3D RANGE SEARCHING USING CHAIN DECOMPOSITION

by

Yingda Guo

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
August 2017

To this wonderful world

Table of Contents

List of Tables

List of Figures

Abstract

We propose the first data structure based on untangled monotonic chains for orthogonal range searching in 3-dimensional space. The idea is an extension of the 2-dimensional chain partition algorithm proposed by Arroyuelo et al. (Untangled monotonic chains and adaptive range search, Theoretical Computer Science, 412(32), 2011). We also provide an improved algorithm for the chain untangling process, which in practice runs 25% percent faster than the untangling algorithm proposed in the experimental studies of Claude et al. (Range queries over untangled chains, SPIRE, Springer 2010). In the experimental evaluations, we first re-examined the experimental studies conducted by Claude et al. for the 2-dimensional range searching algorithm based on untangled monotonic chains and found that the k-d tree implementation in CGAL, which is used as a reference for the experimental evaluation previously, is inefficient for the task at hand. Therefore, we implemented k-d trees ourselves and compared them against two range searching methods based on untangled chains. The experimental results showed that, in 2D, the performance of range searching methods based on untangled monotonic chains is similar to that of the k-d tree, which contradicts the experimental results of Claude et al. We then performed similar experimental studies in three dimensions. In 3D, the chain-decomposition-based range searching methods were unable to match the performance of k-d trees, which is mainly due to the difficulties in decomposing point sets into monotonic chains: our approaches either generated too many chains or used too much time to construct when the point set was large.

List of Abbreviations and Symbols Used

$AUX(v)$	Auxiliary tree associated with node v
$BB(p)$	Bounding box stored in the node associated with point p
C_i	A monotonic chain with index i
$Dis(p)$	Dimension discriminator associated p
$LEFT(v)$	Left pointer of node v
$NULL$	Empty pointer
$P(v)$	Point set associated with the descendants of node v
$Q_d(n)$	Total query time for a d-dimensional range tree with n points
$RIGHT(v)$	Right pointer of node v
$T_d(n)$	Total construction time on n points in d-dimensional space
$[a_1 : b_1] \times ... \times [a_d : b_d]$	An orthogonal range query where $a_1, b_1, ..., a_d, b_d \in \mathbb{R}$
$\cap$	Intersection
$\cup$	Union
$\forall$	For all
$\in$	Set membership
$\mathbb{R}^d$	Euclidean space in dimension d
$\mathbb{Z}^+$	natural numbers
$\mathcal{B}$	A bounding box
$\mathcal{I}$	The intersection of $\mathcal{P}$ and $\mathcal{R}$
$\mathcal{P}(C_i)$	Point set of chain C_i
$\mathcal{P}_{xy\_as}$	Point set associated with a set of ascending chains in xy-plane

$\mathcal{P}_{xy\_de}$	Point set associated with a set of descending chains in xy-plane
$\mathcal{P}$	A set of points
$\mathcal{R}$	A range query
$\mathcal{S}'$	Subset of $\mathcal{S}$
$\mathcal{S}_{xy\_as}$	An ascending chain set in xy-plane
$\mathcal{S}_{xy\_de}$	An descending chain set in xy-plane
$\mathcal{S}_{xz\_as}(C_i)$	An ascending sub-chain set derived from chain C_i in the xz-plane
$\mathcal{S}_{xz\_de}(C_i)$	A descending sub-chain set derived from chain C_i in the xz-plane
$\mathcal{S}$	Chain set $\{C_1, C_2, ..., C_m\}$
$\setminus$	Relative complement
imp	Implicit k-d tree
ptr	Pointer-based k-d tree
$\varnothing$	Empty set
$c_k(p_i)$	The coordinate of p_i in the kth dimension
dim	Parameter used for calculating the dimension discriminator
p_i	A point in $\mathcal{P}$
v	A tree node
$\|\mathcal{S}(C_i)\|$	Number of sub-chains set derived from chain C_i in the 1st-3rd coordinate plane
bi-bi	Range searching methods using two steps of binary searches
bi-seq	Range searching methods using best checking distance method after one binary search
cgal	Implementation of k-d tree in CGAL
imp-kd	Implicit k-d tree

lga	Longest chain decomposition algorithm
lg	Range searching method based on the longest chain decomposition algorithm
utga	Monotonic untangled chain decomposition algorithm
1D	One-dimensional
2D	Two-dimensional
3D	Three-dimensional
CGAL	Computational Geometry Algorithms Libary
MBRs	Minimum bounding rectangles
mod	Modulo operator

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisors Dr. Meng He and Dr. Norbert Zeh for the continuous support of my Master study and research. Besides the academic knowledge they offered me, I learnt a lot on how to become a qualified computer scientist, which involves patience, enthusiasm and motivation.

I also want to thank Dr. Dirk Arnold and Dr. Vlado Keselj for being my readers. Thanks for their detailed comments and suggestions to bring this research to this level.

Besides my advisors, I would like to thank my wife Simeng Cao. Without her support, I can not imagine how I could have got through these tough days. I also would like to thank my parents for their endless love and support.

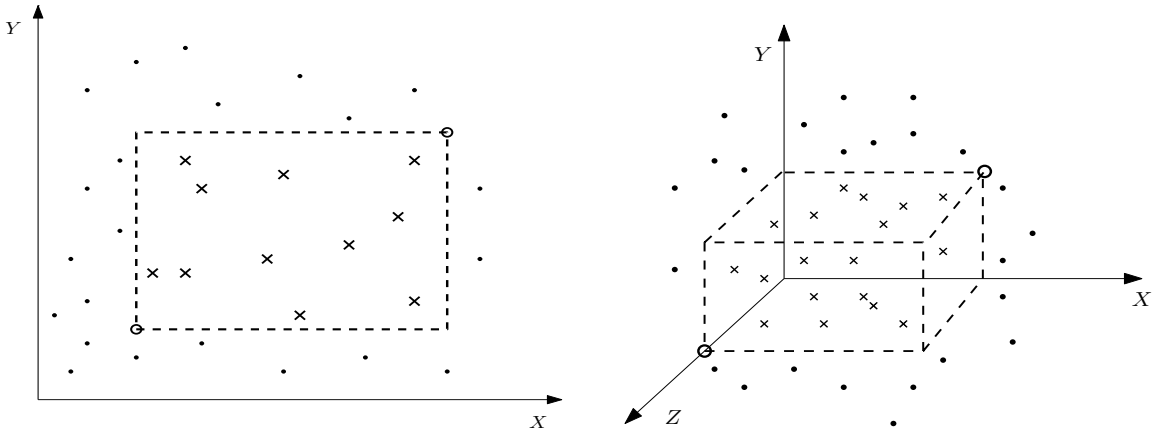
Chapter 1

Introduction

Range searching is one of the most fundamental computational geometry problems and has been studied extensively over the past decades [1, 2, 6, 7, 10, 12, 13, 17, 28]. The definition of range searching is described as follows. Let $\mathcal{P}$ be a set of n points in d-dimensional Euclidean space $\mathbb{R}^d$. Any point $p_i \in \mathcal{P}$ can be represented as a tuple $(c_1(p_i), c_2(p_i), ..., c_d(p_i))$, where $c_k(p_i)$ denotes the coordinate of p_i in the kth dimension. We wish to preprocess the set $\mathcal{P}$ so that, for any given range query $\mathcal{R}$ of a certain type, the set $\mathcal{I} = \mathcal{P} \cap \mathcal{R}$ can be found efficiently. In this thesis, we study orthogonal range searching, that is, each range query to be supported is a *axis-parallel box* $\mathcal{R} = [a_1 : b_1] \times ... \times [a_d : b_d]$, where $a_1, b_1, ..., a_d, b_d \in \mathbb{R}$ and $\mathcal{I} = \{p \in \mathcal{P} | a_k \leq c_k(p) \leq b_k, \forall 1 \leq k \leq d\}$. Figure 1.1 shows two simple orthogonal range search queries in $\mathbb{R}^2$ (Figure 1.1(a)) and $\mathbb{R}^3$ (Figure 1.1(b)). Other well studied types of range queries include simplex range searching [12, 16], half-space range searching [1, 28] and spherical space range searching [13].

In this thesis, we consider two types of orthogonal range searching: *range counting* and *range reporting*. Range counting asks to report the size of $\mathcal{I}$, whereas range reporting asks to output all the points in $\mathcal{I}$. From a complexity perspective, range counting and range reporting should be treated separately since the query time is a combination of both input size and output size. We could also ask if $\mathcal{I} \cap \mathcal{P} = \varnothing$ (*range emptiness*), or we can find the maximal point (*maximum query*) or minimal point (*minimum query*) within $\mathcal{R}$ according to some weights associated with the points. Our goal, in this thesis, is to design a simple and practical data structure that can answer range counting and range reporting queries efficiently for any orthogonal range query $\mathcal{R}$ in multidimensional space.

Many practical applications such as spatial databases [22, 35, 36], network information systems [2, 10, 30] and cloud computing [26, 27] all involve implementations of range searching. One simple example would be using Google Maps to find out



(a) 2-dimensional orthogonal range query (b) 3-dimensional orthogonal range query

Figure 1.1: Orthogonal range query examples. The dotted rectangle and cuboid represent orthogonal range queries in 2D and 3D respectively. The cross marks represent the points that lie in the query range.

all the restaurants in a certain neighbourhood. Another example may come from a university where the administrator may want to find all the students whose ages are between 20 to 23 and whose GPAs lie between 3.5 and 3.8. Many more complicated problems [11, 28] such as ray tracing and hidden-surface removal can also be reduced to range search problems.

1.1 Related Work

If we only need to answer one single range query, it can be easily done by checking whether each point in the point set $\mathcal{P}$ lies in the query range. This process takes linear time and is in fact the best one can do for range reporting or range counting if the query answer is to be exact. However in many applications, we would like to answer many queries of a given shape over a given point set, and support other operations like point insertion and deletion. If enough of these operations are to be carried out, then this justifies even a high preprocessing cost to build a data structure that can answer such queries (and support updates of $\mathcal{P}$) quickly. It would be advantageous to put some efforts into building such data structures to support fast range queries and updates. Next, a range of existing data structures for range searching problems in one and more dimensions will be reviewed.

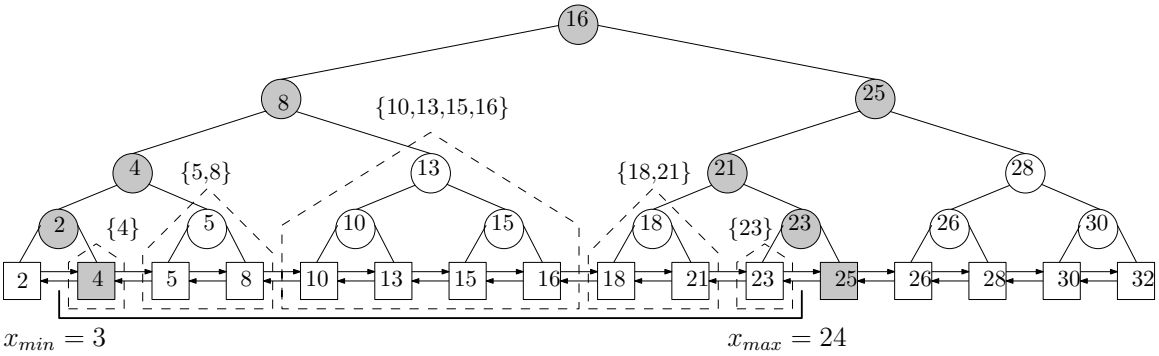


Figure 1.2: 1-dimensional balanced binary search tree for range searching. The point set $\mathcal{P} = \{2, 4, 5, 8, 10, 13, 15, 16, 18, 21, 23, 25, 26, 28, 30, 32\}$ and the range query $r = [3 : 24]$. The subtrees that are enclosed by dashed lines represent the subtrees whose leaf nodes need to be reported during the query but without any testing.

1.1.1 1-Dimensional Range Searching Data Structure

Balanced binary search tree. *Balanced binary search trees* can be used to answer 1-d range queries. The points in $\mathcal{P}$ are stored in the leaves of the tree sorted by their corresponding coordinates. All the internal nodes of the tree are branching nodes for search purpose. For each internal node v, we store two values: the number of leaves that are descended from v and the median value of the coordinates stored in v's subtree. Given a range query $\mathcal{R}$, we search down the tree for the minimal and maximal coordinates within $\mathcal{R}$, and then report all the leaves that lie between the two. Because we store in each internal node the number of leaves descended from it, the range counting cost is the same as the cost of searching for the minimal and maximal coordinates, which is bounded by $O(\lg n)$ [1]. The cost of reporting all leaves in between is $O(k)$, where k is the number of reported leaves. The range reporting time for a balanced binary search tree is thus $O(\lg n + k)$. We can also link all the leaves as a doubly linked list. When the first position in the range has been located through a single tree traversal, the rest of the records can be easily accessed by sequentially scanning the remaining records along the list, and thus the range reporting performance can be improved. Balanced binary search trees are used extensively in database systems that require insertion, deletion, and range searches. Figure 1.2 shows an example for 1D range searching using a balanced binary search tree.

[1] $\lg n$ represents the binary logarithm $\log_2 n$

1.1.2 Multi-Dimensional Orthogonal Range Searching Data Structures

K-d tree. A *k-d tree* [6, 8] (*k-dimensional tree*) is a data structure used for organizing a point set in k-dimensional space. The k-d tree is a space-partition binary tree. Each level of the tree is associated with one of the k dimensions and partitions the entire space into sub-spaces. Each tree node at each level has an associated hyperplane that splits the point set associated with this node into two subsets of the same size. These two subsets are associated with the node's children. The partition steps cycle through the different axes in a predefined order until each sub-space contains one single point. In practice, the k-d tree performs extremely well when the number of dimensions is relatively small [17]. The k-d tree uses linear space and achieves a query bound of $O(d \cdot n^{1-1/d})$ for range counting and a query bound of $O(d \cdot n^{1-1/d} + k)$ for range reporting, where k is the number of points to be reported.

Range tree. Similarly to the k-d tree, the multidimensional range tree [8] was designed for fast range searching queries in multidimensional space. However, it offers a different trade-off between its size and the cost of queries. It achieves a query bound of $O(\lg^d n)$ for range counting, at the expense of using $O(n \lg^{d-1} n)$ space. The key to multidimensional range tree is reducing a range query in d dimensions to $O(\lg n)$ queries in $d-1$ dimensions. We will discuss more about range tree in Section 2.2.

R-tree. An *R-tree* [20] is a tree data structure used for spatial searching. An R-tree and a k-d tree are based on similar ideas (space partitioning based on axis-aligned regions), but nodes in k-d trees are associated with splitting hyperplanes that partition the entire space into regions whereas nodes in R-trees are associated with the *minimum bounding rectangles* (MBRs) of the data records that only partition the subset of space containing the points of interest. The data records stored in R-trees can be points, rectangles and polygons. Like the B-tree, the R-tree is a height balanced tree that stores the data items at its leaf nodes. The query efficiency of an R-tree depends mainly on two factors: the amount of overlap between the bounding boxes of sibling nodes and the ratio of the number of points to the volume of the bounding box of each node. A query needs to inspect every node whose bounding box it intersects, because this subtree may contain objects that lie in the query range. If the bounding boxes stored in a set of sibling nodes overlap with each other heavily, then, many of these sibling nodes have to be examined after we visit the

parent of these nodes, which will decrease the range searching performance. It is also undesirable to have many nodes with bounding boxes that only have several number of points in it. When the ratio of the number of points to the volume of the bounding box within each node is too small, a query may intersect the bounding box on its empty area during the search and the tree traversal can not stop immediately, which will bring in lots of unnecessary node accesses. Many variants of R-trees including the *R+-tree* [32], *R*-tree* [5], and *Priority R-tree* [3] were designed to improve the performance of R-trees. Even though the R-tree does not guarantee a good worst-case performance (the worst case range searching time of an R-tree is $O(n)$), it performs quite well in real-world implementations for a moderate number of dimensions [21].

1.2 Our Work

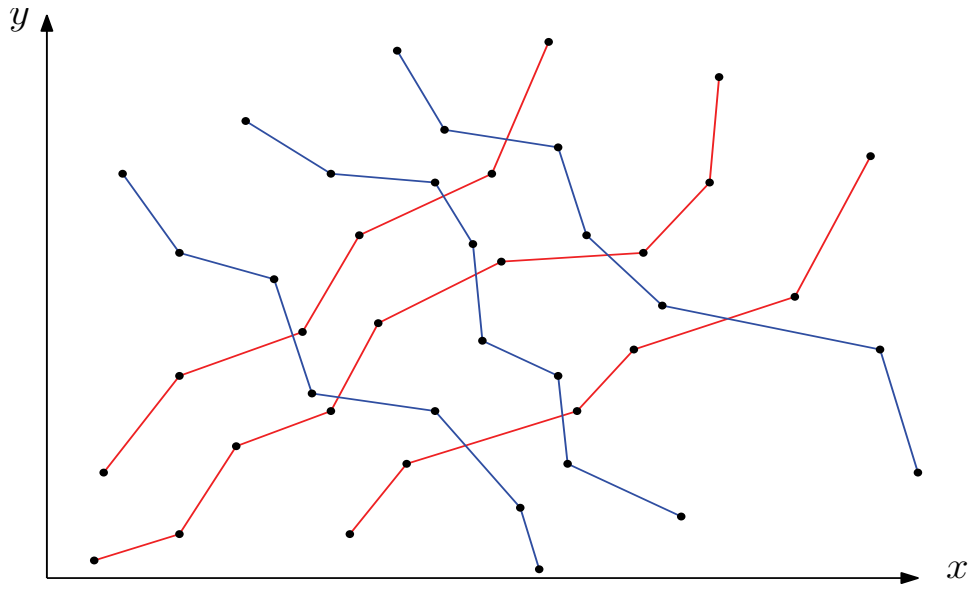


Figure 1.3: An example of reorganizing the point set into a set of monotonic chains. The red chains are monotonic ascending chains in 2D and the blue chains are monotonic descending chains in 2D.

In 2009, Arroyuelo et al. [4] proposed a static data structure for orthogonal range reporting in 2-dimensional space. The key to the algorithm is to reorganize the point set into a set of monotonic chains (as the 1st coordinates (x-coordinates) of the points increase in a chain, the 2nd coordinates (y-coordinates) also increase or

decrease monotonically) (see Figure 1.3). Further improvements of the query performance are achieved by untangling the chains, that is, by ensuring that no two chains cross. Each range query can be answered in $O(m \lg n + k)$ time, where m is the number of monotonic chains into which the point set can be partitioned and k is the output size. The experiments [14] demonstrated the efficiency of the data structure. In some practical data sets, its performance significantly exceeded that of existing implementations of the k-d tree and range tree. Another advantage of the data structure of Arroyuelo et al. is its space efficiency. Many range searching data structures (described in the previous section) are hierarchical tree structures, which means that in addition to the original point set, each node of the tree has to keep two or more pointers to its children. In contrast, the data structure of Arroyuelo et al. only stores the reorganized point set and the length of each chain. Because their data structure performs extremely well in terms of space efficiency and range query efficiency in 2D, it is natural to ask if we can extend it to 3-dimensional space.

In this thesis, we proposed and implemented the first data structure based on untangled monotonic chains for orthogonal range searching in 3-dimensional space, aiming at supporting efficient orthogonal range queries. We also compared our data structure with some existing 3D range query data structures. During the implementation, we found that the existing k-d tree implementation in the CGAL [34] (also used for comparison in [14]) is much slower than necessary for answering orthogonal range queries over a static point set. As a result, we implemented k-d tree ourselves as a reference for performance comparison. Moreover, we implemented another 3-dimensional chain decomposition algorithm proposed by Wei [37] and compared it with ours. The experiments revealed that none of the range searching methods based on these two chain decomposition methods is comparable with the range query performance of the k-d tree for most of the cases. This is because the range query method based on the chain decomposition algorithm proposed by us suffered from its massive number of generated chains and the range query method based on the chain decomposition algorithm by Wei [37] is impractical when the point set size gets larger.

1.3 Overview

The rest of thesis is organized as follows. In Chapter 2, we discuss in detail two data structures that we implemented for the range query problems and then describe some previous work on the 2-dimensional chain decomposition methods and their corresponding range searching algorithms. Chapter 3 first introduces an alternative chain untangling algorithm, and then provides the entire 3D chain decomposition and range query algorithms. Chapter 4 presents the experimental results and discussions. Finally, in Chapter 5, we state our conclusions and discuss some interesting open problems.

Chapter 2

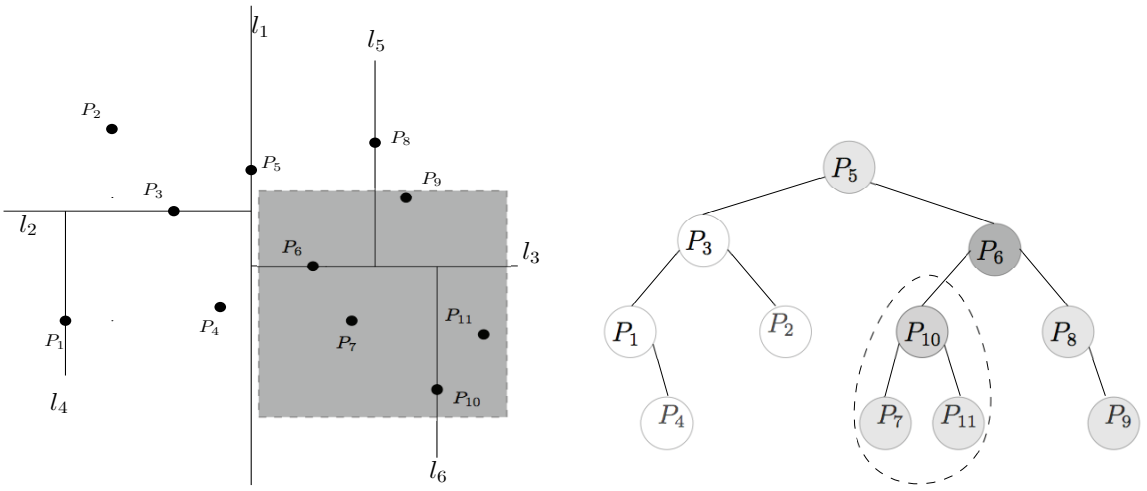
Background

The k-d tree and the multidimensional range tree are described in Section 2.1 and Section 2.2 respectively. Then, in Section 2.3, the 2-dimensional chain decomposition algorithm and its corresponding range query method [4] are introduced. Finally, we discuss the challenges on range searching in 3-dimensions.

2.1 K-d Tree

A k-d tree can be viewed as an extension of a binary search tree to k-dimensional space. If a given n point set $\mathcal{P}$ is represented as a k-d tree, each point in $\mathcal{P}$ is stored as a node in the tree. We assume no two points in $\mathcal{P}$ have the same coordinates in all k dimensions. Since there is a one-to-one mapping between the points and tree nodes, for convenience, we can refer to each node in the tree by the point it stores. Every node $p_i \in \mathcal{P}$ stores two pointers $LEFT(p_i)$ and $RIGHT(p_i)$ to p_i's left and right children. In addition to these two pointers, each node has an associated dimension discriminator $Dis(p_i)$, which is an integer between 1 and k. Let $c_j(p_i)$ denote the coordinate of p_i in the jth dimension and let $m = Dis(p_i)$. Then for any node p_l in $LEFT(p_i)$ (noting that we can use the pointer to represent a subtree), $c_m(p_l) < c_m(p_i)$; and for any node p_r in $RIGHT(p_i)$, $c_m(p_r) > c_m(p_i)$. When multiple points with their coordinates at dimension m are all equal to $c_m(p_i)$, we define a superkey of p_i at dimension m as $s_m(p_i) = c_m(p_i)c_{m+1}(p_i)\ ...\ c_k(p_i)c_1(p_i)\ ...\ c_{m-1}(p_i)$. For any node p_l in $LEFT(p_i)$, we then have $s_m(p_l) < s_m(p_i)$; for any node p_r in $RIGHT(p_i)$, we have $s_m(p_r) > s_m(p_i)$. In this way, the resulting k-d tree is a balanced tree and the height of the tree is at most $\lfloor \lg n \rfloor$.

For any given level of the tree, all nodes have the same dimension discriminator. The root has dimension discriminator 1 and its two children have dimension discriminator 2. In d-dimensional space, any node p_i at distance l from the root has dimension discriminator $Dis(p_i) = 1 + (l \bmod d)$. Thus, the dimension discriminator of a node



(a) 2D plane subdivision (b) Corresponding k-d Tree

Figure 2.1: 2-dimensional pointer-based k-d tree. The shaded rectangle in (a) indicates a range query. The shaded nodes in (b) represent the nodes visited during the tree traversal for the given range query in (a). The subtree that is enclosed by dashed curves represents the subtree whose boundary box lies entirely in the query range.

does not need to be stored explicitly. Figure 2.1 gives an example of a 2-dimensional k-d tree and its corresponding planar subdivison. Algorithm 1 shows how to construct a k-d tree from a point set and the parameter *dim* is used for calculating the dimension discriminators. In Algorithm 1, we first create a new node v for the current point set $\mathcal{P}$ in line 1. If $\mathcal{P}$ contain only one point, we then assign it to v and return (lines 2 to 3). Otherwise, we find the median point p_m of $\mathcal{P}$ based on the current dimension discriminator, assign it v and split $\mathcal{P}$ into two subsets (lines 5 to 11). After that, we recurse on both subsets while increasing the dimension discriminator by one (lines 12 to 13).

Next, let us consider the construction time. The most costly part during each recursion is to find the median point of the current point set (see line 8 in Algorithm 1). This median searching step can be done in $O(n)$ time using linear-time selection [9]. An alternative is to maintain d presorted lists. Each list is presorted by each dimension from 1 to d. Given d sorted lists, at each recursion, the current sorted point set can be obtained in linear time based on the current depth and the median point can also be found in constant time (this method works well when the dimension d is relatively

Algorithm 1 K-dTreeConstruction($\mathcal{P}, dim$)

Input: A set of d-dimensional points $\mathcal{P} = \{p_1, p_2, ..., p_n\}$ and dim with initial value 1.

Output: A d-dimensional pointer-based k-d tree.

```
1: let v be a new node with LEFT(v) = NULL and RIGHT(v) = NULL
2: if P has only one point p then
3:    v.point = p
4: else
5:    num ← number of points in P
6:    m = ⌈num/2⌉
7:    j = 1 + (dim mod d)
8:    s_j(p_m) ← the mth smallest coordinate in {p ∈ P|s_j(p)}
9:    v.point = p_m
10:   P_1 = {p' ∈ P \ {p_m}| m − 1 points with s_j(p') < s_j(p_m)}
11:   P_r = {p' ∈ P \ {p_m}| l − m points with s_j(p') > s_j(p_m)}
12:   LEFT(v) ← K-dTreeConstruction(P_l, dim + 1)
13:   RIGHT(v) ← K-dTreeConstruction(P_r, dim + 1)
14: end if
15: return v
```

small). This gives the following recurrence for the construction time:

$$T(n) = O(n) + 2T(\lceil n/2 \rceil) \tag{2.1}$$

This recursion solves to $T(n) = O(n \lg n)$. By adding the running time for presorting, the total running time for the construction is bounded by $O(n \lg n)$. Since each node in the k-d tree is associated with a point in $\mathcal{P}$, the total storage size is bounded by $O(n)$.

The k-d tree can also be represented implicitly. The construction steps are similar to the pointer-based k-d tree mentioned above: we first store all the points in an array. During the tree construction, at each recursion step, we find the median point p_m and place it at the middle position of current array. Let $j = Dis(p_m)$, we then move all the points p with $s_j(p) < s_j(p_m)$ to the left part of the array and move all the points p with $s_j(p) > s_j(p_m)$ to the right part of the array. After that, we continue recursing on these two subarrays. Given that all splits in the tree are perfectly balanced, the children of each node can be identified using index arithmetic.

A range query on a k-d tree can be answered by performing a tree traversal, which is similar to the 1D range search. Given a range query $\mathcal{R} = [a_1 : b_1] \times ... \times [a_d : b_d]$ and a

Algorithm 2 Implicit_k-dTreeRangeQuery($P, start, end, \mathcal{R}, dim, \mathcal{B}$)

Input: An implicit k-d tree array $P[1...n]$, a range query $\mathcal{R} = [a_1 : b_1] \times ... \times [a_d : b_d]$, a bounding box $\mathcal{B} = [A_1, B_1] \times ... \times [A_d, B_d]$ where $A_i = min\{c_i(p)|p \in \mathcal{P}\}$ and $B_i = max\{c_i(p)|p \in \mathcal{P}\}$, and *start*, *end* and *dim* with initial values 1, n and 1, respectively.

Output: All the points in P_i that lie in $\mathcal{R}$.

1: **if** $start > end$ **then**
2: **return**
3: **end if**
4: $index = \lfloor (end + start)/2 \rfloor$
5: **if** $CONTAINED(\mathcal{R}, \mathcal{B}) = true$ **then**
6: **return** report all the points in $P[start...end]$
7: **end if**
8: **if** $P[index]$ lies in $\mathcal{R}$ **then**
9: report $P[index]$
10: **end if**
11: $\mathcal{B}_l = [A_1, B_1] \times ...[A_{dim}, c_{dim}(P[index])]... \times [A_d, B_d]$
12: $\mathcal{B}_r = [A_1, B_1] \times ...[c_{dim}(P[index]), B_{dim}]... \times [A_d, B_d]$
13: $dim' = (dim \bmod d) + 1$
14: **if** $a_{dim} \leq c_{dim}(P[index])$ **then**
15: K-dTreeRangeQuery($P, start, index - 1, \mathcal{R}, dim', \mathcal{B}_l$)
16: **end if**
17: **if** $b_{dim} \geq c_{dim}(P[index])$ **then**
18: K-dTreeRangeQuery($P, index + 1, end, \mathcal{R}, dim', \mathcal{B}_r$)
19: **end if**

node p where $Dis(p) = m$, all points p_l in $LEFT(p)$ satisfy $c_m(p_l) \leq c_m(p)$; similarly, for any point p_r in $RIGHT(p)$, we have $c_m(r) \geq c_m(p)$. When accessing a node p during the tree traversal, we first compare $c_m(p)$ with a_m and b_m, if $a_m \leq c_m(p) \leq b_m$, then both $LEFT(p)$ and $RIGHT(p)$ may contain points in the query range, so we recurse on both children. Also, p may be in the query range, so we check the remaining $(d-1)$ coordinates and report p if it is in the range; if $a_m > c_m(p)$, then only points in $RIGHT(p)$ can be in the query range, so we recurse on this child; if $b_m < c_m(p)$, it suffices to recurse on $LEFT(p)$; Initially, the node p is set to be the root of the tree with $Dis(p) = 1$. Figure 2.1(b) shows a range query example using a 2-dimensional k-d tree. During this range query, Point P_5, P_6, P_7, P_9, P_{10} and P_{11} are reported and point P_8 is being tested but not reported.

We can improve the range query efficiency of the k-d tree further with an additional data structure. For any node p in the tree, the points in each subtree with root p

are contained in *a bounding box*. Initially for the tree root p_t, we define $BB(p_t) = [A_1, B_1] \times ... \times [A_d, B_d]$ as the bounding box for p_t, where $A_i = min\{c_i(p)|p \in \mathcal{P}\}$ and $B_i = max\{c_i(p)|p \in \mathcal{P}\}$. For any node p, the bounding boxes of $LEFT(p)$ and $RIGHT(p)$ are obtained by splitting $BB(p)$ at coordinate $c_m(p)$ in dimension $m = Dim(p)$. The bounding boxes can be computed on-line during the range query, so we do not need to store them explicitly. For instance in Figure 2.1, $BB(P_{10}) = [c_1(P_5), c_1(P_{11})] \times [c_2(P_{10}), c_2(P_6)]$. This means the points stored in the subtree rooted at P_{10} are bounded by the x-coordinates of P_5 and P_{11} and by the y-coordinates of P_{10} and P_6. Given that definition, we can use the bounding box computed at each node to determine whether all the points stored in the subtree rooted at this node are entirely lying in the query range. As we can observe in Figure 2.1, the node P_{10} corresponds to a region that is fully contained in the query range, so the points P_7, P_{10} and P_{11} are reported immediately without any further tests. Algorithm 2 shows the pseudocode to answer a range query using an implicit k-d tree. The tree is defined by a node array $P[1..n]$. We use *start* and *end* (initially $start = 1$ and $end = n$) to keep track of the subarray storing the points in the current subtree. When the bounding box of current node is fully contained in range query $\mathcal{R}$, then all points stored in the subtree of this node need to be reported and the recursion stops (lines 5 to 7). For lines 11 and 12 in Algorithm 2, the current boundary box is split in two based on the point stored in the current node. For lines 13 to 19, we recurse on the children of current node based on the comparison of the coordinate of the point in current node with the query range $\mathcal{R}$.

D.T. Lee and C.K. Wong [25] showed that the cost of a single range query over a d-dimension k-d tree with n points is $O(d \cdot n^{1-1/d} + k)$ where k is the number of reported points. The following theorem summarizes the properties of the k-d tree:

Theorem 1 *A d-dimensional k-d tree can be constructed in $O(n \lg n)$ time using linear space. An orthogonal range query can be answer in $O(d \cdot n^{1-1/d} + k)$ time, where k is the number of points in the range.*

2.2 Range Tree

The query time of a balanced k-d tree in d-dimensional space is $O(n^{1-1/d}+k)$, where n is the total number of points and k is the number of reported points. Although $O(n^{1-1/d}+k)$ is faster than a linear scan, it is possible to do better. In this section, we will talk about the range tree, a multi-level tree that answers range queries asymptotically faster than the k-d tree, but at the cost of higher space requirements.

Before we describe the higher-dimensional range tree, let us first examine the 1-dimensional range tree. A 1-dimensional range tree is a balanced binary search tree. Each node v in the tree stores a point in $\mathcal{P}$ and two pointers $LEFT(v)$ and $RIGHT(v)$ that point v's left and right children. We define $P(v) \subseteq \mathcal{P}$ to be the *canonical subset* of v, which is associated with the points stored in the descendants of v. Given a range query $\mathcal{R} = [a_1 : b_1]$, we first identify the first node whose coordinate lies in $[a_1 : b_1]$, which we call the split node. We then continue the search using a_1 at the left child of the split node. For every node v, we check whether a_1 is no greater than the point p stored at v. If so, we output p and all the points in $P(RIGHT(v))$ and continue the search at v's left child. Otherwise, the search continues at v's right child. Similarly, for the right child of the split node, we continue the search with b_1. At each node v, we check whether b_1 is no less than the point p stored at v. If so, we output p and all the points in $P(LEFT(v))$ and continue the search at v's right child. Otherwise, the search continues at v's left child. Figure 2.2 shows an example of a 1D range tree. As we can see from the figure, the points whose coordinates lie in $\mathcal{R}$ can be expressed as a disjoint union of $O(\lg n)$ subtrees (shown as the shaded areas in Figure 2.2) whose canonical subsets of the roots (like $P(p_6)$ and $P(p_{10})$ in Figure 2.2) lie in $\mathcal{R}$. The range query time of a 1D range tree with n points is thus $O(\lg n + k)$, where k is the output size.

Given a point set $\mathcal{P}$ of size n in d-dimensional space, a d-dimensional range tree can be constructed recursively. The construction starts with a primary tree (first level tree), which is a balanced binary search tree built based on the 1st coordinates of the points in $\mathcal{P}$. Each internal node v in the primary tree has an associated secondary tree $AUX(v)$, which is a $(d-1)$-dimensional range tree over the last $d-1$ dimensions of the points in $P(v)$ ordered by their second coordinates. We apply this tree construction step recursively until we are left with points restricted to their dth

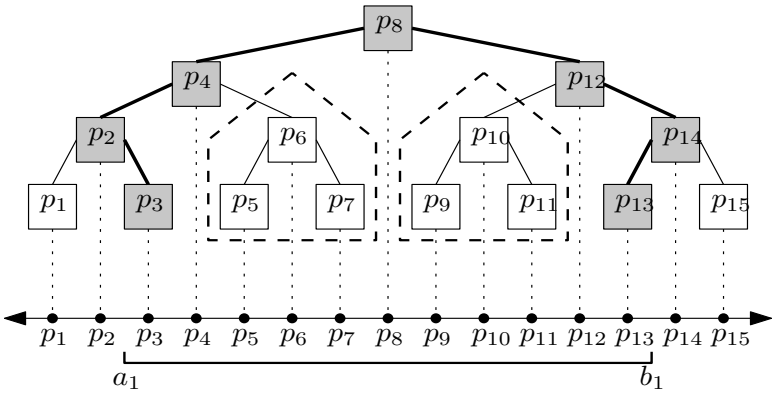


Figure 2.2: A 1D range tree. A 1D point set $\mathcal{P}$ where $n = 16$ and a range query $\mathcal{R} = [a_1 : b_1]$. The subtrees that are enclosed by dashed lines represent the subtrees whose canonical subsets of the roots need to be reported during the range query.

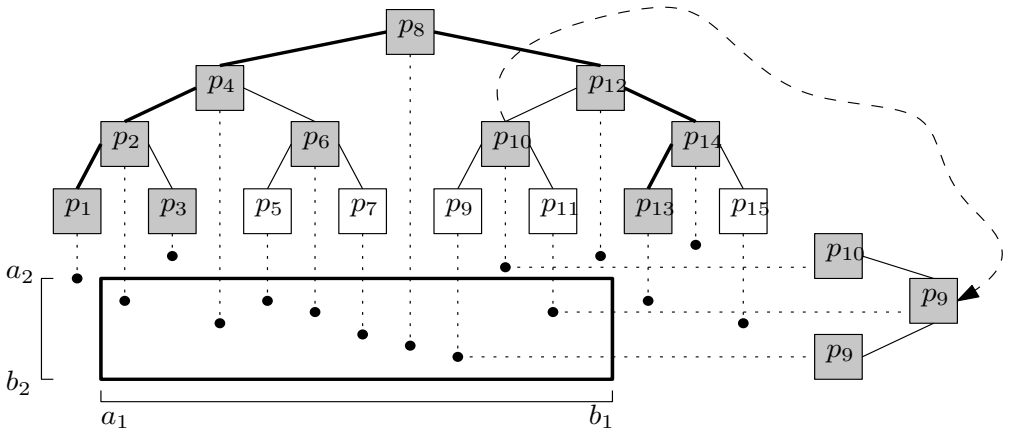


Figure 2.3: A 2D range tree. A 2D point set $\mathcal{P}$ with 16 points and a range query $\mathcal{R} = [a_1 : b_1] \times [a_2 : b_2]$. The node p_{10} at the first level tree is associated with another second level tree using a dashed arrow. The rectangle inclose all points lie in the query range.

coordinates. Figure 2.2 gives an example of a 2-dimensional range tree.

Algorithm 3 gives the pseudocode for the construction of a d-dimensional range tree. We adopt the superkey definition from Section 2.1. In each recursive call, we first create a new node v (line 4). If the current point set contains only one point, we assign it to v (lines 4 to 6). Otherwise, we find the median point p_m based of current dimension, assign it to v and split the current point set into two subsets (lines 7 to 13). We then recurse on both subsets while creating a secondary tree with one dimension higher (lines 14 to 16).

Similarly to the k-d tree, the most costly step in each invocation is to find the

Algorithm 3 RangeTreeConstruction($\mathcal{P}, dim$)

Input: A d-dimensional point set $\mathcal{P}$ and dim with initial value 1.
Output: A d-dimensional range tree.

```
1: if dim > d then
2:    return
3: end if
4: let v be a new node with LEFT(v) = NULL, RIGHT(v) = NULL and
   AUX(v) = NULL
5: if P has only one point p then
6:    v.point = p
7: else
8:    num ← number of points in P
9:    m = ⌈num/2⌉
10:   s_dim(p_m) ← the mth smallest coordinate in {s_dim(p)|p ∈ P}
11:   v.point = p_m
12:   P_1 = {p' ∈ P \ {p_m}| m − 1 points with s_dim(p') < s_dim(p_m)}
13:   P_r = {p' ∈ P \ {p_m}| l − m points with s_dim(p') > s_dim(p_m)}
14:   LEFT(v) = RangeTreeConstruction(P_l, dim)
15:   RIGHT(v) = RangeTreeConstruction(P_r, dim)
16:   AUX(v) = RangeTreeConstruction(P, dim + 1)
17: end if
18: return v
```

median point of the current point set. We can use linear-time selection [9] or maintain d presorted lists to achieve a linear search time for this step. Let $T_d(n)$ be the total running time for the range tree construction on n points in d-dimensional space. We establish the recurrence as follows:

$$T_d(n) = 2T_d(\lceil n/2 \rceil) + T_{d-1}(n) + O(n) \tag{2.2}$$

When using the presorted lists, the base case $T_1(n) = O(n)$. The recurrence solves to $T_d(n) = O(n \lg^{d-1} n)$. By adding the extra time for the pre-sorting steps, the total construction time is bounded by $O(n \lg^{d-1} n)$. The size of storage is the same as the construction time, which can be calculated using the same analysis method.

Next, we consider the range query. Given a d-dimensional range tree and a range query $\mathcal{R} = [a_1 : b_1] \times ... \times [a_d : b_d]$, we apply a 1-dimensional range search on the primary tree using the query interval $[a_1 : b_1]$ to locate $O(\lg n)$ nodes whose canonical subsets ($P(v)$) together contain all the points with their first coordinates

Algorithm 4 RangeSearching($v, dim, \mathcal{R}$)

Input: A d-dimensional range tree T, a range query $\mathcal{R} = [a_1 : b_1] \times ... \times [a_d : b_d]$ and dim with initial value 1.
Output: All the points in $\mathcal{R}$.

```
1: if dim > d then
2:    return
3: end if
4: if v is a leaf node then
5:    if CONTAINED(v, R) then
6:       return point associated with v
7:    end if
8:    return
9: end if
10: if c_dim(v) < a_dim then
11:    RangeSearching(RIGHT(v), dim, R)
12: else if c_dim(v) > b_dim then
13:    RangeSearching(LEFT(v), dim, R)
14: else
15:    if CONTAINED(v, R) then
16:       return point associated with v
17:    end if
18:    if c_dim(v) ≥ a_dim then
19:       RangeSearching(RIGHT(v), dim + 1, R)
20:       RangeSearching(LEFT(v), dim, R)
21:    end if
22:    if c_dim(v) ≤ b_dim then
23:       RangeSearching(LEFT(v), dim + 1, R)
24:       RangeSearching(RIGHT(v), dim, R)
25:    end if
26: end if
```

lie in $[a_1 : b_1]$. For each node v reported during the search of primary tree, we have $a_1 \leq c_1(p) \leq b_1$, $\forall p \in P(v)$). Thus, to decide which of these point in $P(v)$ with their second coordinates lie in $[a_2 : b_2]$, it suffices to perform another 1-dimensonal range search at the v's second level range tree ($AUX(v)$) using the range query interval $[a_2 : b_2]$. In $AUX(v)$, we select $O(\lg n)$ nodes where all points descended from them are with their second coordinates lie in $[a_2 : b_2]$. That means there are $O(\lg^2 n)$ nodes chosen in the second level trees in total. Together, all their descendants contain all the points whose first and second coordinates lie in $[a_1 : b_1] \times [a_2 : b_2]$. We recursive apply this procedure until we reach the d-th level trees. In the d-th level trees, we find

all points whose last coordinates lie in $[a_d : b_d]$ and report them. Algorithm 4 gives the pseudocode for the d-dimensional range searching. The function $CONTAIN(v, \mathcal{R})$ is used to determine whether the point stored at node v lies in $\mathcal{R}$. Lines 18 to 25 show the recursive calls on the secondary trees given $a_{dim} \leq c_{dim}(v) \leq b_{dim}$ for current node v.

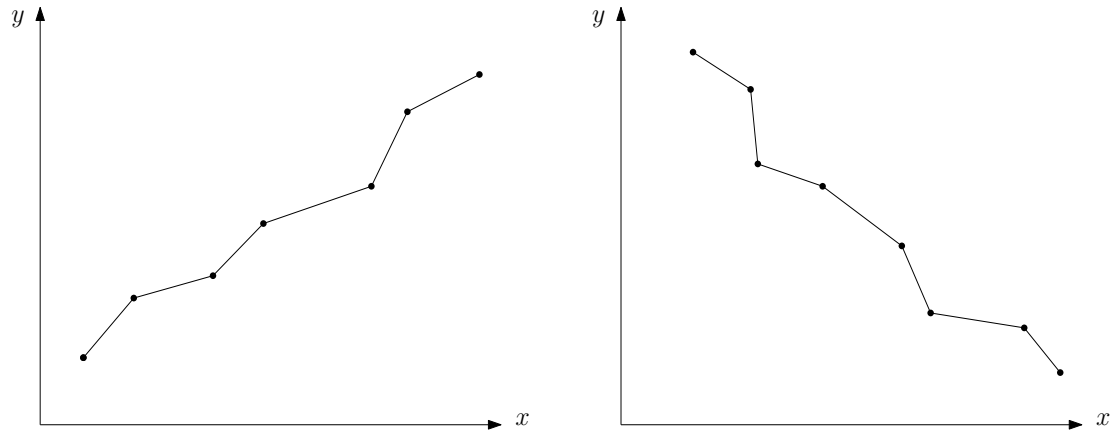
Let $Q_d(n)$ denotes the query time for a d-dimensional range tree with n points. Then, for $d = 1$, we have $Q_1(n) = O(\lg n)$. This gives the following recurrence for the range query time:

$$Q_d(n) = O(\lg n) + O(\lg n) \cdot Q_{d-1}(n) \tag{2.3}$$

This recursive function has the solution $Q_d(n) = O(\lg^d n)$. Based on the above analysis, we have the following theorem.

Theorem 2 *Given a set of point P in d-dimensional space, a d-dimensional range tree can be constructed in $O(n \lg^{d-1} n)$ time using $O(n \lg^{d-1} n)$ space. Each range query $\mathcal{R}$ can be answered in $O(\lg^d n + k)$ time, where k is the number of reported points.*

2.3 2-Dimensional Chain Decomposition and Range Searching



(a) 2D ascending chain (b) 2D descending chain

Figure 2.4: Monotonic chains in 2D.

Arroyuelo et al. [4] presented the first static adaptive data structure based on untangled monotonic chains for orthogonal range searching in the plane. The data structure is adaptive, which means when the point set is decomposed into a relatively small number of chains, the range query time can be improved. If the number of chains is $o(\sqrt{n})$, the data structure outperforms the optimal-time linear space data structures [3, 24, 6]. The basic idea of Arroyuelo et al. algorithm is that given a point set $\mathcal{P}$ in 2D, we partition the points in $\mathcal{P}$ into a number of untangled monotonic ascending or descending chains. We define an ascending or a descending chain in 2D as follows: if we sort the points in the chain by increasing x-coordinates, and by doing so their y-coordinates are sorted in ascending or descending order, we then call it an ascending chain or a descending chain (see Figure 2.4). All the chains are further divided into two sets: an ascending chain set and a descending chain set. All chains in each set are untangled, which means no two chains intersect. Because chains in each set are monotonic and untangled, each range query can be answered through two steps of "searches": first, we apply one search among the chains to find all candidate chains that cross the query rectangle; and we apply another search on each candidate chain to identify the points within the range. The data structure can be constructed in $O(n^3)$ time and uses linear space. A range query can be answered in $O(\lg m \lg n + m' \lg n + k)$ time, where m is the total number of the chains, m' is the number of chains that cross the query box and k is output size [4]. Claude et al. [14] implemented the data structure and compared it with the 2D range tree and the 2D k-d tree. Their experimental results showed that the data structure is both practical and efficient when dealing with a static data set. We will briefly introduce Arroyuelo et al.'s algorithm in the following sections since that is the basis of our 3D range search algorithm.

2.3.1 Splitting the Points into Two Subsets

The first step of Arroyuelo et al.'s algorithm is to partition the point set $\mathcal{P}$ into two subsets: one consists of a set of ascending chains and the other one consists of a set of descending chains. Unfortunately, finding the minimal number of monotonic chains in 2-dimensional space is NP-hard [18], but there are algorithms [19, 38] that can bound the maximum number of chains by $O(\sqrt{n})$. According to the experimental

results of Claude et al. [14], the greedy algorithm of Fomin et al. [39] obtained the fewest number of chains among the methods they compared. The idea of Fomin's algorithm is quite straightforward. We first initiate two empty sets: one is for the ascending chains; and the other is for the descending chains. In each iteration, we use all the points in the current set to find both the longest ascending and descending chain. The longer of the two is chosen and assigned to its corresponding set; in the meantime, we remove its points from the current point set. We repeat this process until no points are left. Calculating the longest ascending and descending monotonic chains in each pass takes $O(n \lg n)$ time. Since the total number of chains is no greater than $O(\sqrt{n})$ [37], the overall running time of Fomin's greedy method is bounded by $O(n^{\frac{3}{2}} \lg n)$.

2.3.2 Supowit's Algorithm

Algorithm 5 SupowitDescending($\mathcal{P}$)

Input: A set of 2-dimensional points $\mathcal{P} = \{p_1, p_2, ..., p_n\}$, where $|\mathcal{P}| = n$.
Output: A partition $\mathcal{S} = \{C_1, C_2, ..., C_m\}$ of $\mathcal{P}$ into a minimal number of chains, for any $p_r, p_q \in C_i$ where $\forall r < q \leq n$, we have $c_1(p_r) < c_1(p_q)$ and $c_2(p_r) > c_2(p_q)$.

```
1: S ← ∅
2: for i = 1...n do
3:    S' = {C|C ∈ S and min_2(C) ≥ c_2(p_i)}
4:    if S' ≠ ∅ then
5:       find the chain C_min that has the point with the minimum 2nd coordinate
         among all the chains in S'
6:       append p_i to C_min
7:    else
8:       generate a new chain C_p with only one point p_i to S
9:    end if
10: end for
11: return S
```

After the point splitting step, we obtain two subsets of points for both directions: ascending and descending. When the direction is fixed, Supowit gives an algorithm to find the minimal number of chains in optimal $\Theta(n \lg n)$ time [33]. Since the total number of chains is bounded by $O(\sqrt{n})$ from the first step and Supowit's algorithm offers the optimal chain partition, the total number of chains after Supowit's algorithm is still bounded by $O(\sqrt{n})$.

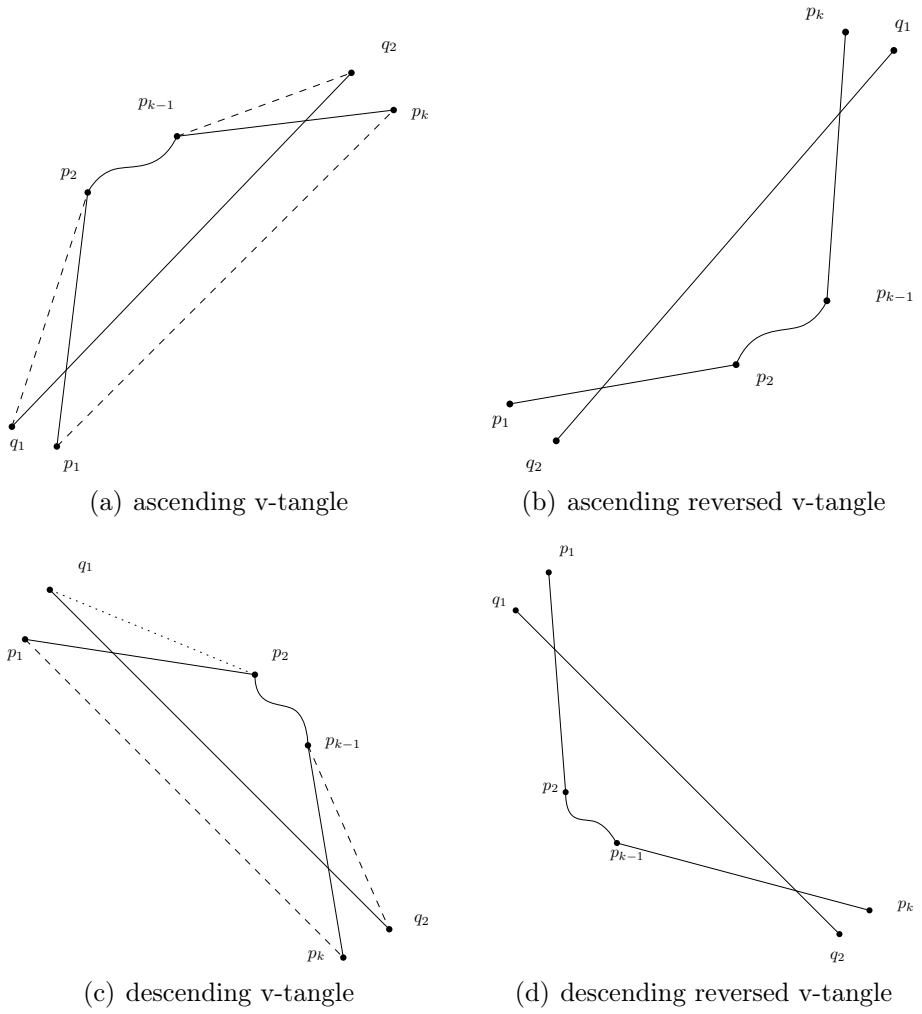
The reason we apply Supowit's algorithm as the second step is because during the chain construction step, there could be intersections generated between chains. As discussed earlier, we want to get an untangled chain set without intersections to achieve a faster range query. Even though Supowit's algorithm may also produce intersecting chains, those intersections are of a special form, which allows us to untangle the chains efficiently.

Given a point set $\mathcal{P} = \{p_1, ..., p_n\}$ sorted based by increasing x-coordinates, the algorithm processes the points in sorted order. Each point is either added to an existing chain or assigned to a new chain based on whether it can be added to an existing while maintaining the monotonicity. Algorithm 5 shows the case for the descending set. The strategy is symmetric for the ascending set. We use C_i to represent the chain with index i and let $min_2(C_i) = min\{c_2(p)|p \in C_i\}$ where $c_2(p)$ denotes the y-coordinate of point p. If we use a balanced binary search tree to store the minimum y-coordinate of each chain, the query time of step 5 in Algorithm 5 is bounded by $O(\lg n)$. Thus, the total running time of Supowit's algorithm is $O(n \lg n)$.

2.3.3 Untangling the Chains

If we want an adaptive and efficient range query algorithm based on the chain decomposition method, the chains should be untangled, that means, no two chains cross. Arroyuelo et al. [4] introduced an algorithm to untangle the chains produced by Supowit's method. The main idea behind the untangling algorithm is that each of the tangles (intersections) created by Supowit's algorithm for both ascending and descending chain sets forms a *v-tangle*. Figure 6(a) and 6(c) show v-tangles of ascending and descending chains, respectively.

Definition 1 [**Ascending tangles**] *Given an edge* (p_i, p_j) *in an ascending chain, let* $H^+(p_i, p_j)$ *be the open half-plane containing the point* $(c_1(p_j) + 1, c_2(p_j) - 1)$ *and* $H^-(p_i, p_j)$ *be the open half-plane containing the point* $(c_1(p_j)-1, c_2(p_j)+1)$*. Assuming that we have two chains* C_1 *and* C_2 *with edges* $(p_1, p_2), ..., (p_{k-1}, p_k) \in C_1$ *and* $(q_1, q_2) \in C_2$*. We call a tangle a v-tangle if* $p_1 \in H^+(q_1, q_2)$*,* $p_k \in H^+(q_1, q_2)$ *and* $p_i \in H^-(q_1, q_2)$ *for all* $1 < i < k$*. We call a tangle a reversed v-tangle if* $p_1 \in H^-(q_1, q_2)$*,* $p_i \in H^-(q_1, q_2)$ *and* $p_i \in H^+(q_1, q_2)$ *for all* $1 < i < k$*.*[14]



(a) ascending v-tangle (b) ascending reversed v-tangle

(c) descending v-tangle (d) descending reversed v-tangle

Figure 2.5: Supowit's algorithm may produce the tangles in Figures (a) and (c). Untangling these by deleting the solid edges and adding dashed ones may produce new tangles between other chains, which must be one of the form shown in Figures (b) and (d).

Definition 2 [**Descending tangles**] *Given an edge* (p_i, p_j) *in an descending chain, let* $H^+(p_i, p_j)$ *be the open half-plane containing the point* $(c_1(p_i) + 1, c_2(p_i) + 1)$ *and* $H^-(p_i, p_j)$ *be the open half-plane containing the point* $(c_1(p_i) - 1, c_2(p_i) - 1)$. *Assuming that we have two chains* C_1 *and* C_2 *with edges* $(p_1, p_2), ..., (p_{k-1}, p_k) \in C_1$ *and* $(q_1, q_2) \in C_2$. *We call a tangle created by* C_1 *and* C_2 *a v-tangle if* $p_1 \in H^-(q_1, q_2)$, $p_i \in H^-(q_1, q_2)$ *and* $p_i \in H^+(q_1, q_2)$ *for all* $1 < i < k$. *We call a tangle a reversed v-tangle if* $p_1 \in H^+(q_1, q_2)$, $p_k \in H^+(q_1, q_2)$ *and* $p_i \in H^-(q_1, q_2)$ *for all* $1 < i < k$. [14]

Each v-tangle can be eliminated by removing $p_2...p_{k-1}$ from C_1 and inserting

Algorithm 6 2DUntangling ($\mathcal{P}$)

Input: A set of 2-dimensional points $\mathcal{P} = \{p_1, p_2, ..., p_n\}$, where $|\mathcal{P}| = n$.
Output: A set of 2-dimensional chains parition $\mathcal{S}' = \{C_1, C_2, ..., C_m\}$ with no intersection.

```
1: S' ← ∅
2: for l = 1...n do
3:    Run Supowit(P) algorithm to get C_l, C_{l+1}, ..., C_m
4:    for i = k...l do
5:       for j = i − 1...l  do
6:          Find all the v − tangles between C_i and C_j and untangle them
7:       end for
8:    end for
9:    P ← P \ C_l
10:   S' = S' ∪ {C_l}
11: end for
12: return S'
```

them to C_2, shown in Figure 2.5(a) and Figure 2.5(c). While this process may create new tangles, fortunately, these new tangles must be *reversed v-tangles* as shown in Figures 2.5(b) and Figure 2.5(d).

Lemma 1 *All tangles generated by Supowit's algorithm are v-tangles.* [4]

Lemma 2 *All tangles created after one pass of untangling algorithm must be reversed v-tangles.* [4]

More importantly, Arroyuelo et al. [4] proved that given an order of the chains, after one untangling pass, the lowest chain does not tangle with any other chains. This allows us to untangle all the chains by running one untangling pass, extracting the lowest chain, and then recursing on the remaining chains until no chains left. Algorithm 6 gives the pseudocode of the Arroyuelo et al.'s untangling algorithm. It is important to note that Supowit's algorithm must be run in each iteration (line 3 in Algorithm 6) because the untangling process relies on the special structure of the tangles produced by the this algorithm.

Claude et al. [14] gave another untangling method by transforming the reversed v-tangles back to v-tangles. This eliminates the need to rerun Supowit's algorithm in each iteration. They didn't prove the correctness of their untangling method, so their implementation includes a function to check if all the chains are untangled at

the end of the untangling step. This test procedure, of course, requires extra time compared with Arroyuelo et al.'s algorithm, but the experimental results show that their algorithm is much faster than Arroyuelo et al.'s algorithm, due to not having to reuse Supowit's algorithm in each iteration.

2.3.4 Range Searching over the Chains

Consider an ascending chain C (symmetrically for a descending chain). For an orthogonal range query $\mathcal{R}$, if there exists points of C that lie in $\mathcal{R}$, they must be a contiguous interval of the ordered list of points along the chain. This allows us to answer any orthogonal range query by performing a binary search. During the search, we use the range query $\mathcal{R} = [a_1 : b_1] \times [a_2 : b_2]$ as the comparison to determine the next step of the binary search. Let point $p = (c_1(p), c_2(p))$ be the median point during one binary search step. The comparison between p and $\mathcal{R}$ has three possible outcomes: if $c_1(p) < a_1$ or $c_2(p) < a_2$, we continue searching the upper part of the chain, because all points stored in the lower part of the chain either have their x-coordinates less than a_1 or have their y-coordinate less than a_2; if $c_1(p) > b_1$ or $c_2(p) > b_2$, we search the lower part of the chain, since all points stored in the upper part of the chain either have their x-coordinates greater than b_1 or have their y-coordinate greater than b_2; if $a_1 \leq c_1(p) \leq b_1$ and $a_2 \leq c_2(p) \leq b_2$, we will return the point p and stop searching since p lies inside $\mathcal{R}$. Then we apply a linear scan along the chain to find the first and last point in the range and report all the points in between. This takes $O(\lg n + k)$ time, where n is the number of points in the chain and k is the number of reported points. We call this procedure a *chain query*. Note that a single extension of this can be used to determine the relative position between a chain and a point. Let p' be a point that does not lie in the chain. When applying a chain query on C for p', the search ends at two consecutive points p_i and p_{i+1}. We call p' is to the left to C if the cross product of p_1, p_{i+1} and p' is negative and call p' is to the right of C if the cross product of p_1, p_{i+1} and p' is positive. We further extend it to a range query box. A query box $\mathcal{R}$ is to the left of a chain C if its four corner points are to the left of C, $\mathcal{R}$ is to the right of a chain C if its four corner points are to the right of C and $\mathcal{R}$ intersect a chain C if one or more corner points on the opposite side of C. Note that a chain could intersect the query box without any point lies in the box.

Algorithm 7 2DRangeSearching $(\mathcal{S}, \mathcal{R})$

Input: A set of untangled monotonic ascending chains $\mathcal{S} = \{C_1, C_2, ..., C_m\}$ in the plane and a range query $\mathcal{R} = [a_1 : b_1] \times [a_2 : b_2]$.
Output: All the points in $\mathcal{R}$.

```
1: low = 1
2: high = m
3: while low ≤ high do
4:    mid = ⌊(low + high)/2⌋
5:    if C_mid is to left of R then
6:       low = mid + 1
7:    else if C_mid is to right of R then
8:       high = mid − 1
9:    else
10:      pos = mid
11:      while pos ≤ high do
12:         perform a chain query on C_pos using R
13:         if C_pos intersects R then
14:            report all the points in C_pos that lie in R
15:            pos + +
16:         else
17:            break
18:         end if
19:      end while
20:      pos = mid − 1
21:      while pos ≥ low do
22:         perform a chain query on C_pos using R
23:         if C_pos intersects R then
24:            report all the points in C_pos that lie in R
25:            pos − −
26:         else
27:            break
28:         end if
29:      end while
30:   end if
31: end while
```

Given a partition of $\mathcal{P}$ into a collection of ascending chains and a collection of descending chains, a naive implementation of range reporting over $\mathcal{P}$ has to apply the above binary search to each chain in turn. In general, this is the best one can do. If the chains are untangled, however, we can do better. Without loss of generality, we focus on the ascending chains.

Let $\mathcal{S} = \{C_1, C_2, ..., C_m\}$ be a set of untangled monotonic ascending chains, where chains are ordered from left to right by their leftmost points. For each chain, we add two extreme points to avoid special boundary cases: one is to the beginning of the chain and the other is to the end of the chain. Note that because $m < n$, adding these $2m$ extreme points does not affect the asymptotic running time of the algorithm. Let l_i be the length (the number of points) of chain C_i. Given a query range $\mathcal{R} = [a_1 : b_1] \times [a_2 : b_2]$, we need to find all the chains that intersect $\mathcal{R}$ and report all the points of each such chain that lie in $\mathcal{R}$. Because chains in $\mathcal{S}$ are ascending, untangled and sorted, we can apply binary search over the chains. Each chain access, during the chain binary search, is equivalent to a chain query. This allows us to discard all the chains to one side of current chain if it does not intersect the range query. If it does, we then sequentially inspect it adjacent chains one by one until we find the last chain C_i that is to the left of $\mathcal{R}$ and the first chain $C_{i+m'}$ that is to the right of $\mathcal{R}$. Algorithm 7 shows the range searching steps over the ascending chains. Lines 5 to 8 represent the cases when the selected chain does not intersect the query box. Lines 10 to 30 indicates the sequential inspection steps once we find a chain that intersects the query box.

Next, we bound the query time. The binary search over chains takes $O(\lg m \lg n)$ time, where m is the total number of untangled chains, which is bounded by $O(\sqrt{n})$, and n is the maximal length of all chains. Assuming that the number of chains that intersect $\mathcal{R}$ is m', then for each of the m' chains, we need to perform a chain query. This takes $O(m' \lg n + k)$ time, where k is the number of reported points. The following theorem summarizes the properties of the 2-dimensional chain decomposition algorithm and its range searching algorithm.

Theorem 3 *A 2-dimensional point set $\mathcal{P}$ with n points can be partitioned into two sets of untangled chains: an ascending chain set and a descending chain set with a total number of chains at most $O(\sqrt{n})$. This process requires $O(n^2)$ time using linear space. A range query can be answered in $O(\lg m \lg n + m' \lg n + k)$ time using this data structure, where m is the total number of chains, m' is the number of chains intersecting the query range, and k is the number of points within the range.*

2.4 3-Dimensional Chain Decomposition

Given the practical efficiency of the 2D range query based on the chain decomposition that are reported by Claude et al. [14], it is natural to ask whether the data structure can be generalized to higher dimensions and whether its performance is as good as in 2D. However, there are three challenges that need to be addressed.

First, a monotonic chain in 2D that is sorted based on the increasing first coordinates (x-coordinates) can only be ascending or descending in its second dimension. In d dimensions, a chain can be ascending or descending in each of the $d-1$ dimensions after the first dimension. Thus, we obtain 2^{d-1}, rather than two, "chain orientations". In 3D, this is probably not a major problem, as there are still only 4 possible chain orientations.

The second issue presents a much greater challenge: there is no obvious notion of untangling chains in $d \geq 3$. Indeed, two arbitrary chains in dimension higher than 3 are unlikely to intersect, but the key observation exploited in 2D is that non-intersecting chains in 2D can be ordered so that, if one chain is to the left (symmetric to the right side) of the query box, all chains on its left side can be discarded because all chains are untangled. This is the key to the binary search for a chain that intersects the query box. No such ordering exists in 3D.

The third and final challenge is the classical curse of dimensionality: as the dimension increases, the number of monotonic chains into which we need to partition $\mathcal{P}$ also increases, which, in turn, increases the cost of producing the partition and the query cost of the data structure.

Chapter 3

3-Dimensional Chain Decomposition and Range Searching

In this chapter, we will talk about decomposing a set of points in 3D into chains and how to perform range queries over these chains. Section 3.1 discusses how to choose a proper chain partition method in 3-dimensional space with the goal of achieving efficient range query performance. In Section 3.2, we propose an alternative chain untangling algorithm in 2D, which in practice runs 25% faster than the Claude et al.'s algorithm [14]. Finally, the complete 3D chain decomposition and range query algorithms are presented in Section 3.3.

3.1 Chains That Can Be Untangled In 3D

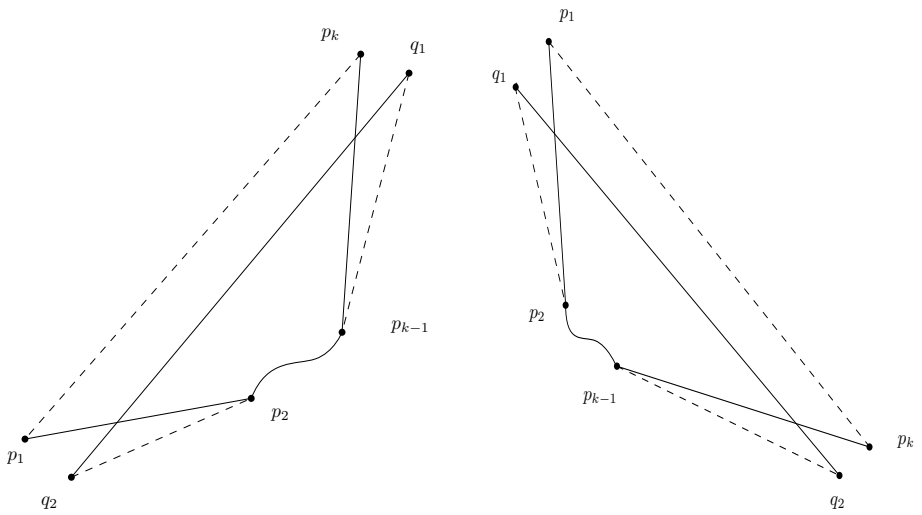
Two algorithms for computing a chain partition in 3-dimensional space have been proposed by Wei [37]. The first one is an extension of Fomin et al.'s algorithm [39]. Without loss of generality, all the points are first projected into the plane defined by their first two coordinates (xy-plane). After that, we find both the longest ascending and descending chains in the xy-plane, choose the longer of the two, and then project all points of the longer chain into the xz-plane (a plane defined by the 1st and 3rd coordinates). Given the points found in the previous step, we find the longest ascending and descending chains again on the xz-plane, choose the longer one, and extract the points on the chain from the point set. We continue this process until no points are left. This partition algorithm produces $O(n^{\frac{3}{4}})$ chains and takes $O(n^{\frac{7}{4}} \lg n)$ time [37].

The second algorithm is an adaptation of Supowit's algorithm, which was described in Section 2.3.2. At the beginning, all the points are sorted based on their x-coordinates. The algorithm process every point iteratively similarly to the 2-dimensional version. Each point is either added to an existing chain, which has the maximal length and satisfies the monotonic properties, or assigned to a new chain. The running time of this algorithm is $O(n^2)$, which can be improved to $O(n \lg^2 n)$ by using a 2-dimensional range tree [37].

Recall that the reason we can apply an efficient range query based on the chain decomposition in 2D is that the chains are monotonic, untangled and ordered. If we want to achieve an efficient range query in 3D, the chains generated in 3D also need to satisfy some properties that can speed up the range query performance. However, it is unclear for us which properties the chains produced by the second algorithm have, so that the range searching performance can be improved, like in 2D. The chains produced by the first algorithm on the other hand suggest a natural query strategy: first, we find the chains in the xy-plane that cross the xy-projection of the query box, then find the subchains of these chains in the xz-plane that cross the xz-projection of the query box, and finally search each of these subchains in the xz-plane using binary search. Essentially, this algorithm is an extension of the 2D chain decomposition algorithm. The projection steps convert a 3-dimensional case into two 2-dimensional sub-cases. In this way, all the properties hold by the chains in the 2D decomposition can be reserved in 3D, which allows us to perform efficient range queries over the chains in 3D. Before we introduce the 3D chain decomposition and range searching algorithms, in the next section, we propose an alternative chain untangling method in 2D, which in practice outperforms Claude et al.'s algorithm [14] by avoiding the transformation step at each iteration.

3.2 An Alternative Chain Untangling Algorithm

Given the chain decomposition algorithm described in the previous section, in order to achieve an efficient range query performance in 3D, all chains generated in either xy-plane or xz-plane must be untangled, so that we can apply adaptive range queries over the resulting chains in 3D. Claude et al. [14] proposed a faster chain untangling algorithm in 2D, which eliminates the need to rerun Supowit's algorithm in each iteration. Instead, Supowit's algorithm is applied only once at the beginning of the algorithm. However, this algorithm needs to apply a chain transformation in each iteration. From Lemma 2, we know that after one pass of the untangling procedure, all newly generated tangles must be reversed v-tangles. What Claude et al.'s algorithm does is to use a mapping function to transform a set of chains that only has reversed v-tangles into a set of chains that has only v-tangles. After the transformation, since all tangles are v-tangles, we now able to untangle them through



(a) ascending *reversed v-tangle* (b) descending *reversed v-tangle*

Figure 3.1: Two different types of the reversed v-tangles are shown in Figure (a) and (b). The untangling process can be done by removing $p_2...p_{k-1}$ from their original chain and inserting them to the chain containing points q_1 and q_2.

another untangling pass. Then, the mapping function is applied again to restore the original position of the chains. However, this linear transformation can have a huge impact on the running time if the point set is large, since we need to apply the mapping function twice for every single point in the point set. In this section, we introduce another chain untangling method, which eliminates the transformation in each iteration while still applying Supowit's algorithm only once. Instead of using the mapping function twice for all points, we only apply the function to the points corresponding to the reversed v-tangles, since the points on the chains that do not intersect will never change after the mapping. This further leads us to an untangling method to the reversed v-tangles while eliminating the need for the transformation. Figure 3.1 shows how to untangle the reversed v-tangles for both ascending and descending chains using the map function. Algorithm 8 gives the pseudocode for our chain untangling algorithm. We divide each untangling iteration into two sub-procedures. First, we untangle all the v-tangles and extract the lowest chain, which is untangled, then symmetrically we untangle all the reversed v-tangles and extract the highest chain. We apply this untangling procedure until no chains intersect.

We currently can not prove that when Algorithm 8 finishes, all chain are untangled. Similarly to Claude et al.'s algorithm, we add a linear tangle checking method

Algorithm 8 2DUntangling_V2 ($\mathcal{P}$)

Input: A set of points $\mathcal{P} = \{p_1, p_2, ..., p_n\}$ in 2D, where $|\mathcal{P}| = n$.
Output: A 2-dimensional chain partition $\mathcal{S} = \{C_1, C_2, ..., C_m\}$ where no two chains intersect.

1: $\{C_1, C_2, ..., C_m\} \leftarrow$ run Supowit's algorithm using $\mathcal{P}$
2: $start = 1$
3: $end = m$
4: **while** $start < end$ **do**
5: **for** $i = end...start$ **do**
6: **for** $j = i - 1...start$ **do**
7: Find all the *v-tangles* between C_i and C_j and untangle them
8: **end for**
9: **end for**
10: $start++$
11: **if** all the chains are untangled **then**
12: break
13: **end if**
14: **for** $i = start...end$ **do**
15: **for** $j = i + 1...end$ **do**
16: Find all the *reversed v-tangles* between C_i and C_j and untangle them
17: **end for**
18: **end for**
19: $end--$
20: **if** all the chains are untangled **then**
21: break
22: **end if**
23: **end while**
24: **return** $\mathcal{S}$

in each iteration in the advent of a failure. However, in practice, we found in no cases Algorithm 8 fails. If there is a failure, we can run the original untangling algorithm, since both algorithms have the same running time [14]. In practice, the tangle checking method, however, turns out to be extremely effective to terminate the entire untangling process, since the number of intersections drops dramatically after one untangling pass, which means it is not necessary to execute all untangling loops from start to end, thus the running time of the untangling algorithm can be further reduced.

3.3 3D Untangled Monotonic Chain Decomposition

Algorithm 9 3DChainDecomposition($\mathcal{P}$)

Input: A point set $\mathcal{P} = \{p_1, p_2, ..., p_n\}$ in 3-dimensional space.

Output: A set of untangled ascending chains and a set of untangled descending chains in the xy-plane; each chain in the xy-plane is associated with a set of untangled ascending chains and a set of descending chains in the xz-plane.

1: run Fomin's algorithm on the xy-projection of $\mathcal{P}$ to obtain two point subsets: $\mathcal{P}_{xy\_as}$ and $\mathcal{P}_{xy\_de}$
2: let $\mathcal{S}_{xy\_as}$= $\{C_1, C_2, ..., C_{m_1}\}$ be the chain set in the xy-plane after running 2DUntangling_V2($\mathcal{P}_{xy\_as}$)
3: **for** $C_i \in \mathcal{S}_{xy\_as}$ **do**
4: run Fomin's algorithm on the xz-projection of $\mathcal{P}(C_i)$ to obtain two point subsets: $\mathcal{P}_{xz\_as}$ and $\mathcal{P}_{xz\_de}$
5: $\mathcal{S}_{xz\_as}(C_i) \leftarrow$ run 2DUntangling_V2($\mathcal{P}_{xz\_as}$) in the xz-plane
6: $\mathcal{S}_{xz\_de}(C_i) \leftarrow$ run 2DUntangling_V2($\mathcal{P}_{xz\_de}$) in the xz-plane
7: **end for**
8: let $\mathcal{S}_{xy\_de}$=$\{C_1, C_2, ..., C_{m_2}\}$ be the chain set in the xy-plane after running 2DUntangling_V2($\mathcal{P}_{xy\_de}$)
9: **for** $C_i \in \mathcal{S}_{xy\_de}$ **do**
10: run Fomin's algorithm on the xz-projection of $\mathcal{P}(C_i)$ to obtain two point subsets: $\mathcal{P}_{xz\_as}$ and $\mathcal{P}_{xz\_de}$
11: $\mathcal{S}_{xz\_as}(C_i) \leftarrow$ run 2DUntangling_V2($\mathcal{P}_{xz\_as}$) in the xz-plane
12: $\mathcal{S}_{xz\_de}(C_i) \leftarrow$ run 2DUntangling_V2($\mathcal{P}_{xz\_de}$) in the xz-plane
13: **end for**
14: **return**

In this section, the algorithm for the 3-dimensional chain decomposition is presented. Algorithm 9 shows the pseudocode for this process. Unlike the chain decomposition method described in Section 3.1 where the chains generated in either xy-plane or xz-plane could intersect, our chain decomposition method eliminates all the tangles generated during the partition by applying the untangling process mentioned in previous section, so that the resulting chain decomposition can support an adaptive range query in 3D.

The algorithm is divided into two stages. In the first stage, all points are sorted based on their x-coordinates and are projected on the xy-plane. Then, the greedy algorithm of Fomin et al. [39] is applied to partition the point set $\mathcal{P}$ in the xy-plane into two subsets: $\mathcal{P}_{xy\_as}$ and $\mathcal{P}_{xy\_de}$ where $\mathcal{P}_{xy\_as}$ contains the points derived the ascending chain set in the xy-plane and $\mathcal{P}_{xy\_de}$ contains the points derived from the

descending chain set in the xy-plane (line 1 in Algorithm 9). For each subset, we first apply Supowit's algorithm to produce a new chain decomposition. Then, we apply the chain untangling algorithm to the two sets of chains to make sure that all the resulting chains in the xy-plane are untangled (line 2 and 8 in Algorithm 9).

In the second stage, we apply the same procedure to the points in each of the chains produced in the first phase to partition these points into untangled chains in the xz-plane. We denote $\mathcal{S}_{xy\_as}$ and $\mathcal{S}_{xy\_de}$ to be the untangled ascending chain set and untangled descending chain set in the xy-plane. For each ascending chain $C_i \in \mathcal{S}_{xy\_as}$, we first project all points in C_i (denote as $\mathcal{P}(C_i)$) to the xz-plane and then apply Fomin et al's algorithm to obtain two subsets ($\mathcal{P}_{xz\_as}$ and $\mathcal{P}_{xz\_de}$) of $\mathcal{P}(C_i)$ that contain the points from the ascending chains and the descending chains in the xz-plane (line 4 in Algorithm 9). After that, for each subset of points, we apply the Supowit's algorithm to get a new chain partition and apply the chain untangling process to make sure all chains are untangled (lines 5 to 6). We denote $\mathcal{S}_{xz\_as}(C_i)$ and $\mathcal{S}_{xz\_de}(C_i)$ to be the untangled ascending chain set and untangled descending chain set in the xz-plane that are derived from $\mathcal{P}(C_i)$.

Next, we analyze the running time of Algorithm 9. In 2D, the total running time for computing the chain decomposition and untangling the chains is $O(m^2n + n\lg n)$, where n is the total number of points and m is the number of resulting chains [4]. This bounds lines 1, 2 and 8 in Algorithm 9. Let $|C_i|$ represents the number of points in chain C_i and $|\mathcal{S}(C_i)|$ indicates the number of subchains generated by C_i in the xz-plane. Let $|\mathcal{P}_{as}|$ and $|\mathcal{P}_{de}|$ be the total number points in $\mathcal{P}_{xy\_as}$ and $\mathcal{P}_{xy\_de}$ respectively and m_{as} and m_{de} be the total number of subchains generated in the xz-plane for the point sets $\mathcal{P}_{xy\_as}$ and $\mathcal{P}_{xy\_as}$ respectively. Then, the running time $\mathcal{T}(|\mathcal{P}_{as}|)$ from lines 3 to 7 can be expressed as:

$$\begin{aligned}\mathcal{T}(|\mathcal{P}_{as}|) &= |\mathcal{S}(C_1)|^2|C_1| + |C_1|\lg(|C_1|) + ... + |\mathcal{S}(C_{m_1})|^2|C_{m_1}| + |C_{m_1}|\lg(|C_{m_1}|) \\ &\leq ((m_{as})^2 + \lg(|\mathcal{P}_{as}|))(|C_1| + ... + |C_{m_1}|) \\ &\leq (m_{as})^2|\mathcal{P}_{as}| + |\mathcal{P}_{as}|\lg(|\mathcal{P}_{as}|)\end{aligned} \tag{3.1}$$

Symmetrically, the running time $\mathcal{T}(|\mathcal{P}_{de}|)$ from lines 9 to 13 can be expressed as:

$$\mathcal{T}(|\mathcal{P}_{de}|) \leq (m_{de})^2|\mathcal{P}_{de}| + |\mathcal{P}_{de}|\lg(|\mathcal{P}_{de}|) \tag{3.2}$$

Let $m = m_{as} + m_{de}$. Since $|\mathcal{P}_{as}| + |\mathcal{P}_{de}| = n$, the total running time

$$\mathcal{T}(n) \leq O(m^2 n + n \lg n) \tag{3.3}$$

In [37], the author proved that the upper bound for m is $O(n^{\frac{3}{4}})$, so the running time for the chain decomposition and chain untangling in 3D is at most $O(n^{\frac{5}{2}})$.

3.4 Two Chain Query Methods

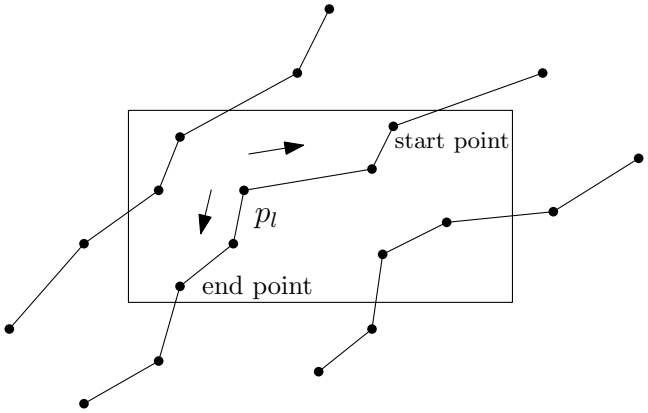


Figure 3.2: A chain query over an ascending chain.

Recall that for the 2D chain-based range searching method discussed in Section 2.3.4, given an ascending chain (symmetrically for a descending chain), we use a binary search to find out whether it intersects the range query box or to determine the relative position between the chain and the query box. We call this a chain query. Suppose a point p_l is found that lies in the range box during a chain query, see Figure 3.2. In order to find all points in the chain that lie in the query box, two search strategies can be applied: first, after the point p_l is located, a one-by-one linear scanning step on the two opposite directions of the chain can be used to find all the points on the path that lie in the query range; the second is to perform another two binary searches to locate the first and the last points of the chain within the range.

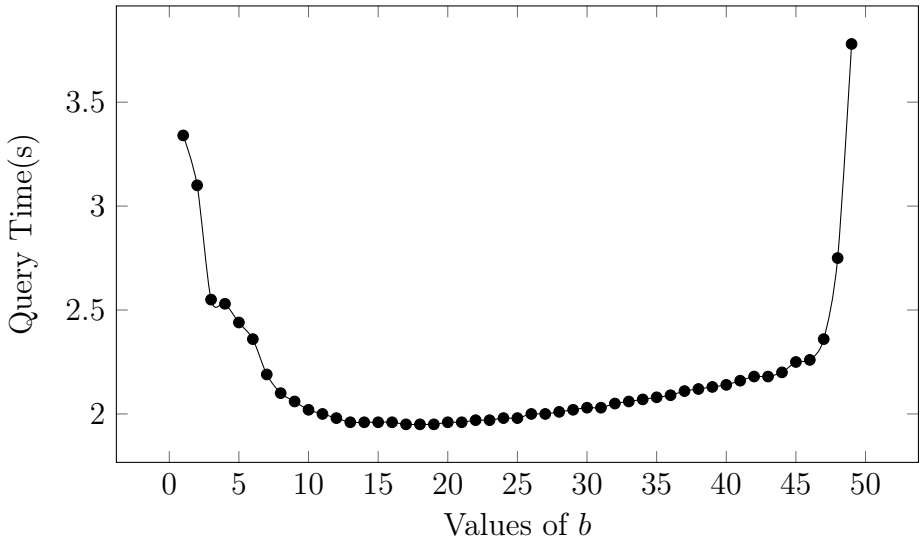


Figure 3.3: One example of revealing the relationship between the values of b and total query time (in seconds). For this one, the optimal value of $b = 17$.

However, in our experimental evaluations, the second method was consistently faster than the first one, which means the cost of the two additional binary searches was outweighed by the cost of checking for every point on the path until we reach out of the query range. In an attempt to avoid the two additional binary searches while not paying the penalty of checking for each point in the query range on the chain, we improve the first method as follows. Given some parameter b, we scan forward and backward from p_l but inspect only every bth point. We apply this procedure until we find the first point in each direction that does not lie in $\mathcal{R}$. Let p_b and p_f be the last points in each direction that lie in $\mathcal{R}$ during the inspection. Note that all points between p_b and p_f lie in the query range, because the chain is a monotonic chain. However, there may still be up to $b - 1$ points preceding p_b and up to $b - 1$ points succeeding p_f that also lie in $\mathcal{R}$. We then check each point in turn backwards from p_b and forward from p_f to find all these points. The parameter b provides a trade-off: a large value of b could potentially identify a large range of points in $\mathcal{R}$ with only few comparisons, but it also increases the number of points before p_b and after p_f that may be lie in $\mathcal{R}$ and all of these points require a one-by-one inspection; on the other hand, with smaller values of b, the amount of inspections gets smaller, but the potential savings in the search also get smaller.

We determine the optimal value of b as follows: we first set $b = 1$ and increase it

by 1 until we reach a predefined maximal value of d; we then choose the value of b with minimal query time. In our experimental evaluation, we observed that, as the value of b increased, the query time first decreased and then increased. Figure 3.3 shows one example of the relationship of the values of b and the total query time. Since determining this value of b is costly and requires a characterization of the queries, it can not be incorporated with any real-world use of the chain-decomposition-based data structure. However, we determined b for two reasons: first, if b is fairly independent of the data set and query type, then determining the optimal value of b is a one-time learning step of the data structure for a given platform, which is reasonable in practice; second, to determine which of the two chain query methods performs best, it is useful to choose a value of b that maximizes the query performance.

3.5 3D Chain Range Searching

After the chain decomposition, the resulting monotonic chains in 3D could have 4 possible monotonic orientations (see details in Section 2.4). We define each of them as follows: for any two points p_i and p_j on an $ascending_{(y)}$-$ascending_{(z)}$ chain, if $c_1(p_i) \leq c_1(p_j)$, then $c_2(p_i) \leq c_2(p_j)$ and $c_3(p_i) \leq c_3(p_j)$; for any two points p_i and p_j on a $ascending_{(y)}$-$descending_{(z)}$ chain, if $c_1(p_i) \leq c_1(p_j)$, then $c_2(p_i) \leq c_2(p_j)$ and $c_3(p_i) \geq c_3(p_j)$; we call a chain in 3D a $descending_{(y)}$-$ascending_{(z)}$ chain if for any two points p_i and p_j on the chain, $c_1(p_i) \leq c_1(p_j)$, then $c_2(p_i) \geq c_2(p_j)$ and $c_3(p_i) \leq c_3(p_j)$; finally, a chain is called a $descending_{(y)}$-$descending_{(z)}$ chain, for any two points p_i and

Algorithm 10 3DRangeSearching_Ascending($\mathcal{S}_{xy\_as}, \mathcal{R}$)

Input: A set of untangled monotonically ascending chains $\mathcal{S}_{xy\_as}$ in the xy-plane, where each chain $C \in \mathcal{S}_{xy\_as}$ is associated with an untangled monotonically ascending chain set $\mathcal{S}_{xz\_as}(C)$ and an untangled monotonically descending chain set $\mathcal{S}_{xz\_de}(C)$ in the xz-plane and a range query $\mathcal{R} = [a_1 : b_1] \times [a_2 : b_2] \times [a_3 : b_3]$.

Output: All the points within $\mathcal{R}$.

1: apply the 2D range searching in the xy-plane to find all the candidate chains $C_l, C_{l+1}, ...C_k$ that intersect query range $[a_1 : b_1] \times [a_2 : b_2]$
2: **for** $i = l...k$ **do**
3: apply the 2D range searching over $\mathcal{S}_{xz\_as}(C_i)$ and $\mathcal{S}_{xz\_de}(C_i)$ in the xz-plane and report all the points within $\mathcal{R}$
4: **end for**

p_j on the chain, if $c_1(p_i) \leq c_1(p_j)$, then $c_2(p_i) \geq c_3(p_j)$ and $c_3(p_i) \geq c_3(p_j)$.

After the chain decomposition process, we get two sets of untangled monotonic chains in the xy-plane: $\mathcal{S}_{xy\_as}$ and $\mathcal{S}_{xy\_de}$. For each chain $C \in \mathcal{S}_{xy\_as}$, we have two sets of untangled monotonic subchains in the xz-plane: $\mathcal{S}_{xz\_as}(C)$ and $\mathcal{S}_{xz\_de}(C)$. By adding the associated y-coordinates to the points of $\mathcal{S}_{xz\_as}(C)$, we will get an $ascending_{(y)}$-$ascending_{(z)}$ chain set in 3D. By adding the associated y-coordinates to the points of $\mathcal{S}_{xz\_de}(C)$, we will get an $ascending_{(y)}$-$descending_{(z)}$ chain set in 3D. For the chains in $\mathcal{S}_{xy\_de}$, the situation is symmetric.

The 3D range searching using this chain decomposition can be considered as an extension of the 2D range searching using the 2D chain decomposition, which was described in Section 2.3.4. Algorithm 10 shows the range query steps over ascending chains $\mathcal{S}_{xy\_as}$. Given a query range $\mathcal{R} = [a_1 : b_1] \times [a_2 : b_2] \times [a_3 : b_3]$, we first find all the candidate chains in the xy-plane that intersect the query range $[a_1 : b_1] \times [a_2 : b_2]$, since these are the chains containing all the points with their x and y coordinates lie in the range (line 1). Then, for each candidate chain C in xy-plane, we apply another secondary search on the subchains derived from C in the xz-plane and report all the points that lie in the query range (lines 2 to 4). The only difference comparing to the 2D range searching comes from the chain query method (binary search over a single chain) in the xz-plane. Instead of using the x and z coordinates of the points to perform binary search over the chain, we also need to consider the y-coordinates of the points. Because all chains generated in the xz-plane are 3D monotonic (either an $ascending_{(y)}$-$ascending_{(z)}$ chain or an $ascending_{(y)}$-$descending_{(z)}$ in our case), which makes the binary search over the points of a chain by using all x, y and z coordinates feasible.

Let us consider the query time. Let n be the total number of points and m be the total number of monotonic chains in the xy-plane. Then, the 2D range searching for the candidate chains in line 1 of Algorithm 10 takes $O(\lg m \lg n + m' \lg n)$ time, where $m' \leq m$ is the number of chains that intersect xy-projection of $\mathcal{R}$. For lines 2 to 4, we assume that the total number of subchains of all the candidate chains in the xz-plane is m_s, then the binary search over these chains takes $O(m' \lg m_s \lg n)$. Let m'_s be the number of chains in the xz-plane that intersect $\mathcal{R}$. After adding the chain query time for each intersected chain in the xz-plane, the total query time is $O(\lg m \lg n +$

$m' \lg m_s \lg n + m'_s \lg n)$, where $m_s \geq m'_s$ is bounded by $O(n^{\frac{3}{4}})$ [37]. The following theorem summarizes the properties of the 3-dimensional chain decomposition and its range searching algorithms.

Theorem 4 *A 3-dimensional n-point set can be partitioned into at most $O(n^{\frac{3}{4}})$ untangled monotonic chains in $O(n^{\frac{5}{2}})$ time using linear space. Each range query can be answered in $O(\lg m \lg n + m' \lg m_s \lg n + m'_s \lg n + k)$ time, where m is the total number of chains in the xy-plane, m' is the number of chains in the xy-plane that intersect the xy-projection of the query range, m_s is the total number of subchains in the xz-plane and m'_s is the number of chains in the xz-plane that intersect $\mathcal{R}$.*

Chapter 4

Experimental Evaluation

The experimental evaluation conducted in this chapter has been divided into two parts: the 2D experimental evaluation and the 3D experimental evaluation. In the 2D evaluation, we first examined the existing k-d tree implementation in CGAL [34] and compared it with our k-d tree implementations. Then, we compared our chain untangling algorithm with the previous untangling algorithm and demonstrated the range query results for different data sets. In 3D, we started with a comparison of the two different 3D chain decomposition algorithms and gave their range query results while comparing them with the 3-dimensional k-d tree.

All the experiments were performed on a machine with an AMD Opteron (tm) processor 4176E 1200MHz with 126K of L1 Cache, 512K of L2 Cache, 5118K of L3 Cache and 16GB of DDR3 main memory with 1333 MHz clock speed. It runs a 64-bit operating system in GNU/Linux-Debian with kernel 3.16.36-1-deb8u1. The complier used was GNU/gcc version 4.9.2. All our implementations were compiled with optimization level `-O2`. We used Perf (version 3.16.7-ckt20) as the profiler tool for the hardware performance evaluation.

4.1 2D Experimental Evaluation

4.1.1 Data Sets and Range Queries

Three different types of data were used to measure the performance of each implemented data structure: a uniform random data set, a real-world geographic data set, and an Amazon review data set. For the first data set, we generated each uniform random n-point set by setting each coordinate sequence (x-coordinate sequence or y-coordinate sequence) to a uniform random permutation of $\{1, 2, ..., n\} \subseteq \mathbb{Z}^+$. The second data is a geographic mapping data used to evaluate "The Traveling Salesman Problem" [15]. For data set **World** that we used in our implementation, each point is

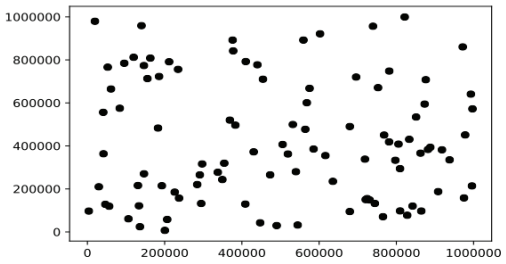
Data type	Data set	Number of points
Random data	Rand_1M	1,000,000
	Rand_2M	2,000,000
Map data	China	71,009
	World	1,904,711
Amazon review data	Movies	1,697,523
	Electronics	1,689,188
	CDs	1,097,592
	Kindle	982,617

Table 4.1: Data set sizes.

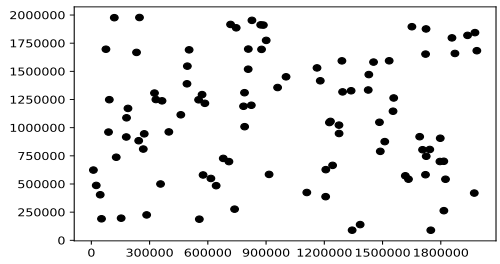
associated with a city on the map with its corresponding latitude (x-coordinate) and longitude (y-coordinate) rounded to four decimals. For data set **China**, all points are derived from the point set **World** with their GPS coordinates (see detail in [15]). The third one is provided by "Web data: Amazon reviews" [23]. Each record represents one review submitted on Amazon. We converted each review into a 2-dimensional point with coordinates: **asin** and **ReviewTime**. **asin** is the integer representation of the product ID and **ReviewTime** is derived from the time record that indicated when the review was submitted, originally represented as a UNIX timestamp. The **ReviewTime** of each record is obtained by first subtracting it by the minimal review time of the data set and then dividing the result by 100. For the Amazon data sets, we expect that numerous points share the same x-coordinate (since products are likely to have more than one review), and it is possible that multiple points share the same y-coordinate (because different products may be reviewed at exactly the same time). We are able to handle the above circumstances where multiple points have identical x or y coordinate in our implementation. However, we only kept one point of each identical group with the same x and y coordinates [1]. For each type of data, at least two sets with different sizes were chosen. Table 4.1 shows the size of each data set that was used during the experimental evaluation and Figure 4.1 shows the point distribution of a small random samples (100 random points) from each data set.

To obtain a comprehensive analysis of the range query performance, 7 types of range queries with different characteristics were used. For each point set $\mathcal{P}$, we first identified its bounding box as $[a_1 : b_1] \times [a_2 : b_2]$, where $a_1 = min_{p \in \mathcal{P}}(c_1(p))$, $a_2 =$

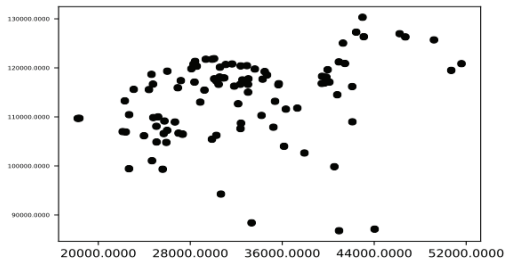
[1] This does not favour the approaches based on chain decomposition as it will decrease the degree of presortedness of the data set. It would however be interesting to rerun these experiments in the future to see how much this affects the experimental results.



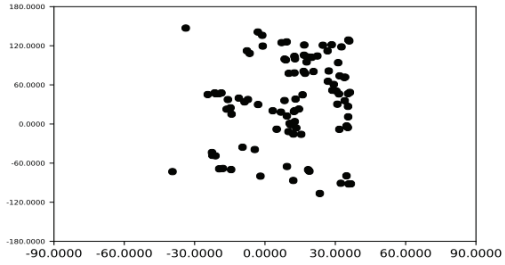
(a) Rand_1M



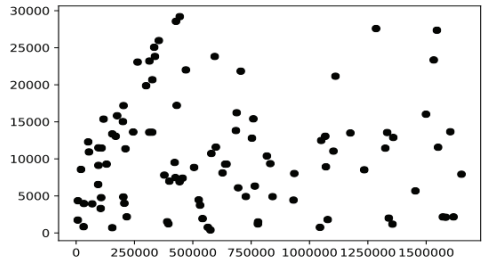
(b) Rand_2M



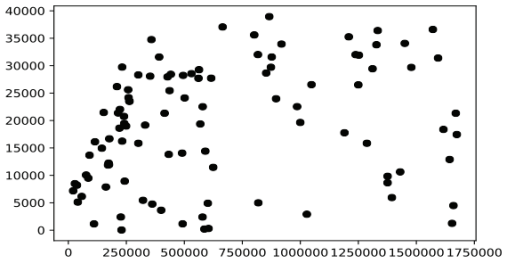
(c) China



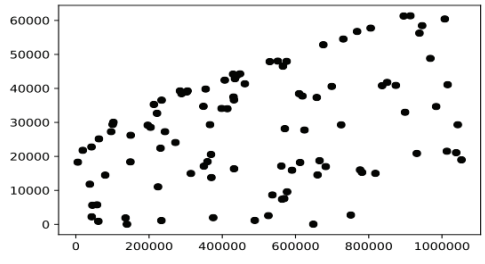
(d) World



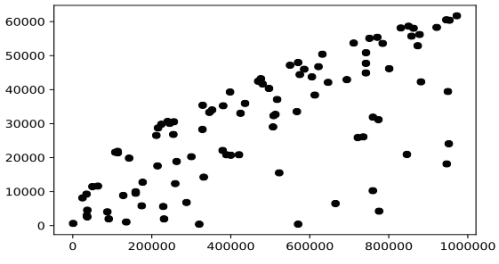
(e) Movies



(f) Electronics



(g) CDs



(h) Kindle

Figure 4.1: Point distribution for all data sets. For each data set, we select 100 points uniformly at random and plot them by their x-coordinates and y-coordinates.

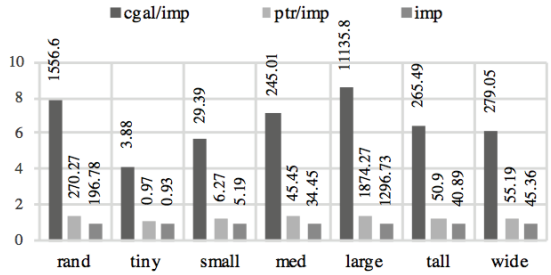
$min_{p\in\mathcal{P}}(c_2(p))$, $b_1 = max_{p\in\mathcal{P}}(c_1(p))$ and $b_2 = max_{p\in\mathcal{P}}(c_2(p))$. Then, we generated a uniform random point $p_r = (c_1(p_r), c_2(p_r))$ within this box. Each orthogonal range query in 2D is a rectangle. We use two extreme points to represent each range query: one is the point at the bottom left corner and the other one is the point at the upper right corner, see Figure 1.1(a). The 7 different range query types are defined as follows:

- **rand**: set the bottom left point to $(c_1(p_r), c_2(p_r))$ and generate the upper right point uniformly at random in the range $[c_1(p_r) : b_1] \times [c_2(p_r) : b_2]$.
- **tiny**: set the bottom left point to $(c_1(p_r), c_2(p_r))$ and generate the upper right point uniformly at random in the range $[c_1(p_r) : c_1(p_r) + \frac{(b_1 - c_1(p_r))}{S}] \times [c_2(p_r) : c_2(p_r) + \frac{(b_2 - c_2(p_r))}{S}]$, where $S = 50$.
- **small**: same as **tiny** with $S = 15$.
- **med**: same as **tiny** with $S = 5$.
- **large**: choose the bottom left point uniformly at random from the range $[a_1 : a_1 + \frac{(b_1 - a_1)}{3}] \times [a_2 : a_2 + \frac{(b_2 - a_2)}{3}]$ and the upper right point uniformly at random from the range $[b_1 - \frac{(b_1 - a_1)}{3} : b_1] \times [b_2 - \frac{(b_2 - a_2)}{3} : b_2]$.
- **tall**: set the bottom left point to $(c_1(p_r), c_2(p_r))$ and generate the upper right point uniformly at random from the range $[c_1(p_r) : c_1(p_r) + \frac{(b_1 - c_1(p_r))}{25}] \times [c_2(p_r) : b_2]$.
- **wide** set the bottom left point to $(c_1(p_r), c_2(p_r))$ and generate the upper right point uniformly at random from the range $[c_1(p_r) : b_1] \times [c_2(p_r) : c_2(p_r) + \frac{(b_2 - c_2(p_r))}{25}]$.

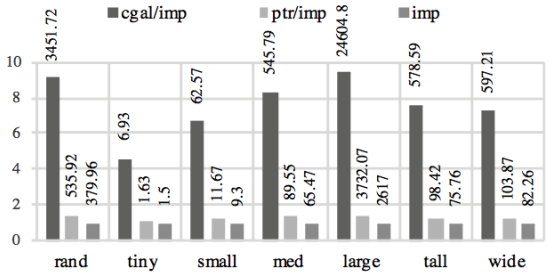
4.1.2 K-d Tree Comparison

We implemented both the pointer-based k-d tree and the implicit k-d tree, and compared them with the k-d tree implementation in CGAL [34]. Because the CGAL implementation does not support range counting, we only considered the range reporting time for the comparison including the k-d tree implementation in CGAL.

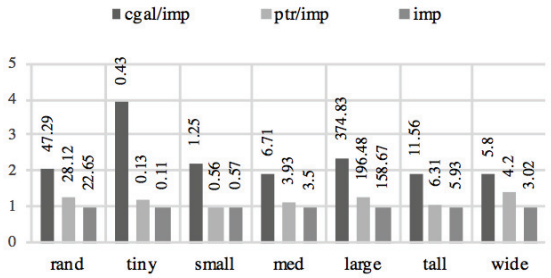
Figure 4.2 shows a comparison of the range reporting time of these three k-d tree implementations across all the data sets and query types. **imp**, **ptr** and **cgal** refer to the implicit k-d tree, the pointer-based k-d tree, and the k-d tree implementation in CGAL, respectively. Each bar in each data set represents the ratio of the range



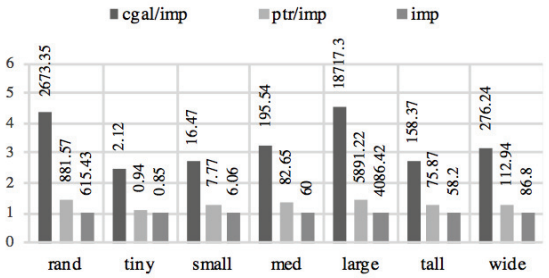
(a) Rand_1M



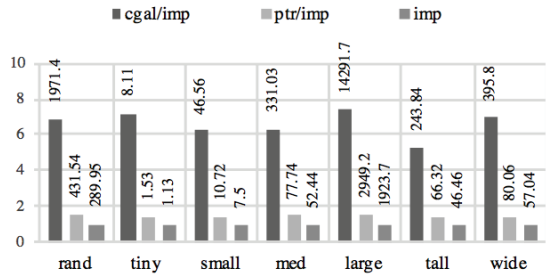
(b) Rand_2M



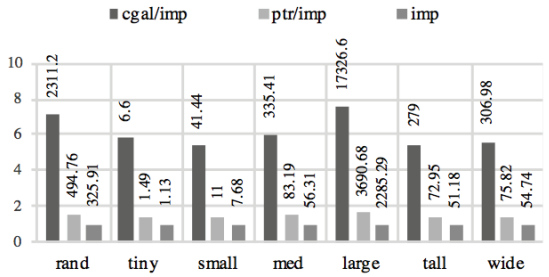
(c) China



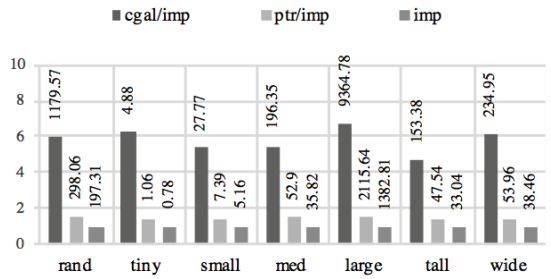
(d) World



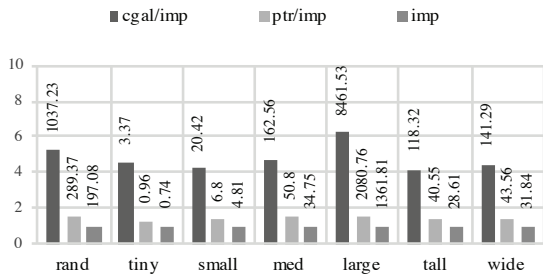
(e) Movies



(f) Electronics



(g) CDs



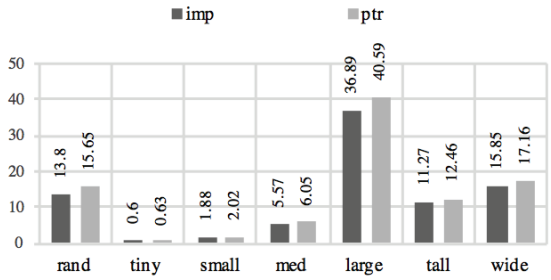
(h) Kindle

Figure 4.2: Comparison of total range reporting time (in seconds) of 3 different k-d tree implementations using 7 different query types with 10,000 queries of each type and across 8 different data sets. The numbers above bars **cgal/imp**, **ptr/imp** and **imp** represent the range reporting time of **cgal**, **ptr** and **imp**, respectively.

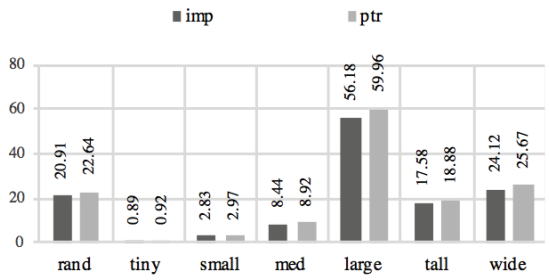
reporting time of two data structures. The bar with **cgal/imp** is the ratio of the range reporting time of the k-d tree implementation in CGAL and the implicit k-d tree. The number above the bar indicates the range reporting time of the k-d tree implementation in CGAL in seconds. The bar with **ptr/imp** is the ratio of the range reporting time of the pointer-based k-d tree and the implicit k-d tree. The number above the bar indicates the range reporting time of the pointer-based k-d tree in seconds. For the bar **imp**, the ratio is always 1 since it compares the reporting time of **imp** to itself and the number above the bar represents the range reporting time for the implicit k-d tree.

As observed from Figure 4.2, **imp** always performs the best among the three, and **cgal** is the slowest. In most of the cases, **ptr** is roughly 1.5 times slower than **imp** and **cgal** is at least five times slower than **imp**. As a result, we can first eliminate the k-d tree implementation in CGAL from further experiments since it runs the slowest among the three. The reason why the k-d tree implementation in CGAL is much slower than our k-d tree implementations could be the "weight" of the library, since the CGAL implementation supports not only simple orthogonal range queries, but also some other operations such as point insertions, point deletions and nearest neighbour searches. In order to support more advanced operations, more complex structures are required, which could result in low range query efficiency. On the other hand, for our k-d tree implementations, we only need to build a static k-d tree that supports fast range queries. For our k-d tree implementations, the implicit k-d tree always outperforms the pointer-based k-d tree for all the data sets for the range reporting queries.

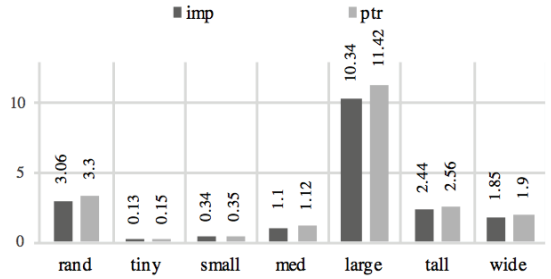
Because we eliminate the CGAL k-d tree implementation, we now can compare the range counting time of our k-d tree implementations. Figure 4.3 shows this range counting time comparison. Similarly to the range reporting time comparison, the range counting time of **imp** is still much faster than that of **ptr**. The reasons why the implicit k-d tree has better range query performance is that the implicit k-d tree is allocated in a contiguous memory block and the allocation of each subtree of a implicit k-d tree is also a single chunk of contiguous memory. This memory allocation is cache friendly, since it can prefetch the data that needs to be assessed in the future into the cache, thus increases the range query performance of the implicit



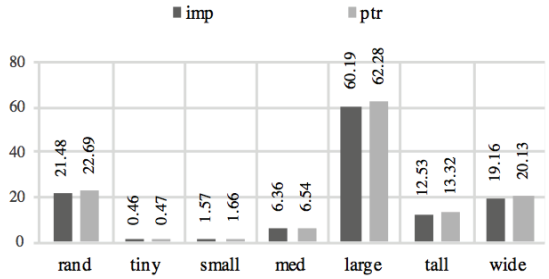
(a) Rand_1M



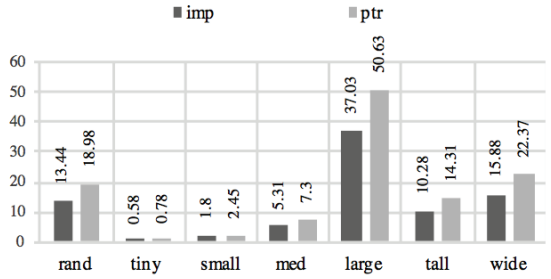
(b) Rand_2M



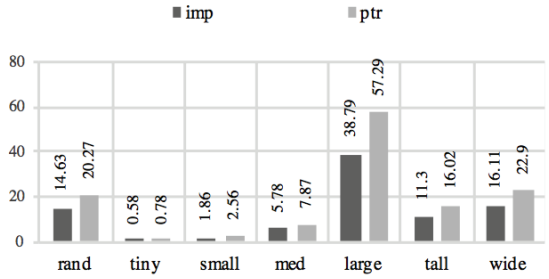
(c) China



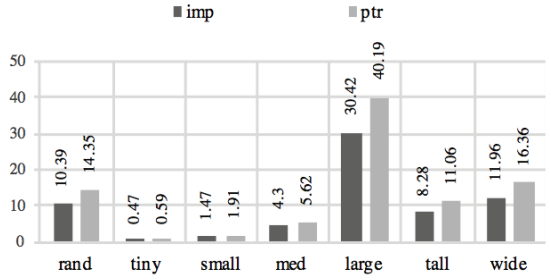
(d) World



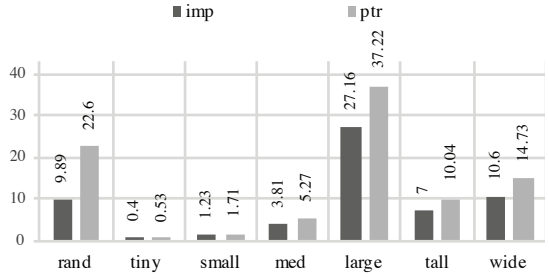
(e) Movies



(f) Electronics



(g) CDs



(h) Kindle

Figure 4.3: Comparison of total range counting time (in seconds) of 2 different k-d tree implementations using 7 different query types with 10,000 queries of each type and across 8 different data sets. The numbers above bars represent the corresponding range counting time of the two k-d tree implementations.

k-d tree. Moreover, the implicit k-d tree has zero structure overhead compared to the pointer-based k-d tree where each node needs to store two pointers and a traversal of the tree requires a pointer dereference, whereas an implicit k-d tree does not need to store any extra information and such a tree traversal can be done using index arithmetic, which can in fact be implemented using a bit shift. Based on the above experimental results, we will use the implicit k-d tree as a reference for our evaluation of the chain-decomposition-based range query algorithms.

4.1.3 Chain Decomposition

Data set	Number of points	Number of chains
Rand_1M	1,000,000	1256
Rand_2M	2,000,000	1780
China	71,009	319
World	1,904,711	1543
Movies	1,697,523	1233
Electronics	1,689,188	1079
CDs	1,097,592	1034
Kindle	982,617	613

Table 4.2: Chain decomposition results.

For our chain-decomposition-based range query algorithms, one of the factors that could affect the range query performance is the number of chains that are generated by the partitioning process. Table 4.2 shows the number of chains obtained from the chain partitioning process for all data sets. The results show that the number of chains varies depending on the point set, not just on its size. Even for roughly equal-sized point sets (**Rand_1M** and **Kindle**), the difference could be significant. Moreover, the real-world point sets generated fewer chains compared with the random point sets. These differences in the numbers of generated chains could have a significant impact on the range query time, which will be explained later.

4.1.4 Chain Untangling Comparison

In Chapter 3, we proposed an alternative algorithm for the 2-dimensional chain untangling process. Table 4.3 shows a running time comparison of our chain untangling algorithm against Claude et al.'s algorithm [14]. Our algorithm shows a 25% running time improvement on average against the original one. The reason for the

Data set	Time of original untangling method [14] (s)	Time of our untangling method (s)
Rand_1M	364.59	225.45
Rand_2M	1093.59	670.1
China	5.49	4.15
World	786.86	573.93
Movies	770.03	624.82
Electronics	701.05	568.66
CDs	380.17	303.4
Kindle	309.51	224.79

Table 4.3: Chain untangling time (in seconds) comparison for different data sets

improvement is that we eliminate the transformation process in each iteration from the original algorithm, see details in Section 3.2.

4.1.5 Range Query Comparison

In this section, the complete range query results for all data sets and all query types in 2D are presented. Three types of range query methods are considered. The first, **imp-kd**, is the implicit k-d tree, which was discussed in Section 4.1.2. As discussed in Section 4.1.2, the implicit k-d tree has the best range query performance and the minimal space requirements compared with other k-d tree variants. The other two are the two chain query methods based on untangled monotonic chains that were discussed in Section 3.4: for each chain that is found with a point lies in the query box after a binary search, the first one, **bi-bi**, applies two binary searches over the points along the chain to find the first and last points in the chain within the range and then reports all points in between; the second one, **bi-seq**, applies an iterative scanning method by scanning forward and backward from the point we found along the chain and finds all points in the chain within the range. Instead of scanning the points one by one, we find the optimal parameter d and inspect every dth point along the chain. Figure 4.4 shows the range counting results and Figure 4.5 shows the range reporting results.

We keep using a bar chart to represent the query time comparison of different range query methods of each data set, which is similar to the bar chart representations of the k-d tree comparison discussed in Section 4.1.2. Each bar in a bar chart represents the ratio of the range query time (range counting or reporting time) of two data

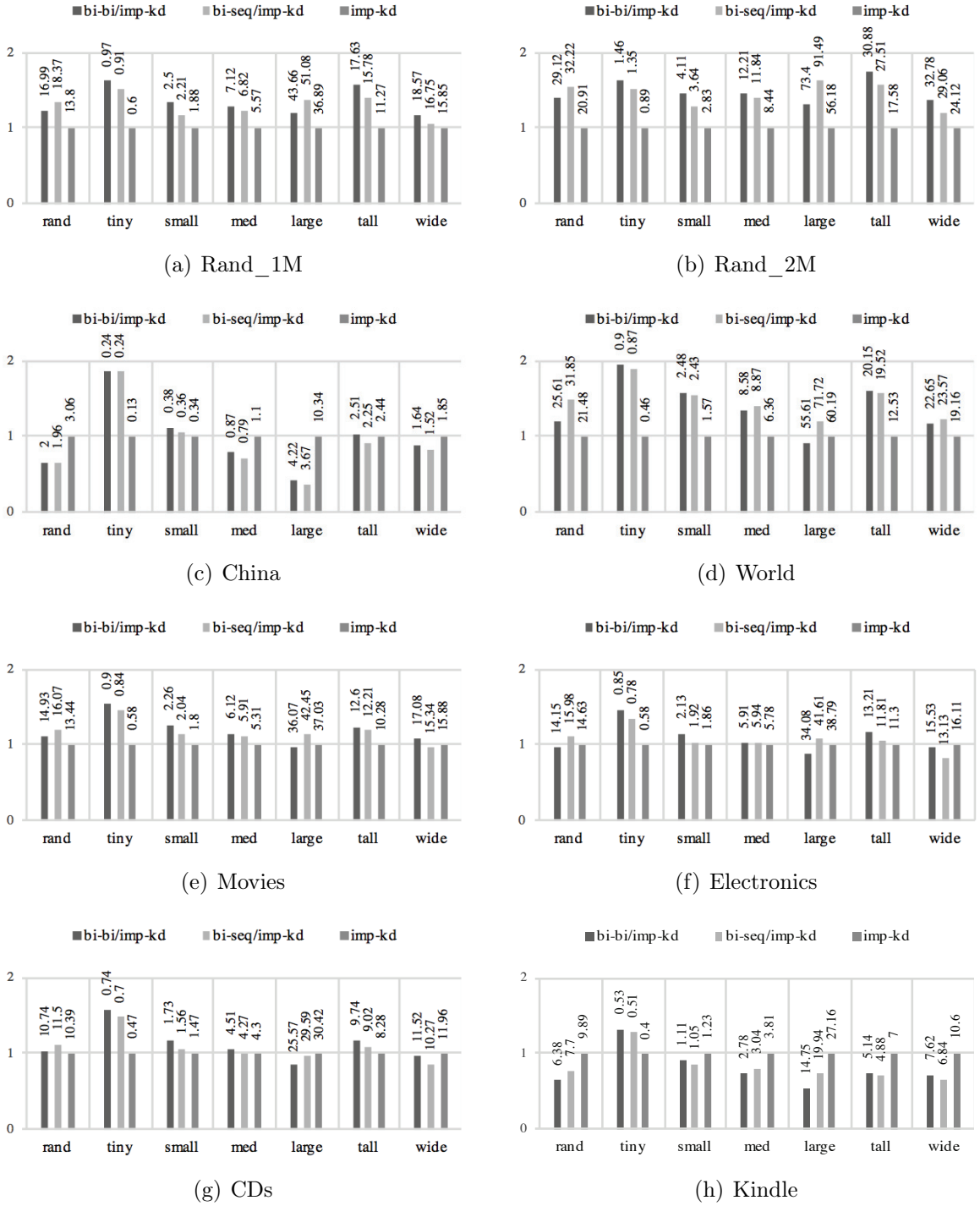
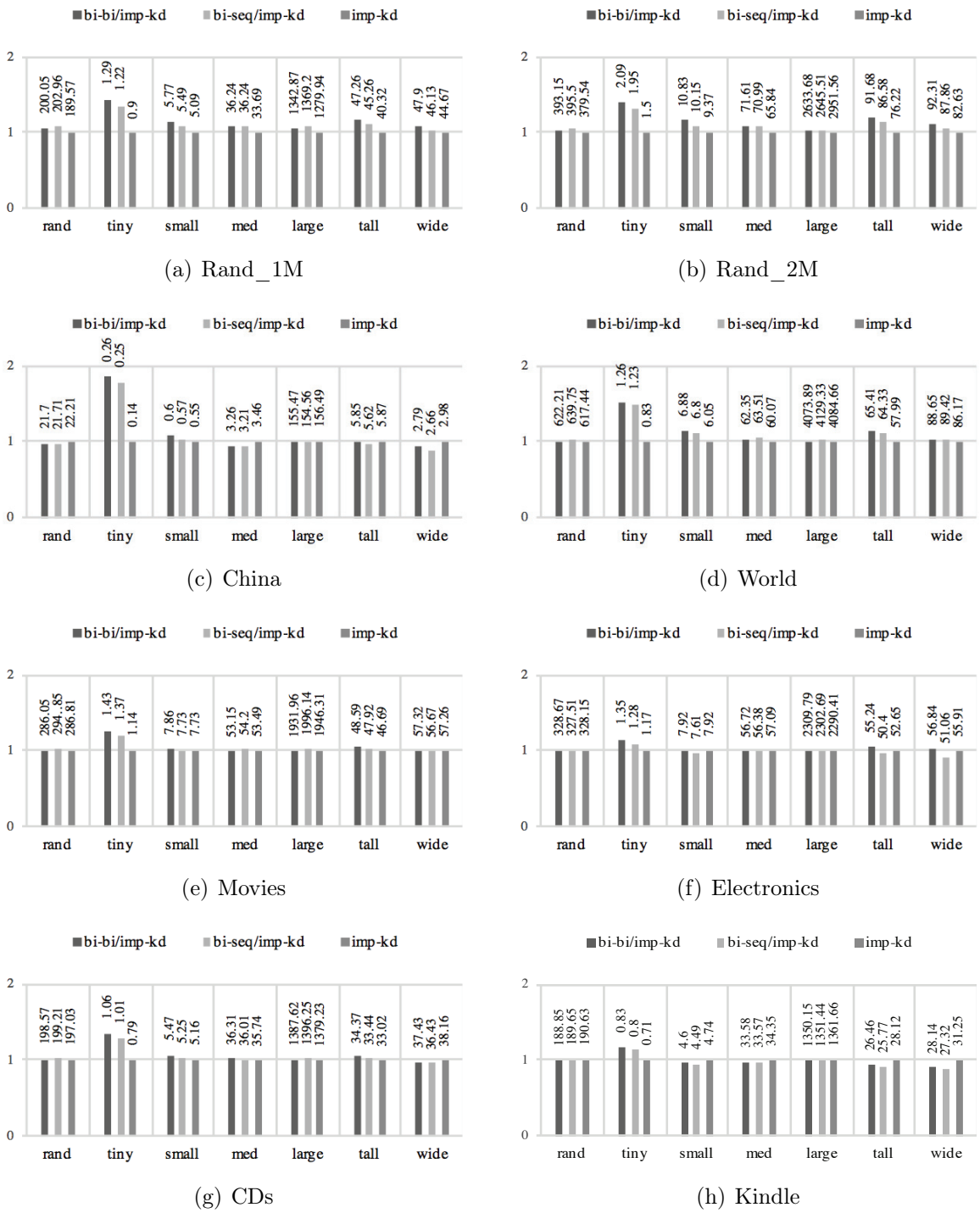


Figure 4.4: Comparison of total range counting time (in seconds) of 3 different range query methods using 7 different query types with 10,000 queries of each type and across 8 different data sets. The numbers above bars **bi-bi/imp**, **bi-seq/imp** and **imp** represent the range reporting time of **bi-bi**, **bi-seq** and **imp**, respectively.



(a) Rand_1M (b) Rand_2M

(c) China (d) World

(e) Movies (f) Electronics

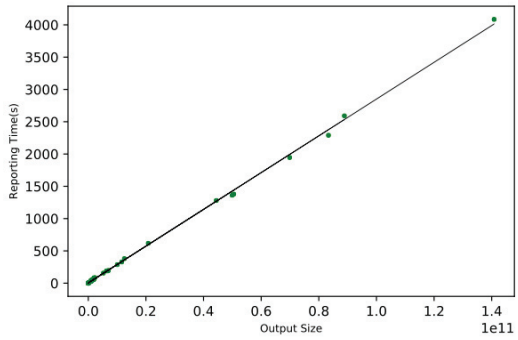
(g) CDs (h) Kindle

Figure 4.5: Comparison of total range reporting time (in seconds) of 3 different range query methods using 7 different query types with 10,000 queries of each type and across 8 different data sets. The numbers above bars **bi-bi/imp**, **bi-seq/imp** and **imp** represent the range reporting time of **bi-bi**, **bi-seq** and **imp**, respectively.

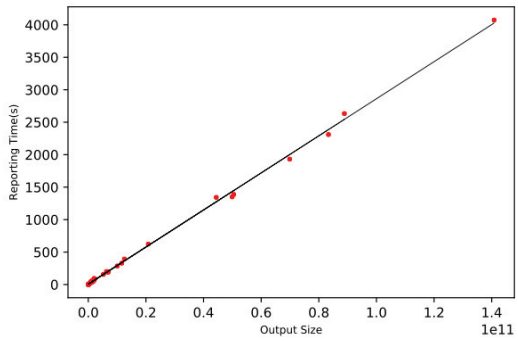
structures of a data set. The bar with **bi-bi/imp** is the ratio of the range query time of the two-binary-search chain query method and the implicit k-d tree query method. The number above the bar is the range query time for the two-binary-search chain query method. The bar with **bi-seq/imp** is the ratio of the range query time of the linear scanning chain query method and the implicit k-d tree query method. The number above the bar is the range query time for the linear scanning chain query method. For the **imp** bar, the ratio is always 1, since it compares the range query time of **imp** to itself and the number above the bar represents the range query time of the implicit k-d tree.

The range counting results from Figure 4.4 shows that **imp-kd** outperforms the chain-decomposition-based range query methods (**bi-bi** and **bi-seq**) for the random data sets (**Rand_1M** and **Rand_2M**) for all query types. The same observation applies to the data set **World**, expect the query type *large*. For query type *tiny*, **imp-kd** has better query performance compared to the other two query methods and the same observation holds for query type *small*, expect for the data set **Kindle**. For query type *large* on **China** and query types *large*, *tall* and *wide* on **Kindle**, the chain-decomposition-based range query methods are faster than the implicit k-d tree. The remaining query time differences where the chain-decomposition-based query methods outperform the implicit k-d tree are insignificant, since those differences have little impact on the overall range counting times, which makes the absolute impact small. In most cases, the implicit k-d tree can achieve better range counting performance compared with the chain-decomposition-based range query methods, but in some cases (for certain types of queries of some data sets), the chain-decomposition-based query methods still have the advantage over the implicit k-d tree.

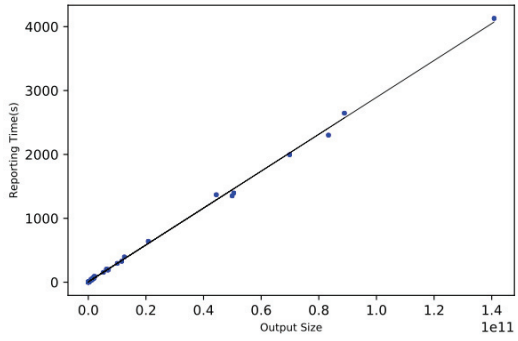
As for the range reporting results shown Figure 4.5, except for the query type *tiny*, all the three methods are competitive. This is because the range reporting time is dominated by the output size (except for the query type *tiny*) and the search cost has little impact on the overall query time. Figure 4.6 shows the graphical representation of the relationship between the range reporting time and output size. We calculated a linear fitting function for each range query method using linear regression algorithm [31] and plotted it on a graph. We use the variance score [31] to measure the fitting accuracy of each line and the best possible variance score is 1.0.



(a) **imp-kd** - the linear fitting function is $y = 2.85 \times 10^{-8}x + 3.93$ with the variance score 1.0.



(b) **bi-bi** - the linear fitting function is $y = 2.86 \times 10^{-8}x + 6.15$ with the variance score 1.0.



(c) **bi-seq** - the linear fitting function is $y = 2.89 \times 10^{-8}x + 5.32$ with the variance score 1.0.

Figure 4.6: The relationship of range reporting time and output size for three range query methods for all data sets with the range reporting time on the Y axis and the output size on the X axis. The coloured trend-line shows a linear relationship of the two parameters for different range query algorithms.

Query type	Count (s)	Report (s)	Nodes visited	Nodes from subtree	Node ratio	Points from subtree	Total Points found	Points ratio
rand	13.8	196.78	420,619,247	46,664,010	11.09%	6,156,553,648	6,278,307,009	98.06%
tiny	0.6	0.93	16,512,457	1,012,931	6.13%	6,311,348	10,016,115	63.01%
small	1.88	5.19	55,270,339	5,146,022	9.31%	96,628,252	111,517,447	86.65%
med	5.57	34.45	167,865,573	17,926,068	10.68%	952,420,893	1,000,250,076	95.22%
large	36.89	1296.73	1,134,038,637	131,714,904	11.61%	44,094,543,930	44,427,772,735	99.25%
tall	11.27	40.89	357,906,427	43,795,402	12.24%	900,822,975	1,002,083,067	89.90%
wide	15.85	45.36	486,342,731	42,472,018	8.73%	857,812,894	998,722,545	85.89%

Count(s): range counting time in seconds.
Report(s): range reporting time in seconds.
Nodes visited: the total number of nodes visited during the tree traversals.
Nodes from subtree: the number of visited nodes where, for each node, all the points stored in the subtree rooted at this node are reported.
Node ratio: the ratio of the number of visited nodes that return their subtrees to the total number of visited nodes.
Points from subtree: the total number of points reported by the subtrees.
Total Points found: the total number of points found during the search.
Points ratio: the ratio of the total number of points returned by subtrees to the total number of points found within the query range.

Table 4.4: Rand_1M-**imp-kd**.

Query type	Count (s)	Report (s)	Chains found	Binary search steps	Binary search_I	Binary search_II	Total Points found
rand	16.99	200.05	45,805,333	761,026,714	142,018,135	619,008,579	6,278,307,009
tiny	0.97	1.29	2,182,485	38,962,272	27,616,828	11,345,444	10,016,115
small	2.5	5.77	6,846,859	105,794,775	46,322,584	59,472,191	111,517,447
med	7.12	36.24	19,234,594	309,361,627	80,567,201	228,794,426	1,000,250,076
large	43.66	1342.87	111,827,535	2,030,179,272	149,210,418	1,880,968,854	44,427,772,735
tall	17.63	47.26	62,129,417	857,755,905	350,100,222	507,655,683	1,002,083,067
wide	18.57	47.9	62,266,616	860,039,632	352,088,859	507,950,773	998,722,545

Chains found: the total number of chains that intersect the query boxes.
Binary search steps: the total number of binary search steps performed during the range query.
Binary search_I: the number of binary search steps that are performed to find out whether there exists a chain that intersect the query box.
Binary search_II: the number of binary search steps performed on each intersecting chain to find all the points on the chain that lie in the range.

Table 4.5: Rand_1M-**bi-bi**

Query type	Count (s)	Report (s)	Chains found	Binary search steps	Best distance	Linear scan	Total Points found
rand	18.37	202.96	45,805,333	142,018,135	20	1,257,251,848	6,278,307,009
tiny	0.91	1.22	2,182,485	27,616,828	2	6,385,658	10,016,115
small	2.21	5.49	6,846,859	46,322,584	2	61,695,261	111,517,447
med	6.82	36.24	19,234,594	80,567,201	10	292,234,210	1,000,250,076
large	51.08	1369.2	111,827,535	149,210,418	40	5,616,605,101	44,427,772,735
tall	15.78	45.26	62,129,417	350,100,222	4	443,242,068	1,002,083,067
wide	16.75	46.13	62,266,616	352,088,859	4	442,501,439	998,722,545

Best distance: the optimal checking distance.
Linear scan: the total number of checking steps performed during the range query.

Table 4.6: Rand_1M-**bi-seq**

Query type	Count (s)	Report (s)	Nodes visited	Nodes from subtree	Node ratio	Points from subtree	Total Points found	Points ratio
rand	9.89	190.63	415,344,775	48,937,486	11.78%	6,700,494,755	6,824,096,611	98.19%
tiny	0.4	0.71	15,115,501	965,276	6.39%	8,820,949	11,905,182	74.09%
small	1.23	4.74	49,926,351	4,638,596	9.29%	119,280,338	131,716,602	90.56%
med	3.81	34.35	156,438,210	16,938,997	10.83%	1,111,216,962	1,153,736,275	96.31%
large	27.16	1361.66	1,218,668,567	173,425,385	14.23%	49,482,520,863	49,921,352,711	99.12%
tall	7	28.12	299,871,046	34,242,595	11.42%	712,362,979	788,855,713	90.30 %
wide	10.6	31.25	449,489,719	37,192,048	8.27%	642,244,496	763,206,892	84.15 %

Columns are the same as on Table 4.4.

Table 4.7: Kindle-**imp-kd**.

Query type	Count (s)	Report (s)	Chains found	Binary search steps	Binary search_I	Binary search_II	Total Points found
rand	6.38	188.85	21,345,319	375,010,605	74,632,343	300,378,262	6,824,096,611
tiny	0.53	0.83	1,133,753	25,193,764	18,406,645	6,787,119	11,905,182
small	1.11	4.6	3,589,764	60,570,835	27,834,381	32,736,454	131,716,602
med	2.78	33.58	9,817,874	165,179,820	44,614,205	120,565,615	1,153,736,275
large	14.75	1350.15	53,116,059	988,183,093	71,396,959	916,786,134	49,921,352,711
tall	5.14	26.46	20,755,587	316,567,613	130,695,668	185,871,945	788,855,713
wide	7.62	28.14	35,899,499	475,006,650	233,580,933	241,425,717	763,206,892

Columns are the same as on Table 4.5.

Table 4.8: Kindle-**bi-bi**

Query type	Count (s)	Report (s)	Chains found	Binary search steps	Best distance	Linear scan	Total Points found
rand	7.7	189.65	21,345,319	74,632,343	21	764,978,409	6,824,096,611
tiny	0.51	0.8	1,133,753	18,406,645	1	10,997,480	11,905,182
small	1.05	4.49	3,589,764	27,834,381	3	50,874,237	131,716,602
med	3.04	33.57	9,817,874	44,614,205	10	207,300,417	1,153,736,275
large	19.94	1351.44	53,116,059	71,396,959	51	3,558,329,675	49,921,352,711
tall	4.88	25.77	20,755,587	130,695,668	5	238,640,294	788,855,713
wide	6.84	27.32	35,899,499	233,580,933	5	260,675,693	763,206,892

Columns are the same as on Table 4.6.

Table 4.9: Kindle-**bi-seq**

Even though the search cost does not significantly affect the overall range reporting time, the differences among the range reporting methods that do exist are due to the search cost, which leads us to shift our attentions back to the range counting time. In order to find out under what circumstances the chain-decomposition-based range searching methods can achieve better range query performance, we chose two representative data sets **Rand_1M** and **Kindle** to analyze in detail. Both of them are of roughly equal size, but the range query results are quite different.

Tables 4.4, 4.5 and 4.6 show the range query analyses of **imp-kd**, **bi-bi** and **bi-seq** for the data set **Rand_1M**, and Tables 4.7, 4.8 and 4.9 show the range query analyses of **imp-kd**, **bi-bi** and **bi-seq** for the data set **Kindle**. For the analyses of the implicit k-d tree (**imp-kd**), we use the number of nodes accessed during the range

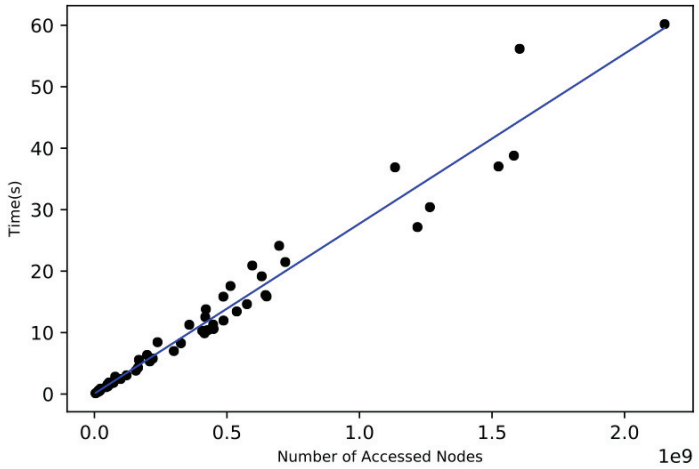


Figure 4.7: The relationship of range counting time and the number of accessed nodes for the implicit k-d tree for all data sets with the range counting time on the Y axis and the number of accessed nodes on the X axis. The blue trend-line shows a linear relationship of the two parameters using the function $y = 2.77 \times 10^{-8}x + 0.067$ that is calculated based on linear regression algorithm [31].

queries to evaluate its range query performance, since it is the most basic step of the k-d tree range query algorithm. Among those node accesses, we also count the number of accessed nodes where, for each node, all points stored in the subtree rooted at this node are reported. One observation from the implicit k-d tree analyses for the data sets **Rand_1M** and **Kindle** in Tables 4.4 and 4.7 is that the range counting time increases with the increase of the number of accessed nodes, which means these two could form a linear relationship. We calculate a linear function using linear regression algorithm [31] and show the graphical representation of the linear relationship of the two for all data sets in Figure 4.7. The variance score of the linear function is 0.96. Another observation is that the differences of the number of accessed nodes for each query type between the two point sets are less than 10%. For both data sets, the number of nodes that report all the points stored in their entire subtrees only occupies a small portion of the total number of accessed nodes, but they contribute the most points within the range. Overall, the range query performance of **imp-kd** is similar for both **Rand_1M** and **Kindle** and the behaviors of the query procedure on both data sets are similar as well.

Tables 4.5 and 4.6 show the analyses of **bi-bi** and **bi-seq** for data set **Rand_1M**

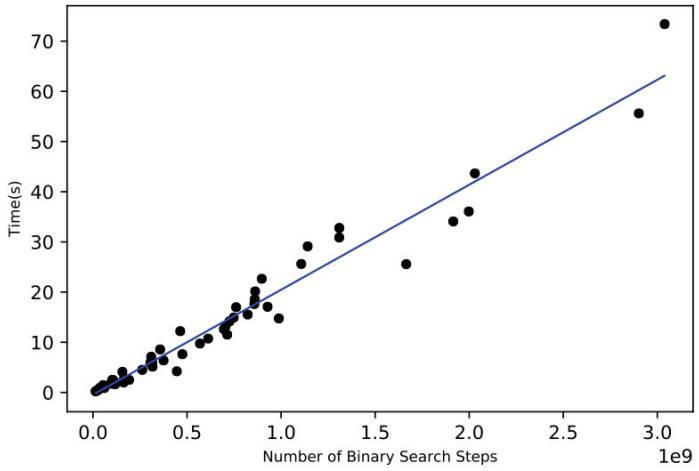
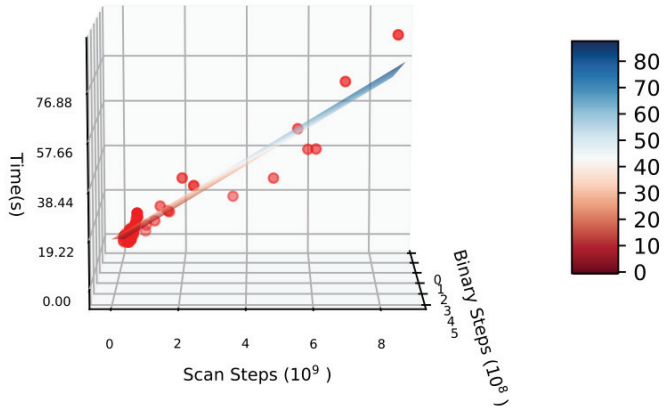


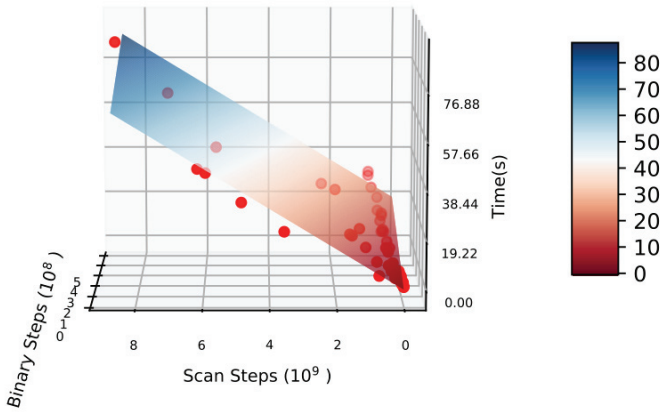
Figure 4.8: The relationship of range counting time and the number of binary search steps for **bi-bi** for all data sets with the range counting time on the Y axis and the number of binary search steps on the X axis. The blue trend-line shows a linear relationship of the two parameters using the function $y = 2.09 \times 10^{-8}x - 0.45$.

and Tables 4.8 and 4.9 give the analyses of **bi-bi** and **bi-seq** for the data set **Kindle**. In Tables 4.5 and 4.8 for **bi-bi**, we use the number of binary search steps to evaluate the range query performance, since the binary search step is the fundamental step of the range query method **bi-bi**. In tables 4.6 and 4.9 for **bi-seq**, instead of the basic binary search steps, we also count linear scanning steps, since for the range query method **bi-seq**, these two steps together constitute the basic query steps for **bi-seq**. As we observe from the above analyses, for both **Rand_1M** and **Kindle**, the range counting time of **bi-bi** is in a linear relationship with the number of binary search steps, and the range counting time of **bi-seq** is in a linear relationship with a combination of the number of binary search steps and the total number of linear scanning steps. Figure 4.8 shows the graphical representation of the linear relationship of the query time and number of binary search steps for **bi-bi**. Figure 4.9 shows the graphical representation of the linear relationship of the query time and the combination of binary search steps and linear scanning steps for **bi-seq**. The variance scores of the linear functions for **bi-bi** and **bi-seq** are 0.96 and 0.94 respectively.

Another observation from the comparison of the chain-decomposition-based range query methods between data sets **Rand_1M** and **Kindle** is that, for range query method **bi-bi**, the number of chains crossing the queries is in a linear relationship with



(a) View angle 1



(b) View angle 2

Figure 4.9: The relationship of range counting time, the number of binary search steps and the number linear scanning steps for all data sets with the range counting time on the Z axis, the number of binary search steps on the X axis and the number linear scanning steps on the Y axis. The coloured surface is drawn using the function $z = 3.99 \times 10^{-8}x + 7.99 \times 10^{-9}y - 1.24$ that is calculated based on linear regression algorithm [31].

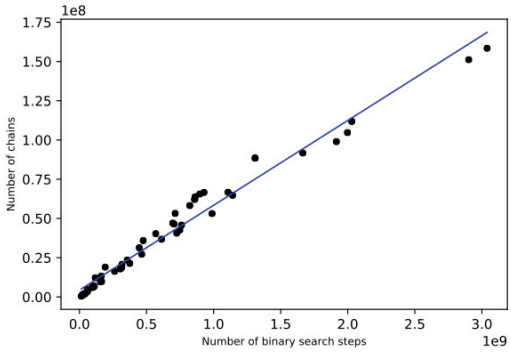
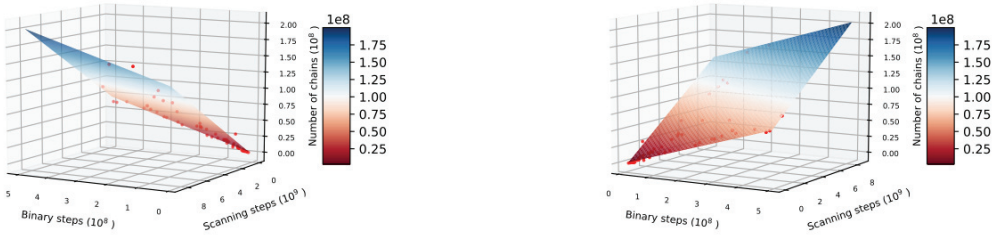


Figure 4.10: The relationship of the number of binary search steps and the number of crossing chains for **bi-bi** for all data sets with the number of crossing chains on the Y axis and the number of binary search steps on the X axis. The blue trend-line shows a linear relationship of the two parameters using the function $y = 0.05x + 4365210.01$. The variance score is 0.97.



(a) View angle 1

(b) View angle 2

Figure 4.11: The relationship of the number of crossing chains, the number of binary search steps and the number of scanning steps for **bi-seq** for all data sets with the number of crossing chains on the Z axis, the number of binary search steps on the X axis and the number scanning steps on the Y axis. The coloured surface is drawn using the function $z = 0.14x + 0.02y - 107900.31$. The variance score is 0.98.

the total number of binary search steps, see Figure 4.10; for the range query method **bi-seq**, the number of chains is in a linear relationship with a combination of the number of binary search steps and number of linear scanning steps, see Figure 4.11. Because the range counting time is also in a linear relationship with these basic steps, there must exist a linear relationship between the range query time and the number of chains crossing the quires. In Tables 4.5 and 4.6, we observe the number of chains found in **Kindle** is roughly half of the number of chains found in **Rand_1M** for

all query types. We conjecture that the differences in the number of crossing chains among different data sets can be further explained by the differences in the number of chains generated during the chain partition steps. From Section 4.1.3, we know the total number of chains generated for the **Kindle** data set is 612, which is half of 1256 chains from **Rand_1M**. For the data set **China**, the number of chains generated during the partitioning step is only 319. The range query performance of the chain-decomposition-based range query methods for the data sets **Kindle** and **China** is as competitive as the performance of the implicit k-d tree and for range query *large*, the chain-decomposition-based range query methods run two times faster than the implicit k-d tree. This fewer number of chains generated during the partition can provide one explanation for the few number of chains crossing the queries and for the better range query performance.

Based on the above analysis, we can come to the conclusion that the range query performance of the implicit k-d tree (**imp-kd**) is in a linear relationship with its number of accessed nodes during the range quries and its query performance is not very dependent on the types of data sets. On the other hand, the range query performance of the chain-decomposition-based range searching methods (both **bi-bi** and **bi-seq**) are largely affected by different point sets, since their basic steps (binary search steps or linear scanning steps) are largely affected by the number of chains that intersect the query boxes. With fewer chains generated from the partition step, the chain-decomposition-based range searching methods tend to have better range query performance.

Because the data structures implemented for the chain-decomposition-based range searching methods are different from the range query method of k-d tree, it is not easy to compare these two structures with a quantifiable measurement. Next, we investigate whether the observed correlation between the range query time and number of elementary steps taken by the these algorithms can be linked to certain hardware parameters, such as the number of cache-misses incurred by a search. Tables 4.10 and 4.11 show the profiling results for the data sets **Rand_1M** and **Kindle** with range query type *rand*. As we observe, the chain-decomposition-based range query algorithms (**bi-bi** and **bi-seq**) outperform the k-d tree in terms of task-clock for the data set **Kindle**, while for the data set **Rand_1M**, the situation is reversed. The

Performance	**imp-kd**		**bi-bi**		**bi-seq**	
task-clock(msec)	15061.39		17864.62		19110.14	
cycles	36,063,961,665		42,757,857,503		45,704,887,000	
context-switches	4184	0.28 K/s	4797	0.27 K/s	5132	0.27 K/s
page-faults	4744	0.32 K/s	5345	0.3 K/s	5346	0.28 K/s
instruction	31,478,826,938	0.87 insns/cycle	20,853,067,298	0.49 insns/cycle	20,343,758,327	0.45 insns/cycle
branches	5,145,718,316	341.65 M/s	3,454,336,260	193.36 M/s	4,473,999,895	234.12 M/s
branch-misses	223,391,369	4.34%	393,508,940	11.39%	240,321,349	5.37%
L1-dcache-loads	14,733,672,800	978.24 M/s	5,115,127,203	286.33 M/s	6,220,254,021	325.5 M/s
L1-dcache-load-misses	111,820,094	0.76%	740,280,904	14.47%	792,194,599	12.74%
LLC-loads	170,545,634	11.32 M/s	860,509,980	48.17 M/s	1,080,021,328	56.52 M/s
LLC-load-misses	102,928,316	60.35%	708,821,542	82.37%	772,596,998	71.54%

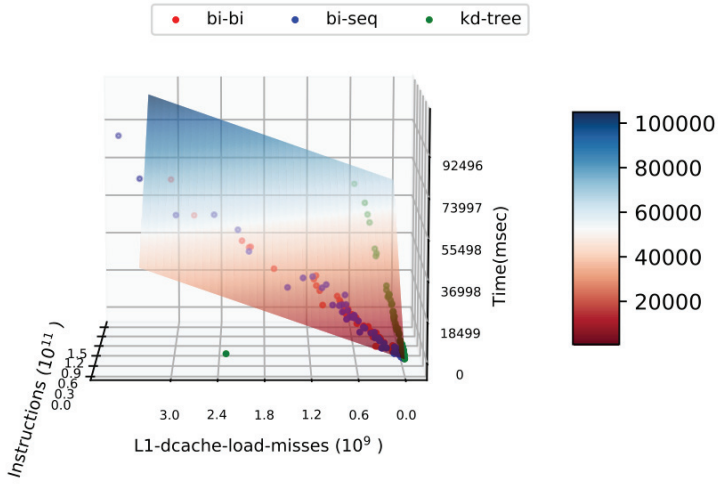
Note. K/s indicates kilobyte per seconds. M/s represents megabyte per seconds. insns/cycle indicates number of instructions per cycle. The L1-dcache-loads and L1-dcache-load-misses represent the number of Level-1 data cache loads and the number of Level-1 data cache-misses. The LLC-loads and LLC-load-misses represent the last level (Level-3) cache loads and cache-misses. The percentage shown in branch-misses is the percentage of the number of branch-misses out of the total number of branches. The percentage shown in L1-dcache-load-misses is the percentage of the number of L1-dcache-load-misses out of the total number of L1-dcache-loads. The percentage of LLC-loads-misses is the percentage of the number of LLC-load-misses out of the total number of LLC-loads.

Table 4.10: Hardware performance analysis of data set **Rand_1M** for query type *rand*

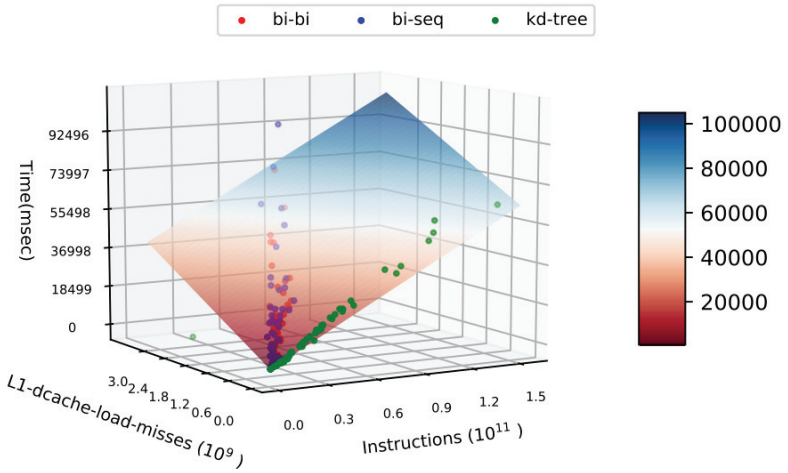
Performance	**imp-kd**		**bi-bi**		**bi-seq**	
task-clock(msec)	10476.52		6719.89		7811.51	
cycles	25,089,042,312		16,033,750,647		18,666,028,018	
context-switches	2911	0.28 K/s	2114	0.32 K/s	2069	0.27 K/s
page-faults	2362	0.23 K/s	2957	0.44 K/s	2957	0.38 K/s
instruction	30,268,677,019	1.21 insns/cycle	10,191,458,394	0.64 insns/cycle	11,588,107,703	0.62 insns/cycle
branches	4,315,793,720	411.95 M/s	2,039,843,540	303.55 M/s	2,848,846,802	364.7 M/s
branch-misses	194,278,257	4.50%	239,456,140	11.74%	149,108,534	5.23%
L1-dcache-loads	12,495,969,165	1192.76 M/s	2,760,828,356	410.84 M/s	3,736,924,600	478.39 M/s
L1-dcache-load-misses	77,511,017	0.62%	336,262,618	12.18%	551,366,977	14.75%
LLC-loads	110,965,547	10.59 M/s	392,342,784	58.39 M/s	610,404,212	78.14 M/s
LLC-load-misses	56,866,745	51.25%	269,107,528	68.59%	510,183,029	83.58%

Note. The columns are the same as on Table 4.10.

Table 4.11: Hardware performance analysis of data set **Kindle** for query type *rand*



(a) View angle 1



(b) View angle 2

Figure 4.12: The relationship of task-clock(msec), the number of instructions and the number L1-dcache-load-misses for all data sets with the task-clock(msec) on the Z axis, the number of instructions on the X axis and the number L1-dcache-load-misses on the Y axis. The coloured surface is drawn using the function $z = 4.14 \times 10^{-7}x + 1.22 \times 10^{-5}y - 331.22$ that is calculated based on linear regression algorithm [31].

chain-decomposition-based range query methods performs roughly 30% instructions of the k-d tree on **Kindle**, while for **Rand_1M**, the percentage increases to 66%. Even though the chain-decomposition-based range query algorithms requires fewer number of instructions, the number of instructions per cycle for the chain-decomposition-based range query algorithms is much lower than the k-d tree. In our experiment, we found nearly half of the cycles are wasted on retrieving the data from the memory which may due to the cache-misses and can be reflected from the high percentage rate of L1-dcache-load-misses for the chain-decomposition-based range query algorithms compared to the k-d tree (nearly 20 times higher).

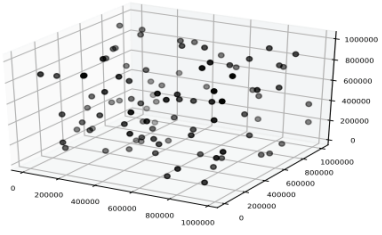
Based on the above analysis, the hardware performance in term of task-clock (equivalent to the range query time) is mainly dependent on the number of instructions and number of L1-dcache-load-misses. We may conjecture that the task-clock is a linear relationship with a combination of instructions and L1-dcache-load-misses. Figure 4.12 shows the graphical representation of this linear relationship and the variance score is 0.99.

To sum up, in 2D, the range query performance of the chain-decomposition-based range searching methods (**bi-bi** and **bi-seq**) is largely dependent on the point set itself. In most of the cases, the implicit k-d tree data structure (**imp-kd**) can still outperform the chain-decomposition-based range searching methods. We attached all the experimental results in Appendix A for reference.

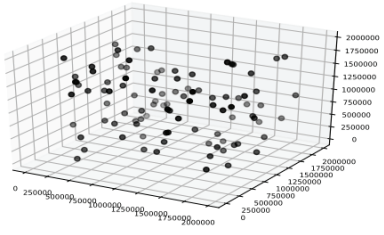
4.2 3D Experimental Evaluation

4.2.1 Data Sets and Range Queries

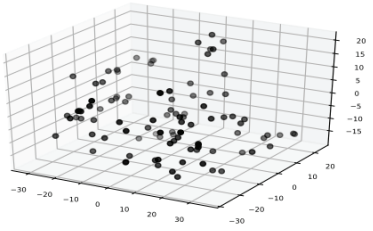
Similarly to the 2D experimental evaluation, we introduce three different types of data to achieve a comprehensive evaluation in 3-dimensional space. The first type is the 3D random data. We generated each uniform random n-point set by setting each coordinate sequence (x,y, and z-coordinate sequence) to a uniform random permutation of $\{1, 2, ..., n\} \subseteq \mathbb{Z}^+$. The second data type is from "A Benchmark for Surface Reconstruction" in the Computer Science Department of University of Utah [29]. The data sets were generated by synthetically scanning the surfaces of different shapes



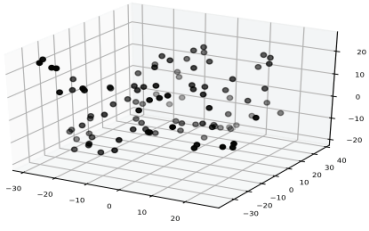
(a) Rand_1M



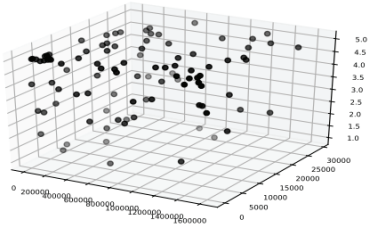
(b) Rand_2M



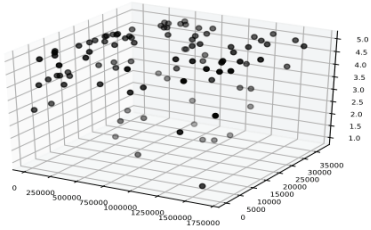
(c) DC



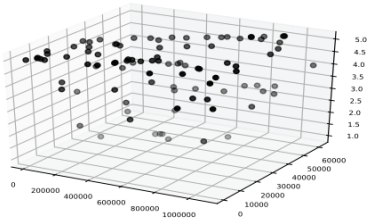
(d) Garalgly



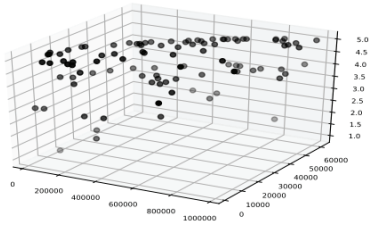
(e) Movies



(f) Electronics



(g) CDs



(h) Kindle

Figure 4.13: Point distribution for all data sets. For each data set, we select 100 points uniformly at random and plot them by their x, y and z-coordinates.

Data type	Data set	Number of points
Random data	Rand_1M	1,000,000
	Rand_2M	2,000,000
3D Object data	DC	468,020
	Garalgly	481,351
Amazon review data	Movies	1,697,523
	Electronics	1,689,188
	CDs	1,097,592
	Kindle	982,617

Table 4.12: Data set sizes.

of 3D objects. The third type, similarly to the 2D case, was acquired from Amazon review data set [23]. In addition to the product ID (**asin**) and the review time (**ReviewTime**), we added the third coordinate "overall rating" (**overall**) where the rating value is an integer ranging from 1 to 5. To avoid having multiple identical points, we kept only one of each group of points with identical x, y and z-coordinate. Table 4.12 shows the size of each data set and Figure 4.13 shows the point distribution of a small random samples (100 random points) of each data set.

8 different types of range queries were used for our 3D evaluation. Similarly to the range query generation in 2D, for each point set $\mathcal{P}$, we first identify its bounding box as $[a_1 : b_1] \times [a_2 : b_2] \times [a_3 : b_3]$, where $a_1 = min_{p \in \mathcal{P}}(c_1(p))$, $a_2 = min_{p \in \mathcal{P}}(c_2(p))$, $a_3 = min_{p \in \mathcal{P}}(c_3(p))$ and $b_1 = max_{p \in \mathcal{P}}(c_1(p))$, $b_2 = max_{p \in \mathcal{P}}(c_2(p))$, $b_3 = max_{p \in \mathcal{P}}(c_3(p))$. Then, let point $p_r = (c_1(p_r), c_2(p_r), c_3(p_r))$ be a random point uniformly generated within the bounding box. Each range query in 3D can be viewed as a rectangular cube defined by two extreme points: one is the point from front bottom left corner and the other one is the point from back upper right corner, see Figure 1.1(b). Then, 8 types of range queries are defined as follows:

- **rand**: set the front bottom left point to $(c_1(p_r), c_2(p_r), c_3(p_r))$ and generate the back upper right point uniformly at random in the range $[c_1(p_r) : b_1] \times [c_2(p_r) : b_2] \times [c_3(p_r) : b_3]$.
- **tiny**: set the front bottom left point to $(c_1(p_r), c_2(p_r), c_3(p_r))$ and generate the back upper right point uniformly at random in the range $[c_1(p_r) : c_1(p_r) + \frac{(b_1 - c_1(p_r))}{S}] \times [c_2(p_r) : c_2(p_r) + \frac{(b_2 - c_2(p_r))}{S}] \times [c_3(p_r) : c_3(p_r) + \frac{(b_3 - c_3(p_r))}{S}]$, where $S = 10$
- **small**: same as **tiny** with $S = 5$.

- **med**: same as **tiny** with $S = 2$.
- **large**: choose the front bottom left point uniformly at random from the range $[a_1 : a_1 + \frac{(b_1-a_1)}{4}] \times [a_2 : a_2 + \frac{(b_2-a_2)}{4}] \times [a_3 : a_3 + \frac{(b_3-a_3)}{4}]$ and the back upper right point uniformly at random the from $[b_1 - \frac{(b_1-a_1)}{4} : b_1] \times [b_2 - \frac{(b_2-a_2)}{4} : b_2] \times [b_3 - \frac{(b_3-a_3)}{4} : b_3]$.
- **long**: set the front bottom left point to $(c_1(p_r), c_2(p_r), c_3(p_r))$ and generate the back upper right point uniformly at random from the range $[c_1(p_r) : c_1(p_r) + \frac{(b_1-c_1(p_r))}{4}] \times [c_2(p_r) : c_2(p_r) + \frac{(b_2-c_2(p_r))}{4}] \times [c_3(p_r) : b_3]$.
- **tall**: set the front bottom left point to $(c_1(p_r), c_2(p_r), c_3(p_r))$ and generate the back upper right point uniformly at random from the range $[c_1(p_r) : c_1(p_r) + \frac{(b_1-c_1(p_r))}{4}] \times [c_2(p_r) : b_2] \times [c_3(p_r) : c_3(p_r) + \frac{(b_3-c_3(p_r))}{4}]$.
- **wide**: set the front bottom left point to $(c_1(p_r), c_2(p_r), c_3(p_r))$ and generate the back upper right point uniformly at random from the range $[c_1(p_r) : b_1] \times [c_2(p_r) : c_2(p_r) + \frac{(b_2-c_2(p_r))}{4}] \times [c_3(p_r) : c_3(p_r) + \frac{(b_3-c_3(p_r))}{4}]$.

4.2.2 Chain Decomposition

Data set	**utga**			**lga**	
	Construction time(s)	Number of chains in the xy-plane	Number of chains in the xz-plane	Construction time(s)	Number of chains
Rand_1M	327.64	1256	42,691	493341.13	9839
Rand_2M	968.54	1780	72,060	-	-
DC	89.26	872	15,246	39006.29	3127
Garalgly	121.94	842	15,437	38412.94	3170
Movies	660.27	1233	6472	99923.95	5956
Electronics	582.85	1079	5644	84067.08	6064
CDs	331.08	1034	5297	45223.62	2884
Kindle	211.12	613	3068	17592.25	2310

Note. - indicates the running time exceeds 10 days.

Table 4.13: The chain decomposition comparison. For **utga**, we count the number of chains generated in the xy-plane and the number of chains generated in the xz-plane.

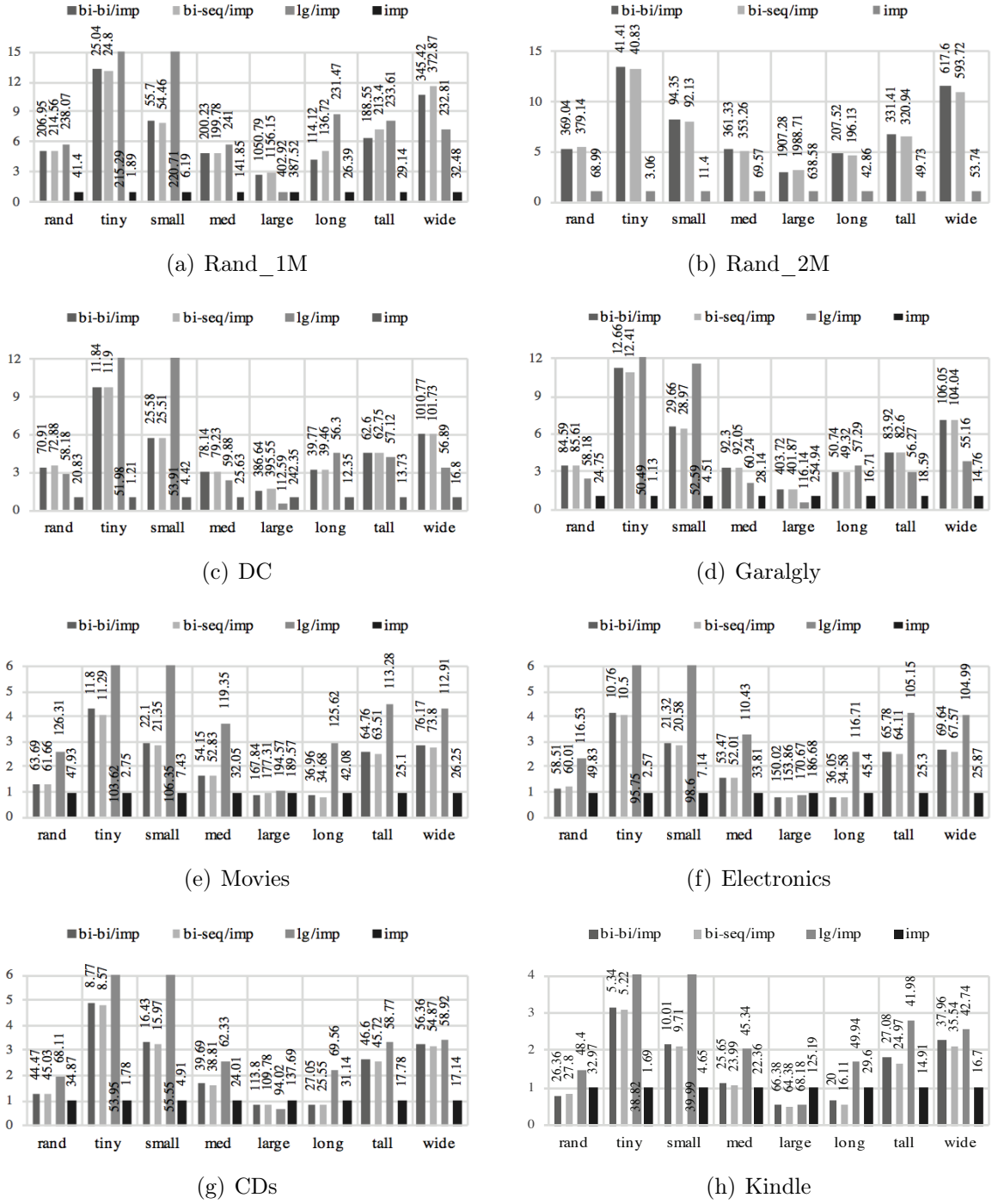
We implemented both the untangled chain decomposition algorithm discussed in Section 3.3 and the longest chain decomposition algorithm in [37]. We use **utga** and **lga** to represent them in the result tables, respectively. Table 4.13 shows the experimental results of the two chain decomposition methods. We are unable to give the partition results of the point set **2M** for **lga** since its running time exceeded

10 days. In Section 2.4, we mentioned that the running time for the longest chain decomposition method is $O(n^2 \lg^2 n)$. Compared with $O(n^{\frac{7}{4}} \lg n)$ for the untangled chain decomposition method, the time difference will increase significantly as the size of the data set grows. That explains why the longest chain decomposition method takes more than 10 days as the size of the point set reaches to two million. Even for relatively small point sets like **DC** (468020 points) and **Garalgly** (481351 points), the untangled chain decomposition method is at least 300 times faster than the longest chain decomposition method, which makes the longest chain decomposition method impractical in real-world scenario. Note that even the untangled chain decomposition can achieve much faster construction time compared to the longest chain decomposition, it is still much slower than the k-d tree, which takes $O(n \lg n)$ for construction.

On the other hand, the number of chains generated by **lga** is much fewer than that generated by **utga** for data sets **Rand_1M**, **Rand_2M**, **DC** and **Garalgly**. This difference in the number of generated chains could have an significant impact on the range query performance. Based on the experimental results in 2D, with fewer generated chains, the range query performance tends to have better query performance. But, as mentioned in Section 2.4, the chains generated by **lga** could potentially be tangled, so a query needs to inspect every single chain from the partition to make sure all the points within the range are found. For **utga**, on the other hand, the range searching can be performed adaptively based on the scale of the queries, since all the generated chains are untangled. In the next section, we present the range query results based on these two partition methods and compare them against the 3-dimensional k-d tree.

4.2.3 Range Query Comparison

In this section, we discuss the range query results for all data sets and all query types. Similarly to the range query comparison in 2D, the implicit k-d tree (**imp-kd**) is used as a reference for the performance comparison, since the implicit k-d tree in 3D still outperforms both the pointer-based k-d tree and the CGAL k-d tree in our experimental evaluation. For the untangled chain decomposition method, we keep the two different chain search methods **bi-bi** and **bi-seq**, where the search strategies are the same as in 2D. We use **lg** to represent the range searching method for the



(a) Rand_1M

(b) Rand_2M

(c) DC

(d) Garalgly

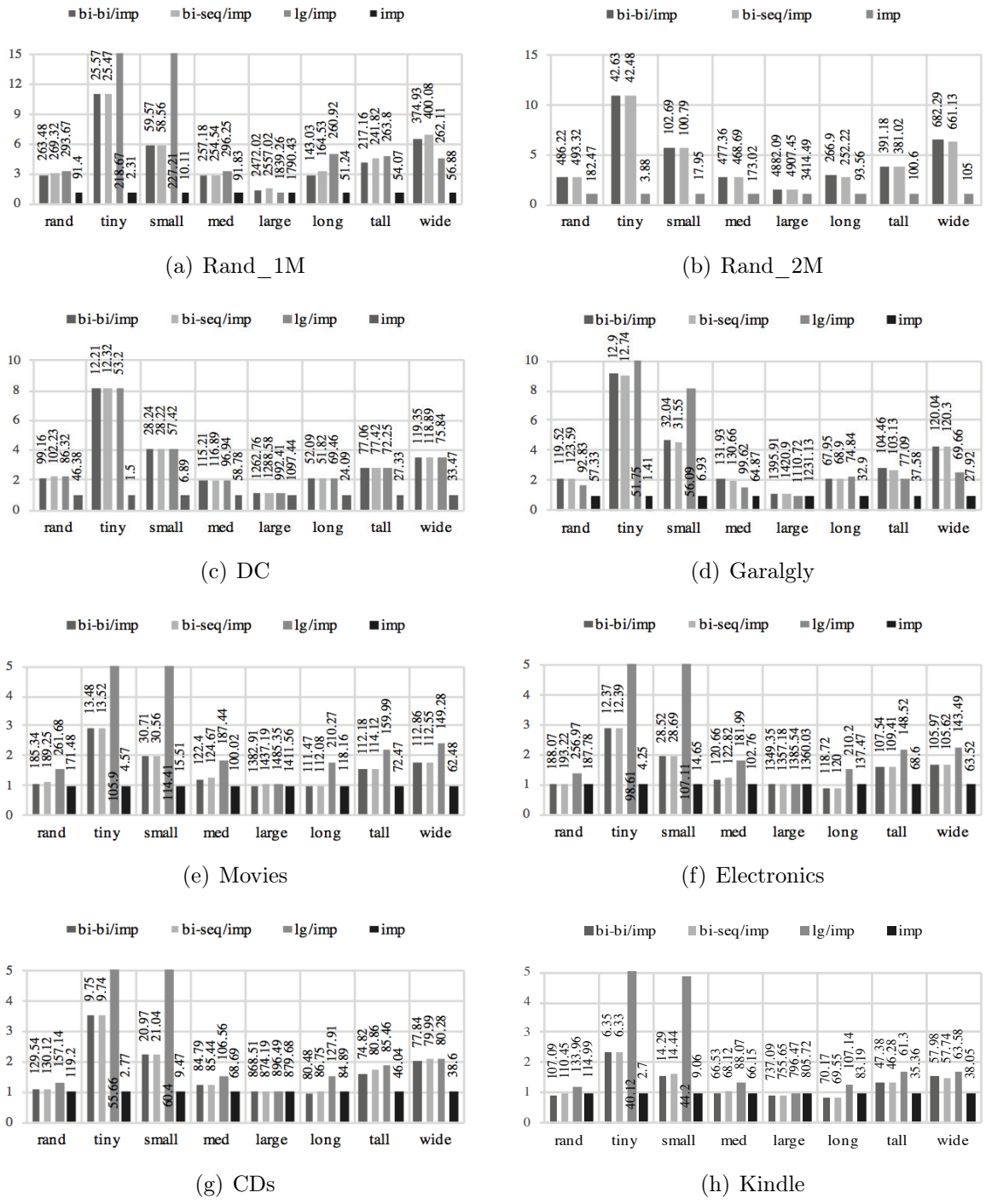
(e) Movies

(f) Electronics

(g) CDs

(h) Kindle

Figure 4.14: Comparison of total range counting time (in seconds) of 4 different range query methods using 8 different query types with 10,000 queries of each type and across 8 different data sets. The numbers above bars **bi-bi/imp**, **bi-seq/imp**, **lg/imp** and **imp** represent the range counting time of **bi-bi**, **bi-seq**, **lg** and **imp**, respectively.



(a) Rand_1M

(b) Rand_2M

(c) DC

(d) Garalgly

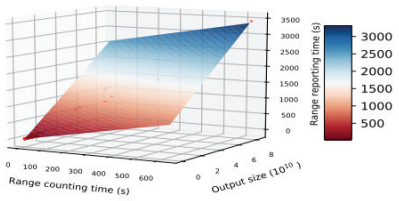
(e) Movies

(f) Electronics

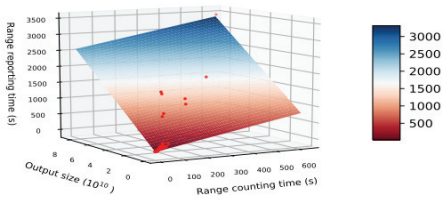
(g) CDs

(h) Kindle

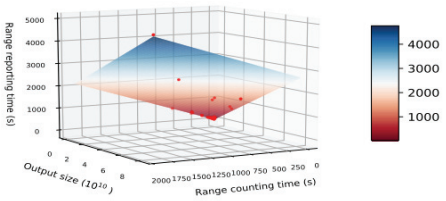
Figure 4.15: Comparison of total range reporting time (in seconds) of 4 different range query methods using 8 different query types with 10,000 queries of each type and across 8 different data sets. The numbers above bars **bi-bi/imp**, **bi-seq/imp**, **lg/imp** and **imp** represent the range reporting time of **bi-bi**, **bi-seq**, **lg** and **imp**, respectively.



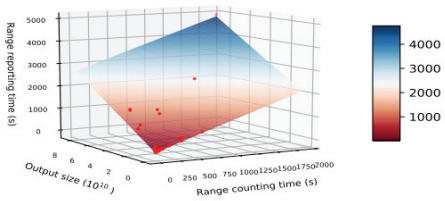
(a) **imp** - view angle 1



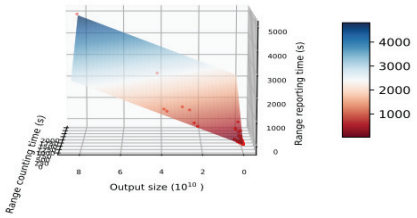
(b) **imp** - view angle 2



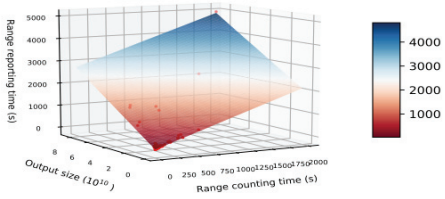
(c) **bi-bi** - view angle 1



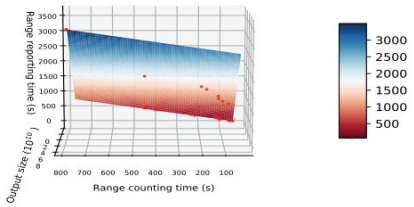
(d) **bi-bi** - view angle 2



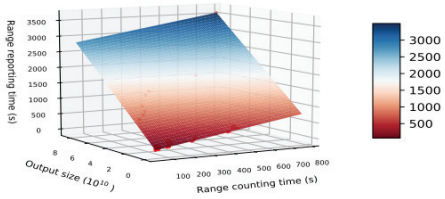
(e) **bi-seq** - view angle 1



(f) **bi-seq** - view angle 2



(g) **lg** - view angle 1



(h) **lg** - view angle 2

Figure 4.16: The relationship of range reporting time, range counting time and output size of four range query methods for all data sets with range reporting time on the Z axis, range counting time on the X axis and output size on the Y axis.

longest chain decomposition method [2].

Figure 4.14 shows the range counting results and Figure 4.15 shows the range reporting results. We keep using a bar chart to represent the query time (range counting or reporting time) comparison of different range query methods for each data set. Each bar in the chart represents the ratio of the range query time of two query methods. For instance, the bar with **lg/imp** is the ratio of the range query time of the longest chain decomposition range query method and the implicit k-d tree query method. The number above bar represents the range query time for the longest chain decomposition range query method. For the range counting results, **imp-kd** outperforms all the other three chain-decomposition-based range searching methods for most cases. Even though in some cases, the chain-decomposition-based range query methods have the advantage over k-d tree, , like **lg** in data sets **DC** and **Garalgly** and **bi-seq** in data set **Kindle** for query type *large*, the overall range query performance of the chain-decomposition-based range searching methods are still unable to match the performance of k-d tree. On the other hand, for the chain-decomposition-based range searching methods, the difference of the range query performance between **bi-bi** and **bi-seq** is not that obvious. Even though **lg** does not support the adaptive range search like the other two, it still can achieve good range query performance for the query type *large*.

Figure 4.16 shows that the range reporting time in 3D is in a linear relationship with a combination of the search cost (equivalent to the range counting time) and the output size. Figure 4.16 shows the graphical representations of this relationship for four different query methods and we calculate four fitting functions for each query methods using the linear regression algorithm [31]: the linear fitting function of **imp** is $z = 1.27x + 3.05 \times 10^{-8}y - 5.44$ with the variance score 1.0; the linear fitting function of **bi-bi** is $z = 1.12x + 3.21 \times 10^{-8}y - 10.77$ with the variance score 1.0; the linear fitting function of **bi-seq** is $z = 1.07x + 3.26 \times 10^{-8}y - 5.29$ with the variance score 1.0; the linear fitting function of **bi-bi** is $z = 0.99x + 3.3 \times 10^{-8}y + 1.82$ with the variance score 1.0, where the z-coordinate represents the range reporting time, the x-coordinate represents the range counting time and the y-coordinate indicates the output size. Similarly to the 2D case, the differences in range reporting time among

[2]The chain query method used in **lg** is **bi-bi**, since in our experiment, it has better range query performance compared to **bi-seq**.

Query type	Count (s)	Report (s)	Nodes visited	Nodes from subtree	Node ratio	Points from subtree	Total Points found	Points ratio
rand	41.4	91.4	1,142,055,539	61,491,892	5.38%	1,275,245,557	1,565,569,550	81.46%
tiny	1.89	2.31	46,876,660	1,015,801	2.17%	4,291,319	12,459,753	34.44%
small	6.91	10.11	183,862,296	7,257,370	3.95%	59,072,684	100,181,493	58.97%
med	41.85	91.83	1,162,741,700	64,006,230	5.50%	1,265,360,216	1,561,852,185	81.02%
large	403.66	1749.43	10,667,916,235	693,280,568	6.5%	39,238,865,226	42,142,374,188	93.11%
long	26.39	51.24	731,723,675	38,427,207	5.25%	592,002,590	780,531,859	75.85%
tall	29.14	54.07	813,573,620	49,175,948	6.04%	578,958,691	777,931,701	74.42%
wide	32.48	56.88	891,599,536	37,533,645	4.21%	558,225,697	778,335,691	71.72%

Table 4.14: Rand_1M - **imp-kd**

Query type	Count (s)	Report (s)	xy-plane chains found	Total xy bi-searches	xz-plane chains found	Total xz bi-searches	Bi-search_I	Total Points found
rand	206.95	263.48	46,550,539	142,718,557	739,455,028	6,027,852,662	4,936,464,372	1,565,569,550
tiny	25.04	25.57	10,250,253	56,817,161	35,623,744	498,850,111	485,750,897	12,459,753
small	55.77	59.57	19,653,088	81,005,339	129,396,277	1,315,842,587	1,211,595,445	100,181,493
med	200.23	257.18	45,935,561	122,517,539	688,545,425	5,983,618,620	4,774,599,709	1,561,852,185
large	1050.79	2472.02	120,121,852	143,219,254	3,925,365,621	36,744,818,552	22,412,644,977	42,142,374,188
long	114.12	143.03	24,216,214	90,397,960	480,234,517	3,689,355,944	2,952,032,425	780,531,859
tall	188.55	217.16	68,418,761	229,816,785	456,571,644	4,806,474,132	4,168,293,066	777,931,701
wide	345.42	374.93	68,870,253	233,039,084	1,612,367,003	10,237,375,522	9,481,079,143	778,335,691

xy-plane chains found: the total number of chains in the xy-plane that intersect the xy-projections of the query boxes.
Total xy bi-search: the total number of binary search steps applied in the xy-plane during the range queries.
xz-plane chains found: the total number of chains in the xz-plane that intersect the xz-projections of the query boxes.
Binary search_I: the number of binary search steps applied in the xz-plane to find out whether there exists a chain that intersect the xz-projection of the query box.

Table 4.15: Rand_1M-**bi-bi**

Query type	Count (s)	Report (s)	xy-plane chains found	Total xy bi-searches	xz-plane chains found	Bi-search_I	Best -dis	Linear scan	Total Points found
rand	214.56	269.32	46,550,539	142,718,557	739,455,028	4,936,464,372	10	1,871,283,025	1,565,569,550
tiny	24.8	25.47	10,250,253	56,817,161	35,623,744	485,750,897	1	3,109,770	12,459,753
small	54.46	58.56	19,653,088	81,005,339	129,396,277	1,211,595,445	1	46,477,465	100,181,493
med	199.78	254.54	45,935,561	122,517,539	688,545,425	4,774,599,709	5	1,182,533,086	1,561,852,185
large	1156.15	2557.02	120,121,852	143,219,254	3,925,365,621	22,412,644,977	24	23,864,456,762	42,142,374,188
long	136.72	164.53	24,216,214	90,397,960	480,234,517	2,952,032,425	1	478,682,293	780,531,859
tall	213.4	241.82	68,418,761	229,816,785	456,571,644	4,168,293,066	1	466,058,098	777,931,701
wide	372.87	400.08	68,870,253	233,039,084	1,612,367,003	9,481,079,143	1	461,706,087	778,335,691

Best-dis: the optimal scanning distance for the chains in the xz-plane.
Linear scan: the total number scanning steps performed in the xz-plane during the range queries.

Table 4.16: Rand_1M-**bi-seq**

Query type	Count (s)	Report (s)	Chains found	Binary search steps	Total Points found
rand	238.07	293.67	108,919,676	8,890,653,344	1,565,569,550
tiny	215.29	218.76	4,321,207	8,512,403,339	12,459,753
small	220.71	227.21	18,337,540	8,543,102,748	100,181,493
med	241	296.25	110,681,280	8,917,582,021	1,561,852,185
large	449.92	1839.26	846,754,921	12,814,032,134	42,142,374,188
long	231.47	260.92	91,918,707	8,776,154,257	780,531,859
tall	233.61	263.8	91,421,435	8,776,071,416	777,931,701
wide	232.81	262.11	92,767,121	8,788,482,355	778,335,691

Chains found: the total number of chains in 3D that intersect the range query boxes.

Table 4.17: Rand_1M-**lg**

the four query methods for each data set are mainly due to the search costs. To sum up, the implicit k-d tree range query method outperforms all the chain-decomposition-based range searching methods for both range counting and range reporting queries in nearly all cases. Next, we use one data set to analyze in detail and explain why the range query performance of the chain-based range query methods is unable to match the range query performance of the implicit k-d tree.

The data set we choose is **Rand_1M**. Tables 4.14, 4.15, 4.16 and 4.17 show the analysis results for the range query methods **imp-kd**, **bi-bi**, **bi-seq** and **lg**, respectively. The k-d tree analysis in 3D is similar to the analysis in 2D. For the untangled chain decomposition range query method (**bi-bi** and **bi-seq**), the partition process in 3D has been divided into two steps. Without lose of generality, we first run the chain decomposition and untangling processes in the xy-plane, then for each chain in the xy-plane, we project all points on the chain to the xz-plane and apply the chain decomposition and untangling processes again in the xz-plane. For the range query method **lg**, we count the total number of chains that intersect the query boxes and the total number of binary search steps during the range queries.

We first analyze each table individually. For range query method **imp-kd** in Table 4.14, the range counting time for the implicit k-d tree is in a linear relationship with the number of accessed nodes during range queries, which is similar to the 2D case. Figure 4.17 gives the graphical representation of this linear relationship. Compared to the 2D case, except for the range query type *large* (roughly 10 times greater 2D), the total number of accessed nodes in 3D is roughly 3 times greater than that in 2D. The number of nodes in the tree that report all points stored in their subtrees is still only a small fraction of the total number of visited nodes, but

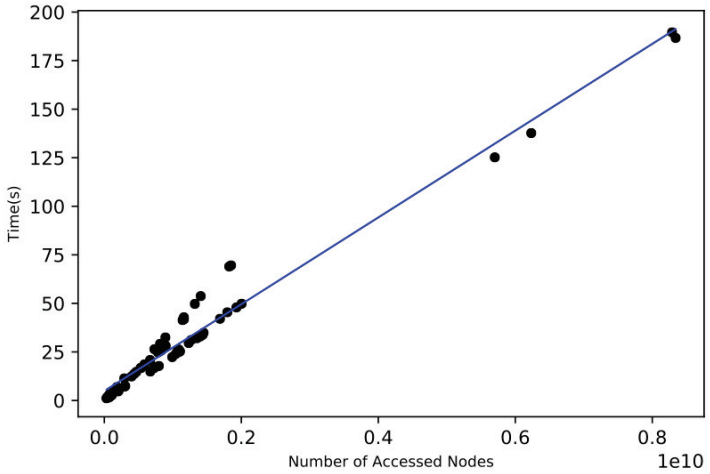


Figure 4.17: The relationship of range counting time and the number of accessed nodes for the implicit k-d tree for all data sets with the range counting time on the Y axis and the number of accessed nodes on the X axis. The blue trend-line shows a linear relationship of the two parameters using the function $y = 2.34 \times 10^{-8}x + 4.89$ that is calculated based on linear regression algorithm [31] and the variance score is 0.97.

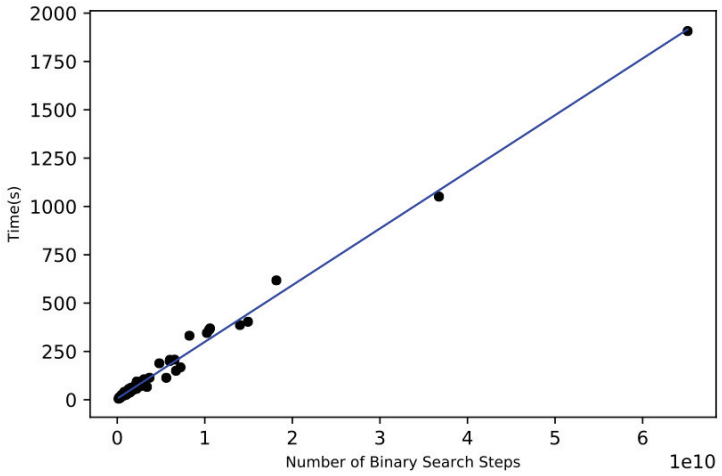
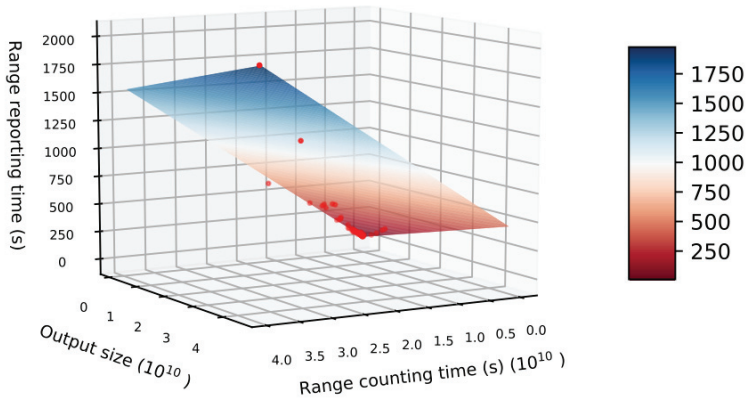
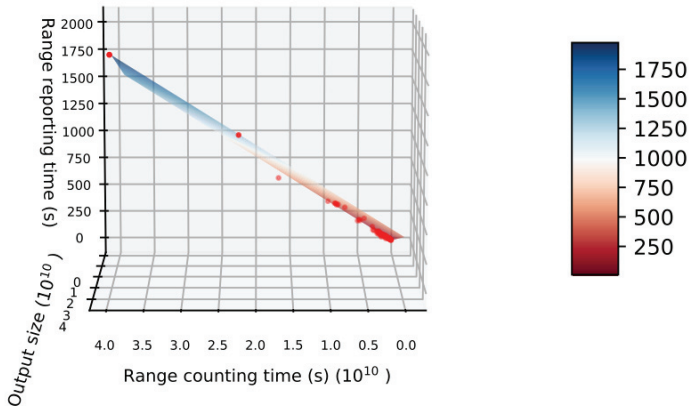


Figure 4.18: The relationship of range counting time and the number of binary search steps for **bi-bi** for all data sets with the range counting time on the Y axis and the number of binary search steps on the X axis. The blue trend-line shows a linear relationship of the two parameters using the function $y = 2.93 \times 10^{-8}x + 6.19$ with the variance score is 0.99.



(a) View angle 1



(b) View angle 2

Figure 4.19: The relationship of range counting time, the number of binary search steps and the number linear scanning steps for all data sets with the range counting time on the Z axis, the number of binary search steps on the X axis and the number linear scanning steps on the Y axis. The coloured surface is drawn using the function $z = 3.93 \times 10^{-8}x + 9.78 \times 10^{-9}y - 0.72$ with variance score 1.0.

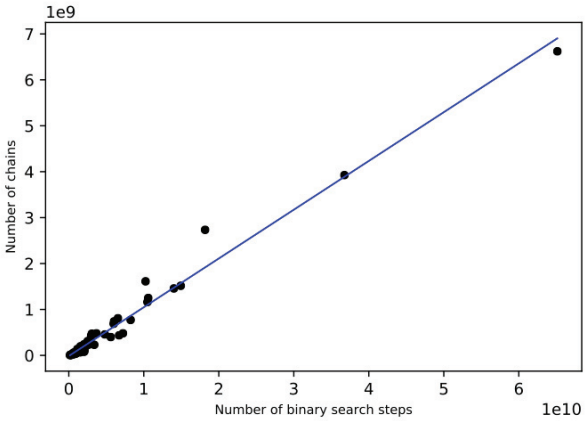
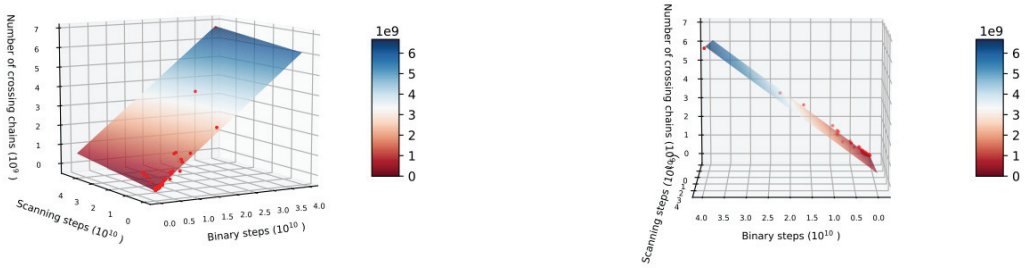


Figure 4.20: The relationship of number of binary search steps and the number of crossing chains for **bi-bi** for all data sets with the range counting time on the Y axis and the number of crossing chains on the X axis. The blue trend-line shows a linear relationship of the two parameters using the function $y = 0.11x - 14346874.72$. The variance score is 0.98.



(a) View angle 1

(b) View angle 2

Figure 4.21: The relationship of the number of crossing chains, the number of binary search steps and the number of scanning steps for **bi-seq** for all data sets with the number of crossing chains on the Z axis, the number of binary search steps on the X axis and the number scanning stepson the Y axis. The coloured surface is drawn using the function $z = 0.16x + 0.01y - 47683559.09$. The variance score is 0.99.

these nodes contribute most of the found points for each type of range queries. As for the untangled chain based range query methods (**bi-bi** and **bi-seq**), the range counting time of **bi-bi** is in a linear relationship with the number of binary search steps performed in the xz-plane and the range counting time of **bi-seq** is in a linear relationship with a combination of the total number binary searches steps and total

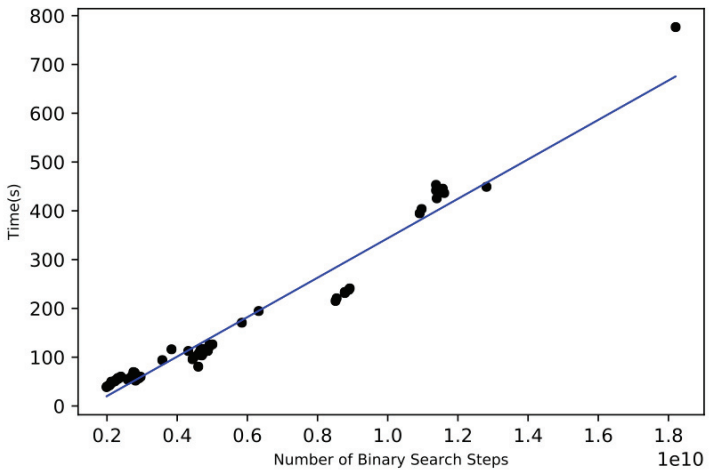


Figure 4.22: The relationship of range counting time and the number of binary search steps for **lg** for all data sets with the range counting time on the Y axis and the number of binary search steps on the X axis. The blue trend-line shows a linear relationship of the two parameters using the function $y = 4.04 \times 10^{-8}x - 60.52$. The variance score is 0.95.

number linear scanning steps performed in the xz-plane. Figures 4.18 and 4.19 show the graphical representations of these two linear relationships. Compared to the 2D case, the number of basic steps performed for the two query methods, especially binary search steps, is at least 5 times greater than in 2D for nearly all query types. For query type *large*, the number of binary search steps is roughly 20 times greater than in 2D. The explanation for this low range query performance is the large number of chains in the xz-plane that cross the query boxes. Figure 4.20 and 4.21 show the relationship of the number of crossing chains and the number of basic steps for the two untangled chain based range query methods. The reason for this large number of crossing chains is likely due to the large number of chains generated in the xz-plane. The number of chains generated in the xz-plane is much greater than the number of chains generated in the xy-plane (5 times greater for the Amazon point sets, 20 times greater for the 3D object point sets and 35 times greater for the random point sets, see Table 4.13). This large number of generated chains in the xz-plane may result in a large number of crossing chains in the xz-plane. For range query method **lg** in Table 4.17, we can observe that the query time differences among different types of queries (except *large*) are not that obvious and its range counting time is

in a linear relationship with the number of binary search steps performed during the range searching, see Figure 4.22. For **lg**, the query type has less impact on the query time, since we need to check every single chain to find whether there exists any point on the chain that lies in the range query box. The longer range query time of *large* is because more binary search steps (looking for the first and last points that lie in the range) need to apply on each chains that intersects the range query boxes (the number of chains found by *large* type is at least 10 times greater than the others, see Table 4.17).

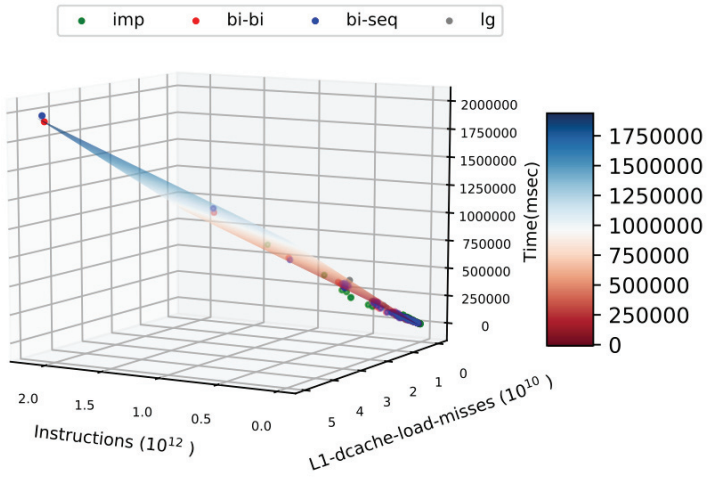
Next, we compare these four range query methods using their hardware performance results. As shown in Table 4.18, **imp-kd** outperforms all the chain-decomposition-based range query methods in all aspects. For the chain decomposition range query methods (**bi-bi**, **bi-seq** and **lg**), the noticeable differences comparing to the implicit k-d tree are the number of instructions and L1-dcache-load-misses. We conjectured that the task-clock (equivalent to the range query time) is mainly dependent on the number of instructions and number of L1-dcache-load-misses, which is similar to the 2D case.

One thing worth mentioning is that the slightly improved performance of chain-decomposition-based range searching methods for the Amazon point sets (especially for query types *large* and *long*) is due to its lower number of generated chains in the xz-plane. This relatively small number of generated chains is due to the z-coordinates for the Amazon point sets being the integers ranging from 1 to 5 and this small range scale makes the z-coordinate have little impact on the chain partition (most of the z-coordinates have the same value). Based on the experimental results and analysis mentioned above, we are able to conclude that the performance of the existing chain-decomposition-based range searching methods are unable to match the performance of k-d tree in 3-dimensional space. The reasons are the large number of chains generated by the chain partition steps and their significant number of basic query steps. We attached all the experimental results in Appendix B for reference.

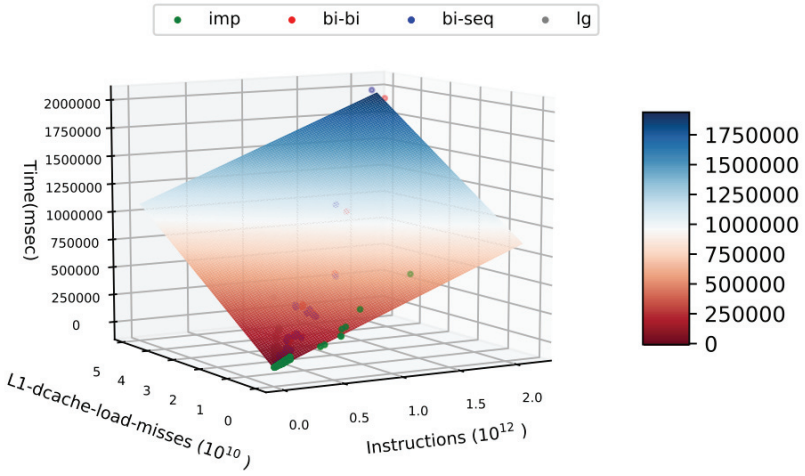
Performance	**imp-kd**		**bi-bi**		**bi-seq**		**lg**	
task-clock(msec)	43087.71		210393.02		216367.94		241844.86	
cycles	103,134,862,359		503,569,289,913		517,874,612,733		578,628,926,636	
context-switches	11,322	0.26 K/s	54,247	0.26 K/s	55,300	0.25 K/s	59,046	0.24 K/s
page-faults	7086	0.16 K/s	14,153	0.07 K/s	14,153	0.07 K/s	17,468	0.7 K/s
instruction	89,570,278,623	0.87 insns/cycle	306,100,422,338	0.61 insns/cycle	306,873,986,261	0.59 insns/cycle	243,905,637,322	0.42 insns/cycle
branches	13,593,779,700	315.49 M/s	54,143,034,044	257.34 M/s	56,406,968,068	260.7 M/s	44,045,153,456	182.12 M/s
branch-misses	702,982,374	5.17%	4,473,557,056	8.26%	4,524,672,941	8.02%	4,429,609,038	10.06%
L1-dcache-loads	47,078,476,318	1092.62 M/s	110,154,937,757	523.57 M/s	114,417,061,715	528.81 M/s	66,022,224,229	273 M/s
L1-dcache-load-misses	297,680,432	0.63%	4,452,851,926	4.04%	4,693,490,088	4.1%	8,325,080,125	12.61%
LLC-loads	443,897,234	10.3 M/s	5,630,780,233	26.76 M/s	5,808,882,659	26.85 M/s	10,989,180,229	45.44 M/s
LLC-loads-misses	288,862,910	65.07%	4,454,321,890	79.11%	4,695,712,851	80.84%	8,338,979,131	75.88%

Note:
K/s indicates kilobyte per seconds.
M/s represents megabyte per seconds.
insns/cycle indicates number of instructions per cycle.
The L1-dcache-loads and L1-dcache-load-misses represent the number of Level-1 data cache loads and the number of Level-1 data cache-misses.
The LLC-loads and LLC-load-misses represent the last level (Level-3) cache loads and cache-misses.
The percentage shown in branch-misses is the percentage of the number of branch-misses out of the total number of branches.
The percentage shown in L1-dcache-load-misses is the percentage of the number of L1-dcache-load-misses out of the total number of L1-dcache-loads.
The percentage of LLC-loads-misses is the percentage of the number of LLC-load-misses out of the total number of LLC-loads.

Table 4.18: Hardware performance analysis of data set **Rand_1M** for type *rand*



(a) View angle 1



(b) View angle 2

Figure 4.23: The relationship of task-clock(msec), the number of instructions and the number L1-dcache-load-misses for all data sets with the task-clock(msec) on the Z axis, the number of instructions on the X axis and the number L1-dcache-load-misses on the Y axis. The coloured surface is drawn using the function $z = 4.07 \times 10^{-7}x + 2.12 \times 10^{-5}y - 18710.8$ that is calculated based on linear regression algorithm [31].

Chapter 5

Conclusion and Future Work

We proposed the first data structure based on untangled monotonic chain for orthogonal range searching in 3-dimensional space. The idea is an extension of the 2-dimensional chain partition algorithm mentioned in [4].

In the experimental evaluation, we first re-evaluated the experimental studies conducted for the 2-dimensional range searching algorithm based on untangled monotonic chains in [14] and found that the k-d tree implementation in CGAL [34], which was used as a reference for the experimental evaluation in [14], is inefficient for the task at hand. Therefore, we implemented k-d trees ourselves with limited functionality (our implementation only supports fast orthogonal range queries over static point sets) and compared them against the chain-decomposition-based range searching methods. The experimental results showed that the performance of the chain-decomposition-based range searching methods is largely dependent on the number of chains generated during the partition steps, which could vary even for equally-sized point sets. Even with relatively few chains, the chain-decomposition-based range searching methods were only as competitive as the k-d tree, which contradicts the experimental results in [14]. Then, we implemented three different 3-dimensional chain-decomposition-based range query methods and compared them against the 3D k-d tree. The experimental results revealed that none of them are comparable with the range query performance of the k-d tree. The range searching method based on untangled chains suffered from its large number of generated chains and the longest chain range searching method is impractical due to its significant amount of construction time as the point set size gets larger.

The future improvement regarding the 3-dimensional chain decomposition method would be to design an efficiency partition algorithm, which could result in the fewer number of chains. Moreover, in [4], the authors mentioned that using the fractional cascading technique, the range query time of the untangled monotonic chain based

range query method in 2D can be improved to $O(\lg n + m + k)$ time, where m is the total number of chains and k is the output size. It would be interesting to see whether we can apply this technique to the range searching methods based on the monotonic chains in 3D.

Bibliography

[1] Pankaj K Agarwal, Jeff Erickson, et al. Geometric range searching and its relatives. *Contemporary Mathematics*, 223:1–56, 1999.

[2] Artur Andrzejak and Zhichen Xu. Scalable, efficient range queries for grid information services. In *Peer-to-Peer Computing, 2002.(P2P 2002). Proceedings. Second International Conference on*, pages 33–40. IEEE, 2002.

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

[4] Diego Arroyuelo, Francisco Claude, Reza Dorrigiv, Stephane Durocher, Meng He, Alejandro López-Ortiz, J Ian Munro, Patrick K Nicholson, Alejandro Salinger, and Matthew Skala. Untangled monotonic chains and adaptive range search. *Theoretical Computer Science*, 412(32):4200–4211, 2011.

[5] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *ACM SIGMOD Record*, volume 19, pages 322–331. Acm, 1990.

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

[7] Jon Louis Bentley. Multidimensional divide-and-conquer. *Communications of the ACM*, 23(4):214–229, 1980.

[8] Jon Louis Bentley and Hermann A. Maurer. Efficient worst-case data structures for range searching. *Acta Informatica*, 13(2):155–168, 1980.

[9] Manuel Blum, Robert W Floyd, Vaughan Pratt, Ronald L Rivest, and Robert E Tarjan. Time bounds for selection. *Journal of computer and system sciences*, 7(4):448–461, 1973.

[10] Min Cai, Martin Frank, Jinbo Chen, and Pedro Szekely. Maan: A multi-attribute addressable network for grid information services. *Journal of Grid Computing*, 2(1):3–14, 2004.

[11] Bernard Chazelle and Leonidas J Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(1):133–162, 1986.

[12] Bernard Chazelle, Micha Sharir, and Emo Welzl. Quasi-optimal upper bounds for simplex range searching and new zone theorems. *Algorithmica*, 8(1-6):407–429, 1992.

[13] Bernard Chazelle and Emo Welzl. Quasi-optimal range searching in spaces of finite vc-dimension. *Discrete & Computational Geometry*, 4(5):467–489, 1989.

[14] Francisco Claude, J Ian Munro, and Patrick K Nicholson. Range queries over untangled chains. In *International Symposium on String Processing and Information Retrieval*, pages 82–93. Springer, 2010.

[15] William Cook. The Traveling Salesman Problem. `http://www.math.uwaterloo.ca/tsp/data/index.html`, 2009. [Online; accessed 19-March-2016].

[16] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. Simplex range searching. *Computational Geometry: Algorithms and Applications*, pages 335–355, 2008.

[17] Mark De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. Computational geometry. In *Computational geometry*, pages 1–17. Springer, 2000.

[18] Gabriele Di Stefano, Stefan Krause, Marco E Lübbecke, and Uwe T Zimmermann. On minimum k-modal partitions of permutations. In *Latin American Symposium on Theoretical Informatics*, pages 374–385. Springer, 2006.

[19] Fedor V Fomin, Dieter Kratsch, and Jean-Christophe Novelli. Approximating minimum cocolorings. *Information Processing Letters*, 84(5):285–290, 2002.

[20] Antonin Guttman. *R-trees: a dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[21] Sangyong Hwang, Keunjoo Kwon, Sang K Cha, and Byung S Lee. Performance evaluation of main-memory r-tree variants. In *International Symposium on Spatial and Temporal Databases*, pages 10–27. Springer, 2003.

[22] Ji Jin, Ning An, and Anand Sivasubramaniam. Analyzing range queries on spatial data. In *Data Engineering, 2000. Proceedings. 16th International Conference on*, pages 525–534. IEEE, 2000.

[23] Jure Leskovec, Polo Chau and Ana Pavlisic. SNAP Datasets. `http://jmcauley.ucsd.edu/data/amazon/`, 2013. [Online; accessed 23-May-2016].

[24] KV Ravi Kanth and Ambuj Singh. Optimal dynamic range searching innonreplicating index structures. In *International Conference on Database Theory*, pages 257–276. Springer, 1999.

[25] Der-Tsai Lee and CK Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica*, 9(1):23–29, 1977.

[26] Jin Li, Qian Wang, Cong Wang, Ning Cao, Kui Ren, and Wenjing Lou. Fuzzy keyword search over encrypted data in cloud computing. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–5. IEEE, 2010.

[27] Ming Li, Shucheng Yu, Ning Cao, and Wenjing Lou. Authorized private keyword search over encrypted data in cloud computing. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 383–392. IEEE, 2011.

[28] Jiří Matoušek. Geometric range searching. *ACM Computing Surveys (CSUR)*, 26(4):422–461, 1994.

[29] Matthew Berger, Joshua A. Levine, Luis Gustavo Nonato, Gabriel Taubin, and Claudio T. Silva. A Benchmark for Surface Reconstruction. `http://www.cs.utah.edu/~bergerm/recon_bench/point/`, 2013. [Online; accessed 2-April-2016].

[30] Dimitris Papadias, Jun Zhang, Nikos Mamoulis, and Yufei Tao. Query processing in spatial network databases. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 802–813. VLDB Endowment, 2003.

[31] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[32] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. 1987.

[33] Kenneth J Supowit. Decomposing a set of points into chains, with applications to permutation and circle graphs. *Information Processing Letters*, 21(5):249–252, 1985.

[34] Hans Tangelder and Andreas Fabri. dD spatial searching. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.2 edition, 2013.

[35] Boyang Wang, Ming Li, and Haitao Wang. Geometric range search on encrypted spatial data. *IEEE Transactions on Information Forensics and Security*, 11(4):704–719, 2016.

[36] Boyang Wang, Ming Li, Haitao Wang, and Hui Li. Circular range search on encrypted spatial data. In *Communications and Network Security (CNS), 2015 IEEE Conference on*, pages 182–190. IEEE, 2015.

[37] Xinjing Wei. 3D chain decomposition and range search. Master's thesis, Dalhousie University, 6299 South St, Halifax, NS B3H 4R2, 7 2015.

[38] Bing Yang, Jing Chen, En-Yue Lu, and Si-Qing Zheng. Design and performance evaluation of sequence partition algorithms. *Journal of Computer Science and Technology*, 23(5):711–718, 2008.

[39] Reuven Bar Yehuda and Sergio Fogel. Partitioning a sequence into few monotone subsequences. *Acta Informatica*, 35(5):421–440, 1998.

Appendix A

The 2D Range Query Results

Query type	Count (s)	Report (s)	Nodes visited	Nodes from subtree	Node ratio	Points from subtree	Total Points found	Points ratio
rand	20.91	379.54	595607326	70400525	11.82%	12351370551	12523213300	98.63%
tiny	0.89	1.5	23316052	1809570	7.76%	14384606	19939312	72.14%
small	2.83	9.37	78526521	8194858	10.44%	201237941	222696159	90.36%
med	8.44	65.84	238051302	27393822	11.51%	1928871952	1996683853	96.6%
large	56.18	2591.56	1604165680	196174433	12.23%	88408238114	88876613481	99.47%
tall	17.58	76.22	513314029	49652492	9.67%	1857755238	2010372899	92.41%
wide	24.12	82.63	696933666	85976430	12.34%	1803103335	1998483230	90.22%

Table A.1: 2M-**imp-kd** analysis

Query type	Count (s)	Report (s)	Chains found	Binary search steps	Binary search_I	Binary search_II	Total Points found
rand	29.12	393.15	64743029	1141168778	198584515	942584263	12523213300
tiny	1.46	2.09	3081187	53870571	34888004	18982567	19939312
small	4.11	10.83	9674346	156223725	62123349	94100376	222696159
med	12.21	71.61	27222755	463855445	111478205	352377240	1996683853
large	73.4	2633.68	158443790	3038902104	210503374	2828398730	88876613481
tall	30.88	91.68	88418456	1309091092	498396894	810694198	2010372899
wide	32.78	92.31	88508063	1309492826	500928698	808564128	1998483230

Table A.2: 2M-**bi-bi** analysis

Query type	Count (s)	Report (s)	Chains found	Binary search steps	Best distance	Linear scan	Total Points found
rand	32.22	395.5	64743029	198584515	21	1984267130	12523213300
tiny	1.35	1.95	3081187	34888004	2	12123967	19939312
small	3.64	10.15	9674346	62123349	3	95364923	222696159
med	11.84	70.99	27222755	111478205	10	469087116	1996683853
large	91.49	2645.51	158443790	210503374	41	8708653719	88876613481
tall	27.51	86.58	88418456	498396894	4	781966827	2010372899
wide	29.06	87.86	88508063	500928698	4	776589160	1998483230

Table A.3: 2M-**bi-seq** analysis

Query type	Count (s)	Report (s)	Nodes visited	Nodes from subtree	Node ratio	Points from subtree	Total Points found	Points ratio
rand	3.06	22.21	121477428	12957501	10.67%	645127571	682196174	94.57%
tiny	0.13	0.14	4318247	64414	1.49%	225696	599988	37.62%
small	0.34	0.55	12352275	557915	4.52%	5125818	7274131	70.47%
med	1.1	3.46	42558015	3184381	7.48%	71882088	82323621	87.32%
large	10.34	156.49	422030508	63308423	15%	5071427875	5243195626	96.72%
tall	2.44	5.87	98501900	8672257	8.8%	92440307	114819141	80.51%
wide	1.85	2.98	71771939	2694500	3.75%	24480792	38083901	64.28%

Table A.4: China-**imp-kd** analysis

Query type	Count (s)	Report (s)	Chains found	Binary search steps	Binary search_I	Binary search_II	Total Points found
rand	2	21.7	13277300	162654594	55992968	106661626	682196174
tiny	0.24	0.26	491100	13628697	12800605	828092	599988
small	0.38	0.6	1611822	23631779	18030816	5600963	7274131
med	0.87	3.26	4870599	59775656	29685930	30089726	82323621
large	4.22	155.47	31332759	445643690	36545422	409098268	5243195626
tall	2.51	5.85	19009748	191826648	115561693	76264955	114819141
wide	1.64	2.79	12177431	117531145	85068119	32463026	38083901

Table A.5: China-**bi-bi** analysis

Query type	Count (s)	Report (s)	Chains found	Binary search steps	Best distance	Linear scan	Total Points found
rand	1.96	21.71	13277300	55992968	21	224145117	682196174
tiny	0.24	0.25	491100	12800605	3	478974	599988
small	0.36	0.57	1611822	18030816	1	6362631	7274131
med	0.79	3.21	4870599	29685930	11	45166482	82323621
large	3.67	154.56	31332759	36545422	41	717215905	5243195626
tall	2.25	5.62	19009748	115561693	2	65560471	114819141
wide	1.52	2.66	12177431	85068119	1	32126444	38083901

Table A.6: China-**bi-seq** analysis

Query type	Count (s)	Report (s)	Nodes visited	Nodes from subtree	Node ratio	Points from subtree	Total Points found	Points ratio
rand	21.48	617.44	720115282	94562551	13.13%	20578346231	20807251013	98.9%
tiny	0.46	0.83	14025120	700576	5%	10063671	12256895	82.11%
small	1.57	6.05	49488214	3960595	8%	141400495	152220939	92.89%
med	6.36	60.07	198787698	22350318	11.24%	1811793567	1867805110	97%
large	60.19	4084.66	2151183537	377921993	17.57%	139965387077	140828409413	99.39%
tall	12.53	57.99	418303773	32181559	7.69%	1457728773	1555687563	93.7%
wide	19.16	86.17	631149556	52679070	8.35%	2155793177	2286821545	94.27%

Table A.7: World-**imp-kd** analysis

Query type	Count (s)	Report (s)	Chains found	Binary search steps	Binary search_I	Binary search_II	Total Points found
rand	25.61	622.21	66700631	1107293907	277199596	830094311	20807251013
tiny	0.9	1.26	1992325	38690130	32080688	6609442	12256895
small	2.48	6.88	6888319	101971273	64254625	37716648	152220939
med	8.58	62.35	23517401	356702422	143474555	213227867	1867805110
large	55.61	4073.89	151164398	2901719554	185406698	2716312856	140828409413
tall	20.15	65.41	63824718	862078454	457992516	404085938	1555687563
wide	22.65	88.65	65586344	897191313	463778869	433412444	2286821545

Table A.8: World-**bi-bi** analysis

Query type	Count (s)	Report (s)	Chains found	Binary search steps	Best distance	Linear scan	Total Points found
rand	31.85	639.75	66700631	277199596	39	2375513440	20807251013
tiny	0.87	1.23	1992325	32080688	2	6788096	12256895
small	2.43	6.8	6888319	64254625	4	49864306	152220939
med	8.87	63.51	23517401	143474555	16	373990284	1867805110
large	71.72	4129.33	151164398	185406698	59	7093831871	140828409413
tall	19.52	64.33	63824718	457992516	4	513723613	1555687563
wide	23.57	89.42	65586344	463778869	4	699517692	2286821545

Table A.9: World-**bi-seq** analysis

Query type	Count (s)	Report (s)	Nodes visited	Nodes from subtree	Node ratio	Points from subtree	Total Points found	Points ratio
rand	13.44	286.81	536838528	60810664	11.33%	9899175508	10057787200	98.42%
tiny	0.58	1.14	21034363	1424429	6.77%	15877587	20550886	77.26%
small	1.8	7.73	68228483	6212956	9.11%	192890218	210395422	91.68%
med	5.31	53.49	208922948	21291535	10.19%	1691598759	1748354644	96.75%
large	37.03	1946.31	1524351421	203488975	13.35%	69354533337	69869697854	99.26%
tall	10.28	46.69	407141119	39539158	9.71%	1212235369	1327876857	91.29%
wide	15.88	57.26	649718148	63800789	9.82%	1326400819	1501859838	88.32%

Table A.10: Movies-**imp-kd** analysis

Query type	Count (s)	Report (s)	Chains found	Binary search steps	Binary search_I	Binary search_II	Total Points found
rand	14.93	286.05	42689253	748256443	143663114	604593329	10057787200
tiny	0.9	1.43	2229979	41629622	27783864	13845758	20550886
small	2.26	7.86	6840701	111249236	47102240	64146996	210395422
med	6.12	53.15	18707415	315090176	82370783	232719393	1748354644
large	36.07	1931.96	104718057	1997753309	144390100	1853363209	69869697854
tall	12.6	48.59	47015950	695729841	291279407	404450434	1327876857
wide	17.08	57.32	66583588	928142125	421285062	506857063	1501859838

Table A.11: Movies-**bi-bi** analysis

Query type	Count (s)	Report (s)	Chains found	Binary search steps	Best distance	Linear scan	Total Points found
rand	16.07	294.85	42689253	143663114	28	1547299670	10057787200
tiny	0.84	1.37	2229979	27783864	1	18678528	20550886
small	2.04	7.73	6840701	47102240	4	73518828	210395422
med	5.91	54.2	18707415	82370783	16	400345779	1748354644
large	42.45	1996.14	104718057	144390100	48	6198023189	69869697854
tall	12.21	47.92	47015950	291279407	5	444533425	1327876857
wide	15.34	56.67	66583588	421285062	5	525974727	1501859838

Table A.12: Movies-**bi-seq** analysis

Query type	Count (s)	Report (s)	Nodes visited	Nodes from subtree	Node ratio	Points from subtree	Total Points found	Points ratio
rand	14.63	328.15	575589873	64519560	11.21%	11424479366	11593777650	98.54%
tiny	0.58	1.17	20801027	1424916	6.85%	14946269	19621599	76.17%
small	1.86	7.92	70181405	6505798	9.27%	197812602	216213883	91.49%
med	5.78	57.09	220211820	22707638	10.31%	1817621064	1878794798	96.74%
large	38.79	2290.41	1582575478	200798013	12.69%	82839010227	83346437539	99.39%
tall	11.3	52.65	448095679	43983894	9.82%	1333845732	1461951892	91.24%
wide	16.11	55.91	645664532	65308885	10.11%	1222013619	1399257235	87.33%

Table A.13: Electronics-**imp-kd** analysis

Query type	Count (s)	Report (s)	Chains found	Binary search steps	Binary search_I	Binary search_II	Total Points found
rand	14.15	328.67	40644339	725235338	130126090	595109248	11593777650
tiny	0.85	1.35	1920716	37750372	25202187	12548185	19621599
small	2.13	7.92	6204468	103441112	42930014	60511098	216213883
med	5.91	56.72	17871135	304755106	76290790	228464316	1878794798
large	34.08	2309.79	98945085	1915294375	122206060	1793088315	83346437539
tall	13.21	55.24	46468613	704843102	280803802	424039300	1461951892
wide	15.53	56.84	58222402	822331704	361116158	461215546	1399257235

Table A.14: Electronics-**bi-bi** analysis

Query type	Count (s)	Report (s)	Chains found	Binary search steps	Best distance	Linear scan	Total Points found
rand	15.98	327.51	40644339	130126090	25	1485770898	11593777650
tiny	0.78	1.28	1920716	25202187	1	17986837	19621599
small	1.92	7.61	6204468	42930014	4	73242346	216213883
med	5.94	56.38	17871135	76290790	12	365455348	1878794798
large	41.61	2302.69	98945085	122206060	42	5944901969	83346437539
tall	11.81	50.4	46468613	280803802	7	488790100	1461951892
wide	13.13	51.06	58222402	361116158	5	477581944	1399257235

Table A.15: Electronics-**bi-seq** analysis

Query type	Count (s)	Report (s)	Nodes visited	Nodes from subtree	Node ratio	Points from subtree	Total Points found	Points ratio
rand	10.39	197.03	427585249	48411392	11.32%	6848943446	6976514085	98.17%
tiny	0.47	0.79	16437344	1064512	6.48%	9263708	12714492	72.86%
small	1.47	5.16	54124674	5012926	9.26%	126658305	140285275	90.29%
med	4.3	35.74	164575578	17151572	10.42%	1140727921	1185088370	96.26%
large	30.42	1379.23	1265507620	169239194	13.37%	50022144965	50468181700	99.12%
tall	8.28	33.02	326491816	36758300	11.26%	848791738	934245421	90.85%
wide	11.96	38.16	486911828	39666212	8.15%	844029077	976704146	86.42%

Table A.16: CDs-**imp-kd** analysis

Query type	Count (s)	Report (s)	Chains found	Binary search steps	Binary search_I	Binary search_II	Total Points found
rand	10.74	198.57	36759489	611956743	124854107	487102636	6976514085
tiny	0.74	1.06	1892280	35435705	25107981	10327724	12714492
small	1.73	5.47	6003246	93832657	41773441	52059216	140285275
med	4.51	36.31	16322037	261760337	71026129	190734208	1185088370
large	25.57	1387.62	91710848	1665159118	125847531	1539311587	50468181700
tall	9.74	34.37	40305074	568571697	245600549	322971148	934245421
wide	11.52	37.43	53230471	713446330	318329922	395116408	976704146

Table A.17: CDs-**bi-bi** analysis

Query type	Count (s)	Report (s)	Chains found	Binary search steps	Best distance	Linear scan	Total Points found
rand	11.5	199.21	36759489	124854107	20	1071747334	6976514085
tiny	0.7	1.01	1892280	25107981	1	11155045	12714492
small	1.56	5.25	6003246	41773441	3	58591028	140285275
med	4.27	36.01	16322037	71026129	16	328519171	1185088370
large	29.59	1396.25	91710848	125847531	43	4854630023	50468181700
tall	9.02	33.44	40305074	245600549	5	342632446	934245421
wide	10.27	36.43	53230471	318329922	5	386714392	976704146

Table A.18: CDs-**bi-seq** analysis

Appendix B

The 3D Range Query Results

Query type	Count (s)	Report (s)	Nodes visited	Nodes from subtree	Node ratio	Points from subtree	Total Points found	Points ratio
rand	68.99	182.47	1823438735	103153067	5.66	2657862097	3134932819	84.78
tiny	3.06	3.88	73362909	2045640	2.79	10705804	25079503	42.69
small	11.4	17.95	291814395	12945111	4.44	131388916	200559494	65.51
med	69.57	173.02	1850613592	106735372	5.77	2644014683	3129512255	84.49
large	638.58	3414.49	16944443753	1112336160	6.56	79797179204	84471760631	94.47
long	42.86	93.56	1161599532	80423026	6.92	1262744966	1557587529	81.07
tall	49.73	100.6	1321427590	56955367	4.31	1219568671	1561371848	78.11
wide	53.74	105	1407496834	75267223	5.35	1204277309	1570310105	76.69

Table B.1: 2M-**imp-kd** analysis

Query type	Count (s)	Report (s)	xy-plane chains found	Total xy bi-searches	xz-plane chains found	Total xz bi-searches	Bi-search_I	Total Points found
rand	369.04	486.22	65598606	198736728	1249770468	10595771209	8543363754	3134932819
tiny	41.41	42.63	14444021	77078622	60526974	822560418	795617274	25079503
small	94.35	102.69	27634038	111487957	217884650	2226589317	2020755949	200559494
med	361.33	477.36	64654126	170376137	1162447195	10487577858	8219759350	3129512255
large	1907.28	4882.09	169312607	200549610	6621030060	65140076747	39375526558	84471760631
long	207.52	266.9	34002182	124852573	808733783	6552025463	5134086857	1557587529
tall	331.41	391.18	97010097	324732812	772006327	8242235167	7008035045	1561371848
wide	617.6	682.29	96908092	328788676	2734356977	18174785922	16710191822	1570310105

Table B.2: 2M-**bi-bi** analysis

Query type	Count (s)	Report (s)	xy-plane chains found	Total xy bi-searches	xz-plane chains found	Bi-search_I	Best -dis	Linear scan	Total Points found
rand	379.14	493.32	65598606	198736728	1249770468	8543363754	10	3458071435	3134932819
tiny	40.83	42.48	14444021	77078622	60526974	795617274	1	7424879	25079503
small	92.13	100.79	27634038	111487957	217884650	2020755949	2	104050817	200559494
med	353.26	468.69	64654126	170376137	1162447195	8219759350	5	2161880906	3129512255
large	1988.71	4907.45	169312607	200549610	6621030060	39375526558	25	45987050664	84471760631
long	196.13	252.22	34002182	124852573	808733783	5134086857	1	1022474280	1557587529
tall	320.94	381.02	97010097	324732812	772006327	7008035045	1	1007617665	1561371848
wide	593.72	661.13	96908092	328788676	2734356977	16710191822	1	1009273672	1570310105

Table B.3: 2M-**bi-seq** analysis

Query type	Count (s)	Report (s)	Nodes visited	Nodes from subtree	Node ratio	Points from subtree	Total Points found	Points ratio
rand	20.83	46.38	668,545,953	41,436,217	6.20%	641,481,364	819,508,008	78.28%
tiny	1.21	1.5	34,835,660	839,257	2.41%	3,758,362	9,142,559	41.11%
small	4.42	6.89	134,909,952	5,783,631	4.29%	46,148,547	74,887,609	61.62%
med	25.63	58.78	825,314,563	51,192,629	6.20%	863,060,921	1,081,054,895	79.84%
large	242.35	1097.44	8,099,917,366	790,684,301	9.76%	22,986,109,745	26,055,333,561	88.22%
long	12.35	24.09	396,930,581	18,939,661	4.77%	267,204,667	360,431,602	74.13%
tall	13.73	27.33	436,521,290	24,956,762	5.72%	317,789,753	423,378,443	75.06%
wide	16.8	33.47	536,952,171	33,770,143	6.29%	402,365,475	527,725,051	76.25%

Table B.4: DC-**imp-kd** analysis

Query type	Count (s)	Report (s)	xy-plane chains found	Total xy bi-searches	xz-plane chains found	Total xz bi-searches	Bi-search_I	Total Points found
rand	70.91	99.16	31,798,716	99,860,254	242,132,940	2,039,855,101	1,723,546,637	819,508,008
tiny	11.84	12.21	7,937,997	42,459,303	13,122,249	240,248,572	233,699,037	9,142,559
small	25.58	28.24	14,718,186	59,574,693	51,045,943	596,784,519	554,135,281	74,887,609
med	78.14	115.21	33,943,935	92,734,002	265,664,336	2,346,112,395	1,920,012,771	1,081,054,895
large	386.64	1262.76	84,519,538	103,191,935	1,457,176,376	14,010,643,367	8,156,052,115	26,055,333,561
long	39.77	52.09	18,105,559	66,960,766	137,728,491	1,136,135,839	962,733,404	360,431,602
tall	62.6	77.06	36,064,691	116,221,434	142,908,393	1,600,383,333	1,409,087,392	423,378,443
wide	100.77	119.35	40,443,124	132,103,007	427,774,821	2,972,816,478	2,717,733,535	527,725,051

Table B.5: DC-**bi-bi** analysis

Query type	Count (s)	Report (s)	xy-plane chains found	Total xy bi-searches	xz-plane chains found	Bi-search_I	Best -dis	Linear scan	Total Points found
rand	71.72	100.15	31,798,716	99,860,254	242,132,940	1,723,546,637	10	404,323,194	819,508,008
tiny	11.74	12.14	7,937,997	42,459,303	13,122,249	233,699,037	1	6,414,513	9,142,559
small	25.16	27.87	14,718,186	59,574,693	51,045,943	554,135,281	1	59,608,214	74,887,609
med	78.14	114.74	33,943,935	92,734,002	265,664,336	1,920,012,771	5	431,702,253	1,081,054,895
large	490.56	1269.05	84,519,538	103,191,935	1,457,176,376	8,156,052,115	28	6,468,125,514	26,055,333,561
long	48.25	51.16	18,105,559	66,960,766	137,728,491	962,733,404	2	186,203,083	360,431,602
tall	74.79	76.37	36,064,691	116,221,434	142,908,393	1,409,087,392	2	219,285,545	423,378,443
wide	126.19	117.69	40,443,124	132,103,007	427,774,821	2,717,733,535	1	444,613,008	527,725,051

Table B.6: DC-**dis-bi** analysis

Query type	Count (s)	Report (s)	Chains found	Binary search steps	Total Points found
rand	58.18	86.32	28,465,707	2,917,204,806	819,508,008
tiny	51.98	53.2	1,399,619	2,812,336,801	9,142,559
small	53.91	57.42	6,229,677	2,825,264,515	74,887,609
med	59.88	96.94	36,235,489	2,957,992,589	1,081,054,895
large	112.59	992.41	286,112,111	4,315,948,770	26,055,333,561
long	56.3	69.46	19,721,904	2,873,271,488	360,431,602
tall	57.12	72.25	21,366,983	2,877,433,978	423,378,443
wide	56.89	75.84	26,842,474	2,908,214,087	527,725,051

Table B.7: DC-**lg** analysis

Query type	Count (s)	Report (s)	Nodes visited	Nodes from subtree	Node ratio	Points from subtree	Total Points found	Points ratio
rand	24.75	57.33	783659811	50825999	6.49%	793711625	1008288397	78.72%
tiny	1.13	1.41	32555177	813377	2.5%	3500126	8351519	41.91%
small	4.51	6.93	137229955	6231241	4.54%	44903505	74336271	60.41%
med	28.14	64.87	895804422	58570331	6.54%	900184218	1140684391	78.92%
large	254.94	1231.13	8543146065	910351143	10.66%	26264600776	29601868455	88.73%
long	16.71	32.9	529289573	25538194	4.82%	375012795	503123286	74.54%
tall	18.59	37.58	585584529	37673162	6.43%	437680976	586037805	74.68%
wide	14.76	27.92	467009973	29640511	6.35%	297166943	408917456	72.67%

Table B.8: Garalgly-**imp-kd** analysis

Query type	Count (s)	Report (s)	xy-plane chains found	Total xy bi-searches	xz-plane chains found	Total xz bi-searches	Bi-search_I	Total Points found
rand	84.59	119.52	32007711	107615682	311216761	2518713106	2097823595	1008288397
tiny	12.66	12.9	8376807	43955230	14904646	258310139	251734643	8351519
small	29.66	32.04	15885643	65463253	64005618	700889798	653512565	74336271
med	92.3	131.93	33483623	96175011	342082005	2868728943	2340006494	1140684391
large	403.72	1395.91	82314606	98567576	1515133709	14927974718	8474145681	29601868455
long	50.74	67.95	19424358	73358199	196930536	1552565930	1284987449	503123286
tall	83.92	104.46	42995033	149761200	198002012	2182991027	1906343791	586037805
wide	106.05	120.04	37394981	130500684	471356916	3067659734	2839547318	408917456

Table B.9: Garalgly-**bi-bi** analysis

Query type	Count (s)	Report (s)	xy-plane chains found	Total xy bi-searches	xz-plane chains found	Bi-search_I	Best -dis	Linear scan	Total Points found
rand	85.61	123.59	32007711	107615682	311216761	2097823595	10	529745630	1008288397
tiny	12.41	12.74	8376807	43955230	14904646	251734643	1	5587340	8351519
small	28.97	31.55	15885643	65463253	64005618	653512565	1	57302561	74336271
med	92.05	130.66	33483623	96175011	342082005	2340006494	5	509617907	1140684391
large	401.87	1420.9	82314606	98567576	1515133709	8474145681	27	6172541037	29601868455
long	49.32	68.9	19424358	73358199	196930536	1284987449	2	261181555	503123286
tall	82.6	103.13	42995033	149761200	198002012	1906343791	1	485119109	586037805
wide	104.04	120.3	37394981	130500684	471356916	2839547318	1	332983446	408917456

Table B.10: Garalgly-**bi-seq** analysis

Query type	Count (s)	Report (s)	Chains found	Binary search steps	Total Points found
rand	58.18	92.83	37333882	2360069578	1008288397
tiny	50.49	51.75	1246164	2223300028	8351519
small	52.59	56.09	6377855	2236494978	74336271
med	60.24	99.62	44917225	2396184488	1140684391
large	116.14	1110.72	303632232	3838818504	29601868455
long	57.29	74.84	28663775	2309536315	503123286
tall	56.27	77.09	30826488	2326014321	586037805
wide	55.16	69.66	24437704	2303918176	408917456

Table B.11: Garalgly-**lg** analysis

Query type	Count (s)	Report (s)	Nodes visited	Nodes from subtree	Node ratio	Points from subtree	Total Points found	Points ratio
rand	47.93	171.48	1929518758	118393467	6.14%	3471658508	4007810458	86.62%
tiny	2.75	4.57	108946850	4081994	3.75%	33933427	56321844	60.25%
small	7.43	15.51	306042679	14053528	4.59%	186526019	258356689	72.2%
med	32.05	100.02	1351827991	74001582	5.47%	1839948291	2208888875	83.3%
large	189.57	1411.56	8289099139	600789292	7.25%	35977515122	38821847881	92.67%
long	42.08	118.16	1688539244	103078222	6.1%	2015515133	2429830204	82.95%
tall	25.1	72.47	1093823780	57575899	5.26%	1245555549	1519737827	81.96%
wide	26.25	62.48	1087045383	55185398	5.08%	866096441	1157068453	74.85%

Table B.12: Movies-**imp-kd** analysis

Query type	Count (s)	Report (s)	xy-plane chains found	Total xy bi-searches	xz-plane chains found	Total xz bi-searches	Bi-search_I	Total Points found
rand	63.69	185.34	46131589	143907945	121748695	2112695963	1118028838	4007810458
tiny	11.8	13.48	11856878	57965213	10412094	303831709	251803490	56321844
small	22.1	30.71	21357780	82374879	26791987	649589178	483014689	258356689
med	54.15	122.4	44240009	122444103	96977568	1885828842	1098427538	2208888875
large	167.84	1382.91	111240006	139247198	480721504	7221436206	2218254017	38821847881
long	36.96	111.47	25739029	91946035	87495500	1341405756	533660504	2429830204
tall	64.76	112.18	60579675	203766281	80740285	2005936489	1427936350	1519737827
wide	76.17	112.86	74903505	329734283	257057077	2874361966	2218972809	1157068453

Table B.13: Movies-**bi-bi** analysis

Query type	Count (s)	Report (s)	xy-plane chains found	Total xy bi-searches	xz-plane chains found	Bi-search_I	Best -dis	Linear scan	Total Points found
rand	61.66	189.25	46131589	143907945	121748695	1118028838	5	1227673872	4007810458
tiny	11.29	13.52	11856878	57965213	10412094	251803490	1	48665662	56321844
small	21.35	30.56	21357780	82374879	26791987	483014689	3	126605741	258356689
med	52.83	124.67	44240009	122444103	96977568	1098427538	4	810722517	2208888875
large	177.31	1437.19	111240006	139247198	480721504	2218254017	30	11436314476	38821847881
long	34.68	112.08	25739029	91946035	87495500	533660504	3	974707141	2429830204
tall	63.51	114.12	60579675	203766281	80740285	1427936350	3	645513347	1519737827
wide	73.8	112.55	74903505	329734283	257057077	2218972809	3	540348186	1157068453

Table B.14: Movies-**bi-seq** analysis

Query type	Count (s)	Report (s)	Chains found	Binary search steps	Total Points found
rand	126.31	261.68	85865549	5004278867	4007810458
tiny	103.62	105.9	3738538	4700854538	56321844
small	106.35	114.41	11352906	4723128512	258356689
med	119.35	187.44	55860019	4897622326	2208888875
large	194.57	1485.35	348268777	6323157245	38821847881
long	125.62	210.27	66416546	4918873893	2429830204
tall	113.28	159.99	53544189	4865032677	1519737827
wide	112.91	149.28	53169912	4870732118	1157068453

Table B.15: Movies-**lg** analysis

Query type	Count (s)	Report (s)	Nodes visited	Nodes from subtree	Node ratio	Points from subtree	Total Points found	Points ratio
rand	49.83	187.78	2004248790	124772790	6.23%	3658899417	4231156591	86.48%
tiny	2.57	4.25	100972387	3645048	3.61%	27846345	48941966	56.9%
small	7.14	14.65	293702890	13154022	4.48%	160813403	231703690	69.4%
med	33.81	102.76	1435044546	77873437	5.43%	1793277514	2201334451	81.46%
large	186.68	1360.03	8341287948	605740306	7.26%	34357066173	37358273481	91.97%
long	45.4	137.47	1795038722	113742425	6.34%	2268619383	2717713204	83.48%
tall	25.3	68.6	1104456243	57109730	5.17%	1065153094	1354211598	78.65%
wide	25.87	63.52	1073325391	54331978	5.06%	882894262	1174777244	75.15%

Table B.16: Electronics-**imp-kd** analysis

Query type	Count (s)	Report (s)	xy-plane chains found	Total xy bi-searches	xz-plane chains found	Total xz bi-searches	Bi-search_I	Total Points found
rand	58.51	188.07	43034438	130367634	116405862	2030665235	1067049250	4231156591
tiny	10.76	12.37	10488261	53076900	9796040	287492084	238415962	48941966
small	21.32	28.52	19576146	76350256	26768379	635097454	473424795	231703690
med	53.47	120.66	42643696	111197160	96508269	1865785741	1084449056	2201334451
large	150.02	1349.35	102896939	115068365	436456418	6706276067	2008825626	37358273481
long	36.05	118.72	23877742	85315834	86308972	1307680711	517881259	2717713204
tall	65.78	107.54	57450524	189964866	82526283	2065377394	1476910761	1354211598
wide	69.64	105.97	66879797	282517730	233165973	2641648550	2022795709	1174777244

Table B.17: Electronics-**bi-bi** analysis

Query type	Count (s)	Report (s)	xy-plane chains found	Total xy bi-searches	xz-plane chains found	Bi-search_I	Best -dis	Linear scan	Total Points found
rand	60.01	193.22	43034438	130367634	116405862	1067049250	5	1258588140	4231156591
tiny	10.5	12.39	10488261	53076900	9796040	238415962	1	41778470	48941966
small	20.58	28.69	19576146	76350256	26768379	473424795	3	116579138	231703690
med	52.01	122.82	42643696	111197160	96508269	1084449056	5	786706618	2201334451
large	153.86	1357.18	102896939	115068365	436456418	2008825626	30	10109376443	37358273481
long	34.58	120	23877742	85315834	86308972	517881259	3	1065934910	2717713204
tall	64.11	109.41	57450524	189964866	82526283	1476910761	4	561107587	1354211598
wide	67.57	105.62	66879797	282517730	233165973	2022795709	3	535280160	1174777244

Table B.18: Electronics-**bi-seq** analysis

Query type	Count (s)	Report (s)	Chains found	Binary search steps	Total Points found
rand	116.53	256.97	80287060	4732988348	4231156591
tiny	95.75	98.61	3345820	4438074953	48941966
small	98.6	107.11	9935269	4459435393	231703690
med	110.43	181.99	50507056	4639905025	2201334451
large	170.67	1385.54	291367748	5842952465	37358273481
long	116.71	210.2	77053205	4677006195	2717713204
tall	105.15	148.52	44462037	4588552292	1354211598
wide	104.99	143.49	44426159	4598965266	1174777244

Table B.19: Electronics-**lg** analysis

Query type	Count (s)	Report (s)	Nodes visited	Nodes from subtree	Node ratio	Points from subtree	Total Points found	Points ratio
rand	34.87	119.2	1449507053	88945560	6.14%	2280675110	2690820885	84.76%
tiny	1.78	2.77	72835061	2423869	3.33%	15675943	29916591	52.4%
small	4.91	9.47	207028860	9566576	4.62%	93348924	142064090	65.71%
med	24.01	68.69	1043992023	61276996	5.87%	1143805959	1437435789	79.57%
large	137.69	879.68	6233368513	515816691	8.28%	21581803972	23955197149	90.09%
long	31.14	84.89	1267842307	70996319	5.6%	1388471428	1700575275	81.65%
tall	17.78	46.04	795961471	51753686	6.5%	692118133	895381126	77.3%
wide	17.14	38.6	733933436	32204882	4.39%	482165271	677117563	71.21%

Table B.20: CDs-**imp-kd** analysis

Query type	Count (s)	Report (s)	xy-plane chains found	Total xy bi-searches	xz-plane chains found	Total xz bi-searches	Bi-search_I	Total Points found
rand	44.47	129.54	40821209	125516990	100224570	1611052652	894931703	2690820885
tile	3.43	3.62	5152667	38265097	3573879	100223501	92914894	5115278
tiny	8.77	9.75	10704073	51509192	8565101	228351971	195870900	29916591
small	16.43	20.97	19457471	72254909	22013965	488625012	379209407	142064090
med	39.69	84.79	40191450	107694340	80121197	1445090677	880488429	1437435789
large	113.8	868.51	97247362	120024939	398457265	5597002766	1861140709	23955197149
long	27.05	80.48	23387896	79972878	73581654	1029861839	425634783	1700575275
tall	46.6	74.82	53241774	168709043	64016996	1482326334	1092915270	895381126
wide	56.36	77.84	61076429	240889377	212289413	2189237567	1770068129	677117563

Table B.21: CDs-**bi-bi** analysis

Query type	Count (s)	Report (s)	xy-plane chains found	Total xy bi-searches	xz-plane chains found	Bi-search_I	Best -dis	Linear scan	Total Points found
rand	45.03	130.12	40821209	125516990	100224570	894931703	5	871966640	2690820885
tiny	8.57	9.74	10704073	51509192	8565101	195870900	1	24107676	29916591
small	15.97	21.04	19457471	72254909	22013965	379209407	1	125344014	142064090
med	38.81	85.44	40191450	107694340	80121197	880488429	3	614244459	1437435789
large	109.78	874.19	97247362	120024939	398457265	1861140709	31	7932151034	23955197149
long	25.55	86.75	23387896	79972878	73581654	425634783	3	698846457	1700575275
tall	45.72	80.86	53241774	168709043	64016996	1092915270	3	399156318	895381126
wide	54.87	79.99	61076429	240889377	212289413	1770068129	3	332240915	677117563

Table B.22: CDs-**bi-seq** analysis

Query type	Count (s)	Report (s)	Chains found	Binary search steps	Total Points found
rand	68.11	157.14	44954831	2779021874	2690820885
tiny	53.95	55.66	2411147	2592753925	29916591
small	55.55	60.4	7094779	2607595442	142064090
med	62.33	106.56	33286297	2721749350	1437435789
large	94.02	896.49	188462518	3575827478	23955197149
long	69.56	127.91	39917171	2752498568	1700575275
tall	58.77	85.46	27424580	2688512251	895381126
wide	58.92	80.28	28591671	2688390448	677117563

Table B.23: CDs-**lg** analysis

Query type	Count (s)	Report (s)	Nodes visited	Nodes from subtree	Node ratio	Points from subtree	Total Points found	Points ratio
rand	32.97	114.99	1399676958	88059261	6.29%	2245054703	2639375760	85.06%
tiny	1.69	2.7	70130960	2415372	3.44%	17090525	31155828	54.85%
small	4.65	9.06	198891734	9352948	4.7%	93723235	140607534	66.66%
med	22.36	66.15	991380796	61212003	6.17%	1126199050	1406367169	80.08%
large	125.19	805.72	5701057099	489683726	8.59%	20126227925	22270828738	90.37%
long	29.6	83.19	1231537571	70657021	5.74%	1404158971	1711647362	82.04%
tall	14.91	35.36	671520727	40550104	6.04%	489331138	654064276	74.81%
wide	16.7	38.05	727155475	34122348	4.69%	490236522	685179912	71.55%

Table B.24: Kindle-**imp-kd** analysis

Query type	Count (s)	Report (s)	xy-plane chains found	Total xy bi-searches	xz-plane chains found	Total xz bi-searches	Bi-search_I	Total Points found
rand	26.36	107.09	24789186	74719764	55725601	1010499126	542174812	2639375760
tiny	5.34	6.35	7214993	33080924	4964438	151164344	126825069	31155828
small	10.01	14.29	12819956	44658703	12540893	322026973	245487407	140607534
med	25.65	66.53	24314563	62873309	44497840	935329045	559325410	1406367169
large	66.38	737.09	56573053	69688530	231241781	3393553432	1088607262	22270828738
long	20	70.17	15209047	49111612	43370149	656933117	254357400	1711647362
tall	27.08	47.38	29823393	90825878	32409945	871510637	642841692	654064276
wide	37.96	57.98	42010615	187402077	118182169	1415199837	1115423487	685179912

Table B.25: Kindle-**bi-bi** analysis

Query type	Count (s)	Report (s)	xy-plane chains found	Total xy bi-searches	xz-plane chains found	Bi-search_I	Best -dis	Linear scan	Total Points found
rand	27.8	110.45	24789186	74719764	55725601	542174812	5	728694232	2639375760
tile	2.1	2.29	2875273	25898598	2018724	57890939	1	3585351	4686627
tiny	5.22	6.33	7214993	33080924	4964438	126825069	3	16630394	31155828
small	9.71	14.44	12819956	44658703	12540893	245487407	3	65722641	140607534
med	23.99	68.12	24314563	62873309	44497840	559325410	5	448998781	1406367169
large	64.38	755.65	56573053	69688530	231241781	1088607262	24	4734721180	22270828738
long	16.11	69.55	15209047	49111612	43370149	254357400	5	512243759	1711647362
tall	24.97	46.28	29823393	90825878	32409945	642841692	3	273470236	654064276
wide	35.54	57.74	42010615	187402077	118182169	1115423487	3	299099117	685179912

Table B.26: Kindle-**bi-seq** analysis

Query type	Count (s)	Report (s)	Chains found	Binary search steps	Total Points found
rand	48.4	133.96	38334005	2146332262	2639375760
tiny	38.82	40.12	1976026	1987519117	31155828
small	39.99	44.2	5699934	2000155612	140607534
med	45.34	88.07	28475296	2102140927	1406367169
large	68.18	796.47	154207100	2788257446	22270828738
long	49.94	107.14	34388770	2122803846	1711647362
tall	41.98	61.3	18400440	2052706706	654064276
wide	42.74	63.58	25113617	2075699412	685179912

Table B.27: Kindle-**lg** analysis