RANGE-AGGREGATE QUERY PROBLEMS
IN COMPUTATIONAL GEOMETRY

By
Saladi Rahul
200401070

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science (by Research) in
Computer Science & Engineering

Lab for Spatial Informatics
International Institute of Information Technology
Hyderabad, India
July 2011
Dedicated to my mother Mrs. Surekha and my father Mr. Durgaprasad.

Whatever little I have achieved in life has been possible only because of them.
Love you.
It is certified that the work contained in this thesis, titled “Range-Aggregate Query Problems in Computational Geometry” by Saladi Rahul (200401070) submitted in partial fulfillment for the award of the degree of Master of Science (by Research) in Computer Science & Engineering, has been carried out under my supervision and it is not submitted elsewhere for a degree.

Date

Advisor :
Dr. K. S. Rajan and Dr. Prosenjit Gupta
Acknowledgements

Two people who have motivated me to take up a career in research are Prof. Prosenjit Gupta and Prof. K. S. Rajan. To be honest, till the end of my second year of B.tech I really did not have any reason as to why I was studying computer science. It was in the third year of my B.tech that I was exposed to the exciting field of computational geometry by Prof. Prosenjit Gupta. This project coupled with the course of computational geometry made me realize the following: that studies can be interesting and that what we study does have a real-world utility ! I got a chance to work on interesting problems (which looked more like puzzles), yet they had real-life applications. Prof. Prosenjit Gupta has always been highly rated as a researcher and a teacher by the students. The discipline with which he does his work is perhaps his most remarkakable quality.

After Prof. Prosenjit Gupta moved away from the campus, I was lucky to get a chance to work with Prof. K. S. Rajan. Under the able guidance of Prof. Prosenjit Gupta and Prof. K. S. Rajan, I decided to shift to the dual degree course. To the best of my knowledge, Prof. K. S. Rajan is the most hardworking professor I have met. He loves his work so much that it does not even look like hardwork to him. I will always cherish the long discussions I have had with him (some of which even crossed four hours !).

Ananda Swarup Das has always been patient enough to listen to my work. He was like a big brother and was always there to support me. I would also like to thank Haritha Bellam for co-authoring one of my papers.

I have been lucky to have made a lot of friends in the campus. I would like to thank my batchmates, seniors and juniors for having made my stay in the campus
the most memorable period of my life. In “utopia” gang and ‘gola-gang”, I have made some friends for life time. I have had a great time playing along with the footballers (along with PD sir), cricketers, hockey players and other athletes in the campus. Lastly, I would like to thank the staff and faculty members of IIT-H for building such a high-quality institute. It was a privilege studying at IIT-H.
Abstract

Computational Geometry has grown into an independent field in the last two decades. It deals with design and rigorous analysis of sophisticated algorithms for problems which involve geometric objects. Apart from trying to come up with efficient algorithms and data structures, a lot of focus is on mathematical analysis for proving the efficiency of the solutions. Computational Geometry problems find a lot of applications in the field of VLSI, Geographic Information Systems (GIS), graphics and databases.

In a class of problems named ‘geometric intersection searching problems’ (GISP) orthogonal range searching, halfspace range searching, orthogonal point enclosure have been some of the most widely studied problems. The inherent beauty in these problems and their applications in various domains have motivated researchers to explore these problems. GISP can be simply stated as follows: Let $S$ be a set of objects in $\mathbb{R}^d$ ($d$-dimensional space). Given a query region $q$, report or count the number of objects in $S \cap q$. The objective is to come up with data structures which uses minimal space (ideally linear space) and query algorithms which answer queries fast!

In many applications (which shall be discussed later), it is not sufficient to merely report or count the objects of $S$ intersecting with the query range $q$. Some form of aggregation (or summarization in some cases) needs to be done on the objects in $S \cap q$ to present the user with more useful information. Range-Aggregate queries are a step in that direction and are the focus of this thesis.
A range-aggregate query can be formally defined as follows: Let $S$ be a set of $n$ objects in $\mathbb{R}^d$, where each object $p$ may have a real-valued weight $w(p)$. We would like to preprocess $S$ into a suitable data structure so that for any given query range $q \subseteq \mathbb{R}^d$, a certain aggregation function that operates on the objects of $S' = S \cap q$ can be performed efficiently and output-sensitively. Existing techniques in the literature cannot be directly applied to answer these queries efficiently. Therefore, in each chapter we consider a unique aggregation function and study them in detail.

In many applications reporting all the points in $S \cap q$ is not a user-friendly experience since $|S \cap q|$ can be very huge and hence, unmanageable. Therefore, we use the aggregation operator ‘top-$k$’ to report only the $k$ most significant points based on the weights associated with each point.

In many scenarios, the points in $S$ are classified into disjoint groups. We associate a unique color with each group and all the points belonging to that group also inherit that color. Given a query region $q$, a straightforward query is to report the unique colors of the points in $S \cap q$. In the range-aggregate versions we considered, additional information would be reported along with each color $c$ such as the point of color $c$ in $q$ with the maximum weight or the sum of the weights of all the points of color $c$ in $q$. In short, a useful summary of each group is reported.

Geometric aggregation functions are relatively more difficult to handle than the regular aggregate functions. We consider some geometric aggregation problems and improve upon the existing solutions in the literature. Finally, we consider a boolean yes or no query asking if $|S \cap q| > 0$ or not. Though existing structures in the literature can answer this query in reasonable query time and space we made an attempt to further improve them.

We come up with efficient data structures and query algorithms to solve these problems. Mathematical analysis is done to show the efficiency of our solutions in terms of space occupied and query time. Real world applications of each type of range-aggregate query is explained as well.
Publications


  A preliminary version of the work presented in Chapter 5 was published at CCCG 2009. The full version was accepted for publication in IJFCS.


  The work done in this paper is presented in Chapter 3.


  Since this work was done along with the Phd student Ananda Swarup Das, I could not put all of the results in my thesis. However, baring one result rest of the work presented in this paper is in Chapter 4.

- Saladi Rahul and Krishnan Rajan. One-Reporting queries, *26th European Workshop on Computational Geometry (preliminary version), The First International Conference on Computer Science and Information Technology (COSIT-2011).*

  The work done in this paper is presented in Chapter 7.


  Chapter 6 is presented as a kind of continuation of Chapter 5. A preliminary version of this work was accepted at CCCG 2010.
# Contents

Table of Contents x  
List of Tables xiii  
List of Figures xv  

1 Introduction 1  
1.1 A brief history ................................................. 1  
1.1.1 Output sensitivity ........................................ 2  
1.2 Range-Aggregate Queries .................................... 4  
1.2.1 Applications of range-aggregate queries ............... 4  
1.2.2 What makes them non-trivial to solve ? ............... 6  
1.2.3 Top-k queries .............................................. 7  
1.2.4 Geometric Aggregate queries ......................... 8  
1.2.5 Colored Range-Aggregate queries .................. 10  
1.2.6 One-Reporting queries ............................... 12  
1.3 Flow of the thesis ............................................ 13  

2 Review of fundamental structures and techniques 14  
2.1 Interval Tree ................................................... 14  
2.2 Segment Tree .................................................. 16  
2.3 Range Trees ................................................... 17  
2.4 Priority Search Tree ....................................... 18  

3 Efficient top-k queries for orthogonal ranges 20  
3.0.1 Related Work ............................................... 22  
3.0.2 Formal Problem Statement .......................... 27  
3.1 Dynamic structure for reporting top-k points in arbitrary order ........... 28  
3.1.1 Handling insertions and deletions .................. 34  
3.2 Reporting top-k points in sorted order ............... 35  
3.2.1 Solution for $\mathbb{R}^1$ ................................. 36  
3.2.2 Solution for $\mathbb{R}^d, d\geq 2$ ....................... 37  
3.3 Conclusions and Future Work .......................... 39  

## 4 Range-Aggregate Queries Involving Geometric Aggregation Operations

4.1 Static Range-Aggregate Point Enclosure ................................................. 41
  4.1.1 One-dimensional scenario. ................................................................. 43
  4.1.2 Extension to higher dimensions \( \left( d \geq 2 \right) \). .......................... 44
4.2 Semi-dynamic Range-Aggregate Point Enclosure ................................. 46
  4.2.1 One-dimensional scenario. ................................................................. 46
  4.2.2 Extending it to higher dimensions ...................................................... 48
4.3 1-d Range-Aggregate Segment Intersection ......................................... 49
  4.3.1 Static Solution ....................................................................................... 49
  4.3.2 Semi-dynamic solution ......................................................................... 50
4.4 Range-Aggregate Orthogonal Segment Intersection ............................ 51
4.5 Conclusions and open problems ............................................................... 55

## 5 Range aggregate structures for colored geometric objects-I

5.0.1 Motivation .............................................................................................. 57
5.1 Our Contributions ...................................................................................... 57
5.2 Related Work ............................................................................................. 58
5.3 The colored weighted sum problem .......................................................... 64
  5.3.1 Query composition technique .............................................................. 64
  5.3.2 The solution for \( \mathbb{R}^1 \) .................................................................. 66
  5.3.3 Adding range restrictions ..................................................................... 67
  5.3.4 Colored weighted sum problem for \( d = 2 \) ......................................... 70
  5.3.5 Colored weighted sum problem for \( d > 2 \) ......................................... 71
5.4 The Colored Bounding Box Problem ....................................................... 72
  5.4.1 A solution in \( \mathbb{R}^1 \) ..................................................................... 72
  5.4.2 Extending to \( \mathbb{R}^2 \) ...................................................................... 74
  5.4.3 A solution for \( \mathbb{R}^3 \) ................................................................... 76
  5.4.4 A solution for \( d \geq 3 \) ................................................................. 77
5.5 Nontrivial colored bounding box problem .............................................. 78
  5.5.1 Querying with quadrants in \( \mathbb{R}^2 \) ................................................. 78
  5.5.2 Querying with three sided rectangle in \( \mathbb{R}^2 \) ............................... 83
5.6 Colored point enclosure weighted sum for \( d = 1 \) .............................. 85
5.7 Colored point enclosure weighted sum for \( d = 2 \) .............................. 86
5.8 Colored segment intersection weighted sum ....................................... 88
5.9 Conclusions and Open Problems ............................................................ 89

## 6 Range aggregate structures for colored geometric objects-II

6.1 Terminology .............................................................................................. 91
6.2 Generalized Orthogonal Range-Max query .......................................... 93
  6.2.1 Dynamic one-dimensional problem: Handling only insertions .......... 93
  6.2.2 Dynamic one-dimensional problem: handling both insertions and deletions ................................................................. 97
  6.2.3 Two dimensional scenario ................................................................. 101
CONTENTS

6.2.4 Generalized orthogonal range-max query for \( d = 3 \) ............ 105
6.2.5 Generalized weighted sum for \( d > 3 \) .......................... 106
6.3 Generalized orthogonal stabbing-max query .......................... 106
   6.3.1 Solution for \( d = 1 \) ........................................... 107
   6.3.2 Solution for \( d = 2 \) ........................................... 107
   6.3.3 Solution for \( d \geq 3 \) ........................................ 109
6.4 Generalized \( d \)-dimensional Range Searching on randomly distributed points 109
6.5 Generalized \( d \)-dimensional Point Enclosure on randomly distributed hyper-boxes .......................... 111
6.6 Open problems .................................................. 111

7 Emptiness or One-reporting queries for Standard Intersection problems 113
   7.0.1 Model of Computations .......................... 114
   7.0.2 Note on the evaluation of the data structures .......................... 115
7.1 Results obtained and Comparision with previous results .......................... 115
7.2 Definitions .................................................. 118
7.3 Orthogonal Range Emptiness query .................................. 119
   7.3.1 One-reporting dominance queries in \( \mathbb{R}^2 \) ................ 119
   7.3.2 One-reporting Dominance problem in \( \mathbb{R}^3 \) .................. 121
   7.3.3 One-reporting dominance problem in \( \mathbb{R}^d, d > 3 \) .......... 122
   7.3.4 One-reporting for bounded orthogonal rectangle queries on \( \mathbb{R}^2 \) .. 123
   7.3.5 One-reporting queries for bounded orthogonal boxes in \( \mathbb{R}^d \) .... 124
7.4 Dominance Emptiness query .................................. 125
7.5 Point Enclosure emptiness query .................................. 127
   7.5.1 Data Structure ........................................ 127
   7.5.2 Analysis ........................................ 128
7.6 Halfplane Range emptiness query ................................ 129
7.7 Conclusions and open problems .................................. 132

8 Conclusions and Open Problems .................................. 133

Bibliography ........................................ 136
List of Tables

3.1 Summary of results obtained for top-$k$ queries for orthogonal ranges. $k$ is also part of the query. Two settings are considered. The first one reports the top-$k$ points in arbitrary order and the other in sorted order by weight. All bounds are worst-case results. .................................................. 28

4.1 Summary of results for semi-dynamic range-aggregate query problems; the query $q$ is an orthogonal range; $k$ is the output size. Insertion time mentioned in amortized time. ................................................................. 41

4.2 Comparison of results of static range-aggregate query problems; the query is an orthogonal range; $k$ is the output size. ................................................................. 42

5.1 Summary of results. All results are “big-Oh” and worst case. Rectangles are axis-parallel. “three sided rectangle” means an orthogonal box of the form $[a_1, b_1] \times [a_2, \infty)$. The “nontrivial bounding box” function returns a bounding box of all those colors that have at least two points that intersect $q$. A “bounding box” for a point set in $\mathbb{R}^d$ is the hyper box with the smallest measure within which all the points lie. $\epsilon$ is an arbitrarily small positive constant. $C$ denotes the number of colors reported. .................................................. 65

6.1 Summary of results obtained for Generalized Orthogonal Range-Max Query. Set $S$ consists of $n$ points lying in $\mathbb{R}^d$ ($d \geq 1$). $k$ denotes the number of colors reported. All results are “big-Oh” and “worst-case” bounds. ............ 92

6.2 Summary of results obtained for Generalized Orthogonal Stabbing-Max Query. Set $S$ consists of $n$ orthogonal hyperboxes in $\mathbb{R}^d$. Query, $q$, is a point in $\mathbb{R}^d$. $k$ is the number of colors reported. All results are “big-Oh” and “worst-case” bounds. .................................................. 92

7.1 The query time of the data structures in the best, average and the worst case scenario for $d>3$ are mentioned. Queries are of the form $\Pi_{i=1}^d[a_i, b_i]$. ............ 116

7.2 Comparison of results for Dominance emptiness problem. Afshani et. al. [2] is the most efficient solution to the reporting version of this problem. ............ 117

7.3 Comparison of results for Point Enclosure emptiness problem. Chazelle et. al. [21] is the most to the reporting version of this problem. ............ 117
7.4  Comparison of results for Hyperplane emptiness problem. Chazelle et. al. [20] is the most efficient solution to the reporting version of this problem. . . . . . . . 118
List of Figures

3.1 Examples for finding the threshold point. ........................................... 32
4.1 Different types of vertical and horizontal segments that can intersect q. .... 52
6.1 For a particular color c, set $M_c$ is being shown. These points are the only candidates from color c which can have the topmost/maximum weight for a Generalized Orthogonal Range-Max query. ......................................................... 102
7.1 P1, P2, P3 and P4 are the maximal points under the Dominance Criteria $C = \{\geq, \leq\}$. .......................................................... 129
7.2 P1, P5 and P6 are the maximal points under the Dominance Criteria $C = \{\leq, \leq\}$. .......................................................... 130
Chapter 1

Introduction

1.1 A brief history

Computational Geometry emerged from the field of algorithm design and analysis in the late 1970s. It is a subfield of algorithm theory that involves the design and analysis of efficient algorithms for problems involving geometric input and output [79]. It has grown into a well recognized and a successful discipline. The beauty of the problems studied and the solutions obtained; and on the other hand the various application domains (geographical information systems (GIS), VLSI, robotics, computer graphics, and others) in which geometric algorithms play a fundamental role make it a very interesting field of research.

The problems solved in the field of Computational Geometry can be classified into two categories: (a) single-shot problems, and (b) multi-shot problems. In a single-shot problem, for a given input, the corresponding output needs to be found or constructed. This computation is done only once and re-computation is not needed as it would lead to the same result. Some classic examples of single-shot problems are:

- **Convex Hull**: Given a set $S$ of points, find the smallest convex polyhedron/polygon containing all the points of $S$.

- **Closest pair problem**: Given $n$ points in the $d$-dimensional euclidean space, find a
pair of points with the smallest distance between them.

- **Line Segment intersection**: Given a set of $n$ segments in the plane, report all the pair of segments which intersect with each other.

In a **multi-shot** problem, the given input is typically preprocessed into a data structure so that given multiple queries, the output for each of the queries can be answered efficiently. A very important class of multi-shot problems which also form the central theme of this thesis are the **geometric intersection searching problems** (GISP). The most exhaustively studied problem under GISP has been the **range searching** problem. A typical range searching problem has the following form: Let $S$ be a set of $n$ points in $\mathbb{R}^d$, and let $\mathcal{R}$ be a family of subsets of $\mathbb{R}^d$; elements of $\mathcal{R}$ are called ranges. We wish to preprocess $S$ into a data structure so that for a query range $q \in \mathcal{R}$, the points in $S \cap q$ can be reported or counted efficiently. Typical examples of ranges include rectangles (or hyperboxes), halfspaces, simplices and balls. If we are interested in answering a single query, it can be done in linear time, using linear space by simply checking for each point $p \in S$ whether $p$ lies in the query range $q$. This would then be classified as a single-shot problem. However, in many applications the same set $S$ is queried several times, in which case we would like to answer a query faster by preprocessing $S$ into a data structure. Range searching has applications in fields such as databases and GIS. [5] is an excellent survey on range searching. Other examples of GISP include **point enclosure**, **segment intersection** etc.

To fully understand the rest of the chapter, it would be very useful to understand the concept of output sensitive solutions which is a fundamental guideline for developing solutions in computational geometry. Therefore, we shall explain it here.

### 1.1.1 Output sensitivity

Developing *output sensitive* solutions is at the heart of all problems considered in the field of computational geometry. We shall explain this concept with the help of a very simple problem (which is actually not a geometric problem !): An array $A$ consists of $n$ integers
stored in a sorted order. Given a query range \( q = [a_1, b_1] \), we want to report all the integers in \( A \) which lie in \( q \).

One could argue that the time taken to answer this query would be \( \Theta(n) \) in the worst case. However, observe that the worst case (of reporting \( O(n) \) elements) would not always occur. Therefore, we try to analyze the query time in terms of both \( n \) and \( k \), where \( k \) are the number of elements reported for a given query \( q \). A smart query algorithm for this problem would be the following: Do a binary search on \( A \) with \( a_1 \) and find the smallest integer which is greater than \( a_1 \). Then scan the adjacent integers till we find an integer greater than \( b_1 \) or the end of the array is reached. Now a routine query time analysis of this algorithm would be \( O(n) \) (worst case scenario). If we add the parameter \( k \) to our analysis, the query time turns out to be \( O(\log n + k) \). This analysis gives us greater insight of the smartness and efficiency of the algorithm.

A dumb query algorithm for this problem would be the following: Given a query \( q \), scan the array from the beginning till the end and report all the integers which satisfy the query. Even if we use parameter \( k \) along with \( n \) to analyze this algorithm it would still remain \( O(n) \). It is clear that the algorithm stated in the previous paragraph will do better than the dumb query algorithm stated in this paragraph. This supremacy of the smart algorithm over the dumb algorithm is shown mathematically by the use of the additional parameter \( k \), which denoted the size of the output and hence the name, output sensitive solution. The smart algorithm is output sensitive while the dumb algorithm is not output sensitive. In most of the real world problems, as the data size increases rapidly, the \( O(n) \) computation can lead to very high response times making the solution highly inefficient whereas, an output sensitive algorithm will improve its performance many-fold.

In this thesis we come up with output sensitive query algorithms for all the problems, i.e., the query time is of the form \( O(f(n) + k) \) or \( O(f(n) + kg(n)) \), where \( f(n) \) and \( g(n) \) are typically \( O(\log^c n) \) (for \( c > 0 \)) or \( O(n^\epsilon) \) (for \( \epsilon > 0 \)).
CHAPTER 1. INTRODUCTION

1.2 Range-Aggregate Queries

There arise situations in real-world in which it is not sufficient to merely report or count
the objects of $S$ intersecting with the query range $q$. In applications like on-line analyt-
cical processing (OLAP), geographic information systems (GIS) and information retrieval
(IR), aggregation plays an important role in summarizing information [86] and hence large
number of algorithms and storage schemes have been proposed to support such queries.

In a class of problems called range-aggregate query problems [86] many composite
queries involving range searching (or other GISP) are considered, wherein one needs to
compute the aggregate function of the objects in $S \cap q$ rather than report (or count) all of
them as in a range reporting (or counting) query [59, 48]. A formal definition is stated next:

- Let $S$ be a set of $n$ objects in $\mathbb{R}^d$, where each object $p$ may have a real-valued weight
  $w(p)$. We would like to preprocess $S$ into a suitable data structure so that for any
given query range $q \subseteq \mathbb{R}^d$, a certain aggregation function that operates on the objects
of $S' = S \cap q$ can be performed efficiently and output-sensitively.

1.2.1 Applications of range-aggregate queries

Consider a generic database having a table (call it $table$) having four attributes $attr_1$, $attr_2$,$attr_3$ and $attr_4$. A GROUP-BY operation is one of the most important SQL queries which
looks as follows:

\[
\text{SELECT } attr_3 \text{ FROM table }
\]

\[
\text{WHERE } 10 < attr_1 < 20 \text{ AND } 25 < attr_2 < 35
\]

\[
\text{GROUP BY } attr_3
\]

We map the above database query to a geometric problem as follows: Each tuple in
the relation $table$ can be mapped to a two-dimensional point in the plane where the $x$-
coordinate and $y$-coordinate are $attr_1$ and $attr_2$. All these points are now categorized into
groups based on their attribute $attr_3$ values (points having the same $attr_3$ value belong to
the same group). For simplicity, each group is assigned a unique color and all the points in 
that group also take up that color. Therefore, we now have a set $S$ of colored points in two-
dimensional plane which need to be pre-processed and kept in a data structure so that given 
a query rectangle $q$ (in the above example $q = [10, 20] \times [25, 35]$), all the unique colors of 
the points of $S$ in $q$ need to be reported. This problem is known as ‘colored/generalized 
range searching’ [51] in computational geometry literature. A GIS system would benefit 
from this query as follows: Suppose we have a set of locations on the earth and we associate 
the soil type of that area with each location. Now given a region on the earth, an interesting 
query would be to find out all the soil types available in that query region. Colored range 
searching and related problems have been extensively studied in the literature [51, 85, 50, 
4, 1, 13, 52, 61, 60]. In Chapter 5 and Chapter 6 we build efficient index structures for SQL 
queries applying aggregation and GROUP-BY operators.

Suppose we have a map of location of villages in Andhra Pradesh and the population 
associated with each city. A demographer might be interested in knowing the following 
information: a) What is the total population of all the villages in a particular district ? b) 
Which is the village with the maximum population in a particular district. The above two 
questions can be posed as SQL queries to a database as follows:

SELECT SUM(attr_3) FROM table 
WHERE 10 < attr_1 < 20 AND 25 < attr_2 < 35

SELECT MAX(attr_3) FROM table 
WHERE 10 < attr_1 < 20 AND 25 < attr_2 < 35

where $attr_1$ and $attr_2$ are locational attributes and $attr_3$ corresponds to population. 
These two queries can be mapped to appropriate geometric problems (as done previously) 
which have been studied in the literature [3, 14].
1.2.2 What makes them non-trivial to solve?

Till now we have only shown the real-world applications of range-aggregate queries. The next question to ponder over is “Are they trivial to solve or not?”. Consider the third SQL query (MAX operator) mentioned in the previous sub-section whose geometric equivalent was the following problem (Agarwal et al. [3]): “A set $S$ of points lie in $\mathbb{R}^2$ and each point has a weight associated with it. Given a query rectangle $q$, the point in $S \cap q$ with the ‘maximum’ weight needs to be reported efficiently”. Instead of reporting all the points in $S \cap q$, we need to apply the aggregation operation “maximum” on $S \cap q$ and report the point having the maximum weight. Existing structures in the literature cannot be used directly to answer this query. For eg, we want to use a two-dimensional range tree to answer this query. A two-dimensional range tree is typically used to answer an orthogonal range reporting query. One naive method would be report all the points lying inside the query orthogonal rectangle using the range tree (i.e $S \cap q$) and then scanning $S \cap q$ to report the point with the “maximum” weight. Though the algorithm is correct, its running time will be $O(\log n + |S \cap q|)$. If $|S \cap q| \gg O(1)$, then scanning part of the algorithm becomes very expensive and hence output insensitive. Applying such traditional structures for solving range-aggregate problems typically leads to output insensitive solutions. This leads to the need for novel and non-trivial solutions for answering range-aggregate queries. The objective while coming up with an algorithm for a range-aggregate problem is to build a linear or near-linear space data structure with query time being sensitive to the output size. Similar example aggregate functions include “minimum” and “sum”.

In this thesis we have considered different kinds of range-aggregate query problems and have tried to come up with efficient solutions. Strong real-world applications for each of the problems had lead to the curiosity to investigate and attempt these problems. First, we shall give a brief classification of aggregate functions [83, 47]:

1. Distributive aggregates can be computed by partitioning the input into disjoint sets, aggregating each set individually and then obtaining the final result by further aggre-
CHAPTER 1. INTRODUCTION


gation of the partial results. minimum, maximum, sum, count and top-k fall under this category.

2. Algebraic aggregates (e.g. average) can be expressed as a function of distributive aggregates. In most of the problems (considered in this thesis), aggregate functions considered fall under the first two categories.

3. Holistic aggregates (e.g. median, mode) cannot be computed by dividing the input into parts.

4. Geometric aggregation in which the functions may be geometric in nature like intersection, convex hull, closest pair and so on. Hardly any work has been done on geometric aggregation [47]. In Chapter 4 we dedicate ourselves to consider these novel aggregate functions.

Next we shall briefly discuss about the various problems considered in this thesis.

1.2.3 Top-k queries

In Chapter 3 titled “Efficient top-k queries for orthogonal ranges”, we consider top-k queries which has gained a lot of significance in recent times. Advances in sensing and data gathering technologies have resulted in an explosion in the volume of data that is being generated, processed, and archived. In particular, this information overload calls for new methods for querying large spatial datasets, since users are often not interested in merely retrieving a list of all data items satisfying a query, but would, instead, like a more informative “summary” of the retrieved items. Top-k problem is one such example, where the goal is to retrieve from a set of \( n \) weighted points in \( \mathbb{R}^d \) the \( k \) most significant points, ranked by their weights, that lie in an orthogonal query box in \( \mathbb{R}^d \) (rather than get a list of all points lying in the query box). The aggregate operation is “top-k”.

Top-k problem is useful for a demographer wanting to know the \( k \)-most populated villages in a state, or an investor to find out the \( k \) best performing stocks over the past five
years that also satisfy requirements specified as a range of price, earnings, dividends, etc. Efficient and output-sensitive solutions are presented for this problem in two settings. In the first setting, the $k$ points are reported in arbitrary order and the underlying set can be updated dynamically through insertions and deletions of points. In the second setting, the $k$ points are reported in sorted order of their weights.

**Potential approaches and pitfalls.** A straightforward approach to this top-$k$ problem uses a traditional range search method to retrieve the points in $S \cap q$ and then applies a linear-time selection algorithm to $S \cap q$ to identify the $k$th largest element. A subsequent linear scan of $S \cap q$ then identifies the desired top-$k$ elements. However, this approach is expensive when $|S \cap q| \gg k$, i.e., it is not sensitive to the output size $k$. We seek an efficient, output-sensitive solution to the problem.

### 1.2.4 Geometric Aggregate queries

In Chapter 4 titled “Range-Aggregate Queries Involving Geometric Aggregation Operations” we consider some range-aggregate query problems with geometric aggregation functions. Following are some of the applications where these problems can be useful:

1. **Line-breaking:** One of the most time-consuming steps in the map overlay processing is line-breaking, which we can abstract as the pairwise segment intersection problem [47]. If an user is interested in an overlay operation in a particular window of interest, a range-aggregate segment intersection query is useful which is solved in this chapter.

2. **In a VLSI layout editing environment:** [84], geometric queries commonly arise. The layout designing process is very error prone. Almost seventy percent of the design cycle is spent in deciding and finding errors in the layout. In that context, the user often zooms to a part of the layout and is interested in queries with respect to the portion of the layout on the screen. In this chapter, we have come up with improved solutions for the 2-d range-aggregate point enclosure query and the 2-d range-aggregate interval intersection query. These are two fundamental operations which can be useful in
this context.

3. Design Rule Checking: VLSI design rules are often based on the so-called lambda (\(\lambda\)) based design rules made popular by Mead and Conway [70]. Design rule checking (DRC) is the process of checking if the layout satisfies the given set of rules. An important problem of interest to the designer is to check whether certain features are apart at least by a required separation. To check for violations in a part of the circuit, we can check if any two points in a query range violate the minimum separation rule. This can be answered using the range-aggregate closest-pair query which was attempted in [47]. Similarly, geometrical objects (often orthogonal rectangles) of two different layers in a circuit intersect to form active components. For correct working of the active component, each rectangle must extend beyond every other rectangle by some distance specified by design rules.

**Potential Approaches and Pitfalls:** At first look we can think of the following two approaches to solve a range-aggregate query problem: a) First solve the range query problem and then apply aggregate function on the appropriate subset of objects, or b) First apply the aggregate function on the whole set of objects and then return the appropriate result which is satisfied by the range query. Unfortunately, neither of these approaches lead to efficient solutions.

Consider the first problem attempted in this chapter: Preprocess a set \(S\) of points and a set \(T\) of segments on the \(x\)-axis, with \(|S| + |T| = n\) such that given a query interval \(q = [a_1, b_1]\), all pairs \((s, t), s \in S, t \in T\) satisfying \(s \in t \cap q\) can be reported efficiently.

Using approach (a) we can query set \(S\) and set \(T\) with \(q\). Let \(\alpha\) be the total number of points (from \(S\)) and segments (from \(T\)) intersecting \(q\). Using standard techniques (like a sweep line approach) we can find out all the point-segment intersections in \(O(\alpha \log \alpha + k)\) time. Therefore, the total query time would be \(O(\log n + \alpha \log \alpha + k)\), where \(k\) is the actual output size. If \(\alpha\) is large, then this solution will not be efficient (a lot of points might not be intersecting any of the segments and vice-versa). Using approach (b), in pre-processing
phase we can find out all the pair of point-segment intersections in $O(n \log n + \beta)$ time, where $\beta$ are the number of pairs of intersection. Then we can put these pairs of intersection in a binary search tree of size $O(\beta)$ and report the appropriate points for a query $q$ in $O(\log \beta + k)$ time. If $\beta$ is large, then the space occupied would be more. In this chapter we come up with linear or near-linear space structures which have low query time and are output-sensitive.

1.2.5 Colored Range-Aggregate queries

In many applications a more general form of a standard intersection problem arises: Here the objects in $S$ come aggregated in disjoint groups and of interest are questions regarding the intersection of $q$ with the groups rather than with the objects. $q$ intersects a group if and only if it intersects some object in the group. For convenience, each group is associated with a different color and all the objects in the group are imagined to have that color. Then, in the colored reporting (resp., colored counting) problem, we want to report (resp., count) the distinct colors intersected by $q$. In the dynamic setting, an object of some (possibly new) color is inserted in $S$ or an object in $S$ is deleted. Note that the colored problem reduces to the standard one when each color class has cardinality 1.

In [51], two examples of colored problems are mentioned: (Databases) Consider a database of mutual funds which contains for each fund its annual total return and its beta (a real number measuring the fund’s volatility). Thus each fund can be represented as a point in two dimensions. Moreover, the funds are aggregated into groups according to the fund family they belong to. A typical query is to determine the families that offer funds whose total return is between say, 15% and 20%, and whose beta is between, say, 0.9 and 1.1. This is an instance of the colored 2-dimensional range searching problem. The output of this query enables a potential investor to initially narrow his/her search to a few families instead of having to plow through dozens of individual funds (all from the same small set of families) that meet these criteria. (VLSI layout) In the Manhattan layout of
CHAPTER 1. INTRODUCTION

a VLSI chip, the wires (line segments) can be grouped naturally according to the circuits they belong to. A problem of interest to the designer is determining which circuits (rather than wires) became electrically connected when a new wire is added. This is an instance of the generalized orthogonal segment intersection searching problem.

In Chapter 5 and Chapter 6 titled “Range-aggregate structures for colored geometric objects-I” and “Range-aggregate structures for colored geometric objects-II”, respectively, we consider a new class of problems named “colored range-aggregate queries” which are formally defined as follows:

- We are given a set $S$ of possibly weighted objects in $\mathbb{R}^d$, $d \geq 1$ which comes aggregated in groups and we indicate the group an object belongs to by assigning each group a unique color. The goal is to preprocess $S$ such that given a query orthogonal range $q$, we can report efficiently for each distinct color $c$ of the objects in $S \cap q$, the tuple $\langle c, F(c) \rangle$ where $F(c)$ is a function of the objects of color $c$ in $q$.

Instead of simply reporting all the groups (or colors) intersecting a query $q$, a colored range-aggregate query would provide the user with more information about each group (or color). These class of queries has been well studied in the database community as “GROUP-BY” queries which are among the most important class of queries in OLAP (as remarked in [1]).

Potential Approaches and Pitfalls: For answering a colored range-aggregate query, one approach is to use the solution to the corresponding standard intersection problem. For instance, we will first find out all the objects intersected by $q$ (a standard reporting problem) and then collect objects corresponding to each color ‘c’ to compute $F(c)$. However, the query time can be very high since $q$ could intersect $\Theta(n)$ objects but only $O(1)$ distinct colors. This approach will prevent us from attaining a output-sensitive query time of the form $O(f(n) + i)$ or $O(f(n) + i \times g(n))$, where $f$ and $g$ are polylogarithmic. Output-sensitiveness can be obtained by this approach if each color class has a cardinality $O(1)$. Also, if the there are only $O(1)$ different color classes, we could simply run a standard
algorithm on each color class \( c \) in turn which would compute the corresponding function \( \mathcal{F}(c) \) and report it if any object of color \( c \) intersects \( q \). So, the real challenge is when the number of color classes and the cardinalities of these color classes are not constants, but rather are (unknown) functions of \( n \).

The other approach could be to use the solution to the corresponding colored/generalized intersection problem. We could find out all the colors intersected by \( q \) and then we could compute the aggregate function \( \mathcal{F}(c) \) for each such color \( c \). The query time turns out to be \( O(f(n) + i \times g(n)) \), where \( g(n) \) is the time taken to answer the aggregate function. Our objective in this work is to come up with data structures with query time \( O(f(n) + i) \) (without having to pay a penalty of \( g(n) \) for each reported color). If query time of the form \( O(f(n) + i) \) is not possible, then we come up with solutions with query time \( O(f'(n) + i \times g'(n)) \), where \( f'(n) \leq f(n) \) and \( g'(n) \leq g(n) \).

### 1.2.6 One-Reporting queries

Orthogonal range searching, halfplane range searching and orthogonal point-enclosure problems are one of the most fundamental problems in the field of computational geometry and hence are extensively studied. In a reporting version of these problems, all the geometric objects lying inside the query range are reported and in a counting version the number of geometric objects lying inside the query range is reported. Most of the work till now in the literature has been on reporting and counting versions. A very important version of these problems is the one-reporting or emptiness scenario:

- Let \( S \) be a set of \( n \) objects in \( \mathbb{R}^d \). We would like to preprocess \( S \) into a suitable data structure so that for any given query range \( q \subseteq \mathbb{R}^d \), we report YES iff \( S \cap q \neq \emptyset \), else we report NO. In other words, instead of reporting/counting all the objects intersecting \( q \), we need to check if there exist any object in \( S \) which intersects \( q \). The aggregation operation applied is “existence” (i.e. \( > 0 \) or \( = 0 \)).
CHAPTER 1. INTRODUCTION

The general line of attack that has been suggested in the past has been to build a data structure which does range reporting or counting and for a given query say “YES” once an object gets reported or if the number of objects intersecting the query is more than zero, respectively. If no point gets reported, say “NO”. In Chapter 7 titled “Emptiness or One-reporting queries for Standard Intersection problems” we made an attempt to come up with algorithms which exclusively answered one-reporting version of a standard intersection problem. These algorithms perform better than the existing solutions (for reporting version) in most of the cases. Unfortunately, in the worst-case scenario, our algorithms show the same performance as the existing solutions for the reporting version.

1.3 Flow of the thesis

In the next chapter we shall review the fundamental data structures which shall be used in this thesis. From Chapter 3 till Chapter 7, we shall take up a new range-aggregate problem and come up with efficient solutions for them. Finally in Chapter 8 we shall conclude our work and provide some directions for future work.
Chapter 2

Review of fundamental structures and techniques

Computational Geometry is a rich field which has evolved its own set of data structures and algorithmic techniques to solve a wide range of geometric problems. In this chapter we discuss about some fundamental data structures and techniques which shall be widely used in this thesis to obtain solutions for most of the problems. In particular we shall explain the range tree, interval tree, segment tree and priority search tree.

2.1 Interval Tree

A classical geometric data structure is called an interval tree ([36], also see e.g. [71]).

Interval trees store a set of intervals such that we can efficiently report all intervals intersecting a query point. Let $S$ be the set of intervals, let $|S| = n$, and let $X$ be the set of endpoints of the intervals in $S$. An interval tree for $S$ consists of a primary balanced binary tree $T_p$ where the leaves store the endpoints $X$, from left to right in symmetric order. We use the dynamic version of interval tree, where the points of $X$ can be arbitrary points on a line. (The static version of an interval tree is for the case where $X$ is a static set known in advance).
CHAPTER 2. REVIEW OF FUNDAMENTAL STRUCTURES AND TECHNIQUES

For each internal node \( v \in T_p \) we define the following: The split point of \( v \), denoted by \( \text{split}(v) \), is the search key of node \( v \). i.e. \( \text{split}(v) \) is a number such that the leaves of the left subtree of \( v \) store points smaller than \( \text{split}(v) \), and the leaves of the right subtree of \( v \) store points larger than \( \text{split}(v) \). The range of \( v \), denoted by \( \text{range}(v) \), is defined recursively as follows. The range of the root is \((\infty, \infty]\). For a node \( v \), where \( \text{range}(v) = (l, r] \), the range of the left child of \( v \) is \((l, \text{split}(v)]\), and the range of the right child of \( v \) is \((\text{split}(v), r]\).

We associate each interval \( s \in S \) with the unique node \( v \) of \( T_p \) such that \( s \) is contained in \( \text{range}(v) \) but not in the range of any child of \( v \). For each node \( v \in T_p \) we denote the set of all intervals associated with \( v \) by \( S(v) \). i.e. \( S(v) = \{ s \in S | \text{split}(v) \in s \subseteq \text{range}(v) \} \).

We also note that \( |S(v)| \leq n(v) \), since endpoints of intervals in \( S(v) \) must be stored at the leaves of the subtree rooted at \( v \).

For each \( v \in T_p \) we maintain two secondary balanced search trees which represent \( S(v) \). The first tree, \( T_l(v) \), stores in its leaves the left endpoints of intervals in \( S(v) \), sorted from left to right. The second tree, \( T_r(v) \) stores in its leaves the right endpoints of the intervals in \( S(v) \), sorted from right to left. It should be clear at this point that the interval tree requires \( O(n) \) space, since each interval belongs to one set \( S(v) \) and therefore its endpoints appear only in two secondary trees.

We answer a query consisting of a point \( q \) by traversing the search path for \( q \) in \( T_p \). At each node \( v \) on the search path, we search \( S(v) \) for all intervals in \( S(v) \) that contain \( q \). We perform this search as follows. If \( q \) is to the left of \( \text{split}(v) \) we traverse the tree \( T_l(v) \) from left to right and report all intervals whose left endpoint is to the left of \( q \). The case where \( q \) is to the right of \( \text{split}(v) \) is symmetric. It is straightforward to check that the query algorithm is correct, and that it takes \( O(\log n + k) \) time where \( k \) is the number of intervals reported.

To make interval trees dynamic, i.e. to support insertions and deletions of intervals we implement \( T_p \) as a \( BB(\alpha) \) tree [77]. A \( BB(\alpha) \) tree is a kind of balanced binary tree, where for each node \( v \), the ratio between \( n(\text{leftchild}(v)) \) and \( n(v) \) is in \([\alpha, 1 - \alpha] \), where \( \alpha \in (1, 1 - 1/\sqrt{2}] \). As other kinds of balanced binary trees, \( BB(\alpha) \) trees allow access
and update operations in worst-case logarithmic time. Furthermore, $BB(\alpha)$ trees have the following property.

**Theorem 2.1.1.** Let the time of a rotation at node $v$ be $n(v)f(n(v))$ for some non-decreasing function $f$. Then the amortized re-balancing time of an update (insert or delete) operation is $O(f(n(v)) \log n)$.

This property makes $BB(\alpha)$ trees especially useful for dynamization of augmented trees where each node stores a secondary data structure. The above Theorem shows that with a $BB(\alpha)$ tree as the primary structure we can amortize expensive updates to the secondary structures during rotations.

Specifically for interval trees, we insert a new interval $s$ to an interval tree by first inserting the endpoints of $s$ to $T_p$. During this insertion we may need to perform a rotation in $T_p$. When performing a rotation around an edge $(v = p(u), u)$, the sets $S(v)$ and $S(u)$ change and therefore we rebuild the secondary structures of $v$ and $u$. This rebuilding takes time proportional to $n(v)$ and therefore by the above stated theorem we obtain that each insertion takes $O(\log n)$ amortized time. After we inserted the endpoints of $s$ into $T_p$ we find the node $w$ which is the lowest common ancestor of the endpoints of $s$ and insert the left endpoint of $s$ into $T_l(w)$ and the right endpoint of $s$ into $T_r(w)$. We can implement the secondary structures by any kind of balanced search tree that supports logarithmic update time, to obtain that the total amortized time required for the insertion is $O(\log n)$. The implementation of delete is similar to the implementation of insert.

### 2.2 Segment Tree

Another data structure which is widely used is the segment tree [31]. Let $S$ be a set of intervals in real line. The intervals in $S$ partition the $x$ axis into $2|T| + 1$ elementary intervals (some of which may be empty). A segment tree $T_p$ defined on $S$ stores these elementary intervals at the leaves. Let $v \in T_p$. We associate with $v$ an interval $\text{int}(v)$
which is the union of elementary intervals stored at the leaves in \( v \)'s subtree. We say that an interval \( s \in S \) is allocated to a node \( v \in T_\mu \) iff \( \text{int}(v) \neq 0 \) and \( s \) covers \( \text{int}(v) \) but not \( \text{int}(\text{parent}(v)) \). Thus each interval \( s \in S \) is allocated to \( O(\log n) \) nodes of \( T_\mu \). Therefore, the space occupied by a segment tree will be \( O(n \log n) \).

Segment trees are most commonly used for the following query: Report all the intervals of \( S \) which are intersected by a query point \( q \). As done in the case of an interval tree, we follow the path of \( q \) in \( T_\mu \). At each node \( v \) on the path, we shall report all the intervals allocated to \( v \). The query time turns out to be \( O(\log n + k) \). It can handle updates in \( O(\log n) \) amortized time.

### 2.3 Range Trees

Range searching is a fundamental multidimensional search problem that consists of reporting the points of a set \( S \) contained in an orthogonal query range (an interval in one dimension, a rectangle with sides parallel to the coordinate axes in two dimensions, etc.). As stated in Chapter 1, orthogonal range searching is perhaps the most widely studied problem in computational geometry.

In two dimensions, range queries can be answered using a two-level tree structure called range tree \([26]\). The primary structure is a balanced search tree \( T \) whose leaves are associated with the points of \( S \), sorted by \( x \)-coordinate. For each internal node \( \mu \), let \( S(\mu) \) be the set of points associated with the leaves of \( T \) in the subtree of \( \mu \), called the proper points of \( \mu \). The secondary structure of \( \mu \) is a one-dimensional data structure for range searching by \( y \)-coordinate in the set \( S(\mu) \). Every point \( p \) of \( S \) is stored in the secondary structures of the \( O(\log n) \) nodes on the path from the root to the leaf associated with \( p \). Hence, the space requirement is \( O(n \log n) \). By presorting the points both by \( x \)- and \( y \)-coordinate, \( O(n \log n) \) preprocessing time can be achieved \([9]\).

Let \( q = [a_1, b_1] \times [a_2, b_2] \) be the query range. A node of \( T \) is an allocation node for \([a_1, b_1]\) if the vertical strip \([a_1, b_1]\) contains \( S(\mu) \) but not \( S(v) \), where \( v \) is the parent of \( \mu \).
CHAPTER 2. REVIEW OF FUNDAMENTAL STRUCTURES AND TECHNIQUES

Hence, \([a_1, b_1]\) has \(O(\log n)\) allocation nodes, the sets of proper points of the allocation nodes are disjoint, and their union consists of the points of \(S\) in the strip \([a_1, b_1]\). The points of \(S\) inside \(q\) can be found by performing one-dimensional range queries for the range \([a_2, b_2]\) in the secondary structures of the allocation nodes of \([a_1, b_1]\). The total time complexity is \(O(\log^2 n + k)\).

When adding a point \(p\) to \(S\), we add a new leaf to \(T\), and then insert \(p\) into the secondary structures of the nodes from the new leaf to the root. This takes \(O(\log^2 n)\) time. Next, we re-balance \(T\) by means of rotations. When performing a rotation at a node \(\mu\), we rebuild the secondary structures of the nodes involved in the rotation, which takes time proportional to the number of leaves in the subtree of \(\mu\). Hence, by using a weight-balanced tree for \(T\), the amortized re-balancing time is \(O(\log n)\), and thus the amortized insertion time is \(O(\log^2 n)\). Analogous considerations hold for deletions.

The query and update times can be reduced using fractional cascading. Here, each node \(\mu\) of \(T\) stores catalog \(P(\mu)\), and query subgraphs are root-to-leaf paths. In the static case we obtain \(O(\log n + k)\) query time. In the dynamic case we have \(O(\log n \log \log n + k)\) query time and \(O(\log n \log \log n)\) update time [72].

This technique readily extends to \(d\)-dimensional space \(\mathbb{R}^d\), yielding a data structure with \(O(n \log^{d-1} n)\) space requirement, \(O(\log^{d-1} n \log n)\) update time, and \(O(\log^{d-1} n \log \log n + k)\) query time \((O(\log^{d-1} n + k)\) in the static case).

2.4 Priority Search Tree

The priority search tree of McCreight [69] efficiently supports a restricted type of range queries, where the query rectangle has at least one side at infinity. It is a hybrid of a heap (for the \(y\)-coordinates) and of a balanced search tree (for the \(x\)-coordinates).

A static priority search tree \(T\) for a set \(S\) of \(n\) points in the plane is built as follows. The root \(r\) of \(T\) stores the highest point of \(S\), denoted \(p(r)\), and the median \(x\)-coordinate of the remaining points, denoted \(x(r)\). The left subtree is a priority search tree for the points of \(S\)
CHAPTER 2. REVIEW OF FUNDAMENTAL STRUCTURES AND TECHNIQUES

\(- p(r)\) to the left of the vertical line \(x = x(r)\), and the right subtree is a priority search tree for the remaining points.

First, we show how to perform a range query for a query rectangle \(q = (-\infty, b_1] \times [a_2, \infty)\) unbounded on top and to the left. If \(p(r)\) lies in \(q\), then report it, else if \(p(r)\) is below \(a_2\), then stop. If \(b_1 < x(r)\), then recursively call the procedure on the left subtree, since all the points in the right subtree are outside \(q\). Else \((b_1 \ge x(r))\), recursively call the procedure on the left and right subtrees. (Note that the recursive calls in the left subtree scan the subtree from top to bottom and terminate whenever a point below \(a_2\) is reached.) The entire query takes \(O(\log n + k)\) time, using \(O(n)\) space, where \(k\) is the number of points reported.

This procedure can be generalized to handle ranges with three sides (unbounded on top) by traversing two paths instead of one, with the time bound remaining the same. Note that the heap structure of the priority search tree is based on the top-down order of the points, so that it supports queries in rectangles unbounded on top; if the query rectangles are unbounded on bottom, we build the tree using the bottom-up order. The other two cases (unbounded to the left or right) are analogous.

The priority search tree discussed so far is static; to dynamize it, we replace the fixed tree structure with a red-black tree \(T\), whose leaves store the points of \(S\) sorted by \(x\)-coordinate. Each internal node also stores the highest point among the leaves below it that is not stored in any ancestor of that node (heap property). Note that some interior nodes may not store any point. Since the red-black tree has height \(O(\log n)\), the query time is \(O(\log n + k)\) with space complexity \(O(n)\). To insert a point, we add a new leaf, restructure the path from the leaf to the root to maintain the heap property, and perform the necessary rotations. A rotation at a node requires updating the path from to the root. Since a red-black tree can be rebalanced with only \(O(1)\) rotations, insertion of a point takes \(O(\log n)\) time. To delete a point, we remove from the tree the nodes (a leaf and at most one internal node) storing the point. Hence, deletion of a point takes \(O(\log n)\) time.
Chapter 3

Efficient top-$k$ queries for orthogonal ranges

In recent years, advances in computing, communication, and sensing technologies have led to an explosion in the quantity of data that is generated on a daily basis. (For instance, according to a recent study (http://tinyurl.com/2frbdeu), the total volume of electronic data stored worldwide reached 1.2 billion terabytes (a growth rate of 62% from 2009) and is projected to reach 35 billion terabytes by the year 2020.) This information overload calls for new methods to query large datasets since users are often interested not in merely retrieving a list of all the data items that satisfy a query but would instead like a more informative and manageable “summary” of the query results. Generating such a summary requires applying some kind of aggregation function to the query results and doing so efficiently. One of the simplest and most useful aggregation functions is reporting the $k$ best items among the ones that satisfy the query, where the notion of “best” is based on a weight associated with each data item. This has led to the formulation of the so-called top-$k$ problem, where the goal is to retrieve from a set, $S$, of $n$ weighted points in $\mathbb{R}^d$ the $k$ most significant points, ranked by their weights, that lie in a user-specified query range $q$, for instance an orthogonal box in $\mathbb{R}^d$. The top-$k$ problem has been studied intensively in the context of information retrieval [16], multimedia similarity search [19], text and
data integration [64], business analytics [6], preference queries over product catalogs and Internet-based recommendation sources [68], distributed aggregation of network logs and sensor data and so on. In these domains, the end-users are most interested in the important (i.e. top-k) query answers in the potentially huge answer space. The top-k problem is a non-trivial generalization of the well-studied range search problem in computational geometry, where the goal is to preprocess $S$ so that the points that lie in $q$ can be reported or counted efficiently. (See [5] for a survey of geometric range search.)

In this chapter, we present efficient geometric algorithms and data structures for two versions of the top-k problem. In the first version, the top $k$ points can be reported in any order and the underlying set $S$ can be updated through insertion and deletion of points. In the second case, the top $k$ points are required to be reported in sorted order by weight. Besides being efficient in the input size, $n$, our solutions are also efficient in terms of the output size, $k$, i.e., the query time is a function of $k$ rather than the actual number of items satisfying the query, which can be much larger. In other words, our solutions are output-sensitive.

Before proceeding to the formal problem statement and solution technique, we begin with two motivating examples for the top-k problem and also discuss briefly prior related work.

- Modern GIS systems allow users to query the underlying spatial dataset for useful information. For instance, consider a GIS for rural India which contains location and population information for the villages in a large state. For census or resource allocation purposes, a demographer might wish to identify the $k$ most populated villages in a rectangular query region of interest. If we model each village as a point in the $(x, y)$-plane and associate with it an integer weight equal to its population, then the problem can be framed as an orthogonal top-k range query.

- Consider a financial database of stocks. Each stock is represented by a point in $\mathbb{R}^d$, where each of the $d$ dimensions encodes an attribute of the stock (e.g., price, earn-
ings, dividends, trading volume, market capitalization, etc.) Also associated with the stock is a weight equal to (say) its 5-year performance. An investor might be interested in identifying the ten best-performing stocks over the past five years that also satisfy requirements specified as a range of price, earnings, dividends, etc. This can be accomplished by performing an orthogonal range query to determine all stocks satisfying the $d$ ranges specified by the query and then sifting through this potentially large output for the ten best stocks. However, a more informative and efficient approach is to issue a top-$k$ ($k = 10$) query on the database to get exactly the desired stocks.

The top-$k$ version of range searching falls under a class of problems called range-aggregate query problems [86] in which many composite queries involving range searching are considered, wherein one needs to compute the aggregate function of the objects in $S \cap q$ rather than report all of them as in a range reporting query. In many applications like online analytical processing (OLAP), geographic information systems (GIS) and information retrieval (IR), aggregation plays an important role in summarizing information [86] and hence a large number of algorithms and storage schemes have been proposed to support such queries.

### 3.0.1 Related Work

In the field of databases, the systematic study of top-$k$ query processing was started and currently an active area of research. [56] is a excellent survey paper on the recent results of top-$k$ query processing techniques in relational database systems. Based on [56], we shall provide a *summarized* overview of the different directions of research initiated w.r.t. top-$k$ query processing techniques. A taxonomy to classify top-$k$ processing techniques based on their capabilities and assumptions are as follows:

1. *Query Model:* Top-$k$ processing techniques adopt different query models to specify the data objects to be scored. Three major models include:
• **Top-k selection query.** In this model, the scores are assumed to be attached to base tuples. A top-k selection query is required to report the $k$ tuples with the highest scores. Scores might not be readily available since they could be the outcome of some user-defined scoring function that aggregates information coming from different tuple attributes. The NRA algorithm [81] is one example of top-k techniques that adopt the top-k selection model. The input to the NRA algorithm is a set of sorted lists; each ranks the “same” set of objects based on different attributes. The output is a ranked list of these objects ordered on the aggregate input scores.

• **Top-k join query.** In this model, scores are assumed to be attached to join results rather than base tuples. A top-k join query joins a set of relations based on some arbitrary join condition, assigns scores to join results based on some scoring function, and reports the top-k join results. Many top-k join techniques address the interaction between computing the join results and producing the top-k answers. Examples are the J* algorithm [75], and the Rank-Join algorithm [55]. Some techniques, for example, PREFER [54], process top-k join queries using auxiliary structures that materialize join results, or by ranking the join results after they are generated.

• **Top-k aggregate query.** In this model, scores are computed for tuple groups, rather than individual tuples. A top-k aggregate query reports the $k$ groups with the largest scores. Group scores are computed using a group aggregate function such as $sum$. Top-k aggregate queries add additional challenges to top-k join queries: (1) interaction of grouping, joining, and scoring of query results, and (2) nontrivial estimation of the scores of candidate top-k groups. A few recent techniques, for example, [66], address these challenges to efficiently compute top-k aggregate queries.

2. **Data Access.** Based on the assumptions they make about available data access meth-
CHAPTER 3. EFFICIENT TOP-\( k \) QUERIES FOR ORTHOGONAL RANGES

ods in the underlying data sources, the top-\( k \) processing techniques are classified as follows:

- **Both sorted and random access.** In this category, top-\( k \) processing techniques assume the availability of both sorted and random access in all the underlying data sources. Examples are TA [81], and the Quick-Combine algorithm [45].

- **No random access.** In this category, top-\( k \) processing techniques assume the underlying sources provide only sorted access to data objects based on their scores. Examples are the NRA algorithm [81], and the Stream-Combine algorithm [46].

- **Sorted access with controlled random probes.** In this category, top-\( k \) processing techniques assume the availability of at least one sorted access source. Random accesses are used in a controlled manner to reveal the overall scores of candidate answers. Examples are the Rank-Join algorithm [55], the MPro algorithm [17], and the Upper and Pick algorithms [15].

3. **Implementation level.** We classify top-\( k \) processing techniques based on their level of integration with database engines as follows:

- **Application level.** This category includes top-\( k \) processing techniques that work outside the database engine. Some of the techniques in this category rely on the support of specialized top-\( k \) indexes or materialized views. However, the main top-\( k \) processing remains outside the engine. Examples are [18], and [54]. Another group of techniques formulate top-\( k \) queries as range queries that are repeatedly executed until the top-\( k \) objects are obtained. We refer to this group of techniques as filter-restart. One example is [33].

- **Query engine level.** This category includes techniques that involve modifications to the query engine to allow for rank-aware processing and optimization. Some of these techniques introduce new query operators to support efficient
CHAPTER 3. EFFICIENT TOP-K QUERIES FOR ORTHOGONAL RANGES

top-k processing. For example, [55] introduced rank-aware join operators. Other techniques, for example, [67, 66], extend rank-awareness to query algebra to allow for extensive query optimization.

4. *Query and data uncertainty.* We classify top-k processing techniques based on query and data certainty as follows:

- **Exact methods over certain data.** This category includes the majority of current top-k processing techniques, where deterministic top-k queries are processed over deterministic data.

- **Approximate methods over certain data.** This category includes top-k processing techniques that operate on deterministic data, but report approximate answers in favor of performance. The approximate answers are usually associated with probabilistic guarantees indicating how far they are from the exact answer. Examples include [88] and [8].

- **Uncertain data.** This category includes top-k processing techniques that work on probabilistic data. The research proposals in this category formulate top-k queries based on different uncertainty models. Some approaches treat probabilities as the only scoring dimension, where a top-k query is a Boolean query that reports the k most probable query answers. Other approaches study the interplay between the scoring and probability dimensions. An example is [82].

5. *Ranking function* We classify top-k processing techniques based on the restrictions they impose on the underlying ranking function as follows:

- **Monotone ranking function.** Most of the current top-k processing techniques assume monotone ranking functions since they fit in many practical scenarios, and have appealing properties allowing for efficient top-k processing. One example is [81].
CHAPTER 3. EFFICIENT TOP-K QUERIES FOR ORTHOGONAL RANGES

• Generic ranking function. A few recent techniques, for example, [91], address top-\(k\) queries in the context of constrained function optimization. The ranking function in this case is allowed to take a generic form.

• No ranking function. In some applications we might be interested in presenting the results of a query to many users. In such a scenario, we would like to present all the important and relevant tuples instead of giving specific weightage to each attribute (which will change with the preference of each user). Many techniques have been proposed to answer skyline-related queries which provide a good list of all the important tuples, for example, [12] and [90].

Though the work on top-\(k\) has been an actively pursued by the researchers in the database community, there has been relatively very little work done on these problems in the field of computational geometry. The assumption is that in the pre-processing stage each point has a weight associated with it and is not determined explicitly after the query is given. In the past, the following problem (top-1) was considered in Chazelle [22]: Preprocess a set \(S\) of weighted points in \(\mathbb{R}^d\) so that given a query orthogonal box \(q\), report the point of \(S\) in \(q\) with the maximum weight. On a pointer machine model he obtained a linear space solution which answered queries in \(O(\log^3 n)\) time, for \(d=2\). In the dynamic setting, the query time increased to \(O(\log^3 n \log \log n)\) and the update time was \(O(\log^3 n \log \log n)\).

In Agarwal et al. [3], this problem is solved on an external memory model. For \(d=2\), they obtain a static solution which takes up linear space and answers queries in \(O(\log_B^2 n)\) time, where \(n = \lceil N/B \rceil\). In the dynamic setting the query time increases to \(O(\log_B^3 n)\) and the update time is \(O(\log_B^2 n)\). The problem being attempted in this paper is a generalized version of this problem wherein we want to report the \(k\) most weighted points. In Gabow et al. [41] they solve the range minimum query by transforming it into a lowest common ancestor problem using the technique of Cartesian trees.

Closely related to the problem attempted in this chapter are the following works. Recently Brodal et al. [14] studied the following problem: Preprocess a one-dimensional
array $A$ having $n$ elements, so that given two indices $i$ and $j$ and an integer $k$, report the $k$ smallest elements in the subarray $A[i...j]$ in sorted order. They assume a RAM model and obtain linear space solution and answer queries in $O(k)$ time. A variation of this problem was attempted in [43], where the objective is to report the $k^{th}$ smallest element (called range quantile queries). They construct a data structure of size $O(n \log \sigma)$ bits and answer range quantile queries in $O(\log \sigma)$ time, where $\sigma$ is the number of distinct elements. In [29], they assess the practical performance of the above data structure. They also describe algorithms for listing the top-$k$ items in the query range. Ref. [42] is a good survey paper on these problems.

### 3.0.2 Formal Problem Statement

Formally, the problem we consider is the following.

- Let $S$ be a set of $n$ weighted points in $\mathbb{R}^d$, where each point $p$ has a real-valued weight $w(p)$. We would like to preprocess $S$ into a suitable data structure so that for any given orthogonal query box $q = \prod_{i=1}^{d} [a_i, b_i]$ and an integer $k \in [1, n]$, the top-$k$ points in $S \cap q$, ranked by their weights can be reported efficiently and output-sensitively.

We note that the top-$k$ problem is decomposable: Given two disjoint point sets $A$ and $B$, the solution to the top-$k$ problem on $A \cup B$ can be obtained from the solutions to the top-$k$ problems on $A$ and $B$ respectively in $O(k)$ time by using the linear-time selection algorithm [28].

A straightforward approach to the top-$k$ problem uses a traditional range search method to retrieve the points in $S \cap q$ and then applies a linear-time selection algorithm to $S \cap q$ to identify the $k$th largest element. A subsequent linear scan of $S \cap q$ then identifies the desired top-$k$ elements. However, this approach is expensive when $|S \cap q| \gg k$, i.e., it is not sensitive to the output size $k$. We seek an efficient, output-sensitive solution to the problem.

In the following sections, we describe such solutions for this problem in two settings. In the first one (Section 3.1) the top-$k$ points may be output in arbitrary order and another
CHAPTER 3. EFFICIENT TOP-K QUERIES FOR ORTHOGONAL RANGES

### Table 3.1

<table>
<thead>
<tr>
<th>Reporting points in arbitrary order</th>
<th>Underlying Space</th>
<th>Space occupied</th>
<th>Query time</th>
<th>Update time (amortized)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbb{R}^d$ ($d \geq 1$)</td>
<td>$O(n \log^d n)$</td>
<td>$O(\log^d n \log \log n + k)$</td>
<td>$O(\log^d n \log \log n)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$O(n \log^d n)$</td>
<td>$O(\log^d n + k)$</td>
<td>$\quad$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reporting points in sorted order</th>
<th>Underlying Space</th>
<th>Space occupied</th>
<th>Query time</th>
<th>Update time (amortized)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbb{R}^1$</td>
<td>$O(n \log n)$</td>
<td>$O(\log n + k)$</td>
<td>$\quad$</td>
<td>$\quad$</td>
</tr>
<tr>
<td>$\mathbb{R}^d$ ($d \geq 2$)</td>
<td>$O(n \log^d n)$</td>
<td>$O(\log^{d-1} n + k \log \log n)$</td>
<td>$\quad$</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1 Summary of results obtained for top-$k$ queries for orthogonal ranges. $k$ is also part of the query. Two settings are considered. The first one reports the top-$k$ points in arbitrary order and the other in sorted order by weight. All bounds are worst-case results.

(Section 3.2) where they are requested in sorted order by weight. (The first solution also supports updates in the underlying set $S$). The results obtained have been summarized in Table 3.1.

### 3.1 Dynamic structure for reporting top-k points in arbitrary order

In this section we report the top-$k$ points in an arbitrary order for a set of $n$ weighted points in $\mathbb{R}^d$. This problem has not been tackled previously in the literature. We shall first present some approaches which one might be tempted to use to answer this query (assume a two-dimensional space for simplicity of presentation):

- Scan the whole database to find out all the points which lie inside the query rectangle.

Call it set $S'$. Find out the $k^{th}$ largest point $p_k$ in $S'$ where the ordering is done based
CHAPTER 3. EFFICIENT TOP-K QUERIES FOR ORTHOGONAL RANGES

on the weights of the points. Let the weight of \( p_k \) be \( w(p_k) \). All the points in \( S' \) whose weight is greater than or equal to \( w(p_k) \) are the top-\( k \) points. Scanning the whole database takes \( O(n) \) time. Finding the \( k^{th} \) largest point and reporting the top-\( k \) points in \( S' \) takes \( O(|S'|) \) time. The overall time taken is \( O(n) \) which is wasteful since there is no pre-processing being done.

• The previous technique did not do any pre-processing which lead to a high query time. Suppose we put the points of \( S \) in a two-dimensional range tree \( RT \) based on the \( x \) and \( y \) coordinates. This is done in the pre-processing stage. Querying \( RT \) with a query rectangle \( q \) reports all the points of \( S \) which lie inside \( q \). Then applies a linear-time selection algorithm to \( S \cap q \) to identify the \( k \)th largest element. A subsequent linear scan of \( S \cap q \) then identifies the desired top-\( k \) elements. The query time turns out to be \( O(\log n + |S \cap q|) \). However, this approach is expensive when \( |S \cap q| \gg k \), i.e., it is not sensitive to the output size \( k \).

Our objective is to obtain an efficient, output-sensitive solution to the problem. Since the existing techniques in the literature do not help us in obtaining these objectives, we shall present a novel solution to this problem below.

In the preprocessing phase, all the points of \( S \) are sorted based on their weights in non-increasing order. Let the sequence of points obtained be \( w(p_1), w(p_2), \ldots, w(p_n) \). These points are made the leaves of a \( BB(\alpha) \) tree \( T \) [78] from left to right. At each internal node \( v \) of \( T \), we build a data structure \( M_v^C \) on the set of points stored in the subtree rooted at \( v \). This data structure supports \( d \)-dimensional orthogonal range counting queries and allows updates (insertion/deletion of points). A dynamic \( d \)-dimensional range tree can be used for supporting these operations. It uses \( O(n \log^{d-1} n) \) space and handles updates in \( O(\log^{d-1} n \log \log n) \) time. It can count points lying inside an orthogonal query box in \( O(\log^{d-1} n \log \log n) \) time [72].

Next we transform each point \( p(p_1, p_2, \ldots, p_d) \in S \) having weight \( w(p) \) into a \( (d + 1) \)-dimensional point \( p'(p_1, p_2, \ldots, p_d, w(p)) \). Call this new set of transformed points \( S' \). We
build a data structure $M_R$ on $S'$, so that given a query of the form $q = [a_1, b_1] \times \ldots \times [a_d, b_d] \times [w_q, \infty)$, all the points in $S'$ lying within $q$ can get reported efficiently. This structure also supports updates (insertion/deletion of points). A dynamic $(d + 1)$-dimensional range tree [72] can be used where the first $d$ level trees are built using the $d$ coordinate values of each point and dynamic fractional cascading is applied at the innermost level involving the weights of the points.

In the query, we are given an orthogonal $d$-dimensional box and an integer $k$. The query algorithm consists of the following steps: (a) Using $T$ determine the number of points of $S$ lying within $q$. If $|S \cap q| \leq k$, then report all the points in $S \cap q$; (b) If $|S \cap q| > k$, search for a threshold point $p \in S$ in $T$, such that, the number of points of $S'$ in the region $q' = \Pi_{i=1}^d [a_i, b_i] \times [w(p), \infty)$ is exactly $k$; (c) Using $M_R$, report all the $k$ points lying in the region $q'$. Steps (a) and (c) are fairly straightforward. The challenge is to find the threshold point in step (b).

The query algorithm is explained in the form of a pseudo-code in Algorithm 1. The query algorithm takes as input a node $v \in T$, integer $k'$, and the given query box $q$. (Initially, $v$ is the root $r$ of $T$ and $k' = k$). At the root node, we first query $M_R$ with $q$ (line 2) to find out the number of points of $S$ lying within $q$. If $|S \cap q| \leq k$, then all the points lying within $q$ are reported. Line 4–6 and 14–16 handle this case (step a).

If $|S \cap q| > k$, then the objective of the algorithm is to find the threshold point. We go to the left child (i.e. $lc(r)$) of $r$ in search of our threshold point (lines 11–12). Let $S_v$ denote the set of points lying in the subtree rooted at an internal node $v$. At node $lc(r)$, if $count > k$, i.e., $|S_{lc(r)} \cap q| > k$, which means that the threshold point lies within the subtree of $lc(r)$. So, in our search we will next visit the left child of $lc(r)$ and query it as $Query(lc(lc(r)), k, q)$ (again lines 11–12). However, if at node $lc(r)$, $count < k$, it means that the threshold point lies in the subtree rooted at the sibling of $lc(r)$, i.e., $sib(lc(r))$. Then we query the sibling node of $lc(r)$ as $Query(sib(lc(r)), k - count, q)$ (line 8). Note carefully that the second argument is $k - count$. 

30
CHAPTER 3. EFFICIENT TOP-K QUERIES FOR ORTHOGONAL RANGES

Algorithm 1 Query (Node $v$, Integer $k'$, Query Box $q$)

1: int $count = \mathcal{M}_C^v(q)$;

3: 

4: if $count < k'$ then
5: if $v$ is the root of $T$ then
6: $\mathcal{M}_R(q \times (-\infty, \infty))$;
7: else
8: Query($sib(v)$, $k' - count$, $q$); \{sib($v$) is the sibling of the node $v$\}
9: end if

10: else if $count > k'$ then
12: Query($lc(v)$, $k'$, $q$); \{lc($v$) is the left child of the node $v$\}
13: 
14: else if $count == k'$ then
15: $\mathcal{M}_R(q \times [w(p_r), \infty))$; \{$p_r$ is the rightmost point in the subtree rooted at $v$\}
16: STOP the execution of the query algorithm.
17: 
18: end if
CHAPTER 3. EFFICIENT TOP-K QUERIES FOR ORTHOGONAL RANGES

Similarly, at a particular internal node \( v \in T \), if the threshold point is found to lie inside the subtree of \( v \), then we descend to the left child of \( v \); else we move to the sibling of node \( v \). Finally at a node \( v' \), the else condition in line 14 gets satisfied. \( p_r \) is defined as the rightmost point in the subtree rooted at \( v' \). This assures that 
\[
k = |W' \cap q| \text{ where } W' = p_1, p_2, \ldots, p_r.
\]
Now we bring in our range reporting data structure \( \mathcal{M}_R \) and query it with \( \Pi_{i=1}^{d} [a_i, b_i] \times [w(p_r), \infty) \). The top-\( k \) points of \( S \) lying inside \( q \) get reported.

The query algorithm is illustrated via an example shown in Figure 3.1. In 3.1(a), eight points lying in a two-dimensional plane are shown along with their weights. The query rectangle \( q \) is also shown in it. Points having weight 20, 40, 50, 60, 80 lie inside \( q \). Figure 3.1(b) and 3.1(c) depict the flow of the query algorithm within the data structure \( T \) for \( k=2 \) and \( k=4 \), respectively. The nodes shown in black are the nodes visited during the execution of the query algorithm. With each black node \( v \) we have associated a label of the form \( v(count, k') \), where \( count \) denotes the number of points in the subtree of node \( v \) which
lie inside $q$ and $k'$ is the parameter passed while visiting node $v$. The value of count is determined by the orthogonal range counting structure stored at $v$. For $k=2$, we need to report points with weight 80 and 60. In Figure 3.1(b), at the root we have the label $r(5, 2)$ which implies that a threshold point exists. Then we visit its left child $a$. Since it has the label $a(3, 2)$, it means that the threshold point lies in the subtree of $a$ and hence we visit node $b$. However, only one point in the subtree of $b$ lies inside $q$ (point with weight 80) which means that the threshold point will lie in its sibling node (i.e. node $c$). From $c$ we go its left child $d$ where both the values match. Then we query our reporting structure with $q \times [60, \infty)$ to report points with weight 80 and 60.

For $k=4$ (Figure 3.1(c)), we need to report points with weights 80, 60, 50 and 40. The subtree under $a$ contains three of the required points (namely 80, 60 and 50). Now we are in search of one more point under the subtree of $e$. At node $f$ we notice that both the values match ($f(1, 1)$ as point with weight 40 lies inside $q$). Again we query our reporting structure with $q \times [30, \infty)$, since 30 is the rightmost point in the subtree of $f$. The appropriate points get reported. An important lemma is stated next.

**Lemma 3.1.1.** For a given query $q$ and an integer $k$, the number of nodes visited by the query algorithm in the primary structure of $T$ is $O(\log n)$.

**Proof.** The height of a $BB(\alpha)$ tree built on $n$ points is $O(\log n)$. Assume that we are at an internal node $v \in T$. If $v$ is a left child of its parent node (i.e. $p(v)$), then if there is a next step, we might have to go either to its right child (i.e. $\text{sib}(v)$) or to its left child (i.e $\text{lc}(v)$). However, if $v$ is a right child of its parent node (i.e. $p(v)$), then if there is a next step then we will always go its left child (i.e $\text{lc}(v)$). The reason is the following: We reached node $v$ because at node $p(v)$ it must have been observed that the threshold point lies in the subtree rooted at $p(v)$ but not in the subtree rooted at node $\text{lc}(p(v))$ (i.e left sibling of $v$). Therefore, the threshold point has to lie inside the subtree rooted at node $v$ and hence we visit $\text{lc}(v)$. So, after every two steps we go down at least one level in $T$. Hence, the number of nodes visited in the primary structure of $T$ is bounded by $O(\log n)$. 

33
CHAPTER 3. EFFICIENT TOP-K QUERIES FOR ORTHOGONAL RANGES

3.1.1 Handling insertions and deletions

As noted earlier, the \(d\)-dimensional range reporting/counting structures can handle insertion and deletion of points of \(S\). Suppose that we want to insert/delete a point in \(S\). We first insert/delete \(p\) in \(M_{R}\). Then we search down \(T\) and insert/delete a leaf and then update the secondary structure (i.e. \(M_{vC}\)) at each node on the search path. Then \(T\) might have to be rebalanced via rotations. The rotations will change the set of descendant leaves of certain nodes and thus render obsolete their secondary structures (\(M_{vC}\)). Then the secondary structures will have to be rebuilt. As shown in Willard et al. [89], the amortized update time for \(T\) will be \(O(U(n) \log n)\), where \(U(m)\) is the amortized update time for \(M_{vC}\) built on \(m\) points. The overall update time will be \(O(U(n) \log n + U'(n))\), where \(U'(n)\) is the amortized update time for \(M_{R}\).

**Theorem 3.1.1.** Suppose that we have a data structure \(M_{C}\) which supports \(d\)-dimensional range counting queries. Let the performance of \(M_{C}\) be represented by the tuple \(\langle S_{c}(n), Q_{c}(n), I_{c}(n), D_{c}(n) \rangle\), where \(S_{c}(n)\) is the space occupied, \(Q_{c}(n)\) is the time taken to answer a counting query, \(I_{c}(n)\) is the amortized time taken to insert a new point and \(D_{c}(n)\) is the amortized time taken to delete a point. Similarly, suppose that we have a data structure \(M_{R}\) which supports orthogonal range reporting for queries of the form \(\Pi_{i=1}^{d}[a_{i}, b_{i}] \times [a_{d+1}, \infty)\).

Let its performance be represented by the tuple \(\langle S_{r}(n), Q_{r}(n), I_{r}(n), D_{r}(n) \rangle\).

Then, a set \(S\) of \(n\) weighted points in \(\mathbb{R}^{d}\) can be preprocessed into a data structure, so that given an orthogonal \(d\)-dimensional box and an integer \(k \in [1, n]\), the top-\(k\) points in \(S \cap q\), ranked by their weights can be reported efficiently, with the following bounds:

- **The space occupied by the data structure** is \(O(S_{c}(n) \log n + S_{r}(n))\).
- **The time taken to answer a query** is \(O(Q_{c}(n) \log n + Q_{r}(n) + k)\).
- **The amortized time taken to insert a new point** is \(O(I_{c}(n) \log n + I_{r}(n))\).
- **The amortized time taken to delete a new point** in \(O(D_{c}(n) \log n + D_{r}(n))\).
CHAPTER 3. EFFICIENT TOP-K QUERIES FOR ORTHOGONAL RANGES

In a dynamic setting, a dynamic range tree when built on \( n \) \( d \)-dimensional points uses \( O(n \log^{d-1} n) \) space, handles updates in \( O(\log^{d-1} n \log \log n) \) time. It can report and count points lying inside an orthogonal query box in
\( O(\log^{d-1} n \log \log n + k) \) and \( O(\log^{d-1} n \log \log n) \) time, respectively [72]. Substituting these values in Theorem 3.1.1, we obtain the following theorem.

**Theorem 3.1.2.** A set \( S \) of \( n \) weighted points in \( \mathbb{R}^d \) \((d \geq 1)\) can be preprocessed into a data structure of size \( O(n \log^d n) \) so that given a query orthogonal box \( q \) and an integer \( k \in [1, n] \), the top-\( k \) points in \( q \) ranked by their weights can be reported in \( O(\log^d n \log \log n + k) \) time. Additionally points can be inserted and deleted in \( O(\log^d n \log \log n) \) amortized time. The model of computation is assumed to be a pointer machine model.

If we do not want any updates to happen on our underlying set \( S \), then we can seek to improve the query time. The dynamic range tree is replaced by a static range tree. A static range tree when built on \( n \) \( d \)-dimensional points uses \( O(n \log^{d-1} n) \) space. It can report and count points lying inside the query box in \( O(\log^{d-1} n + k) \) and \( O(\log^{d-1} n) \) time, respectively [5]. Substituting these values in Theorem 3.1.1, we obtain the following theorem.

**Theorem 3.1.3.** A set \( S \) of \( n \) weighted points in \( \mathbb{R}^d \) \((d \geq 1)\) can be preprocessed into a data structure of size \( O(n \log^d n) \) so that given a query orthogonal box \( q \) and an integer \( k \in [1, n] \), the top-\( k \) points in \( q \) ranked by their weights can be reported in \( O(\log^d n + k) \) query time. The model of computation is assumed to be a pointer machine model.

### 3.2 Reporting top-\( k \) points in sorted order

In this section we propose a static solution that reports the top-\( k \) points in sorted order of their weights. First we present a solution for this problem in one-dimensional space. Then we present a separate solution for \( d \)-dimensional space, where \( d \geq 2 \). One might be tempted to use the solution described in the previous section (for reporting arbitrary
points) to answer this query. Specifically, it can be done as follows: Use the data structure described in the previous section to report the top-$k$ points and then sort them based on their weights. The query time turns out to be $O(\log^d n + k \log k)$. The solution is output-sensitive but when $k$ is very large, i.e., $k \equiv O(n)$, then the query time would be $O(\log^d n + k \log n)$. It means that we will have to pay a penalty of $O(\log n)$ to report each point. The objective in this section is to come up with a novel solution which can reduce the penalty from $O(\log n)$ to $O(\log \log n)$ for $d \geq 2$. For $d = 1$ we were successful in reducing the penalty per reported point from $O(\log n)$ to $O(1)$.

3.2.1 Solution for IR$^1$

First we consider the case wherein the queries are of the form $q = [a_1, \infty)$. For a fixed $a_1$, we can store all the points of $S$ that lie in the interval $[a_1, \infty)$ in a list $L(a_1)$ in the non-increasing order of weights. Given $k$, we can retrieve the top-$k$ points in $[a_1, \infty)$, simply by walking down $L(a_1)$ and reporting the top-$k$ points in $\Theta(k)$ time. We build such a list for each $a_1 \in S$.

We sort the points of $S$ in non-decreasing order of their $x$ coordinates (ties broken arbitrarily) as $P_1, P_2, \ldots, P_n$ and store them along with the points $P_0 = -\infty$ and $P_{n+1} = \infty$ in an array $A$. This defines $n + 1$ intervals. Let $I_j$ denote the interval $(P_j, P_{j+1}]$. For any $a_1 \in I_j$, the top-$k$ points in $[a_1, \infty)$ are the same. With $I_j$ we can store the list $L(P_{j+1})$. Given a query $q$, we search in $A$ to locate the interval $I_j$ containing $q$. Then we query $L(P_{j+1})$ with $q$ and report the top-$k$ points. The overall space requirement is $O(n^2)$ and the query time is $O(\log n + k)$.

The space requirement can be reduced to $O(n)$ by observing that $L(P_j)$ and $L(P_{j+1})$ are only slightly different. $L(P_{j+1}) = L(P_j) \setminus P_j$. Hence $L(P_{j+1})$ can be obtained from $L(P_j)$ with $O(1)$ memory modifications. Treating the $x$-coordinate as time, we store all the lists in a partially persistent structure [34]. We start with $L_0$ containing the points in $S$ in non-increasing order of their weights. Then we obtain $L(P_{j+1})$ from $L(P_j)$ for $j = 0,$
CHAPTER 3. EFFICIENT TOP-K QUERIES FOR ORTHOGONAL RANGES

1, ..., n. These operations require $O(n)$ memory modifications overall and hence the data structure occupies $O(n)$ space.

**Lemma 3.2.1.** A set, $S$, of $n$ weighted points in $\mathbb{R}^1$ can be preprocessed into a data structure of size $O(n)$ so that for any query range $q = [a_1, \infty)$, and a given integer $k$ satisfying $1 \leq k \leq n$, the points in $S \cap q$ with the top-$k$ weights can be reported in $O(\log n + k)$ time.

Now we extend our solution to finite range (i.e. $[a_1, b_1]$). We store the points of $S$ at the leaves of a balanced binary search tree $T$ in non-decreasing order of their $x$-coordinates. At each internal node $v$, we store an instance $DL(v)$ of the data structure of Lemma 3.2.1 built on $S(Left(v))$, the set of points stored in the leaves of the left subtree of $v$. Similarly we store another data structure $DR(v)$ built on $S(Right(v))$, the set of points stored in the leaves of the right subtree of $v$ supporting top-$k$ queries of the form $(-\infty, b_1]$. To answer a query $q = [a_1, b_1]$ we search with $a_1$ and $b_1$ in $T$. This generates paths $l$ and $r$ in $T$ that possibly diverge at some non-leaf node $u$ of $T$. We query $DL(u)$ (respectively $DR(u)$) with $[a_1, \infty)$ (respectively $(-\infty, b_1]$) to retrieve $S_L$ (respectively $S_R$), the top-$k$ points in the range. We can retrieve the overall top-$k$ points in $[a_1, b_1]$ from $S_L$ and $S_R$ in $O(k)$ time due to the fact that the top-$k$ problem is decomposable, as discussed in Section 3.0.2.

**Theorem 3.2.1.** A set, $S$, of $n$ weighted points in $\mathbb{R}^1$ can be preprocessed into a data structure of size $O(n \log n)$ so that for any query range $q = [a_1, b_1]$, and a given integer $k$ satisfying $1 \leq k \leq n$, the points in $S \cap q$ with the top-$k$ weights can be reported in $O(\log n + k)$ time.

### 3.2.2 Solution for $\mathbb{R}^d$, $d \geq 2$

We first handle queries of the form $q = [a_1, b_1] \times \ldots \times [a_d, \infty]$. Let the points in $S$ be lying in a $d$-dimensional space. A $d$-dimensional range tree $T$ is built based on the points in $S$. At the innermost level of $T$ we employ fractional cascading [5]. Fractional cascading does not cause the space to increase but saves a log factor in the query time when a range search
query is performed. Since fractional cascading has been applied, at the innermost level we will have arrays sorted based on their increasing $x_d$-coordinate values. At each such array (say $M$) we do the following: Take each point $p$ in $M$ and associate a list $L_p$ with it. The points which have $x_d$-coordinate value $\geq$ than $p$ in $M$ are sorted in decreasing order based on their weights and put in the list $L_p$.

If $M$ had $m$ elements in it, then the total size of all the lists $L_p$, for all $p \in M$ will be $O(m^2)$. However, notice that given two consecutive points $p$ and $q$ in $M$, $L_p$ can be obtained from $L_q$ by making $O(1)$ changes. Now treating the $x_d$-coordinate as time, we store all the lists $L_p, \forall p \in M$, in a partially persistent structure [34]. The overall memory modifications will be $O(m)$ and hence the total size of all the lists reduces from $O(m^2)$ to $O(m)$. Therefore, the overall size of the $d$-dimensional range tree remains $O(n \log^{d-1} n)$.

To answer a query, we do a standard search on our range tree with $q$. Let $C$ be the set of $O(\log^{d-1} n)$ canonical arrays selected. At each canonical array $c \in C$, we consider a list $L_p$, where $p$ is the point having the least $x_d$-coordinate value among all the points in $c$ having their $x_d$-coordinate value $\geq a_d$. A max-heap $H$ is initialized with the first point from each of these lists $L_p$, using the point’s weight as key. Then we repeat the following $k$ times or until $H$ is empty: We extract and report the point with maximum key in $H$ and then access the list $L_p$ that the reported point belongs to and insert the next point from this list into $H$.

To see the correctness of the query algorithm, note that the points of $S$ that are in $q$ are stored in the canonical arrays. The points of $S$ not lying in $q$ are never considered at any stage. When $H$ is initialized, its root contains an unreported point from $q$ with the largest weight and this property is maintained while $H$ is queried and updated.

The time taken to search in $T$ is $O(\log^{d-1} n)$. The time to initialize $H$ is $O(|C|)$. The time to query and update $H$ is $O(\log |C|)$. Since there are at most $k$ queries and $k$ updates on $H$, the total query time is $O(\log^{d-1} n + k \log C) = O(\log^{d-1} n + k \log \log n)$, since $|C| = O(\log^{d-1} n)$.

**Lemma 3.2.2.** A set, $S$, of $n$ weighted points in $\mathbb{R}^d$ can be preprocessed into a data struc-
CHAPTER 3. EFFICIENT TOP-K QUERIES FOR ORTHOGONAL RANGES

ture of size $O(n \log^{d-1} n)$ so that given a query $q=\left[a_1, b_1\right] \times \ldots \times \left[a_d, \infty\right)$ and an integer $k \in [1, n]$, the top-$k$ points in $S \cap q$ can be reported in sorted order of their weights in $O(\log^{d-1} n + k \log \log n)$ time.

Next we show how to handle queries of the form $q=\left[a_1, b_1\right] \times \ldots \times \left[a_d, b_d\right]$. We store the points of $S$ at the leaves of a balanced binary search tree $BST$ in non-decreasing order of their $x_d$ coordinates. At each internal node $v$, we store an instance $DL(v)$ of the data structure of Lemma 3.2.2 built on $S(left)$, the set of points stored in the leaves of the left subtree of $v$. Similarly we store another data structure $DR(v)$ built on $S(Right(v))$, the set of points stored in the leaves of the right subtree of $v$ supporting top-$k$ queries of the form $q=\left[a_1, b_1\right] \times \ldots \times (-\infty, b_d]$.

Let $q'=\left[a_1, b_1\right] \times \ldots \times \left[a_{d-1}, b_{d-1}\right]$. To answer a query $q = q' \times [a_d, b_d]$ we search with $a_d$ and $b_d$ in $BST$. This generates paths $l$ and $r$ in $BST$ that possibly diverge at some non-leaf node $u$ of $BST$. We query $DL(u)$ (respectively $DR(u)$) with $q' \times [a_d, \infty)$ (respectively $q' \times (-\infty, b_d]$) to retrieve $S_L$ (respectively $S_R$), the top-$k$ points in the range. We can retrieve the overall top-$k$ points in $q$ from $S_L$ and $S_R$ in $O(k)$ time due to the fact that the top-$k$ problem is decomposable, as discussed in Section 3.0.2.

**Theorem 3.2.2.** A set, $S$, of $n$ weighted points in $\mathbb{R}^d$ can be preprocessed into a data structure of size $O(n \log^d n)$ so that given a query $q=\left[a_1, b_1\right] \times \ldots \times \left[a_d, b_d\right]$ and an integer $k \in [1, n]$, the top-$k$ points in $S \cap q$ can be reported in sorted order of their weights in $O(\log^{d-1} n + k \log \log n)$ time.

3.3 Conclusions and Future Work

We have given efficient solutions for finding the top-$k$ points inside an orthogonal query box. Removing the $O(\log \log n)$ penalty per reported point in the query time for the version wherein points are reported in sorted order of weights remains an important open problem.
Chapter 4

Range-Aggregate Queries Involving Geometric Aggregation Operations

As discussed in Chapter 1, geometric aggregation is an important class of functions and range-aggregate problems with geometric aggregate functions have been not been much studied. In Chapter 1 the wide range of applications of these problems were discussed. Also, it was shown why the existing techniques are not sufficient to answer these queries efficiently.

In this chapter we revisit the range-aggregate query problems with geometric functions which were previously attempted in [30] and [47]. We come up with improved solutions to these problems. The results are shown and compared with previous results in Table 4.2. The problems discussed in these papers were static. In this chapter we come up with semi-dynamic (insertions) solutions to some of these problems. To the best of our knowledge, this is the first attempt being made at finding dynamic solutions to problems involving geometric aggregation operations. See Table 4.1 for the results obtained for semi-dynamic case.

In Section 4.2, we consider static $d$-dimensional range-aggregate point enclosure problem ($d \geq 1$). In Section 4.3 we show to how to handle insertions efficiently for this problem. In Section 4.4, the static and the semi-dynamic (insertions) 1-d range-aggregate segment
### 4.1 Static Range-Aggregate Point Enclosure

In this section we shall consider the static $d$-dimensional range-aggregate version of the point enclosure problem. A set $S$ of points and a set $T$ of orthogonal hyperboxes in $\mathbb{R}^d (d \geq 1)$ are given and we need to report all point-hyperbox incidences inside a query orthogonal hyperbox.

**Problem 1.** Preprocess a set $S$ of points and a set $T$ of orthogonal (axes-parallel) hyperboxes in $\mathbb{R}^d (d \geq 1)$ with $|S| + |T| = n$, such that given a query orthogonal hyperbox $q = [a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_d, b_d]$, all pairs $(s, t), s \in S, t \in T$ satisfying $s \in (t \cap q)$ can be reported efficiently.

In Chapter 1, there was a discussion on what makes this problem non-trivial. We shall repeat the same argument here for maintaining the flow: At first look we can think of the following two approaches to solve all the problems considered in this chapter: a) First solve the range query problem and then apply aggregate function on the appropriate subset.
CHAPTER 4. RANGE-AGGREGATE QUERIES INVOLVING GEOMETRIC AGGREGATION OPERATIONS

<table>
<thead>
<tr>
<th>Underlying space</th>
<th>Objects in $S$</th>
<th>Objects in $T$</th>
<th>Query</th>
<th>Space</th>
<th>Query time</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbb{R}^1$</td>
<td>points</td>
<td>segments</td>
<td>point enclosure</td>
<td>$O(n \log n)$</td>
<td>$O(\log n + k)$</td>
<td>[47]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$O(n)$</td>
<td>$O(\log n + k)$</td>
<td>New</td>
</tr>
<tr>
<td>$\mathbb{R}^d$</td>
<td>points</td>
<td>hyper rectangles</td>
<td>point enclosure</td>
<td>$O(n \log^d n)$</td>
<td>$O(\log^d n + k)$</td>
<td>[47]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$O(n \log^{d-1} n)$</td>
<td>$O(\log^{d-1} n + k)$</td>
<td>New</td>
</tr>
<tr>
<td>$\mathbb{R}^1$</td>
<td>segments</td>
<td>segments</td>
<td>intersections</td>
<td>$O(n \log n)$</td>
<td>$O(\log n + k)$</td>
<td>[47]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$O(n)$</td>
<td>$O(\log n + k)$</td>
<td>New</td>
</tr>
<tr>
<td>$\mathbb{R}^d$</td>
<td>horizontal</td>
<td>vertical</td>
<td>intersections</td>
<td>$O(n \log^2 n)$</td>
<td>$O(\log^2 n + k)$</td>
<td>[47]</td>
</tr>
<tr>
<td></td>
<td>segments</td>
<td>segments</td>
<td></td>
<td>$O(n \log n)$</td>
<td>$O(\log n + k)$</td>
<td>New</td>
</tr>
</tbody>
</table>

Table 4.2 Comparison of results of static range-aggregate query problems; the query is an orthogonal range; $k$ is the output size.

of objects, or b) First apply the aggregate function on the whole set of objects and then return the appropriate result which is satisfied by the range query. Unfortunately, neither of these approaches lead to efficient solutions. Next we shall show specifically how these techniques fail to answer efficiently Problem 1.

Using approach (a) we can query set $S$ and set $T$ with $q$. Let $\alpha$ be the total number of points (from $S$) and segments (from $T$) intersecting $q$. Using standard techniques (like a sweep line approach) we can find out all the point-segment intersections in $O(\alpha \log \alpha + k)$ time. Therefore, the total query time would be $O(\log n + \alpha \log \alpha + k)$, where $k$ is the actual output size. If $\alpha$ is large, then this solution will not be efficient (a lot of points might not be intersecting any of the segments and vice-versa). Using approach (b), in pre-processing phase we can find out all the pair of point-segment intersections in $O(n \log n + \beta)$ time, where $\beta$ are the number of pairs of intersection. Then we can put these pairs of intersection in a binary search tree of size $O(\beta)$ and report the appropriate points for a query $q$ in
CHAPTER 4. RANGE-AGGREGATE QUERIES INVOLVING GEOMETRIC AGGREGATION OPERATIONS

$O(\log \beta + k)$ time. If $\beta$ is large, then the space occupied would be more.

Since these approaches do not work, Gupta et al. [47] came up with a new technique to answer this query (as shown in Table 4.2). In this section we come up with new ideas to improve upon the results obtained by Gupta et al. [47] (our results are also shown in Table 4.2). We shall start by considering the problem in a 1-dimensional scenario. Next, we show how to extend the solution to higher dimensions ($d \geq 2$).

### 4.1.1 One-dimensional scenario.

Preprocess a set $S$ of points and a set $T$ of segments on the $x$-axis, with $|S| + |T| = n$ such that given a query interval $q = [a_1, b_1]$, all pairs $(s, t), s \in S, t \in T$ satisfying $s \in t \cap q$ can be reported efficiently.

We sort the points in $S$ in non-decreasing order and remove any point in $S$ which does not stab (or intersect) any segment in $T$. The reduced set $S' \subseteq S$ is stored at the leaves of a balanced binary search tree $BST$. $BST$ is searched with segments $t \in T$ to find if any point in $S$ stabs $t$. If no such point exists for a segment $t \in T$, then it is removed. Let $T' \subseteq T$ be the reduced set. The segments in $T'$ partition the $x$-axis into $2|T'| + 1$ elementary intervals. It might happen that some of these intervals are empty. Let $I$ be the set of these elementary intervals. With each interval $i \in I$, we maintain a list ($L_i$) of segments, $t' \in T'$, such that $t' \cap i \neq \emptyset$. Also, with each leaf point in $BST$, we maintain a pointer to that interval in $I$ which it stabs (each point can stab only one interval in $I$).

The total size of all the lists $L_i, \forall i \in I$ will be $O(n^2)$. Notice that if we add up the total number of changes occurring in every pair of consecutive lists $L_i$ and $L_{i+1}$, it will turn out to be $O(n)$. The reason being that changes between consecutive lists occur either due to an entry of a new segment $t' \in T'$ or removal of a segment $t' \in T'$ and the number of endpoints in $T'$ are $2|T'| \equiv O(n)$. Hence, we make use of this fact and build a partially persistent data structure $D$ [34], instead of separately storing the lists $L_i, \forall i \in I$. We start with an initially empty structure $D$ and by treating the $x$-axis as time, we store the lists $L_i, \forall i \in I$.
CHAPTER 4. RANGE-AGGREGATE QUERIES INVOLVING GEOMETRIC AGGREGATION OPERATIONS

into $D$. Since, the number of modifications will be $O(n)$, the total size of $D$ will be $O(n)$.

Given a query interval $q = [a_1, b_1]$, BST is searched and the all the points in $S'$ lying within $q$ are found out. For each reported point $p$, the pointer stored with it is followed to reach the elementary interval $i \in I$ it is stabbing. Then, the tuples $(p, l_i), \forall l_i \in L_i$, are reported. The list $L_i$ is obtained by accessing $D$.

**Theorem 4.1.1.** A set $S$ of points and a set $T$ of segments on the $x$-axis where $|S| + |T| = n$, can be preprocessed into a data structure of size $O(n)$ such that given a query interval $q = [a_1, b_1]$, all pairs $(s, t), s \in S, t \in T$ such that $s \in t \cap q$ can be reported in time $O(\log n + k)$ where $k$ is the output size.

### 4.1.2 Extension to higher dimensions ($d \geq 2$).

We begin with $d = 2$ and then generalize that solution to higher dimensions. First, we discard all the points in $S$ which do not stab any rectangle in $T$. The remaining points in $S$ (say $S'$) are put in a two-dimensional Range Tree $RT$. One way of solving this problem is to maintain with each point in $S'$ the list of rectangles in $T$ that it stabs. Then given a query rectangle $q$, we shall query $RT$ with $q$ and for each point inside $q$, report the rectangles it stabs. However, the space required in this case can blow up to $O(n^2)$. We overcome this issue by maintaining a sparse list with each point in $S'$.

Next we will have to build a new data structure that solves the standard “point enclosure” problem of reporting all the rectangles stabbed by a query point. Based on the $x$-projections of the rectangles in $T$, we build a segment tree $ST$. Let $I(v)$ be the set of segments allocated to a node $v$ in $ST$. At each node $v \in ST$, based on the $y$-projection’s of the rectangles whose $x$-projections are in $I(v)$, we build an instance of data structure $D$ of Theorem 4.1.1 for storing the $y$-projection’s. Hence, $ST$ can be used for answering a standard point enclosure problem, for a given query point in $\mathbb{R}^2$. The space occupied by the Range Tree $RT$ and the augmented Segment Tree is $O(n \log n)$.

Consider a point $p \in S$. Do a stabbing query on only the primary structure of $ST$. Let
Π(p) be the path from root till the leaf obtained by querying with p. At each node v ∈ Π(p), if I(v) ≠ ∅ query the secondary structure and find out the elementary interval within which point p lies. Make a list L(p) storing the pointer to the elementary interval in which p lies ∀ v ∈ Π(p). The list L(p) is prepared for each point p ∈ S. The total size of the lists L(p), ∀ p ∈ S′ will be O(n log n). In this way we have successfully reduced the space complexity from O(n^2) to O(n log n).

Given a query rectangle q = [a_1, b_1] × [a_2, b_2], we query RT with q. For each point p ∈ q, we follow all the pointers in L(p). By following each pointer we reach an elementary interval, say i. Then we start reporting pairs (p, r) where r is the rectangle corresponding to each segment stored in L_i (L_i is the set of segments stored corresponding to the elementary interval i). The time taken to query RT is O(log n + k'), where k' is the number of points reported by RT and the time taken to report all the k pairs is O(k). Therefore, the total query time is O(log n + k).

This solution can be directly extended to higher dimensions (d > 2). In a d-dimensional space, RT will be a d-dimensional Range Tree built on points in S. ST will be a d-dimensional Segment tree. However, at the deepest level we shall replace a segment tree by an instance of the structure D of Theorem 4.1.1. By doing a stabbing query on ST each point p ∈ S will create a list L(p) of size O(log^{d-1} n) size. The size of RT and ST will be O(n log^{d-1} n). The time taken to answer for a query hyperbox will be O(log^{d-1} n + k).

**Theorem 4.1.2.** A set S of points and a set T of axes-parallel hyperboxes in IR^d for d ≥ 2 with |S| + |T| = n, can be preprocessed into a data structure of size O(n log^{d-1} n) such that given a query hyperbox q = [a_1, b_1] × [a_2, b_2] × ... × [a_d, b_d], all pairs (s, t), s ∈ S, t ∈ T such that s ∈ (t ∩ q) can be reported in time O(log^{d-1} n + k) where k is the output size.
4.2 Semi-dynamic Range-Aggregate Point Enclosure

In this section we shall build data structures which can handle insertions efficiently for solving the Range-Aggregate Point Enclosure problem. As done previously, the one-dimensional scenario is presented first and then extended to higher dimensions. To the best of our knowledge, the previous work on this problem only deals with the static case. Ours is the first attempt to come up with a semi-dynamic solution for this problem.

4.2.1 One-dimensional scenario.

The preprocessing steps are as follows. Using the segments in $T$, a standard segment tree $ST$ is built. Structure $ST$ is equipped to handle both the reporting and the counting queries (query here will be a point). Construction of the segment tree takes $O(n \log n)$ time and space. Point set $S$ is divided into three disjoint subsets $S_1$, $S_2$ and $S_3$. If a point $p \in S$ intersects with none of the segments in $T$, then $p$ falls into set $S_1$. If the number of segments of $T$ with which $p$ intersects lies in the range $(0, \log n)$, then it falls into set $S_2$. $S_3 \subseteq S$ contains the points which intersect with more than “$\log n - 1$” segments of $T$, i.e., in the range $[\log n, n]$. Segment tree $ST$ is used for finding the number of segments of $T$ being intersected by each point in $S$. This partition of point set $S$ into three subsets takes $O(n \log n)$ time.

Based on the $x$-coordinates of the points in $S_i$ ($\forall i=1,2,3$), we build a balanced binary search tree $BT_i$. We pick $BT_2$ for further augmentation while $BT_1$ and $BT_3$ are not augmented further. The points in $S_2$ are placed at the leaf nodes of $BT_2$. With each point $p \in S_2$ present at a leaf node of $T_2$, we store a list, $L_p$, of all the segments of $T$ it intersects. Also, the size of each list $L_p$ is maintained. The list $L_p$ can be found by querying $ST$, which will take time $O(\log n + |L_p|) \equiv O(\log n)$ since $|L_p| \in (0, \log n)$. Therefore, the time taken to augment $BT_2$ will be $O(n \log n)$. The time taken to build the trees $BT_1$, $BT_2$ and $BT_3$ is $O(n \log n)$. The total preprocessing time is $O(n \log n)$. The space occupied is $O(n \log n)$ since both $BT_2$ and $ST$ take up $O(n \log n)$ space.
CHAPTER 4. RANGE-AGGREGATE QUERIES INVOLVING GEOMETRIC AGGREGATION OPERATIONS

Given a query interval \( q = [a_1, b_1] \), \( BT_3 \) is queried with \( q \). For each point \( p \) in \( BT_3 \) lying within \( q \), we query \( ST \) with \( p \) and report all the pairs \((p, t)\) satisfying \( p \in (t \cap q) \) and \( t \in T \). Next, we query \( BT_2 \) with \( q \). For each point \( p \) in \( BT_2 \) lying within \( q \), we shall report the pairs \((p, t), \forall t \in L_p \).

The query time of the algorithm is analyzed. The time taken to query \( BT_2 \) is \( O(\log n + \Sigma |L_p|) \equiv O(\log n + k') \), where \( k' \) is the number of pairs formed by points of \( S_2 \) lying within \( q \). Let \( k_p \) be the number of segments intersected by point \( p \in (S_3 \cap q) \). Then the time taken to query \( BT_3 \) will be: \( O(\Sigma (\log n + k_p)) \equiv O(\Sigma k_p) \equiv O(k'') \), since \( k_p \geq \log n \) and \( k'' \) be the number pairs formed by points of \( S_3 \) lying within \( q \). Therefore, the total query time will be \( O(\log n + k' + k'') \equiv O(\log n + k) \), where \( k \) is the total number of pairs to be reported.

Handling insertions. Suppose a new point \( p \) is added to the set \( S \). It is first queried on \( ST \) to find out the number of segments of \( T \) it intersects with (say \( k_p \)). Based on the value of \( k_p \), it is kept in one of the subsets \( S_1, S_2 \) or \( S_3 \) and then inserted appropriately into one of the binary trees \( BT_1, BT_2 \) or \( BT_3 \). If \( p \) is inserted into \( BT_2 \), then the list \( L_p \) of the segments of \( T \) it intersects is also prepared. Thus insertion of a point \( p \) can be handled in \( O(\log n) \) time.

Now, suppose a new segment \( t \) is to be added to the set \( T \). \( t \) is first inserted into the segment tree \( ST \). This takes \( O(\log n) \) amortized time. Next, \( BT_2 \) is queried with \( t \). For each point \( p \in BT_2 \) which intersects \( t \), \( t \) is added to the list \( L_p \). If \( |L_p| = \log n \), then \( p \) is shifted from set \( S_2 \) to \( S_3 \). \( p \) and its list \( L_p \) is deleted from \( BT_2 \), and \( p \) is inserted into \( BT_3 \). Let \( \lambda_1 \) be the number of points in \( BT_2 \) which are intersected by \( p \) and \( \lambda_2 \) be the no. of points shifting from \( BT_2 \) to \( BT_3 \). Then the time taken to update the lists in \( BT_2 \) and the shifting process from \( BT_2 \) to \( BT_3 \) takes \( O(\log n + \lambda_1 + \lambda_2 \log n) \). Then \( BT_1 \) is queried with \( t \). For each point \( p \in BT_1 \) which intersects \( t \), \( p \) is deleted from \( BT_1 \) and inserted into \( BT_2 \). In \( BT_2 \), the list \( L_p \) is initialized for point \( p \) with \( t \) being the only entry in it. Let \( \lambda_3 \) be the number of points shifting from \( BT_1 \) to \( BT_2 \). This will take \( O(\log n + \lambda_3 \log n) \) time. Therefore, the total time for inserting a new segment \( t \) is \( O(\log n + \lambda_1 + \lambda_2 \log n + \lambda_3 \log n) \).
CHAPTER 4. RANGE-AGGREGATE QUERIES INVOLVING GEOMETRIC AGGREGATION OPERATIONS

\( \lambda_3 \log n \).

An amortized analysis is carried out to get an efficient bound. Assume that we insert \( n \) segments and points in an arbitrary order. Notice that a point in set \( S_1 \) can jump only once into set \( S_2 \) and a point in set \( S_2 \) can jump only once into set \( S_3 \). Therefore, the value of \( \Sigma \lambda_2 \) and \( \Sigma \lambda_3 \) where the summation is over \( n \) insertions are bounded by \( O(n) \). A point in \( S \) can remain in the set \( S_2 \) till it intersects with less than \( \log n \) segments. Also, the number of points in set \( S \) after \( n \) insertions is still bounded by \( O(n) \). Therefore, the value of \( \Sigma \lambda_1 \) where the summation is over \( n \) insertions is \( O(n \log n) \). Therefore, the total time taken for insertion of \( n \) segments and points is: \( O(\Sigma(\log n + \lambda_1 + \lambda_2 \log n + \lambda_3 \log n)) \equiv O(n \log n) \). The amortized time turns out to be \( O(\log n) \).

After \( n \) insertions of points and segments, the whole data structure is deleted and reconstructed. After \( n \) insertions, the total number of points and segments become \( 2n \). Therefore, we shall update the criteria for a point \( p \) to enter set \( S_2 \) and \( S_3 \) from \((0, \log n)\) to \((0, \log 2n)\) and \([\log n, n]\) to \([\log 2n, 2n]\), respectively. Since, the preprocessing time is \( O(n \log n) \) when built on \( n \) points and segments, the amortized time of insertion does not change.

**Theorem 4.2.1.** A set \( S \) of points and a set \( T \) of segments on the x-axis where \( |S| + |T| = n \), can be preprocessed into a data structure of size \( O(n \log n) \) such that given a query interval \( q = [a_1, b_1] \), all pairs \((s, t), s \in S, t \in T \) such that \( s \in t \cap q \) can be reported in time \( O(\log n + k) \) where \( k \) is the output size. Also, insertion of a point or a segment can be handled in \( O(\log n) \) amortized time.

### 4.2.2 Extending it to higher dimensions.

Extending our solutions to higher dimensions turns out to be a straightforward process. In \( \mathbb{R}^d \), we have a set \( S \) of \( d \)-dimensional points and a set \( T \) of \( d \)-dimensional orthogonal hyperboxes. A dynamic \( d \)-dimensional segment tree \( ST \) \[26\] is built based on the hyperboxes in set \( T \). Set \( S \) is again divided into three disjoint sets \( S_1, S_2 \) and \( S_3 \). If a point intersects no hyperbox of \( T \) then it goes into set \( S_1 \), if the number of intersections is in the range

\( \lambda_3 \log n \).
(0, \log^d n) \) then the point goes into set \( S_2 \) and finally if the point intersects with more than \( \log^d n - 1 \) hyperboxes then it goes into set \( S_3 \). Balanced binary trees \( BT_1, BT_2 \) and \( BT_3 \) are replaced by dynamic range trees [26] which can answer range queries in \( d \)-dimensional space. A dynamic range tree when built on \( m \) points occupies \( O(m \log^{d-1} m) \) space and answers queries in \( O(\log^d m + k) \) time. It handles insertions and deletions in \( O(\log^d m) \) amortized time. With each point \( p \in BT_2 \) (or \( S_2 \)) list \( L_p \) is prepared. Appropriate pointers are maintained by points in \( BT_2 \) to their lists. As done previously, after \( n \) insertions of points and hyperboxes, the whole data structure is deleted and reconstructed.

The total space occupied will be \( O(n \log^d n) \) as \( BT_2 \) and \( ST \) occupy \( O(n \log^d n) \) space. The query time will be \( O(\log^d + k) \) (the analysis done for 1-d case holds here as well). The time taken to insert a point or a hyperbox is \( O(\log^d n) \) amortized time. Using fractional cascading [72], the query time can be reduced to \( O(\log^{d-1} n \log \log n + k) \).

**Theorem 4.2.2.** A set \( S \) of points and a set \( T \) of axes-parallel hyperboxes in \( \mathbb{R}^d \) for \( d \geq 2 \) with \( |S| + |T| = n \), can be preprocessed into a data structure of size \( O(n \log^d n) \) such that given a query hyperbox \( q = [a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_d, b_d] \), all pairs \((s, t), s \in S, t \in T\) such that \( s \in (t \cap q) \) can be reported in time \( O(\log^{d-1} n \log \log n + k) \) where \( k \) is the output size. Insertion of a new point or a hyperbox takes \( O(\log^d n) \) amortized time.

### 4.3 1-d Range-Aggregate Segment Intersection

**Problem 2.** Preprocess a set \( S \) of \( n \) segments on the \( x \)-axis, such that given a query interval \( q = [a_1, b_1] \), all pairs \((s, t), s \in S, t \in S\) satisfying \( s \cap t \cap q = \emptyset \) can be reported efficiently.

#### 4.3.1 Static Solution

We need to find out pairwise intersections of the segments in \( S \) which overlap with the query interval \( q \). The above problem is characterized in the following lemma [47].
Lemma 4.3.1. A pair of segments \((s, t)\) of \(S\) satisfies \(s \cap t \cap q \neq \emptyset\) iff

(i) An endpoint of \(s\) is in \(t \cap q\) or
(ii) An endpoint of \(t\) is in \(s \cap q\) or
(iii) \(q \subseteq (s \cap t)\)

In [47], to report pairs satisfying conditions (i) and (ii) of Lemma 4.3.1, they use a data structure that takes up \(O(n \log n)\) space and \(O(\log n + k)\) time. We shall reduce it to \(O(n)\) by the following steps: Discard all the segments of \(S\) which do not intersect with any other segment of \(S\). Call the reduced set \(S'\). However, we can simply preprocess the segments of \(S'\) and the endpoints of the segments of \(S'\) into an instance of the data structure of Theorem 4.1.1. For a query interval \(q\), we query the data structure and for each reported tuple \((p, i)\), we report \((p', i)\) where \(p'\) is the segment to which \(p\) is an endpoint. This will reduce the space requirement to \(O(n)\).

To report segment pairs satisfying condition (iii) of Lemma 4.3.1, in [47], they map each segment \(s[c, d]\) of \(S'\) into the point \((c, d) \in \mathbb{R}^2\). These points are processed into a data structure \(D\) for 2-d quadrant searching. This can be implemented using a priority search tree [69]. The query interval \(q = [a_1, b_1]\) is mapped into the northwest quadrant \(NW(q)\) of the point \((a_1, b_1) \in \mathbb{R}^2\). \(D\) is queried with \(NW(q)\) and the result is stored in a temporary list \(L(q)\). For each pair of points \((p_1, p_2), p_1 \in L(q), p_2 \in L(q)\), the interval pair \((p'_1, p'_2)\) is reported where \(p'_1\) (respectively \(p'_2\)) is the segment corresponding to \(p_1\) (respectively \(p_2\)).

Theorem 4.3.1. A set \(S\) of \(n\) segments on the x-axis can be preprocessed into a data structure of size \(O(n)\) such that given a query interval \(q = [a, b]\), all pairs \((s, t), s \in S, t \in S\) such that \(s \cap t \cap q \neq \emptyset\) can be reported in time \(O(\log n + k)\) where \(k\) is the output size.

4.3.2 Semi-dynamic solution

The solution to the semi-dynamic version of the problem is similar to that of the static solution. We shall preprocess the segments of \(S\) and the endpoints of the segments of \(S\) into an instance of the data structure of Theorem 4.2.1. However, one important consideration is
taken into account while preparing lists $L_p$ for $p \in BT_2$. An endpoint of a segment $t$ is not considered to be intersecting with the segment it comes from (which in this case is $t$). This observation has to be incorporated into the data structure of Theorem 4.2.1. This makes sense since in conditions (i) and (ii) of Lemma 4.3.1 we want an endpoint of a segment $t$ to intersect with a segment other than $t$. For handling condition (iii), instead of using a static priority search tree (used in the static solution), we shall use a Dynamic priority search tree $D$ [27]. A Dynamic priority search tree when built on $m$ points takes up $O(m)$ space, answers queries in $O(\log m + k)$ time and updates take $O(\log m)$ time. In our case $m \equiv O(n)$. Given a query interval $q$, the same procedure as followed for the static solution is repeated.

**Theorem 4.3.2.** A set $S$ of $n$ segments on the $x$-axis can be preprocessed into a data structure of size $O(n \log n)$ such that given a query interval $q = [a_1, b_1]$, all pairs $(s, t), s \in S, t \in S$ such that $s \cap t \cap q \neq \emptyset$ can be reported in time $O(\log n + k)$ where $k$ is the output size. Also, insertion of a new segment can be done in $O(\log n)$ amortized time.

### 4.4 Range-Aggregate Orthogonal Segment Intersection

**Problem 3.** Given a set $H$ of horizontal segments and a set $V$ of vertical segments ($|H| + |V| = n$), preprocess them into a data structure such that given query rectangle $q = [a_1, b_1] \times [a_2, b_2]$, we can efficiently report all the pairs of horizontal-vertical segments $(h, v)$ such that $h \in H, v \in V$, and $h \cap v \cap q \neq \emptyset$.

As stated previously, there are two approaches to answer this query:

- Using an existing data structure in the literature, find out all the horizontal and vertical segments intersecting the query rectangle $q$ (call them $S'$). Then find out all the intersections happening among the segments of $S'$. The pitfall of this technique is that most segments of $S'$ might not have an intersection with any other segment of $S'$ which leads to wastage of query time. Finding intersections among segments of $S'$
CHAPTER 4. RANGE-AGGREGATE QUERIES INVOLVING GEOMETRIC AGGREGATION OPERATIONS

Figure 4.1 Different types of vertical and horizontal segments that can intersect $q$.

... takes $O(|S'| \log |S'| + k)$ time, which can be expensive if $|S'|$ is large (i.e. $\gg k$).

- In the pre-processing phase find out all points of intersection between the horizontal and the vertical segments and store them in a two-dimensional range tree. Querying the range tree with query rectangle $q$ will report all the intersections happening inside the query rectangle $q$. The pitfall of this technique is that the number of intersections among the segments can be $O(n^2)$ in the worst case which is not preferable.

Gupta et al. [47] came up with new data structures to answer this query in an efficient manner. However, we noticed that there is scope for improving the results. This lead to the need for new data structure for answering this query.

The vertical (resp. horizontal) segments $V$ (resp. $H$) intersecting $q$ can be categorized into three categories: (a) segments whose both the endpoints are inside $q$, (b) segments whose one endpoint is inside $q$, (c) segments which cross $q$ completely. In Figure 4.1 we show an example of segments in $H$ and $V$ classified into these three categories.

First, we shall build a data structure to report all the intersections involving vertical segments of type (a) and (b), for a given query $q$. Then another data structure is built to help in reporting all the intersections involving horizontal segments of type (a) and (b). The data structures for these are described next. We assume that the endpoints of no two horizontal...
(resp. vertical) segments have the same x and y coordinate. In the preprocessing phase, create a bounding box $B$ for the segments in the set $H \cup V$. Decompose the bounding box $B$ into vertical slabs by shooting vertical rays upward and downward from the endpoints of all the horizontal segments in set $H$ till they hit the walls of $B$. This divides the plane into many vertical slabs. For each vertical slab $S_i$, we create a list $L_i$ which stores the y-projection of all the horizontal segments of $H$ passing through the slab $S_i$ (similar to the technique used earlier in the paper). The list $L_i$ stores the y-projections of the horizontal segments in a sorted order. Note that the list $L_i$ is almost similar to the list $L_{i+1}$ except there is an inclusion or deletion of a value from the list $L_i$. Hence these lists $L_i$ can be implemented using persistence as done previously.

A vertical segment $v(v_x; v_l, v_u) \in V$ represents a segment with $v_x$ as its vertical projection and $[v_l, v_u]$ as its y-projection. For each vertical segment $v \in V$, we first find the slab $S_i$ where the $x$-projection of $v$ (i.e., $v_x$) lies. Then in the slab $S_i$, we consider the $y$-projection of $v$, i.e., $[v_l, v_u]$ and find out the smallest element $h_l$ and the largest element $h_u$ in the list $L_i$ such that $v_l \leq h_l \leq h_u \leq v_u$. Create two 2-d points $(v_x, h_l)$ and $(v_x, h_u)$. Store these points in a 2-d range searching data structure $RT_v$. With each point $p$ in $RT_v$ we shall maintain appropriate pointer to the element in $L_i$ from which $p$ got generated. The space occupied by $RT_v$ will be $O(n \log n)$. These structures will help us in reporting intersections involving vertical segments of type (a) and (b).

Analogously, in order to report horizontal segments of type (a) and (b), we shall build similar data structures as done to handle vertical segments. This will lead to creating lists $L_i$ based on the vertical segments and an analogous tree $RT_h$.

Now we shall build data structures to handle intersections where both the vertical and horizontal segments are of type (c). A data structure $T_v$ (resp. $T_h$) shall be built, which for a given query rectangle $q$ shall report all the vertical (resp. horizontal) segments completely crossing $q$. Convert each vertical segment $v \in V$ into a 3-d point $v'(v_x, v_l, v_u)$. Create a binary search tree $T_v$ whose leaf nodes are sorted in terms of the $v_x$-values of these 3-d points. At each internal node $\mu \in T_v$, collect all the points of the subtree rooted at $\mu$. Let
$X(\mu)$ denote the average of the $v_x$-value in the rightmost leaf in $\mu$’s left subtree and the $v_x$-value in the leftmost leaf in $\mu$’s right subtree. Let $\mu$ be a left child of its parent. At the node $\mu$, create a data structure $D_\mu$ which is an instance of [2] for handling 3D-dominance reporting queries and handles queries of the form $q' = [a_1, \infty) \times (-\infty, a_2] \times [b_2, \infty)$. If $\mu$ is a right child, $D_\mu$ will be a 3D-dominance reporting data structure to report points in $q'' = (-\infty, b_1] \times (-\infty, a_2] \times [b_2, \infty)$. The data structure $D_\mu$ takes linear storage space and answers queries in $O(\log n + k)$ query time [2]. Hence, the space occupied by $T_v$ will be $O(n \log n)$. Similarly, based on horizontal segments ($H$) we build tree $T_h$ but now the primary structure is based on the $y$-projection of $H$.

Given a query rectangle $q = [a_1, b_1] \times [a_2, b_2]$, search $RT_v$ and for each point $(v_x, h_i)$ reported, jump to the slab $S_i$ where the coordinate $v_x$ lies. Here $v_x$ is the $x$ projection of a vertical segment $v$. Next, starting from the index of the value $h_i$ in the list $L_i$ we descend (ascend) the list $L_i$ until we find a value $h'_y \in L_i$ such that (1) $h'_y < a_2$ (resp $h'_y > b_2$) or (2) $h'_y < v_i$ (resp $h'_y > v_u$). Here $[v_i, v_u]$ is the $y$-projection of the vertical segment $v$. This will report all the intersections involving vertical segments of type (a) and (b). Similarly, query $RT_h$ to report intersections involving horizontal segments of type (a) and (b). This will take $O(\log n + k_1)$ time, where $k_1$ is the number of reported pairs involving segments of type (a) and (b).

Now, all that is left is to report all intersections $(h, v)$ in which both $h \in H$ and $v \in V$ completely cross $q$. We search $T_v$ with $[a_1, b_1]$ to find the highest node $\pi \in T_v$ such that $X(\pi)$ lies in the interval $[a_1, b_1]$. Let $l$ and $r$ be the left and the right child of $\pi$, respectively. Search $\pi_l$ with $q_l=[a_1, \infty) \times (-\infty, a_2] \times [b_2, \infty)$ and $\pi_r$ with $q_r=(-\infty, b_1] \times (-\infty, a_2] \times [b_2, \infty)$. If $\pi_l$ or $\pi_r$, report at least one point, then quit the query procedure on $T_v$. Similarly, query $T_h$ and quit the query procedure if it reports at least one point. If both $T_v$ and $T_h$ report at least one point on being queried with $q$, then once again query both the structures with $q$ and this time all the points satisfying the query $q$ are reported. Let $H^q_v$ and $V^q_v$ be the set of segments be reported by querying $T_h$ and $T_v$, respectively. Then we shall report all the pairs $(h, v), \forall h \in H^q_v$ and $v \in V^q_v$. This procedure will take $O(\log n + k_2)$, where $k_2$ is the
number of pairs involving type (c) segments. The overall query time is \( O(\log n + k) \).

**Theorem 4.4.1.** A set \( H \) of horizontal segments and a set \( V \) of vertical segments (\(|H| + |V| = n\)) in \( \mathbb{R}^2 \) can be preprocessed into a data structure of size \( O(n \log n) \) such that given a query rectangle \( q = [a_1, b] \times [a_2, b_2] \), all pairs of horizontal-vertical segments \((h, v)\), \( h \in H \), \( v \in V \) and \( h \cap v \cap q \neq \emptyset \) can be reported in time \( O(\log n + k) \), where \( k \) is the output size.

### 4.5 Conclusions and open problems

In this chapter we consider geometric aggregation functions and come up with improved solutions for problems attempted previously in [30] and [47]. Also, we come up with semi-dynamic solutions for some of these problems. An important open problem is to come up with fully dynamic solutions (both insertions and deletions) for these problems.
Chapter 5

Range aggregate structures for colored geometric objects-I

In this chapter and the following chapter we consider range-aggregate queries involving colored geometric objects. In a colored (or generalized) intersection searching problem [60], a set of objects $S$ comes aggregated in groups and we indicate the group a point belongs to by assigning each group an unique color. Our goal is to preprocess these points such that given a query $q$, the distinct colors of the objects in $S$ that intersect $q$ can be reported or counted efficiently [51, 50, 52, 49].

In this work (which has been described in two chapters), we consider a general class of problems and the problem statement is as follows:

- We are given a set $S$ of possibly weighted objects in $\mathbb{R}^d$, $d \geq 1$ which comes aggregated in groups and we indicate the group an object belongs to by assigning each group a unique color. The goal is to preprocess $S$ such that given a query orthogonal range $q$, we can report efficiently for each distinct color $c$ of the objects in $S \cap q$, the tuple $\langle c, F(c) \rangle$ where $F(c)$ is a function of the objects of color $c$ in $q$. The function $F$ is assumed to be a commutative semigroup which allows us to combine results of subproblems without losing correctness.
If \( \mathcal{F}(c) = \text{NULL} \), the unweighted variant of the problem is the generalized orthogonal range reporting problem [60]. Lai et al. [65] studied this class of problems for approximate queries for functions like \( \min, \max, \sum, \text{count}, \text{report} \) and \( \text{heavy} \).

### 5.0.1 Motivation

The above class of queries has been well studied in the database community as “GROUP-BY” queries. The “GROUP-BY” is a common basic operation in databases and is applied to the categorical attributes [83, 1, 44]. As remarked in [1], they are among the most important class of queries in OLAP (“Online Analytical Processing”) applications in decision support systems. As an example of a GROUP-BY query, consider a database of mutual funds which has three attributes for each fund: its annual total return, its beta and the fund family the fund belongs to. Whereas the first two attributes are range attributes (on which we can perform range searching), the third one is not. A typical query is to determine the families that offer funds whose total return is between say 15% and 20% and whose beta is between, say, 0.9 and 1.1 and report the number of funds in the given range for each such family. In database terminology, this is a GROUP-BY on the fund family attribute with aggregation operation \( \text{COUNT} \). This is also an instance of the weighted version of the problem above for \( d = 2 \), wherein each object is a point having unit weight. If \( \mathcal{F}(c) = \text{NULL} \), the problem is the generalized orthogonal range reporting problem [60].

### 5.1 Our Contributions

In this chapter, we have come up with output sensitive solutions while using near-linear space. The results obtained in this chapter are summarized in Table 5.1. The flow of the chapter is as follows: In Section 5.2, the previous work done on colored intersection searching is explained in detail. In Section 5.3, we consider the ‘colored weighted sum problem’ (\( \mathcal{F}(c) \) is weighted sum) in \( \mathbb{R}^d, d \geq 1 \). The technique of ‘adding range restrictions’ is suitably adapted in this section for solving some of the problems considered in this
chapter. In Section 5.4, the ‘colored bounding box problem’ ($F(c)$ is bounding box) in $\mathbb{R}^d$, $d \geq 1$. A variation of this problem is considered in Section 5.5, wherein each color needs to have at least two points inside the query region. Incidentally, we also came up with an optimal solution for a static colored range-searching problem (see Theorem 5.5.1). In Section 5.6 and 5.7, the ‘colored point enclosure weighted sum’ problem is considered in $\mathbb{R}^1$ and $\mathbb{R}^2$, respectively. Section 5.8 deals with ‘Colored segment intersection weighted sum’ problem in $\mathbb{R}^2$. Finally in Section 5.9 we conclude this work and mention some open problems related to it.

5.2 Related Work

Gupta et al. [49] is a good survey paper on “Colored/Generalized Intersection Searching”. Janardan et al. [60] introduced the problem of orthogonal colored range searching. In [60, 51] a systematic study of finding efficient solutions for orthogonal objects was conducted.

**Orthogonal colored range reporting.** Gupta et al. [51] describe solutions for colored range reporting in $\mathbb{R}^1$, $\mathbb{R}^2$, and $\mathbb{R}^3$, with poly-logarithmic query time and near-linear storage. For $\mathbb{R}^1$, they perform orthogonal colored range searching (counting and reporting) in both static and dynamic settings, by a reduction to standard orthogonal range searching in $\mathbb{R}^2$. Specifically, they obtain a dynamic data structure of size $O(n)$, such that the $C$ distinct colors of the points in a query interval can be reported in $O(\log n + C)$ time (or counted in $O(\log n)$ time), while supporting updates (insertions and deletions of points) in $O(\log n)$ time. Specifically, if the points of some color have $x$-coordinates $p_1 < p_2 < \ldots < p_n$, then they are mapped to the points $(\infty, p_1), (p_1, p_2), \ldots, (p_{n-1}, p_n)$ in $\mathbb{R}^2$. A query interval $q = [a, b]$ is mapped to the semi-unbounded rectangle $q = [a, b] \times (\infty, a]$. It is an easy and cute observation that $[a, b]$ contains at least one (resp., no) point of color $c$ if and only if $[a, b] \times (\infty, a]$ contains exactly one (resp. no) transformed point of color $c$. Hence, counting or reporting colors in intervals in $\mathbb{R}^1$ is equivalent to counting or reporting points in the above kind of semi-unbounded rectangles in the transformed set in $\mathbb{R}^2$. 

58
CHAPTER 5. RANGE AGGREGATE STRUCTURES FOR COLORED GEOMETRIC OBJECTS-I

In $\mathbb{R}^2$, the static data structure of Janardan et al. [60] uses $O(n \log^2 n)$ space and answers queries in time $O(\log n + C)$, where as above, $C$ is the output size. In [85], they improve the space to $O(n \log n)$ which is the best known solution for $d = 2$. Gupta et al. [51] obtain a semi-dynamic solution for the reporting problem by first dealing with the special case, when the queries are quadrants of the form $[a, \infty) \times [b, \infty)$. They reduce this special case to standard ray-segment intersection searching in $\mathbb{R}^2$. In order to process arbitrary rectangular queries, they decompose a rectangular query into four quadrant queries, each answered over an appropriate subset of points. The resulting data structure requires $O(n \log^2 n)$ space and allows reporting the $C$ distinct colors of points contained in a query rectangle in $O(\log^2 n + C)$ time. We can insert a point into this data structure in $O(\log^3 n)$ amortized time. Finally, Gupta et al. [51] gave a fully dynamic solution to the reporting problem in $\mathbb{R}^2$. Their solution is based on decomposing the two-dimensional query into $O(\log n)$ one-dimensional queries, each answered over a proper subset of points. This data structure uses $O(n \log n)$ space, supports queries in $O(\log^2 n + C \log n)$ time, where $C$ is the number of reported colors, and allows insertions and deletions in $O(\log^2 n)$ time.

In $\mathbb{R}^3$, Gupta et al. [51] extend their two-dimensional semi-dynamic solution and describe a static data structure of size $O(n \log^4 n)$ with $O(\log^2 n + C)$ query time, where $C$ is the number of reported colors. In [13], Gupta et al. [52] describe a static data structure for orthogonal colored range reporting in any dimension, which requires storage $O(n^{1+\epsilon})$, such that for any query box in $\mathbb{R}^d$, the $C$ distinct colors of points contained in it are reported in $O(\log n + C)$ time.

One of the standard approaches to answer a static colored/generalized intersection searching problem in $\mathbb{R}^d$ is to design a dynamic data structure for the problem in $\mathbb{R}^{d-1}$ (which is usually simpler), which supports efficient query as well as update operations, followed by making this data structure partially persistent, using the technique of Driscoll et al. [34]. In Shi et al. [85], they observe that it is not necessary for the solution to the simpler problem to be the best in terms of space and query time. The more important criteria for the solution to the simpler problem is to enable update operations with a small number
of update steps. Using this observation they answer orthogonal colored range reporting problem in $\mathbb{R}^2$ in $O(\log n + C)$ query time using $O(n \log n)$ space.

Gupta et al. [51] came up with dynamic (both insertions and deletions) solutions to this problem in 2-dimensional space and a static solution in 3-dimensional space. For $d=3$, the only known theoretical solution takes up $O(n \log^4 n)$ space and answers query in $O(\log^2 n + C)$ time [51]. For $d > 3$, there exists a data structure which answers queries in $O(\log n + C)$ time but takes up $O(n^{1+\epsilon})$ space.

**Counting colors in boxes.** As has been observed, colored variants of range searching are in general much harder to solve than the standard variants. The reason for this discrepancy is that colored problems are not decomposable. For example, partitioning the query box into two (disjoint) sub-ranges and counting the number of colors in each sub-range tells us practically nothing about the number of colors in the full range. In [61], they consider the counting version of the problem where the number of distinct colors lying inside the orthogonal query box need to be efficiently reported. In $\mathbb{R}^d$, they provide a simple solution which has poly-logarithmic query time but a worst-case storage of about $O(n^d)$. They map this problem to a dual version to come up with the solution. For random inputs they show that the data structures requires almost linear expected storage. Later several techniques are presented for achieving space-time tradeoff. In $\mathbb{R}^2$, the most efficient solution uses fast matrix multiplication in the preprocessing stage. In higher dimensions simpler tradeoff mechanisms are used, which behave just as well. The hardness of this problem is shown by giving a reduction from matrix multiplication to the offline version of problem. They shows that in $\mathbb{R}^2$ their time-space tradeoffs are close to optimal in the sense that improving them substantially would improve the best exponent of matrix multiplication. They also present a generalized matrix multiplication problem and show its intimate relation to counting colors in boxes in any dimension.

Agarwal et al. [4] show applications of colored range searching problem on a grid. For one-dimensional case, they come up with a structure of size $O(n \log U)$ and answer queries in $O(\log \log U + C)$ time, where the $n$ colored points are in $[0, U]$. By dynamizing the one-
dimensional solution they come up with a solution for two-dimensional scenario. The final structure obtained takes up \( O(n \log U) \) space and answers queries in \( O(\log \log U + C) \) time, where the \( n \) colored points are in \( [0, U] \times [0, U] \). Their solutions can be extended to higher dimensions as well.

**Orthogonal colored point enclosure problem.** The earliest results for this problem were obtained by Janardan et al. [60]. Though the query time obtained was \( O(\log n + C) \), the space occupied was high \( (O(n^{1.5}) \) for \( d=2 \) and \( O(n^{1.75}) \) for \( d=3 \) and so on). In [51], for \( d=2 \) the space occupied by the structure was reduced to \( O(n \log n) \) while the query time increased by a log factor. This result was further improved in [85], where they came up with a structure which took \( O(n) \) space and answered queries in \( O(\log n + C) \) time, which is optimal. [51] also provided a dynamic solution for \( d=1 \). They used a linear-space structure to answer queries in \( O((C + 1) \log n) \) time and updates in \( O(\log n) \) amortized time. The counting version of this problem has been attempted for \( d=1 \) and \( d=2 \) ([51]). For \( d=1 \) and \( d=2 \), the space occupied is \( O(n) \) and \( O(n \log n) \), respectively and the query time is \( O(\log n) \) for both cases.

On a grid, Agarwal et al. [4] attempted point enclosure problem as well. For one-dimensional case, they built a structure of size \( O(nl \log U) \) so that the query can be answered in \( O(\log U + C) \) time, where the endpoints of intervals lie in \( [0, U] \) and \( l \) is a parameter. In two-dimensional case, the space occupied becomes \( O(n^{1+\epsilon}) \) and the query time becomes \( O(\log \log U + C) \), where the endpoints of the rectangles lie on a grid \( [0, U]^2 \).

**Type-2 problems.** In a 1-dimensional static type-2 range counting problem, we wish to preprocess a set \( S \) of \( n \) colored points, so that for each color intersected by a query interval \( q = [a_1, b_1] \), the number of points of that color in \( q \) can be reported efficiently. A solution that takes \( O(n \log n) \) space and supports queries in time \( O(\log n + C) \), \( C \) being the number of colors reported, was given in [51]. The space bound was improved to \( O(n) \) in [13]. For the 2-dimensional static type-2 range counting problem, a solution that takes \( O(n \log n) \) space and \( O(\log^2 n + C \log n) \) query time was given in [13]. For the 1-dimensional static type-2 point enclosure counting problem (here set \( S \) is \( n \) intervals and query \( q \) is a point), a
solution that takes $O(n)$ space and $O(\log n + C)$ query time was given in [51]. To the best of our knowledge, no other results are known for these problems.

**Colored halfspace range searching in $\mathbb{R}^2$ and $\mathbb{R}^3$.** Let $S$ be a set of $n$ colored points in $\mathbb{R}^d$, $d = 2, 3$. Set $S$ is preprocessed so that for any query halfplane $Q$, the $C$ distinct colors of the points lying in the closed halfspace $Q^-$ (i.e. below $Q$) can be reported or counted efficiently. In [50], they use the well-known point-hyperplane duality transform to map $S$ to a set $S'$ of hyperplanes and map $Q$ to a point $q$, both in $\mathbb{R}^d$. The problem is now equivalent to: “Report or count the $C$ distinct colors of the hyperplanes lying on or above $q$.” In $\mathbb{R}^2$, they built an $O(n \log n)$ size structure, so that for any query halfplane the reporting (resp. counting) time was $O(\log^2 n + C)$ (resp. $O(n^{1/2})$). In $\mathbb{R}^3$, the reporting version was solved using $O(n \log^2 n)$ (resp. $O(n^{2+\epsilon})$) space and $O(n^{1/2+\epsilon} + C)$ (resp. $O(\log^2 n + C)$) query time, where $\epsilon > 0$ is an arbitrarily small constant. The counting version was solved in $O(n \log n)$ space and $O(n^{2/3+\epsilon})$ query time. In [50], other non-orthogonal input/query pairs were considered (such as points/fat-triangle and fat-traingles/point in $\mathbb{R}^2$).

**Non-Intersection Queries:** In [53], the authors considered novel queries where, given the colored geometric data the goal is to report the distinct colors such that no objects from those colors are intersected by the query. They came up with solutions in which the query time is sensitive to the size of the output (which is the number of colors reported). The following two approaches to solve this problem fail: (i) reporting the complement of the set of distinct colors found in the range, or (ii) querying with the complement of the query range. Approach (i) is not output-sensitive since the query time would depend on the number of distinct colors in the range, which can be much greater than the number of colors that avoid the range (the output size). Approach (ii) may not even yield the correct answer: a color found in the complement of the query range could also occur within the range! These problems have applications in mutual funds analysis, and VLSI layout design and verification.

**Other problems considered on a grid.** Problems involving document retrieval or string manipulation can often be cast in the framework of colored intersection searching.
For eg, in the context of document retrieval, the following problem (among others) was considered in [73]: Preprocess an array of colored non-negative integers (i.e., points on the 1-dimensional grid) such that, given two indices into the array, each distinct color for which there is a pair of points in the index range at distance less than a specified constant can be reported efficiently. In the context of substring indexing, the following problem was considered in [38]: Preprocess a set of colored points on the 1-dimensional grid, so that given two non-overlapping intervals, the list of distinct colors that appear in their intersection can be reported efficiently. I/O efficient algorithms were given in the standard external memory model for this problem.

**Other directions of work.** A couple of variations of the above mentioned problems have been tried, which are summarized below:

- **External memory data structures:** Colored range searching query is a fundamental query in a DBMS (GROUP-BY operation). Motivated by this, in [74], the colored range searching queries are studies in the context of large, disk-resident data sets. Using an R-tree they propose an external memory data structure, which is augmented with additional information at each internal node. Maximal and Minimal properties of the points are exploited to augment the R-tree which helps in reducing the query time.

- **Significant Presence Queries:** In a traditional colored range searching query, all colors which have at least one point inside the query region is reported. In [32], they argue that such a point, however, could be an ‘outlier’ in its color class. Therefore, they consider a variant of this problem where one has to report only those colors such that at least a fraction $\alpha$ of the objects of that color intersects the query range, for some parameter $\alpha$. They obtain results for an approximate version of this problem, where they are also allowed to report those colors for which a fraction $(1 - \epsilon)\alpha$ intersects the query range, for some fixed $\epsilon > 0$. Data structures were built for an approximate version of orthogonal range searching queries and stabbing queries.
CHAPTER 5. RANGE AGGREGATE STRUCTURES FOR COLORED GEOMETRIC OBJECTS-I

- **Approximate Colored Range Queries**: In [65], the authors used sketching techniques to come up with approximate solutions for colored range query problems. In this way they could come up with more efficient solutions for these problems and also dynamize them.

5.3 **The colored weighted sum problem**

**Problem**: Preprocess a set $S$ of $n$ colored points in $\mathbb{R}^d$, where the points additionally come with a real-valued weight $w(p) \geq 0$, into a data structure such that given a $d$-dimensional orthogonal query box $q = \prod_{i=1}^{d}[a_i, b_i]$, the tuples $\langle c, s_c \rangle$ are reported where $s_c$ is sum of the weights of points of color $c$ in $q$.

We present a solution to the static $d$-dimensional orthogonal generalized weighted sum problem for $d \geq 2$ that takes $O(n^{1+\epsilon})$ space (for an arbitrarily small positive constant $\epsilon$) and $O(\log n + C)$ time. For $d = 1$, our solution takes $O(n \log n)$ space and $O(\log n + C)$ time.

5.3.1 **Query composition technique**

The number of colors for a given problem defined on a set of $n$ points, can range from 1 to $n$. We encode each color as an integer in the range $[1, n]$. This allows us to use colors as array indices. While answering a query for some of the problems in this chapter, we may sometimes encounter the same color more than once while querying data structures built on disjoint sets. The count corresponding to these partial results need to be added up efficiently. We can do this by using an array, $A[1:n]$, of counts indexed by colors (encoded as integers) to keep track of the count of each distinct color that is found during a query. While executing a query, we also store the distinct colors found in a linked list. This way we can avoid reading the zero counts in $A$ later. After the query, the counts can be output and $A$ can be reset in time proportional to the output size by scanning the list.
**Table 5.1** Summary of results. All results are “big-Oh” and worst case. Rectangles are axis-parallel. “three sided rectangle” means an orthogonal box of the form \([a_1, b_1] \times [a_2, \infty)\). The “nontrivial bounding box” function returns a bounding box of all those colors that have at least two points that intersect \(q\). A “bounding box” for a point set in \(\mathbb{R}^d\) is the hyper box with the smallest measure within which all the points lie. \(\epsilon\) is an arbitrarily small positive constant. \(C\) denotes the number of colors reported.

<table>
<thead>
<tr>
<th>Aggregate function</th>
<th>Colored objects in (S)</th>
<th>Query (q)</th>
<th>Ambient Space</th>
<th>Space</th>
<th>Query time</th>
<th>Theorem</th>
</tr>
</thead>
<tbody>
<tr>
<td>weighted sum</td>
<td>points</td>
<td>interval</td>
<td>(\mathbb{R}^1)</td>
<td>(O(n \log n))</td>
<td>(O(\log n + C))</td>
<td>5.3.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rectangle</td>
<td>(\mathbb{R}^2)</td>
<td>(O(n^{1+\epsilon}))</td>
<td>(O(\log n + C))</td>
<td>5.3.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hyper box</td>
<td>(\mathbb{R}^d)</td>
<td>(O(n^{1+\epsilon}))</td>
<td>(O(\log n + C))</td>
<td>5.3.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(d \geq 3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bounding box</td>
<td>points</td>
<td>interval</td>
<td>(\mathbb{R}^1)</td>
<td>(O(n))</td>
<td>(O(\log n + C))</td>
<td>5.4.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rectangle</td>
<td>(\mathbb{R}^2)</td>
<td>(O(n \log^2 n))</td>
<td>(O(\log n + C))</td>
<td>5.4.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hyper box</td>
<td>(\mathbb{R}^d)</td>
<td>(O(n^{1+\epsilon}))</td>
<td>(O(\log n + C))</td>
<td>5.4.3, 5.4.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(d \geq 3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nontrivial bounding box</td>
<td>points</td>
<td>quadrant</td>
<td>(\mathbb{R}^2)</td>
<td>(O(n \log n))</td>
<td>(O(\log n + C \log n))</td>
<td>5.5.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>three sided rectangle</td>
<td>(\mathbb{R}^2)</td>
<td>(O(n \log^2 n))</td>
<td>(O(\log^2 n + C \log n))</td>
<td>5.5.7</td>
</tr>
<tr>
<td>weighted sum</td>
<td>intervals</td>
<td>point</td>
<td>(\mathbb{R}^1)</td>
<td>(O(n))</td>
<td>(O(\log n + C))</td>
<td>5.6.1</td>
</tr>
<tr>
<td>weighted sum</td>
<td>rectangles</td>
<td>point</td>
<td>(\mathbb{R}^2)</td>
<td>(O(n \log n))</td>
<td>(O(\log^2 n + C \log n))</td>
<td>5.7.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(O(n^{1+\epsilon}))</td>
<td>(O(\log n + C))</td>
<td>5.7.2</td>
</tr>
<tr>
<td>weighted sum</td>
<td>orthogonal segments</td>
<td>rectangle</td>
<td>(\mathbb{R}^2)</td>
<td>(O(n^{1+\epsilon}))</td>
<td>(O(\log n + C))</td>
<td>5.8.1</td>
</tr>
</tbody>
</table>
5.3.2 The solution for $\mathbb{IR}^1$

First, we consider the semi-infinite problem for $d = 1$. We need to preprocess a set $S$ of $n$ weighted, colored points in $\mathbb{IR}^1$ (or the x-axis) into a data structure such that given a query interval $q = [a_1, \infty)$, the tuples $(c, s_c)$ are reported where $s_c$ is sum of the weights of color $c$ in $q$.

For each color $c$, we sort the points in $S$ by non-decreasing order of their $x$ coordinates. For each point $p \in S$ of color $c$, let $\text{pred}(p)$ be its predecessor in the sorted order, with $\text{pred}(p) = -\infty$ for the leftmost point. We then map the point $p$ to the point $p' = (p, \text{pred}(p))$ in $\mathbb{IR}^2$ and associate with it the color $c$ and weight $w(p')$ set to the cumulative weight of all the points of color $c$ in $S$ whose $x$-coordinate is greater than or equal to $p$. Let $S'$ be the set of such points in $\mathbb{IR}^2$. We preprocess the points in $S'$ into a priority search tree [69] $PST$ to support the query of reporting points in a quadrant. Given a query $q = [a_1, \infty)$, we map it to the quadrant $q' = [a_1, \infty) \times (-\infty, a_1)$ in $\mathbb{IR}^2$ and query $PST$ with $q'$. For each point $p'$ retrieved, we report $(c', w(p'))$ where $c'$ is the color of $p'$.

**Lemma 5.3.1.** The query algorithm reports a pair $(c, w)$ iff the total weight of the points of color $c$ in $S \cap q$ is $w$. Moreover at most one such pair is reported for each color $c$.

**Proof.** $\Rightarrow$ Let the query algorithm report a pair $(c, w)$. Then there exists a point $p' \in q'$ of color $c$ and weight $w$. Hence there exists a point $p \in q$ of color $c$ such that $\text{pred}(p) \in (-\infty, a_1)$ and the cumulative weight of all the points of color $c$ in $S$ whose $x$-coordinate is greater than or equal to $p$ is equal to $w$.

$\Leftarrow$ Let the total weight of the points of color $c$ in $S \cap q$ be $w$. Let $p$ be the leftmost point of color $c$ in $q$. Then there is a point $p' = (p, \text{pred}(p))$ of color $c$ and weight $w$ in $S'$. Since $p \in [a_1, \infty)$ and $\text{pred}(p) \in (-\infty, a_1)$, point $p'$ is retrieved by the query algorithm and the pair $(c, w)$ is reported.

To prove uniqueness, let the query algorithm report a pair $(c, w_1)$ corresponding to a point $p'_1 = (p_1, \text{pred}(p_1))$ and another pair $(c, w_2)$ corresponding to a point $p'_2 = (p_2, \text{pred}(p_2))$. Hence $p_1 \in q$, $p_2 \in q$, $\text{pred}(p_1) \in (-\infty, a_1)$ and $\text{pred}(p_2) \in (-\infty, a_1)$. Without loss of
CHAPTER 5. RANGE AGGREGATE STRUCTURES FOR COLORED GEOMETRIC OBJECTS

generality, let \( p_1 \) be to the left of \( p_2 \). Then, since \( p_1 \in q \) and \( p_1 \) is of color \( c \), \( \text{pred}(p_2) \in q \).

This contradicts the fact that \( \text{pred}(p_2) \in (-\infty, a_1) \). Hence for any color \( c \), at most one pair is reported.

**Theorem 5.3.1.** The colored weighted sum problem in \( \mathbb{R}^1 \) can be solved for semi-infinite queries using a structure of size \( O(n) \) and query time \( O(\log n + C) \).

**Extending to finite ranges**

We store the points of \( S \) at the leaves of a balanced binary search tree \( T \) in non-decreasing order of their \( x \) coordinates. At each internal node \( v \), we store an instance \( DL(v) \) of the data structure of Theorem 5.3.1 built on \( S(\text{Left}(v)) \), the set of points stored in the leaves of the left subtree of \( v \). Similarly we store another data structure \( DR(v) \) built on \( S(\text{Right}(v)) \), the set of points stored in the leaves of the right subtree of \( v \) supporting queries of the form \((\infty, b_1 \]) \). To answer a query \( q = [a_1, b_1] \) we search with \( a_1 \) and \( b_1 \) in \( T \). This generates paths \( \ell \) and \( r \) in \( T \) that possibly diverge at some non-leaf node \( v \) of \( T \). We query \( DL(v) \) (respectively \( DR(v) \)) with \([a_1, \infty) \) (resp., \((\infty, b_1) \)) to retrieve the partial results. The partial results are then composed using the technique of Section 5.3.1. We conclude:

**Theorem 5.3.2.** The colored weighted sum problem in \( \mathbb{R}^1 \) can be solved for bounded queries (i.e. \([a_1, b_1]\)) using a structure of size \( O(n \log n) \) and query time \( O(\log n + C) \).

**5.3.3 Adding range restrictions**

In [52] a general technique was proposed to add range restrictions to colored (or generalized) reporting problems. In this section we show how to adapt the technique to add range restrictions to our current problem. Adding range restrictions shall be defined in detail in some time. The idea used here is similar to [52], except that when we combine solutions to two sub problems, instead of taking an union of colors reported, we need to add up the
weights. Note that this is only possible if we decompose the problem in a way that the sub problems are defined on disjoint partitions of points.

Similar to [52], let \( PR(q, S) \) denote the answer to a colored weighted sum problem \( PR \) with query object \( q \) and object set \( S \). To add a range restriction to \( PR \), we give each object \( p \) in \( S \) an additional parameter \( k_p \in \mathbb{R} \). In the transformed searching problem, we only query objects in \( S \) that have their parameter in a given range.

**Adding a semi-infinite range restriction**

Let \( S \) be a set of \( n \) colored objects, and let \( PR(q, S) \) be a colored weighted sum problem for \( S \) with query object \( q \). To add a semi-infinite range restriction, we associate with each object \( p \) of \( S \) an additional parameter \( k_p \in \mathbb{R} \). Now, let \( TPR \) be the colored weighted sum problem that is obtained by adding a semi-infinite range restriction to \( PR \), i.e., \( TPR(q, [a, \infty), S) := PR(q, \{p \in S|a \leq k_p\}) \).

Assume we have a data structure \( DS \) that stores the set \( S \), such that colored weighted sum queries \( PR(q, S) \) can be solved in \( O(\log n + C) \) time. Let the size of \( DS \) be bounded by \( O(n^{1+\epsilon}) \), where \( \epsilon \) is an arbitrarily small positive constant. Also, assume we have a data structure \( TDS \) for the set \( S \), such that colored weighted sum queries \( TPR(q, [a, \infty), S) \) can be solved in \( O(\log n + C) \) time. Let the size of \( TDS \) be bounded by \( O(n^w) \) for some constant \( w > 1 \). Extending the idea of [52], we can show how to construct a data structure that solves colored weighted sum queries \( TPR(q, [a, \infty), S) \) in \( O(\log n + C) \) time, using \( O(n^{1+\epsilon}) \) space, for an arbitrarily small positive constant \( \epsilon \).

Let \( S = \{p_1, p_2, \ldots, p_n\} \), where \( k_{p_1} \geq k_{p_2} \geq \ldots \geq k_{p_n} \). Let \( m \) be an arbitrary parameter with \( 1 \leq m \leq n \). We assume for simplicity that \( n/m \) is an integer. Let \( S_j = \{p_1, p_2, \ldots, p_{jm}\} \) and \( S'_j = \{p_{jm+1}, p_{jm+2}, \ldots, p_{(j+1)m}\} \) for \( 0 \leq j \leq n/m \).

The transformed data structure consists of the following. For each \( j \) with \( 0 \leq j < n/m \), there is a data structure \( DS_j \) (of type \( DS \)) storing \( S_j \) for solving generalized weighted sum queries of the form \( PR(q, S_j) \) and a data structure \( TDS_j \) (of type \( TDS \)) storing \( S_j' \) for solving generalized weighted sum queries of the form \( TPR(q, [a, \infty), S_j') \).
CHAPTER 5. RANGE AGGREGATE STRUCTURES FOR COLORED GEOMETRIC OBJECTS-I

To answer a query $TPR(q, [a, \infty), S)$, we do the following. Compute the index $j$ such that $k_{p(j+1)m} < a \leq k_{pjm}$. Solve the query $PR(q, S_j)$ using $DS_j$, solve the query $TPR(q, [a, \infty), S'_j)$ using $TDS_j$, and output the union of the colors reported by these two queries. The correctness of the query algorithm is trivial to observe. Next we state a lemma which leads us to the main theorem.

**Lemma 5.3.2.** The basic transformation results in a data structure of size $O(n^{2+\epsilon}/m + nm^{w-1})$ and answers the colored weighted sum queries in $O(\log n + C)$ time.

**Theorem 5.3.3.** Let $S$, $DS$ and $TDS$ be as defined above. Then, there exists a data structure that solves colored weighted sum queries $TPR(q, [a, \infty), S)$

1. with a query time of $O(\log n + C),$

2. using $O(n^{1+\epsilon})$ space, for an arbitrarily small positive constant $\epsilon$.

**Proof.** First assume that $w > 2$, i.e., the data structure $TDS$ uses more than quadratic space. Choosing $m = n^{1/w}$ in Lemma 5.3.2 gives a data structure for solving queries $TPR(q, [a, \infty), S)$, with a query time of $O(\log n + C)$ and space $O(n^2)$. By applying Lemma 5.3.2 repeatedly, we obtain, for each integer constant $a \geq 1$, a data structure of size $O(n^{1+\epsilon+1/a})$ that answers queries $TPR(q, [a, \infty), S)$ in $O(\log n + C)$ time. This claim follows by induction on $a$; in the inductive step from $a$ to $a + 1$, we apply Lemma 5.3.2 with $m = n^{a/(a+1)}$. \qed

**Extending to a full range restriction**

Analogous to the result for colored reporting problems in [52], we can build a data structure $TPR(q, [a, b], S)$ that solves colored weighted sum queries. We store the points of $S$ at the leaves of a balanced binary search tree, sorted in non-decreasing order of their parameter $k_p$. For each non-root node $u$ of this tree, let $S_u$ denote the subset of $S$ that is stored at the leaves of $u$’s subtree. If $u$ is a left (resp., right) child, then we store at $u$, an instance of the data structure that solves colored weighted sum queries $TPR(q, [a, \infty), S_u)$ (resp.,
CHAPTER 5. RANGE AGGREGATE STRUCTURES FOR COLORED GEOMETRIC OBJECTS-I

$TPR(q, (-\infty, b], S_u)$). While querying, we search the tree for values $a$ and $b$. Let $u$ be the node at which the search paths diverge. Let $u_\ell$ (resp., $u_r$) be the left (resp., right) child of $u$. Then we query the secondary structure stored in the left (resp., right) child of $u$ with $TPR(q, [a, \infty), S_{u_\ell})$ (resp., $TPR(q, (-\infty, b], S_{u_r})$). For each color $c$, weights are retrieved at most once by the query on the secondary structure stored at $u_\ell$ and at most once by the query on the secondary structure stored at $u_r$. For any color that is in the answer for both the queries, the total weight of the points to be reported is the sum of the weights reported by the two queries for the same color; for any color which is the answer to exactly one of the queries, the weight for the color is reported as is. Other colors (having no points in $q$) are not reported at all. The fact that $S_{u_\ell}$ and $S_{u_r}$ are disjoint sets, ensures that the counts are correct.

**Corollary 5.3.1.** Let $DS$ and $TDS$ be as defined above. Then, there exists a data structure that solves colored weighted sum queries $TPR(q, [a, b], S)$

1. with a query time of $O(\log n + C)$,
2. using $O(n^{1+\epsilon})$ space, for an arbitrarily small positive constant $\epsilon$.

Theorem 5.3.3 and Corollary 5.3.1 imply that in order to solve the colored weighted sum problem $TPR$, it suffices to have (i) a data structure for $PR$ with $O(\log n + C)$ query time and $O(n^{1+\epsilon})$ space, and (ii) a data structure for $TPR$ with $O(\log n + C)$ query time and polynomial space.

### 5.3.4 Colored weighted sum problem for $d = 2$

In this section, we show how to solve the colored weighted sum problem for a query rectangle, $q = [a_1, b_1] \times [a_2, b_2]$.

We make use of Theorem 5.3.3 to solve the problem. Let $DS$ in Theorem 5.3.3 be the data structure of Theorem 5.3.2 for solving the colored weighted sum problem for $d = 1$ and queries of the form $q' = [a_2, b_2]$. We need a data structure $TDS$ to solve colored
weighted sum queries $TPR(q', [a_1, \infty), S)$. Given $n$ colored points in $\mathbb{R}^2$, we sort the points by their $x$-coordinates and rank the points in left to right order (ties broken arbitrarily) and store their $x$-coordinates in an auxiliary array $AUX$. We create data structures $DA_i$ for $1 \leq i \leq n$. Each such data structure is an instance of the data structure of Theorem 5.3.2 for the 1-dimensional static colored weighted sum problem which takes $O(n \log n)$ space and supports queries in time $O(\log n + C)$. We build data structure $DA_i$ on the $y$-coordinates of the points $p$ whose $x$-coordinates are at least $AUX[i]$. Given a query $TPR(q', [a_1, \infty), S)$, we first binary search in $AUX$ with $a_1$ to determine the index $i$ of the leftmost point whose $x$-coordinate is greater than or equal to $a_1$. Then we simply query $DA_i$ with $q'$.

**Lemma 5.3.3.** The colored weighted sum problem in $\mathbb{R}^2$ can be solved for query $TPR(q', [a_1, \infty), S)$ using $O(n^2 \log n)$ space and $O(\log n + C)$ query time.

**Proof.** The correctness follows from the correctness of Theorem 5.3.2 and the fact that $DA_i$ is really built on all points in $S$ with $x$-coordinates in $[a_1, \infty)$. Since each data structure $DA_i$ takes space $O(n \log n)$ and there are $O(n)$ of them, the total space taken is $O(n^2 \log n)$. The query time is $O(\log n + C)$, since it takes $O(\log n)$ time to determine the index $i$ and $O(\log n + C)$ time to query $DA_i$. \qed

Now since we have both the structures $DS$ and $TDS$, we can apply Theorem 5.3.3 with $w = 3$, and Corollary 5.3.1 to conclude:

**Theorem 5.3.4.** The colored weighted sum problem in $\mathbb{R}^2$ for orthogonal rectangular queries can be solved using $O(n^{1+\epsilon})$ space and $O(\log n + C)$ query time.

### 5.3.5 Colored weighted sum problem for $d > 2$

To solve the problem in dimension $d > 2$ we again make use of Theorem 5.3.3. Assume that as $DS$, we have the data structure to solve the problem $PR$ in dimension $d - 1$, which takes $O(n^{1+\epsilon})$ space and $O(\log n + C)$ time. As $TDS$, we create a structure similar
to Lemma 5.3.3, by taking \( O(n) \) instances of \( DS \), which gives us a data structure with \( O(n^{2+\epsilon}) \) space and \( O(\log n + C) \) query time. Now we apply Theorem 5.3.3, with \( w = 3 \) and Corollary 5.3.1 to conclude:

**Theorem 5.3.5.** The colored weighted sum problem in \( \mathbb{R}^d \) can be solved for orthogonal query box using \( O(n^{1+\epsilon}) \) space and \( O(\log n + C) \) query time.

### 5.4 The Colored Bounding Box Problem

**Problem:** Preprocess a set \( S \) of \( n \) colored points in \( \mathbb{R}^d \) into a data structure such that given an orthogonal query box \( q \), the tuples \( \langle c, bb_c \rangle \) are reported where \( bb_c \) is the bounding box of all the points of color \( c \) which lie within \( q \). If a color has a single point \( p \) inside \( q \), then the bounding box of that color will be the point \( p \) which is reported as a degenerate box.

In subsection 5.4.1 a fully dynamic solution to this problem is presented on the real line. Then in subsection 5.4.2 a static solution to the problem is found on a plane. Next we give a solution to the static version of the problem in \( \mathbb{R}^d \) for \( d \geq 3 \) that takes \( O(n^{1+\epsilon}) \) space (for an arbitrarily small positive constant \( \epsilon \)) and \( O(\log n + C) \) time.

### 5.4.1 A solution in \( \mathbb{R}^1 \)

On the real line the problem reduces to finding, for each color \( c \) that has at least one point inside the query interval \( q \), the interval spanned by all the points of color \( c \) lying inside \( q \).

For each color \( c \), sort all the points in \( S \) by non-decreasing order of their \( x \)-coordinates and build a balanced binary search tree \( T_c \). For each point \( p \in S \) of color \( c \), let \( \text{pred}(p) \) and \( \text{succ}(p) \) be its predecessor and successor in the sorted order, with \( \text{pred}(p) = -\infty \) for the leftmost point and \( \text{succ}(p) = \infty \) for the rightmost point. Then each point \( p \) is mapped to a new point \( p' = (p, \text{pred}(p)) \) (and \( p'' = (p, \text{succ}(p)) \)) in \( \mathbb{R}^2 \) and \( p' \) (and \( p'' \)) is assigned the color of point \( p \). Call this set of points in \( \mathbb{R}^2 \), \( S' \) (and \( S'' \)). We build a dynamic priority search tree \( D' \) (and \( D'' \)) [69] based on the points in \( S' \) (and \( S'' \)). Given a query \( q = [a_1, b_1] \),
we map it to \( q' = [a_1, b_1] \times (-\infty, a_1) \) (and \( q'' = [a_1, b_1] \times (b_1, \infty) \)) in \( \mathbb{R}^2 \) and query \( D' \) (and \( D'' \)) with \( q' \) (and \( q'' \)). If a point \( p' = (p, \text{pred}(p)) \) (or \( p'' = (p, \text{succ}(p)) \)) gets reported and \( p' \) (or \( p'' \)) has color \( c \), then it can be inferred that among all the points of color \( c \) lying inside \( q \), the leftmost point (or the rightmost point) is \( p \). We state this formally in the following lemma.

**Lemma 5.4.1.** The following statements are observed from the discussion above:

1. A point \( p' = (p, \text{pred}(p)) \) having color \( c \) is reported by \( D' \) iff \( p \) happens to be the leftmost point among all the points of color \( c \) that lie inside the query interval \( q \). Also for each color, exactly one point is reported by \( D' \).

2. A point \( p'' = (p, \text{succ}(p)) \) having color \( c \) is reported by \( D'' \) iff \( p \) happens to be the rightmost point among all the points of color \( c \) that lie inside the query interval \( q \). Also for each color, exactly one point is reported by \( D'' \).

**Proof.** Consider the first part of the lemma. Let a point \( p' = (p, \text{pred}(p)) \) of color \( c \) be reported. If \( p \) is not the leftmost point then \( \text{pred}(p) \in [a_1, b_1] \). Since \( D' \) has reported point \( p' \), \( \text{pred}(p) \) should lie in the interval \((−\infty, a_1] \). A contradiction arises.

Let \( p \) be the leftmost point among all the points of color \( c \) that lie inside \( q = [a_1, b_1] \). So, \( \text{pred}(p) \) should lie to the left of the query interval \( q \), i.e., in the interval \((−\infty, a_1] \). Therefore, point \( p' \) is reported by \( D' \). From the above discussion it is clear that exactly one point is reported for each color by \( D' \). The second part of the lemma follows a symmetric argument.

A dynamic priority search tree build on \( m \) points takes \( O(m) \) space and answers three-sided rectangular queries in \( O(\log m + k) \) time. The size of set \( S' \) and \( S'' \) is \( n \) and from the above lemma, it is clear that \( k = C \).

Next we show how the solution can be made dynamic. Let \( r \) be the new point having color \( c \) which is to be inserted. First we insert \( r \) into \( T_c \). Let \( r_p \) and \( r_s \) be the points in the leaf nodes to the immediate left and to the immediate right of \( r \), respectively. Before
the insertion of \( r, r_p = \text{pred}(r_s) \) and \( r_s = \text{succ}(r_p) \) were valid relations. After insertion of \( r \), the following new relations arise: \( r_p = \text{pred}(r) \), \( r = \text{pred}(r_s) \), \( r = \text{succ}(r_p) \) and \( r_s = \text{succ}(r) \). To represent these changes in our data structures we delete \((r_s, r_p)\) from \( D'\) and delete \((r_p, r_s)\) from \( D''\). Then we insert \((r, r_p)\) and \((r_s, r)\) into \( D'\), and insert \((r_p, r)\) and \((r, r_s)\) into \( D''\). The total time taken for handling these operations is \( O(\log n) \). Deletions are symmetric to insertions. Therefore, insertions and deletions can be handled in \( O(\log n) \) time.

**Theorem 5.4.1.** The colored bounding box problem in \( \mathbb{R}^1 \) can be solved using a structure of size \( O(n) \) and query time \( O(\log n + C) \). Also, insertion or deletion of a point can be handled in \( O(\log n) \) time.

### 5.4.2 Extending to \( \mathbb{R}^2 \)

In this subsection, the solution developed for one-dimensional case is combined with the technique of *persistence* to build a solution in \( \mathbb{R}^2 \).

For a general rectangle \( r = [x, x'] \times [y, y'] \) is defined to be the *x-projection* and \([y, y']\) is defined to be the *y-projection* of \( r \). Now given a query \( q \), each color \( c \) having at least one point in \( q \) is defined as a *valid color*. Reporting of the bounding box, \( BB_c \), for each *valid color* \( c \) is done by first finding out the *x-projection* of \( BB_c \) and then the *y-projection* of \( BB_c \).

First the *x-projection’s* of all the *valid colors* are found out. We do this by initially considering a query region of the form \( q = [a_1, b_1] \times [a_2, \infty) \). Using the technique of persistence described in [34], a partially persistent version of the data structure of Lemma 5.4.1 is built, by treating the *y-coordinate* as time and inserting the points by non-increasing *y-coordinate* into an initially empty data structure. In fact only \( D' \) and \( D'' \) needs to be made persistent. The trees \( T_c \) are only needed to do updates efficiently in the current version. While querying they are not needed and can be discarded once the persistent version of \( D' \) and \( D'' \) have been built. To answer the query \( q = [a_1, b_1] \times [a_2, \infty) \), we access the version
corresponding to the smallest $y$-coordinate greater than or equal to $a_2$ and query it with $[a_1, b_1]$.

**Lemma 5.4.2.** A set $S$ of $n$ colored points in $\mathbb{R}^2$ can be preprocessed into a data structure of size $O(n \log n)$, such that given a query $q = [a_1, b_1] \times [a_2, \infty)$, the $x$-projection of each valid color is reported in $O(\log n + C)$ time.

**Proof.** The structures of $D'$ and $D''$ of Lemma 5.4.1 have a constant in-degree and so the results of [34] apply. The correctness of the query algorithm follows from the fact that since the version accessed does not contain any point having $y$-coordinate less than $a_2$, querying $D'$ and $D''$ with $q = [a_1, b_1] \times [a_2, \infty)$ is same as querying with $[a_1, b_1]$. The query time follows from Lemma 5.4.1. To build the persistent structure we do $n$ insertions, each of which does $O(n \log n)$ memory modifications. Thus the persistent structure uses $O(n \log n)$ space. □

To extend the solution in the above lemma for query boxes, $q = [a_1, b_1] \times [a_2, b_2]$, we follow an approach similar to that described previously: The points are stored in a balanced binary tree sorted by $y$-coordinates. Each internal node $v$ in the tree is associated with the auxiliary structure of Lemma 5.4.2 for answering queries of the form $[a_1, b_1] \times [a_2, \infty)$ (resp. $[a_1, b_1] \times (-\infty, b_2)$) on the points in the leaves of $v$’s left (resp. right) subtree. To perform a query this tree is searched using the interval $[a_2, b_2]$. The query time remains $O(\log n + C)$ but the space increases by a logarithmic factor. The $y$-projections can be found in a similar way by switching the role of $x$ and $y$. Next we summarize the above discussion in the following lemma which leads us to the final result.

**Lemma 5.4.3.** A set $S$ of $n$ colored points in $\mathbb{R}^2$ can be preprocessed into a data structure of size $O(n \log^2 n)$, such that given a query $q = [a_1, b_1] \times [a_2, b_2]$, the $x$-projection (or $y$-projection) of each valid color is reported in $O(\log n + C)$ time.

**Theorem 5.4.2.** The colored bounding box problem in $\mathbb{R}^2$ can be solved using a structure of size $O(n \log^2 n)$ and query time $O(\log n + C)$. 75
5.4.3 A solution for \( \mathbb{R}^3 \)

In this section we consider points in a 3-dimensional space \((XYZ)\) and solve the colored bounding box problem for a query cuboid \( q = [a_1, b_1] \times [a_2, b_2] \times [a_3, b_3] \). The solution is obtained by applying Theorem 5.3.3. Let \( DS \) in Theorem 5.3.3 be a data structure of Theorem 5.4.2 for colored bounding box query in the \( XY \)-plane. A data structure \( TDS \) needs to be built for finding the \( x \)-projection’s and the \( y \)-projection’s of all the valid colors for queries of the form \( TPR(q', [a_3, \infty), S) \), where \( q' = [a_1, b_1] \times [a_2, b_2] \). Note that \( z \)-projection’s won’t be found out by \( TDS \). Given \( n \) colored points in \( \mathbb{R}^3 \), we sort the points by their \( z \)-coordinates and rank the points in left to right order (ties broken arbitrarily) and store the \( z \)-coordinates in an auxiliary array \( AUX \). Data structures \( DA_i \) for \( 1 \leq i \leq n \) are created. Each \( DA_i \) is an instance of the data structure of Theorem 5.4.2 for the 2-dimensional static colored bounding box problem which takes \( O(n \log^2 n) \) space and answers queries in time \( O(\log n + C) \). We build data structure \( DA_i \) on the \( x \) and \( y \) coordinates of the points in \( S \) whose \( z \)-coordinates are at least \( AUX[i] \). Given a query \( TPR(q', [a_3, \infty), S) \), we first binary search in \( AUX \) with \( a_3 \) to determine the index \( i \) of the leftmost point whose \( z \)-coordinate is greater than or equal to \( a_3 \). Then \( DA_i \) is simply queried with \( q' \).

Lemma 5.4.4. A set of \( n \) colored points in \( \mathbb{R}^3 \) can be preprocessed into a data structure of size \( O(n^2 \log^4 n) \) such that given a query \( TPR(q', [a_3, \infty), S) \) the \( x \)-projection and the \( y \)-projection of each valid color is reported in \( O(\log n + C) \) time. Here \( q' = [a_1, b_1] \times [a_2, b_2] \).

Proof. The correctness follows from the correctness of the solution to the 2-dimensional problem in Theorem 5.4.2 and the fact that \( DA_i \) is really built on all points in \( S \) with \( z \)-coordinates in \([a_3, \infty)\). Since each data structure \( DA_i \) takes space \( O(n \log^2 n) \) and there are \( O(n) \) of them, the total space taken is \( O(n^2 \log^4 n) \). The query time is \( O(\log n + C) \), since it takes \( O(\log n) \) time to determine the index \( i \) and \( O(\log n + C) \) time to query \( DA_i \).

Now since we have both the structures \( DS \) and \( TDS \), we can apply Theorem 5.3.3 with \( w = 3 \), and Corollary 5.3.1 to reach the following lemma:
CHAPTER 5. RANGE AGGREGATE STRUCTURES FOR COLORED GEOMETRIC OBJECTS-I

Lemma 5.4.5. A set of $n$ colored points in $\mathbb{R}^3$ can be preprocessed into a data structure of size $O(n^{1+\epsilon})$, for an arbitrarily small positive constant $\epsilon$, such that given a query cuboid $q' = [a_1, b_1] \times [a_2, b_2] \times [a_3, b_3]$, the $x$-projection and the $y$-projection of each valid color is reported in $O(\log n + C)$ time where $C$ is the output size.

The $z$-projection can be found by switching its role with $x$ or $y$. This leads us to the following result.

Theorem 5.4.3. The colored bounding box problem in $\mathbb{R}^3$ can be solved using a structure of size $O(n^{1+\epsilon})$ and query time $O(\log n + C)$.

5.4.4 A solution for $d \geq 3$

In order to solve the problem in dimension $d \geq 3$, first consider a $(d - 1)$-dimensional subspace of $\mathbb{R}^d$. Call it $\Delta$ and let $d_i$ be the dimension excluded from $\Delta$. Let $DS$ be a data structure to solve the problem $PR$ in the dimensional subspace $\Delta$ which takes $O(n^{1+\epsilon})$ space and $O(\log n + C)$ time. As $TDS$, we create a structure similar to Lemma 5.4.4, by sorting points based on their values in dimension $d_i$ and then taking $O(n)$ instances of $DS$, which gives us a data structure with $O(n^{2+\epsilon})$ space and $O(\log n + C)$ query time. Next another $(d-1)$-dimensional subspace, $\Delta'$ of $\mathbb{R}^d$ is considered, such that $\Delta' \neq \Delta$. Again $DS$ is built to solve $PR$ in subspace $\Delta'$. $TDS$ is also built on similar lines. Using the structures for these two distinct subspaces we can find out for each valid color, the projection of its bounding box in each dimension. Now we apply Theorem 5.3.3, with $w = 3$ and Corollary 5.3.1 to conclude:

Theorem 5.4.4. The colored bounding box problem in $\mathbb{R}^d$ can be solved using a structure of size $O(n^{1+\epsilon})$ and query time $O(\log n + C)$. 
5.5 Nontrivial colored bounding box problem

**Problem:** Preprocess a set $S$ of $n$ colored points in $\mathbb{R}^2$ into a data structure such that given an orthogonal query box $q$, the tuples $\langle c, bb_c \rangle$ are reported where $bb_c$ is the bounding box of all the points of color $c$ which lie within $q$ and there is an additional constraint that color $c$ should have at least two points lying within $q$.

In section 5.5.1 the problem is solved for the case when the query region $q$ is a quadrant. In section 5.5.2, a query of the form $[a_1, b_1] \times [a_2, \infty)$ is considered.

5.5.1 Querying with quadrants in $\mathbb{R}^2$

Given a set, $S$, of $n$ colored points in $\mathbb{R}^2$, the nontrivial colored bounding box query is solved for a north-east quadrant defined by a query point $q = (a_1, a_2)$. The north-east quadrant, $NE(q)$, is defined as the set of all points $(x, y)$ such that $x \geq a_1$ and $y \geq a_2$.

The solution to this problem is found in two stages. In the first stage, all the colors $c$ that have at least two points of color $c$ in $NE(q)$ are found out. Then for each such color $c$, the bounding box of all the points of color $c$ which lie inside $q$ is reported.

The details of the first stage are stated next. Let $S_c$ be the set of points of color $c$ and $M_c$ be the set of maximal points of $S_c$, i.e., $M_c \subseteq S_c$ consists of points which do not contain any other point of $S_c$ in their north-east quadrant. Define $S'_c = S_c \setminus M_c$ and $MM_c$ to be the set of maximal points of $S'_c$. Next we state a lemma.

**Lemma 5.5.1.** There are at least two points of color $c$ in $NE(q)$, if and only if, either a) at least two points of $M_c$ and none of the points of $MM_c$ lie in $NE(q)$, or b) at least one point of $M_c$ and at least one point of $MM_c$ lie in $NE(q)$.

**Proof.** We classify set $S_c$ into three categories: $M_c$, $MM_c$ and $R_c = S'_c \setminus MM_c$. Let there be at least two points of color $c$ in $NE(q)$. Two possibilities arise:

1. If any of these points belong to $R_c$, then by definition of a maximal point, at least one point of $MM_c$ and one point of $M_c$ should also lie in the $NE(q)$. This satisfies
CHAPTER 5. RANGE AGGREGATE STRUCTURES FOR COLORED GEOMETRIC OBJECTS-I

condition b.

2. If none of these points which lie in $NE(q)$ are from $R_c$, then they need to belong to $M_c$ and $MM_c$ only. None of the points belonging to $M_c$ and all the points belonging to $MM_c$ is an impossible condition (fundamental property of maximal points). The remaining valid conditions are, both $M_c$ and $MM_c$ having at least one point (condition b) or $M_c$ having all the points (condition a).

The converse is quite trivial to prove. \(\Box\)

Based on the above lemma, two data structures are built. The first data structure, $D_1$, reports all those colors of $S$ which partially satisfy condition (a) of the above lemma (Lemma 5.5.1). $D_1$ is built on the points in $M_c$, for each color $c$, such that given a query point $q$, we shall report all colors $c$ which have at least two points of color $c$ in $NE(q)$. Therefore, colors reported by $D_1$ might have some points of $MM_c$ lying in $NE(q)$ but that will not concern us, as each color which gets reported will still be having at least two points in $NE(q)$.

The second one, $D_2$, is a data structure which is built on the points in $MM_c$, for each color $c$, so that given a query point $q$, the distinct colors of all the points lying in $NE(q)$ are reported. It must be noted that if a color $c$ has at least one point of $MM_c$ in $NE(q)$, then it will also have at least one point of $M_c$ in $NE(q)$. So, indirectly $D_2$ reports all the colors that satisfy condition b) of Lemma 5.5.1. The answers obtained by querying $D_1$ and $D_2$ are combined to find out all the colors which have at least two points in $NE(q)$. A color will get reported at most twice (once by $D_1$ and once by $D_2$).

**Description of data structure $D_2$**

Define $S' = \bigcup MM_c$, for all colors $c$. Based on the points $S'$ lying in the plane, a data structure $D_2$ is built on these points such that all the distinct colors of the points that lie in the north-east quadrant get reported.
CHAPTER 5. RANGE AGGREGATE STRUCTURES FOR COLORED GEOMETRIC OBJECTS-I

For a point \( p(p_1, p_2) \in MM_c \), denote by \( Q_p = (-\infty, p_1] \times (-\infty, p_2] \subseteq \mathbb{R}^2 \), the region within which query point \( q \) should lie for point \( p \) to lie in the \( NE(q) \). Let \( R(MM_c) = \bigcup_{p \in MM_c} Q_p \). \( R_c \) is the set of pairwise disjoint axis-parallel rectangles obtained by decomposing \( R(MM_c) \). If a color \( c \) has at least one point of \( MM_c \) in the north-east quadrant of \( q \), then \( q \) should lie inside exactly one of the rectangle in \( R_c \). For each color \( c \), rectangles \( R_c \) are built and a standard data structure for reporting all the rectangles containing a query point \( q \in \mathbb{R}^2 \) is built. For \( m \) axis-parallel rectangles in a plane, this data structure takes \( O(m) \) space and given a query point \( q \) the query is reported in \( O(\log m + k) \) time \([24]\). In \([61]\) it is proved that \( R(MM_c) \) built over points in \( MM_c \) can be decomposed into \( O(|MM_c|) \) pairwise disjoint axis-parallel rectangles. So, the total number of rectangles obtained on decomposing \( R(MM_c) \), for each color \( c \), will be \( O(n) \). Also, each color \( c \) is reported only once, i.e., \( k = C \). Therefore, \( D_2 \) will take \( O(n) \) space and \( O(\log n + C) \) time.

**Theorem 5.5.1.** For all colors \( c \), let \( MM_c \) be the second layer of maximal set of points of color \( c \) in the plane. Then there exists a data structure \( D_2 \) of size \( O(n) \) such that given a query point \( q \), it reports the \( C \) distinct colors of the points that lie in northeast quadrant of \( q \) in \( (\log n + C) \) time.

**Note:** Incidentally, Theorem 5.5.1 can be used to solve the following problem: A set \( S \) of \( n \) colored points lie in a plane. Given a query quadrant \( q=[a_1, \infty) \times [a_2, \infty) \), report the distinct colors of the points that lie in \( q \). For each color find out its maximal points in the plane. Then apply the above theorem can be applied on the maximal points obtained for each color to solve the problem. Theorem 5.5.1 provides an optimal solution for this problem. The previously known solution to this problem used linear space and \( O(\log^2 n + k) \) query time \([49]\).
CHAPTER 5. RANGE AGGREGATE STRUCTURES FOR COLORED GEOMETRIC OBJECTS-I

Description of data structure $D_1$

Based on the set of points $M_c$, for all colors $c$, we build our data structure $D_1$, which will report all the colors $c$ having at least two points of $M_c$ in the $NE(q)$.

Fix a color $c$. Let the points in $M_c$ be sorted in decreasing order based on their $y$-coordinates. With $p_i(x_i, y_i) \in M_c$, $\forall 1 \leq i \leq |M_c|$, we associate an axis-parallel rectangle $R(p_i) \subset \mathbb{R}^2$ which might be unbounded on a few sides. Define $R(p_i) = (-\infty, x_i) \times (-\infty, y_2); R(p_i) = (x_{i-1}, x_i) \times (-\infty, y_{i+1})$, $\forall 2 \leq i \leq |M_c| - 1$ and $R(p_{|M_c|}) = \emptyset$. The union of $R(p_i), \forall 1 \leq i \leq |M_c|$, is denoted by $R(M_c)$ and all the $R(p_i)$’s are disjoint to each other.

**Lemma 5.5.2.** At least two points of $M_c$ will lie in the $NE$ quadrant of $q$ if and only if the query point lies within $R(M_c)$.

**Proof.** Suppose that there are at least two points of $M_c$ which lie in the $NE$ quadrant of $q(a_1, a_2)$. Without loss of generality, let the $x$-coordinate of $q$ (i.e $a_1$) lie in the interval $(x_{i-1}, x_i)$. If $a_2 \geq y_i$, then none of the points of $M_c$ can lie in the $NE(q)$. If $y_{i+1} < a_2 \leq y_i$, then only $p_i$ lies in the $NE(q)$. So, $b$ must lie in the interval $(-\infty, y_{i+1}]$ for at least two points of $M_c$ to get reported. Thus in this case $q$ will have to lie within the region $R(p_i)$. As $R(M_c)$ is the union of $R(p_i), \forall 1 \leq i \leq |M_c|$, $q$ has to lie within $R(M_c)$.

Conversely, suppose that $q(a_1, a_2)$ lies within $R(M_c)$ and let $a_1$ lie in the interval $(x_{i-1}, x_i)$. If $a_2$ lies in the interval $(y_{i+2}, y_{i+1})$, then points $p_i$ and $p_i+1$ get reported. If $a_2$ lies in the interval $(y_{i+3}, y_{i+2})$, then points $p_i$, $p_{i+1}$ and $p_{i+2}$ get reported and so on. Therefore, at least two points of $M_c$ will lie in the $NE(q)$.

Now, we need to solve the standard point enclosure problem of reporting all the axis-parallel rectangles $R(p_i) \in M_c, \forall 1 \leq i \leq |M_c|$, for all colors $c$, which contain the query point $q(a_1, a_2)$. Note that for each color at most one rectangle will get reported. The total size of $R(p_i) \in M_c, \forall 1 \leq i \leq |M_c|$, for all colors $c$ is $O(n)$. For $m$ axis-parallel rectangles in a plane, the data structure which solves point enclosure problem takes $O(m)$ space and
given a query point \( q \) the query is reported in \( O(\log m + k) \) time [24]. Therefore, data structure \( D_1 \) takes \( O(n) \) space and \( O(\log n + C) \) time since \( k = C \).

**Theorem 5.5.2.** For all colors \( c \), let \( M_c \) be a maximal set of points of color \( c \) in the plane. Then there exists a data structure \( D_1 \) of size \( O(n) \) such that given a query point \( q \), it reports \( C \) distinct colors, such that each reported color \( c \) has at least two points of color \( c \) in northeast quadrant of \( q \), in \( (\log n + C) \) time.

Based on the two conditions stated in Lemma 5.5.1, data structures \( D_1 \) (Theorem 5.5.2) and \( D_2 \) (Theorem 5.5.1) have been built. By combining the results obtained by querying these two data structures, the following result is obtained.

**Theorem 5.5.3.** Let \( S \) be a set of colored points in the plane. Then there exists a data structure of size \( O(n) \) such that given a query point \( q \), it reports \( C \) distinct colors, such that each reported color \( c \) has at least two points of color \( c \) in northeast quadrant of \( q \), in \( (\log n + C) \) time.

**Finding the bounding box**

Let \( S_c \) be the set of points of color \( c \) and the query region \( q \) be a rectangle. Given that \( |S_c \cap q| \geq 2 \), the problem of the finding the bounding box (\( BB_c \)) of the points in \( S_c \cap q \) is discussed in this section. Note that a quadrant query is a special case of a query rectangle.

Let \( S_c \) be the set of points of color \( c \). Two data structures \( T_{xy}^c \) and \( T_{yx}^c \) are constructed based on the points in \( S_c \). Given that \( |S_c \cap q| \geq 2 \), \( T_{xy}^c \) reports two points from \( S_c \cap q \), one with the minimum y-coordinate and one with the maximum y-coordinate; \( T_{yx}^c \) reports two points from \( S_c \cap q \), one with the minimum x-coordinate and one with the maximum x-coordinate. The four values reported by these two data structures are used to find the bounding box of the points in \( S_c \cap q \).

\( T_{xy}^c \) is actually a 2-dimensional range tree where the primary structure is built on the x-coordinates of the points in \( S_c \) and the associated structure is built on the y-coordinates of the points in \( S_c \). The technique of fractional cascading [25] is used to build the associated
structures of $T_{xy}^c$. $T_{yx}^c$ is the same as $T_{xy}^c$ except that the primary structure here is built on the y-coordinates of the points in $S_c$ and the associated structure is built on the x-coordinates of the points in $S_c$. Given a query rectangle $q = [a_1, b_1] \times [a_2, b_2]$, the primary structure of $T_{xy}^c$ is searched with $a_1$ and $b_1$ to determine set $V$ of nodes whose canonical subsets together contain the points with x-coordinate in the range $[a_1, b_1]$. For each node $v \in V$, $A(v)$ is the set of points associated with the node $v$ and is sorted based on the y-coordinates of the points. Let $L(v) \subseteq A(v)$ be the set of points which lie in the interval $[a_2, b_2]$. $v_{\text{min}}$ and $v_{\text{max}}$, the points with the minimum and the maximum y-coordinate values in $L(v)$ are picked. Define $y_c = \min\{v_{\text{min}}, \forall v \in V\}$ and $y'_c = \max\{v_{\text{max}}, \forall v \in V\}$. In the same manner $T_{yx}^c$ is queried and the variables $x_c$ and $x'_c$ are found out analogously. The bounding box of color $c$, $BB_c$, then turns out to be $[x_c, x'_c] \times [y_c, y'_c]$.

**Theorem 5.5.4.** Let $S_c$ be a set of $n_c$ points of color $c$ and the query region $q$ be a rectangle. Given that $|S_c \cap q| \geq 2$, the data structure for finding the bounding box $(BB_c)$ of the points in $S_c \cap q$ takes $O(n_c \log n_c)$ space and answers the query in $O(\log n_c)$ time.

Putting together Theorem 5.5.3 and Theorem 5.5.4 we obtain the following result.

**Theorem 5.5.5.** The nontrivial colored bounding box problem in $\mathbb{R}^2$ can be solved for a query quadrant (i.e. $[a_1, \infty) \times [a_2, \infty)$) using a structure of size $O(n \log n)$ and query time $O(\log n + C \log n)$.

### 5.5.2 Querying with three sided rectangle in $\mathbb{R}^2$

Given a set, $S$, of $n$ colored points in $\mathbb{R}^2$, the colored bounding box query is solved for the query range $q = [a_1, b_1] \times [a_2, \infty)$. As done before, the solution is found in two stages. In the first stage all the colors $c$ which have at least two points inside $q$ are reported. Then for each color $c$, the corresponding bounding box $(BB_c)$ is found out.

The details of the first stage is discussed next. The points of $S$ are stored in sorted order by x-coordinate at the leaves of a height balanced binary tree $T$. At each internal node $v$,
an instance of the structure of Theorem 5.5.3 for NE-queries (resp., NW-queries) built on the points in the leaves of v’s left (resp., right) subtree. Call them $T_{ne}$ and $T_{nw}$. Let $X(v)$ denote the average of the x-coordinate in the rightmost leaf in v’s left subtree and the x-coordinate in the leftmost leaf in v’s right subtree; for a leaf v, we take $X(v)$ to be the x-coordinate of the point stored at v. Let $L(v)$ and $R(v)$ be the regions to the left and the right of the line $x = X(v)$.

Two more structures $T_{vl}$ and $T_{vr}$ are built at node v, which together report all the colors $c$ which have point(s) in the intersection of $q$ and $L(v)$, and also have point(s) in the intersection of $q$ and $R(v)$. Fix a color $c$. For the points having color $c$ in v’s left subtree, $M_{vl}^c$ is the set of maximal points. Here a point $p$ becomes a maximal point if there is no other point in the set lying in the NE-quadrant of $p$. Similarly, for the points having color $c$ in v’s right subtree, $M_{vr}^c$ is the set of maximal points. Here a point $p$ becomes a maximal point if there is no other point that lies in the NW-quadrant of $p$. $M_{vl}^c$ and $M_{vr}^c$ are sorted in decreasing order of their y-coordinates. For every point $p(p_x, p_y) \in M_{vl}^c$ (or $M_{vr}^c$), $p_y’$ is the y-coordinate of the next point to $p$ in $M_{vl}^c$ (or $M_{vr}^c$); and $p_x’$ is the x-coordinate of the point in $M_{vl}^c$ (or $M_{vr}^c$) which has the least y-coordinate among the points having their y-coordinate values greater than $p_y$. Now each point $p \in M_{vl}^c$ in $\mathbb{R}^2$ is transformed to a point $p’(p_x, p_y, p_x’, p_y’)$ in $\mathbb{R}^4$. Let $M_{vl}’$ be the set of transformed points. $T_{vl}$ is a data structure which can handle 4-dimensional dominance query and is built over points in $M_{vl}’$, for all colors $c$. Similarly, we transform each point $p \in M_{vr}^c$ to build $T_{vr}$.

To perform a query $q$, we do a binary search down $T$, using $[a_1, b_1]$, until either the search runs of $T$ or a (highest) node $v$ is reached such that $[a_1, b_1]$ intersects $X(v)$. In the former case, we stop. In the latter case, if $v$ is a leaf then we stop. If $v$ is a non-leaf, then we query the structures $T_{ne}$ and $T_{nw}$ at $v$ using the NE-quadrant and the NW-quadrant derived from $q$ (i.e., the quadrants with corners at $(a_1, a_2)$ and $(b_1, a_2)$, respectively). Structure $T_{vl}$ is queried with $q_1 = [a_1, \infty) \times [a_2, \infty) \times (-\infty, b_1] \times (-\infty, a_2]$ and $T_{vr}$ with $q_2 = (-\infty, b_1] \times [a_2, \infty) \times [a_1, \infty) \times (-\infty, a_2]$. The answers obtained are combined to find out all the colors $c$ which have at least two points inside $q$. 84
Structure $T_{ne}$ (and $T_{nw}$) reports all the colors which have at least two points in the NE-quadrant of $[a_1, a_2]$ (and NW-quadrant of $[b_1, a_2]$). All the colors which have at least one point in the NE-quadrant of $[a_1, a_2]$ and NW-quadrant of $[b_1, a_2]$ are reported by $T_{vl}$ and $T_{vr}$. It must be observed that $T_{vl}$ (and $T_{vr}$) report each color only once. Also, it is clear that no color is missed, although a single color might get reported a constant number of times.

A recent result by Afshani [2] solves 4-dimensional dominance queries using $O(n \log n)$ space and $O(\log^2 n + k)$ time. So, the total space occupied by our data structure will be $O(n \log^2 n)$ and the time to answer a query will be $O(\log^2 n + C)$. The result is summarized below.

**Theorem 5.5.6.** Let $S$ be a set of colored points in the plane. Then there exists a data structure of size $O(n \log^2 n)$ such that given a query $q = [a_1, b_1] \times [a_2, \infty)$, it reports $C$ distinct colors, such that each reported color $c$ has at least two points of color $c$ inside $q$, in $(\log^2 n + C)$ time.

In the case of a query quadrant once the appropriate colors were known, next the bounding boxes were found out. The same procedure is followed here. This leads to the following theorem.

**Theorem 5.5.7.** The nontrivial colored bounding box problem in $\mathbb{R}^2$ can be solved for a three sided rectangle query (i.e. $[a_1, b_1] \times [a_2, \infty)$) using a structure of size $O(n \log^2 n)$ and query time $O(\log^2 n + C \log n)$.

### 5.6 Colored point enclosure weighted sum for $d = 1$

**Problem:** Preprocess a set $S$ of $n$ colored intervals on the real line, where the intervals additionally come with a real-valued weight, such that given a query point $q$ on the real line, the tuples $(c, s_c)$ are reported where $s_c$ is the sum of the weights of intervals of color $c$ stabbed by $q$. 

85
Consider a color $c$ and let $S_c$ be the set of intervals of color $c$, s.t., $|S_c| = n_c$. Let the list of distinct interval endpoints of $S_c$ be sorted from left to right. These endpoints induce partitions on the real line and the regions in this partitioning shall be called “elementary intervals”. Let $I_c$ be the set of these intervals. With each interval $i \in I_c$, we shall maintain an attribute $w_i$ which holds the sum of the weights of intervals in $S_c$ which intersect $i$. Based on the elementary intervals in $I_c$, for all colors $c$, an interval tree, $IT$, is built. Given a query point $q$, a standard search on the interval tree, $IT$, is carried out and all the intervals stabbing $q$ are found out and the $w_i$ attribute associated with each of them is reported. Note that if a color $c$ has at least one interval of $S_c$ stabbed by $q$, then exactly one elementary interval in $I_c$ will be reported and the appropriate weight is reported.

**Theorem 5.6.1.** The colored point enclosure weighted sum problem can be solved in $\mathbb{R}^1$ using a structure of size $O(n)$ and query time $O(\log n + C)$.

**Proof.** The correctness of the query algorithm is quite trivial to observe. Now, if a color $c$ has $n_c$ points, then the size of the set $I_c$ will be $O(n_c)$. Therefore, the total size of all the intervals in set $I_c$, for all colors $c$, is $O(n)$. An interval tree built on $n$ intervals on the real line uses $O(n)$ space and given a query point $q$ on the real line, reports the intervals stabbed by $q$, in $O(\log n + k)$ time, where $k$ is the number of intervals stabbed. Our structure $IT$ will report each color at most once. Therefore, the size of $IT$ will be $O(n)$ and the query time will be $O(\log n + C)$, where $C$ is the number of colors which have at least one interval stabbed by $q$. 

### 5.7 Colored point enclosure weighted sum for $d = 2$

**Problem:** Preprocess a set $S$ of $n$ colored orthogonal rectangles in the plane, where the rectangles additionally come with a real-valued weight, such that given a query point $q$, the tuples $\langle c, s_c \rangle$ are reported where $s_c$ is the sum of the weights of rectangles of color $c$ stabbed by $q$. 

86
A rectangle $r \in S$ is represented as $r = i_x \times i_y$, where $i_x$ and $i_y$ are the projection of $r$ on the $x$-axis and $y$-axis, respectively. A segment tree $T$ is built based on the projections of rectangles in $S$ on the $x$-axis. At a particular internal node $v$ of $T$, if an interval $i_x$ of some rectangle $r$ gets assigned, then we associate $i_y$ with node $v$. Based on the intervals associated with node $v$, we build an auxiliary data structure of Theorem 5.6.1.

Given a query point $q = (a, b)$, we start by querying $T$ with $a$. If $a$ is not in the range of the root, then stop; else proceed. Let $V$ be the set of nodes of $T$ visited by querying with $a$. At each node $v \in V$, we perform a secondary query on the auxiliary data structure built at $v$ by querying it with $b$. For each reported color $c$, the weight obtained from each relevant secondary query is added up.

**Theorem 5.7.1.** The colored point enclosure weighted sum problem can be solved in $\mathbb{R}^2$ using a structure of size $O(n \log n)$ and query time $O(\log^2 n + C \log n)$.

**Proof.** Consider an orthogonal rectangle $r = i_x \times i_y$ of set $S$ having color $c$ and say a query point $q = (a, b)$. Let $q \in r$. When the primary structure of $T$ is queried with $a$, then by the property of a segment tree the node $v$ containing interval $i_x$ will be selected. Querying the auxiliary structure at $v$, the weight of color $c$ reported at this node will contain $i_y$ as well (from Theorem 5.6.1). Hence, no rectangle stabbed by $q$ is missed out. Similarly, all rectangles not stabbed by $q$ are also discarded.

A segment tree built on $n$ intervals takes $O(n \log n)$ space. Also, the auxiliary structure built at each internal node $v$ takes up linear space (from Theorem 5.6.1). Hence, the total space complexity of the structure is $O(n \log n)$.

Given a query point $q$, $O(\log n)$ nodes of $T$ are selected. The time taken to query the auxiliary structure at each node of $T$ is $O(\log n + k)$, where $k$ is the number of colors at that node having at least one interval stabbed by $b$. Also, a color $c$ can get reported at $O(\log n)$ nodes. Therefore, the query time is $O(\log^2 n + C \log n)$. 

In the above solution there is a penalty of $O(\log n)$ associated with each color that is reported. We can overcome this limitation (at the cost of increasing the space) by reduc-
CHAPTER 5. RANGE AGGREGATE STRUCTURES FOR COLORED GEOMETRIC OBJECTS-I

ing the point enclosure problem in \( \mathbb{R}^2 \) to a range search problem in \( \mathbb{R}^4 \). Each rectangle \( r([x_1, x_2] \times [y_1, y_2]) \in S \), is reduced to a 4-dimensional point \((x_1, x_2, y_1, y_2)\) and the query point \( q(a, b) \in \mathbb{R}^2 \) is reduced to \((-\infty, a] \times [a, \infty) \times (-\infty, b] \times [b, \infty)\). The original problem has been transformed into a “colored weighted sum problem” in \( \mathbb{R}^4 \).

**Theorem 5.7.2.** The colored point enclosure weighted sum problem can be solved in \( \mathbb{R}^2 \) using a structure of size \( O(n^{1+\epsilon}) \) and query time \( O(\log n + C) \).

### 5.8 Colored segment intersection weighted sum

**Problem:** Preprocess a set \( S \) of \( n \) colored orthogonal line segments in the plane, where the segments additionally come with a real-valued weight, such that given a query orthogonal rectangle \( q \), the tuples \( \langle c, s_c \rangle \) are reported where \( s_c \) is the sum of the weights of segments of color \( c \) stabbed by \( q \).

Consider one of the vertical segments, say \( s \). Let it’s lower end point be \((s_x, s_l)\) and the upper end point be \((s_x, s_u)\). Given a query rectangle, \( q = [a_1, b_1] \times [a_2, b_2] \), \( s \) will intersect with \( q \), if the following conditions are satisfied: 1) \( a_1 \leq s_x \leq b_1 \), 2) \( s_u \geq a_2 \) and 3) \( s_l \leq b_2 \). Each vertical segment having weight \( w \) in \( \mathbb{R}^2 \) is transformed into a point in \( \mathbb{R}^3 \), such that the segment \( s \) is mapped to \((s_x, s_l, s_u)\) and weight \( w \) is associated. Based on these transformed points, we shall build a data structure \( D \) of Theorem 5.3.5. Thus, for all colors having at least one vertical segment intersecting \( q \), \( D \) will report the sum of the weight of vertical segments of these colors intersecting \( q \). We shall build a similar data structure to handle horizontal segments. The two solutions can be trivially merged without affecting output sensitivity.

**Theorem 5.8.1.** The colored segment intersection weighted sum problem can be solved in \( \mathbb{R}^2 \) using a structure of size \( O(n^{1+\epsilon}) \) and query time \( O(\log n + C) \).
5.9 Conclusions and Open Problems

We considered several range-aggregate queries for colored objects and provided efficient solutions for them. Our techniques have been based mainly on persistent data structures, geometric transformation and on the concept of ‘adding range restrictions’.

Several open problems remain. For some of the problems, the space occupied by the structures is $O(n^{1+\epsilon})$. It would be interesting to see if it can be reduced to $O(n \log^c n)$ (for $c > 0$) while keeping the query time poly logarithmic. It would be desirable to improve the query time of some of the solutions from $O(\log n + C \log n)$ or $O(\log^2 n + C \log n)$ to $O(polylog(n) + C)$. Considering non-trivial bounding box problem for cases wherein each color has some $c$ objects ($c > 2$) intersecting the query region will be a challenging open problem. [32] is a work closely related to this open problem. Finally, extensions to problems involving multiple categorical attribute per object would be an interesting future work.
Chapter 6

Range aggregate structures for colored geometric objects-II

In this chapter we continue the discussion on range-aggregate queries for colored geometric objects which was initiated in the last chapter. The words *colored* and *generalized* are used interchangeably. In this chapter, we describe solutions to the following problems:

- A set $S$ of weighted geometric objects lie in $\mathbb{R}^d$. Let $S_c$ denote the set of geometric objects of $S$ having color $c$. Given a query region $q$ ($\subseteq \mathbb{R}^d$), we need to efficiently report for each distinct color $c$ of the objects in $q$, the tuple $\langle c, p_c \rangle$ where $p_c$ is the point in $S_c \cap q$ with the topmost/maximum weight. Two settings are considered. In the first setting, the set $S$ consists of points and $q$ is an orthogonal query box. We call this problem “Generalized Orthogonal Range-Max query”. In the second setting, the set $S$ consists of orthogonal boxes and $q$ is a point. We call this problem “Generalized Orthogonal Stabbing-Max query”. These two problems are the major focus of this chapter.

- When $F(c) = \text{NULL}$, then a range-aggregate colored problem reduces to a standard colored (generalized) intersection searching problem. We revisit the Generalized range searching and Generalized point enclosure problems. Efficient algorithms are
provided for these problems assuming that the input set of objects are randomly generated.

Our contribution

In this work, we came up with efficient output-sensitive solutions for Generalized Orthogonal Range-Max query (input $S$ is colored and weighted points in $\mathbb{R}^d$, $q$ is an orthogonal query box) and Generalized Orthogonal Stabbing-Max query (input $S$ is colored and weighted hyperboxes in $\mathbb{R}^d$, $q$ is a query point). The results obtained are shown in Table 1 and Table 2. Apart from that, we also provide results for generalized orthogonal range searching and point enclosure problems where the objects are assumed to be randomly generated. The flow of the chapter is as follows: In Section 2, we introduce the terminology used in the paper. Section 3 deals with ‘generalized orthogonal range-max query’. Subsection 3.1 considers a one-dimensional case where insertion of points are allowed. Then in subsection 3.2 we allow both insertion and deletion of points. Subsection 3.3 deals with the static two-dimensional problem which is extended to higher dimensions in Subsection 3.5 and 3.6. Section 4 deals with ‘generalized orthogonal stabbing-max query’. In Section 5 and 6, we consider generalized orthogonal range searching and point enclosure problems, respectively, for randomly generated objects. We conclude and throw some open problems in Section 7.

6.1 Terminology

We define a couple of terms. Consider two points $p(p_1, p_2, \ldots, p_d)$ and $q(q_1, q_2, \ldots, q_d)$. If $p_i > q_i, \forall 1 \leq i \leq d$ then $p$ is defined to be dominating $q$ and $q$ is defined to be dominated by $p$. Given a point set $S$, a point $q \in S$ is called a maximal point iff $q$ is not dominated by any other point in $S$. We denote by $S_c \subseteq S$, all the points of $S$ having color $c$. We assume that the dimension $d$ is a constant relative to $n$ and denote the coordinate axes by $x_1, x_2, \ldots, x_d$. 

91
### Table 6.1
Summary of results obtained for Generalized Orthogonal Range-Max Query. Set $S$ consists of $n$ points lying in $\mathbb{R}^d$ ($d \geq 1$). $k$ denotes the number of colors reported. All results are “big-Oh” and “worst-case” bounds.

<table>
<thead>
<tr>
<th>Underlying space</th>
<th>Query region</th>
<th>Space occupied</th>
<th>Query time</th>
<th>Insertion (amortized)</th>
<th>Deletion (amortized)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d = 1$</td>
<td>$[a_1, \infty)$</td>
<td>$O(n)$</td>
<td>$O(\log n + k)$</td>
<td>$O(\log n)$</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>$[a_1, b_1]$</td>
<td>$O(n \log n)$</td>
<td>$O(\log n + k)$</td>
<td>$O(\log^2 n)$</td>
<td>$O(\log^2 n)$</td>
</tr>
<tr>
<td></td>
<td>$[a_1, b_1]$</td>
<td>$O(n \log n)$</td>
<td>$O(\log n + k)$</td>
<td>$O(\log^3 n)$</td>
<td>$O(\log^3 n)$</td>
</tr>
<tr>
<td>$d = 2$</td>
<td>$[a_1, \infty) \times [a_2, \infty)$</td>
<td>$O(n)$</td>
<td>$O(\log n + k)$</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>$[a_1, b_1) \times [a_2, b_2]$</td>
<td>$O(n \log^2 n)$</td>
<td>$O(\log n + k)$</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$d &gt; 2$</td>
<td>$\prod_{i=1}^d [a_i, b_i]$</td>
<td>$O(n^{1+\epsilon})$ ($\epsilon &gt; 0$)</td>
<td>$O(\log n + k)$</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

### Table 6.2
Summary of results obtained for Generalized Orthogonal Stabbing-Max Query. Set $S$ consists of $n$ orthogonal hyperboxes in $\mathbb{R}^d$. Query, $q$, is a point in $\mathbb{R}^d$. $k$ is the number of colors reported. All results are “big-Oh” and “worst-case” bounds.

<table>
<thead>
<tr>
<th>Underlying Space</th>
<th>Space occupied</th>
<th>Query time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d = 1$</td>
<td>$O(n)$</td>
<td>$O(\log n + k)$</td>
</tr>
<tr>
<td>$d = 2$</td>
<td>$O(n \log n)$</td>
<td>$O(\log n + k \log n)$</td>
</tr>
<tr>
<td>$d &gt; 2$</td>
<td>$O(n^{1+\epsilon})$ ($\epsilon &gt; 0$)</td>
<td>$O(\log n + k)$</td>
</tr>
</tbody>
</table>
6.2 Generalized Orthogonal Range-Max query

Problem: $S$ is a set of $n$ colored objects in $\mathbb{R}^d$ where each point $p$ is assigned a weight $w(p)$. $S_c$ is the set of points of $S$ having color $c$. We wish to preprocess $S$ into a data structure so that given a query orthogonal hyperbox $q = \prod_{i=1}^{d} [a_i, b_i]$, we need to report for each distinct color $c$ of the points in $q$, the tuple $\langle c, p_c \rangle$ where $p_c = \max \{ w(p_c) \mid p_c \in S_c \text{ and } p_c \in q \}$, i.e., the point in $S_c \cap q$ with the topmost/maximum weight.

First, we shall consider the case where the points lie on $\mathbb{R}^1$. Semi-dynamic (only insertions) and dynamic solutions are provided. Next, we turn our attention to solving the static version of the problem in $\mathbb{R}^2$. Finally we conclude the section by providing a static solution for $\mathbb{R}^d, d \geq 3$.

6.2.1 Dynamic one-dimensional problem: Handling only insertions

We shall start of with a solution which handles only insertions. The solution to this problem serves as a warm-up to the fully dynamic problem. Also, we obtain better results for this version compared to the fully dynamic problem. We shall first present a solution to the semi-infinite query $q=[a_1, \infty)$ and then extend it to bounded query $q=[a_1, b_1]$.

Semi-infinite query $q=[a_1, \infty)$

Consider a color $c$. Map each point of $p(p_x) \in S_c$ into a two-dimensional point $p'(p_x, w(p)) \in \mathbb{R}^2$. Call this new set of points $S'_c$. Let the maximal points of this set $S'_c$ in $\mathbb{R}^2$ be denoted by $M'_c$. Next we state a lemma which forms the basis for solving this problem.

Lemma 6.2.1. Consider the set $S'_c$. For a given query $q=[a_1, \infty)$, only the maximal points of $S'_c$ (i.e. $M'_c$) in $\mathbb{R}^2$ can be candidates for topmost/maximum weight among $S_c \cap q$. Rest of the points of $S'_c$ can be discarded.

Proof. Assume that for a particular query $q$, none of the points in $M'_c$ has the topmost weight among $S_c \cap q$. Instead one of the point (say $p$) from $S'_c \setminus M'_c$ has the topmost
CHAPTER 6. RANGE AGGREGATE STRUCTURES FOR COLORED GEOMETRIC OBJECTS-II

weight. We shall prove by contradiction that such a case can never arise. Since \( p \) is not a maximal point of set \( S'_c \), then there must exist at least one point of set \( S'_c \) (say \( p' \)) such that \( p'_x > p_x \) and \( w(p') > w(p) \). Since \( p \) lies within \( q \), even \( p' \) will lie within \( q \). Hence \( p \) cannot have the topmost weight among \( S'_c \cap q \) for the given query \( q \). This holds true for each point in \( S'_c \setminus M'_c \). This leaves us with the set \( M'_c \) as candidates for the topmost weight.

All the data structures which are built in the preprocessing phase are described next. For each color \( c \), based on the points in \( M'_c \), a “priority search tree” \( PST_c \) is built. A \( PST_c \) when built on \( m \) points takes up \( O(m) \) space and reports the points lying inside a 3-sided rectangle in \( O(\log m + k) \) time, where \( k \) is the number of points lying within the 3-sided rectangle. It supports insertions and deletions in \( O(\log m) \) time. Now we shall map the points \( p'(p_x, w(p)) \in M'_c \), for each color \( c \), back to the original point \( p(p_x) \in \mathbb{R}^1 \). Call this set of points \( M_c \). For each color \( c \), we maintain a balanced binary tree, \( T_c \), in which the points \( M_c \) are stored in increasing order of \( x \)-coordinate. We maintain the colors of set \( S \) in a balanced search tree \( CT \). We store with each color \( c \) in \( CT \) a pointer to \( T_c \) and \( PST_c \). The data structures described till now are useful for handling insertions efficiently.

To handle queries efficiently, we build the following structure. For each color \( c \), we sort the points of \( M_c \) by increasing \( x \)-coordinate. For each point \( p \) of color \( c \), let \( \text{pred}(p) \) be its predecessor in the sorted order; for the leftmost point of color \( c \), we take the predecessor to be the point \( -\infty \). We then map \( p \) to the point \( p'(p, \text{pred}(p)) \) in the plane and associate with it the color \( c \). Let \( S' \) be the resulting set of points. We shall build a priority search tree \( D \) based on the points in \( S' \). Given a query \( q=[a_1, \infty) \) it is mapped to a new query \( q'=[a_1, \infty) \times (-\infty, a_1] \). This structure is similar to the one used in [51]. The space occupied by all the structures is bounded by \( O(n) \).

**Lemma 6.2.2.** Given a query \( q=[a_1, \infty) \), when \( D \) is queried with \( q'=[a_1, \infty) \times (-\infty, a_1] \), for each color \( c \) lying within \( q \), the point of \( M_c \cap q \) with the topmost weight gets reported.

**Proof.** First we shall prove that for each color at most one point gets reported. Next we will prove that if a point of color \( c \) gets reported, then it has the topmost weight among \( M_c \).
These points are also deleted from $M$. Then we shall query $PST_c$ into set $S = M \cap q$. Let $p'$ be a point in $q'$, where $p' = (p, \text{pred}(p))$ for some point $p \in M_c$. Since $p'$ is in $[a_1, \infty) \times (-\infty, a_1]$, it is clear that $p_x > a_1$ and hence $p \in q$. This proves that only the points lying within $q$ are reported.

Let $p$ be the leftmost point of $M_c$ in $q = [a_1, \infty)$. Thus $p_x > a_1$ and since $\text{pred}(p) \notin [a_1, \infty)$, we have $a_1 > \text{pred}(p)$. It follows that $p' = (p_x, \text{pred}(p))$ is in $[a_1, \infty) \times (-\infty, a_1]$. We prove that $p'$ is the only point of color $c$ in $q'$. Suppose for a contradiction that $t' = (t_x, \text{pred}(t))$ is another point of color $c$ in $q'$. Thus we have $a_1 < t_x$. $t_x > p_x$ (since we have assumed that $p$ is the leftmost point in $[a_1, \infty)$) we also have $\text{pred}(t) \geq p_x > a_1$. Thus $t'$ cannot lie in $q'$, which is a contradiction. This proves that at most one point (say $p$) for each color (say $c$) gets reported. Notice that $p$ which is the leftmost point of $M_c$ in $q$ is also the point in $M_c \cap q$ with the topmost weight. Suppose for a contradiction that $t \neq p$ is the point of color $c$ with the maximum weight for the query $q$. Then it implies that $t_x > p_x$ and $w(t) > w(p)$. This is a contradiction since both $t$ and $p$ were maximal points in the set $S_c$ and were part of the set $M_c'$. Hence, $p$ is the point of color $c$ with the topmost weight. □

To answer a query $q = [a_1, \infty)$, we simply query $\mathcal{D}$ with $q' = [a_1, \infty) \times (-\infty, a_1]$ and report the weights of the points found. The query time will be $O(\log n + k)$, where $k$ is the number of colors lying inside $q$.

Next let us see how insertions can be handled. Suppose a point $p$ of color $c$ is inserted into set $S$. If $c \notin \mathcal{C}_T$, then we create a new tree $T_c$ containing $p$, insert $(p_x, w(p))$ into a new tree $PST_c$ and insert $(p_x, -\infty)$ into $\mathcal{D}$. Then insert $c$ into $\mathcal{C}_T$ with pointers to $T_c$ and $PST_c$. Suppose $c \in \mathcal{C}_T$. Then we shall query $PST_c$ with $(-\infty, p_x) \times (-\infty, w(p))$. Let $i$ be the number of points reported by $PST_c$. If $i = 0$, then we stop; else if $i > 0$, then each of the $i$ points reported by $PST_c$ is deleted from the tree $PST_c$ itself and also deleted from $\mathcal{D}$. These $i$ points are also deleted from $T_c$. Then $p$ is inserted into $T_c$. Let $u$ be the successor of $p$ in $T_c$. If $u$ exists, then we set $\text{pred}(p)$ to $\text{pred}(u)$ and $\text{pred}(u)$ to $p$; otherwise, we set $\text{pred}(p)$ to the rightmost point in $T_c$. We then insert $(p_x, w(p))$ into $PST_c$ and $(p_x, \text{pred}(p))$ into $\mathcal{D}$. The time taken for insertion of a point turns out to be $O(\log n + i \log n)$. However,
CHAPTER 6. RANGE AGGREGATE STRUCTURES FOR COLORED GEOMETRIC OBJECTS-II

notice that a point of \(S\) which gets deleted once does not reappear again. Therefore, the total time taken for inserting \(n\) points would be \(O(n \log n)\), as \(\Sigma \equiv O(n)\). Hence, the amortized cost for an insertion of a point turns out to be \(O(\log n)\).

**Theorem 6.2.1.** A set of \(n\) colored points in \(\mathbb{R}^1\) can be stored in a linear-size structure such that given a query \(q = [a_1, \infty)\), a generalized orthogonal range-max query can be answered in \(O(\log n + k)\) time. Also, insertion of a new point can be done in \(O(\log n)\) amortized time.

**Bounded query** \(q = [a_1, b_1]\)

Now we shall extend the problem on \(\mathbb{R}^1\) for bounded query \(q = [a_1, b_1]\). The solution is based on the semi-infinite range-max structure of Theorem 6.2.1. We store the points of \(S\) in sorted order by \(x\)-coordinate at the leaves of a \(BB(\alpha)\) tree \(T\). At each internal node \(v\), we store an instance of the structure of Theorem 6.2.1 for queries of the form \([a_1, \infty)\) (resp., \((-\infty, b_1]\)) built on the points in \(v\)'s left (resp., right) subtree. Let \(X(v)\) denote the average of the \(x\)-coordinate in the rightmost leaf in \(v\)'s left subtree and the \(x\)-coordinate in the leftmost leaf of \(v\)'s right subtree.

To answer a query \(q\), we do a binary search down \(T\), using \([a_1, b_1]\) until a highest node \(v\) is reached such that \([a_1, b_1]\) intersects \(X(v)\). If \(v\) is a leaf, then if \(v\)'s point is in \(q\) we report it. If \(v\) is a non-leaf, then we query the structures at \(v\) with \([a_1, \infty)\) and \((-\infty, b_1]\); and then combine the answers. The space occupied by \(T\) becomes \(O(n \log n)\) while the query time remains \(O(\log n + k)\). The amortized time for an insertion into the secondary structures is \(O(\log n)\). So, the amortized insertion time into \(T\) will be \(O(\log^2 n)\).

**Theorem 6.2.2.** A set of \(n\) colored points in \(\mathbb{R}^1\) can be stored in a structure of size \(O(n \log n)\) such that given a query \(q = [a_1, b_1]\), a generalized orthogonal range-max query can be answered in \(O(\log n + k)\) time. Also, insertion of a new point can be done in \(O(\log^2 n)\) amortized time.
6.2.2 Dynamic one-dimensional problem: handling both insertions and deletions

As done previously, we will first handle semi-infinite queries (i.e., \([a_1, \infty)\)) and then extend it to bounded queries (i.e., \([a_1, b_1]\)).

**Semi-infinite query** \(q=[a_1, \infty)\)

In subsection 6.2.1, for each color \(c\), \(PST_c\) (a priority search tree) was being used to efficiently update the set of points of of color \(c\) which can be candidates for topmost point. Since only insertion of points was allowed a simple priority search tree was enough to maintain an updated list of candidate points for topmost point. However, if deletions are also to be handled then a priority search tree would not be enough. As a replacement of \(PST_c\), we would need a data structure \(B_c\) (for each color \(c\)) that can dynamically maintain maximal points of color \(c\) in a plane so that the following two operations should be efficiently handled:

- Insertion of a point \(p\) into the \(B_c\) should be efficiently handled. Also, all the points of color \(c\) which were maximal points before the insertion of \(p\) but are no longer maximal points due to insertion of \(p\) need to reported.

- Deletion of a point \(p\) from \(B_c\) should be efficiently handled. Also, all the points of color \(c\) which were not maximal points before the deletion of \(p\) but have become maximal points due to deletion of \(p\) need to reported.

[80], [40], [58], [62] are some of the important works done on dynamic maintenance of maximal points in the plane. They dynamically maintain the list of maximal points of a set \(S\) of points (as a staircase-shaped countour called \(m\)-contour of \(S\)) under online insertions and deletions in \(S\). However, in our problem there are additional operations which need to be done while inserting/deleting a point. We shall now briefly describe the data structure
$B_c$ proposed by [80]. Then we will show how a modification to their insertion and deletion algorithm can help us in handling the above two mentioned operations efficiently.

**Review of Overmars-van Leeuwen structure [80].** Consider a color $c$. As done in the previous subsection, we map each point $p(p_z) \in S_c$ into a two-dimensional point $p'(p_x, w(p)) \in \mathbb{R}^2$. Call this new set of points $S'_c$. In the Overmars-van Leeuwen structure, the points of $S'_c$ are stored by $x$-coordinate at the leaves of a balanced binary search tree, $B_c$, which is taken to be a red-black tree [87]. For any internal node $v$, let $B(v)$ be the subtree rooted at $v$ and let $l(v)$ (resp. $r(v)$) be $v$’s left (resp. right) child. Let $C(v)$ be the $m$-contour for the points stored at the leaves of $B(v)$. Since $C(l(v))$ and $C(r(v))$ are separated by a vertical line, $C(v)$ is simply a concatenation of $C(r(v))$ and the “head”, $H(l(v))$, of $C(l(v))$ consisting of the points that are strictly above the highest point of $C(r(v))$. At $l(v)$, only the “tail”, $T(l(v)) = C(l(v)) - H(l(v))$, is stored. At $v$, the lowest point of $C(l(v))$ that is strictly above the highest point of $C(r(v))$ is stored in the field $\text{split}(v)$, thus telling where $C(v)$ should be split to regain $C(r(v))$ and $H(l(v))$. For a leaf node $w$, $C(w)$ is just the point stored in $w$ and at the root of $B_c$, $C(\text{root}(B_c)) = m$-contour of $S'_c$.

When the data structure is at “rest” (i.e. in between updates), the only contours present in it are $C(\text{root}(B_c))$ and $T(l(v))$ (possibly empty) for each nonleaf $v$. Each of these is implemented as a concatenable queue on $y$-coordinates. During updates (see below), the splitting and concatenation of these contours will produce new contours, also as concatenable queues.

Since $C(\text{root}(B_c))$ is implemented as a concatenable queue, the $m$-contour can be listed out in order in $O(m)$ time. To insert/delete a point $p$, a binary search is done down $B_c$ and at each node $v$, on the search path, $C(l(v))$ and $C(r(v))$ are reconstructed, as concatenable queues, by splitting $C(v)$ and concatenating $H(l(v))$ with $T(l(v))$. When the search terminates, a leaf for $p$ is inserted/deleted in $B_c$ and the search path is traversed back to the root. At each node $v$, $C(v)$ is reconstructed from $C(l(v))$ and $C(r(v))$ (which are now up-to-date) by recomputing $\text{split}(v)$ by binary search on $C(l(v))$, splitting $C(l(v))$, concatenating $H(l(v))$ and $C(r(v))$, and leaving $T(l(v))$ at $l(v)$. Each split, concatenation and
CHAPTER 6. RANGE AGGREGATE STRUCTURES FOR COLORED GEOMETRIC OBJECTS-II

binary search takes $O(\log n)$ time and since $B_c$ has a height of $O(\log n)$ the total time is $O(\log^2 n)$. In addition, rebalancing rotations are performed as necessary. The rebalancing time is $O(\log n)$.

Enhancement to their structure. For our problem, when a new point $p$ is being inserted (or deleted), we also want to know all the points of $S'_c$ which have disappeared (or appeared) as maximal points which the existing algorithm of Overmax-van Leeuwan cannot handle.

A concatenable queue is a variant of the standard binary search tree which can handle searches, splits and concatenation efficiently. When a point $p$ of color $c$ is to be inserted into $B_c$, then we do a search on $C(root(B_c))$ with $w(p)$ to find out all the points which lie in the $SW$-quadrant of $(p_x, w(p))$. Note that all these points (if any) will lie continously in the leaves of the $C(root(B_c))$. If $k$ points are reported, then this takes $O(\log n + k)$ time. Then, $p$ is inserted into $B_c$ using the standard routine discussed above. Therefore, the time taken for insertion of a point will be $O(\log^2 n + k)$. In case of deletion, we first delete the point (say $p$) from $B_c$ using the standard routine. Then we do a search on $C(root(B_c))$ with $w(p)$ to find out all the points which lie in the $SW$-quadrant of $(p_x, w(p))$. Therefore, deletion will take $O(\log^2 n + k)$ time, where $k$ are the number of newly appearing maximal points due to the deletion of point $p$.

Now we are ready with all the structures needed to handle both insertions and deletions. The structures $CT$, $T_c$ for each color $c$ and $D$ described in subsubsection 6.2.1 are used again here and their functionality remains the same. $PST_c$ is replaced by $B_c$ whose functionality has just now been discussed.

To answer a query $q=[a_1, \infty)$, we simply query $D$ with $q'=[a_1, \infty) \times (-\infty, a_1]$ and report the weights of the points found. The query time will be $O(\log n + k)$, where $k$ is the number of colors lying inside $q$.

Suppose a point $p$ of color $c$ is inserted into set $S$. If $c \notin CT$, then we create a new tree $T_c$ containing $p$, insert $(p_x, w(p))$ into a new tree $B_c$ and insert $(p_x, -\infty)$ into $D$. Then insert $c$ into $CT$ with pointers to $T_c$ and $B_c$. Suppose $c \in CT$. Then we shall insert $p$ into $B_c$. Let $i$ be the number of points reported on inserting $p$ into $B_c$. If $i = 0$, then we stop;
else if $i > 0$, then each of the $i$ points reported by $B_c$ is deleted from $D$. These $i$ points are also deleted from $T_c$. Let $u$ be the successor of $p$ in $T_c$. If $u$ exists, then we set $\text{pred}(p)$ to $\text{pred}(u)$ and $\text{pred}(u)$ to $p$; otherwise, we set $\text{pred}(p)$ to the rightmost point in $T_c$. We then insert $p$ into $T_c$, $(p_x, w(p))$ into $PST_c$ and $(p_x, \text{pred}(p))$ into $D$. The time taken for insertion of a point turns out to be $O(\log^2 n + i \log n)$. However, if we perform a sequence of $n$ insertions, then a point of $S$ having color $c$ will get reported only once by $B_c$. Therefore, the total time taken for inserting $n$ points would be $O(n \log^2 n)$, as $\sum i \equiv O(n)$. Hence, the amortized cost for an insertion of a point turns out to be $O(\log^2 n)$.

If a point $p$ is to be deleted from $S$, then it is first deleted from $B_c$. Let $i$ be the number of points reported by $B_c$. Then delete $B_c$ from $T_c$ and $D$. The $i$ reported points are inserted into $T_c$. Each of these $i$ points (say $p'$) find out their predecessor in $T_c$ and insert $p'(p'_x, \text{pred}(p'))$ into $D$. The time taken for deletion of a point turns out to be $O(\log^2 n + i \log n)$. However, if we perform a sequence of $n$ deletions, then a point of $S$ having color $c$ will get reported only once by $B_c$. Therefore, the total time taken for deleting $n$ points would be $O(n \log^2 n)$, as $\sum i \equiv O(n)$. Hence, the amortized cost for a deletion of a point turns out to be $O(\log^2 n)$.

**Theorem 6.2.3.** A set of $n$ colored points in $\mathbb{R}^1$ can be stored in a linear-size structure such that given a query $q = [a_1, \infty)$, a generalized orthogonal range-max query can be answered in $O(\log n + k)$ time. If we do $n$ insertions, then the amortized insertion time is $O(\log^2 n)$. If we do $n$ deletions, then the amortized deletion time is $O(\log^2 n)$.

**Bonded query** $q=[a_1, b_1]$  

Using the same technique used in subsubsection 6.2.1, the semi-infinite query solution can be extended to bounded query solution. The space and the update time increase by a $O(\log n)$ factor while the query time remains the same.

**Theorem 6.2.4.** A set of $n$ colored points in $\mathbb{R}^1$ can be stored in a structure of size $O(n \log n)$ such that given a query $q = [a_1, b_1]$, a generalized orthogonal range-max query can be answered in $O(\log n + k)$ time. If we do $n$ insertions, then the amortized insertion
6.2.3 Two dimensional scenario

To solve the problem for $\mathbb{R}^2$, we first consider queries of the form $[a_1, \infty) \times [a_2, \infty)$. The solution to this problem forms the basis for the answering bounded queries $[a_1, b_1] \times [a_2, b_2]$.

**Quadrant query, $q = [a_1, \infty) \times [a_2, \infty)$**

We shall consider the problem for quadrant queries, $q = [a_1, \infty) \times [a_2, \infty)$. Consider points $p(p_x, p_y)$ and $r(r_x, r_y)$ both having the same color $c$. Let $r_x > p_x, r_y > p_y$ and $w(r) > w(p)$. For an arbitrary query $q$, if $p$ lies in $q$ then $r$ will also lie in $q$ and since $w(r)$ is greater than $w(p)$, point $p$ cannot have the maximum/topmost weight among $S_c \cap q$. Hence, such points are can be removed from consideration.

In order to remove the points of $S$ which cannot be candidates for maximum/topmost weight, we do the following: Fix a color $c$. Map each point $p(p_x, p_y) \in S_c$ to a three-dimensional point $p'(p_x, p_y, w(p))$. Call this new set of transformed points $S'_c$. Maximal points, $M'_c$, of $S'_c$ are found out in $\mathbb{R}^3$. This can be done in time $O(|S'_c| \log^2 |S'_c| + |M'_c|)$. $M'_c$ represents the set of points from $S'_c$ (or $S_c$) which are possible candidates for topmost/maximum weight for color $c$. This process is repeated for each color $c$. Denote $M' = \bigcup_c M'_c$.

The total time taken for finding $M'$ will be $O(\Sigma_{c}(|S'_c| \log^2 |S'_c| + |M'_c|)) \equiv O(\log^2 n \times \Sigma |S'_c| + \Sigma |M'_c|) \equiv O(n \log^2 n + |M'|) \equiv O(n \log^2 n)$ since $|M'| \leq n$.

Once again fix a color $c$. New we shift our attention from $S_c$ to $M'_c$. Each point $p'(p_x, p_y, w(p)) \in M'_c$ is mapped back to its original two-dimensional point $p(p_x, p_y)$ with weight $w(p)$. Call this set $M_c$. Notice that all the points in $M_c$ need not be maximal w.r.t. to the two-dimensional plane, though they were all maximal points in three-dimensional space as part of set $M'_c$ (as shown in Figure 6.1). This process is repeated for each color $c$.

For each color $c$, set $M_c$ is divided into *layers of maximal points* in $\mathbb{R}^2$. Layer 1 is
Figure 6.1 For a particular color $c$, set $M_c$ is being shown. These points are the only candidates from color $c$ which can have the topmost/maximum weight for a Generalized Orthogonal Range-Max query.

denoted by $M^1_c$ which is nothing but the set of maximal points of $M_c$ in $\mathbb{R}^2$. Layer $l$, $M^l_c$ ($l > 1$), is defined to be the set of maximal points of $M_c \setminus \bigcup_{j=1}^{l-1} M^j_c$ in $\mathbb{R}^2$. These layers are defined until we reach an empty layer. In Figure 6.1, we show an example of a set $M_c$ having seven points. The weight associated with each point is also shown. For the purpose of discussion, each point is uniquely referred to by its weight. In this example $M_c$ gets divided into three layers of maximal points. Layer 1, $M^1_c = \{10\}$; Layer 2, $M^2_c = \{30, 20, 25\}$ and Layer 3, $M^3_c = \{50, 100, 80\}$.

These “layers of maximal points” for each set $M_c$ are obtained as follows: Based on the points in $M_c$ we build a data structure $T$ described in [62]. $T$ maintains the set of maximal points in the plane of set $M_c$. Initial building of the structure takes $O(|M_c| \log |M_c|)$ time. Insertion or Deletion of a point is handled in $O(|M_c| \log |M_c|)$ amortized time. The reporting of maximal points takes $O(r)$ time where $r$ is the number of maximal points. Using $T$ we can directly find out the set of maximal points of $M_c$ which constitutes $M^1_c$. Now all the points in $M^1_c$ are deleted from $T$. Now the maximal points reported by $T$ will be $M^2_c$. Next all the points in $M^2_c$ are deleted from $T$. This process is repeated iteratively till $T$ becomes empty. The time taken for finding layers of $M_c$ will be $O(|M_c| \log |M_c|)$. Total time taken for finding all sets $M_c$ will be $O(n \log n)$. 102
For a query quadrant \( q = [a_1, \infty) \times [a_2, \infty) \), call \((a_1, a_2)\) to be an apex point of \( q \). Now for each point \( p \in M_c \), we shall define a region \( r(p) \). If \((a_1, a_2)\) stabs a particular region \( r(p) \) then the point \( p \) corresponding to that region will have the maximum weight among \( S_c \cap q \). Let \( M_c \) have \( l \) layers of maximal points. We start with \( M_c^l \) (layer \( l \)) and go till \( M_c^1 \) (layer 1). The points in \( M_c^1 \) are sorted in decreasing order based on their weights. The first point in the list, \( p(p_x, p_y) \), is assigned the region \( r(p) = (-\infty, p_x] \times (-\infty, p_y] \). The \( i^{th} \) point in the list \( M_c^l \) is assigned the region \( r(p) = (-\infty, p_x] \times (-\infty, p_y] \setminus \bigcup r(p') \), where the union is over the first \( i-1 \) points in the list \( M_c^l \). See \( M_c^3 \) (layer 3) in Figure 6.1 on how \( r(100) \), \( r(80) \) and \( r(50) \) have been assigned. Let \( R(l') = \bigcup r(p) \), where the union is over all the points in \( M_c^{l'} \). For layers above \( M_c^l \) we do the following: Suppose we are at a layer \( l' \). Then we sort the points in \( M_c^{l'} \) in decreasing order of their weights. Then the \( i^{th} \) point \( p(p_x, p_y) \) in \( M_c^{l'} \) is assigned the region \( r(p) = (-\infty, p_x] \times (-\infty, p_y] \setminus \bigcup_{j=l' + 1}^{l+1} R(j) \setminus \bigcup r(p') \), where the union over \( r(p') \) is the first \( i-1 \) points in the list \( M_c^{l'} \). The region associated with each point in \( M_c \) is shown in Figure 6.1. The shaded region shows the region associated with point 20. So given a query \( q \), we need to check in which region \((a_1, a_2)\) lies and the point of \( M_c \) corresponding to that region has to be reported.

If the region \( r(p) \) associated with point \( p \in M_c \) is not in the form of an axis-parallel rectangle, then \( r(p) \) is broken into axis-parallel rectangles (see \( r(30) \) in Figure 6.1). So, for each color \( c \) we have a collection of disjoint rectangles \( \chi_c \). \( \chi_c \equiv O(|M_c|) \), since from each point in \( M_c \) at most three rays are shooting out (see Figure 6.1) and it is a planar surface. Based on the rectangles \( \chi_c \) obtained for each color \( c \) we build an instance of the structure \( D \) in [24]. Given a query point, \( D \) reports all the rectangles stabbed by the query point. The number of rectangles stored in \( D \) will be \( O(n) \).

Finally, when we are given a query quadrant \( q = [a_1, \infty) \times [a_2, \infty) \), we shall query \( D \) with \((a_1, a_2)\) and for each rectangle that gets stabbed, the point that corresponds to that rectangle along with its weight is reported. Note that at most only one rectangle of each color is reported.

**Theorem 6.2.5.** A set of \( n \) colored points in \( \mathbb{R}^2 \) can be stored in a linear-size structure such
that given a query quadrant \( q = [a_1, \infty) \times [a_2, \infty) \), a generalized orthogonal range-max query can be answered in \( O(\log n + k) \) time.

Bounded rectangular query

In this subsection we solve the problem for bounded orthogonal query rectangle \( q = [a_1, b_1] \times [a_2, b_2] \). The solution is based on the quadrant range-max structure of Theorem 6.2.5.

We first show how to solve the problem for query rectangles \( q' = [a_1, b_1] \times [a_2, \infty) \). In this discussion, \( NE \)-query would mean \( [a_1, \infty) \times [a_2, \infty) \) and \( NW \)-query would mean \((-\infty, b_1) \times [a_2, \infty) \). We store the points of \( S \) in sorted order by \( x \)-coordinate at the leaves of a balanced binary tree \( T' \). At each internal node \( v \), we store an instance of the structure of Theorem 6.2.5 for \( NE \)-queries (resp., \( NW \)-queries) built on the points in \( v \)'s left (resp., right) subtree. Let \( X(v) \) denote the average of the \( x \)-coordinate in the rightmost leaf in \( v \)'s left subtree and the \( x \)-coordinate in the leftmost leaf of \( v \)'s right subtree.

To answer a query \( q' \), we do a binary search down \( T' \), using \([a_1, b_1]\) until a highest node \( v \) is reached such that \([a_1, b_1]\) intersects \( X(v) \). If \( v \) is a leaf, then if \( v \)'s point is in \( q' \) we report its color. If \( v \) is a non-leaf, then we query the structures at \( v \) using the \( NE \)-quadrant and the \( NW \)-quadrant derived from \( q' \) (i.e., the quadrants w.r.t. points at \((a_1, a_2)\) and \((b_1, a_2)\), respectively), and then combine the answers. The space occupied by \( T' \) becomes \( O(n \log n) \) and the query time remains \( O(\log n + k) \).

To solve the problem for general query rectangles \( q = [a_1, b_1] \times [a_2, b_2] \), we use the above approach again, except that we store the points in the tree sorted by \( y \)-coordinates. At each internal node \( v \), we store an instance of the data structure above to answer queries of the form \([a_1, b_1] \times [a_2, \infty) \) (resp., \([a_1, b_1] \times (-\infty, b_2) \) on the points in \( v \)'s left (resp., right) subtree. The query strategy is similar to the previous one, except that we use the interval \([a_2, b_2]\) to search in the tree. The space increases by a log factor though the query time remains the same.

**Theorem 6.2.6.** A set of \( n \) colored points in \( \mathbb{R}^2 \) can be stored in a structure of size...
CHAPTER 6. RANGE AGGREGATE STRUCTURES FOR COLORED GEOMETRIC OBJECTS-II

\(O(n \log^2 n)\) such that given a query rectangle \(q = [a_1, b_1] \times [a_2, b_2]\), a generalized orthogonal range-max query can be answered in \(O(\log n + k)\) time.

6.2.4 Generalized orthogonal range-max query for \(d = 3\)

In this section, we show how to solve the generalized weighted sum problem for a query cuboid, \(q = [a_1, b_1] \times [a_2, b_2] \times [a_3, b_3]\).

We make use of Theorem 5.3.3 to solve the problem. Let \(DS\) in Theorem 5.3.3 be the data structure of Theorem 6.2.6 for solving the generalized orthogonal range-max problem for \(d = 2\) and queries of the form \(q' = [a_2, b_2] \times [a_3, b_3]\). We need a data structure \(TDS\) to solve generalized orthogonal range-max queries \(TPR(q', [a_1 : \infty), S)\). Given \(n\) colored points in \(\mathbb{R}^3\), we sort the points by their \(x\)-coordinates and rank the points in left to right order (ties broken arbitrarily) and store their \(x\)-coordinates in an auxiliary array \(AUX\). We create data structures \(DA_i\) for \(1 \leq i \leq n\). Each such data structure is an instance of the data structure of Theorem 6.2.6 for the 2-dimensional static generalized orthogonal range-max problem which takes \(O(n \log^2 n)\) space and supports queries in time \(O(\log n + k)\).

We build data structure \(DA_i\) on the \(y\)-coordinates and \(z\)-coordinates of the points \(p\) whose \(x\)-coordinates are at least \(AUX[i]\). Given a query \(TPR(q', [a_1 : \infty), S)\), we first binary search in \(AUX\) with \(a_1\) to determine the index \(i\) of the leftmost point whose \(x\)-coordinate is greater than or equal to \(a\). Then we simply query \(DA_i\) with \(q'\).

Lemma 6.2.3. A set of \(n\) colored weighted points in \(\mathbb{R}^3\) can be preprocessed into a data structure of size \(O(n^2 \log^2 n)\) such that a generalized orthogonal range-max query \(TPR(q', [a_1 : \infty), S)\) can be answered in \(O(\log n + k)\) time where \(C\) is the output size.

Proof. The correctness follows from the correctness of the solution to the 2-dimensional problem in Theorem 6.2.6 and the fact that \(DA_i\) is really built on all points in \(S\) with \(x\)-coordinates in \([a_1 : \infty)\). Since each data structure \(DA_i\) takes space \(O(n \log^2 n)\) and there are \(O(n)\) of them, the total space taken is \(O(n^2 \log^2 n)\). The query time is \(O(\log n + \ldots)\)
since it takes $O(\log n)$ time to determine the index $i$ and $O(\log n + k)$ time to query $DA_i$. \hfill \Box

Now since we have both the structures $DS$ and $TDS$, we can apply Theorem 5.3.3 with $w = 3$, and Corollary 5.3.1 to conclude:

**Theorem 6.2.7.** A set of $n$ colored weighted points in $\mathbb{R}^3$ can be preprocessed into a data structure of size $O(n^{1+\epsilon})$, for an arbitrarily small positive constant $\epsilon$, such that given a query cuboid $[a_1, b_1] \times [a_2, b_2] \times [a_3, b_3]$, a generalized orthogonal range-max query can be answered in $O(\log n + k)$ time where $k$ is the output size.

### 6.2.5 Generalized weighted sum for $d > 3$

To solve the problem in dimension $d > 3$ we again make use of Theorem 5.3.3. Assume that as $DS$, we have the data structure to solve the problem $PR$ in dimension $d - 1$, which takes $O(n^{1+\epsilon})$ space and $O(\log n + k)$ time. As $TDS$, we create a structure similar to the one for Lemma 6.2.3, by taking $O(n)$ instances of $DS$, which gives us a data structure with $O(n^{2+\epsilon})$ space and $O(\log n + k)$ query time. Now we apply Theorem 5.3.3, with $w = 3$ and Corollary 5.3.1 to conclude:

**Theorem 6.2.8.** A set of $n$ colored points in $\mathbb{R}^d$ ($d \geq 3$), each of which is associated with a real-valued non-negative weight, can be preprocessed into a data structure of size $O(n^{1+\epsilon})$, for an arbitrarily small positive constant $\epsilon$, such that given a query $d$-dimensional orthogonal box, a generalized orthogonal range-max query can be answered in $O(\log n + k)$ time where $k$ is the output size.

### 6.3 Generalized orthogonal stabbing-max query

**Problem:** $S$ is a set of $n$ colored hyperboxes in $\mathbb{R}^d$ where each hyperbox $\gamma$ is assigned a weight $w(\gamma)$. $S_c$ is the set of hyperboxes of $S$ having color $c$. We wish to preprocess $S$ into
a data structure so that given a query point \( q \) in \( \mathbb{R}^d \), we can report for each distinct color \( c \) of the hyperboxes stabbed by \( q \), the tuple \( \langle c, \gamma_c \rangle \) where \( \gamma_c = \max \{ w(\gamma_c) \mid \gamma_c \in S_c \text{ and } q \in \gamma_c \} \), i.e., the hyperbox in \( S_c \cap q \) with the topmost/maximum weight.

We begin by providing a solution to this problem for \( d = 1 \). Then a solution is provided for \( d = 2 \). Finally it is extended for \( d \geq 3 \).

### 6.3.1 Solution for \( d = 1 \)

We start of with \( n \) colored segments on the real-line. Consider a color \( c \) and the segments \( S_c \) of that color. Let \( p_1, p_2, \ldots, p_m \) be the list of segment endpoints of \( S_c \) sorted from left to right. These endpoints induce partitions on the real-line and these partitions are called “elementary intervals”. The elementary intervals, say \( I_c \), from left to right are: \((-\infty, p_1), [p_1, p_1], (p_1, p_2), [p_2, p_2], \ldots, (p_m-1, p_m), [p_m, p_m], (p_m, \infty)\). With each interval \( i \in I_c \), we shall store \( w(\gamma) \), where \( \gamma \) is the segment with the maximum weight among all the segments of \( S_c \) which intersects interval \( i \). If any interval is not intersected by any of the segments of \( S_c \) then it is discarded from \( I_c \). Based on the elementary intervals in \( I_c \), for all colors \( c \), we build an interval tree \( IT \). The number of intervals stored in \( IT \) will be bounded by \( O(n) \). Given a query point \( q \), we search \( IT \) and the intervals stabbed by \( q \) are reported. The weight associated with each interval is the required answer.

**Theorem 6.3.1.** A set of \( n \) segments can be stored in a linear-size structure such that given a query point \( q \), a generalized stabbing-max query can be answered in \( O(\log n + k) \) time.

### 6.3.2 Solution for \( d = 2 \)

Our approach for solving the problem in two-dimensional plane is to design a dynamic data structure for the one-dimensional version of the problem which can handle query as well as updates efficiently. This is followed by making this data structure partially persistent using the technique of Driscoll et. al. [34].

107
First, we build the dynamic data structure for the 1D version. Based on the segments in $S$ an augmented segment tree $T$ is built as follows: The segments of $S$ divide the real-line into elementary intervals. A balanced binary tree $T$ is built with these elementary intervals as the leaves of the tree. Then each segment in $S$ is inserted into $T$. Consider a node $v$ of $T$. Let $S^c_v$ be the set of segments of color $c$ assigned to node $v$. A red-black tree $T^c_v$ is built based on the weights of the segments in set $S^c_v$ (in decreasing order of weights). Also a pointer is maintained from the root of $T^c_v$ to the leftmost leaf in it. In this way red-black trees are built for each unique color of the segments assigned to node $v$. Given a query point $q$ on the real-line, we search in $T$. At each node $v$ visited from root to leaf, the weight stored in the leftmost leaf of each red-black tree $T^c_v$ is reported. For each color, the maximum among all the weights reported of that color is found out. The space occupied by $T$ is $O(n \log n)$ and the query time is $O(\log n + k \log n)$.

Now let us consider updates. Insertion of a segment of color $c$ would involve going to $O(\log n)$ nodes in $T$ and inserting itself into the secondary red-black tree $T^c_v$. A new tree $T^w_v$ is created if it does not exist previously. So, insertion time will be $O(\log^2 n)$ amortized due to possibility of a rotation taking place. Similar analysis holds for deletion of segments.

Now the 1D solution has to be made partially persistent. Following the technique of [85], the $x$-span of all the rectangles are considered, then broken into elementary intervals and the primary structure of the segment tree is built. We make it persistent by sweeping a horizontal line $l$ from top to bottom, inserting the $y$-span of a rectangle when it is “entered” by $l$ and delete the same $y$-span when the sweeping line “leaves” that rectangle. Note that now there won’t be any rotations taking place during insertions and deletions of $y$-spans since the primary structure has already been built on $O(n)$ segments. Hence, the number of changes taking place during an update will be bounded by $O(\log n)$ (constant changes at each of the $O(\log n)$ nodes it is/was assigned to). A 2D-Generalized orthogonal stabbing-max query thus can be answered by first identifying the appropriate version of $D$ and then using it to answer the 1D problem. This leads to the following theorem.

**Theorem 6.3.2.** A set of $n$ rectangles can be stored in a structure of size $O(n \log n)$ such
that given a query point \( q \in \mathbb{R}^2 \), a generalized stabbing-max query can be answered in \( O(\log n + k \log n) \) time.

### 6.3.3 Solution for \( d \geq 3 \)

Using a standard technique in the literature, any point enclosure problem in \( \mathbb{R}^d \) can be transformed to a range search problem in \( \mathbb{R}^{2d} \). So in our case we can transform our Generalized orthogonal stabbing-max query in \( \mathbb{R}^d \) \((d \geq 3)\) to Generalized orthogonal range-max query in \( \mathbb{R}^{2d} \). Theorem 5.3.5 leads us to the following result.

**Theorem 6.3.3.** A set of \( n \) hyperboxes can be stored in a structure of size \( O(n^{1+\epsilon}) \) such that given a query point \( q \in \mathbb{R}^d \), \( d \geq 3 \), a generalized stabbing-max query can be answered in \( O(\log n + k) \) time.

### 6.4 Generalized \( d \)-dimensional Range Searching on randomly distributed points

**Problem:** Preprocess a set \( S \) of \( n \) colored points in \( \mathbb{R}^d \), so that for any given orthogonal query box \( q = \prod_{i=1}^{d}[a_i, b_i] \), report the distinct colors of the points inside \( q \).

In \( \mathbb{R}^2 \) and \( \mathbb{R}^3 \), there exists solutions to this problem which use \( O(n \text{ poly log} n) \) space and \( O(\text{poly log} n + k) \) query time [49]. However, for \( d \geq 4 \), the only known solution takes \( O(n^{1+\epsilon}) \) space and query time \( O(\log n + k) \) [49]. In this subsection, we consider the case when the points of \( S \) are randomly distributed in \( \mathbb{R}^d \). We obtain a data structure which takes up expected \( O(n \log^{2d-2} n) \) space and \( O(\log^{d-1} n + k) \) query time. In this way we partially succeed in obtaining a \( O(n \text{ poly log} n) \) space and \( O(\text{poly log} n + k) \) query time solution.

First we consider queries of the form \( q = \prod_{i=1}^{d}[a_i, \infty) \). Consider a color \( c \) and let \( M_c \subseteq S_c \) be the set of maximal points of \( S_c \). Clearly, the color \( c \) will be reported iff at least one point of \( M_c \) lies inside \( q \). Hence we deal only with the set \( M_c \), for each color \( c \).
For a point \( p(p_1, p_2, \ldots, p_d) \in M_c \), denote by \( r(p) = \prod_{i=1}^{d} (\mathbb{R}, p_i] \subseteq \mathbb{R}^d \), the hypercube within which the query apex point \((a_1, a_2, \ldots, a_d)\) should lie for point \( p \) to lie in \( q \). Let \( R(M_c) = \bigcup_{p \in M_c} r(p) \). \( R(M_c) \) is decomposed into a set of pairwise disjoint orthogonal boxes denoted by \( R_c \). Now a color \( c \) will be reported iff point \((a_1, a_2, \ldots, a_d)\) lies inside one of the boxes in \( R_c \). A standard data structure \( DS \) is built based on the boxes in \( R_c \), for all colors \( c \), for reporting all the boxes containing a query point \((a_1, a_2, \ldots, a_d)\) [24]. Note that for each color \( c \) at most one box will be reported.

Consider a set \( S_c \) having \( n_c \) points. If these points have been generated by a random process, where the values in each coordinate are independently generated random real numbers, then the expected number of maximal points in \( M_c \) will be \( O(\log^{d-1} n_c) \) and hence \( O(\log^{d-1} n_c) \) hypercubes. The maximum number of vertices of the union of \( m \) axis-parallel hypercubes of the same size is \( \Theta(m^{\lfloor d/2 \rfloor}) \), for \( d \geq 2 \) [11]. So, the number of boxes in \( R_c \) will be \( O((\log^{d-1} n_c)^{\lfloor d/2 \rfloor}) \equiv O(n_c) \). Therefore, the expected number of boxes for all colors \( c \) will be bounded by \( \Sigma O(n_c) \equiv O(n) \). \( D \) when build on \( O(n) \) boxes takes up \( O(n \log^{d-2} n) \) space and answers queries in \( O(\log^{d-1} n + k) \) time. This solution can be extended to bounded queries of the form \( q = \prod_{i=1}^{d} [a_i, b_i] \) by using the same technique used in subsection 6.2.3. The query time remains the same but the space is increased by a factor of \( O(\log^d n) \). Therefore, the total expected space becomes \( O(n \log^{2d-2} n) \) space.

**Theorem 6.4.1.** Let \( S \) be a set of \( n \) colored and randomly distributed points in \( \mathbb{R}^d \), \( d \geq 4 \). We can build a data structure of expected size \( O(n \log^{2d-2} n) \) size such that given an orthogonal query box, we can report the \( k \) distinct colors of the points that are contained in it in \( O(\log^{d-1} n + k) \) worst case time.
6.5 Generalized $d$-dimensional Point Enclosure on randomly distributed hyperboxes

Problem: Preprocess a set $S$ of $n$ colored orthogonal hyperboxes in $\mathbb{R}^d$, so that for any given orthogonal query point $q$, report the distinct colors of the hyperboxes stabbed by $q$.

For $d \geq 3$, there do not exist solutions to this problem which take $O(n \text{polylog } n)$ space. In this section, we consider the case where the hyperboxes of $S$ are randomly distributed in $\mathbb{R}^d$. We obtain a data structure which takes $O(n \log^{d-2} n)$ expected space and $O(\log^{d-1} n + k)$ query time.

Fix a color $c$ and let $S_c$ be the set of hyperboxes of color $c$. Union of all the hyperboxes in $S_c$ is found out and denoted by $R(c)$. $R(c)$ is next decomposed into a set of pairwise disjoint orthogonal boxes denoted by $R_c$. For a given query $q(a_1, a_2, \ldots, a_d)$, color $c$ will be reported if $q$ stabs any of the boxes in $R_c$. A standard data structure $DS$ is built based on the boxes in $R_c$, for all colors $c$, for reporting all the boxes containing a query point $(a_1, a_2, \ldots, a_d)$ [24]. Note that for each color $c$ at most one box will be reported. It can be shown that for a color $c$ having $n_c$ points, if all its hyperboxes are randomly generated then the expected size of the union will be $O((\log^{2d-1} n_c)^{\lfloor d/2 \rfloor}) = O(n_c)$. The expected number of boxes stored in $DS$ will be $O(n)$.

Theorem 6.5.1. Let $S$ be a set of $n$ colored and randomly distributed hyperboxes in $\mathbb{R}^d$, $d \geq 3$. We can build a data structure of expected size $O(n \log^{d-2} n)$ such that given a query point, we can report the $k$ distinct colors of the points that are contained in it in $O(\log^{d-1} n + k)$ worst case time.

6.6 Open problems

Most of the open problems raised in the previous chapter are applicable in this chapter as well. Specifically, for $d > 2$ there is need for $O(n \text{polylog } n)$ solutions. For generalized
orthogonal stabbing max query in $d=2$, how do we improve the query time from $O(\log n + k \log n)$ to $O(f(n) + k)$ (where $f(n)$ is preferably poly-logarithmic)?
Chapter 7

Emptiness or One-reporting queries for Standard Intersection problems

Suppose a set $S$ of geometric objects in $\mathbb{R}^d$ are stored in a data structure and given a geometric query region $q$, we want to compute $S \oplus q$. Typically, in the Computational Geometry literature the $\oplus$ operator turns out be either a reporting or a counting query. In a reporting query we are interested in reporting all the objects of $S$ intersecting with $q$ and in a counting query we want to count the number of objects of $S$ intersecting with $q$. Very little focus has been given to the Emptiness/One-reporting queries where we are interested to know if any object of $S$ has intersected with $q$ or not. This query can be treated as a special case of reporting and counting queries. So, the general strategy to solve Emptiness queries is to build a data structure which solves the reporting or counting version and then to use it for answering the emptiness query as well. In this chapter we make an attempt to show that building data structures specifically for emptiness query can lead to more efficient structures for most of the cases.
CHAPTER 7. EMPTINESS OR ONE-REPORTING QUERIES FOR STANDARD INTERSECTION PROBLEMS

The problem which is being tackled in this chapter is abstractly stated next.

- **Emptiness/One-reporting query.** A set $S$ of $n$ geometric objects in $\mathbb{R}^d$ are stored in a data structure so that given a geometric query region $q$, we need to determine efficiently if any object of $S$ intersects with $q$ or not.

Specifically, the following four geometric problems are solved:

1. **Orthogonal Range Emptiness query.** Here $S$ is a set of points and $q$ is an orthogonal hyperbox of the form $[a_1, b_1] \times [a_2, b_2] \times \ldots [a_d, b_d]$. Section 7.3 deals with this problem.

2. **Dominance Emptiness query.** Here $S$ is a set of points and $q$ is an orthant of the form $[a_1, \infty) \times [a_2, \infty) \times \ldots [a_d, \infty)$. Section 7.4 deals with this problem.

3. **Point Enclosure Emptiness query.** Here $S$ is a set of orthogonal hyperboxes and $q$ is a query point, both in $\mathbb{R}^d$. Section 7.5 deals with this problem.

4. **Halfplane Range Emptiness query.** Here $S$ is a set of points and $q$ is the region below a hyperplane. Section 7.6 deals with this problem.

### 7.0.1 Model of Computations

For solving *orthogonal range emptiness query*, our model of computation is the RAM model as modified by Fredman and Willard [39]. In this model it is assumed that each word is of size $w$ and that the number of data elements $n$ never exceeds $2^w$, that is, $w \geq \log_2 n$. In addition, arithmetic and bitwise logical operations take constant time. We shall also refer to it as word-RAM model.

For solving the remaining three problems, the *typical pointer-machine model* is assumed.
CHAPTER 7. EMPTINESS OR ONE-REPORTING QUERIES FOR STANDARD INTERSECTION PROBLEMS

7.0.2 Note on the evaluation of the data structures

The performance of the data structures built in this paper are dependent on the configuration or positioning of the points (or hyperboxes) in the $d$-dimensional space. We will classify our analysis of a data structure into three categories: “best”, “average” and “worst”. “best” (or “worst”) refers to the configuration of points/hyperboxes which leads to the best (or poorest) possible performance of the data structure. “Average” refers to the case where the points/hyperboxes are assumed to be generated randomly. The exact process of generation is defined later.

There has been very little work done specifically on the emptiness queries. Therefore, the performance of the structures built in this paper are compared with the existing structures in the literature which solve the reporting version of it. A reporting or a counting data structure can be trivially modified to answer the emptiness queries. Our structures perform better than these structures in the “best” and the “average” case. In the “worst” case our data structures perform equally with them.

7.1 Results obtained and Comparision with previous results

There has been very little work done exclusively on “One-reporting” problem except for [35]. Dube et. al. [35] initiated the work on “One-reporting” problem for queries of the form $\prod_{i=1}^{d} [a_i, \infty)$. Due to lack of previous work on this problem, we compare our data structures with previous structures which did range searching for query orthogonal box. We briefly review the existing range searching data structures.

Range Trees [9] take up $O(n \log^{d-1} n)$ space and “One-emptiness” queries can be answered by it in $O(\log^{d} n)$ time. The query time can be reduced to $O(\log^{d-1} n)$ by applying fractional cascading technique [23]. Chazelle et. al. [22] further improved the space of range tree. Specifically, for word-RAM model the following structures exist. Alstrup et.
CHAPTER 7. EMPTINESS OR ONE-REPORTING QUERIES FOR STANDARD INTERSECTION PROBLEMS

<table>
<thead>
<tr>
<th>Source</th>
<th>Best</th>
<th>Average</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>$O\left(\frac{\log n}{\log\log n}\right)$</td>
<td>$O\left(\frac{\log n}{\log\log n}\right)$</td>
<td>$O\left(\left(\frac{\log n}{\log\log n}\right)^{d-1}\right)$</td>
</tr>
<tr>
<td>[63]</td>
<td>$O\left(\log^{d-3} n / (\log \log n)^{d-6}\right)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[2, 76]</td>
<td>$O\left(\log^{d-3} n / (\log \log n)^{d-5}\right)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[7]</td>
<td>$O\left(\log^{d-2} n / (\log \log n)^{d-3}\right)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[13]</td>
<td>$O\left(\log^{d-1} n\right)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[23, 22]</td>
<td>$O\left(\log^{d} n\right)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1 The query time of the data structures in the best, average and the worst case scenario for $d>3$ are mentioned. Queries are of the form $\prod_{i=1}^{d}[a_i, b_i]$.

al. [7] came up with a structure that answered queries in $O\left(\log^{d-2} n / (\log \log n)^{d-3}\right)$ time using $O\left(n \log^{d-2+\epsilon} n\right)$ space, $\epsilon > 0$. The query time was improved by Nekrich [76] to $O\left(\log^{d-3} n / (\log \log n)^{d-5}\right)$ but with an increase in space to $O\left(n \log^{d+1+\epsilon} n\right)$. Later, Afshani [2] reduced the space to $O\left(n \log^{d+\epsilon} n\right)$. Recently, Karpinski et. al. [63] gave a structure which uses $O\left(n \log^{d-2+\epsilon} n\right)$ space and answers query in $O\left(\log^{d-3} n / (\log \log n)^{d-6}\right)$ time.

In Table 7.1, the results obtained for the orthogonal range emptiness query are shown and compared w.r.t query time. As can be seen in the table 7.1, our structure clearly outperforms the existing structures in the “best” and “average” case scenario. Even in the “worst” case our structure either performs better or far off from some of the structures by only a small fraction. The efficiency of our structures can be claimed from the observation that a “worst” possible configuration of the points will not happen frequently. Performance of our structure is mentioned in Theorem 7.3.4.

The results obtained on the pointer-machine model are summarized and compared with existing results in Table 7.2, 7.3 and 7.4. We compare the result obtained for each problem with the best known solution to the reporting version of that problem. Our structures do better than the existing counting data structures. Observe that our data structures will never perform poorer than the best known solutions to the reporting version of those problems.
CHAPTER 7. EMPTINESS OR ONE-REPORTING QUERIES FOR STANDARD INTERSECTION PROBLEMS

<table>
<thead>
<tr>
<th>Source</th>
<th>New</th>
<th>Afshani et. al. [2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query time</td>
<td>Best</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>$O((\log \log n)^{d-2})$</td>
</tr>
<tr>
<td></td>
<td>Worst</td>
<td>$O(\log^{d-2} n)$</td>
</tr>
<tr>
<td>Space</td>
<td>Best</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>$O((\log^{d-1} n) \times (\log \log n)^{d-3}))$</td>
</tr>
<tr>
<td></td>
<td>Worst</td>
<td>$O(n \log^{d-3} n)$</td>
</tr>
</tbody>
</table>

Table 7.2 Comparison of results for Dominance emptiness problem. Afshani et. al. [2] is the most efficient solution to the reporting version of this problem.

<table>
<thead>
<tr>
<th>Source</th>
<th>New</th>
<th>Chazelle et. al. [21]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query time</td>
<td>Best</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>$O((\log \log n)^{d-1})$</td>
</tr>
<tr>
<td></td>
<td>Worst</td>
<td>$O(\log^{d-1} n)$</td>
</tr>
<tr>
<td>Space</td>
<td>Best</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>$O((\log^{2d-1} n) \times (\log \log n)^{d-2}))$</td>
</tr>
<tr>
<td></td>
<td>Worst</td>
<td>$O(n \log^{d-2} n)$</td>
</tr>
</tbody>
</table>

Table 7.3 Comparison of results for Point Enclosure emptiness problem. Chazelle et. al. [21] is the most to the reporting version of this problem.
CHAPTER 7. EMPTINESS OR ONE-REPORTING QUERIES FOR STANDARD INTERSECTION PROBLEMS

<table>
<thead>
<tr>
<th>Source</th>
<th>New</th>
<th>Chazelle et. al. [20]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query time</td>
<td>Best</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>$O(\log \log n)$</td>
</tr>
<tr>
<td></td>
<td>Worst</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Space</td>
<td>Best</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td></td>
<td>Worst</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

Table 7.4 Comparison of results for Hyperplane emptiness problem. Chazelle et. al. [20] is the most efficient solution to the reporting version of this problem.

7.2 Definitions

Definition 7.2.1. (Dominance Criteria). Dominance Criteria “C” is defined as a set where each element $C[i] = \leq$ or $\geq$, $\forall 1 \leq i \leq |C|$, i.e., each element either takes the value $\leq$ or $\geq$.

Definition 7.2.2. (Dominance). Let $p_1=(x_1, \ldots, x_d)$ and $p_2=(y_1, \ldots, y_d)$ be two $d$-dimensional points. Also, consider a particular dominance criteria $C$ of size $d$. If $x_i \geq y_i, \forall 1 \leq i \leq d$, then we define that $p_1$ dominates $p_2$ and that $p_2$ is dominated by $p_1$. In other words, $\forall 1 \leq i \leq d$, if $C[i]$ is $\geq$ and $x_i \geq y_i$; and if $C[i]$ is $\leq$ and $x_i \leq y_i$, then $p_1$ dominates $p_2$ and $p_2$ is dominated by $p_1$.

Suppose in a 2d plane we are given two points $p(10, 12)$ and $q(5, 6)$. Let the dominance criteria $C=\{\geq, \geq\}$. Then $p$ is said to dominate $q$ since $10 \geq 5$ and $12 \geq 6$. 

118
### 7.3 Orthogonal Range Emptiness query

**Problem.** A set $S$ of $n$ points in $\mathbb{R}^d$ are stored in a data structure so that given an orthogonal query hyperbox $q=\Pi_{i=1}^d [a_i, b_i]$, we need to determine efficiently if any point of $S$ intersects with $q$ or not. The computational model assumed is a word-RAM model.

We shall start of by answering orthant queries of the form $q=\Pi_{i=1}^d [a_i, \infty)$. Then we go on to answer bounded queries of the form $q=\Pi_{i=1}^d [a_i, b_i]$. We shall interchably use the terms Emptiness and One-reporting.

In this section we shall consider Dominance Criteria $C = \{\geq, \geq, \ldots, \geq\}$ and $|C| = d$ when dealing with points lying in a $d$-dimensional space. Then a point $p_1=(x_1, \ldots, x_d)$ will dominate another point $p_2=(y_1, \ldots, y_d)$ iff $x_i \geq y_i, \forall 1 \leq i \leq d$.

#### 7.3.1 One-reporting dominance queries in $\mathbb{R}^2$

**Definition 7.3.1. (Maximal Point).** Let $S$ be a set of points. A point $p_i \in S$ is a maximal point if there is no other point $p_j \in S$, such that $p_j$ dominates $p_i$.

For the given point set $S$, we denote $M (\subseteq S)$ to be the set of maximal points.

**Lemma 7.3.1.** For a point set $S$ and a query quadrant $q=\Pi_{i=1}^d [a_i, \infty)$, $S \cap q \neq \emptyset$ iff $M \cap q \neq \emptyset$. Therefore, it is enough to consider only the maximal points ($M \subseteq S$) for answering “One-reporting dominance problem”.

**Proof.** First assume $M \cap q \neq \emptyset$. Then it is trivial to observe that there is atleast one point of $S$ lying within $q$. Now assume $S \cap q \neq \emptyset$. Pick any point $p_i \in S \cap q$. If $p_i$ happens to be a maximal point then the lemma holds true. However, if $p_i$ is not a maximal point then there must be atleast one point $p_j \in M$ which dominates $p_i$. By Definition 7.2.2 since $p_i$ lies within $q$, even $p_j$ has to lie within $q$. Therefore, $M \cap q \neq \emptyset$. Hence, proved.

We start building our solutions by considering dominance queries in $\mathbb{R}^2$. Specifically, for a point set $S$ in $\mathbb{R}^2$ and a given query quadrant $q=[a_1, \infty) \times [a_2, \infty)$, we need to report YES if $S \cap q \neq \emptyset$, else report NO.
CHAPTER 7. EMPTINESS OR ONE-REPORTING QUERIES FOR STANDARD INTERSECTION PROBLEMS

Lemma 7.3.2. For a point set S, let M be the set of maximal points. Consider a point \( p_i(i_x, i_y) \in M \). Now all the points \( p_j(j_x, j_y) \in M \), which satisfy the condition \( j_x > i_x \), will lie below the line \( y = i_y \).

Firstly, following Lemma 7.3.1 we find out the set of maximal points \( M \) in \( S \) [31]. A Static Fusion Tree, \( T \), [39] is built based on the \( x \)-coordinates of the maximal points \( M \). Given a query \( q = [a_1, \infty) \times [a_2, \infty) \), we perform a successor query on \( T \) with \( a_1 \). Let \( p(p_x, p_y) \) be the point reported. Clearly, \( p_x \geq a_1 \). If \( p_y \geq a_2 \), then we report YES. Else if \( p_y < a_2 \), then from Lemma 7.3.2 it can be inferred that the other points in \( M \) which have \( x \)-coordinate \( \geq a_1 \), will also have their \( y \)-coordinate values \( < a_2 \). Hence, NO will be reported. Also, while performing a successor query if no point is found, report NO.

Theorem 7.3.1. A set \( S \) of \( n \) points in \( \mathbb{R}^2 \) having \( m \) maximal points can be preprocessed into a data structure of size \( O(m) \), such that given a query quadrant \( q = [a_1, \infty) \times [a_2, \infty) \), the “One-reporting dominance” query can be answered in \( O(\log m / \log \log n) \) time.

Proof. A static fusion tree when built on \( m \) points takes up linear space. It has a branching factor of \( O(\log^{1/5} n) \) at each node. Therefore, its height is \( O(\log m / \log \log n) \) and hence answers successor query in \( O(\log m / \log \log n) \) time. \( \square \)

If the points in \( S \) are assumed to be randomly generated on a plane, then expected (or average) number of maximal points is \( O(\log n) \) [10]. In the best case the no. of maximal points will be \( O(1) \) and in the worst case the no. of maximal points will be \( O(n) \). This leads to the following corollary.

Corollary 7.3.1. Let \( S(n) \) and \( Q(n) \) denote the space and the query time of the above data structure. Then the following results can be inferred from the above discussion :-

1. \( S(n) = O(1) \) and \( Q(n) = O(1) \), in the best case.

2. \( S(n) = O(\log n) \) and \( Q(n) = O(1) \), in the average (or expected) case.

3. \( S(n) = O(n) \) and \( Q(n) = O(\log n / \log \log n) \), in the worst case.
7.3.2 One-reporting Dominance problem in IR^3

In this section we provide a solution to the One-reporting dominance problem on IR^3 (referred to as xyz-space in this section). First, we find out the maximal points (M) of S, w.r.t., xyz-space [31]. The primary structure will be a Static Fusion Tree, D, built based on the z-coordinates of the points in M. The points in D are stored in non-decreasing order of their z-coordinate values. p(v) denotes the set of points lying in the subtree rooted at an internal node v ∈ D. For an arbitrary internal node v ∈ D, let v_1, v_2, ..., v_k be its children from left to right. With each child node v_i, we associate a point set P_i as follows: P_i = \bigcup_{j=1}^{k} p(v_j). Each point set P_i is projected onto the xy-plane and based on the xy-projections of points in P_i, its maximal points M_i are found out (the z-coordinates are ignored). At node v_i, based on these maximal points M_i we build a secondary structure of Theorem 7.3.1 to handle “One-reporting” dominance queries in the plane.

Given a query q = [a_1, ∞) × [a_2, ∞) × [a_3, ∞), we first run a successor query on the primary structure of D with a_3. Let v be the leaf node selected. Now we choose some cannonical nodes in D and on each of them a secondary query is performed. Let C = v_1, v_2, ..., v_l, where v_i is the immediate right sibling of the i^{th} node on the path from v to the root, excluding v and root. v_i will not exist if this i^{th} node turns out to be the rightmost child. Then C ∪ v form our cannonical nodes. We query the secondary structures at each of the cannonical nodes with q' = [a_1, ∞) × [a_2, ∞). If any of the secondary structures reports the presence of a point in q', then we say YES, else we say NO.

**Theorem 7.3.2.** A set S of n points in IR^3 having m maximal points can be preprocessed into a data structure of size O(m \times \frac{\log m}{\log \log n} \times \log^{1/5} n), such that given a query q = [a_1, ∞) × [a_2, ∞) × [a_3, ∞), the “One-reporting dominance” query can be answered in O((\log m/ \log \log n)^2) time.

**Proof.** The height of the primary structure when built on m points will be O(\log m/ \log \log n). Each point p ∈ S is associated with O(\log^{1/5} n) internal nodes at each level of the tree. Since the secondary structure takes up linear space in the worst case, the total space occu-
CHAPTER 7. EMPTINESS OR ONE-REPORTING QUERIES FOR STANDARD INTERSECTION PROBLEMS

pied will be $O(m \times \frac{\log m}{\log \log n} \times \log^{1/5} n)$. Given a query $q$, $O(\frac{\log m}{\log \log n})$ canonical nodes are selected. The time taken to query each secondary structure will be $O(\frac{\log m}{\log \log n})$. Therefore, the total query time is $O((\log m/ \log \log n)^2)$. 

If the points in $S$ are assumed to be randomly generated on a $xyz$-space, then the expected (or average) number of maximal points is $O(\log^2 n)$ [10]. In the best case the no. of maximal points will be $O(1)$ and in the worst case the no. of maximal points will be $O(n)$. This leads to the following corollary.

Corollary 7.3.2. Let $S(n)$ and $Q(n)$ denote the space and the query time of the above data structure. The following results can be inferred:-

1. $S(n)=O(1)$ and $Q(n)=O(1)$, in the best case.

2. $S(n)=O(\log^{11/5} n)$ and $Q(n)=O(1)$, in the average (or expected) case.

3. $S(n)=O(n(\log^{4/5} n/ \log \log n))$, $Q(n)=O((\log n/ \log \log n)^2)$, in the worst case.

7.3.3 One-reporting dominance problem in $\mathbb{R}^d$, $d > 3$

In this section we shall generalize the the solution built for $d = 3$ to higher dimensional points as well. Assume that we already have a data structure for this problem in $\mathbb{R}^{d-1}$. The construction is shown in an inductive manner. Let $x_1, x_2, \ldots, x_d$ be the individual coordinates of our $d$-dimensional space. First, we find out the maximal points ($M$) of $S$, w.r.t., $\mathbb{R}^d$ [31]. The primary structure will be a Static Fusion Tree, $D$, built based on the $x_d$-coordinates of the points in $M$. Then as done for $d = 3$, at each internal node an instance of the “One-reporting dominance” problem for $\mathbb{R}^{d-1}$ is built. The query algorithm is similar to the one described in the previous section.

If the points in $S$ are assumed to be randomly generated in $\mathbb{R}^d$, then expected (or average) number of maximal points is $O(\log^{d-1} n)$ [10]. In the best case the no. of maximal points will be $O(1)$ and in the worst case the no. of maximal points will be $O(n)$. This leads to the following corollary.
Corollary 7.3.3. A set $S$ of $n$ points in $\mathbb{R}^d$ having $m$ maximal points can be preprocessed into a data structure of size $O(S(n))$, such that given a query region $\Pi_{i=1}^d [a_i, \infty)$, the “One-reporting dominance” query can be answered in $O(Q(n))$ time, where

1. $S(n) = O(1)$ and $Q(n) = O(1)$, in the best case.
2. $S(n) = O(\log^{d-4/5} n)$ and $Q(n) = O(1)$, in the average (or expected) case.
3. $S(n) = O(n(\log^{6/5} n / \log \log n)^{d-2})$, $Q(n) = O((\log n / \log \log n)^{d-1})$, in the worst case.

7.3.4 One-reporting for bounded orthogonal rectangle queries on $\mathbb{R}^2$

Now, we generalize our queries to orthogonal bounded rectangles on $\mathbb{R}^2$. First we consider queries of the form $q' = [a_1, b_1] \times [a_2, \infty)$. The solution is based on the structure built in Theorem 7.3.1.

Based on the $x$-coordinates of the points in $S$ we build a static fusion tree $SFT$. We store the points of $S$ sorted by $x$-coordinate at the leaves of a complete balanced binary tree $T'$. At each internal node $v$, we store an instance of the structure of Theorem 7.3.1 for handling queries of the form $[a_1, \infty) \times [a_2, \infty)$ (resp., $(-\infty, b_1] \times [a_2, \infty)$) built on the points in $v$’s left (resp., right) subtree. Let $X(v)$ denote the average of the $x$-coordinate in the rightmost leaf in $v$’s left subtree and the $x$-coordinate in the leftmost leaf in $v$’s right subtree; for a leaf we take $X(v)$ to the $x$-coordinate of the point stored at $v$. The root of $T'$ holds a pointer to $SFT$.

Given a query $q'$, we first find out the successor of $a_1$ (say $a'_1$) and the predecessor of $b_1$ (say $b'_1$) in $SFT$. Leaf nodes of points $a'_1$ and $b'_1$ are found out in the primary structure of $T$. This identification can be done in $O(1)$ time if we maintain appropriate pointers. Since $T$ is a complete balanced binary tree, the LCA (least common ancestor) of $a'_1$ and $b'_1$ can found out in $O(1)$ time. Let the LCA be $v$. Then we query the structures at $v$ using the $NE$-quadrant and the $NW$-quadrant derived from $q'$ (i.e. the quadrants with corners at $(a_1, a_2)$ and $(b_1, a_2)$, respectively). If any of the structure reports YES, then the overall
CHAPTER 7. EMPTINESS OR ONE-REPORTING QUERIES FOR STANDARD INTERSECTION PROBLEMS

answer will be YES. If both the structures report NO, then the overall answer will be NO.

To solve the problem for general bounded orthogonal rectangles \( q = [a_1, b_1] \times [a_2, b_2] \), we use the above approach again, but now the points in the tree \( T \) are stored by sorted \( y \)-coordinates. At each internal node \( v \) of \( T \), we store an instance of the data structure above to answer queries of the form \( [a_1, b_1] \times [a_2, \infty) \) (resp. \( [a_1, b_1] \times (-\infty, b_2] \)) on the points in \( v \)'s left (resp. right) subtree. Also, a static fusion tree is built based on the \( y \)-coordinates of the points in \( S \) and a pointer from the root of \( T \) to this structure is maintained. The query strategy is similar to the previous one, but now we use the interval \( [a_2, b_2] \) to search in \( T \).

**Theorem 7.3.3.** A set \( S \) of \( n \) points on \( \mathbb{R}^2 \) can be preprocessed into a data structure of size \( O(S(n)) \) such that given a query orthogonal box \( q = [a_1, b_1] \times [a_2, b_2] \), the “One-reporting problem” can be solved in \( O(Q(n)) \) time, where

1. \( S(n) = O(n \log n) \) and \( Q(n) = O(\log n / \log \log n) \), in the best case.
2. \( S(n) = O(n \log n) \) and \( Q(n) = O(\log n / \log \log n) \), in the average (or expected) case.
3. \( S(n) = O(n \log^2 n) \) and \( Q(n) = O(\log n / \log \log n) \), in the worst case.

### 7.3.5 One-reporting queries for bounded orthogonal boxes in \( \mathbb{R}^d \)

Here we generalize our queries to orthogonal bounded boxes in \( \mathbb{R}^d \), i.e., the query box \( q = [a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_d, b_d] \). The solution built here is an extension of the structure built in Theorem 7.3.3 for \( d=2 \). First we build a structure for handling queries of the form \( q' = [a_1, b_1] \times \prod_{i=2}^{d} [a_i, \infty) \) by the same technique that was used in the previous section. In the same manner, we next build a structure for handling queries of the form \( q'' = [a_1, b_1] \times [a_2, b_2] \times \prod_{i=3}^{d} [a_i, \infty) \). In this way we iteratively build a structure \( D \) which finally handles queries \( q = \prod_{i=1}^{d} [a_i, b_i] \). The space occupied by \( D \) will increase by a factor of \( O(\log^{d-2} n) \) compared to the structure built for \( d=2 \) in Theorem 7.3.3.
CHAPTER 7. EMPTINESS OR ONE-REPORTING QUERIES FOR STANDARD INTERSECTION PROBLEMS

Theorem 7.3.4. A set $S$ of $n$ points on $\mathbb{R}^d$ can be preprocessed into a data structure of size $O(S(n))$ such that given a query orthogonal box $q=[a_1, b_1] \times [a_2, b_2] \times \ldots [a_d, b_d]$, the “One-reporting problem” can be solved in $O(Q(n))$ time, where

1. $S(n)=O(n \log^{d-1} n), Q(n)=O(\log n / \log \log n)$, in the best case.
2. $S(n)=O(n \log^{d-1} n), Q(n)=O(\log n / \log \log n)$, in the average (or expected) case.
3. $S(n)=O(n \log^d n), Q(n)=O((\log n / \log \log n)^{d-1})$, in the worst case.

7.4 Dominance Emptiness query

Problem. Preprocess a set $S$ of $n$ points in $\mathbb{R}^d$, such that given a query $q = \Pi_{i=1}^{d} [a_i, \infty)$ report if at least one point of $S$ lies in $q$.

In this section we shall consider Dominance Criteria $C = \{\geq, \geq, \ldots, \geq\}$ and $|C| = d$ when dealing with points lying in a $d$-dimensional space. Then a point $p_1=(x_1, \ldots, x_d)$ will dominate another point $p_2=(y_1, \ldots, y_d)$ iff $x_i \geq y_i, \forall 1 \leq i \leq d$.

Definition 7.4.1. (Maximal Point). Let $S$ be a set of points in $\mathbb{R}^d$. A point $p_i \in S$ is a maximal point if there is no other point $p_j \in S$, such that $p_j$ dominates $p_i$.

Lemma 7.4.1. For a given point set $S$, we denote $M (\subseteq S)$ to be the set of maximal points. Given a query $q=\Pi_{i=1}^{d} [a_i, \infty)$, $S \cap q\neq\phi$ iff $M \cap q\neq\phi$. Therefore, it is enough to consider only the maximal points ($M \subseteq S$) for answering “Dominance Emptiness query”.

Following Lemma 7.4.1 we find out the set of maximal points $M$ in $S$ [31]. If a static structure is needed, then based on the points in $M$ a structure $D$ as described in [2] is built. Given a query $q$, $D$ will find out if any point of $M$ lies within $q$. It is clear that the performance of $D$ is directly dependent on the size of $M$, i.e., the number of maximal points. In the best (or worst) case the number of maximal points in set $S$ can be $O(1)$ (or $O(n)$). If the points in $S$ are assumed to be randomly generated in $\mathbb{R}^d$, then expected (or average) number of maximal points is $O(\log^{d-1} n)$ [10]. See Table 7.2 for comparison.
CHAPTER 7. EMPTINESS OR ONE-REPORTING QUERIES FOR STANDARD INTERSECTION PROBLEMS

Suppose we want to build a data structure which can also handle insertions efficiently. This can be done by dynamically updating set $M$ of maximal points. When a new point $p$ is added to set $S$, if $p$ is not a maximal point then it is discarded; else $p$ is added to set $M$. All the points in $M$ which are dominated by $p$ are removed from $M$ since they are no longer maximal points. Finally, given a query $q$ we need to check if any point in $M$ lies within $q$.

To carry out all these operations we shall use a dynamic dominance reporting data structure $D$ as described in [57]. Though we are using a dynamic data structure, our final emptiness data structure won’t be able to handle deletions. This is due to the fact that when we delete a point in $M$, some other point(s) previously not in $M$ might now become maximal point(s).

However, following Lemma 7.4.1, in the preprocessing stage we are storing only the set $M$ and the remaining points are discarded to reduce the space occupied by our final structure.

**Theorem 7.4.1.** A set $S$ of $n$ points in $d$-dimensional space can be preprocessed into a data structure of size $S(n)$, such that given a query point $q$ the emptiness problem can be solved in $Q(n)$ time and insertion of a new point can be handled in $I(n)$ amortized time, where

1. $S(n) = O(1)$, $Q(n) = O(1)$, $I(n) = O(1)$, in the best case.

2. $S(n) = O((\log^{d-1} n) \times (\log \log n)^{d-2})$, $Q(n) = O((\log \log n)^{d-1})$, $I(n) = O((\log \log n)^{d-1})$, in the average case.

3. $S(n) = O(n \log^{d-2} n)$, $Q(n) = O(\log^{d-1} n)$, $I(n) = O(\log^{d-1} n)$, in the worst case.

**Proof.** Let $m$ be the number of maximal points ($M$) of set $S$. When $D$ is built on $m$ points it takes $O(m \log^{d-2} m)$ space and answers queries in $O(\log^{d-1} m)$ time. It handles insertion and deletion of a point in $O(\log^{d-1} m)$ amortized time. Suppose a point $p$ is added to $S$ and let it be a maximal point. So, it is inserted into $D$ and $\lambda$ be number of points in $M$ which are dominated by $p$ due to which they need to be deleted from $D$. The time taken to update $D$ will be $O(\log^{d-1} m + \lambda \log^{d-1} m)$. If we assume an arbitrary insertion of $m$ points into $S$ then the total time taken to update $D$ will be $O(m \log^{d-1} m + \sum_{i=1}^{m} \lambda \log^{d-1} m)$. However, observe that a point can get deleted only once from $D$. Therefore, the total taken to update
CHAPTER 7. EMPTINESS OR ONE-REPORTING QUERIES FOR STANDARD INTERSECTION PROBLEMS

$D$ over $m$ insertions will be $O(m \log^{d-1} m)$. The amortized time will be $O(\log^{d-1} m)$. $m$ is $O(1)$ (or $O(n)$) in the best (or worst) case. If the points are assumed to be randomly generated then $m$ will be $O(\log^{d-1} n)$ (average case). This leads to the above theorem.

7.5 Point Enclosure emptiness query

**Problem.** Preprocess a set $S$ of $n$ hyperboxes in $\mathbb{R}^d$, such that given a query point $q \in \mathbb{R}^d$, report if at least one hyperbox of $S$ is stabbed by $q$.

**Definition 7.5.1. (Dominant Hyperbox).** Given two hyperboxes $r_1 \in S$ and $r_2 \in S$, $r_1$ becomes a Dominant Hyperbox w.r.t. $r_2$ iff $r_2 \subseteq r_1$, i.e., $r_2$ is completely contained within $r_1$.

**Definition 7.5.2. (Maximal Hyperbox).** In a given set $S$ of hyperboxes, a hyperbox $r \in S$ is a “Maximal Hyperbox” iff there exists no hyperbox $r' \in S \setminus r$, such that $r'$ is a dominant hyperbox w.r.t. $r$ (i.e. $r \not\subseteq r'$).

An interesting point to be observed is that our entire set $S$ of hyperboxes need not be always stored to answer the emptiness query. All those hyperboxes in $S$ which satisfy the definition of Maximal Hyperbox (stated above) are the only hyperboxes which are enough to answer an emptiness query.

7.5.1 Data Structure

Accordingly, in the preprocessing phase we try to find out the Maximal Hyperboxes as follows: Each hyperbox denoted by $r = [a_1, b_1] \times \ldots \times [a_d, b_d]$, is mapped to a $2d$-dimensional point $r'(-a_1, -a_2, \ldots, -a_d, b_1, b_2, \ldots, b_d)$. Denote these new set of points in $\mathbb{R}^{2d}$ by $S'$. Now a hyperbox $r \in S$ is a Maximal Hyperbox in $\mathbb{R}^d$ iff its corresponding point in $\mathbb{R}^{2d}$ is a maximal point among $S'$ for the Dominance Criteria $C = \{\leq, \geq, \leq, \geq, \ldots, \leq, \geq\}$. of size
2d. We find out the maximal points of $S'$. Let $M$ be the set of corresponding Maximal Hyperboxes, s.t., $\forall r \in M$, $r'$ is a maximal point among $S'$. Based on the hyperboxes in $M$ a standard data structure $D$ is built which given a query point in $\mathbb{R}^d$, reports all the hyperboxes stabbed by the query point [21].

Given a query point $q$, $D$ is queried with $q$ and the moment even one hyperbox is found to be stabbed, we stop and report “yes”. If no hyperbox is stabbed by $q$, then we report “no”.

7.5.2 Analysis

Let us now analyze the data structure $D$. $D$ is a structure which when built on $m$ hyperboxes in $d$-dimensional space takes up $O(m \log^{d-2} m)$ space and answers stabbing queries in $O(\log^{d-1} m + k)$ time, where $k$ is the number of hyperboxes stabbed [21]. In the best case, the number of hyperboxes stored in $D$ will be $O(1)$ and in the worst case, the number of hyperboxes stored in $D$ will be $O(n)$.

The average case is defined by a random generation of hyperboxes. Consider two different dimensional spaces $\Gamma_1$ and $\Gamma_2$. $\Gamma_1$ is a $d$-dimensional space in which hyperboxes of set $S$ lie. $\Gamma_2$ is a $2d$-dimensional space. In $\Gamma_2$ $2d$-dimensional points are generated which are mapped to $d$-dimensional hyperboxes in $\Gamma_1$. The mapping is done as follows: $n$ points are randomly generated in $\Gamma_2$. By random, we mean that for each point the coordinate value in each dimension is an independently generated random real number. Call the set of random points $S'$. A point $p(x'_1, x''_1, x'_2, x''_2, \ldots, x'_d, x''_d) \in S'$ in $\Gamma_2$ gets mapped to a hyperbox $r = [l_1, r_1] \times \ldots \times [l_d, r_d] \in S$ in $\Gamma_1$, where $l_i = \min\{x_i', x''_i\}$ and $r_i = \max\{x_i', x''_i\}$, $\forall 1 \leq i \leq d$. Since, points in $\Gamma_2$ are generated randomly and the fact that each hyperbox in $\Gamma_1$ can be generated by $2^d$ distinct points in $\Gamma_2$, it turns out that hyperboxes in $\Gamma_1$ are also generated randomly.

Observe that if a hyperbox in $S$ is a Maximal Hyperbox, then the corresponding point in $S'$ in $\Gamma_2$ is also a maximal point under the Dominance Criteria $C = \{\leq, \geq, \leq, \geq, \ldots\$. 
≤, ≥}, i.e., for odd dimensions ≤ and for even dimensions ≥. Since, the points have been randomly generated, the average number of maximal points in \( \mathbb{R}^{2d} \) will be \( O(\log^{2d-1} n) \). Hence, the average number of Maximal Hyperboxes will also be \( O(\log^{2d-1} n) \).

**Theorem 7.5.1.** A set \( S \) of \( n \) hyperboxes in \( d \)-dimensional space can be preprocessed into a data structure of size \( S(n) \), such that given a query point \( q \) the “One-reporting” problem can be solved in \( Q(n) \) time, where

1. \( S(n) = O(1), Q(n) = O(1), \) in the best case.
2. \( S(n) = O((\log^{2d-1} n) \times (\log \log n)^{d-2}), Q(n) = O((\log \log n)^{d-1}), \) in the average case.
3. \( S(n) = O(n \log^{d-2} n), Q(n) = O(\log^{d-1} n), \) in the worst case.

### 7.6 Halfplane Range emptiness query

**Problem.** Preprocess a set \( S \) of \( n \) points in a \( xy \)-plane, so that given a query halfplane \( Q \), report if there is any point of \( S \) lying in the closed halfplane \( Q^- \) (i.e. on/below \( Q \)).

![Figure 7.1](Image)

**Figure 7.1** \( P_1, P_2, P_3 \) and \( P_4 \) are the maximal points under the Dominance Criteria \( C = \{≥, ≤\}. \)

The query halfplane \( Q \) can be categorized into three categories based on its slope and we shall discuss how to handle each case. The cases are:
CHAPTER 7. EMPTINESS OR ONE-REPORTING QUERIES FOR STANDARD INTERSECTION PROBLEMS

Figure 7.2 P1, P5 and P6 are the maximal points under the Dominance Criteria \( C = \{ \leq, \leq \} \)

1. **Q is a vertical line**, i.e., slope of \( Q \) is \( \infty \). Let us assume that \( Q^- \) in this case is the portion of the plane to the right of \( Q \). To handle this query, all we need to do is to store the point in \( S \) with the maximum \( x \)-coordinate. Rest of the points are not needed for answering this query.

2. **Q has a positive slope**. Consider two points \( p_1 \) and \( p_2 \) of \( S \) such that \( p_2 \) lies in the South-East quadrant w.r.t. \( p_1 \). For a given query \( Q \) (having a +ve slope) if \( p_1 \) lies in \( Q^- \), then \( p_2 \) will definitely lie in \( Q^- \). However, the reverse is not true. Thus all the points in \( S \) having at least one point of \( S \) lying in their South-East quadrant can be safely eliminated. Only points which have no point of \( S \) in their South-East quadrant are enough to answer our query. Call this set \( S_+ \). Observe that \( S_+ \) is nothing but maximal points of \( S \) under the Dominance Criteria \( C = \{ \geq, \leq \} \). In Figure 7.1 we show 12 points on a plane. \( P1, P2, P3 \) and \( P4 \) are the points which have no other point in their South-East quadrant and form the set \( S_+ \). These four points are enough to answer an emptiness version of halfplane range searching where the query has a positive slope. For a query \( Q \), if there exists any point in \( Q^- \), then at least one among \( P1, P2, P3 \) and \( P4 \) will lie in \( Q^- \).

3. **Q has a negative slope**. Following from the previous case it can be observed that
if $Q$ has a negative slope then points in $S$ which have no other points of $S$ in their South-West quadrant are enough to the answer the query. Call this set of points $S_-$. $S_-$ is nothing but maximal points of $S$ under the Dominance Criteria $C = \{ \leq, \leq \}$. In Figure 7.2, for the same set of 12 points $P_1$, $P_5$ and $P_6$ form the set $S_-$. 

Case 1 is quite trivial to handle. We shall now discuss the data structure that is built to handle queries of type Case 2. Let $\pi$ denote the well known point-hyperplane duality transform [37]. If $p = (p_1, p_2)$ is a point in $\mathbb{R}^2$, then $\pi(p)$ is the hyperplane $y = p_1 x - p_2$. If $H : y = a_1 x + a_2$, is a non-vertical hyperplane in $\mathbb{R}^2$, then $\pi(H)$ is the point $(a_1, -a_2)$. It can be easily verified that $p$ is above (resp. on, below) $H$, in the $y$-direction, if and only if, $\pi(p)$ is below (resp. on, above) $\pi(H)$. Also, $\pi(\pi(p)) = p$ and $\pi(\pi(H)) = H$.

First, we find out the maximal points of $S$ (i.e $S_+$) under the Dominance Criteria $C = \{ \geq, \leq \}$. Using $\pi$ we map $S_+$ to a set $S'_+$ of hyperplanes and map $Q$ to a point $q=\pi(Q)$, both in $\mathbb{R}^2$. The problem is now equivalent to: “Is there any hyperplane lying on or above $q$, i.e., is there any hyperplane that is intersected by the vertical ray $r$ emanating upwards from $q$.”

We shall construct an upper envelope $E$ of the hyperplanes in $S'_+$. Then by definition of upper envelope $r$ intersects a hyperplane in $S'_+$ if and only if $r$ intersects $E$, and $r$ can intersect at most one segment of the upper envelope $E$. Based on these observations, it is enough if we solve the following problem: “Check if any segment of $E$ is intersected by $r$.” Since all the segments in the upper envelope are disjoint, we maintain a balanced binary tree $T$ with each leaf corresponding to a segment in $E$. To answer a query $q(q_x, q_y)$ with $Q$ having a positive slope, we do a search in $T$ with $q_x$ and $l$ be the segment in $E$ whose $x$-coordinates of the endpoints lie on either side of $q_x$. So, $l$ is the only segment which can be stabbed by the ray emanating from $q$. We next simply check if $q$ lies on/below $l$ or above $l$. If $q$ lies on/below $l$, then report “yes” else report “no”.

Case 3 is analogous to Case 2 and can be handled in the same manner as done for Case 2. The final result is stated next.
CHAPTER 7. EMPTINESS OR ONE-REPORTING QUERIES FOR STANDARD INTERSECTION PROBLEMS

Theorem 7.6.1. A set \( S \) of \( n \) points in a plane can be preprocessed into a data structure of size \( S(n) \), such that given a query halfplane \( Q \) the One-reporting version of halfplane range searching can be solved in \( Q(n) \) time, where

1. \( S(n) = O(1), Q(n) = O(1) \) in the best case.

2. \( S(n) = O(\log n), Q(n) = O(\log \log n) \) in the average (or expected) case.

3. \( S(n) = O(n), Q(n) = O(\log n) \) in the worst case.

Proof. Case 1 can be handled by using \( O(1) \) space and given a query vertical line, it can be answered in \( O(1) \) time. Consider Case 2. Let \( m \) be the number of maximal points of \( S \) under the dominance criteria \( C = \{\geq, \leq\} \). After applying duality, the size of the upper envelope will be bounded by \( m \). Since \( T \) is a binary tree built over \( m \) items, it takes \( O(m) \) space and answers queries in \( O(\log m) \) time. \( m \) will be \( O(1) \) and \( O(n) \) in the best and the worst case, respectively. If the points of \( S \) are assumed to be generated randomly (as defined in Sec. 2 and Sec. 3), then \( m \) will be \( O(\log n) \). Similar analysis holds for Case 3 as well. This leads us to the final result.

7.7 Conclusions and open problems

In the chapter we tried to look at the one-reporting version of standard geometric intersection searching problems. The major limitation of this work is that even though it provides improved solutions for the average case, in the worst case the space and time bounds are the same as the existing solutions for the reporting version. Naturally the most important open problem would be to try to come up with solutions for one-reporting version which perform better than the reporting version even in the worst case scenario. Proving lower bounds for the one-reporting version of these problems will help us in understanding if it is really possible.
Chapter 8

Conclusions and Open Problems

In this thesis, we motivated the need to work on range-aggregate queries. Existing data structures and algorithmic techniques cannot be trivially used to obtain efficient solutions for range-aggregate queries. We considered different kinds of range-aggregate queries (distributive aggregation, geometric aggregation etc.) and came up efficient data structures and query algorithms for each of them. The efficiency of the solutions was determined based on the space occupied by the structure and the time taken to answer a query. Luckily, for most of the problems it was possible to satisfy both the requirements.

Open problems specific to each chapter have been mentioned at the end of those chapters. There are a lot of directions in which the work on range-aggregate queries can be taken forward to increase our understanding of them. We mention some of them:

1. Simplified Solutions: In this thesis, our objective was to come up with solutions with optimal worst-case bound. However, in some scenarios we are guaranteed that the queries which lead to worst case performance will appear rarely. For such cases it would be nice to come up with simplified solutions which handle average case queries well. This will not hurt the efficiency of the solution much while simplifying the implementation a lot.

2. Proving Lower bounds: Finding upper bounds and lower bounds for a problem (say
CHAPTER 8. CONCLUSIONS AND OPEN PROBLEMS

$P$) helps us in determining if an optimal solution has been reached for $P$ or further work is needed to reach the optimal solution. For eg., consider comparison based sorting. Sorting algorithms such as mergesort and quicksort can perform sorting in $O(n \log n)$ time. Hence, the upper bound for sorting is $O(n \log n)$. Now people were curious if comparison based sort could be done better than $O(n \log n)$. Then a lower bound of $\Omega(n \log n)$ was found out for comparison based sort which meant that mergesort and quicksort are indeed optimal solutions for sorting. There are many problems in the literature whose upper bounds and the lower bounds still have a significant gap.

In this thesis, we have come up with upper bounds for various range-aggregate problems. To know if there is further scope for improvement of those solutions, it is critical to come up with lower bounds for them. In general, proving lower bounds is significantly tougher than proving upper bounds!

3. Other memory models: For the problems considered in this thesis, we have come up with internal memory or main memory algorithms. It would be interesting to solve these problems on other models of computations. This could mean coming up with I/O-efficient algorithms on slow (external) mass storage devices, cache-oblivious algorithms on systems with complicated memory hierarchies or streaming algorithms on data that naturally arrive continually in a streaming way.

4. Approximate solutions: With the exponential growth in size of datasets, there is a feeling among many researchers in the community that coming up with exact solutions would no longer be feasible and in many cases unnecessary. Approximate solutions for range-aggregate problems will help in coming up with faster and low-space solutions while providing the user with meaningful results (if not exact).

5. Non-orthogonal objects: We have mostly considered the objects in our dataset and the query objects to be orthogonal (except for problems which are variations of half-
space range searching). Considering non-orthogonal objects (such as polygons, balls and spheres) would be a challenging field to look into.

6. Fewer index structures: Suppose we want to build a system on a massive dataset which needs to handle different kinds of queries. Practically, it would be impossible to have a separate index structure for answering each kind of query. Having minimal number of index structures would be the objective in such a system. Therefore, it would be nice if we consider a set of queries and try to come up with a single index structure for all of them (though the query algorithm can differ for each kind of query). This will have to be looked into carefully by the computational geometry community in future.
Bibliography


BIBLIOGRAPHY


