A Fault Tolerant Data Structure for Peer-to-Peer Range Query Processing

by

Zahra Mirikharaji

TR15-237, August 2015

This is an unaltered version of the author's MCS thesis
Supervisor: Bradford G. Nickerson

Faculty of Computer Science
University of New Brunswick
Fredericton, N.B. E3B 5A3
Canada

Phone: (506) 453-4566
Fax: (506) 453-3566
E-mail: fcs@unb.ca
http://www.cs.unb.ca

Copyright © 2015 Zahra Mirikharaji

Abstract

We present a fault tolerant dynamic data structure based on a constant-degree Distributed Hash Table called FissionE that supports orthogonal range search in d-dimensional space. A publication algorithm, which distributes data objects among all nodes in the network is described, along with a search algorithm that processes range queries and reports all objects in range to the query issuer. The worst case orthogonal range search cost in our data structure with n nodes is $O(\log n + m)$ messages plus reporting cost, where m is the minimum number of nodes intersecting the query. We have proved that in our data structure the cost of reporting data in range to the query issuer is $\sum_{i=1}^{m} \lceil \frac{K_i}{B} \rceil O(\log n) \in O((\frac{K}{B} + m) \log n)$ messages, where K is the number points in range, K_i is the number of points in range stored in node i, and B is the number of points fitting in one message. Storing d copies of each data objects on d different nodes provides redundancy for our scheme. This redundancy permits completely answering a query in the case of simultaneous failure of $d - 1$ nodes. Results of our experimental simulation with up to 12,288 nodes show the practical application of our data structure.

Dedication

This thesis is dedicated to my beloved spouse, Mohammadreza, without whose love and support I would never have made it this far. I also wish to dedicate this thesis to my parents who have supported me in each step of my life.

Acknowledgements

I would like to express my special appreciation and thanks to my supervisor, Dr. Bradford Nickerson for his continuous support and encouragement throughout this thesis. Without his supervision and constant guidance this dissertation would not have been possible. I take this opportunity to express gratitude to all of the Faculty of Computer Science faculty members for their help and support. Also, I would like to acknowledge the support of the Natural Sciences and Engineering Research Council (NSERC) of Canada and the UNB Faculty of Computer Science.

Table of Contents

List of Tables

List of Figures

List of Algorithms

List of Symbols, Nomenclature or Abbreviations

B	The number of points that fit in one message
d	Number of data dimensions
DHT	Distributed Hash Table
$DSDS$	Distributed Spatial Data Structure
K	Number of points reported in range query Q
K_i	The number of points in range on node i
$K(b,k)$	A Kautz graph whose nodes identifiers are Kautz strings with base b and length k
$\bar{K}_Q$	Average search cost in messages
L_i	The lowerbound of the entire space in dimension i
m	minimum number of nodes intersecting a range query Q
n	Number of nodes in the network
$\bar{n}_Q$	Average number of nodes intersecting queries
N	Number of data points in the data structure
O	An object in d-dimensional space
P	The set of all points in the data structure
$P2P$	Peer-to-peer
Q	Range query
Q_{jL}	The lower bound of the query in dimension j

Q_{jU}	The upper bound of the query in dimension j
r	A range
$\mathbf{R}$	The set of all ranges
$R(d,k)$	A rectangular range in d-dimensional space with k dimensions having finite interval where $0 \leq k \leq d$
R_M	Message cost ratio
RSG	non-redundant rainbow skip graph
S_i	The set of points on node i
$\bar{T}$	Average reporting cost in messages
U_i	The upperbound of the entire space in dimension i
w	a subspace in d-dimensional space

Chapter 1

Introduction

1.1 Range Searching

Range searching is an algorithmic problem in the area of computational geometry with applications in geographic information systems (GIS), data bases, graphics and subroutines of other problems related to range queries. Before presenting a formal definition of the range search problem, we need to speak about range and its various types.

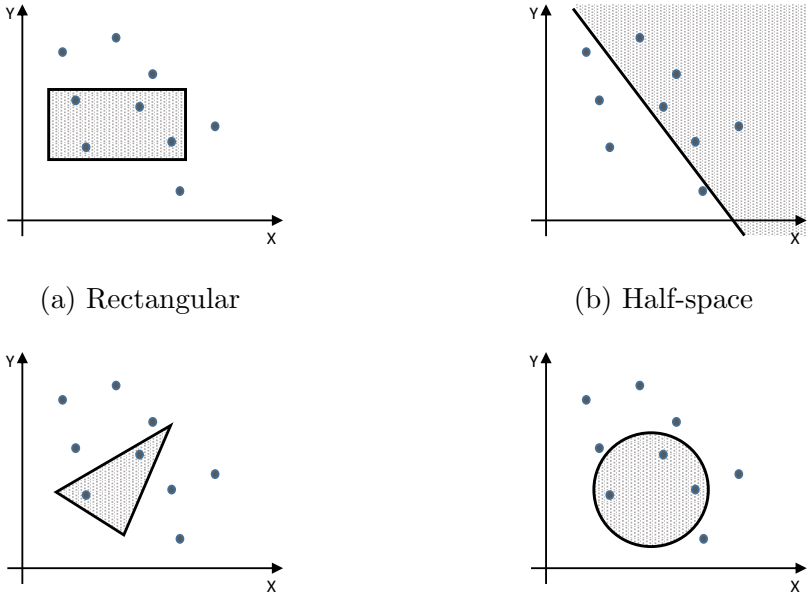
Let R^d be d-dimensional Euclidean space and $\mathbf{R}$ be a group of subsets of R^d. Each member of $\mathbf{R}$ is called a range. Some typical kinds of ranges are as follows:

1. Orthogonal ranges (R_{orthog}) are axis parallel boxes which all sets are in the form of $\prod_{i=1}^{d}[a_i, b_i]$ where $a_1, b_1, ..., a_d, b_d \in \mathbb{R}$.

2. Half space range (R_{half}) is the set of all halfspaces in R^d space.

3. Simplex range is the set of all simplices in R^d where a d-simplex is a d-dimensional polytope which is the convex hull of its $d + 1$ vertices.

4. Ball range (R_{ball}) is the set of all balls in R^d [18].



(a) Rectangular (b) Half-space

(c) Simplex (d) Ball

Figure 1.1: Examples of range queries in two dimensions.

Figure 1.1 shows examples of **R** for these cases in 2-dimensional space. Figure 1.1(a) shows a specific example of rectangular ranges. To define a general form, let $R(d, k)$ be a rectangular range in d-dimensional space with k dimensions having a finite interval where $0 \leq k \leq d$ [1]. According to this definition, Figure 1.1(a) shows a rectangular range $R(2, 2)$. Figures 1.2 and 1.3 show various rectangular ranges respectively in 2-d and 3-d space.

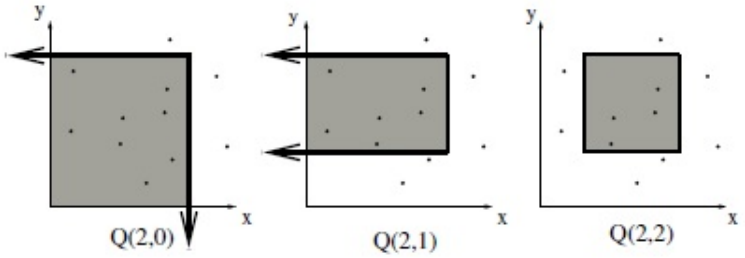


Figure 1.2: 2-dimensional rectangular rangess (from [1]).

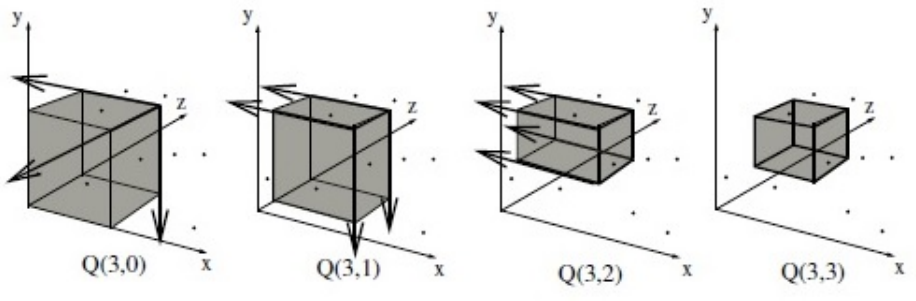


Figure 1.3: 3-dimensional rectangular range (from [1]).

Let P be a set of N points in R^d and r be a range $\in \mathbf{R}$. The range searching problem is to design an efficient algorithm which reports all points of P lying in r.

Reporting points of the set P which intersect with range r (a range-reporting query) is just one of the possible range search problems. Counting points lying in a given range is called a range-counting query. Sometimes we just want to check the emptiness of $P \cap r$ (range emptiness query). Another case of range searching problems is optimization range query. In this case, we are looking for a specific point with certain properties among points intersecting r [19].

In reality, spatial data objects usually occupy areas in multi-dimensional space. In 2-dimensional space, these objects are represented by points, triangles or polygons. In range searching, when objects are represented by triangles or polygons, we want to determine all the objects intersecting a given range. For example, on a map, we may want to find all counties within 20 km of our location. Different representations of objects use various approaches for solving range search problems. In this work, we assume that all objects are represented by points.

1.2 Motivation and Applications

Different types of range-search problems are motivated by different practical applications. In the following, we give a simple example which is presented in many range search references for explaining the application of point range search.

Consider a Company employees data table. In this table, each record is related to one employee and can be interpreted as a point in d-dimensional space where each dimension corresponds to one field of the table. For example, if we want to report all employee born between 1950 and 1955 who earn between \$3,000 and \$4,000 per month and have 2, 3 or 4 children, we can use a 3-dimensional space where each point represents an employee. Figure 1.4 shows this example. We can see that one coordinate is devoted to date of birth, the second one to salary and the last one to number of children. All points lying in the grey parallel-sided box are the answer to this orthogonal range query [9].

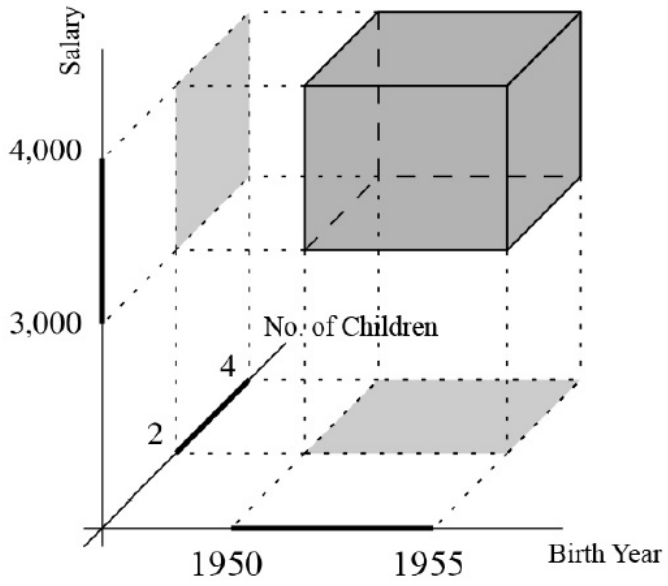


Figure 1.4: Orthogonal range in 3-dimensional space (from [9]).

Locating an airport nearby a defective airplane in emergency conditions can be an application of ball range search. In ball range search, all the ranges are in the form of a ball range, and we are looking for all points within distance r from one specific point that is moving. More important than direct applications, range search has applications as subroutines in other geometric algorithms. For example, in the facility location problem, we are looking for optimal placement of facilities in an area to minimize some distances and costs like transportation cost. Range searching algorithms may be used as subroutines in optimal placement of facilities algorithms.

In computational geometry, the range searching problem can be generalized to merge all kinds of range searching and making a unified problem. Given a set of points P, we assign a weight $w(p) \in S$ to each point $p \in P$. If $(S, +)$ is a commutative semigroup, for any subset $P' \subseteq P$, $w(P') = \Sigma_{p \in \mathbb{P}'} w(p)$. For each query r, our purpose is computing $w(P \cap r) = \Sigma_{p \in \mathbb{P} \cap \setminus} w(p)$ by this definition all types of range searching problems can be defined. For example, for range counting queries, S is the set of integers Z and $+$ is the standard integer addition. By setting $w(p) = 1$ for each $p \in P$, the range counting problem has been defined. For emptiness query, the semigroup is $(\{0, 1\}, or)$ and $w(p) = 1$ for each $p \in P$ [2].

1.3 Computing Models

Assuming a *Random Access Machine* (RAM) model, the set of all points, P and the range r are available in main memory at the same time. The simplest range search algorithm goes through all points of P one by one and reports points intersecting r. This "linear search" algorithm uses $\Theta(N)$ space and $\Theta(N)$ time. Typically, however, the point set P is given in advance and we want to preprocess points into a data structure to answer different queries as efficiently as possible.

The performance of a data structure is affected by three elements with different levels of importance. The first one is the preprocessing cost, the cost that we need for constructing a data structure. Preprocessing time is less important than the search cost and storage space. The size of the data structure or storage space includes the original data size N plus auxiliary information required to create the data structure. The search cost of a range-reporting query on any reasonable machine depends on the output size, so the search cost consists of two parts. The first part depends on the data structure and the second part is the output sensitive part, and is a function of K, the number of points found in range.

Under the RAM model of computation, the run time of an algorithm is measured by counting the number of steps. Addition, multiplication, subtraction, division and comparison are considered simple operations which take exactly one time step, and each memory cell is accessible in constant time [2]. In the RAM model, range search cost is the time spent to answer a query and depends on the number of points, N and d, the number of dimensions.

In the Pointer Machine (PM) model of computation, each memory cell can be accessed through a set of pointers. The RAM and pointer machine models take no notice of whether an item is in cache or on the disk, which simplifies the analysis. If data is too large to fit into main memory, data must be stored in secondary memory and portions of it must be moved into main memory when needed. The model is known as the I/O model, and query complexity measures only the number of I/Os [2]. The I/O model is important as the time for one disk access is approximately 1×10^6 more than the time for one memory access. In the I/O efficient model, search cost is measured by the number of disk blocks accessed during the search process and depends on N, d and B, where B is the number of points in one disk block.

When secondary memory does not have enough space to store data, a distributed computing model is used to spread out data among different nodes of a network. Another motivation behind the distributed model are scalability and increasing reliability of data access by having multiple copies of data stored at different locations. In addition, different organizations of data at different locations can provide flexibility in distributed data structures. Two popular types distributed systems are the client-server or hierarchical model and peer-to-peer (P2P) networks. Peer-to-peer networks are a class of networks without any central point that use distributed resources to achieve a specific purpose. Each node, called a peer in the network, is a consumer and supplier of data. Furthermore, they are more reliable and scalable in comparison with client-server systems, since there is no single point of failure in peer-to-peer systems and every node can issue a query. Such characteristics of peer-to-peer networks make them a natural choice for use in our research on the distributed range search problem.

In the distributed computing model, it is assumed that the cost of sending a message is higher than the cost of an I/O, so the search cost is the number of required messages exchanged between nodes to answer a query. For range reporting, query cost also depends on the number of reported points, K, in addition to other parameters. It should be mentioned that for dynamic data structures, the cost to update the data structure (insertion and deletion) may be important [14].

1.4 Objectives

In this thesis, we investigate a spatial data structure supporting efficient orthogonal range search of d-dimensional points using a distributed computing model. Other desired features of such a data structure include:

- Node size and high efficiency: Can the use of a constant-degree graph rep-

resenting the network topology achieve the lower bound range search cost of $\Omega(\log n) + m - 1$ messages, where m is the number of nodes intersecting the query [15], for general range search schemes on constant-degree networks?

- Fault tolerance: are we able to automatically adjust to the failure of some nodes and repair the structure at relatively small cost?

- Dynamic P2P networks: can we achieve dynamic updates to our data structure to permit addition or deletion of nodes or points as needed?

Chapter 2

Background

2.1 Peer-To-Peer Networks

Peer-to-peer (P2P) networks are a class of networks without any central point or hierarchical organization that uses distributed resources to achieve a specific purpose. Various applications exchange information among a set of distributed machines. Advantages of peer-to-peer applications such as self organization, redundant data storage, scalability, fault tolerance and efficient search have attracted significant attention in recent years. Communication in peer-to-peer networks is done in a distributed manner and all the participants are peers. A peer-to-peer network is illustrated in Figure 2.1.

2.2 Peer-To-Peer Indexing Data Structures

Lua et al. in *A Survey and Comparison of Peer-to-Peer Overlay Network Schemes* [17] classify peer-to-peer overlay networks into two classes; *Structured* and *Unstructured*. In structured peer-to-peer networks, the topology of nodes in the network and data placement are specifically organized to be able to efficiently search for data. In comparison, in unstructured peer-to-peer networks there is no control on

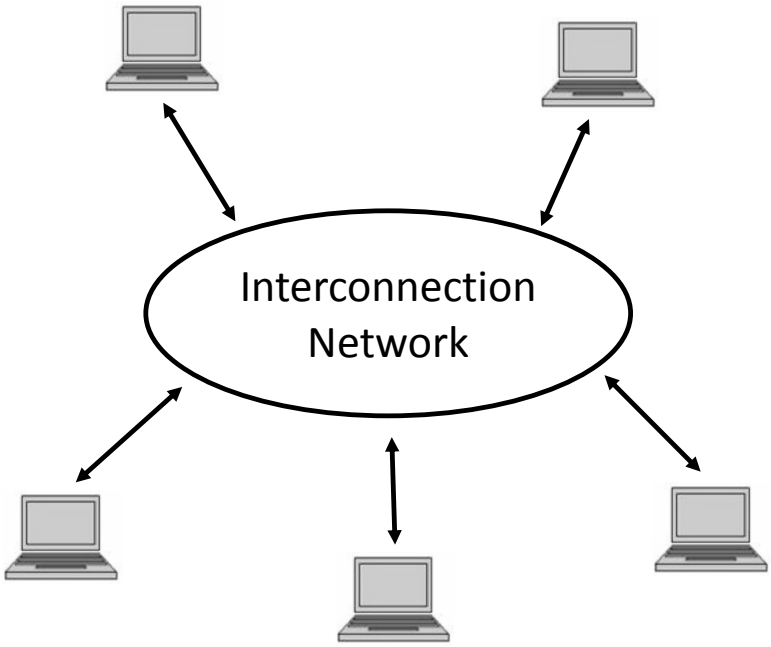


Figure 2.1: A peer-to-peer network.

the network topology and data placement, and the connections between nodes are arbitrarily formed. To search an unstructured peer-to-peer network for a piece of data, the query is flooded over the network.

Many structured P2P systems like Chord [26], Tapestry [31], Pastry [22], CAN [20] and FissionE [16] use the Distributed Hash Table (DHT) [21] to distribute data objects deterministically at the peers and look them up with the data object's unique key. DHT-based systems use hashing to assign IDs to the peers and each peer is responsible for a small specific segment of the data namespace. Peer-to-peer networks support routing of a key/value pair to a particular peer with certain guarantees (e.g., logarithmic number of hops for searching on average). When peers join or leave the network, the distribution of data to nodes is reassigned among the peers.

DHT schemes are normally capable of processing exact match searches. More complex searches such as orthogonal range search have vast applications in areas like data management systems and computer aided design. A number of recent papers has investigated DHTs to process range queries. Zhang et al. [30] classify DHT-based techniques for range queries into two groups; layered indexing and customized indexing. The layered indexing techniques use DHTs for the underlying topology and message routing algorithm without any modification to answer range queries. Our work falls into this category. In contrast, customized indexing uses a custom-designed P2P overlay or modifies an existing P2P overlay network to support range search.

In layered indexing, Gupta et al. [12] use a probabilistic scheme that relies on locality sensitive hashing to ensure that, with high probability, similar ranges are mapped to the same node. However, these methods can only help to get approximate answers for one dimensional range queries on Chord [26]. Squid [23] and DCF-CAN [3] use space-filling curves (SFC) to map multi-dimensional keys to the peers. Space-filling curves are locality preserving, but they provide less efficient range queries, because a single range query may cover several parts of the curve, which have to be queried separately.

Network routing table size or degree is an important measure in peer-to-peer networks. Most distributed indexing structures supporting range search don't work on a constant-degree graph. Among the existing schemes in layered indexing supporting range queries, Armada [15] and Distributed Spatial Data Structure (DSDS) [7] work on top of a constant-degree graph.

Armada [15] provides a higher efficiency in terms of query delay and number of required messages. Armada is a range query scheme for d-dimensional space running

on top of a constant degree FissionE [16] DHT. For Armada, Li et al. [15] have proven that the average message cost of one dimensional queries in PIRA (PrunIng Routing Algorithm) is about $\log n + 2m - 2$ where m is the number of nodes intersecting the query. This average cost assumes the data is distributed in a uniform random fashion. For multi-dimensional indexing, Li et al. [15] have not presented any guarantee on the number of messages required to answer a d-dimensional orthogonal range. They present a simulation to show that the average message cost of MIRA is about $\log n + 4m - 1$ messages. Since Armada is the most similar work to our proposed research, in section 2.3, we present an overview of data distribution and searching in Armada.

Bisadi et al. in [7] present a peer-to-peer distributed spatial data structure (DSDS) that employs a non-redundant rainbow skip graph [10] to route the messages. The worst case orthogonal range search cost in a 2-dimensional DSDS with n nodes is $O(n)$ messages plus reporting cost. A complete backup copy of data points stored in the other nodes of network provides redundancy for DSDS. A routing recovery algorithm for DSDS is presented in [6] that requires $O(\log n)$ messages to recover the routing information after failure of one node . The backup copy and the routing recovery algorithm permit completely answering a query in the case of single node failure. Since our data distribution algorithm is inspired from the slab partitioning of space in DSDS, in section 2.4 we further explain the DSDS.

In customized indexing, the skip graph [4] and SkipNet [13] are P2P networks having $O(\log n)$ degree that can just support one dimensional orthogonal indexing. Family trees [29] and rainbow skip graph [10] both are constant degree. They also support just one dimensional range queries. The P-tree proposed in [8] uses the distributed version of a B+-tree to build an indexing structure for range queries. The P-tree requires $O(b \log_b n)$ space in each peer where b is the order or branching

factor of tree.

Mercury [5], Znet [24] and MIDAS [27] provide indexing schemes for multi-dimensional space. Mercury [5] provides multi-dimensional range queries by indexing the data set along each dimension. The latency of the message routing algorithm in Mercury [5] is $\log^2 \frac{n}{k}$ when each node maintains k links to other nodes. MIDAS [27] resolves the request in $O(\log n)$ hops when each peer's degree is $O(\log n)$. In Znet [24], SFCs (Space Filling Curves) are used to map multi-dimensional space to 1-dimensional space, and skip graphs [4] are extended for query routing, with each node maintaining $O(\log n)$ states.

2.3 Armada

Armada [15] is a delay bounded structured peer-to-peer network that uses FissionE [16] as an underlying DHT to organize nodes in an overlay network. FissionE is a constant degree peer-to-peer network with an average degree of 4 and an average path length of about $\log n$. Since FissionE only supports exact-match queries of distributed objects on nodes, Armada which is built on top of FissionE is designed to provide support range queries in 1 and d-dimensional space. In this section, we first give an overview of FissionE and then we discuss the components of Armada. In the next chapter, the FissionE distributed hash table (DHT) scheme is explained in detail as the base of our proposed method.

The topology of FissionE is based on the topology of Kautz graphs. The Kautz graph $K(b, k)$ is a directed graph with $b^{k-1}(b + 1)$ vertices whose identifiers are Kautz strings. The string $u_1u_2...u_k$ of length k and base b is a Kautz string where $u_i \in \{0, 1, 2, ..., b\}$ and $u_i \neq u_{i+1}$ $(1 \leq i \leq k - 1)$. The connection of nodes in a Kautz graph is dependent on each node's Kautz string. Each node $U = u_1u_2...u_k$ of

a graph has an out-degree b to the nodes $V = u_2u_3...u_k\alpha$ where $\alpha \in \{0, 1, ..., b\}$ and $\alpha \neq u_k$.

Li et al. in [15] have proposed two main components for Armada. The first part is an *order-preserving naming* algorithm to assign objectIDs from the Kautz namespace to the objects and then distribute them on corresponding nodes in such a way that data locality is preserved. The second part is a *range query processing* algorithm that efficiently forwards queries to the nodes intersecting a range query. In the following, we discuss order-preserving naming and range query processing algorithms for d-dimensional range queries.

2.3.1 Multiple_hash Algorithm

In order to support range query, naming algorithms that generate IDs for objects must keep the locality of data values in different dimensions (attribute values). Order-preserving algorithms in Armada assign adjoining objectIDs in Kautz namespace to objects with close attribute values.

Before discussing the order-preserving naming algorithm, MULTIPLE_HASH for d-dimensional objects, we give some definitions. In addition, we assume that the $\prec$ symbol provides the "no more than" relation between Kautz strings in lexicographical order. We assume that each object has d attributes, $A_0, A_1, ..., A_{d-1}$ and the values of objects are in d-dimensional subspace $w = <r_0, r_1, ..., r_i, ..., r_{d-1}>$ where r_i is the domain of values of attribute A_i.

Definition 2.3.1. *The Kautz region* $[\![\alpha, \beta]\!]$ *is the subset of* $KautzSpace(2, k)$ *which includes all strings* s *that are in* $KautzSpace(2, k)$ *and* $\alpha \prec s$ *and* $s \prec \beta$.

Definition 2.3.2. *For two objects in d-dimensional subspace w,* $\delta_1 = <u_0, u_1, ..., u_{d-1}>$ *and* $\delta_2 = <v_0, v_1, ..., v_{d-1}>$, $\delta_1 \lhd \delta_2$ *if for each* $0 \leq i < d-1$, $u_i \leq v_i$.

Definition 2.3.3. *Subjective function F from multiple dimensional space D to Kautz namespace V is multiple attribute partial order preserving iff for any* δ_1 *and* δ_2 *in D, if* $\delta_1 \lhd \delta_2$ *then* $F(\delta_1) \prec F(\delta_2)$.

The partition tree presented in [15] describes the MULTIPLE_HASH algorithm to assign partial order preserving objectIDs to objects. Partition tree $P(2, k)$ partitions the entire d-dimensional space w into smaller subspaces and maps the subspace in each leaf node to a specific Kautz string. A Partition tree has $k + 1$ levels and the root node represents the entire space w. The root node has three children while other intermediate nodes have two children. The label of edges in a partition tree depends on the label of the parent node. Edge labels can be 0, 1 or 2, increasing from left to right. All nodes have a specific label at level j of the partition tree, and the space covered by each node is divided into two subspaces along the ith attribute where $i = j \bmod d$ with d the number of attributes for each object. Figure 2.2 illustrates an example of a partition tree $P(2, 4)$ for 2-dimensional space $< [0, 6], [0, 8] >$.

It is worth noting that in the MULTIPLE_HASH algorithm, we don't need to construct the partition tree. In other words, each node of a partition tree doesn't correspond to any node of a network. The partition tree is a model presented to describe the MULTIPLE_HASH algorithm. As explained, the MULTIPLE_HASH algorithm maps the multiple dimensional space to Kautz strings in $KautzSpace(2, k)$. So to publish objects in Armada, we first provide the ObjectID using MULTIPLE_HASH algorithm. Then we use the FissionE routing approach to find the unique node whose nodeID is the prefix of the ObjectID. The pseudocode of the MULTIPLE_HASH algorithm is

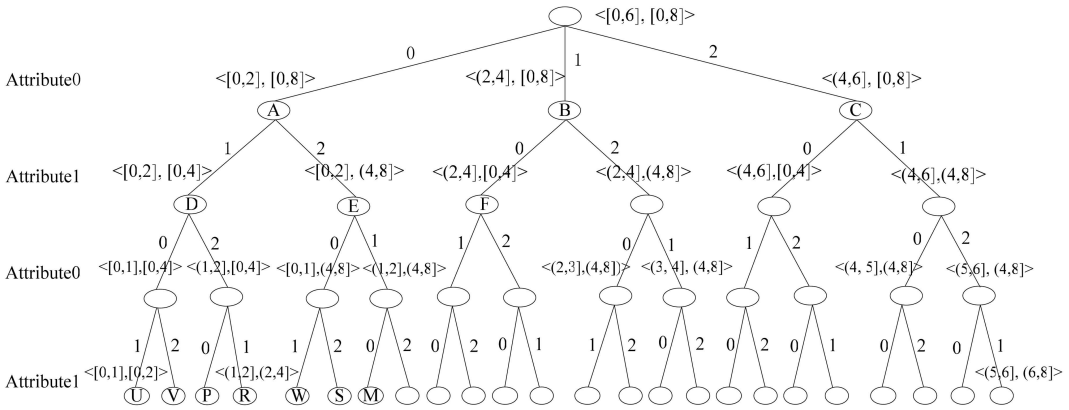


Figure 2.2: An example of partition tree $P(2,4)$ for 2-dimensional space $< [0,6],[0,8] >$ (from [15]).

shown in Algorithm 1.

2.3.2 MIRA Algorithm

It is straightforward to prove that the MULTIPLE_HASH algorithm is a multiple attribute partial order preserving function from multiple attribute space to the $KautzSpace(2,k)$. When a node publishes a range query $Q = <[x_0, y_0]$, $..., [x_i, y_i], ..., [x_{d-1}, y_{d-1}]>$, Armada uses a multiple attribute range query processing algorithm called MIRA to forward queries to the nodes in range. Suppose that $\delta_1 = <x_0, x_1, ..., x_{d-1}>$ and $\delta_2 = <y_0, y_1, ..., y_{d-1}>$. We use the MULTIPLE_HASH algorithm to find the Kautz strings corresponding to objects δ_1 and δ_2, and call them LowT and HighT, respectively. It is clear that for each object O in range Q, $\delta_1 \triangleleft O$ and $O \triangleleft \delta_2$.

Algorithm 1 The pseudocode of the *Multiple_hash* algorithm.

```
1: procedure MULTIPLE_HASH
   ( ObjectVal O, Lowerbound L, Upperbound
   U, Length k)
   // Generate an ObjectID (a Kautz string S
   of length k)
   // O is a multiple dimensional object and
   O_i is the attribute value of object in
   dimension i.
   // L_i and U_i are the lowerbound and the
   upperbound of the entire space in
   dimension i.
2:     left ← 0; right ← 1;
   d ← Numofdimension(O);
   // Initialize the vector nextID
3:     nextID[0][left] ← 1;
   nextID[0][right] ← 2;
4:     nextID[1][left] ← 0;
   nextID[1][right] ← 2;
5:     nextID[2][left] ← 0;
   nextID[2][right] ← 1;
6:     for (i ← 0, d − 1) do
7:         A[i] ← L[i]; B[i] ← U[i];
8:     end for
   // value O lies in the subspace represented
   by the first child of the root node
9:     if
   (O[0] ∈ [L[0], L[0] + 1/3 ∗ (U[0] − L[0])])
   then
10:        S[0] ← 0;
11:        A[0] ← L[0];
   B[0] ← L[0] + 1/3 ∗ (U[0] − L[0]);
12:    end if
   // value O lies in the second child of the
   root node
13:    if (O[0] ∈ [L[0] + 1/3 ∗ (U[0] −
   L[0]), L[0] + 2/3 ∗ (U[0] − L[0])]) then
14:        S[0] ← 1;
15:        A[0] ← L[0] + 1/3 ∗ (U[0] − L[0]);
   B[0] ← L[0] + 2/3 ∗ (U[0] − L[0]);
16:    end if
   // value O lies in the third child of the root
   node
17:    if
   (O[0] ∈ [L[0] + 2/3 ∗ (U[0] − L[0]), U[0]])
   then
18:        S[0] ← 2;
19:        A[0] ← L[0] + 2/3 ∗ (U[0] − L[0]);
   B[0] ← U[0];
20:    end if
   // Determine whether value O is in the
   subspace represented by the left or right
   child
21:    for (i ← 1, k − 1) do
22:        j ← i mod d;
23:        if (O(j) > (A[j] + B[j])/2) then
24:            direction ← 1;
25:            A[j] ← (A[j] + B[j])/2;
26:        else
27:            direction ← 0;
28:            B[j] ← (A[j] + B[j])/2;
29:        end if
30:        S[j] ← nextID[S[j − 1]][direction];
31:    end for
32:    return S;
33: end procedure
```

The MULTIPLE_HASH algorithm is a multiple attribute partial order preserving function. So the range Q is a subset of the Kautz region $[\![LowT, HighT]\!]$. For example, if $Q = <[1.2, 1.8], [1, 5]>$ is the range query, then $\delta_1 = <1.2, 1>$ and $\delta_2 = <1.8, 5>$. We use the partition tree shown in Figure 2.2 to obtain Kautz strings corresponding to δ_1 and δ_2. Thus, $[\![LowT, HighT]\!] = [\![0120, 0210]\!]$. This Kautz region

contains five leaf nodes P, R, W, S and M. Based on the partial order preserving property of the MULTIPLE_HASH algorithm, all the nodes that intersect a query are *among* five Kautz region nodes, but they may not be adjoining leaf nodes. Figure 2.2 shows that W and S leaf nodes don't intersect the query. So to process a range query, we need an algorithm to forward the query to *only* those nodes which intersect the query. MIRA is the algorithm proposed in [15] to forward a range query to the appropriate nodes. Algorithm 2 shows the pseudocode of MIRA.

Algorithm 2 The pseudocode of the MIRA algorithm.

1: **procedure** P.MIRA(RangeQuery Q)
// Node P invokes a multiple dimensional range query
$Q =< [Q_{L0}, Q_{U0}], ..., [Q_{Li}, Q_{Ui}], ...$
$, [Q_{L(d-1)}, Q_{U(d-1)}] >$
// $Q.L =< Q_{L0}, ..., Q_{Li}, ..., Q_{L(d-1)} >$
// $Q.U =< Q_{U0}, ..., Q_{Ui}, ..., Q_{U(d-1)} >$
2: $LowT \leftarrow$ MULTIPLE_HASH$(Q.L, L, U, k)$;
3: $HighT \leftarrow$ MULTIPLE_HASH$(Q.U, L, U, k)$;
4: $ComT \leftarrow$ COMMONPREFIX$(LowT, HighT)$;
5: **if** $ComT = null$ **then**
6: $Rangeset \leftarrow$DIVIDERANGE $(LowT, HighT)$;
// Divide Kautz region $[\![LowT, HighT]\!]$ into several sub-regions ($Range_i$)
// parallel search for each $Range_i$
7: **for** ($eachRange_i \in Rangeset$) **do**
8: P.MULSEARCH $(Q, range_i.LowT, range_i.HighT)$;
9: **end for**
10: **else**
11: P.MULSEARCH$(Q, LowT, HighT)$;
12: **end if**
13: **end procedure**
14: **procedure** P.MULSEARCH(RangeQuery Q,String T_1,String T_2)
// Node P invokes a pruning search for the multiple dimensional range query Q
// The destination nodes take charge of a part of the Kautz region $[\![T_1, T_2]\!]$
15: COMMONPREFIX(T_1, T_2);
// If node P's nodeID is a prefix of $ComT$, the destination peer is P and the search is finished
16: **if** (ISPREFIX$(P, ComT)$) **then**
17: QUERY(P);
18: **return** ;
19: **end if**
// $ComS$ is the longest Kautz string that both a prefix of $ComT$ and the suffix of P's nodeID
20: $ComS \leftarrow$ SUFFICPREFIX$(NodeID(P), ComT)$;
21: $MaxLevel \leftarrow$ LENGTHOFSTRING$(NodeID(P))$-LENGTHOFSTRING$(ComS)$;
22: P.MULTIPLEPRUNING$(Q, MaxLevel)$;
23: **end procedure**
24: **procedure** U.MULTIPLEPRUNING (RangeQuery Q,LeftDepth h)
// Node $U = a_1...a_hX$ deals with the pruning search message for the multiple dimensional range query Q
// The level of node U is h higher than that of destination nodes in the FRT

```
25:     if (h = 0) then // reach a destination peer
26:         QUERY(U);
27:         return ;
28:     else
29:         for (eachR ∈ outneighbors(U)) do // R = a_2...a_h XY
30:             if (INTERSECTION(XY, Q)) then
31:                 R.MULTIPLEPRUNING(Q, h − 1);
32:             end if
33:         end for
34:     end if
35: end procedure
36: procedure INTERSECTION(String S, RangeQuery Q)
    // Determine whether the subspace Q' represented by the Kautz string S in the partition tree intersect with Q
    // Q =< [Q_L0, Q_U0], ..., [Q_Li, Q_Ui], ..., [Q_L(d−1), Q_U(d−1)] >
    // The entire value interval of dimension i is [L[i], U[i]]
37:     left ← 0 right ← 1;
    // vector direct presents the branch corresponding to the next symbol for different current symbol in Kautz string
38:     direct[0][1] ← left; direct[0][2] ← right;
39:     direct[1][0] ← left; direct[1][2] ← right;
40:     direct[2][0] ← left; direct[2][1] ← right;
    // Calculate the subspace
    Q' =< [A[0], B[0]], ..., [A[i], B[i]], ..., [A[d − 1], B[d − 1]] >
41:     if (S[0] = 0) then
42:         A[0] ← L[0];
    B[0] ← L[0] + 1/3 ∗ (U[0] − L[0]);
43:     end if
44:     if (S[0] = 1) then
45:         A[0] ← L[0] + 1/3 ∗ (U[0] − L[0]);
46:         B[0] ← L[0] + 2/3 ∗ (U[0] − L[0]);
47:     end if
48:     if (S[0] = 2) then
49:         A[0] ← L[0] + 2/3 ∗ (U[0] − L[0]);
    B[0] ← U[0];
50:     end if
51:     for (i ← 1, d − 1) do
52:         A[i] ← L[i]; B[i] ← U[i];
53:     end for
54:     lenk ← LENGTHOFSTRING(S);
55:     for (i ← 1, lenk − 1) do
56:         j ← i mod d;
57:         if direct[S[i − 1]S[i]] = left then
58:             B[j] ← (A[j] + B[j])/2;
59:         elseA[j] ← (A[j] + B[j])/2;
60:         end if
61:     end for
    // Determine whether there is overlap between subspace Q' and Q
62:     if ISOVERLAP(A, B, Q) then
63:         return 1;
64:     else
65:         return 0;
66:     end if
67: end procedure
```

In FissionE, the out-neighbours of node $P = u_1u_2...u_t$ are in the form of $P = u_2u_3...u_tv_1...v_q$ with $0 \leq q \leq 2$. For demonstrating the routing path from a query issuer to a destination peer, the *forward routing tree* (FRT) is defined in [15]. Each node of the FRT is a peer of FissionE. The root node is a query issuer. Children of each node are its out-neighbours which are sorted in increasing order from left to right. The number of levels in a FRT is $t + 1$. Figure 2.3 shows an example of the

FRT.

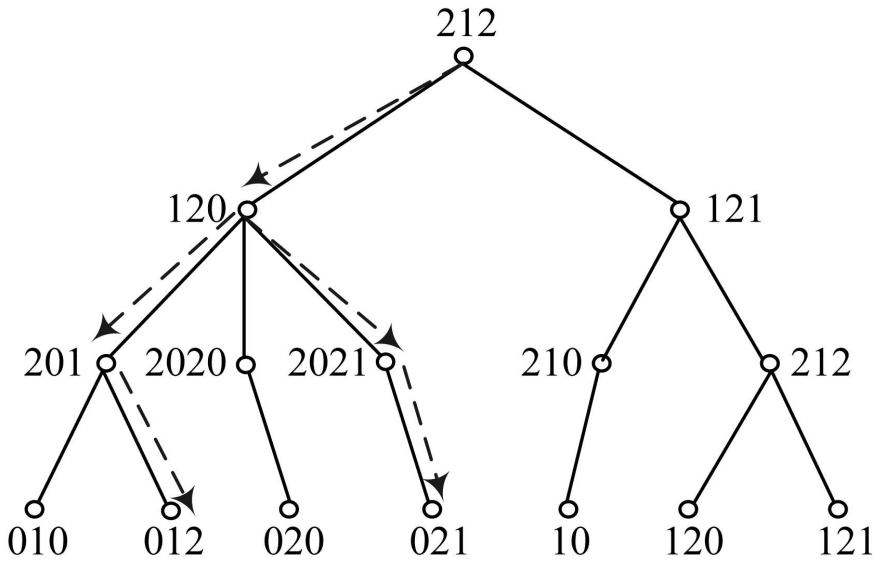


Figure 2.3: An example of the FRT and MIRA search path (from [15]).

To process a range query, we first obtain the Kautz region $[\![LowT, HighT]\!]$. If the Kautz string $LowT$ and $HighT$ don't have a common prefix, we need to divide the Kautz region into several subregions. In the case that $LowT$ and $HighT$ have a common prefix, we consider $ComT$ as the longest common prefix of $LowT$ and $HighT$. Then we get $ComS$ with length f as the longest Kautz string which is both the prefix of $ComT$ and the root nodeID (see line 20 of Algorithm 2). All the nodes that intersect query Q are at the level of $t - f$ of the FRT. At level i of the FRT, the node $B = u_{i+1}u_{i+2}...u_{t-f}X$ have some children in the form of $C = u_{i+2}u_{i+3}...u_{t-f}XY$. When B receives the query, it forwards the query to the children with their corresponding XY having an intersection with range query Q. Line 30 of algorithm 2 (which calls the InterSection algorithm) handles the sending of query messages only to nodes intersecting the query range.

In Figure 2.3, an example of processing a range query by MIRA is shown. The dashed arrows shows the search path. The root node 212 issues the range query $Q = <[1.2, 1.8], [1, 5]>$. As explained before, for this query $LowT = 0120$ and $HighT = 0210$. So $ComT = 0$, $ComS =$ "$null$" and $f = 0$. In the result, all the destination nodes are in level 3. In the node 120 of FRT, we calculate the subspace stored in node 2020 and find that the subspace does not intersect with Q. Therefore the query is not forwarded to 2020.

2.3.3 Performance of Armada

The performance of Armada is evaluated by analysis and simulation in [15]. Armada is built on top of FissionE which is a constant degree DHT. That means we need to store a constant number of pointers per node. Li et al. [15] have proven that the average message cost of 1-dimensional queries in PIRA (PrunIng Routing Algorithm) is $\log n + 2m - 2$ where m is the number of nodes intersecting the query. This average cost assumes the data is distributed in a uniform random fashion. For multi-dimensional indexing, Li et al. [15] have not presented any guarantee on the number of messages required to answer a d-dimensional orthogonal range. They present a simulation to show that the average message cost of MIRA is about $\log n + 4m - 1$ messages.

In addition, Armada uses the failure recovery mechanisms of the underlying DHT structure, FissionE [16] to accommodate routing recovery, but they don't provide data recovery. In addition, the maximum query delay for single and multiple attribute queries is less than $2 \log N$ hops in Armada. Query delay is the number of hops we need to traverse from the query issuer to detect the first node containing query results.

2.4 Distributed Spatial Data Structure (DSDS)

Distributed Spatial Data Structure (DSDS) [7] is a peer-to-peer spatial data structure supporting range search in 2-dimensional space. DSDS is a layered range query scheme that uses a non-redundant rainbow skip graph for the network topology and routing algorithm. A non-redundant rainbow skip graph [11] is a constant-degree peer-to-peer network originating from the idea of a skip graph [4]. Non-redundant rainbow skip graphs support exact match queries and range queries for ordered data in 1-dimensional space. In a non-redundant rainbow skip graph, each data point is assigned to a different node in the network. In other words, the number of nodes in the network is equal to the number of points stored on the distributed data structure. This approach of storing one point per node is the main disadvantage of the rainbow skip graph. To use a rainbow skip graph for range searching on spatial data, Bisadi et al. [7] proposed a data distribution algorithm that assigns a group of data to each node in such a way that closer points are stored in neighbouring node. In this section, we first introduce the topology of rainbow skip graphs and then discuss the data distribution and range search algorithm in DSDS.

In the topology of a non-redundant rainbow skip graph, the connection of nodes is based on defining a skip graph on $\Theta(\frac{n}{\log n})$ supernodes of size $\Theta(\log n)$, where n is the network size. To construct the structure of a non-redundant rainbow skip graph, first all nodes are ordered based on their keys in a *core list* which is a doubly linked list. Then the ordered nodes are partitioned into $\Theta(\frac{n}{\log n})$ supernodes where each supernode consists of a consecutive order of keys. The smallest node key in the node list of a supernode is the supernode key. A non-redundant rainbow skip graph has $\Theta(\log n)$ levels. Each supernode of a non-redundant rainbow skip graph has a representative in each level of graph. A *tower list* representative of supernode V in level i is connected to representatives of supernode V in levels $i-1$ and $i+1$. As each

level of a skip graph contains different level lists, the same notion of level lists in a skip graph is used for a non-redundant rainbow skip graph to connect representatives of different supernodes in one level to each other. Figure 2.4 shows an example of a non-redundant rainbow skip graph. Three different types of lists of a non-redundant rainbow skip graph are shown in this figure.

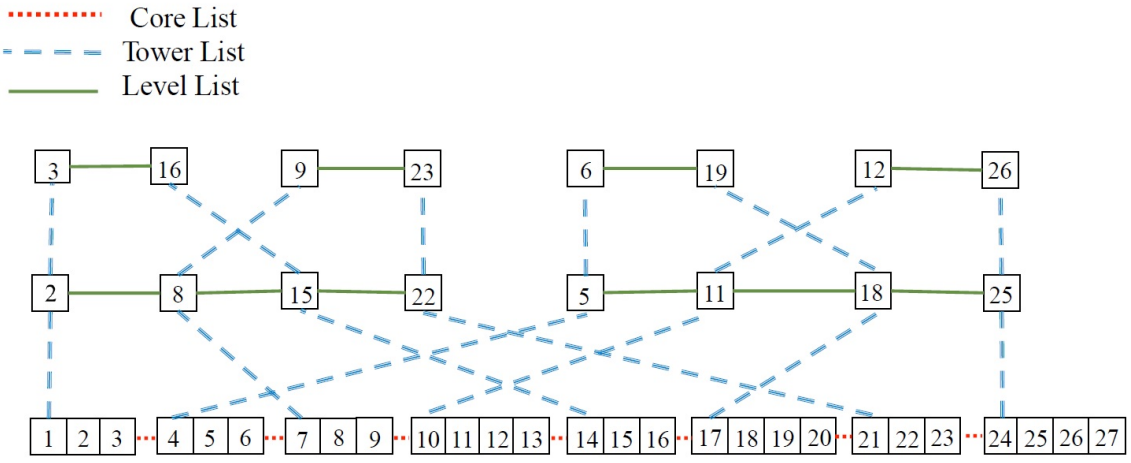


Figure 2.4: Three different types of lists in a non-redundant rainbow skip graph. Six level lists (two in level 1 and four in level 2), eight tower lists and one core list in level 0 are shown in the figure.

To distribute the data on nodes of a non-redundant rainbow skip graph in DSDS, a slab partitioning of the space along one of the axes is used. In this approach, we assign the keys of nodes in the non-redundant rainbow skip graph network to the slabs of data in an increasing order, and store the data of slab i on the node with key i. The core list of the non-redundant rainbow skip graph contains the data ordered by the chosen slab partitioning axis. Figure 2.5 shows a distribution of 2-dimensional points among the non-redundant rainbow skip graph nodes. In this figure, $n2_L$ and $n2_U$ are the lower and upper bounds of the second node region, respectively and $Q_{xL}, Q_{yL}, Q_{xU}, Q_{yU}$ are the range query bounds.

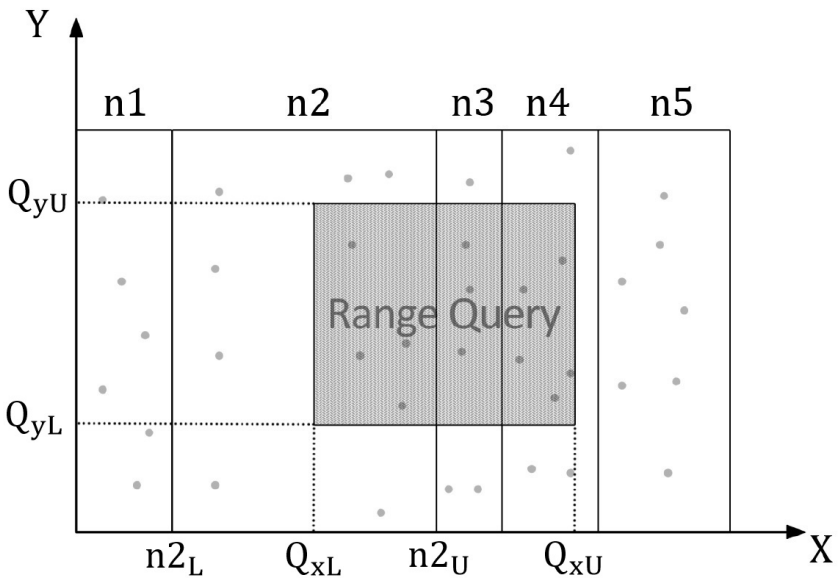


Figure 2.5: distribution of a 2D space among 5 nodes (adapted from [7]).

Searching the non-redundant rainbow skip graph for range query Q with $Q_L = [Q_{xL}, Q_{yL}]$ and $Q_U = [Q_{xU}, Q_{yU}]$ requires two steps. In the first step, if we assume that data is partitioned based on the x coordinate, we need to find the node whose data contains the x coordinate of the lower left point of query Q_{xL}. This node reports a part of data in range to the query issuer node, and checks whether the query intersects with the next node in the core list or not. If Q_{xU} is greater than the upper bound of its region, the node updates the query and forwards it to the successor node. This step is continued until all the nodes intersecting with the query report the points in range to the query issuer node.

DSDS is proposed for range query processing in 2-dimensional space and doesn't support range query processing in d-dimensional space. Since the non-redundant rainbow skip graph is a constant-degree graph and two copies of data are stored in DSDS, the required storage space for DSDS is $O(n + N)$. The average range search message cost for DSDS in 2-dimensional space is $\Theta(\log n + n\sqrt{\alpha})$ messages, where

n is the number of nodes in the network and α is the area $\in [0, 1)$ of a rectangular query Q. The DSDS worst case query in $[0, 1]^2$ is a rectangular query with the side length 1 in the dimension that is used for slab partitioning of the space, and arbitrary side length in another dimension. To answer this query, DSDS needs to forward the query to all the nodes of the network, so the number of required messages to answer a range query Q in DSDS in the worst case is $O(n)$ plus reporting cost.

Chapter 3

Data Structure

3.1 Overview

In this chapter, we propose our new fault tolerant dynamic distributed data structure that operates on top of FissionE [16], a constant-degree peer-to-peer network. Our data structure supports both point and range query processing in d-dimensional space. In addition, range search cost in our data structure is close to the lower bound on message cost of range search on distributed peer-to-peer networks using a topology index based on constant-degree distributed hash tables (DHTs). A low congestion routing algorithm in a Kautz graph is used for message routing, and $d-1$ backup copies of data is stored in our data structure to provide redundancy. This redundancy permits completely answering a query in the case of simultaneous failure of $d-1$ nodes.

We first introduce FissionE, a distributed hash table that our data structure is built on, in section 3.2. Sections 3.3 and 3.4 give a detailed description of how distribution of data among network nodes are done, and how orthogonal range search is carried out. Section 3.5 gives the theoretical analysis of our data structure. In section 3.6, we explain dynamic operations and fault tolerance on FissionE and finally, in section

3.7, load balancing is discussed.

3.2 Introduction to FissionE

In this section we explain FissionE, a constant-degree distributed hash table based on the Kautz graph [16]. A Kautz graph is a directed graph with static topology that uses Kautz strings as node identifiers. In the following, we present related definitions to understand Kautz graph topology.

Definition 3.2.1. *The string $u_1u_2...u_k$ of length k and base b is a Kautz string where $u_i \in \{0, 1, 2, ..., b\}$ and $u_i \neq u_{i+1}$ $(1 \leq i \leq k-1)$.*

Definition 3.2.2. *All Kautz strings of length k and base b create the $KautzSpace(b, k)$ of size $b^k + b^{k-1}$.*

To show the size of $KautzSpace(b, k)$, we know that the first symbol in a Kautz string has $b + 1$ possibilities. Because two consecutive symbols in a Kautz string must be different, all other symbols have b possibilities.

The Kautz graph $K(b, k)$ is a directed graph of degree b with $b^k + b^{k-1}$ nodes labelled by strings in $KautzSpace(b, k)$. Each node $U = u_1u_2...u_k$ of a Kautz graph has the same out-degree and in-degree b. There is an outgoing edge from U to V if and only if $V = u_2u_3...u_k\alpha$ where $\alpha \in \{0, 1, ..., b\}$ and $\alpha \neq u_k$. Figure 3.1 shows Kautz graph $K(2, 3)$ with out-degree 2 and 12 nodes.

A Kautz graph has desirable properties, like optimal diameter, that are important in peer-to-peer networks. Diameter is the longest shortest path between any two vertices of a graph and is always in trade-off with the degree of a graph. For a graph with $n = b^k + b^{k-1}$ nodes and degree b, the Kautz graph has the smallest diameter of

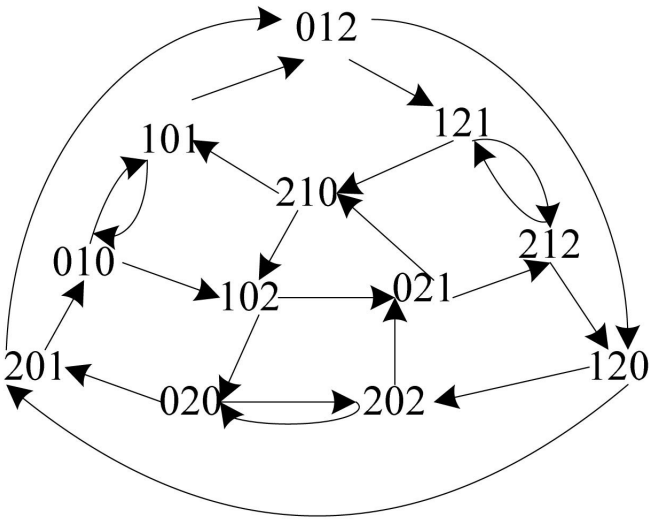


Figure 3.1: Kautz graph $K(2,3)$ (from [16]).

any possible directed graph. In addition, in Kautz graph $K(b,k)$, there are b disjoint paths between any two nodes.

FissionE uses a Kautz graph $K(2,k)$. Because a Kautz graph is a static topology, it needs some adjustment to be used for dynamic peer-to-peer networks. Li et al. in [16] propose a new topology called approximate Kautz graph. To achieve an approximate Kautz graph, first the network topology is initiated with a Kautz graph, and then in dynamic operations (addition and deletion of nodes) a topological rule called the *neighbourhood invariant* rule is adopted. Based on this rule, the length of identifiers may be different for different peers and the difference in length of node identifiers of any two neighbours must be one or zero.

Figure 3.2 shows an example of neighbourhoods in FissionE topology. This topology is first initiated with Kautz graph $K(2,3)$. Node 202 is split to permit node 2021 to join the network with existing node 202 becoming node 2020. Data in nodes 101 and 102 are merged to provide one less node which results in node 101 being relabelled to to node 10, and node 102 departing from the network. In section 3.6,

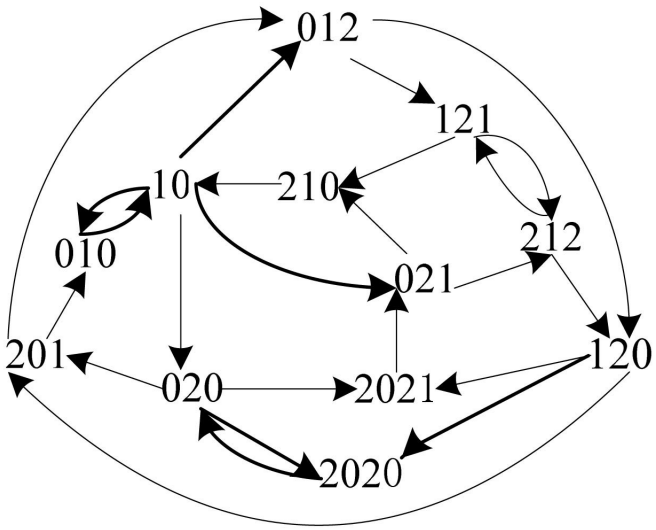


Figure 3.2: An example of FissionE topology (from [16]).

dynamic operations and their influence on the length of node identifiers are further explained.

To distribute objects among nodes in the FissionE scheme, the *Kautz_hash* algorithm is proposed in [16]. The Kautz_hash algorithm maps an object's unique key (of any length) to the destination Kautz string of length l consisting of digits $v_i \in 0, 1, 2$, where consecutive digits must be different. Li et al. [16] show that when $l = 100$, the Kautz_hash algorithm uniformly distributes the Kautz strings it generates in Kautz namespace KautzSpace(2,100). As mentioned in Algorithm 1.2.2 of [16], this namespace has size $2^{100} + 2^{99} \simeq 1.9 \times 10^{30}$. To publish an object on FissionE by node p, the Kautz string V of the object is first computed. Next, node p routes the generated Kautz string V to place the object in the node whose identifier is a prefix of V. To locate an object in the network, the same process is performed.

The long path routing algorithm in a Kautz graph is chosen as the Routing algorithm in FissionE. Algorithm 3 from [16] shows the FissionE routing algorithm. In this algorithm, routing from node U to the node where destination Kautz string V

Algorithm 3 FissionE Routing Algorithm.

1: **procedure** FissionE.Route(srcNode U, dstString V)
// routing from source node $U = u_1u_2...u_k$ to node where destination Kautz string $V = v_1v_2...v_m$ is placed.
// $SP = sp_1sp_2...sp_t$ the longest Kautz string that is the prefix of V and the suffix of U
2: $SP \leftarrow$SuffixPrefix(U, V)
3: U.Routing$(V, k - t, SP)$
4: **end procedure**
5: **procedure** W.Routing(dstString V, pathLen L, prefix S)
// node $W = w_1w_2...w_k$ routes message to the node that destination Kautz string $V = v_1v_2...v_m$ must be stored on.
6: **if** $(L = 0)$ **then** return
// reached destination node.
7: **else if** $\exists Q \in Outneighbors(W)\ \&\ Q = w_2...w_kX\ \&\ IsPrefix(SX, V)$ **then**
8: $S \leftarrow SX$
9: Q.Routing$(V, L - 1, S)$
// Routing messages is forwarded to Q
10: **end if**
11: **end procedure**

resides is performed by left shifting the symbols of U and adding the symbols of V from left to right at the end of U. For example, if $U = 021$ and $V = 12010$, the longest common prefix of V and suffix of U is equal to 1. So the length of path from node U to the node whose identifier is a prefix of V is 2 and the routing path is $021 \rightarrow 212 \rightarrow 120$.

It is proven in [16] that the average degree of vertices in a FissionE network is 4 and its Kautz graph diameter is less than $2 \log n$. These desirable characteristics motivate us to use FissionE as our overlay network to route messages between nodes and provide dynamic operations of node arrival and departure.

3.3 Data Distribution

DHT-based peer-to-peer networks usually use consistent hashing functions to map data objects and peers to a namespace. In the namespace, each node takes the responsibility of storing values with the IDs close to its own ID. In the case of orthogonal range query, as a peer-to-peer network requires data ordering, the hash function used to map values into the namespace is replaced by a locality preserving mapping function. Although the FissionE scheme is a high performance distributed peer-to-peer network and achieves optimal diameter on a constant-degree graph, it only supports processing of exact match queries (point queries). Since in FissionE, the Kautz_hash algorithm is based on the hash algorithm SHA-1, range query processing is not supported.

In this work, we present a general range query scheme that uses FissionE for routing messages. Two main components of our work are the data distribution algorithm and range query processing. Our data distribution algorithm publishes d copies of each object on d different nodes in a way that preserves data locality. Our range search algorithm, efficiently forwards queries to the appropriate nodes in range. We first give a formal definition of a total order relation, and then explain our data distribution algorithm. To efficiently perform orthogonal range search over a peer-to-peer network, it is required to define a total order relation on the dataset to keep the order of data in each dimension. A total order relation is a binary relation on set X denoted by $\leq$ which has the following properties for all a, b and $c \in X$:

1. Antisymmetry: If $a \leq b$ and $b \leq a$ then $a = b$.

2. Transitivity: if $a \leq b$ and $b \leq c$ then $a \leq c$.

3. Totality: $a \leq b$ or $b \leq a$.

A total order relation on data provides propagation of objects on FissionE nodes in such a way that objects with close values are placed on the same or neighbouring nodes. In our data structure, we define a total order relation for each dimension i as follows:
For two points $P(p_1, p_2, ..., p_d)$ and $Q(q_1, q_2, ..., q_d)$, $P \leq Q$ in dimension i if $p_i \leq q_i$.

As we explained in section 3.2, in FissionE the identifier of nodes are Kautz strings and the network node is initiated to a Kautz graph. All Kautz graphs have a Hamiltonian path. A Hamiltonian path in a graph is a path that visits each node of a graph exactly one time. In our work, we use the Hamiltonian path in the underlying Kautz graph of FissionE, and assign the index of each node in the Hamiltonian path as the key to each node.

To preserve data locality along all dimensions, we distribute data objects among nodes by partitioning the space based on point coordinates. In d-dimensional space, we assign d set of points to each node i on the network, each set corresponding to one dimension. Set S_{ji} is the data stored in node i based on the total order relation in dimension j. For example, in 2-dimensional space, if we denote dimension 0 with x coordinates, and dimension 1 with y coordinates, we assign two set of points S_{xi} and S_{yi} to every FissionE node i.

The distribution of points based on each dimension over nodes is a noncrossing partition $NC(\mathcal{S}) = \{S_{j1}, S_{j2}, ..., S_{jn}\}$ [25]. A partition over set $\mathcal{S}$ on dimension j has the following properties:

- The union of the sets of $NC(\mathcal{S}) = \{S_{j1}, S_{j2}, ..., S_{jn}\}$ is equal to $\mathcal{S}$. The elements of $NC(\mathcal{S})$ are said to cover $\mathcal{S}$; i.e. For any j, $0 \leq j \leq d-1$, $\cup_{i=1}^{n} S_{ji} = \mathcal{S}$ where d is the number of dimensions in data structure.

- The intersection of any two distinct sets of $NC(\mathcal{S})$ is empty; i.e. the elements of $NC(\mathcal{S})$ are pairwise disjoint. Thus $S_{ji} \cap S_{jk} = \emptyset$ if $S_{ji} \in NC(\mathcal{S})$, $S_{jk} \in NC(\mathcal{S})$, $i \neq k$.

In 2-dimensional space, our data structure provides one backup copy of data published on all nodes to achieve search cost near the lower bound, in addition to providing data recovery. A copy of data stored in node i is stored in all $n-1$ other nodes except node i. Figure 3.3 shows an overview of data distribution in 2-dimensional space over the Kautz graph $K(2,2)$ shown in Figure 3.4. The Hamiltonian path A, B, C, D, E for the Kautz graph $K(2,2)$ in Figure 3.4 is shown in Figure 3.5. In Figure 3.3, if the data is distributed in a uniform random fashion in space, a balanced load for each node results. The horizontal colour bar in each cell indicates the place of the first copy of data in that cell based on dimension 0 (X), and the vertical colour bar indicates the place of the second copy of data in that cell based on dimension 1 (Y). For example, assume $L = [0,0]$ and $U = [12,12]$ are the lower and upper bound of the entire 2-dimensional space, and $P = [0.8,1.2]$ is a point. By uniformly partitioning the space among the six nodes in Figure 3.4, point P is placed in the lower left cell with red and orange bars. The red and orange bars show that the first and second copies of point P are stored in nodes 12 and 20, respectively.

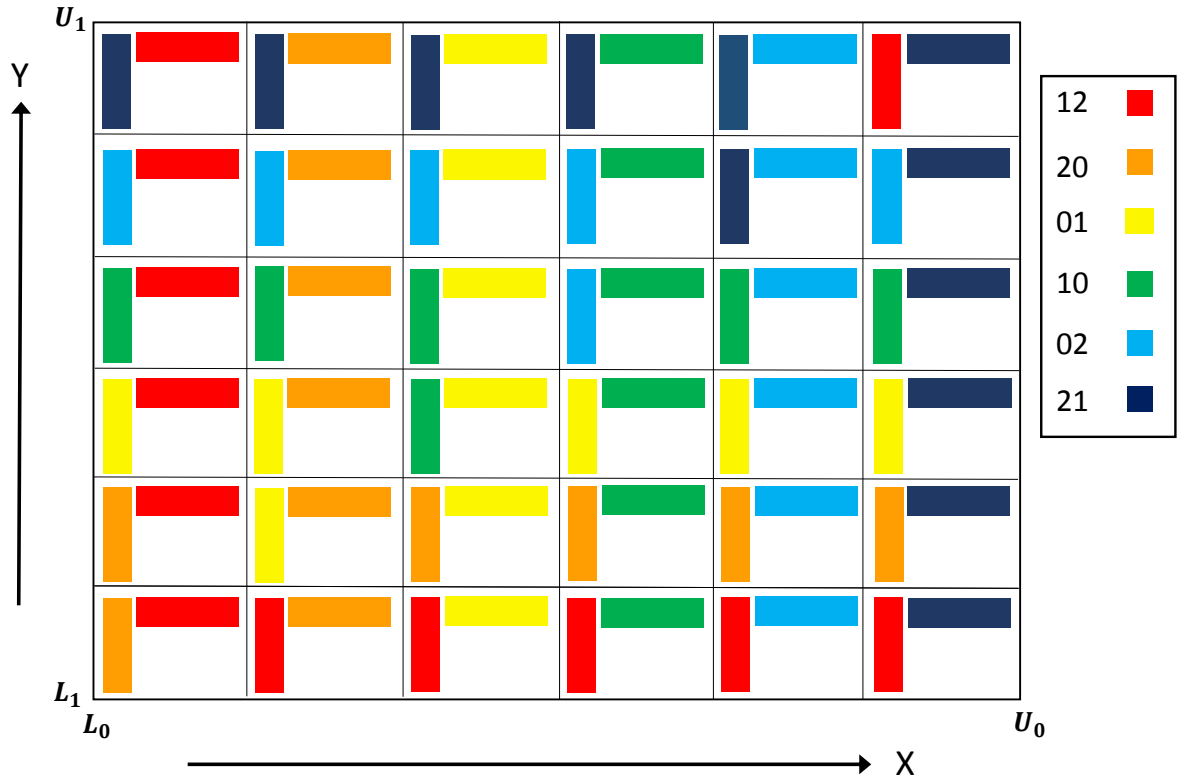


Figure 3.3: An overview of 2-dimensional data distribution in our data structure.

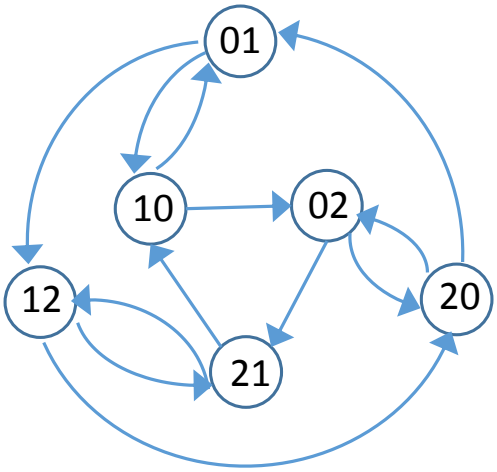


Figure 3.4:
Kautz graph $K(2,2)$.

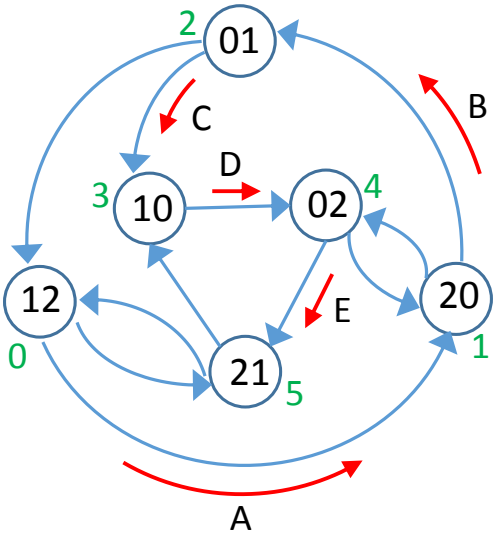


Figure 3.5:
A Hamiltonian path on Kautz graph $K(2,2)$.

Algorithm 4 shows how data objects are distributed on network nodes. This algorithm publishes d copies of object O on d different nodes. The place of the ith copy of object O depends on O_i and the place of the other $i-1$ copies of O that are already specified. $[L_i, U_i]$ in this algorithm is the entire interval of all N object values in dimension i.

Algorithm 4 Publish data object O on a network with n nodes.

1: **procedure** DATADISTRIBUTION(ObjectVal O, NumofNodes n, NumofDims d, LowerBound L, UpperBound U, HamiltonianPath P)
   // $O = [O_0, O_1, ..., O_{d-1}]$ is the coordinate of a point that should be published on d nodes.
   // $L = [L_0, L_1, ..., L_{d-1}]$ and $U = [U_0, U_1, ..., U_{d-1}]$ are the lower and upper bounds, respectively, of the entire space.
2: **if** $(O < L \| O > H)$ **then**
3: **return** (O is not in range.)
4: **end if**
5: $NodeIndex \leftarrow null$ // $NodeIndex$ is an array of size d showing the indices of nodes that O will be published on.
6: **for** $i \leftarrow 0, d-1$ **do**
7: $node = \lceil \frac{O_i - L_i}{U_i - L_i}(n) \rceil - 1$
8: **for** $j \leftarrow 0, i$ **do**
9: **if** $NodeIndex[j] == node$ **then**
10: $node = (node + 1) \mod n$
11: $j \leftarrow 0$
12: **end if**
13: **end for**
14: $NodeIndex[i] \leftarrow node$
15: **end for**
16: $NodeID \leftarrow null$ // $NodeID$ is an array of size d showing the Kautz string of nodes where O is stored.
17: **for** $i \leftarrow 0, d-1$ **do**
18: $NodeID[i] \leftarrow P[NodeIndex[i]]$
19: **end for**
20: **return**($NodeID$)
21: **end procedure**

To describe the data distribution algorithm in our data structure, we propose a *distribution tree* $DT(d, n)$ where d is the number of dimensions and n is the network size. The distribution tree $DT(d, n)$ has $d+1$ levels and n children for each interior or root node. The label of a node is chosen to give a unique integer path from the root to any leaf node. This label is decided by the loop from lines 6 to 15 in Algorithm 4. The label of the root node is *null* and the label of a *node* is the concatenation of the labels of the nodes on the path from the root node to *node*, separated by commas.

The distribution tree shows how all N objects, both primary and all $d-1$ backup copies, defined in space $[L, H]$ are distributed among the n nodes of the network. $L = [L_0, L_1, ..., L_{d-1}]$ and $U = [U_0, U_1, ..., U_{d-1}]$ are the lower and upper bounds, respectively, of the entire space. The root node represents the entire space $[L, H]$ and other nodes represent subintervals of $[L, H]$. The label of each leaf node is the path of unique comma separated integers $\in 0, 1, ..., n-1$. Note that no integer repeats in the path due to the $d-1$ backup copies being stored in different dimensions.

Figure 3.6 shows an example of the distribution tree $DT(2, 6)$ correspondence to the data in Figure 3.3. In the example shown in Figure 3.6, the entire interval is $[L, H]$, where $L = [0, 0]$ and $H = [12, 12]$. The first leaf node in Figure 3.6 represents the rectangle with the lower left $[0, 0]$ and the upper right $[2, 2]$ and its label is $0, 1$. The label indicates that nodes 12 and 20 (the first and second nodes in the Hamiltonian path A, B, C, D, E shown in Figure 3.5) contain the primary and backup copies of points in the space [0, 2] $\times$ [0, 2]. The green integers in Figure 3.5 indicate the node index used in Algorithm 4.

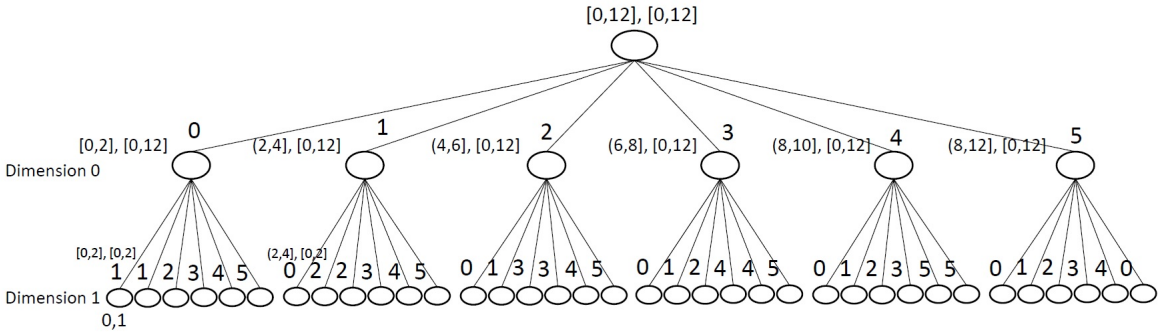


Figure 3.6: Example of a distribution tree.

3.4 Search Algorithms

3.4.1 Point Search Algorithm

We initially assume that the query Q is searching for one specific object O in our data structure. The query can be issued at any one of the nodes. Routing starts at the query issuer node. The query issuer uses Algorithm 4 to find the $NodeID$ list corresponding to object O which contains d distinct addresses of O. The query issuer then determines which of the object's addresses to use. To do that, the longest Kautz string SP_i which is a suffix of the query issuer ID and a prefix of $NodeID[i]$ is calculated for each i, $0 \leq i \leq d-1$. Then, the query issuer uses the FissionE routing algorithm 3 to pass the query to the $NodeID[j]$, where SP_j has the maximum length among all elements of list SP. The point search algorithm is shown in Algorithm 5.

Algorithm 5 Point Search Algorithm.

1: **procedure** PointSearch(ObjectVal O, NumofNodes n, NumofDims d, LowerBound L, UpperBound U, HamiltonianPath P)
   // Find a list $NodeID$ of size d that shows the Kautz string of d nodes that each one stores one copy of the queried Object O.
   // $O = [O_0, O_1, ..., O_{d-1}]$ is the coordinate of a point that should be published on d nodes.
   // $L = [L_0, L_1, ..., L_{d-1}]$ and $U = [U_0, U_1, ..., U_{d-1}]$ are the lower and upper bounds, respectively, of the entire space.
2: **if** $(O < L \| O > H)$ **then**
3: **return** (O is not in range.)
4: **end if**
5: $j \leftarrow -1$
6: $NodeID \leftarrow$ DataDistribution(O, n, d, L, U, P)
7: **for** $i \leftarrow 0$ to $d-1$ **do**
8: $SP \leftarrow$SuffixPrefix$(ThisNode.ID, NodeID[i])$
9: **if** $j < SP[i].length$ **then**
10: $j \leftarrow i$
11: **end if**
12: **end for**
13: FissionE.Route$(ThisNode, NodeID[j])$
14: **end procedure**

3.4.2 Range Search Algorithm

Algorithm 6 answers a d-dimensional range query Q in a network of n nodes, utilizing point search. Our range search algorithm has two main parts: First, it determines which dimension is the most appropriate to process a query in terms of minimizing the number of required messages. Second, the potential node i, the first node in range is found, and the first portion of data in range is reported. The node i checks if the query upper bound is greater than node i's upper bound. If so, the rest of the query result might be in the next node and an updated query is sent to the next node in range. The new query rectangle is the result of subtraction of the range covered with node i from the old query rectangle. The same process continues

until the last node intersecting the query reports the last part of the result to the query issuer node.

For example, assume that j is the dimension intersecting the fewest nodes. Node i is the first node that sends the result back to the query issuer if $ThisNode_{jL} < Q_{jL} < ThisNode_{jU}$ where $ThisNode_{jL}$ and $ThisNode_{jU}$ are node i's lower bound and upper bound respectively, in dimension j and Q_{jL} is the lower bound of the query in dimension j. Then, if the upper bound Q_{jU} of the query in dimension j is greater than $ThisNode_{jU}$, the query is passed to the next node in range. Node $i+1$ receives an updated query, where the query upper bound Q_{jU} remains unchanged, and the query lower bound Q_{jL} is changed to $ThisNode_{jU}$.

Algorithm 6 Report data objects in a range query to its issuer node.

1: **procedure** IssueQuery(rangeQuery Q)
// Find the dimension j with minimum range $Q_{jU} - Q_{jL}$ in query Q
// L_j and U_j are the lower bound and upper bound, respectively, of all possible values for dimension j.
2: $j \leftarrow$ MinRangeDim(Q) // Proper dimension for query processing
3: $dstIndex \leftarrow \lceil \frac{Q_{jL} - L_j}{U_j - L_j}(n) \rceil - 1$
4: $dstID \leftarrow HamiltonianPath[dstIndex]$
5: FissionERouting($thisNode, dstID, Q, j$)
6: **end procedure**
7: **procedure** FissionERouting(srcNode src, dstNode dst, RangeQuery Q, properD j)
// Assume that $src = src_1 src_2 ... src_k$ and $dst = dst_1 dst_2 ... dst_m$.
8: $SP \leftarrow$SuffixPrefix(src, dst)
// $SP = SP_1 SP_2 ... SP_t$ the longest Kautz string that is a prefix of dst and a suffix of src.
9: src.Routing($dst, k - t, SP, Q, j$)
10: **end procedure**
11: **procedure** V.Routing(dstNode dst, pathLen L, sufPre SP, rangeQuery Q, properD j)
12: **if** $(L = 0)$ **then**
13: **if** $(Q_{jL} > ThisNode_{jL})$ and $(Q_{jL} < ThisNode_{jU})$ **then**
// Report all objects in range where $O_j < ThisNode_{jU}$.
14: ReportAnswer(LocalSearch(Q), Q.issuer)
// If not the last node in range
15: **if** $(Q_{jU} > ThisNode_{jU})$ **then**
16: $Q_{jL} \leftarrow ThisNode_{jU}$
// Route updated query to the next node in Hamiltonian path
17: FissionERouting($thisNode, thisNode.next, Q, j$)
18: **end if**
19: **end if**

20: **else if** $\exists Q \in Outneighbors(V)$ & $Q = V_2...V_kX$ & $IsPrefix(SX, dst)$ **then**
// ROUTING method has been called i times.
21: $S \leftarrow SX$
22: Q.ROUTING$(dst, L-1, S, Q, j)$
23: **end if**
24: **end procedure**

3.5 Theoretical Analysis

Storage space and search cost are two important elements that affect the performance of a data structure. In this section, we show and prove some theorems about the performance of our data structure and compare the performance of our data structure with distributed spatial data structure (DSDS) [6].

Theorem 3.5.1. *The cost of searching for an object in our data structure is* $O(\log n)$ *messages.*

Proof. Our point search algorithm (Algorithm 5) is based on the routing algorithm of FissionE (Algorithm 3). In [16], it has proven that the diameter of Fissione is $O(\log n)$. Since the diameter is the longest shortest path between any two vertices of a graph, we have the claimed point search cost. □

Theorem 3.5.2. *The worst case orthogonal range search cost in our fault tolerant data structure for any data distribution in d-dimensional space with n nodes is* $O(\log n + m)$ *messages plus reporting cost, where m is the minimum number of nodes intersecting the query on d dimensions.*

Proof. Since the most efficient dimension for the issued query is selected at the beginning of the range search algorithm, the worst case search cost occurs when the query is an equal-sided box. Li et al. have proven in [16] that the diameter of

FissionE is $O(\log n)$. So, in the worst case, the cost of finding the node containing the lower bound Q_{jL} of the orthogonal range query is $O(\log n)$. After that we need $O(m)$ messages to pass the updated query to the following nodes in range using the current Hamiltonian path to find data objects intersecting the query. □

Bisadi et al. in [6] have proven that the cost of point search in DSDS is $O(\log n)$ messages. So based on Theorem 3.5.1, the point search cost in DSDS is equal to the point search cost in our data structure. However, different ways of space partitioning result in different range search cost. In DSDS, the cost of range search in the worst case is different from the average search cost. As space in DSDS is partitioned among nodes in a rectangular slab fashion across one dimension (Figure 2.5), the worst case query may intersect all the nodes in the network while having no data in range. It is proven that the worst case orthogonal range search cost in DSDS with n nodes is $O(n)$ messages plus reporting cost. The expected orthogonal range search cost of query Q in DSDS is $\Theta(\log n + n\sqrt{\alpha})$ messages plus reporting cost where α is the area $\in [0, 1)$ of a query square Q.

Theorem 3.5.3. *In our data structure, assuming B points fit in one message, the cost of reporting K points found in range back to the query issuer node in a d-dimensional space with n nodes is $\sum_{i=1}^{m} \lceil \frac{K_i}{B} \rceil O(\log n) \in O((\frac{K}{B} + m) \log n)$ where m is the minimum number of nodes in range and n is the number of peer-to-peer network nodes.*

Proof. The size of the reporting set is K points that are stored in m different nodes in range. If each node i in range stores K_i points, it requires to send $\lceil \frac{K_i}{B} \rceil$ messages to report the data. Since the diameter of FissionE network is $O(\log n)$, the total reporting cost is $\sum_{i=1}^{m} \lceil \frac{K_i}{B} \rceil O(\log n)$ messages.

To find the upper bound of $\sum_{i=1}^{m} \lceil \frac{K_i}{B} \rceil$, we consider the worst case that:

$$\forall 1 \leq i \leq m, \lceil \frac{K_i}{B} \rceil = \lfloor \frac{K_i}{B} \rfloor + 1 \tag{3.1}$$

Since $\sum_{i=1}^{m} \lfloor \frac{K_i}{B} \rfloor \leq \frac{K}{B}$, we have $\sum_{i=1}^{m} \lceil \frac{K_i}{B} \rceil \leq \frac{K}{B} + m$. As $\sum_{i=1}^{m} \lceil \frac{K_i}{B} \rceil \in O((\frac{K}{B} + m))$, we have the claimed reporting cost $O((\frac{K}{B} + m) \log n)$. □

Theorem 3.5.4. *The required storage space for our data structure is $O(dN + n)$, where d is the number of dimensions, n is the number of nodes in the network and N is the number of objects stored in the data structure.*

Proof. It was proven in [16] that the average degree of FissionE is 4. So if the network contains n nodes, the required space for storage of routing tables on the nodes is $O(n)$. To store d complete copies of data in our data structure, we need dN space. So the overall required space is $O(dN + n)$. □

DSDS is designed for data in 2-dimensional space and uses a non-redundant rainbow skip graph [10] to route messages among nodes. Since the non-redundant rainbow skip graph is a constant-degree graph and 2 complete copies of data is stored in DSDS, the required storage space in DSDS is $O(n + N)$. In comparison, our data structure supports range query processing in d-dimensional space. We store d copies of data in our data structure to be able to achieve the worst case range search cost close to its lower bound proved in [15]. In addition, storing d copies of data in our data structure permits supporting simultaneous failure of $d-1$ nodes. This comparison confirms the trade-off between storage space and message cost in peer-to-peer networks.

3.6 Dynamic Operations

Dynamic operations such as node failure, insertion and deletion of points and insertion and deletion of node are important. Since our data structure is built on top of the FissionE scheme, it uses the maintenance mechanisms of FissionE to handle dynamic operations.

In FissionE [16], split large and merge small policies are used to support dynamic joining and departure of nodes. When we delete a node from a FissionE network, another node in the network takes the responsibility of deleted node data. When a node fails, data replication in our data structure is used to provide complete answers to queries. In this section, we explain the procedures of dynamic operations in our data structure.

When node A receives a message to add node B to the network, if A has a neighbour node with more data stored in it, A forwards the add node message to the neighbour node. If there is more than one neighbour with more data, one of them is selected randomly to forward the add node message to. This process continues until the add node message reaches a node C which has no neighbour with more data and the message cannot be forwarded any more. The data stored in node C is split into two sets. One set is assigned to node B and another set remains on node C. In this step, the Kautz string identifier of C is changed and the Kautz string identifier of B is generated. Assume that the identifier of C is $c_1c_2...c_k$. The new identifier of C is $c_1c_2...c_kx_0$ and the identifier of B is $c_1c_2...c_kx_1$ where $0 \leq x_0, x_1 \leq 2, x_1 \neq x_2, x_1 \neq c_k, c_k \neq x_2$. After generation of new identifiers, the routing table of nodes B and C and their neighbours are updated. Algorithm 7, adapted from [16], shows the complete process for inserting a new node.

Algorithm 7 Insertion of a new node to our data structure.

1: **procedure** INSERTNEWNODE(NewNode B)
   // node A receives a message to add node B to the network
2: **while** $\exists C \in neighbors(A)$ & $|S_C| > |S_A|$ **do**
3: $A \leftarrow C$
4: **end while**
5: SPLIT(S_A)
   // split data of node A into two sets
6: ASSIGN_NEWIDENTIFIERS(A, B)
   // assign new identifiers for nodes A and B
7: UPDATE_ROUTINGTABLES
   // update the routing table of A, B and their neighbours
8: **end procedure**

The above split large procedure shows that the longer the identifier of a node is, the smaller the amount of data stored in it. So, determining the neighbour with more data to forward an add node message to is performed using the length of a node's identifier. In addition, our data structure stores d copies of the data.

If we want to delete node A from the network, node A produces a delete node message. The delete node message is forwarded continuously to a neighbour node with less data until two nodes $B_1 = b_1b_2...b_{k-1}b_k$ and $B_2 = b_1b_2...b_{k-1}\tilde{b_k}$ in the network are found which have no neighbour node with less data where $0 \leq b_k, \tilde{b_k} \leq 2, b_k \neq \tilde{b_k}, b_k \neq b_{k-1}, b_{k-1} \neq \tilde{b_k}$. Nodes B_1 and B_2 are merged into a new node $B = b_1b_2...b_{k-1}$ and the neighbour list of B and related nodes are updated. We now have one extra node (B_1 or B_2) to get the $NodeID$ of the deleted node A and be responsible for its data. Algorithm 8, adapted from [16], shows the complete process for deleting a node from our distributed spatial data structure.

When failure of a node occurs, problems arise due to an outdated routing table and the fact that the data set assigned to the failed node will be unavailable. To enhance fault tolerance, most distributed data indexing schemes use replication based

Algorithm 8 Deletion of a node from our data structure.

procedure DELETENODE(Node A)
// node A generates a message in order to be deleted from network.
    $U \leftarrow A$
    $flag \leftarrow 1$
    repeat
        while $\exists Q \in neighbors(U)$ & $|S_Q| < |S_U|$ **do**
            $U \leftarrow Q$
        end while
        get neighbour $R = au_1...u_i$ $(k-2 \leq i \leq k-1)$ of node $U = u_1...u_{k-1}u_k$
        get neighbour $W = u_1...u_{k-1}\tilde{u_k}q_1...q_m$ $(0 \leq m \leq 1)$ of node R
        if $m = 0$ **then**
            if $\exists T \in neighbors(W)$ & $|S_T| < |S_W|$ **then**
                $U \leftarrow T$
            else
                $B_1 \leftarrow U$
                $B_2 \leftarrow W$
                $flag \leftarrow 0$
            end if
        else
            get neighbours $W' = u_1...u_{k-1}\tilde{u_k}\tilde{q_1}$ of R
            if $\exists T \in neighbors(W, W')$ & $|S_T| < |S_W|$ **then**
                $U \leftarrow T$
            else
                $B_1 \leftarrow W$
                $B_2 \leftarrow W'$
                $flag \leftarrow 0$
            end if
        end if
    until flag=0
    $B \leftarrow$MERGE(B_1, B_2)
// merge two nodes B_1 and B_2
    ASSIGN_NEWIDENTIFIERS(B)
    UPDATE_ROUTINGTABLES
// update the routing table of B and its neighbours
end procedure

mechanisms. Data redundancy is part of our distributed spatial data structure as explained in section 3.3. In d-dimensional space, our data structure stores d copies of data on d different nodes.

If one node in the network fails, we use the involuntary departure of nodes methods in FissionE. Each node periodically checks whether its neighbours are alive. When the failure of node F is detected by its neighbour A, A generates a deletion message for node F. The remaining process is similar to the node deletion procedure. After a recovery procedure all queries can be processed completely. Since the network retrieves the data of the failed nodes whenever failure of one or more (up to $d-1$) nodes occurs, our data structure can support simultaneous failure of $d-1$ nodes.

Theorem 3.6.1. *Assuming a load balanced peer-to-peer network of n nodes storing N points, the cost of recovering network topology and data after failure of one node in our data structure, in d-dimensional space is $O(\frac{dN}{nB}\log n)$ messages, where B is the number of points that fit in one message.*

Proof. It has been proven in [16] that when one node fails, deletion messages are propagated less than $\log n$ hops. So, the cost of merging two nodes and maintenance of the overlay network is $O(\log n)$. After that, each node finds which parts of its own data were stored in the failed node, and sends this data to the replacement node. If we assume that B points can fit in one message, the data recovery process requires $O(\frac{dN}{nB}\log n)$ messages since when one node fails, $\frac{dN}{n}$ points residing on the failed node are lost. So, $O(\frac{dN}{nB})$ messages are forwarded at most $O(\log n)$ hops to send back the lost data to the replacement node. The overall cost is thus $O(\log n + \frac{dN}{nB}\log n) = O(\frac{dN}{nB}\log n)$ messages. □

In [6], a network recovery algorithm is proposed to support failure of one node. Using DSDS network recovery algorithm and the backup copy of data stored in

the network, DSDS requires $O(\frac{N}{nB} + \log n)$ messages to recover a failed node. In comparison to DSDS, our data structure requires more messages to recover failure of one node. The reason for this cost is that DSDS stores the backup copy of data on its neighbouring nodes but in our data structure data is distributed on $d-1$ other nodes.

3.7 Load Balancing

Load Balancing is another important issue of efficient operations in peer-to-peer networks. Load balancing distributes computation and storage resources across nodes in a network. In a distributed hash table (DHT), in order to map objects to network node, a hash function is implemented. Most DHT schemes use a "good enough" hash function randomizing the placement of objects and ensuring a balanced load for each node in the network.

For FissionE, as explained in section 3.2, the Kautz_hash algorithm is used to uniformly distribute the objects among nodes. For some applications like range searching, the randomization of data addresses cannot be used, and the objects must be placed on nodes with a specific order.

Armada [15] presents a Probability-based lOad Balancing Mechanism (POBM) to generate uniform Kautz strings as ObjectIDs. The main idea of POBM is to use the probability density function (PDF) of attribute values. In this method, instead of equally partitioning each interval of the partition tree (in Figure 2.2), the probability distribution of attribute values is used to guarantee the number of points in each subinterval of a node are balanced. For example, assume $\rho(x)$ is the Probability Density Function (PDF) of attribute values in 1-dimensional space. At node A with f children and interval $[a, b]$, subinterval $[a_i, b_i]$ for each child satisfies equation

$\int_{a_i}^{b_i} \rho(x) = \frac{\int_a^b \rho(x)}{f}$. This mechanism of load balancing has two disadvantages; first we need to know the distribution of objects in advance. Second, if the distribution of objects is changed due to dynamic operations, the objects are not uniformly distributed among nodes any more.

In our data structure, the DataDistribution algorithm produces a load balanced network only if the data is uniformly distributed in d-dimensional space. If the distribution of data is skewed, the load across the nodes may become unbalanced. We present a mechanism to distribute objects evenly among nodes and keep load balancing after dynamic operations.

In the first step, we preprocess the data to set d lower bounds and d upper bounds for every node, with each lower bound and upper bound corresponding to one dimension. In this preprocessing step, we ensure that $\Theta(\frac{N}{n})$ objects are placed between L_{ji} and U_{ji} where L_{ji} and U_{ji} are the lower bound and upper bound of node i in dimension j ($0 \leq i \leq n-1$ and $0 \leq j \leq d-1$). To process queries in this scheme, each node needs an extra nd space to store the boundaries of other nodes.

In the second step, for keeping the number of points stored in each node balanced, we sometimes need to update the node boundaries and move some objects from one node to another. For this reason, we periodically update the boundaries of nodes in a random dimension. Let t be the length of an update period, N' an estimate of the number of points stored in the network, and j a random dimension. This load balancing procedure starts from the first node in the Hamiltonian path. For each node this procedure checks whether $c_1\frac{dN'}{n} \leq |S_i| \leq c_2\frac{dN'}{n}$ where $|S_i|$ is the number of points stored in node i, c_1 and c_2 are two constant values , and $c_1 < c_2$. If this inequality is true, the load on node i is balanced and if not, by changing the upper bound of node i in dimension j the load on node i becomes balanced. Since d copies

of data are stored in our data structure, all d copies of points between the old and the new boundaries need to be moved to new locations based on new boundaries. The cost of these load balancing operations depend on the data distribution and the number of updates made to insert and delete points or nodes. In the worst case, all data points reside on one node, and these points need to be evenly distributed to the other $n - 1$ nodes.

Chapter 4

Experimental Validation

Simulations of the our data structure were implemented using a machine with 3.10 GHz Intel(R) Core(TM) i5-2400 CPU with 8.00 GB RAM and running a `C#` 3.0 program on the `.NET Framework 3.5`. The program implementing Algorithms 4 and 6 consists of 573 lines of `C#` code. The code implementing the main test harness simulation (Algorithm 9) is given in Appendices A, B and C. For this simulation, points are distributed in a uniform random fashion in $[0, 1]^d$. By equally partitioning the entire interval of values in each dimension among n nodes, each node stores an equal numbers of points $\frac{dN}{n}$. We performed a simulation of the performance of our data structure for $n \in$ 24, 48, 96, 192, 384, 768, 1,536, 3,072, 6,144, 12,288; in each case N =1,000n.

The reporting cost of $\sum_{i=1}^{m} \lceil \frac{K_i}{B} \rceil O(\log n)$ messages proven in theorem 3.5.3 cannot be avoided. In the following experiments, it is assumed that all points in range at one node can be reported by one message. So, the number of points stored at each node doesn't affect the number of messages required to answer a query. For each experiment, we average the result of c =1,000 randomly generated range queries such that each query is issued by a random node. The test harness program executing the data publication and searching algorithms is given in Algorithm 9.

The cost of answering a range query in our data structure consists of three parts: the cost of finding the first node intersecting the query $O(\log n)$, the cost of passing the query to every node that has intersection with the query and the cost of reporting the data in range to the query issuer. In the following sections, we vary the network size (number of nodes in the network n) and query range size, and measure the following parameters:

1. Average search cost in messages $\bar{K}_Q = \frac{\Sigma_{i=1}^{c} K_{Qi}}{c}$, where K_{Qi} is the number of messages to answer one query Q_i.

2. Average number of nodes intersecting queries $\bar{n}_Q = \frac{\Sigma_{i=1}^{c} n_i}{c}$, where n_i is the number nodes intersecting query Q_i.

3. Message cost ratio $R_M = \frac{\bar{K}_Q}{\bar{n}_Q}$ which is the average number of required messages per node intersecting the query.

4. Average reporting cost in messages $\bar{T} = \frac{\Sigma_{i=1}^{c} T_i}{c}$, where T_i is the reporting cost for one query Q_i.

Metrics 1, 2 and 3 were used by Li et al. [15] in reporting the simulation results for Armada. Since the average search cost $\bar{K}_Q$ doesn't include the reporting cost, we also define $\bar{T}$ as explained above.

Figures 4.1 (a) and (b) show the impact of network size on average search cost $\bar{K}_Q$, $\bar{n}_Q$ and R_M in 2-dimensional space. In the simulations, the query rectangle is defined randomly as shown in Algorithm 10. In Algorithm 10, the query side length $\Delta \in [0, 1]$. Experimental results show that the average query side length of the $c =$ 1,000 query rectangles is 0.497. The network size n varies from 24 to 12,288 nodes. Figure 4.1 (a) shows that the average search cost $\bar{K}_Q$ and $\bar{n}_Q$ increasing linearly with network size. From Figure 4.1 (b), we observe that when the network size increases, R_M is closer to one. In other words, the number of required messages to process

Algorithm 9 The test harness algorithm.

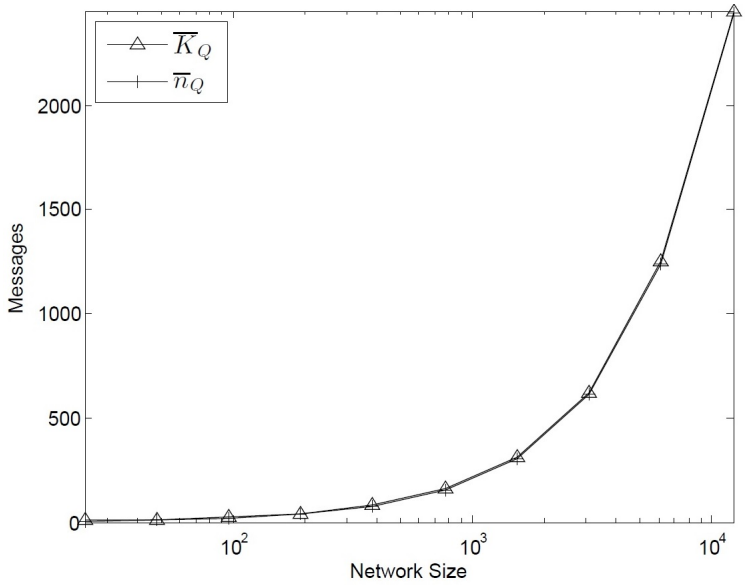
```
1: procedure TestHarness1(networkSize n, numofObjects N, numofDim d,
   numofQueries c)
2:     Kautznetwork K ← makeGraph(n)
   // The Kautz network is made based on Kautz graph K(2, k).
3:     Fill_Nodes_PointLists(N, d)
4:     queries ← GenerateRangeQueries(c, d, K)
   // Each query is assigned to one of the nodes of network K to be its issuer
   node.
5:     for each QueryMessage Q in queries do
6:         Q.Issuernode.IssueQuery(Q)
7:     end for
8: end procedure
```

a randomly generated query is dominated by the number of messages required for passing query Q to the nodes in range.

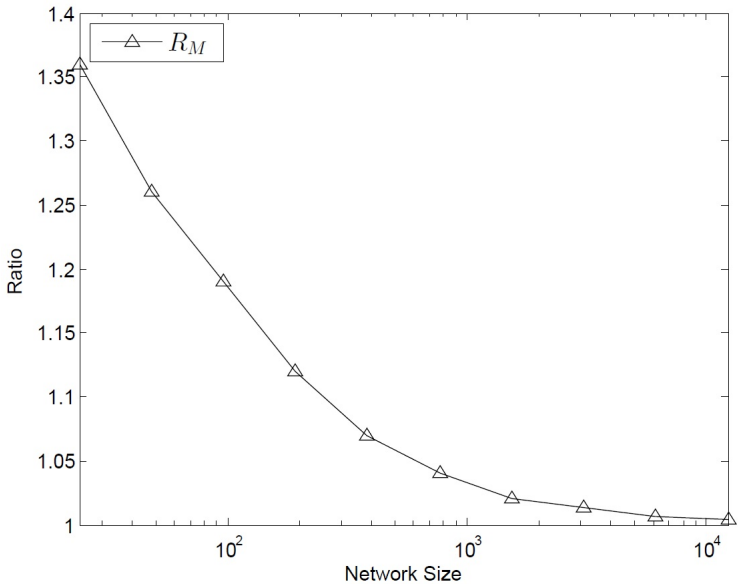
Algorithm 10 Two-dimensional random side query rectangle generation algorithm.

```
1: procedure 2dQueryRectangle(Kautz network K)
2:     issuerIndex ← a uniform random number in [0, K.Nodes.Count − 1]
3:     Q.issuernode ← K.Nodes[issuerIndex]
4:     Q_L ← a uniform random point in [0, 1]^2 as Q's lower left corner
5:     Δ_x, Δ_y ← uniform random values ∈ [0, 1]
6:     Q_xU ← min(1, Q_xL + Δ_x)
7:     Q_yU ← min(1, Q_yL + Δ_y)
8:     return Q
9: end procedure
```

Figures 4.2 (a) and (b) show the impact of network size on average search cost $\bar{K}_Q$, $\bar{n}_Q$ and R_M in 6-dimensional space. An extended form of Algorithm 10 appropriate for 6-dimensional space is used for generation of query rectangles. The network size n varies from 24 to 12,288 nodes. Figures 4.2 (a) and (b) confirm that independent from the number of dimensions, the number of required messages to process a randomly

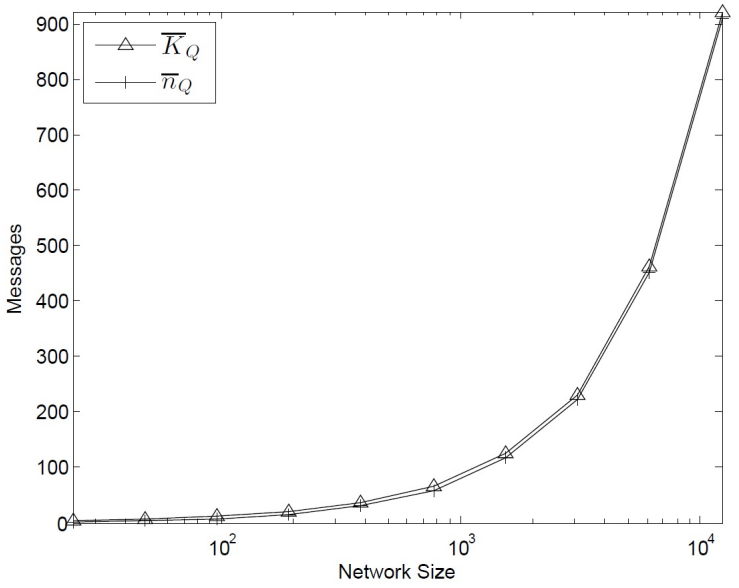


(a)

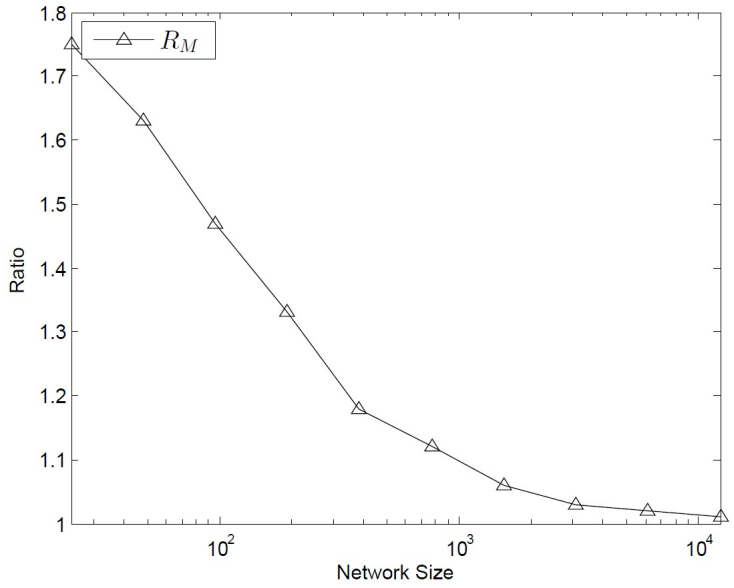


(b)

Figure 4.1: Variation of network size and its impact for 2-dimensional random queries on (a) the average search cost $\bar{K}_Q$ and average no. of nodes intersecting a query $\bar{n}_Q$, and (b) $R_M = \frac{\bar{K}_Q}{\bar{n}_Q}$.



(a)

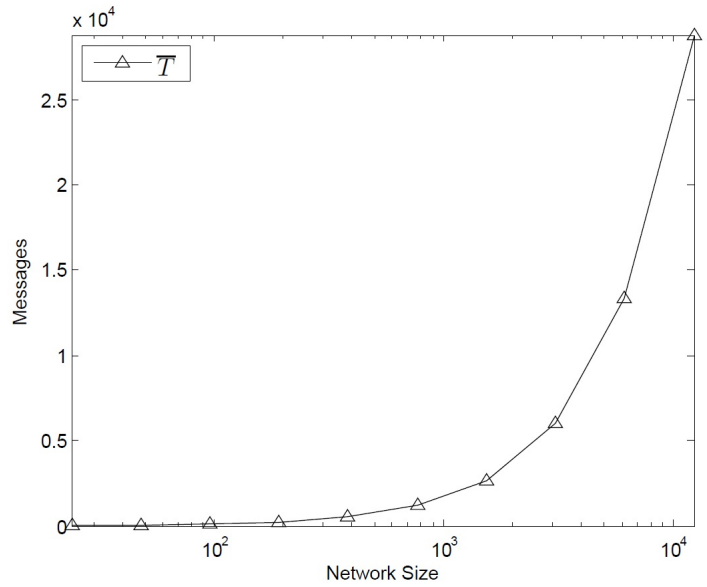


(b)

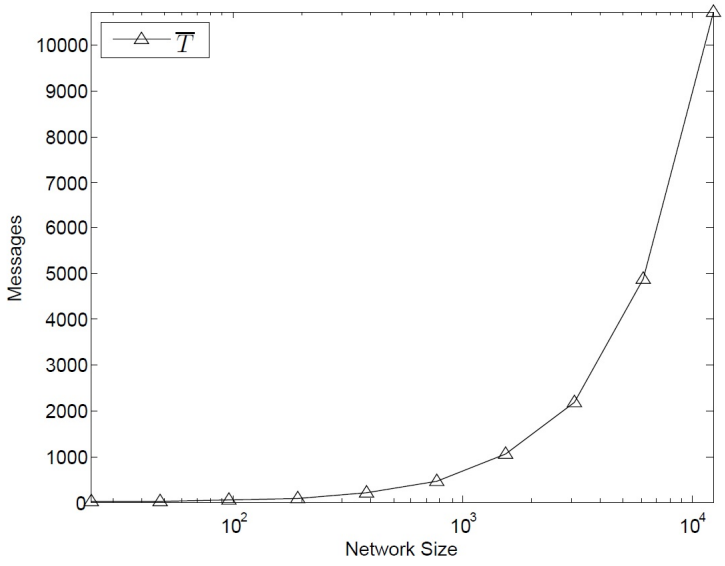
Figure 4.2: Variation of network size and its impact for 6-dimensional random queries on (a) the average search cost $\bar{K}_Q$ and average no. of nodes intersecting a query $\bar{n}_Q$, and (b) $R_M = \frac{\bar{K}_Q}{\bar{n}_Q}$.

generated query is dominated by the number of messages required for passing query Q to the nodes in range. In addition, by comparing Figures 4.1 (a) and 4.2 (a), we observe that when the number of dimensions d increases, for the same number of nodes in the network and the same query generation algorithm, average search cost $\bar{K}_Q$ and $\bar{n}_Q$ decrease. The rational behind this result is that in Algorithm 6, we find the most proper dimension of a query with the minimum number of nodes in range to process the query. When the number of dimensions increases, the probability of having a smaller side in one of the dimensions of a query increases. Furthermore, by comparing Figures 4.1 (b) and 4.2 (b), we observe that when the number of dimensions increases, message cost ratio, R_M, increases. Based on Theorem 3.5.2, the difference of $\bar{K}_Q$ and $\bar{n}_Q$ is equal to $O(\log n)$. For a constant number of nodes n, when the number of dimensions increases, $\bar{K}_Q$ and $\bar{n}_Q$ decrease and $O(\log n)$ is constant. Since $R_M = \frac{\bar{K}_Q}{\bar{n}_Q}$ and $\bar{n}_Q \leq \bar{K}_Q$, when the number of dimensions increases, R_M increases.

In Figures 4.3 (a) and (b), the cost of reporting data in range to the query issuer node ($\bar{T}$) are shown. Based on Theorem 3.5.3, the reporting cost in a network of n nodes for a query that intersects m nodes is $\sum_{i=1}^{m} \lceil \frac{K_i}{B} \rceil O(\log n)$ messages. Since we assume that all the points in range at one node can be reported by one message, $\sum_{i=1}^{m} \lceil \frac{K_i}{B} \rceil$ is equal to the number of nodes in range m. As explained before, increasing the number of dimensions in our data structure results in a decrease in the minimum number of nodes in range. So the number of required messages to report data back to the query issuer node ($\bar{T}$) decreases when the number of dimensions (d) increases. Tables 4.1 and 4.2 compare the the experimental results of reporting cost from Figures 4.3 (a) and (b) and the theoretical results from Theorem 3.5.3, respectively, for 2 and 6-dimensional spaces. Since we have the assumption of reporting



(a)



(b)

Figure 4.3: Variation of network size and its impact on reporting cost $\bar{T}$ of (a) 2-dimensional and (b) 6-dimensional rectangular range queries.

all the points in range at one node by one message, we can assume that $\frac{K_i}{B} = 1$ and the reporting cost is equal to $mO(\log n)$. Based on this assumption, the theoretical results in Tables 4.1 and 4.2 are computed by $\bar{n}_Q \log_2 n$, where $\bar{n}_Q$ is the average number of nodes in range.

Table 4.1: Actual number of messages counted vs. the theoretical number of messages required for reporting data back to the query issuer in 2-dimensional space.

Network size n	Experimental results $\bar{T}$	Theoretical results $\bar{n}_Q \log_2 n$
24	17.14	25.92
48	42.98	60.82
96	95.9	129.69
192	214.83	279.09
384	534.03	674.55
768	1,180.8	1,451.37
1,532	2,651.23	3,195.47
3,072	6,004.05	7,085.68
6,144	13,381.08	15,595.55
12,288	28,805.9	33,173.49

Table 4.2: Actual number of messages counted vs. the theoretical number of messages required for reporting data back to the query issuer in 6-dimensional space.

Network size n	Experimental results $\bar{T}$	Theoretical results $\bar{n}_Q \log_2 n$
24	7.78	12.09
48	17.68	25.05
96	39.20	53.16
192	86.75	111.80
384	212.39	266.20
768	454.83	559.95
1,532	1,036.71	1,248.54
3,072	2,189.41	2,581.99
6,144	4,898.338	5,693.33
12,288	10,742.4	12,381.97

Algorithm 11 Constant volume query rectangle generation algorithm in d-dimensional space.

1: **procedure** ContsantVQuery(Volume v, Number of dimensions d, Kautz network K)
2: $issuerIndex \leftarrow$ a uniform random number in $[0, K.Nodes.Count - 1]$
3: $Q.issuernode \leftarrow K.Nodes[issuerIndex]$
4: $Q_L \leftarrow$ a uniform random point in $[0, 1]^d$ as Q's lower corner.
5: $v' \leftarrow 1$
6: **for** $i \leftarrow 1, d-1$ **do**
7: $\Delta_i \leftarrow$ uniform random values $\in [0, 1]$
8: $Q_{iU} \leftarrow Q_{iL} + \Delta_i$
// Q_{iU}: query upper corner in dimension i
// Q_{iL}: query lower corner in dimension i
9: $v' \leftarrow v' * \Delta_i$
10: **end for**
11: $\Delta_d \leftarrow \frac{v}{v'}$
12: $Q_{dU} \leftarrow Q_{dL} + \Delta_d$
// Q_{dU}: query upper corner in dimension d
// Q_{dL}: dimension d's lower corner
13: **if** $Q_{dU} > 1$ **then**
14: Clear Q and go to line 2
15: **end if**
16: **return** Q
17: **end procedure**

Figures 4.4 (a) and (b) illustrate the impact of query volume variations on average search cost $\bar{K}_Q$. The network size is 6,144 nodes and the query volume varies from 0.05^d to 0.4^d where in Figure 4.4 (a), $d = 2$ and in Figure 4.4 (b), $d = 6$. Test harness Algorithm 12 shows how the volume of queries is changed and constant volume queries (Algorithm 11) are generated. Algorithm 11 makes sure that a query with constant volume v is generated while each query side length is between 0 and 1. From Figures 4.4 (a) and (b), it can be observed that an increase in query volume increases the average range query cost $\bar{K}_Q$ and the average number of nodes in range $\bar{n}_Q$. In addition, when the number of dimensions d increases, $\bar{K}_Q$ and $\bar{n}_Q$ decrease. The reason for this relationship is that in our range processing algorithm, the most proper dimension of a query with the minimum number of nodes in range is chosen. When the number of dimensions increases, the probability of having a smaller side in one of the dimensions of a query increases. Tables 4.3 and 4.4 show the range search cost in messages and the number of nodes in range , respectively, in 2 and 6-dimensional space. In Tables 4.3 and 4.4, it is shown that the difference of the range search cost and the number of nodes intersecting the query is about 10. Based on the theoretical analysis, the difference of the range search cost and the number of nodes intersecting the query is $O(\log n)$. Since in the experiments of Tables 4.3 and 4.4 the network size is fixed to 6,144 nodes and $\log_2 6,144 = 12.58$, the experimental results match the theoretical results.

Algorithm 12 The test harness algorithm for generation of constant volume range queries.

```
1: procedure TestHarness2(numofDim d, numofQueries c)
2:     for s ← 0.05, 0.4 do
3:         v ← s^d
4:         for i ← 1, c do
5:             Q ← ConstantVQuery(v, d)
6:         end for
7:     end for
8: end procedure
```

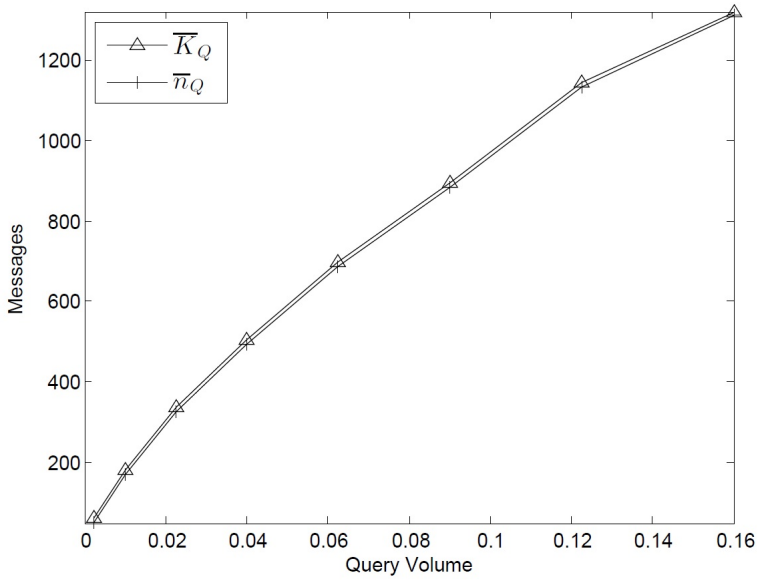
Table 4.3: The range search cost in messages vs. the number of nodes in range in 2-dimensional space.

Query volume v	Range search cost $\bar{K}_Q$	Number of nodes in range $\bar{n}_Q$
0.05^2	61.52	51.71
0.1^2	181.44	171.53
0.15^2	336.48	326.69
0.2^2	504.8	495.03
0.25^2	696.94	687.13
0.3^2	895.05	885.12
0.35^2	1,143.71	1,134.00
0.4^2	1,319.128	1,309.349

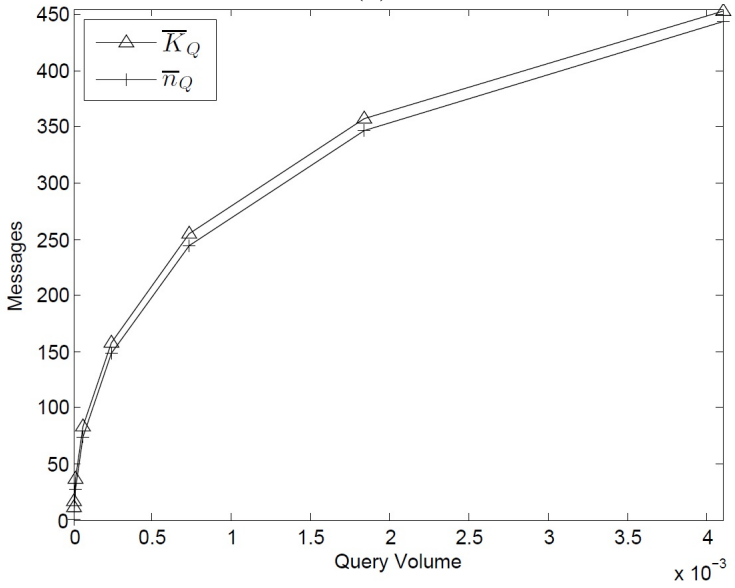
Table 4.4: The range search cost in messages vs. the number of nodes in range in 6-dimensional space.

Query volume v	Range search cost $\bar{K}_Q$	Number of nodes in range $\bar{n}_Q$
0.05^6	11.01	1.25
0.1^6	16.62	6.82
0.15^6	36.71	26.97
0.2^6	83.1	73.38
0.25^6	157.62	147.85
0.3^6	254.34	244.58
0.35^6	356.61	346.85
0.4^6	453.35	443.65

Figure 4.5 shows the average range query cost $\bar{K}_Q$ in 6-dimensional space where queries are generated by two different query generation algorithms ; i.e. constant volume and cubic. The network size n varies from 24 to 12,288 nodes and the volume of queries is fixed to $(0.2)^6 = 0.000064$. It can be observed that $\bar{K}_Q$ for constant volume queries generated by Algorithm 11 is higher than $\bar{K}_Q$ for cubic queries generated by Algorithm 13. In Algorithm 11, queries are random side length



(a)



(b)

Figure 4.4: Variation of query volume and its impact the query cost $\bar{K}_Q$ and $\bar{n}_Q$ for (a) 2-dimensional space, and (b) 6-dimensional space with n =6,144 nodes.

Algorithm 13 Cubic query rectangle generation algorithm in d-dimensional space.

1: **procedure** CUBICQUERY(Range Size Δ_l , Number of dimensions d, Kautz network K)
2: $issuerIndex \leftarrow$ a uniform random number in $[0, K.Nodes.Count - 1]$
3: $Q.issuernode \leftarrow K.Nodes[issuerIndex]$
4: $Q_L \leftarrow$ a uniform random point in $[0, (1 - \Delta_l)]^d$ as Q's lower corner
5: **for** $i \leftarrow 1, d$ **do**
6: $Q_{iU} \leftarrow Q_{iL} + \Delta_l$
// Q_{iU}: query upper corner in dimension i
// Q_{iL}: query lower corner in dimension i
7: **end for**
8: **return** Q
9: **end procedure**

but in Algorithm 13, queries are all boxes with equal side length. We have observed in chapter 3 that query boxes with all sides of equal length are the worst case for our data structure. Due to the rectangular slab space partitioning of our data structure, and the processing of minimum query side length first, processing constant volume queries with random side length is faster than processing cubic queries with equal side length. Tables 4.5 and 4.6 show the actual number of messages counted versus the theoretical number of messages required for constant volume queries and cubic queries, respectively. For the theoretical results, we have computed $\log_2 n$ for each experiment as the number of nodes a query needs to pass through to find the first node in range. In addition, we have used $\bar{n}_Q$ in the experimental results as an estimation of m, the number of nodes in range.

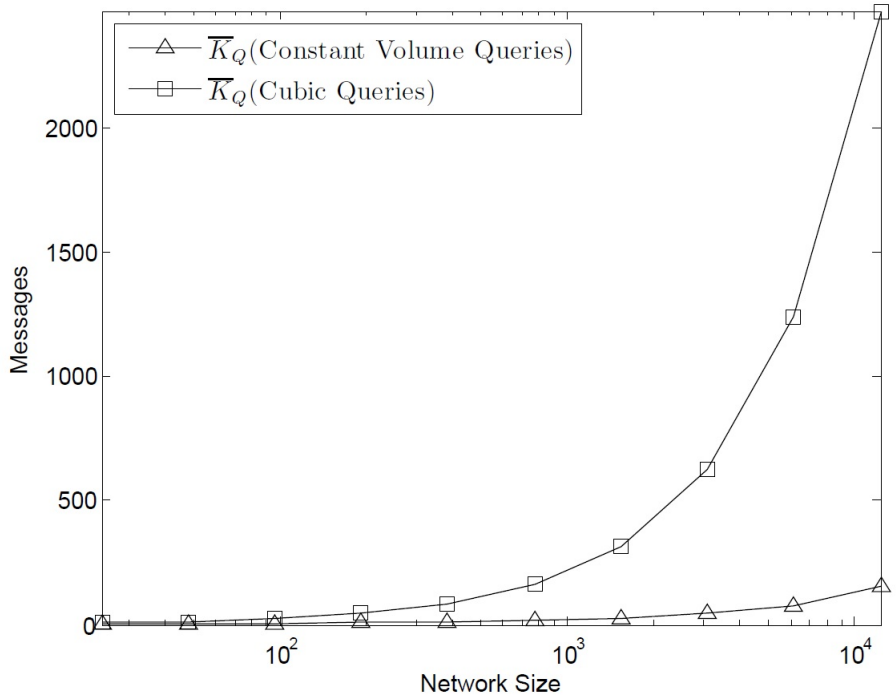


Figure 4.5: Variation of network size and its impact on the query cost $\bar{K}_Q$ for 6-dimensional space.

Table 4.5: Actual number of messages counted vs. the theoretical number of messages required for range search of constant volume queries.

Network size n	Experimental results $\bar{K}_Q$	Theoretical results $\log_2 n + \bar{n}_Q$
24	3.2	5.84
48	4.3	7.08
96	5.99	4.12
192	8.14	10.86
384	11.37	14.05
768	16.6	19.44
1,532	25.58	28.37
3,072	46	48.76
6,144	79.25	82
12,288	155.72	158.5

Table 4.6: Actual number of messages counted vs. the theoretical number of messages required for range search of cubic queries.

Network size n	Experimental results $\bar{K}_Q$	Theoretical results $\log_2 n + \bar{n}_Q$
24	7.78	10.38
48	13.43	16.14
96	23.95	26.78
192	44.20	47
384	83.61	86.4
768	161.39	164.21
1,532	315.92	318.81
3,072	624.13	626.99
6,144	1,239.57	1,242.37
12,288	2,469.33	2,482.91

Chapter 5

Summary and Conclusion

We have designed and implemented a dynamic peer-to-peer data structure for d-dimensional data that is capable of efficient orthogonal range search on a set of N points. The aim of using a distributed model for orthogonal range search is to provide increased reliability, flexibility and robustness to large scale data stores. FissionE topology was used to coordinate message passing among nodes. In FissionE the identifier of nodes are Kautz strings and network nodes are initiated to a Kautz graph. All Kautz graphs have a Hamiltonian path. Our novel idea, as far as we are aware, is to use the Hamiltonian path in the underlying Kautz graph of FissionE to distribute the N points among the n nodes of the peer-to-peer network.

Two main components of our work are the data distribution algorithm and range query processing. Our data distribution algorithm publishes d copies of each object on d different nodes in a way that preserves data locality. To process a range query, our range search algorithm efficiently forwards queries to the appropriate nodes in range. In the IssueQuery algorithm, we first use the FissionE routing algorithm to find the first node in range, and then this node sends the updated query to the next node on the Hamiltonian path.

We have proved that the worst case orthogonal range search cost for any distribution in our data structure is $O(\log n + m)$ messages plus the reporting cost, where n is the number of nodes in the peer-to-peer network and m is the minimum number of nodes intersecting the query. In addition, we have proved that in our data structure the cost of reporting data in range to the query issuer is $\sum_{i=1}^{m} \lceil \frac{K_i}{B} \rceil O(\log n) \in O((\frac{K}{B} + m) \log n)$ messages, where K is the number points in range, K_i is the number of points in range stored in node i, and B is the number of points fitting in one message.

To the best of our knowledge, our data structure is the first distributed dynamic spatial data structure to fully support orthogonal range search with simultaneous failure of $d-1$ nodes. In comparison with Armada, our work contains a theoretical analysis for the cost of orthogonal range search in our data structure. In [28], it has been shown the lower bound on the diameter of a constant-degree graph is $\Omega(\log n)$. Based on this, it has been proven in [15] that the lower bound on the message cost of general range query schemes on constant-degree distributed hash tables (DHTs) is $\Omega(\log n) + m - 1$ where m is the number of peers intersecting the query. We have achieved a search cost $O(\log n + m)$ messages which is close to the lower bound on range search cost in d-dimensional space over a constant-degree DHT. Furthermore, we use redundancy to completely process all queries when up to $d-1$ nodes fail simultaneously.

Table 5.1 shows the costs of our fault tolerant data structure compared to the distributed spatial data structure (DSDS) [6].

Split large and merge small policies are used to support dynamic joining and departure of nodes. When we delete a node from our network, another node in the network takes the responsibility of deleted node data. When a node fails, data

Table 5.1: Performance comparison of our data structure and DSDS.

Parameters	DSDS	Our data structure
Number of dimensions supported	2	d
Point search cost	$O(\log n)$	$O(\log n)$
Range search cost in the worst case	$O(n)$	$O(\log n + m)$
Expected range search cost	$\Theta(\log n + n\sqrt{\alpha})$	$\Theta(\log n + m)$
Storage space	$n + N$	$n + dN$
Cost of recovery after failure of a single node	$O(\frac{N}{nB} + \log n)$	$O(\frac{dN}{nB} \log n)$

replication in our data structure is used to provide complete answers to queries. A failure recovery method was introduced for the our data structure to support simultaneous failure of $d-1$ nodes. It is proven that the cost of recovering network topology and data after failure of one node in d-dimensional space is $O(\frac{dN}{nB} \log n)$ messages.

We also presented an experimental simulation with up to 12,288 nodes that demonstrates the practical application of our data structure. The test data is drawn from a uniform random distribution. In addition, we have performed our experiments for 2 and 6-dimensional spaces and compared our results with our theoretical analysis. We used three different query generation algorithms to evaluate performance of our data structure; random side length query generation algorithm, constant volume query generation algorithm and equal side length or cubic query generation algorithm. The experimental results validate the claimed theoretical bounds on the messages required for orthogonal range search and reporting data back to the query issuer node.

References

[1] Peyman Afshani, Lars Arge, and Kasper Dalgaard Larsen, *Orthogonal range reporting in three and higher dimensions*, Foundations of Computer Science, 2009. FOCS'09. 50th Annual IEEE Symposium on, IEEE, 2009, pp. 149–158.

[2] Pankaj K. Agarwal, *Range searching*, Handbook of Discrete Computational Geometry (1997).

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

[4] James Aspnes and Gauri Shah, *Skip graphs*, ACM Transactions on Algorithms (TALG) **3** (2007), no. 4, 37.

[5] Ashwin R Bharambe, Mukesh Agrawal, and Srinivasan Seshan, *Mercury: supporting scalable multi-attribute range queries*, ACM SIGCOMM Computer Communication Review **34** (2004), no. 4, 353–366.

[6] Pouya Bisadi, *A distributed spatial data structure*, Master's thesis, Universsity of New Brunswick, 2012.

[7] Pouya Bisadi and Bradford G Nickerson, *Orthogonal range search using a distributed computing model.*, CCCG, 2011.

[8] Adina Crainiceanu, Prakash Linga, Johannes Gehrke, and Jayavel Shanmugasundaram, *P-tree: a p2p index for resource discovery applications*, Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters, ACM, 2004, pp. 390–391.

[9] Mark De Berg, Otfried Cheong, Marc Van Kreveld, and Mark Overmars, *Computational geometry*, 3rd ed., Springer, 2008.

[10] Michael T Goodrich, Michael J Nelson, and Jonathan Z Sun, *The rainbow skip graph: a fault-tolerant constant-degree distributed data structure*, Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm, ACM, 2006, pp. 384–393.

[11] ———, *The rainbow skip graph: A fault-tolerant constant-degree p2p relay structure*, arXiv preprint arXiv:0905.2214 (2009).

[12] Abhishek Gupta, Divyakant Agrawal, and Amr El Abbadi, *Approximate range selection queries in peer-to-peer systems.*, CIDR, vol. 3, 2003, pp. 141–151.

[13] Nicholas JA Harvey, Michael B Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman, *Skipnet: A scalable overlay network with practical locality properties.*, USENIX Symposium on Internet Technologies and Systems, vol. 274, Seattle, WA, USA, 2003.

[14] Joseph O'Rourke Jacob E. Goodman (ed.), *Handbook of discrete computational geometry*, 2nd ed., CRC Press, 2004.

[15] Dongsheng Li, Jiannong Cao, Xicheng Lu, and Kaixian Chan, *Efficient range query processing in peer-to-peer systems*, IEEE Transactions on Knowledge and Data Engineering **21** (2009), no. 1, 78–91.

[16] Dongsheng Li, Xicheng Lu, and Jie Wu, *Fissione: A scalable constant degree and low congestion dht scheme based on kautz graphs*, INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE, vol. 3, IEEE, 2005, pp. 1677–1688.

[17] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, Steven Lim, et al., *A survey and comparison of peer-to-peer overlay network schemes.*, IEEE Communications Surveys and Tutorials **7** (2005), no. 1-4, 72–93.

[18] Jiří Matoušek, *Geometric range searching*, ACM Computing Surveys (CSUR) **26** (1994), no. 4, 422–461.

[19] J Erickson Pankaj K. Agarwal, *Geometric range searching and its relatives*, Advances in Discrete and Computational Geometry, Citeseer, 1999.

[20] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker, *A scalable content-addressable network*, vol. 31, ACM, 2001.

[21] Sylvia Ratnasamy, Ion Stoica, and Scott Shenker, *Routing algorithms for dhts: Some open questions*, Peer-to-peer systems, Springer, 2002, pp. 45–52.

[22] Antony Rowstron and Peter Druschel, *Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems*, Middleware 2001, Springer, 2001, pp. 329–350.

[23] Cristina Schmidt and Manish Parashar, *Enabling flexible queries with guarantees in p2p systems*, IEEE Internet Computing **8** (2004), no. 3, 19–26.

[24] Yanfeng Shu, Beng Chin Ooi, K-L Tan, and Aoying Zhou, *Supporting multi-dimensional range queries in peer-to-peer systems*, Peer-to-Peer Computing, 2005. P2P 2005. Fifth IEEE International Conference on, IEEE, 2005, pp. 173–180.

[25] R. Simion, *Noncrossing partitions*, Discrete Mathematics **217** (2000), no. 1, 367–409.

[26] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan, *Chord: A scalable peer-to-peer lookup service for internet applications*, ACM SIGCOMM Computer Communication Review **31** (2001), no. 4, 149–160.

[27] George Tsatsanifos, Dimitris Sacharidis, and Timos Sellis, *Midas: multi-attribute indexing for distributed architecture systems*, Advances in Spatial and Temporal Databases, Springer, 2011, pp. 168–185.

[28] Jun Xu, Abhishek Kumar, and Xingxing Yu, *On the fundamental tradeoffs between routing table size and network diameter in peer-to-peer networks*, Selected Areas in Communications, IEEE Journal on **22** (2004), no. 1, 151–163.

[29] Kevin C Zatloukal and Nicholas JA Harvey, *Family trees: an ordered dictionary with optimal congestion, locality, degree, and search time*, Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, 2004, pp. 308–317.

[30] Yiming Zhang, Xicheng Lu, and Dongsheng Li, *Survey of dht topology construction techniques in virtual computing environments*, Science China Information sciences **54** (2011), no. 11, 2221–2235.

[31] Ben Y Zhao, Ling Huang, Jeremy Stribling, Sean C Rhea, Anthony D Joseph, and John D Kubiatowicz, *Tapestry: A resilient global-scale overlay for service deployment*, Selected Areas in Communications, IEEE Journal on **22** (2004), no. 1, 41–53.

Appendix A

Kautz Network Code

This code corresponds to the MAKEGRAPH call in line 2 of Algorithm 9 in chapter 4.

```
1  public List<Node> Nodes { get; set; }
       public List<List<int>> Conections = new List<List<int>>();
3      public KautzGraph(int nodecount, int nodeidlength, int
           pointcount, int d)

5      {
           List<int> input = new List<int>();
7          Nodes=new List<Node>();
           // generateing n nodes of network with their specific
               kautz strings as their identifiers
9          nodeID(input, nodeidlength);
           // finding a Hamiltonian path on the generated Kautz string
11
           List<int>[] graph = new List<int>[Nodes.Count];
13
           for (int i = 0; i < Nodes.Count; i++)
15             graph[i] = new List<int>();

17         for (int j = 0; j < Nodes.Count; j++){
           foreach(string s in Nodes[j].outNeighbors)
19             {
               for(int k=0;k<Nodes.Count;k++)
21             {
                   if (Nodes[k].KautzString.Equals(s))
23                 {
                       graph[j].Add(k);
25                     break;
                   }
27             }
               }
29             }
           List<int> HamiltonianCycle = Algorithm(graph);
31         //change the order of nodes in Nodes list
           List<Node> templist = new List<Node>();
33         for (int m = 0; m < Nodes.Count;m++ )
           {
```

```
35                templist.Add(Nodes[HamiltonianCycle[m]]);

37            }
              Nodes = templist;
39
              for (int n = 0; n < Nodes.Count;n++ )
41            {
                  Nodes[n].HamPath = Nodes;
43                Nodes[n].PathIndex = n;
              }
45                //distribution of points among nodes based on their
                      order on the Hamiltonian path.

47            if(nodecount!= (3*Math.Pow(2,nodeidlength-1)))
              {
49                int currentNum=(int)(3*Math.Pow(2,nodeidlength-1));
                  while (currentNum < nodecount)
51                {
                      addition();
53                    currentNum++;
                  }
55            }
              Fill_Nodes_PointLists(pointcount, d);
57
          }
59


61
          public void nodeID(List<int> input, int k)
63        {
              if (input.Count == 0)
65            {
                  input.Add(0);
67                nodeID(input, k - 1);
                  input.Add(1);
69                nodeID(input, k - 1);
                  input.Add(2);
71                nodeID(input, k - 1);
              }
73            else if (k == 0)
              {
75                string ks=null;
                  // converting an array list input of ints to an
                      integer ks.
77                for (int j = 0; j< input.Count;j++ )
                  {
79
                      ks = ks + input[j].ToString();
81                }
                  //insert a new node with nodeID ks.
83                    Nodes.Add(new Node());
                  (Nodes[Nodes.Count - 1]).KautzString = ks;
85                //updating outneighbours of new inserted node.
                  switch((input[input.Count - 1])){
```

```
87
                    case(0):
89                      (Nodes[Nodes.Count -
                            1]).outNeighbors.Add(ks.Substring(1,
                            ks.Length - 1) + "1");
                        (Nodes[Nodes.Count -
                            1]).outNeighbors.Add(ks.Substring(1,
                            ks.Length - 1) + "2");
91                      break;

93                  case (1):
                        (Nodes[Nodes.Count -
                            1]).outNeighbors.Add(ks.Substring(1,
                            ks.Length - 1) + "0");
95                      (Nodes[Nodes.Count -
                            1]).outNeighbors.Add(ks.Substring(1,
                            ks.Length - 1) + "2");
                        break;
97
                    case (2):
99                      (Nodes[Nodes.Count -
                            1]).outNeighbors.Add(ks.Substring(1,
                            ks.Length - 1) + "0");
                        (Nodes[Nodes.Count -
                            1]).outNeighbors.Add(ks.Substring(1,
                            ks.Length - 1) + "1");
101                     break;
                }
103
                //updating inneighbours of new inserted node.
105             switch ((input[0]))
                {
107
                    case (0):
109                     (Nodes[Nodes.Count -
                            1]).inNeighbors.Add("1"+ks.Substring(0,
                            ks.Length - 1) );
                        (Nodes[Nodes.Count - 1]).inNeighbors.Add("2" +
                            ks.Substring(0, ks.Length - 1));
111                     break;

113                 case (1):
                        (Nodes[Nodes.Count - 1]).inNeighbors.Add("0" +
                            ks.Substring(0, ks.Length - 1));
115                     (Nodes[Nodes.Count - 1]).inNeighbors.Add("2" +
                            ks.Substring(0, ks.Length - 1));
                        break;
117
                    case (2):
119                     (Nodes[Nodes.Count - 1]).inNeighbors.Add("0" +
                            ks.Substring(0, ks.Length - 1));
                        (Nodes[Nodes.Count - 1]).inNeighbors.Add("1" +
                            ks.Substring(0, ks.Length - 1));
121                     break;
```

```
                }
123             input.RemoveAt(input.Count - 1);
                return;
125         }
            else if (input[input.Count - 1] == 0)
127         {
                input.Add(1);
129             nodeID(input, k - 1);
                input.Add(2);
131             nodeID(input, k - 1);
                input.RemoveAt(input.Count - 1);
133

135         }
            else if (input[input.Count - 1] == 1)
137         {
                input.Add(0);
139             nodeID(input, k - 1);
                input.Add(2);
141             nodeID(input, k - 1);
                input.RemoveAt(input.Count - 1);
143

            }
145         else if (input[input.Count - 1] == 2)
            {
147             input.Add(0);
                nodeID(input, k - 1);
149             input.Add(1);
                nodeID(input, k - 1);
151             input.RemoveAt(input.Count - 1);

153         }

155     }
        private void Fill_Nodes_PointLists(int pointCount, int d)
157     {
            float slob = 1 / (float)Nodes.Count; //The size of each
                node range
159         Random r = new Random(0);
            //number of points in each node
161         int count = pointCount / Nodes.Count;
            for (int index = 0; index < Nodes.Count; index++)
163         {
                for (int i = 0; i < d; i++)
165             {
                    // we assume that the entire interval for each
                        dimension is 1. We can change that if we want.
167                 Nodes[index].Lower.Add((float)(index * slob));

169             }
                for (int i = 0; i < d; i++)
171             {
                    Nodes[index].Upper.Add((index + 1) * slob);
173
```

```
                }
175
            }
177
            for(int j=0; j<pointCount;j++)
179         {
                List<float> temp = new List<float>();
181             List<int> nodeIndex = new List<int>();
                for (int i = 0; i < d;i++ )
183             {
                    temp.Add( ((float)r.Next(0, int.MaxValue) /
                        int.MaxValue ));
185             }
                for(int l=0; l<d;l++)
187             {
                   int node= (int)(Math.Ceiling(temp[l] *
                       (Nodes.Count)))-1;
189                for (int p = 0; p < l; p++)
                   {
191                    if (nodeIndex[p] == node)
                           node = (node + 1) % Nodes.Count;
193                }
                       nodeIndex.Add(node);
195             }

197         for(int m=0;m<d;m++)
            {
199             Nodes[nodeIndex[m]].Points.Add(temp);
            }
201         }

203     }

205     //addition of one node to the network
        private void addition()
207     {
            Node newnode = new Node();
209         int min_l=1000;
            int min_l_index = 0;
211         //finding the node with minimum length of Kautz String
            for(int i=0;i<Nodes.Count;i++)
213         {
                if (Nodes[i].KautzString.Length < min_l)
215             {
                    min_l = Nodes[i].KautzString.Length;
217                 min_l_index = i;
                }
219         }

221         //changing the kautz sting of old and new nodes.
            switch (Nodes[min_l_index].KautzString[Nodes[min_l_index].
223         KautzString.Length - 1])
            {
225             case('0'):
```

```
                    {
227                     char[] all = new char[] { '0', '1', '2' };
                        char[] others = new char[] { '0',
                            Nodes[min_l_index + 1].KautzString[min_l-1]
                            };
229                     List<char> list = new
                            List<char>(all.Except(others));
                        newnode.KautzString =
                            Nodes[min_l_index].KautzString + list[0];
231                     Nodes[min_l_index].KautzString =
                            Nodes[min_l_index].KautzString +
                            Nodes[min_l_index + 1].KautzString[min_l -
                            1];

233                     break;
                    }
235
                case ('1'):
                    {
237                     char[] all = new char[] { '0', '1', '2' };
                        char[] others = new char[] { '1',
239                         Nodes[min_l_index + 1].KautzString[min_l -
                            1] };
                        List<char> list = new
                            List<char>(all.Except(others));
241                     newnode.KautzString =
                            Nodes[min_l_index].KautzString + list[0];
                        Nodes[min_l_index].KautzString =
                            Nodes[min_l_index].KautzString +
                            Nodes[min_l_index + 1].KautzString[min_l -
                            1];
243                     break;

245                 }

247             case ('2'):
                    {
249                     char[] all = new char[] { '0', '1', '2' };
                        char[] others = new char[] { '2',
                            Nodes[min_l_index + 1].KautzString[min_l -
                            1] };
251                     List<char> list = new
                            List<char>(all.Except(others));
                        newnode.KautzString =
                            Nodes[min_l_index].KautzString + list[0];
253                     Nodes[min_l_index].KautzString =
                            Nodes[min_l_index].KautzString +
                            Nodes[min_l_index + 1].KautzString[min_l -
                            1];
                        break;
255                 }
            }
257         //updating outneigbours of nodes.
            List<string> oldnodenewoutNeighbors = new List<string>();
```

```
259             foreach (string s in Nodes[min_l_index].outNeighbors)
                {
261
                    if (s[s.Length -
                        1].Equals(newnode.KautzString[newnode.KautzString.Length-1]))
263                     newnode.outNeighbors.Add(s);
                    if (s[s.Length -
                        1].Equals(Nodes[min_l_index].KautzString[Nodes[min_l_index]
265                 .KautzString.Length-1]))
                        oldnodenewoutNeighbors.Add(s);
267             }
                // updating neghbours of outnegbours of splited node.
269             foreach (string s in Nodes[min_l_index].outNeighbors)
                {
271                 int i = 0;
                    for( i=0;i<Nodes.Count;i++)
273                 {
                        if (Nodes[i].KautzString.Equals(s))
275                         break;
                    }
277                 Nodes[i].inNeighbors.Remove(Nodes[min_l_index].KautzString.
                    Substring(0, min_l));
279                 if(s[s.Length -
                        1].Equals(newnode.KautzString[newnode.KautzString.Length-1]))
                    {
281                     Nodes[i].inNeighbors.Add(newnode.KautzString);
                    }
283                 if (s[s.Length -
                        1].Equals(Nodes[min_l_index].KautzString[Nodes[min_l_index].
                    KautzString.Length - 1]))
285                     Nodes[i].inNeighbors.Add(Nodes[min_l_index].KautzString);
                }
287
                // updating neghbours of innegbours of splited node.
289             foreach (string s in Nodes[min_l_index].inNeighbors)
                {
291                 int i = 0;
                    for (i = 0; i < Nodes.Count; i++)
293                 {
                        if (Nodes[i].KautzString.Equals(s))
295                         break;
                    }
297                 Nodes[i].outNeighbors.Remove(Nodes[min_l_index].KautzString.
                    Substring(0, min_l ));
299                 Nodes[i].outNeighbors.Add(newnode.KautzString);
                    Nodes[i].outNeighbors.Add(Nodes[min_l_index].KautzString);
301             }

303                 //updating inneighbours of new and splited node
                Nodes[min_l_index].outNeighbors = oldnodenewoutNeighbors;
305
                newnode.inNeighbors = Nodes[min_l_index].inNeighbors;
307
                //defining updated Hamiltonian path
```

```
309             List<Node> newHamPath = Nodes;
                newHamPath.Insert(min_l_index + 1, newnode);
311
               for(int k=0;k<Nodes.Count;k++)
313            {
                    Nodes[k].HamPath = newHamPath;
315                 Nodes[k].PathIndex = k;
               }
317


319


321        }


323
           // Find the hamiltonian path of a graph (Input: Conections)
325          static List<int>[] graph, oppositeGraph;
           static List<int> HamiltonianCycle;
327        static bool endOfAlgorithm;
           static int level, v1, v2;
329        static List<int> Algorithm(List<int>[] graphArgument)
           {
331            graph = SaveGraph(graphArgument);
               HamiltonianCycle = new List<int>();
333            endOfAlgorithm = false;
               level = 0;
335            RemoveMultipleEdgesAndLoops(graph); //3.1
               CreateOppositeGraph(graph);
337            bool HamiltonianCycleCantExist = AnalyzeGraph(new
                   List<Edge>()); //6.1.a
               ReverseGraph();
339            if (!HamiltonianCycleCantExist)
                   FindHamiltonianCycle(GetNextVertex()); //5.3
341            HamiltonianCycle.Reverse();
               return HamiltonianCycle;
343        }
           static void ReverseGraph()
345        {
               graph = SaveGraph(oppositeGraph);
347            CreateOppositeGraph(graph);
           }
349        static void FindHamiltonianCycle(int a)
           {
351            if (!endOfAlgorithm)
               {
353                ++level;
                   if (HamiltonianCycleFound())
355                    endOfAlgorithm = true;
                   SortList(a); //5.4
357                while (graph[a].Count > 0 && !endOfAlgorithm)
                   {
359                    List<Edge> removedEdges = new List<Edge>();
                       int chosenVertex = graph[a][0];
361                    graph[a].Remove(chosenVertex);
```

```
                    List<int>[] currentGraph = SaveGraph(graph);
363                 #region 6.2
                    foreach (int b in graph[a])
365                 {
                        removedEdges.Add(new Edge(a, b));
367                     oppositeGraph[b].Remove(a);
                    }
369                 graph[a].Clear();
                    #endregion
371                 graph[a].Add(chosenVertex);
                    v1 = a;
373                 v2 = chosenVertex;
                    bool HamiltonianCycleCantExist =
                        AnalyzeGraph(removedEdges); //6.1.b
375                 if (!HamiltonianCycleCantExist)
                    {
377                     FindHamiltonianCycle(GetNextVertex()); //5.5
                        RestoreGraphs(currentGraph); //6.4
379                 }
                    else
381                 {
                        foreach (Edge e in removedEdges) //6.3
383                     {
                            graph[e.from].Add(e.to);
385                         oppositeGraph[e.to].Add(e.from);
                        }
387                     RemoveEdge(new Edge(a, chosenVertex), graph,
                            oppositeGraph);
                    }
389             }
                if (!endOfAlgorithm)
391             {
                    --level;
393                 if (level == 0)
                        endOfAlgorithm = true;
395             }
            }
397     }
        static bool HamiltonianCycleFound()
399     {
            foreach (List<int> list in graph)
401             if (list.Count != 1)
                    return false;
403         HamiltonianCycle = GetHamiltonianCycle(graph);
            return true;
405     }
        static List<int> GetHamiltonianCycle(List<int>[] graphArgument)
407     {
            List<int> cycle = new List<int>() { 0 };
409         while (true)
            {
411             if (cycle.Count == graphArgument.Length &&
                    graphArgument[cycle.Last()].Contains(cycle[0]))
                    return cycle;
```

```
413                 if (cycle.Contains(graphArgument[cycle.Last()][0]))
                        return new List<int>();
415                 else
                        cycle.Add(graphArgument[cycle.Last()][0]);
417             }
            }
419         static int GetNextVertex()
            {
421             List<int> correctOrder = GetCorrectOrder(graph);
                foreach (int a in correctOrder)
423                 if (graph[a].Count != 1)
                        return a;
425             return 0;
            }
427         static bool AnalyzeGraph(List<Edge> removedEdges)
            {
429             bool HamiltonianCycleCantExist = false;
                int a;
431             do
                {
433                 a = removedEdges.Count;
                    HamiltonianCycleCantExist =
                        RemoveUnnecessaryEdges(graph, oppositeGraph,
                        removedEdges, false);
435                 if (!HamiltonianCycleCantExist)
                        HamiltonianCycleCantExist =
                            RemoveUnnecessaryEdges(oppositeGraph, graph,
                            removedEdges, true);
437             }
                while (a != removedEdges.Count &&
                    !HamiltonianCycleCantExist);
439             if (!HamiltonianCycleCantExist)
                    HamiltonianCycleCantExist = GraphIsDisconnected(graph);
441             return HamiltonianCycleCantExist;
            }
443         static bool RemoveUnnecessaryEdges(List<int>[] graphArgument,
                List<int>[] oppositeGraphArgument, List<Edge> removedEdges,
                bool oppositeGraph)
            {
445             bool HamiltonianCycleCantExist = false;
                for (int a = 0; a < graphArgument.Length; ++a)
447             {
                    if (graphArgument[a].Count == 0 //4.1
449                     || (graphArgument[a].Count == 1 &&
                            SearchForCycleAmongVerticesOfDegreeEqual1(graphArgument,
                            a)) //4.2.1
                        || (graphArgument[a].Count > 1 &&
                            SearchForCycleAmongVerticesOfDegreeGreaterThan1(a,
                            graphArgument, oppositeGraphArgument))) //4.2.2
451                     return true;
                    List<Edge> edges = new List<Edge>();
453                 #region 3.2
                    if (graphArgument[a].Count == 1 &&
                        oppositeGraphArgument[graphArgument[a][0]].Count !=
```

```
                    1)
455             {
                    foreach (int c in
                        oppositeGraphArgument[graphArgument[a][0]])
457                     if (c != a)
                            if (!oppositeGraph)
459                             edges.Add(new Edge(c,
                                    graphArgument[a][0]));
                            else
461                             edges.Add(new
                                    Edge(graphArgument[a][0], c));
                }
463             #endregion
                #region 3.4
465             if (graphArgument[a].Count == 1 &&
                    graphArgument[graphArgument[a][0]].Contains(a))
                {
467                 if (!oppositeGraph)
                        edges.Add(new Edge(graphArgument[a][0], a));
469                 else
                        edges.Add(new Edge(a, graphArgument[a][0]));
471             }
                #endregion
473             foreach (Edge edge in edges)
                {
475                 removedEdges.Add(edge);
                    if (!oppositeGraph)
477                     RemoveEdge(edge, graphArgument,
                            oppositeGraphArgument);
                    else
479                     RemoveEdge(edge, oppositeGraphArgument,
                            graphArgument);
                }
481         }
            return HamiltonianCycleCantExist;
483     }
        static bool
            SearchForCycleAmongVerticesOfDegreeEqual1(List<int>[]
            graphArgument, int a)
485     {
            if(!(a==v1 || a == v2))
487             return false;
            List<int> cycle = new List<int>() { a };
489         while (true)
                if (graphArgument[cycle.Last()].Count == 1 &&
                    cycle.Count < graphArgument.Length)
491                 if (cycle.Contains(graphArgument[cycle.Last()][0]))
                        return true;
493                 else
                        cycle.Add(graphArgument[cycle.Last()][0]);
495             else
                    return false;
497     }
```

```
        static bool
            SearchForCycleAmongVerticesOfDegreeGreaterThan1(int a,
            List<int>[] graphArgument, List<int>[]
            oppossiteGraphArgument)
499     {
            if (!ListsAreEqual(graphArgument[a],
                oppossiteGraphArgument[a], true))
501             return false;
            int b = 1;
503         for (int c = 0; c < graphArgument.Length &&
                graphArgument.Length - c > graphArgument[a].Count - b;
                ++c)
            {
505             if (c == a)
                    continue;
507             if (ListsAreEqual(graphArgument[c], graphArgument[a],
                    false) && ListsAreEqual(graphArgument[c],
                    oppositeGraphArgument[c], true))
                    ++b;
509             if (b == graphArgument[a].Count)
                    return true;
511         }
            return false;
513     }
        static bool ListsAreEqual(List<int> firstList, List<int>
            secondList, bool EqualCount)
515     {
            if (EqualCount && firstList.Count != secondList.Count)
517             return false;
            foreach (int a in firstList)
519             if (!secondList.Contains(a))
                    return false;
521         return true;
        }
523     static void SortList(int a)
        {
525         List<int> correctOrder = GetCorrectOrder(oppositeGraph);
            for (int b = 1; b < graph[a].Count; ++b)
527             for (int c = 0; c < graph[a].Count - 1; ++c)
                    if (correctOrder.IndexOf(graph[a][c]) >
                        correctOrder.IndexOf(graph[a][c + 1]))
529         {
                int n = graph[a][c];
531             graph[a][c] = graph[a][c + 1];
                graph[a][c + 1] = n;
533         }
        }
535     static List<int> GetCorrectOrder(List<int>[] graphArgument)
            //5.1
        {
537         Dictionary<int, int> vertices = new Dictionary<int, int>();
            List<int> order = new List<int>();
539         for (int a = 0; a < graphArgument.Length; ++a)
                vertices.Add(a, graphArgument[a].Count);
```

```
541             IEnumerable<int> v = from pair in vertices orderby
                    pair.Value ascending select pair.Key;
                foreach (int a in v)
543                 order.Add(a);
                return order;
545         }
            static void RemoveEdge(Edge e, List<int>[] graphArgument,
                List<int>[] oppositeGraphArgument)
547         {
                graphArgument[e.from].Remove(e.to);
549             oppositeGraphArgument[e.to].Remove(e.from);
            }
551         static void RemoveMultipleEdgesAndLoops(List<int>[]
                graphArgument)
            {
553             for (int a = 0; a < graphArgument.Length; ++a)
                {
555                 graphArgument[a] =
                        graphArgument[a].Distinct().ToList();
                    graphArgument[a].Remove(a);
557             }
            }
559         static void CreateOppositeGraph(List<int>[] graphArgument)
            {
561             oppositeGraph = new List<int>[graphArgument.Length];
                for (int a = 0; a < graphArgument.Length; ++a)
563                 oppositeGraph[a] = new List<int>();
                for (int a = 0; a < graphArgument.Length; ++a)
565                 foreach (int b in graphArgument[a])
                        oppositeGraph[b].Add(a);
567         }
            static void RestoreGraphs(List<int>[] graphArgument)
569         {
                graph = new List<int>[graphArgument.Length];
571             for (int a = 0; a < graphArgument.Length; ++a)
                {
573                 graph[a] = new List<int>();
                    graph[a].AddRange(graphArgument[a]);
575             }
                CreateOppositeGraph(graph);
577         }
            static List<int>[] SaveGraph(List<int>[] graphArgument)
579         {
                List<int>[] savedGraph = new
                    List<int>[graphArgument.Length];
581             for (int a = 0; a < graphArgument.Length; ++a)
                {
583                 savedGraph[a] = new List<int>();
                    savedGraph[a].AddRange(graphArgument[a]);
585             }
                return savedGraph;
587         }
            static bool GraphIsDisconnected(List<int>[] graphArgument)
589         {
```

```
            Stack<int> stack = new Stack<int>();
591         Color[] colors = new Color[graphArgument.Length];
            colors[0] = Color.Gray;
593         stack.Push(0);
            while (stack.Count > 0)
595         {
                int a = stack.Pop();
597             foreach (int b in graphArgument[a])
                {
599                 if (colors[b] == Color.White)
                    {
601                     colors[b] = Color.Gray;
                        stack.Push(b);
603                 }
                }
605             colors[a] = Color.Black;
            }
607         foreach (Color c in colors)
                if (c != Color.Black)
609                 return true;
            return false;
611     }
    }
613 class Edge
    {
615     public int from, to;
        public Edge(int f, int t)
617     {
            from = f;
619         to = t;
        }
```

Appendix B

Routing Messages Code

This code corresponds to the IssueQuery call in line 6 of Algorithm 9 in chapter 4.

```
1  public void IssueQuery(QueryMessage msg)
       {
3          int dstIndex = (int)(Math.Ceiling(msg.Lower[msg.Dim_best]
               * (msg.Issuer.HamPath.Count ))) -1;
           if (dstIndex == -1)
5              dstIndex = 0;
           string dstID = msg.Issuer.HamPath[dstIndex].KautzString;
7          int netsize = msg.Issuer.HamPath.Count;
           int k = 1;
9          while (3 * Math.Pow(2, k - 1) <= netsize)
           {
11             k++;
           }
13         Boolean firstone=false;
           string secondID="";
15         if (dstID.Length == k)
           {
17             int i = 0;
               for (i = 0; i < msg.Issuer.HamPath.Count; i++)
19             {
                   if
                       (msg.Issuer.HamPath[i].KautzString.Equals(dstID))
21                     break;
               }
23             if (dstID.Substring(0, k -
                   1).Equals(msg.Issuer.HamPath[i +
                   1].KautzString.Substring(0, k - 1)))
               {
25                 firstone = true;
                   secondID = msg.Issuer.HamPath[i + 1].KautzString;
27             }

29

               if (firstone)
31             {
                   float newupper = msg.Upper[msg.Dim_best];
```

```
33                      FissionERouting(msg.Issuer.KautzString, dstID,
                            msg, msg.Dim_best);
                        //find the node with ID equals to dstID
35                      int j = 0;
                        for (j= 0; j < msg.Issuer.HamPath.Count; j++)
37                      {
                            if
                                (msg.Issuer.HamPath[j].KautzString.Equals(dstID))
39                              break;
                        }
41
                        if (newupper > msg.Issuer.HamPath[j +
                            1].Lower[msg.Dim_best])
43                      {
                            msg.Lower[msg.Dim_best] =
                                msg.Issuer.HamPath[j].Upper[msg.Dim_best];
45                          msg.Upper[msg.Dim_best] =
                                Math.Min(msg.Issuer.HamPath[j +
                                1].Upper[msg.Dim_best], newupper);
                            FissionERouting(msg.Issuer.KautzString,
                                secondID, msg, msg.Dim_best);
47                      }
                    }
49
                    else
51                  {
                        float oldUpper = msg.Upper[msg.Dim_best];
53                      int j = 0;
                        for (j= 0; j < msg.Issuer.HamPath.Count; j++)
55                      {
                            if
                                (msg.Issuer.HamPath[j].KautzString.Equals(dstID))
57                              break;
                        }
59                      msg.Upper[msg.Dim_best] =
                            msg.Issuer.HamPath[j].Upper[msg.Dim_best];
                        FissionERouting(msg.Issuer.KautzString, dstID,
                            msg, msg.Dim_best);
61
                        msg.Lower[msg.Dim_best] =
                            msg.Issuer.HamPath[j].Upper[msg.Dim_best] +
                            (float)0.001;
63                      msg.Upper[msg.Dim_best] = oldUpper;

65                      msg.Issuer.IssueQuery( msg);

67                  }
                }
69              else
                    FissionERouting(msg.Issuer.KautzString, dstID, msg,
                        msg.Dim_best);
71

73          }
```

```
75
        public void FissionERouting(string src, string dst,
            QueryMessage msg,int Dim_best)
77      {
            Boolean done=false;
79
            string SP=null;
81          string prefix=src;
            while (!done)
83          {
                if (prefix.Length==0)
85                  done = true;
                else
87              {
                    if (dst.StartsWith(prefix))
89                  {

91                      done = true;
                    }
93                  else
                    {
95                      prefix=prefix.Remove(0, 1);
                    }
97              }

99          }

101             SP=prefix;
            this.Routing(dst, src.Length-SP.Length, SP, msg, Dim_best);
103
              }
105     public void Routing(string dst, int PathLength, String SP,
            QueryMessage msg,int Dim_best){

107      if (PathLength==0)
         {
109        if(this.Lower[Dim_best]<=msg.Lower[Dim_best] &&
               this.Upper[Dim_best]>=msg.Lower[Dim_best])
           {
111
           int netsize = msg.Issuer.HamPath.Count;
113        int k = 1;
           while (3 * Math.Pow(2, k - 1) <= netsize)
115          {
               k++;
117          }

119          if (this.KautzString.Length == k)
             {
121            //check if it is first node.
               if (this.KautzString.Substring(0, k - 1).
123                Equals(this.HamPath[this.PathIndex +
                       1].KautzString.Substring(0, k - 1)))
```

```
                {
125
                    if (this.Upper[Dim_best] < msg.Upper[Dim_best])
127                 {

129                     if (this.HamPath[this.PathIndex +
                            1].Upper[Dim_best] <
                            msg.Upper[Dim_best])//intersect
                                thisnode+2
131                     {

133                         if (this.HamPath[this.PathIndex +
                                2].KautzString.Length == k)
                            {
135                          float newupper =
                                msg.Upper[msg.Dim_best];
                             msg.Lower[Dim_best] =
137                              this.HamPath[this.PathIndex +
                                    2].Lower[Dim_best];
                             this.FissionERouting
139                              (this.KautzString,
                                 this.HamPath[this.PathIndex +
                                    2].KautzString, msg, Dim_best);
141


143                          if (newupper >
                                 msg.Issuer.HamPath[this.PathIndex
                                    + 3].Lower[msg.Dim_best])
145                          {
                               // makin a new query msg
147                          msg.Lower[msg.Dim_best] =
                                 msg.Issuer.HamPath[this.PathIndex
                                    + 3].Lower[msg.Dim_best];
149                          msg.Upper[msg.Dim_best] =
                                 Math.Min(msg.Issuer.HamPath[this.PathIndex
                                    + 3].Upper[msg.Dim_best]
151                              ,newupper);

153                          this.FissionERouting(this.KautzString,
                                 this.HamPath[this.PathIndex +
                                    3].KautzString, msg, Dim_best);
155                          }
                                }
157                             else //the length of
                                    pathindex+2 ==k-1
                                {
159                                 msg.Lower[Dim_best] =
                                        this.HamPath[this.PathIndex
                                            +
                                            2].Lower[Dim_best];
161                                 this.FissionERouting(this.
                                    KautzString,
163                                     this.HamPath[this.PathIndex
                                            +
```

```
                                                    2].KautzString,
                                                   msg,
                                                Dim_best);
165                                         }

167                                     }

169                                 }

171                         }
                            else
173                         {

175                             if (this.Upper[Dim_best] <
                                    msg.Upper[Dim_best])
                                {
177                                 msg.Lower[Dim_best] =
                                        this.Upper[Dim_best];

179                                 this.FissionERouting(this.KautzString,
                                        this.HamPath[this.PathIndex +
                                            1].KautzString, msg, Dim_best);
181                             }
                            }
183
                        }
185                     else if (this.KautzString.Length == k - 1)
                        {
187                         if (this.PathIndex + 1 < this.HamPath.Count)
                                //check whether this the last node in ham
                                    path
189                         {
                            if (this.HamPath[this.PathIndex +
                                1].KautzString.Length == k)
191                         {
                                if (this.Upper[Dim_best] <
                                    msg.Upper[Dim_best])
193                             {
                                    msg.Lower[Dim_best] =
                                        this.Upper[Dim_best];
195

197                                 this.FissionERouting(this.KautzString,
                                        this.HamPath[this.PathIndex +
                                            1].KautzString, msg, Dim_best);
199                                 if (this.HamPath[this.PathIndex +
                                    1].Upper[Dim_best] <
                                        msg.Upper[Dim_best])// check
                                            intersecton with a second node
201                                 {
                                        if (this.PathIndex + 2 <
                                            this.HamPath.Count)
203                                     //check exsiting of the second node
                                        {
```

```
205                                         // makin a new query msg
                                            msg.Lower[msg.Dim_best] =
207                                             msg.Issuer.HamPath[this.PathIndex].
                                                Upper[msg.Dim_best];
209
                                            msg.Upper[msg.Dim_best] =
211                                             msg.Issuer.HamPath[this.PathIndex
                                                    +
                                                    1].Upper[msg.Dim_best];

213                                         this.FissionERouting(this.KautzString,
                                                this.HamPath[this.PathIndex
                                                    + 2].KautzString, msg,
                                                    Dim_best);
215                                     }
                                    }
217                             }
                            }
219                         else
                            {
221                             //report
                                if (this.Upper[Dim_best] <
                                    msg.Upper[Dim_best])
223                             {
                                    msg.Lower[Dim_best] =
                                        this.Upper[Dim_best];
225
                                    this.FissionERouting(this.KautzString,
227                                     this.HamPath[this.PathIndex +
                                            1].KautzString, msg, Dim_best);
                                }
229                         }
                        }
231                     }

233
                }
235         }
            else
237         {
                for(int j=0; j<this.outNeighbors.Count;j++)
239             {
                    String X = this.outNeighbors[j].
241                     Substring(this.KautzString.Length -1,
                    this.outNeighbors[j].Length-this.KautzString.Length
                        +1);
243
                    if (dst.StartsWith(SP + X))
245                 {
                        SP = SP + X;
247                     for (int k = 0; k < this.HamPath.Count; k++)
                        {
249                         if (this.HamPath[k].KautzString ==
                                this.outNeighbors[j])
```

```
                                {
251
                                    msg.HopCount++;
253                                 this.HamPath[k].Routing(dst,
                                        PathLength - 1, SP, msg,
                                        Dim_best);
                                    break;
255                             }
                            }
257                         break;
                        }
259
                    }
261
                }
263         }
```

Appendix C

Query Generation Code

This code corresponds to the GenerateRangeQueries call in line 4 of Algorithm 9 in chapter 4.

```
1  private List<QueryMessage>
       GenerateUniformRandomPointQueries(KautzGraph KG, int numofdim)
           {
3
               List<QueryMessage> result = new List<QueryMessage>();
5              int nodeCount = KG.Nodes.Count();

7              for (int i = 0; i < nodeCount; i++)
               {
9                  for (int j = 0; j < nodeCount; j++)
                   {
11                     QueryMessage Q = new QueryMessage(KG.Nodes[i]);
                       Q.Issuer = KG.Nodes[i];
13                     Random rnd = new Random();
                       for (int k = 0; k < numofdim; k++) {
15                         float tmp = rnd.Next(1);
                           Q.Lower.Add(tmp);
17                         Q.Upper.Add(tmp);
                       }
19                         Q.ID = i * nodeCount + j;
                           result.Add(Q);
21

23                 }
               }
25
               return result;
27         }

29         private List<QueryMessage>
               GenerateUniformRandomRangeQueries_ConstantV(KautzGraph KG,
               int queryCount, int numofDim)
           {
31
               List<QueryMessage> result = new List<QueryMessage>();
33             int nodeCount = KG.Nodes.Count();
```

```
35              Random R = new Random();
                int k = 0;
37              while (k < queryCount)
                {
39                  int i = R.Next(0, nodeCount);

41                  QueryMessage Q = new QueryMessage(KG.Nodes[i]);
                    Q.Issuer = KG.Nodes[i];
43
                    double min = 1000;
45                  float leftSide = 0;
                    float multiplicaton = 1;
47                  for (int d = 0; d < numofDim - 1; d++)
                    {
49                      leftSide = (float)R.Next(0, 1000) / 1000;
                        float size = (float)R.Next(0, 1000) / 1000;
51                      Q.Lower.Add(leftSide);
                        Q.Upper.Add(Math.Min(leftSide + (float)size, 1));
53                      multiplicaton *= size;
                        min = Math.Min(min, Q.Upper[d] - Q.Lower[d]);
55                      if (min == (Q.Upper[d] - Q.Lower[d]))
                            Q.Dim_best = d;
57                  }
                    leftSide = (float)R.Next(0, 1000) / 1000;
59                  Q.Lower.Add(leftSide);
                    float upper = ((float)Math.Pow(0.05, numofDim)) /
                        multiplicaton;
61                  if (leftSide + (float)upper < 1)
                    {
63
                        Q.Upper.Add(Math.Min(leftSide + (float)upper, 1));
65                      min = Math.Min(min, Q.Upper[numofDim - 1] -
                            Q.Lower[numofDim - 1]);
                        if (min == (Q.Upper[numofDim - 1] -
                            Q.Lower[numofDim - 1]))
67                          Q.Dim_best = numofDim - 1;
                        Q.ID = k;
69
                        int firstnodeinrangeIndex =
                            (int)(Math.Ceiling(Q.Lower[Q.Dim_best] *
                            (Q.Issuer.HamPath.Count))) - 1;
71                      int lastnodeinrangeIndex =
                            (int)(Math.Ceiling(Q.Upper[Q.Dim_best] *
                            (Q.Issuer.HamPath.Count))) - 1;
                        Q.Nodesinrange = lastnodeinrangeIndex -
                            firstnodeinrangeIndex + 1;
73
                        StreamWriter SW1 = new StreamWriter(path +
                            Q.ID.ToString() + "info.txt", false);
75                      SW1.WriteLine(Q.ID.ToString() + "\t" +
                            Q.Dim_best.ToString() + "\t" +
77                          Q.Lower[0].ToString() + "\t" +
                            Q.Lower[1].ToString() + "\t" +
```

```
79                          Q.Upper[0].ToString() + "\t" +
                            Q.Upper[1].ToString() + "\t" +
81                          Q.Issuer.KautzString);
                        SW1.Close();
83
                        result.Add(Q);
85                      k++;
                    }
87              }

89
                return result;
91          }

93          private List<QueryMessage>
                GenerateUniformRandomRangeQueries(KautzGraph KG, int
                queryCount, int numofDim)
            {
95
                List<QueryMessage> result = new List<QueryMessage>();
97              int nodeCount = KG.Nodes.Count();

99              Random R = new Random();
                int k = 0;
101             while (k < queryCount)
                {
103                 int i = R.Next(0, nodeCount);

105                 QueryMessage Q = new QueryMessage(KG.Nodes[i]);
                    Q.Issuer = KG.Nodes[i];
107
                    double min = 1000;
109                 float leftSide = 0;
                    for (int d = 0; d < numofDim ; d++)
111                 {
                        leftSide = (float)R.Next(0, 600) / 1000;
113
                    // float size = (float)R.Next(0, 100) / 1000;
115                     float size =(float)0.4;
                        Q.Lower.Add(leftSide);
117                     Q.Upper.Add(Math.Min(leftSide + (float)size, 1));
                        if ((Q.Upper[d] - Q.Lower[d]) < min)
119                     {
                            min =  Q.Upper[d] - Q.Lower[d];
121                             Q.Dim_best = d;
                        }
123                 }

125
                    Q.ID = k;
127
                    int firstnodeinrangeIndex =
                        (int)(Math.Ceiling(Q.Lower[Q.Dim_best] *
                        (Q.Issuer.HamPath.Count))) - 1;
```

```
129                     int lastnodeinrangeIndex =
                            (int)(Math.Ceiling(Q.Upper[Q.Dim_best] *
                            (Q.Issuer.HamPath.Count))) - 1;
                        Q.Nodesinrange = lastnodeinrangeIndex -
                            firstnodeinrangeIndex + 1;
131
                        StreamWriter SW1 = new StreamWriter(path +
                            Q.ID.ToString() + "info.txt", false);
133                     SW1.WriteLine(Q.ID.ToString() + "\t" +
                            Q.Dim_best.ToString() + "\t" +
135                         Q.Lower[0].ToString() + "\t" +
                            Q.Lower[1].ToString() + "\t" +
137                         Q.Upper[0].ToString() + "\t" +
                            Q.Upper[1].ToString() + "\t" +
139                         Q.Issuer.KautzString);
                        SW1.Close();
141
                        result.Add(Q);
143                     k++;

145             }

147
                return result;
149         }
```

Vita

Candidate's full name: Zahra Mirikharaji
University attended (with dates and degrees obtained):

- 2008-2013 Isfahan University of Technology (Isfahan, Iran) Bachelor of Science in Electrical Engineering
- 2013-2015 University of New Brunswick (Fredericton, Canada) Master of Computer Science

Publications:

- Zahra Mirikharaji and Bradford G. Nickerson, "A Fault Tolerant Data Structure for Peer-to-Peer Range Query Processing", 27th Canadian Conference on Computational Geometry (CCCG 2015).
- Pouya Bisadi, Zahra Mirikharaji and Bradford G. Nickerson, "A Fault Tolerant Constant Degree Distributed Spatial Data Structure", 2015, submitted to *Peer-to-Peer Networking and Applications* journal paper.

Conference Presentations:

- "Distributed Spatial Data Structures for Big Data", Poster at UNB Research Expo Conference 2014.
- "Range Query Processing in Peer-to-Peer Networks", Poster at UNB Research Expo Conference 2015.

Other Documents:

- Zahra Mirikharaji and Bradford G. Nickerson, Distributed Spatial Data Structures, Technical Report TR14-231, Faculty of Computer Science, University of New Brunswick, Fredericton, August 2014.