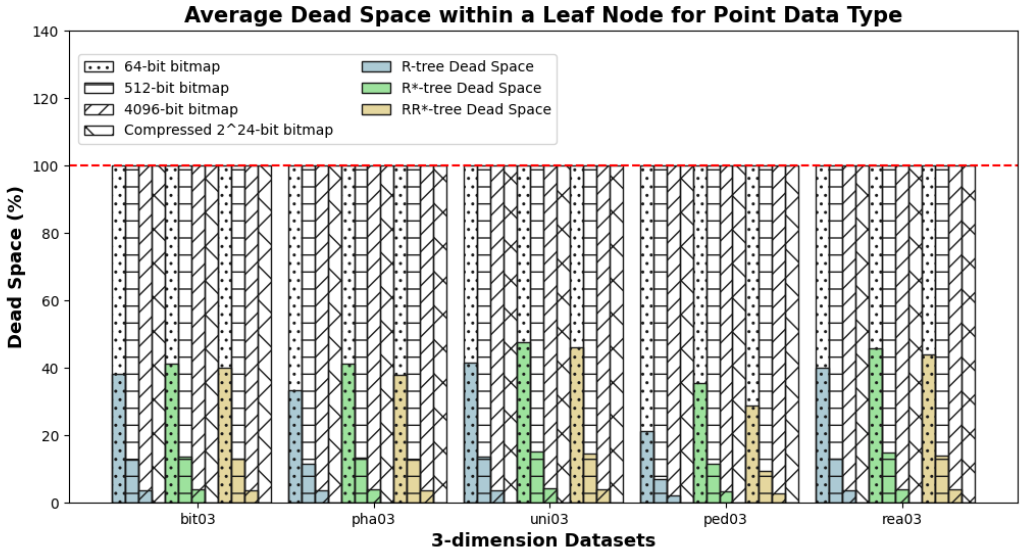


Figure 5.3: Elimination of Dead Space on R-tree variants for 3 dimensional point data

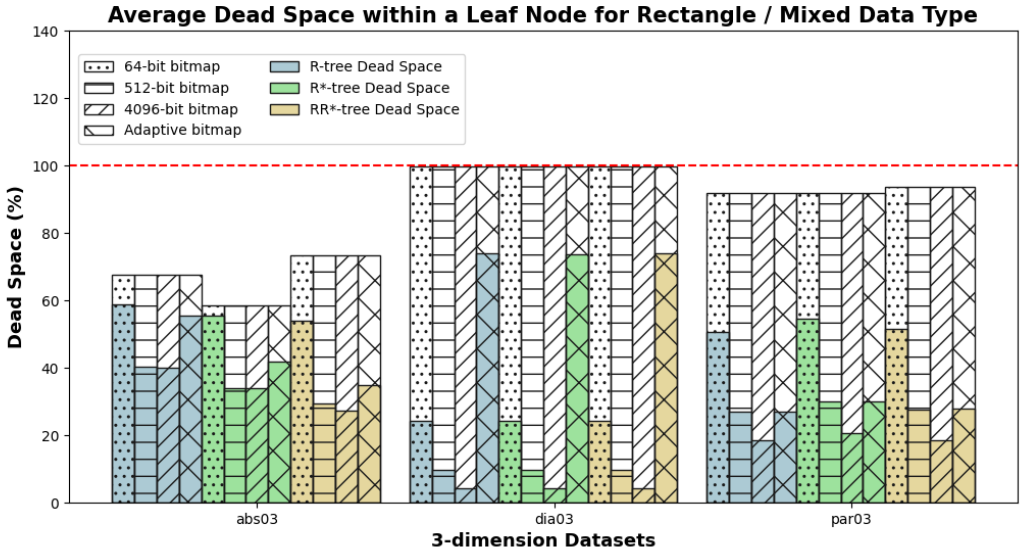


Figure 5.4: Elimination of Dead Space on R-tree variants for 3 dimensional rectangle data

5.3 Query Performance

In this section, we explore the enhancements in query performance facilitated by the Grid Minimum Bounding Box (GMB) across various search queries. Specifically, we have conducted Query 0, Query 2, and Query 3 for each dataset on all tree variants. The performance metrics are visualized through the distribution of useful and wasted leaf node accesses among all accesses incurred, as defined in the Background Section. A leaf node access is deemed useful if it contains objects that are desired by the search query; otherwise, the leaf node access is considered wasted.

The respective figures, 5.5 5.6 5.7 5.8 5.10 5.12 5.14 5.15 5.16 5.17 5.19 and 5.21, illustrate this comparison. In these figures, the plain colored bars represent the useful leaf node accesses, the patterned sections indicate the volume of wasted leaf node accesses prior to GMB implementation, and the patterned dark-colored bars show the remaining wasted accesses post-GMB application. The uncolored patterned segments, therefore, quantify the leaf node accesses that GMB has successfully mitigated.

It is noteworthy that the portion of useful leaf nodes accesses is unaffected by GMB, as GMB reduces only wasteful accesses. The effectiveness of the R*-tree is evident in its substantial share of useful accesses, surpassing other variants. Conversely, the R-tree exhibits a lower proportion of useful accesses, indicating inefficiencies in its clustering mechanism. The R*-tree stands out for its effectiveness and is considered the state-of-the-art for query performance in literature. The R-tree exhibits lower search performance; however, its simplicity and efficiency in handling updates makes it a more suitable choice for datasets that require frequent modifications. Consequently, our experiments span all tree variants to evaluate the extent of improvement that GMB offers to each. Usually, a higher rate of useful leaf node accesses suggests superior clustering by the tree choice, which inherently limits the scope for GMB's enhancements. Furthermore, datasets characterized by higher dead space, such as point datasets, typically show a greater proportion of wasted access, thereby yielding more significant improvements with GMB. This effect is similarly pronounced across different query types; higher selectivity in a query correlates with increased ratio of wasted accesses and a correspondingly larger impact of GMB, most notably observed in the most selective query type 0. Another observation is that GMB with larger grid bitmap also brings more improvements on query performance consistently across all datasets and tree variants.

In summary, GMB consistently enhances performance, with its impact most pronounced in scenarios marked by a high rate of wasted access, which indicates substantial opportunities for improvement. Subsequent subsections will provide a comparative analysis of

dataset and tree variants across different dimensions and data types. We will discuss performance impacts of GMB across tree-variants and datasets mostly based on the 64-bit grid bitmap to avoid repetitive discussion.

5.3.1 Query Performance for 2 Dimensional Point Data

As shown in Figures 5.5, 5.6, and 5.7, this subsection examines six datasets containing two-dimensional point data, namely bit02, pha02, uni02, and ped02 from the multi-dimensional benchmark, along with Tweets and Geolife from the UCR Star datasets. Notably, the Tweets and Geolife datasets are substantially larger than the other four datasets as shown in Table 5.2. Particularly in Query 0, we observe more pronounced improvements with GMB. The implementation of a 64-bit bitmap consistently reduces wasted accesses by approximately 40% to 60% across all tree variants for each query type. The actual performance enhancement is notably significant with the R-tree during more selective queries, where the proportion of wasted accesses is substantially higher. Utilizing larger sized grid bitmaps can nearly eliminate all wasted accesses. Nevertheless, the overall enhancement in performance for search Query type 3 is modest for smaller datasets, as shown in Figure 5.7, especially with the state-of-the-art R*-tree, as the majority of accesses are useful and inevitable. Conversely, with the larger and more complex datasets such as Tweets and Geolife, an improvement ranging from 8% to 15% is still observable on Query Type 3.

5.3.2 Query Performance for 2 Dimensional Rectangle Data

As depicted in Figures 5.8, 5.10, 5.12, this subsection examines the remaining four datasets with rectangle or mixed data types from the multi-dimensional benchmark, specifically abs02, dia02, par02, and rea02. Notable improvements are observed in par02 and rea02, where the application of a 64-bit Grid Minimum Bounding Box (GMB) facilitates a reduction of 20% to 30% in wasted accesses across tree variants. This outcome is anticipated due to the substantial potential for reducing wasted access, likely stemming from the large proportions of dead space identified in the Dead Space Elimination chapter.

Conversely, the dataset dia02 shows minimal improvement, ranging from 5% to zero. This dataset exhibits a unique spatial distribution as discussed above, with data objects typically positioned along the diagonal of the space, and queries similarly aligned. This diagonal arrangement results in a high average dead space per leaf node, as previously discussed; however, the alignment of queries with this spatial distribution yields a high rate of useful accesses, minimizing the potential for further improvement by GMB. This

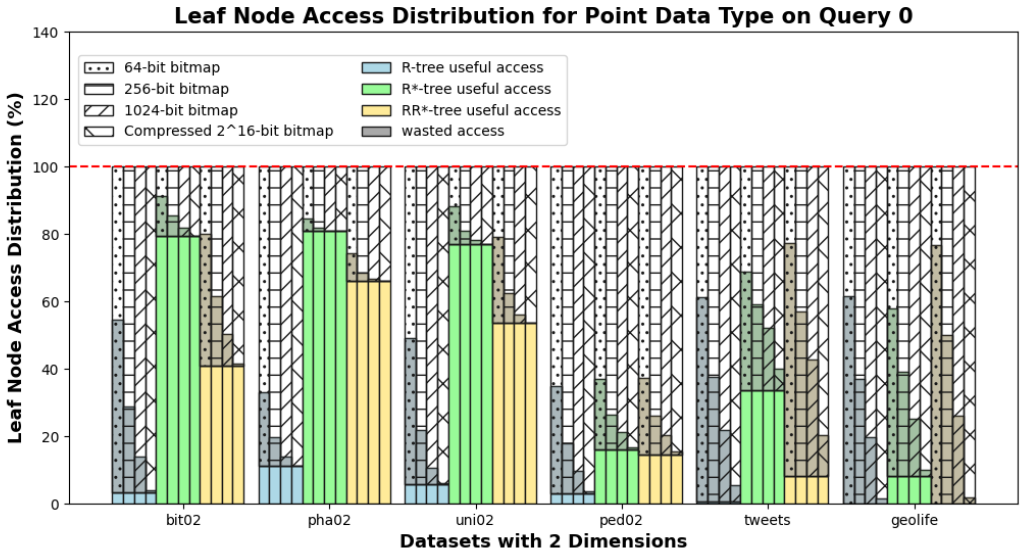


Figure 5.5: Leaf Node Accesses Distribution for 2-dim Point Data on Query 0

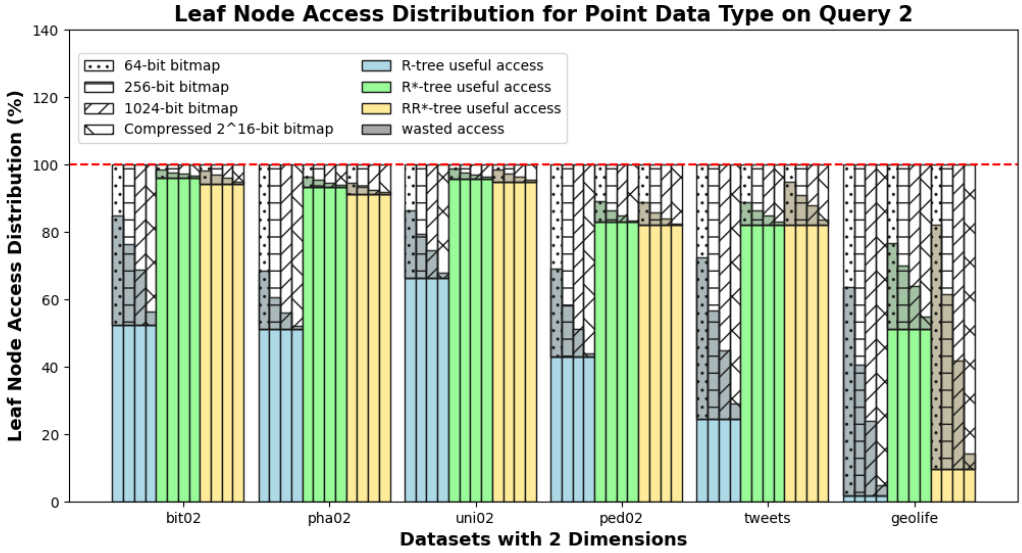


Figure 5.6: Leaf Node Accesses Distribution for 2-dim Point Data on Query 2

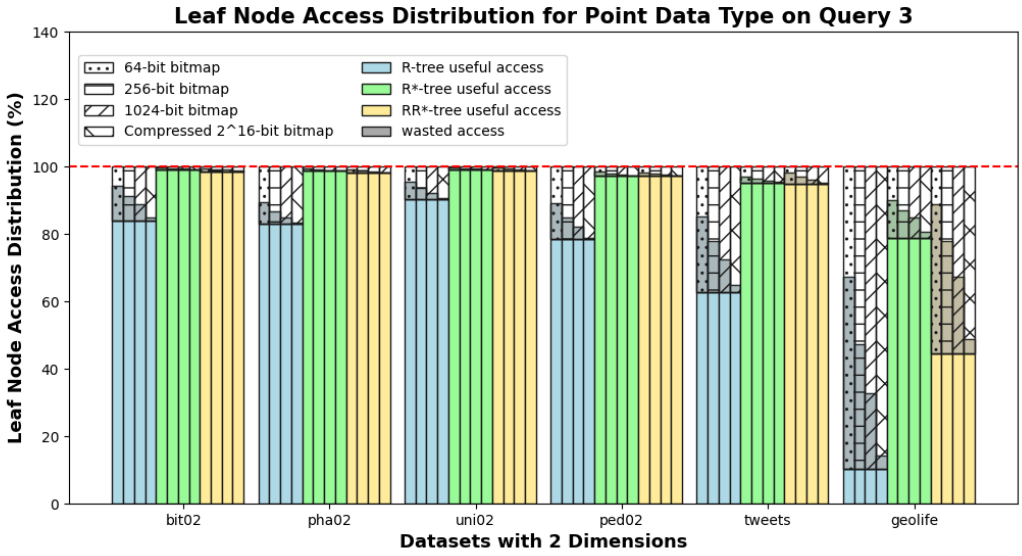


Figure 5.7: Leaf Node Accesses Distribution for 2-dim Point Data on Query 3

scenario illustrates a case where GMB does not significantly enhance performance when the query set inherently aligns with the data distribution. Similarly, due to lack of dead space within abs02 shown in previous section, the potential for improvement is also quite limited. Using the adaptive grid bitmap, small-sized grid bitmaps are selected, reflecting the minimal opportunity for further optimization.

Another noteworthy observation with rectangular data is the relatively small variance in useful accesses among different tree variants, which is significantly less than the variance observed with point datasets. Particularly for the least selective Query Type 3, nearly all accesses are useful across all tree variants, indicating that no leaf node accesses are wasted. This suggests that the distinctions among tree variants diminish with less selective queries, highlighting the convergence of different clustering techniques applied across tree variants. As improvements at query performance is less observable globally, in Figures 5.9, 5.11, and 5.13, we specifically focus on improvements solely in reducing wasted accesses. We can observe even when the potential for improvement is limited, GMB consistently enhances search performance at readuing wasted accesses.

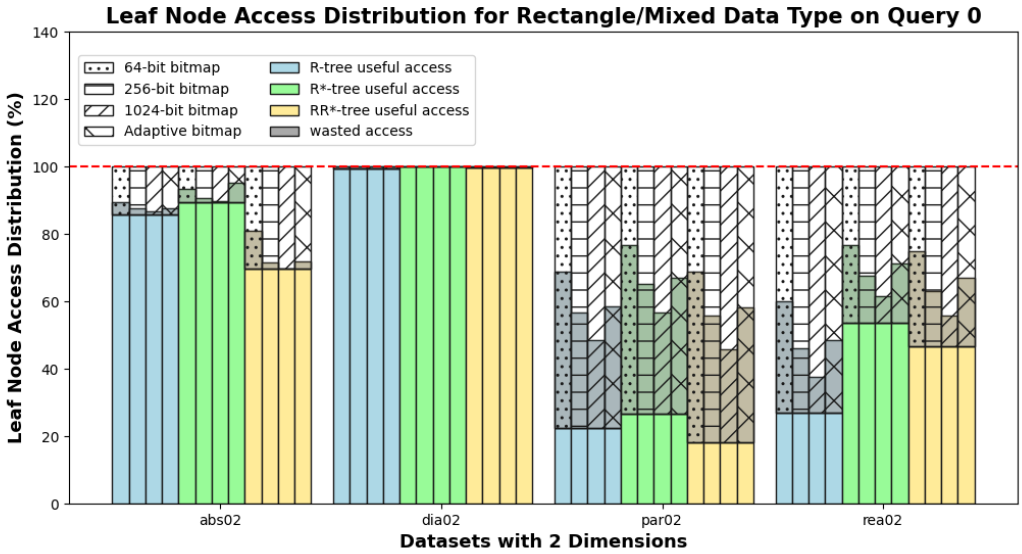


Figure 5.8: Leaf Node Accesses Distribution for 2-dim Rectangle Data on Query 0

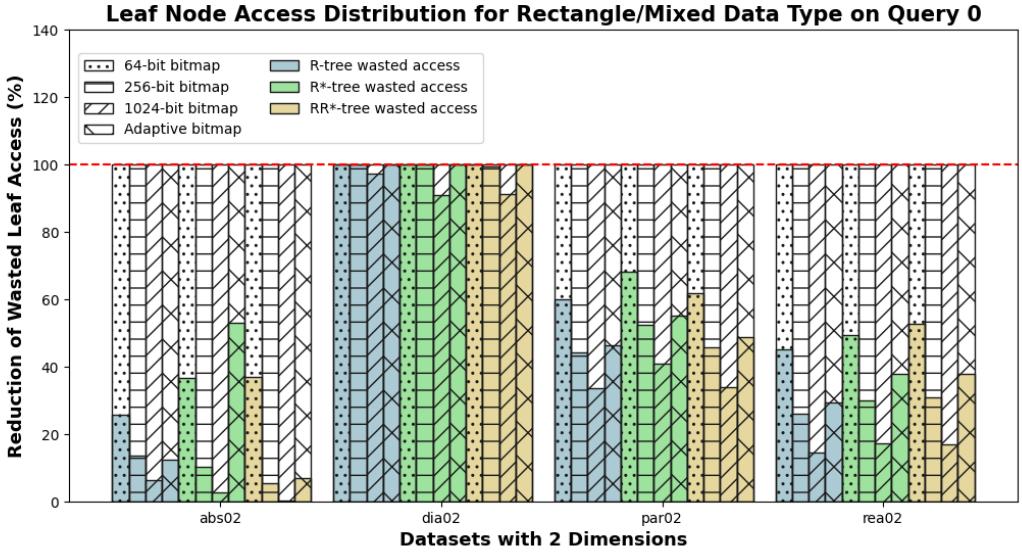


Figure 5.9: Leaf Node Wasted Accesses Reduction for 2-dim Rectangle Data on Query 0

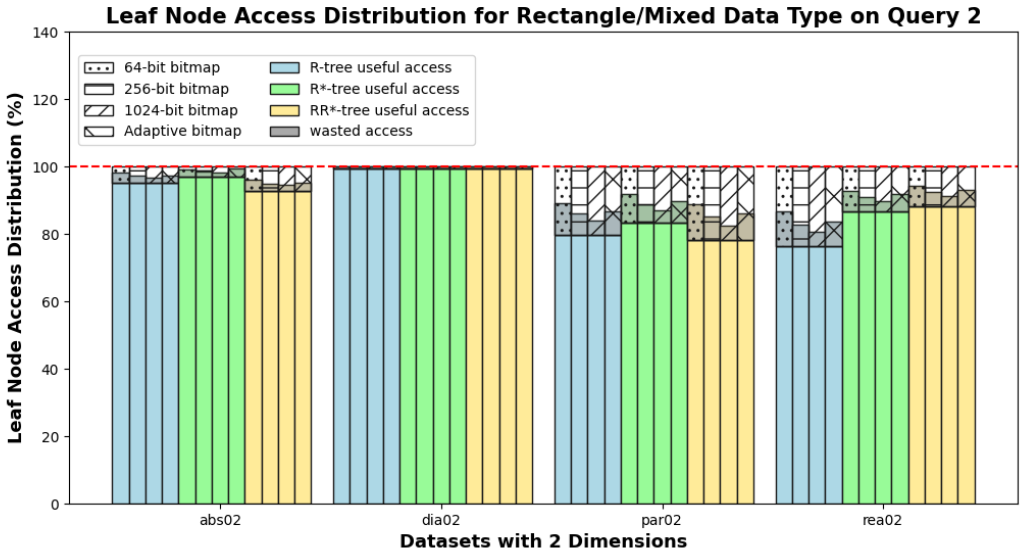


Figure 5.10: Leaf Node Accesses Distribution for 2-dim Rectangle Data on Query 2

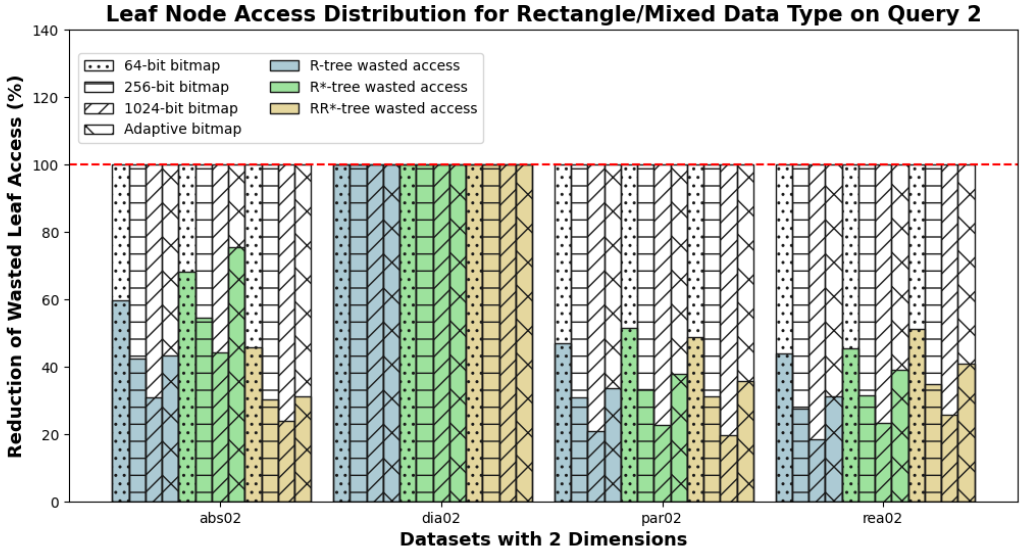


Figure 5.11: Leaf Node Wasted Accesses Reduction for 2-dim Rectangle Data on Query 2

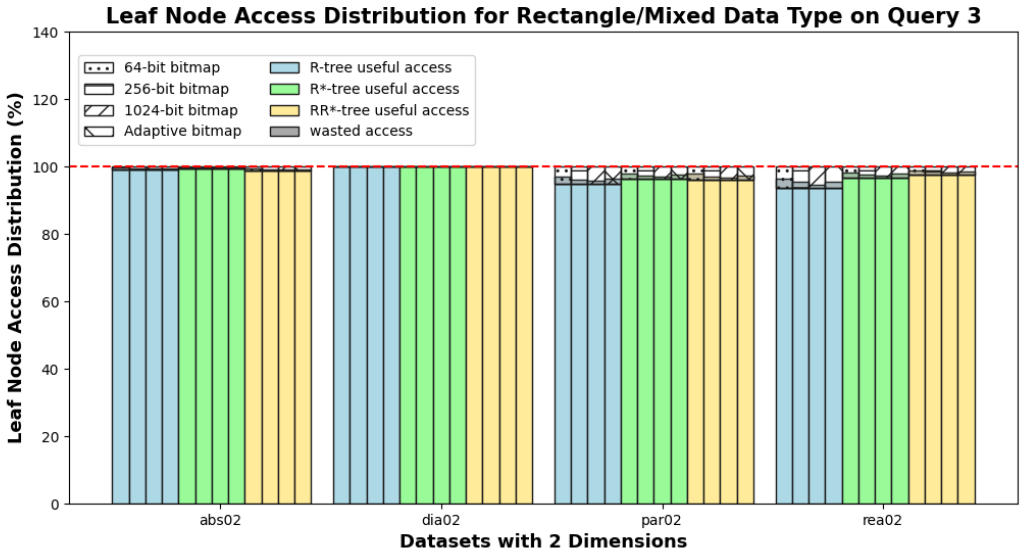


Figure 5.12: Leaf Node Accesses Distribution for 2-dim Rectangle Data on Query 3

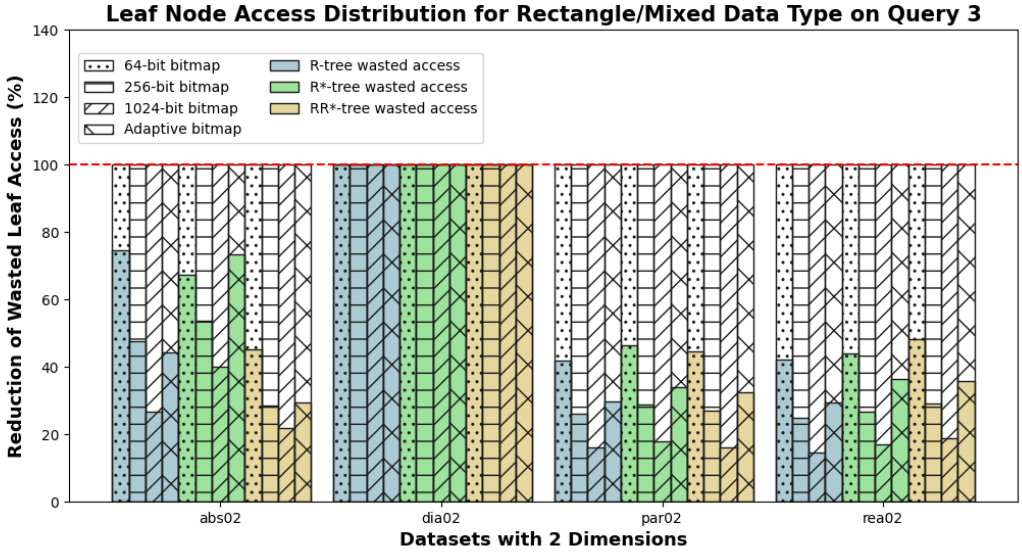


Figure 5.13: Leaf Node Wasted Accesses Reduction for 2-dim Rectangle Data on Query 3

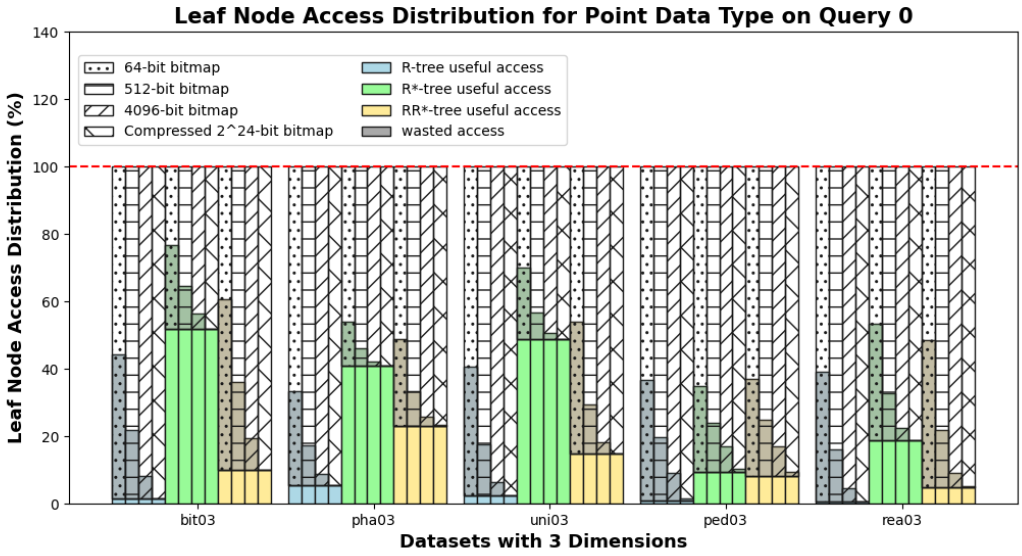


Figure 5.14: Leaf Node Accesses Distribution for 3-dim Point Data on Query 0

5.3.3 Query Performance for 3 Dimensional Point Data

From Figures 5.14, 5.15, and 5.16, observations made here closely align with those discussed in the previous section on two-dimensional point data, albeit with consistently more improvements for datasets within two dimensions. The introduction of an additional dimension tends to reduce the scattering of data objects. This hypothesis is supported by the observed increase in dead space percentage detailed in the preceding Dead Space Elimination chapter. Consequently, the application of the GMB proves to be more effective overall. With a 64-bit grid bitmap, for same data distribution, we can observe approximately 5% to 10% more performance enhancement compared with two-dimensional data.

5.3.4 Query Performance for 3 Dimensional Rectangle Data

Similar to the analysis for three-dimensional point datasets, we observe an overall increased effectiveness of the Grid Minimum Bounding Box (GMB) in Figures 5.17, 5.19, and 5.21. The contrast between par02 and par03 is particularly striking for Query Type 3. Specifically, par02 exhibits only 5% wasted accesses without GMB, whereas par03 demonstrates 20% wasted accesses, thereby offering a greater potential for improvement through the

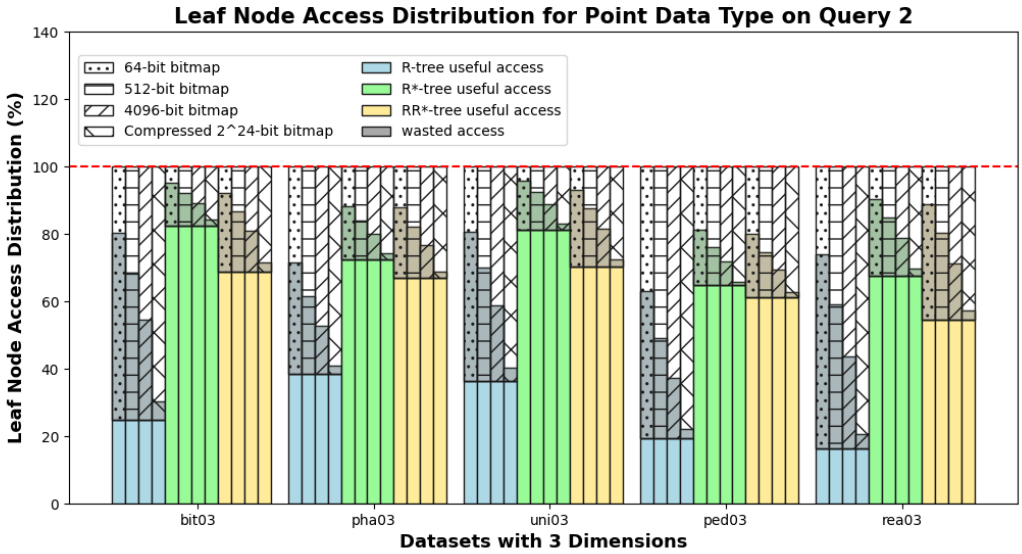


Figure 5.15: Leaf Node Accesses Distribution for 3-dim Point Data on Query 2

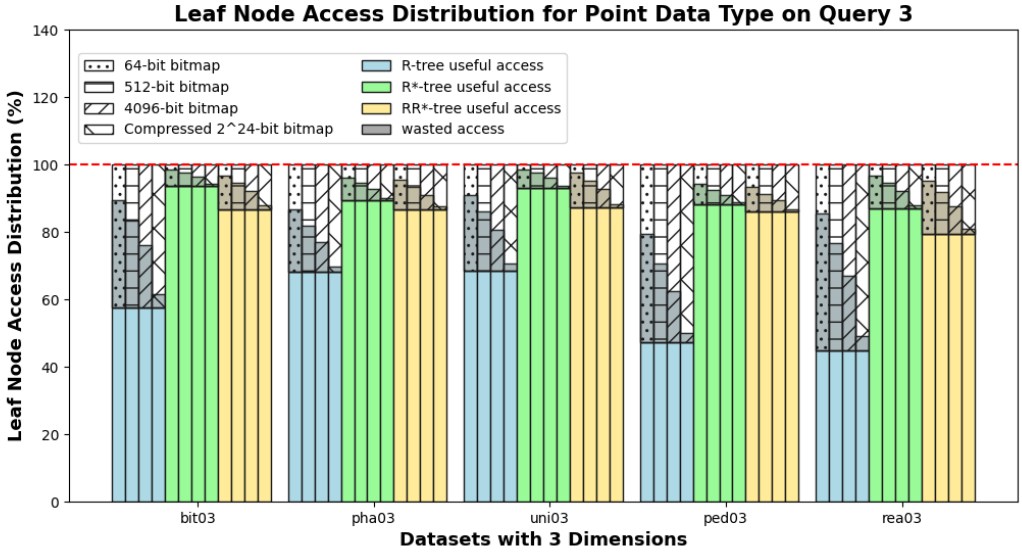


Figure 5.16: Leaf Node Accesses Distribution for 3-dim Point Data on Query 3

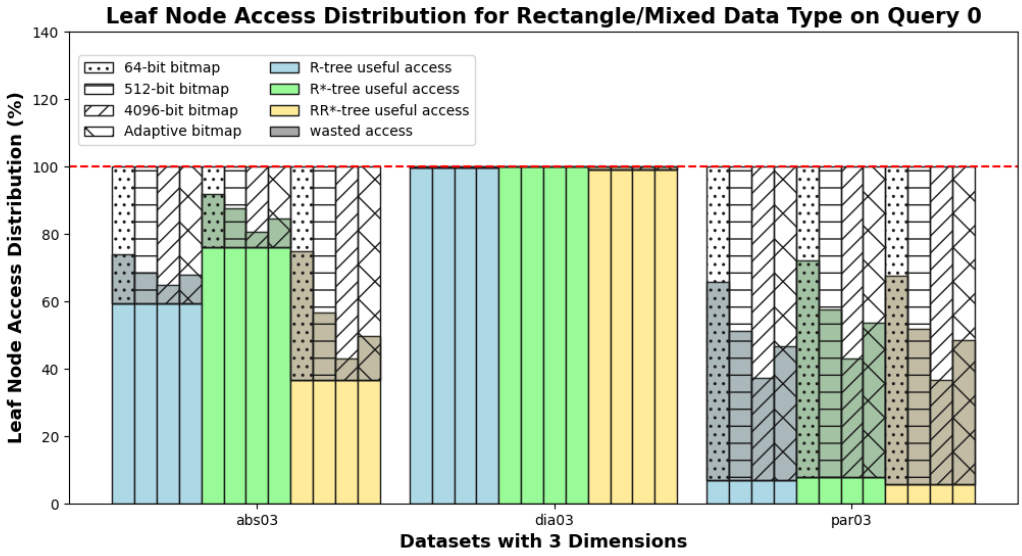


Figure 5.17: Leaf Node Accesses Distribution for 3-dim Rectangle Data on Query 0

application of GMB. par03 is observing 25%, 15%, and 10% performance improvement for Q0, Q2, and Q3. As shown in Figures 5.18, 5.20, and 5.22, we can observe consistently increasing reduction on wasted leaf node accesses with increasing grid bitmap size.

5.4 Overheads

The Grid Minimum Bounding Box (GMB) enhances query performance by providing additional metadata on dead space, however, this improvement is at the expense of extra information kept in the main memory. Thus, the performance gains afforded by GMB represent a trade-off against the cost of extra storage. Furthermore, GMB implementation necessitates additional CPU computation time to construct and update the grid bitmap for each leaf node. This section discusses the costs associated with these benefits. We examine the storage costs involved in maintaining the GMB and the computational demands required for its construction across two separate subsections.

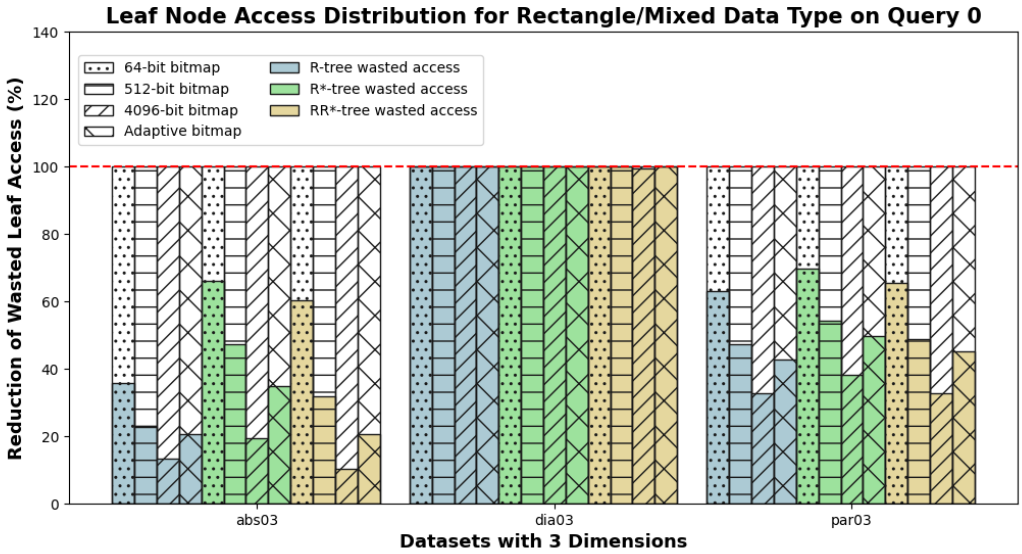


Figure 5.18: Leaf Node Wasted Accesses Reduction for 3-dim Rectangle Data on Query 0

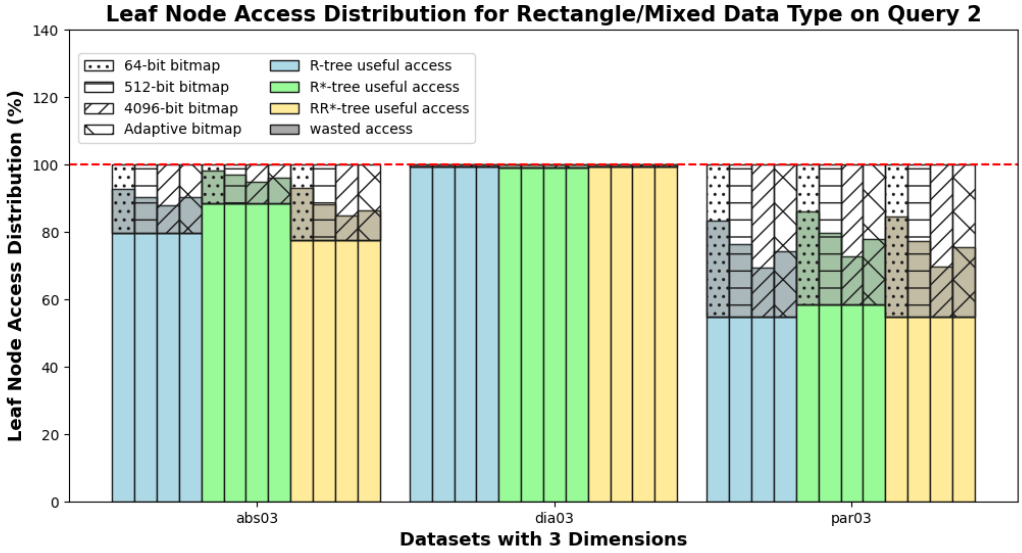


Figure 5.19: Leaf Node Accesses Distribution for 3-dim Rectangle Data on Query 2

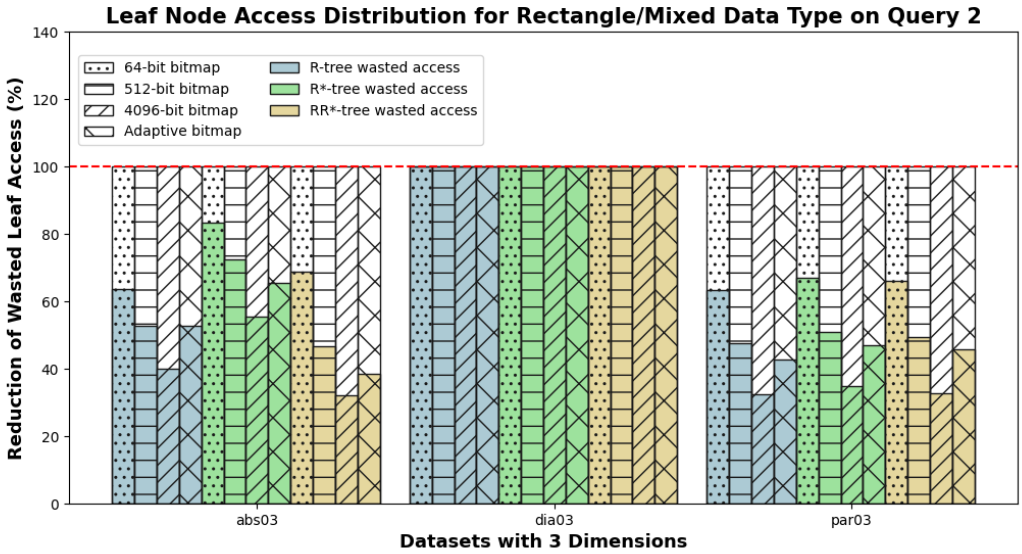


Figure 5.20: Leaf Node Wasted Accesses Reduction for 3-dim Rectangle Data on Query 2

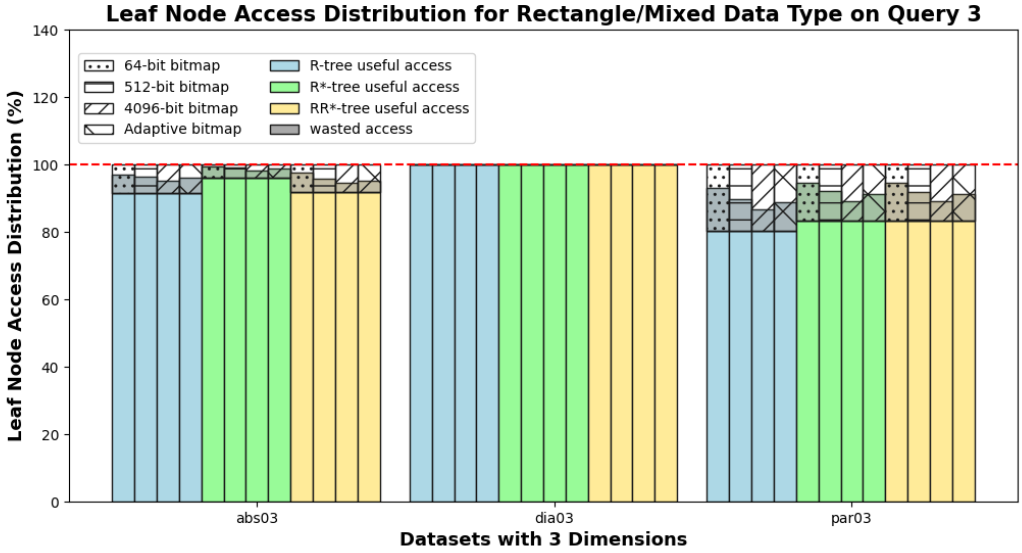


Figure 5.21: Leaf Node Accesses Distribution for 3-dim Rectangle Data on Query 3

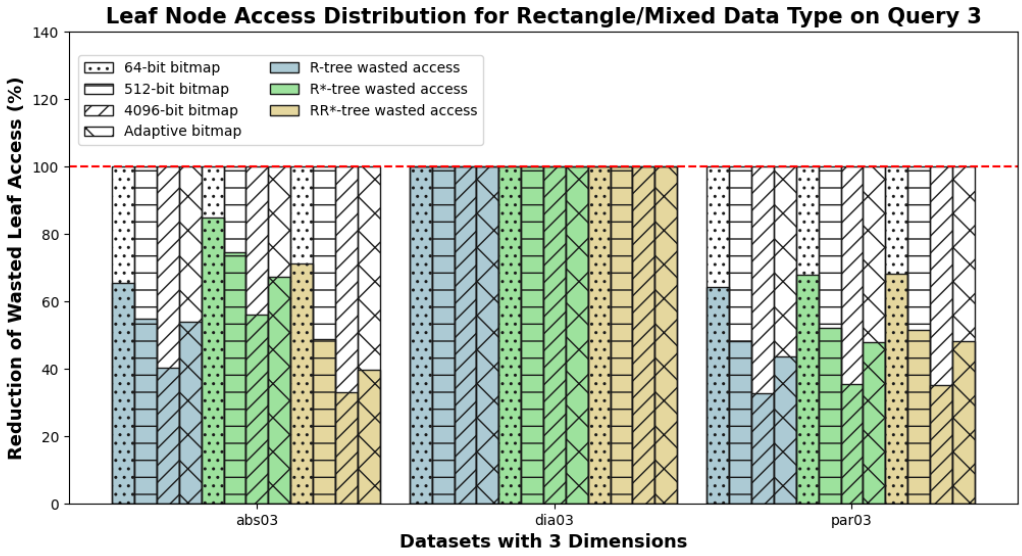


Figure 5.22: Leaf Node Wasted Accesses Reduction for 3-dim Rectangle Data on Query 3

5.4.1 Storage Overheads

In this section, we quantify the storage overhead by comparing the proportions of bytes allocated to branch nodes, leaf nodes, and the Grid Minimum Bounding Box (GMB) across various datasets and tree variants. For fixed-size bitmaps, the storage cost can be approximated as the product of the number of leaf nodes and the bitmap size. In Figures 5.23 5.25 5.24 5.26, the orange segments of the bars indicate the proportion of storage occupied by branch nodes, approximately 5% of the total storage. The predominant portion of the bar denotes leaf node storage. The dark segments on the top represent the percentage of storage attributed to the grid bitmap. GMB occupies about 0.5% for a 64-bit GMB, 2% for a 256-bit GMB, and 9% for a 1024-bit GMB, as depicted in Figure 5.23. The size of the GMB scales exponentially with the number of split for constructing the grid bitmap. With grid location representation, the grid bitmap's storage demand can be reduced to less than 0.5% of the total size. Consequently, the grid location based grid bitmap emerges as a highly competitive technique, optimizing storage efficiency and enhancing query performance. Nevertheless, this representation is not universally applicable, particularly for rectangle datasets, as discussed in the Design chapter. Consequently, the grid location method is not tested on rectangle datasets. Instead, adaptive bitmaps are used for rectangle datasets, where we observe reduced storage costs with competitive query performance preserved, as

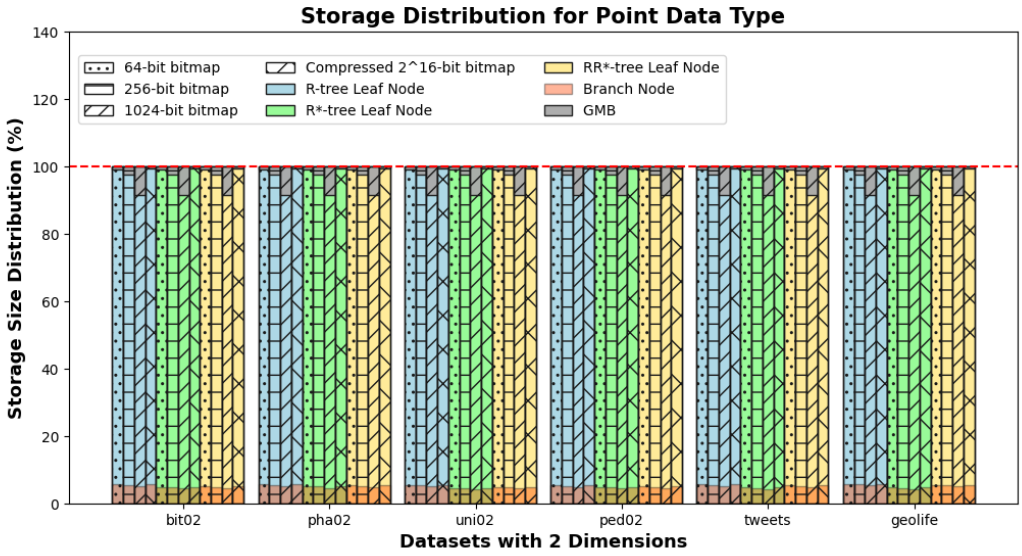


Figure 5.23: Storage Distribution for R-tree variants on 2-dim Point Data

shown in previous sections. The adaptive bitmap approach balances storage cost and performance by allocating more resources to create larger grid bitmaps for leaf nodes with higher potential for improvement, and the grid bitmap can be completely omitted when there is no potential for improvement.

5.4.2 Construction Overheads

In this section, we evaluate the latency involved in constructing the Grid Minimum Bounding Box (GMB) as shown in Figures 5.27, 5.28, 5.29, and 5.30. For this experiment, we assume that all leaf nodes are resident in main memory, thereby eliminating the measurement of disk access times. Instead, our focus is solely on the CPU computation costs associated with tree loading and GMB construction. The construction time for GMB with point data is negligible, consuming less than 1.5% of CPU computation time as shown in Figures 5.27, and 5.29. This efficiency is due to the fact that a point can only intersect with one grid in the GMB, thus sets an upper bound equivalent to the maximum fan-out of the leaf node. In contrast, rectangle data may intersect with multiple grids within the GMB which is only upper-bounded by size of grid bitmap, necessitating greater computational effort to mark the grids as occupied within the bitmap. This difference is clearly

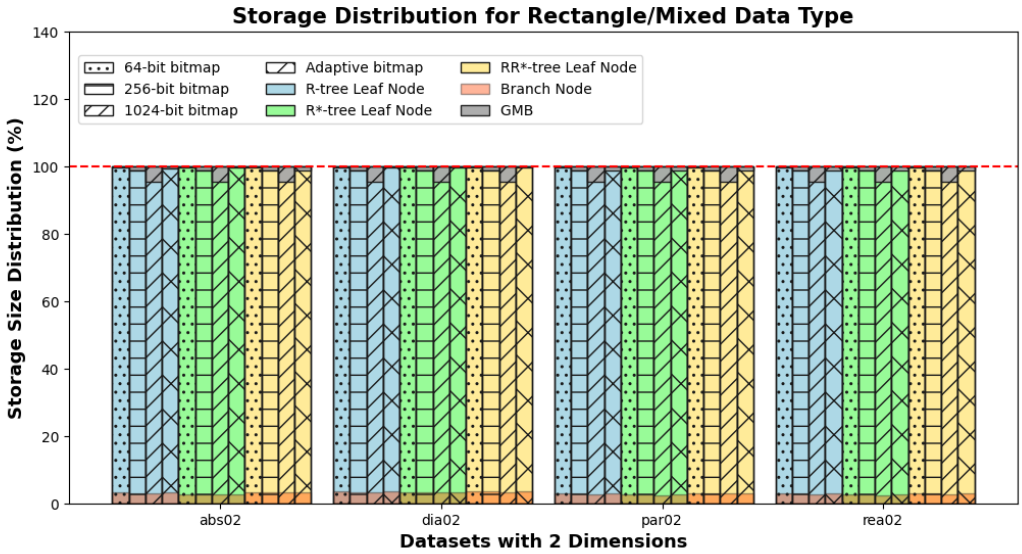


Figure 5.24: Storage Distribution for R-tree variants on 2-dim Rectangle Data

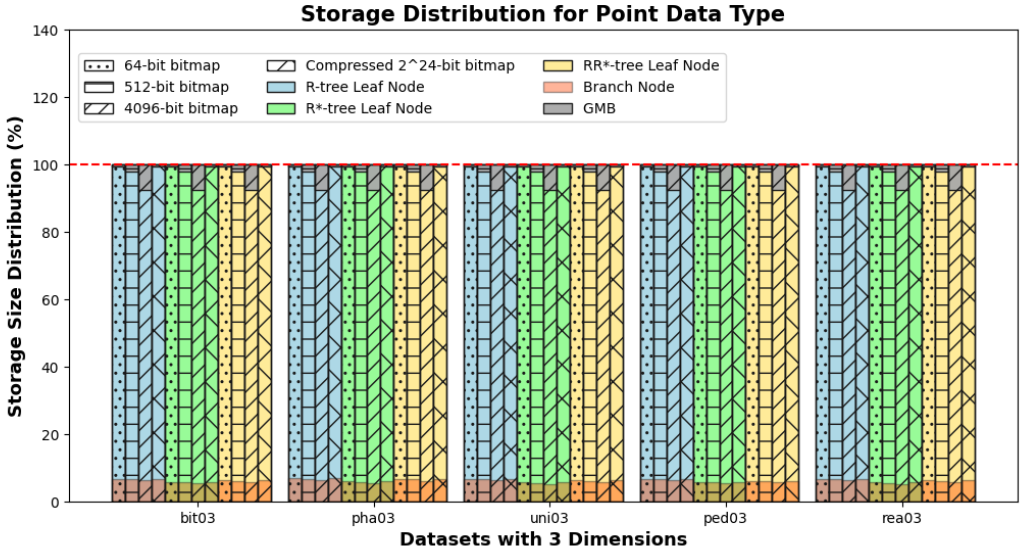


Figure 5.25: Storage Distribution for R-tree variants on 3-dim Point Data

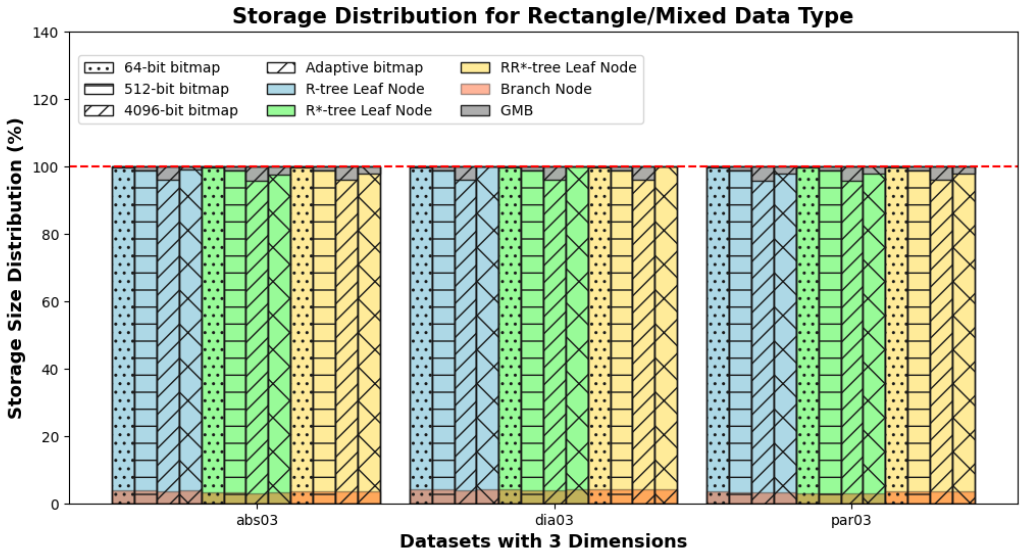


Figure 5.26: Storage Distribution for R-tree variants on 3-dim Rectangle Data

illustrated in Figures 5.28, and 5.30, where less than 15% of the CPU computation is dedicated to constructing the 64-bit grid bitmap. In the case of the R*-tree, the proportion of time spent on GMB construction is still low, as the R*-tree incurs higher latency for tree loading due to its reinsertion technique, compared to other R-tree variants.

5.5 Summary

We conducted extensive experiments across multiple datasets and various R-tree variants to evaluate the advantages and associated costs of GMB with different grid bitmap sizes. In the previous sections, we highlighted the impact of GMB on each dataset and R-tree variant at a granular level. Here, we present the average performance impact and storage costs across all datasets, as shown in Table 5.4. It should be noted that the performance of the Grid Location representation is summarized for point data only, while the Adaptive Bitmap results are summarized for rectangle data only. With the 64-bit grid bitmap, we observed performance improvements ranging from 38% to 9% across Q0 to Q3 for two-dimensional datasets, with a storage cost of 0.7% and a computation cost of 3.3%. When using the 1024-bit bitmap, performance improvements doubled, though this came

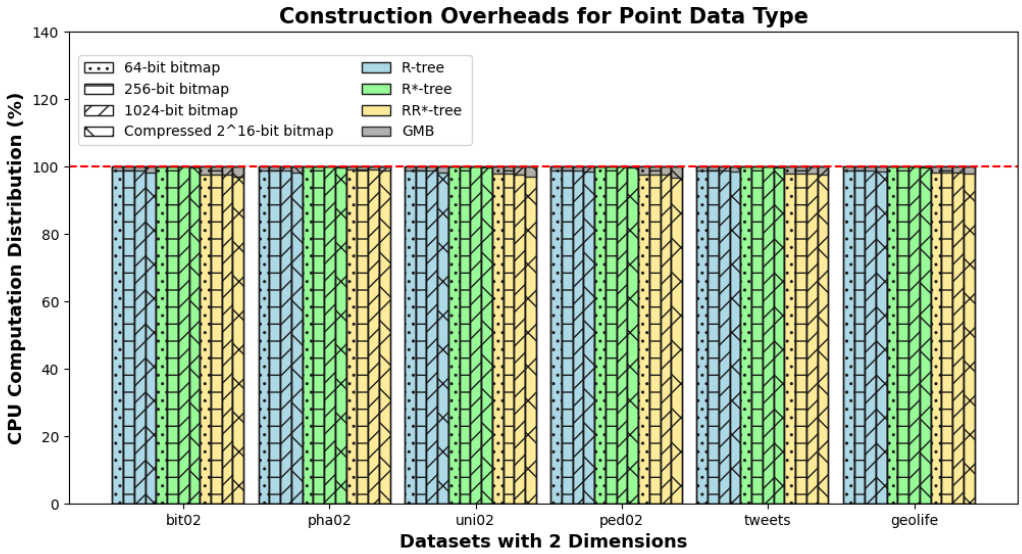


Figure 5.27: CPU time distribution for R-tree variants on 2-dim Point Data

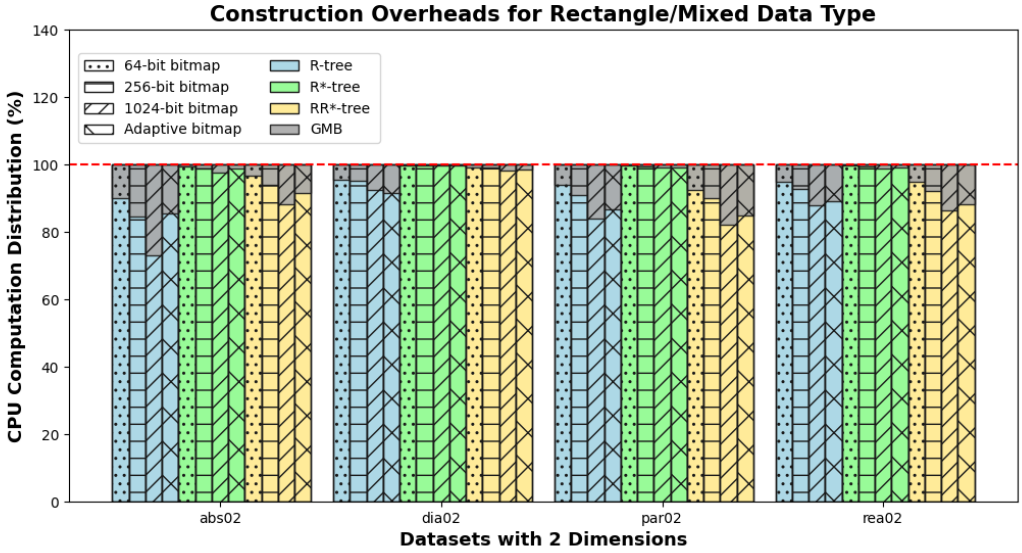


Figure 5.28: CPU time distribution for R-tree variants on 2-dim Rectangle Data

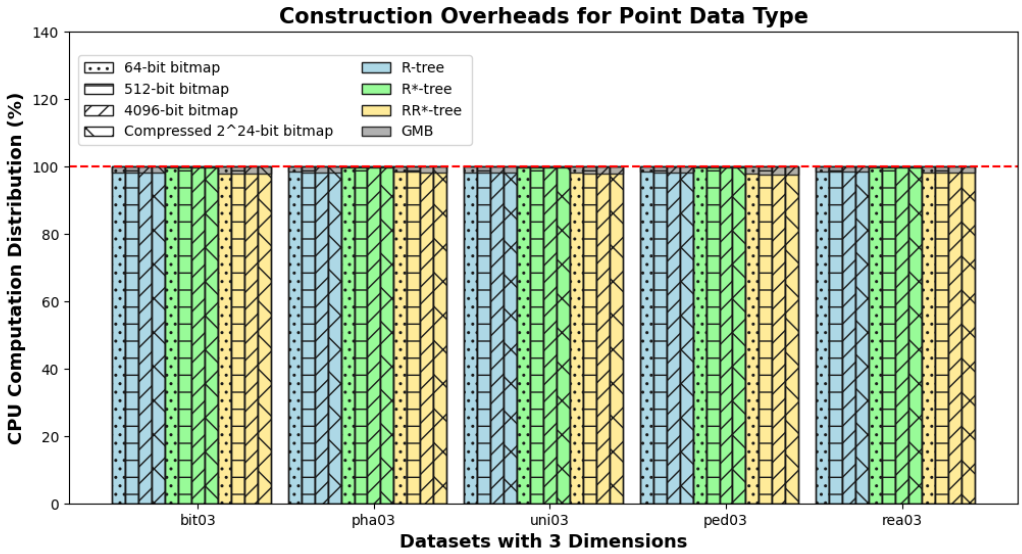


Figure 5.29: CPU time distribution for R-tree variants on 3-dim Point Data

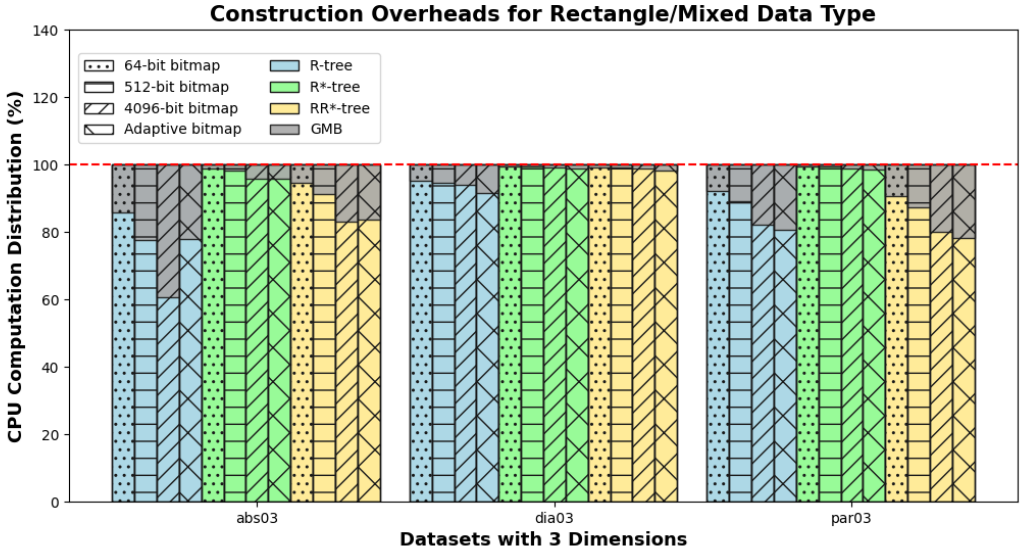


Figure 5.30: CPU time distribution for R-tree variants on 3-dim Rectangle Data

	2 Dimensions					3 Dimensions				
	64-bit	**256-bit**	**1024-bit**	**2^16-bit (pt)**	**Adaptive (rect)**	**64-bit**	**512-bit**	**4096-bit**	**2^24-bit (pt)**	**Adaptive (rect)**
Dead Space (%)	54.88	77.2	88.37	99.92	48.83	57.46	80.43	87.48	99.99	38.15
Query 0 (%)	38.74	54.7	63.81	94.71	26.4	45.78	60.96	70.08	97.63	28.47
Quert 2 (%)	18.13	26.19	31.93	57.6	8.116	19.33	28.18	37.04	69.15	11.85
Query 3 (%)	8.632	12.73	15.73	30.54	2.148	9.75	14.41	19.41	39.79	5.011
Storage (%)	0.67	1.981	6.871	0.6241	0.7895	0.6078	1.8	6.275	0.6802	1.013
Computation (%)	3.336	4.418	7.085	1.661	11.87	4.406	5.899	9.045	1.758	16.71

Table 5.4: Average Performance Impact Summary on R-tree

at an exponential increase in storage costs. The Grid Location representation exhibited the strongest performance over point datasets, with more than a 90% improvement on Q0 and over 30% on Q3, while maintaining minimal storage costs, comparable to the 64-bit bitmap.

Chapter 6

Related Work

Given the extensive history of spatial data indexing, this chapter reviews some of the most influential indexing methods and closely related work. Spatial indexes can be roughly classified into two categories based on how objects are grouped: data-partitioned and space-partitioned. Data-partitioned indexes are object-oriented where objects are clustered into subgroups hierarchically based on their locations, with the R-tree family belonging to this category. Space-partitioned indexes are space-oriented, where multi-dimensional space is split into disjoint partitioned sub-regions and objects are assigned to corresponding sub-regions based on its location. In addition to spatial indexes, various bounding box shapes proposed in the literature are also presented in this chapter. Finally, this chapter concludes with other augmentation techniques on tree or minimum bounding box. Therefore, the following discussion is organized into four subsections: space-partitioned indexes, data-partitioned indexes, bounding box shapes and augmentation techniques.

6.1 Space-partitioned Indexes

Space-partitioned indexes have long served as foundational structures for spatial data management by recursively dividing space into non-overlapping regions. This division is typically based on a predetermined rule or geometric structure, which enables efficient recursive organization. The primary advantage of this approach is the non-overlapping nature of the partitions, which simplifies the traversal path during search operations, ensuring a direct route to potentially relevant data. However, this approach faces limitations when dealing with volumetric data, such as rectangles. Objects that overlap multiple sub-regions must be replicated to maintain query accuracy, which can lead to inefficiencies. Additionally,

empty sub-regions can result in leaf nodes filled with dead space or even empty leaf nodes, both of which negatively impact query performance.

The Quad-tree [17] is one of the early in-memory spatial data management solutions, adapted from binary search trees for two-dimensional data. It recursively divides the two-dimensional space into four (2^2) blocks at each node until each node contains only a single object. For three-dimensional space, the corresponding structure is the Oct-tree [36], which divides space into eight (2^3) homogeneous blocks. These methods are particularly effective for managing sparse datasets, as they allow the tree to dynamically expand in regions with higher data concentrations. However, both Quad-trees and Oct-trees can suffer from deep nesting and unbalanced partitions when data is unevenly distributed across the space, potentially leading to inefficient query performance.

The KD-tree [12] improves upon the Quad-tree by considering data distribution. Instead of partitioning space evenly into volume-balanced sub-regions, the KD-tree uses hyperplanes aligned with the median data to create data-balanced sub-regions at each node. The partitioning dimension alternates at each tree level to maintain balance. While the KD-tree remains an in-memory solution, the KDB-tree [43] extends this approach to disk-based indexing by combining the KD-tree's partitioning strategy with the balanced nature of B-trees [8]. However, both KD-tree and KDB-tree become less efficient as dimensionality increases, as maintaining balance grows increasingly complex. Furthermore, the KDB-tree assumes evenly distributed data across dimensions, which can lead to unbalanced trees and inefficient search times when data is skewed or clustered along certain dimension.

For managing multi-dimensional data, the UB-tree [42] offers a different approach by combining the structured approach of B-trees with the geometric efficiency of space-filling curves. The UB-tree maps multidimensional data into a one-dimensional key space using space-filling curves such as Z-order (Morton Order) [38], preserving the locality of data objects. This mapping allows the UB-tree to integrate seamlessly with existing database systems that support B-tree indexes, as well as maintaining the balance properties as B-trees to ensure logarithmic time complexity for operations such as insertions, deletions, and searches. The study of space-filling curves can be treated as another extensive field, with popular examples including the Hilbert curve [37] and the C curve [27].

Another early adaptation for multi-dimensional data management is the Grid File [40], which partitions space into a uniform grid. Each cell in the grid functions as an independent bucket that directly maps to disk blocks, enabling efficient direct access and enhancing point query performance. However, in high-dimensional spaces, the number of grid cells increases exponentially with each added dimension, leading to sparse distributions and inefficient storage use, a problem commonly referred to as the "curse of dimensionality".

6.2 Data-partitioned Indexes

In contrast to space-partitioned indexes, which partition space into sub-regions, data-partitioned indexes organize objects into subgroups by clustering. The objective of clustering is to group spatially closed objects within the same node, thereby balancing node sizes and enhancing query performance. However, achieving the global optimal in clustering is an NP-hard problem [20], makes it infeasible for large-scale spatial data management. To address this, various efficient local-optimal clustering metrics have been developed that prioritize computational efficiency over global clustering optimality. Clustering metrics for spatial data management usually includes metrics for node expansion and overfull node split. Those metrics often focus on reducing coverage, minimizing dead space, or limiting overlap. A significant advantage of data-partitioned indexes lies in their ability to dynamically adjust to the distribution of spatial data, resulting in tighter, data-driven groupings. However, since overlap is not strictly prohibited within these indexes, queries may require traversal of multiple paths, leading to increased disk access and potentially impacting performance.

The R-tree [23] is a foundational structure in spatial data management, organizing spatial objects within hierarchical, nested Minimum Bounding Boxes (MBBs). This organization supports efficient point and range queries by enabling straightforward intersection tests. Due to its simplicity and effectiveness, the R-tree has become the most widely used data structure for spatial data management in existing database systems [4, 3]. For metrics of expansion and splitting, R-tree takes minimization of coverage as priority for clustering.

The R*-tree [9] is widely regarded as the state-of-the-art within the R-tree family. It introduces several refinements that enhance spatial indexing efficiency, particularly in managing overfull nodes. A key innovation of the R*-tree is its reinsertion strategy, which significantly improves the tree's overall performance. Instead of immediately splitting an overflowing node, the R*-tree algorithm reinserts a subset of the entries back into the tree, thereby optimizing node utilization. However, this improved query performance comes at the cost of increased workload during the insertion phase, leading to longer construction times. In addition to reinsertion, the R*-tree employs more sophisticated expansion metrics that consider the area, perimeter, and overlap of bounding boxes. These enhancements provide the R*-tree with significant advantages over other R-tree variants.

In addition to the high construction latency, the reinsertion strategy in the R*-tree also limits concurrency. To address these challenges, the authors of the Revised R*-tree (RR*-tree) [11] aimed to achieve similar query performance without relying on reinsertion. By applying carefully designed metrics for node expansion and node splitting, the RR*-tree

offers competitive query performance compared to the R*-tree, while significantly reducing construction latency and allowing concurrency.

The HR-tree [29] leverages space-filling curves, similar to the UB-tree [42], using Hilbert Curves [37] to linearize multi-dimensional data into a single dimension while preserving the locality of objects. The HR-tree also employs Minimum Bounding Boxes (MBBs) to store closely related objects. It is specifically designed to accommodate dynamic datasets efficiently, adapting to changes in data distribution with minimal performance degradation.

The R+-tree [44] focuses on eliminating overlapping nodes at the same tree level, similar to the KDB-tree [43]. However, unlike the KDB-tree, the R+-tree uses Minimum Bounding Boxes (MBBs) for storage, as other R-trees. This approach ensures that nodes are disjoint, thereby reducing the query ambiguity caused by overlaps. However, this non-overlapping requirement can complicate node splits, necessitating more complex downward split propagation to maintain non-overlapping conditions. Additionally, similar to space-partitioned indexes, rectangular data need to be replicated across nodes to ensure query accuracy.

Similar to R+-tree, many recent research work have also focused on eliminating overlap between Minimum Bounding Boxes (MBBs), such as Waffle [39] and Nir+-tree [6]. Waffle achieves node disjointness by combining space and data partitioning, but it is limited to point datasets where a clean split can always be made within a sub-region along a specific dimension. In contrast, the Nir+-tree eliminates overlap by clipping MBBs into polygonal shapes, making it suitable for volumetric datasets. However, the Nir+-tree does not provide an upper bound guarantee on memory usage, potentially leading to excessive memory consumption when constructing complex polygons.

All the space-partitioned and data-partitioned indexing methods mentioned above are more suitable for low-dimensional spatial data, such as 2D or 3D, and do not scale well to higher dimensions due to the "curse of dimensionality". To address the challenges of high-dimensional data, work such as the TV-tree [33] and X-tree [13] were proposed. The TV-tree allows selective dimensionality reduction by indexing only a subset of the most relevant dimensions, thereby improving both storage efficiency and query performance. The X-tree extends the R-tree by utilizing supernodes, which are larger than standard nodes and can hold more entries. This design reduces the likelihood of node splits and minimizes overlap, which is particularly problematic in high-dimensional spaces.

Recently, learned indexing and Approximate Nearest Neighbor (ANN) vector search on embeddings have gained increased attention in multi-dimensional data indexing. The RLR-Tree [21] employs reinforcement learning to develop learned models for node expansion and splitting. A key challenge in this approach is designing an effective reward function that strongly correlates node selection for expansion or splitting with improvements in query

performance. With the growing importance of vector search and embeddings, specialized ANN indexing structures like HNSW [35] and IVF [28] have been developed. HNSW (Hierarchical Navigable Small World) uses a multi-layer graph to enable fast and scalable nearest neighbor queries by navigating through increasingly fine-grained neighborhoods. IVF (Inverted File Index) clusters data into centroids and builds an inverted index for each, narrowing the search space to relevant clusters for efficient querying.

6.3 Bounding Shapes

Minimum Bounding Boxes (MBBs) offer several advantages, such as efficient intersection tests, linear construction costs, and low storage requirements, since they are represented by just two multi-dimensional points [45]. However, this straightforward representation only captures the minimum and maximum values for each dimension, which can lead to substantial empty space within the bounding volume and degrade performance, as discussed in the Background chapter. Other geometrics have also been proposed to achieve a tighter fit, including the use of more complex shapes like spheres (balls) or polygons. These alternatives trade storage and computational efficiency for a more accurate representation of the data. In this chapter, we explore various geometric options for bounding shapes beyond MBBs.

The Sphere-tree [46] utilizes bounding spheres (balls) instead of rectangles, encapsulating data points within spherical volumes. This structure is particularly advantageous in scenarios where distance between objects are crucial, such as in k-nearest neighbor (KNN) queries. Spheres can often provide a tighter fit around clusters of objects compared to rectangular boxes, thereby reducing the amount of empty space within the bounds. However, constructing a spherical bounding volume for a node is more computationally intensive than using a rectangle.

SS-tree (Similarity-Search-tree) [47] also utilize bounding spheres like Sphere-tree, and it introduces optimizations specifically aimed at reducing overlap and improving search efficiency in high-dimensional spaces. The SS-tree also refines the way spheres are used to encapsulate data, optimizing the tree structure for similarity searches.

SR-tree (Sphere-Rectangle tree) [30] integrates both bounding spheres and bounding rectangles in its structure. The coverage of a node is defined as the intersection of a sphere and a rectangle, combining the strengths of both shapes to provide a tighter and more accurate fit around the data. This hybrid approach minimizes overlap and dead space within the node, leading to improved query performance. The SR-tree enhances the

flexibility and efficiency of spatial indexing by balancing the advantages of spherical and rectangular bounds. However, this increased complexity results in higher computational costs during construction and maintenance, as well as greater storage requirements due to the need to store both bounding shapes.

Sphere is a widely used bounding shape in addition to rectangle, particularly in similarity search applications. Several other shapes are also popular choices for spatial indexing. The P-tree (Polygon-tree) [26] utilizes polygonal bounding regions to better enclose objects with greater precision. This approach aligns more naturally to skewed data distributions compared to axis-aligned boxes, reducing unnecessary overlaps and dead space. The eR-tree (ellipsoid R-tree) [15] employs more complex bounding shapes, specifically Minimum Volume Covering Ellipsoids (MVCE), which are more adaptable to the inherent distribution of the data, potentially offering a more precise fit and improved partitioning. The Cell-tree [22] uses a hierarchy of nested convex polyhedra, with leaf nodes represented as unions of convex point sets, offering flexibility in representing complex spatial regions.

6.4 Augmentation Techniques

Instead of using more complex geometric shapes to reduce dead space as last subchapter, this thesis work eliminates dead space by utilizing an auxiliary structure that stores additional information on top of the Minimum Bounding Box (MBB). MBBs have a proven track record of success due to their efficiency in computation and low storage requirements, which is why they have been widely adopted in various indexing methods. This makes our approach compatible with many existing index structures. Previous work have also proposed enhancing query performance through auxiliary structures. In this chapter, we explore various augmentation techniques that can be integrated with multiple indexing methods as well as improve query performance.

Mambo [16] uses bitmaps to index dead space, similar to GMB. However, Mambo creates a uniform grid of cells over the entire dataset. Because Mambo's grid cells are uniformly sized, it provides less detailed information in densely populated areas, making it less efficient for datasets with skewed distributions. In contrast, GMB customizes the bitmap for each leaf node, taking into account the node's characteristics and the local data distribution. GMB uses smaller grid cells for more compact leaf nodes, offering greater precision, and larger cells for nodes with broader coverage to provide more efficiency. Another key difference is that Mambo requires prior knowledge of the global boundary values for all objects in the dataset to construct the bitmap. If new data objects fall outside these boundaries, a completely new bitmap must be recalculated for all objects,

making updates inefficient. Deletions also require the entire bitmap to be recreated. GMB, on the other hand, supports more frequent updates, as bitmap is much smaller scaled and recreation is more efficient. GMB employs a lazy update strategy for deletions, and a new grid bitmap could be triggered when the MBB is updated.

The BR-tree [24] augments the entire dataset with an additional data structure, similar to Mambo. Specifically, the BR-tree employs a bloom filter by hashing all objects, which can be highly efficient for point queries. Like a bitmap, the Bloom filter allows for quick existence checks without the need to traverse the tree. Bloom filters use multiple hash functions to map objects into a bit array, resulting in a probabilistic structure that can introduce false positives but never false negatives. While using a larger bloom filter can increase accuracy by reducing false positives, it comes at the cost of additional storage. The BR-tree also supports dynamic deletions by using an extra counter. However, it's important to note that bloom filters do not enhance the performance of range searches, as the locality properties of the data are not preserved after hashing, making it unsuitable for efficiently identifying all objects within a specific range.

Like GBBs, Clipped Bounding Boxes (CBBs) [45] are designed to reduce dead space by augmenting MBBs with an auxiliary data structure stored in main memory. CBBs accomplish this by clipping the corners of the MBB at each node using score-based selective clipping points. Those points are determined through a technique known as skylines (Pareto optimality) [14]. The authors also introduce stairline points to further expand the clipping area, thereby enhancing dead space elimination. To optimize storage and computation costs, only the clipping points that yield the most significant reduction in dead space are selected. However, the skyline algorithm for computing these clipping points is computationally intensive. Moreover, the effectiveness of CBBs relies on the assumption that most dead space occurs at the corners of the MBB, which may not always hold true, especially for point datasets. In contrast, GMB offers a simpler and more robust approach that does not rely on specific assumptions about the locality of dead space.

Chapter 7

Conclusion

Minimum Bounding Boxes (MBBs) are widely used in spatial data management as efficient means of approximatively representing clustered objects, offering reliable true negative results in intersection tests. However, when clustering is suboptimal, dead space within MBBs can lead to a high rate of false positives, significantly diminishing query performance. To address this issue, we proposed the Grid Minimum Bounding Box (GMB), which indexes dead space within the MBB adaptively and dynamically. Using a lightweight data structure, substantial information about dead space is encoded into a bitmap. As an auxiliary data structure, GMB offers the additional advantage of seamless integration into R-tree variants that rely on MBBs.

With the implementation of the grid-bitmap, we observed significant improvements in query performance. The GMB technique not only reduces the dead space within nodes but also enhances the overall efficiency of spatial queries by providing more precise information about the distribution of objects within the MBB. Our experiments demonstrate the effectiveness of this approach across various datasets and R-tree variants, highlighting the robustness of GMB at improving spatial data indexing and querying.

In conclusion, the GMB presents a promising solution to the challenge of dead space in MBBs, offering a balance between storage overhead and query accuracy. Future work could explore the application of grid bitmap techniques to branch nodes and further refine the algorithm to adapt to varying data distributions and query patterns.

References

[1] Apache Sedona — sedona.apache.org. https://sedona.apache.org/1.6.0/. [Accessed: 2024-08-15].

[2] ArcGIS Pro — esri.com. https://www.esri.com/en-us/arcgis/products/arcgis-pro/overview. [Accessed: 2024-08-15].

[3] PostGIS — postgis.net. https://postgis.net/. [Accessed: 2024-08-15].

[4] PyGIS — pygis.io. https://pygis.io/docs/a_intro.html. [Accessed: 2024-08-15].

[5] QGIS — qgis.org. https://www.qgis.org/. [Accessed: 2024-08-15].

[6] Fadhil Abubaker. Disk-based indexing for nir-trees using polygon overlays. Master's thesis, University of Waterloo, 2024. [Accessed: 2024-08-15].

[7] Lars Arge, Mark de Berg, Herman Haverkort, and Ke Yi. The priority r-tree: A practically efficient and worst-case optimal r-tree. *ACM Transactions on Algorithms (TALG)*, 4(1):1–30, 2008.

[8] Rudolf Bayer and Edward McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, page 107–141, New York, NY, USA, 1970. Association for Computing Machinery.

[9] 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*, pages 322–331, 1990.

[10] Norbert Beckmann and Bernhard Seeger. A benchmark for multidimensional index structures. https://www.mathematik.uni-marburg.de/~rstar/benchmark/distributions.pdf, 2008. [Accessed: 2024-08-15].

[11] 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*, pages 799–812, 2009.

[12] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

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

[14] Stephan Borzsony, Donald Kossmann, and Konrad Stocker. The skyline operator. In *Proceedings 17th international conference on data engineering*, pages 421–430. IEEE, 2001.

[15] Ondrej Danko. Eliptic indexing of multidimensional databases. Master's thesis, Univerzita Karlova, Matematicko-fyzikální fakulta, 2008. [Accessed: 2024-08-15].

[16] 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*, pages 73–84, 2021.

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

[18] Thanasis Georgiadis and Nikos Mamoulis. Raster intervals: an approximation technique for polygon intersection joins. *Proceedings of the ACM on Management of Data*, 1(1):1–18, 2023.

[19] 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.

[20] Teofilo F Gonzalez. On the computational complexity of clustering and related problems. In *System Modeling and Optimization: Proceedings of the 10th IFIP Conference New York City, USA, August 31–September 4, 1981*, pages 174–182. Springer, 1982.

[21] Tu Gu, Kaiyu Feng, Gao Cong, Cheng Long, Zheng Wang, and Sheng Wang. The rlr-tree: A reinforcement learning based r-tree for spatial data. *Proceedings of the ACM on Management of Data*, 1(1):1–26, 2023.

[22] Oliver Gunther. The design of the cell tree: An object-oriented index structure for geometric databases. In *Proceedings. Fifth International Conference on Data Engineering*, pages 598–599. IEEE Computer Society, 1989.

[23] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, 1984.

[24] Yu Hua, Bin Xiao, and Jianping Wang. Br-tree: A scalable prototype for supporting multiple queries of multidimensional data. *IEEE Transactions on Computers*, 58(12):1585–1598, 2009.

[25] Andreas Hutflesz, Hans-Werner Six, and Peter Widmayer. Twin grid files: Space optimizing access schemes. *ACM Sigmod Record*, 17(3):183–190, 1988.

[26] H Vi Jagadish. Spatial search with polyhedra. In *Proceedings. Sixth International Conference on Data Engineering*, pages 311–312. IEEE Computer Society, 1990.

[27] Hosagrahar V Jagadish. Linear clustering of objects with multiple attributes. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 332–342, 1990.

[28] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2010.

[29] Ibrahim Kamel and Christos Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 500–509. Citeseer, 1994.

[30] Norio Katayama and Shin'ichi Satoh. The sr-tree: An index structure for high-dimensional nearest neighbor queries. *ACM Sigmod Record*, 26(2):369–380, 1997.

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

[32] 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*, pages 157–168, 2021.

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

[34] David B Lomet and Betty Salzberg. The hb-tree: A multiattribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems (TODS)*, 15(4):625–658, 1990.

[35] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

[36] Donald Meagher. Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-d objects by computer. *Electrical and Systems Engineering Department Rensseiaer Polytechnic Institute Image Processing Laboratory*, 1980.

[37] Bongki Moon, Hosagrahar V Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on knowledge and data engineering*, 13(1):124–141, 2001.

[38] Guy M Morton. A computer oriented geodetic data base and a new technique in file sequencing. *International Business Machines Company New York*, 1966.

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

[40] Jürg Nievergelt, Hans Hinterberger, and Kenneth C Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems (TODS)*, 9(1):38–71, 1984.

[41] Octavian Procopiuc, Pankaj K Agarwal, Lars Arge, and Jeffrey Scott Vitter. Bkd-tree: A dynamic scalable kd-tree. In *Advances in Spatial and Temporal Databases: 8th International Symposium, SSTD 2003, Santorini Island, Greece, July 2003. Proceedings 8*, pages 46–65. Springer, 2003.

[42] 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, page 263–272, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[43] John T Robinson. The kdb-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 10–18, 1981.

[44] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th International Conference on Very Large Data Bases*, VLDB '87, page 507–518, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc.

[45] 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. IEEE, 2018.

[46] Peter van Oosterom and Eric Claassen. Orientation insensitive indexing methods for geometric objects. In *Proceedings of the 4th International Symposium on Spatial Data Handling*, volume 2, pages 1016–1029, 1990.

[47] David A White and Ramesh Jain. Similarity indexing with the ss-tree. In *Proceedings of the Twelfth International Conference on Data Engineering*, pages 516–523. IEEE, 1996.