# Distributed High-Dimensional Similarity Search with Music Information Retrieval Applications

by

Aidin Faghfouri
B.Eng., Sadjad Institute of Higher Education, 2009

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science

in the Department of Computer Science

© Aidin Faghfouri, 2011
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

**Distributed High-Dimensional Similarity Search**

**with Music Information Retrieval Applications**

by

Aidin Faghfouri

B.Eng., Sadjad Institute of Higher Education, 2009

Supervisory Committee

---

Dr. Jianping Pan, Supervisor

(Department of Computer Science)

---

Dr. Kui Wu, Departmental Member

(Department of Computer Science)

**Supervisory Committee**

Dr. Jianping Pan, Supervisor

(Department of Computer Science)

Dr. Kui Wu, Departmental Member

(Department of Computer Science)

## ABSTRACT

Today, the advent of networking technologies and computer hardware have enabled more and more inexpensive PCs, various mobile devices, smart phones, PDAs, sensors and cameras to be linked to the Internet with better connectivity. In recent years, we have witnessed the emergence of several instances of distributed applications, providing infrastructures for social interactions over large-scale wide-area networks and facilitating the ways users share and publish data. User generated data today range from simple text files to (semi-) structured documents and multimedia content. With the emergence of Semantic Web, the number of features (associated with a content) that are used in order to index those large amounts of heterogenous pieces of data is growing dramatically. The feature sets associated with each content type can grow continuously as we discover new ways of describing a content in formulated terms.

As the number of dimensions in the feature data grow (as high as 100 to 1000), it becomes harder and harder to search for information in a dataset due to the curse of dimensionality and it is not appropriate to use naive search methods, as their performance

degrade to linear search. As an alternative, we can distribute the content and the query processing load to a set of peers in a distributed Peer-to-Peer (P2P) network and incorporate high-dimensional distributed search techniques to attack the problem.

Currently, a large percentage of Internet traffic consists of video and music files shared and exchanged over P2P networks. In most present services, searching for music is performed through keyword search and naive string-matching algorithms using collaborative filtering techniques which mostly use tag based approaches. In music information retrieval (MIR) systems, the main goal is to make recommendations similar to the music that the user listens to. In these systems, techniques based on acoustic feature extraction can be employed to achieve content-based music similarity search (i.e., searching through music based on what can be heard from the music track). Using these techniques we can devise an automated measure of similarity that can replace the need for human experts (or users) who assign descriptive genre tags and meta-data to each recording and solve the famous cold-start problem associated with the collaborative filtering techniques.

In this work we explore the advantages of distributed structures by efficiently distributing the content features and query processing load on the peers in a P2P network. Using a family of Locality Sensitive Hash (LSH) functions based on $p$-stable distributions we propose an efficient, scalable and load-balanced system, capable of performing K-Nearest-Neighbor (KNN) and Range queries. We also propose a new load-balanced indexing algorithm and evaluate it using our Java based simulator.

Our results show that this P2P design ensures load-balancing and guarantees logarithmic number of hops for query processing. Our system is extensible to be used with all types of multi-dimensional feature data and it can also be employed as the main indexing scheme of a multipurpose recommendation system.

# Contents

**Supervisory Committee** ii

**Abstract** iii

**Table of Contents** v

**List of Tables** vii

**List of Figures** viii

**List of Symbols** x

**Acknowledgements** xi

**Dedication** xii

**1 Introduction** 1

**2 Background and related work** 5

2.1 Music information retrieval . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.2 Marsyas framework . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.3 High-dimensional similarity search . . . . . . . . . . . . . . . . . . . . . . 9

2.3.1 Distributed similarity search in high-dimensional spaces . . . . . . 11

2.4 Locality Sensitive Hashing techniques . . . . . . . . . . . . . . . . . . . . 13

**3 System Architecture** 17

3.1 DHT-based overlay network . . . 18

3.1.1 P2P Simulator . . . 20

3.2 Data indexing based on Locality Sensitive Hashing . . . 21

3.2.1 Mapping to the peer identifier space . . . 21

3.2.2 Local DHT creation . . . 26

3.2.3 Local DHT entry points . . . 28

3.3 Load balancing . . . 30

3.3.1 Load blancing with bucket width prediction . . . 30

3.4 Queries . . . 35

3.4.1 KNN Query processing . . . 36

3.4.2 Range Query processing . . . 40

**4 Experimental results** 44

4.1 Experimental Setup . . . 44

4.2 Experimental Results . . . 48

4.2.1 KNN query results . . . 48

4.2.2 Range query results . . . 53

4.2.3 Load balancing results . . . 58

**5 Conclusions and future work** 60

5.1 Further research issues . . . 62

**Bibliography** 64

**A LSH Parameters** 70

**Glossary** 72

# List of Tables

Table 4.1 Simulation setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48

Table 4.2 Gini coefficient for synthetic dataset when distributing 2 replicas of the data points . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59

# List of Figures

Figure 3.1 An identifier circle and three existing nodes 0, 1, and 3 and their finger tables (picture taken from [38]) . . . . . . . . . . . . . . . . . . 19
Figure 3.2 Mapping from the d-dimensional feature space to the peer identifier space . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
Figure 3.3 Mapping from the d-dimensional feature space to the local DHTs . . 23
Figure 3.4 10 Local DHTs on a Chord network with 1100 peers . . . . . . . . . 28
Figure 3.5 Skewed load on peers in a local DHT . . . . . . . . . . . . . . . . . 31
Figure 3.6 $\psi_{enhanced}$ and bucket width prediction . . . . . . . . . . . . . . . . . 35
Figure 3.7 Linear Range query processing and the associated empty ranges problem . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
Figure 3.8 Linear Range query processing . . . . . . . . . . . . . . . . . . . . 43
Figure 3.9 Sampling-based Range query processing . . . . . . . . . . . . . . . 43
Figure 4.1 Recall vs. the number of network hops for different placement methods, employing the linear KNN query processing for the standard dataset when distributing one replica of the dataset. . . . . . . . . . . 49
Figure 4.2 Recall vs. the number of network hops for different placement methods, employing the linear KNN query processing for the standard dataset when distributing five replicas of the dataset. . . . . . . . . . 50

Figure 4.3 Recall vs. the number of network hops for different placement methods, employing the linear KNN query processing for the standard dataset when distributing 10 replicas of the dataset. . . . . . . . . . . 51

Figure 4.4 Recall vs. the number of network hops for different placement methods, employing the linear KNN query processing for the mixed dataset when distributing 10 replicas of the dataset. . . . . . . . . . . . . . . 52

Figure 4.5 Recall vs. the number of network hops for different placement methods, employing the linear KNN query processing for the synthetic dataset when distributing 10 replicas of the dataset. . . . . . . . . . . 52

Figure 4.6 The effect of varying the range on recall. The results are shown for linear and sampling-based query processing methods using the standard dataset with 10 replicas and $\psi_{enhanced}$ placement method. . . 55

Figure 4.7 The effect of varying the range on recall. The results are shown for linear and sampling-based query processing methods using the standard dataset with 10 replicas and $\psi_{old}$ placement method. . . . . . 56

Figure 4.8 The effect of varying the range on recall. The results are shown for linear and sampling-based query processing methods using the mixed dataset with 10 replicas and $\psi_{enhanced}$ placement method. . . . 57

Figure 4.9 The effect of varying the range on recall. The results are shown for linear and sampling-based query processing methods using the mixed dataset with 10 replicas and $\psi_{old}$ placement method. . . . . . . 57

Figure 4.10 The effect of varying the number of peers inside each local DHT on load balancing factor (Gini coefficient) for $\psi_{enhanced}$ and $\psi_{old}$ placement methods. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59

# List of Symbols


$d$ the number of dimensions of a feature vector ........ 16

$h$ a locality sensitive hash function ........ 16

$g()$ a hash function wrapper for k hash functions ........ 16

$G$ the global network ........ 27

$k$ the number of hash functions ........ 16

$K$ K in the K-Nearest-Neighbor queries ........ 9

$l$ the number of hash tables or local DHTs in the global network ........ 27

$M$ the number buckets in a hash table ........ 27

$N$ the number of peers in global network ........ 18

$P$ a peer in the peer-to-peer network ........ 18

$q$ the query point ........ 9

$r$ the range for Range queries ........ 9

$v$ a feature vector ........ 16

$\alpha$ the relaxing parameter for KNN queries ........ 39

$\gamma$ the number of gateway peers in a local DHT ........ 28

$\tau$ the distance of the K-th item (regarding a query point) in a peer storage 37

$\xi_{sum}$ sum function that maps a vector of integers to an integer number ........ 22

$\psi$ mapping function that maps an integer value to a bucket ........ 27

$\rho$ mapping function that maps a k-dimensional vector to a bucket ........ 27

# ACKNOWLEDGEMENTS

I would like to thank all people who have helped and inspired me during my Master's program.

I would like to thank my supervisor, Dr. Jianping Pan, whose encouragement, guidance and support from the initial to the final level enabled me to develop an understanding of the subject.

I want to express my gratitude to my thesis committee member, Dr. Kui Wu for his valuable suggestions.

It is a great pleasure to thank Dr. Tzanetakis, Parisa Haghani and Steven Ness who helped me during my research.

I would also like to thank my External Examiner Dr. Driessen for agreeing to take the time for assessing this work.

Last but not least, I wish to thank my family. Without their support, I would never be where I am today.

*I believe that music is a force in itself. It is there and it needs an outlet, a medium. In a way, we are just the medium.*

Maynard James Keenan

# DEDICATION

To Chi and the humanity

# Chapter 1

# Introduction

Today the Internet has become an integral part of our society. People are becoming increasingly dependent on Internet applications such as the World Wide Web, Email and Facebook to have their daily life, business and entertainment. Continued advances in networking technologies and computer hardware have enabled more and more inexpensive PCs, various mobile devices, smart phones, PDAs, sensors, and cameras to be linked to the Internet with better connectivity. In recent years, we have witnessed the emergence of several instances of distributed applications that are pushing the limits of content sharing, massive computations and social interactions over large-scale wide-area networks, providing an infrastructure for the user generated data to be published and shared on the Internet.

User generated data today range from simple text files to (semi-) structured documents and multimedia content. With the emergence of Semantic Web, the number of features (associated with a content) that are used to index those large amounts of heterogenous pieces of data is growing dramatically. The feature sets associated with each content type can grow continuously as we discover new ways of describing a content in formulated terms. As instances of such features we can mention the color, texture, shape and spacial location for image data, acoustic features for audio data and temporal features for video

data. These features can be described as high-dimensional feature vectors representing an item, and they can be used for item similarity search, item classification and clustering.

As the number of dimensions in the feature data grows (as high as 100 to 1000), it becomes harder and harder to search for information in a dataset due to the curse of dimensionality [9] and it is not possible to use naive search methods as their performance degrades to linear search. Nowadays, in real-world applications, these high-dimensional feature data are distributed in large-scale networks and traditional centralized indexing techniques become impractical. Distributed similarity search in high-dimensional vector spaces has been the subject of substantial research, motivated in part by the need to provide query support for image, audio, video and other complex data types [9, 19, 4, 29, 3, 22].

We know that a large percentage of Internet traffic consists of music files shared and exchanged over peer-to-peer (P2P) networks. In most present services, search for music is performed through specifying keywords and naive string-matching algorithms as well as collaborative filtering techniques which mostly use tag based approaches. In such systems, the main goal is to recommend a music that sounds similar to the ones that the user listens to. During the past years, the emerging field of music information retrieval (MIR) has produced a variety of new ways of looking at the problem. Thus, as an alterative approach, techniques based on acoustic features extracted from audio data can be employed to achieve content-based music similarity search [8]. Using these techniques we can devise an automated measure of similarity that can replace the need for human experts (or users) who assign descriptive genre tags and meta-data to each recording and solve the famous cold-start problem associated with the collaborative filtering techniques [2].

In this work we explore the advantages of distributed structures by efficiently distributing the content features and query processing load to the peers in a P2P network. This is done by means of an indexing scheme that uses a family of Locality Sensitive Hash (LSH) functions based on $p$-stable distributions. We also use the Chord [38] distributed hash table

(DHT) as an underlying P2P structure which enables us to have a small number of peers maintaining the indexed data while having a logarithmic number of routing hops for the lookups. Due to the fact that in music information retrieval systems it is important for us to find a specific number of music tracks that are closest to a music track of our interest, we consider two important types of queries namely K-Nearest-Neighbor (KNN) and Range queries. In order to ensure high precision searches, we create multiple local DHTs (dynamically growing and shrinking) within the global Chord network. We also use gateway peers placed at the hot spots to act as entry points to local DHTs. As a result, this system guarantees an upper bound for query processing in terms of the number of hops, logarithmically smaller than the global number of peers. The details about the system can be found in Chapter 3.

The contributions of this thesis have three main aspects. First, we extend the Chord module of the Java based simulator PeerSim. Our extended Chord module implements the approach proposed by [20, 19] and also implements multiple search algorithms on top of the LSH indexing scheme (Section 3.1.1). Second, we study the performance of the LSH technique using three different musical datasets. Our results are based on a set of experiments using the extended simulator and three different datasets (two real datasets and one synthetic dataset). In our simulation we consider KNN and Range search (Section 4.1). As the third and the main contribution of this work, we propose a new mapping function incorporated in the indexing scheme (Section 3.3) which improves the load balancing of the system and we evaluate the system performance (Chapter 4) using both the newly proposed mapping technique and a mapping technique used in [20, 19].

The results show that having such a design, we can create an efficient, scalable and load-balanced P2P network, capable of searching for high-dimensional data. This system can also be employed in multipurpose recommendation systems as the main item indexing module, providing efficient content-based similarity search and access to multimedia

contents.

The rest of the thesis is organized as follows. In Chapter 2, we review the existing research on music information retrieval and high-dimensional search and provide a backgrounder on the LSH technique. In Chapter 3, we describe our system architecture and present our load balancing technique in details. In Chapter 4, we present our experimental results and discuss our results regarding different query types on different datasets. In Chapter 4 we also compare our load balancing scheme to the technique used in [20, 19] and show how it improves the load balancing on the peers in the network. In Chapter 5, we conclude our work followed by the discussion on the ways of further improving the load balancing and system performance and how this system can be incorporated as a building block in multipurpose recommendation systems.

# Chapter 2

# Background and related work

The motivation of this work is to create an efficient and distributed high-dimensional similarity search system, with music information retrieval as a main application. Thus, we first provide background on music similarity and music information retrieval in Section 2.1 and Section 2.2. In Section 2.3 respectively we discuss similarity search in high-dimensional spaces, explaining two types of queries namely K-Nearest-Neighbor (KNN) and Range queries that are useful in a music information retrieval system, and we also discuss the two categories of existing approaches. A review of the related work on distributed similarity search systems with a special focus on music information retrieval is also available in Section 2.3. At the end, Section 2.4 classifies hash functions and provides a conceptual and mathematical background on the Locality Sensitive Hashing (LSH) technique as the fundamental approach used throughout this work.

## 2.1 Music information retrieval

In the field of Music Information Retrieval (MIR) one goal is to devise an automatic measure of the similarity between two musical recordings based only on the analysis of their audio content [8]. This measure can be used as a tool to build classification, retrieval,

browsing, and recommendation systems. However, to develop such a measure we require ground truth. This ground truth is a hidden element in a single underlying similarity that forms the desired output of the measure. The concept of similarity has been studied many times in various fields such as psychology, information retrieval, and epistemology [8].

Music similarity is an abstruse concept because of its subjectiveness. The fact that people have individual tastes and preferences makes developing a reliable ground truth a hard goal to achieve. In fact, individuals taste may evolve over time and it even depends on factors such as individual's mood. The question that "how similar two artists or songs are?" can be answered from various perspectives. Music may be similar or distinct in terms of genre, melody, rhythm, tempo, geographical origin, instrumentation, lyric content and historical time-frame. Besides that, subjective similarity often violates the definition of a metric, in particular the properties of symmetry and the triangle inequality [8].

Despite all of these difficulties, techniques to automatically determine music similarity have attracted much attention in recent years [12, 40, 25, 15]. Genre hierarchies, typically created manually by human experts, are currently one of the ways used to structure music content on the Web [40]. As we know, the number of new songs that are being produced everyday is quickly growing, and having a system based on expert opinion (i.e., services using a group of music experts that actually listen to the music and specify its genre) contradicts with the required scalability of such systems. Automatic musical content analysis can potentially automate this process and provide an important component for a complete music information retrieval system for audio signals.

Similarity lies at the core of the classification and ranking algorithms needed to organize and recommend music. In order to have an automated measure of similarity, we should first transform the raw audio into a feature space; i.e., a numerical representation in which dimensions measure different properties of the audio recording. A good feature space compactly represents the audio, extracting important information and omitting irrelevant

noise.

Many features have been proposed for music analysis, such as MFCC (Mel-Frequency Cepstral Coefficients), spectral centroid, bandwidth, loudness, and sharpness [28]. MFCC captures the overall spectral shape, which holds important information such as timbral and instrumental characteristics, the quality of a singers voice, and production effects in an audio track. On the other hand, being a purely local feature calculated over a window of tens of milliseconds, in contrast to features such as pitch class (or Chroma value), it does not capture information about melody, rhythm, or long-term song structure. MFCC features originally developed for speech-recognition systems and have shown to give good performance for a variety of audio classification tasks [10]. These features are favored by a number of groups working on audio similarity [40, 10, 17, 25].

MFCC features can also be mapped to an anchor space as an alternate approach to the problem of music similarity search. The anchor space technique is inspired by a folk-wisdom approach to music similarity. This approach is based on how people describe artists by statements such as "Radiohead sounds like Pink Floyd meets BBKing, but more electronic." Here, meaningful musical categories and well-known anchor artists serve as convenient reference points for describing the music [8]. This idea builds up the anchor space technique, wherein classifiers are trained to recognize musically meaningful categories, and music is subsequently "described" in terms of these categories. Once the classifiers are trained, the audio is fed to each classifier, and the outputs, representing the activation or likelihood of the categories, locate the music in the new anchor space. As a drawback for these systems we can mention the cold start problem. Cold start problem refers to the start-up condition in which there is not enough initial training data provided to the classifier, and thus leads to inaccurate or noisy results. Further details about the choice of anchors and classifier training techniques are available in [7].

In all types of information retrieval systems, similarity needs to be calculated for two

items (i.e., audio recordings in our case) at some point. This similarity is usually the inverse of the mathematical distance between the feature sets extracted from the two items. The results for the aforementioned approaches using different distance measures such as Centroid Distance, Earth Movers Distance, and the Asymptotic Likelihood Approximation are discussed in [8]. In this work we assume that the distance measure is the widely used $l_2$ norm (Euclidean distance or Centroid distance) which is proven to give promising results, yet being easy to calculate. It is also worth mentioning that only considering the feature space and the Euclidean distance between the feature sets of different songs, we can achieve reasonable results in terms of item similarity [13].

## 2.2 Marsyas framework

Marsyas is an open source software framework for music analysis, retrieval and synthesis with specific emphasis on Music Information Retrieval applications. These include Audio Classical Composer Identification, Audio Genre Classification (Latin and Mixed), Audio Music Mood Classification, Audio Beat Tracking, Audio Onset Detection, Audio Music Similarity and Retrieval and Audio Tagging Tasks. This framework has been in development for more than 10 years and has been used for a variety of projects in both academia and industry in several countries. Based on a novel dataflow architecture named implicit patching it provides a variety of existing processing modules for digital signal processing, machine learning and audio input/output that can be combined at run-time to form complex dataflow networks expressing audio processing algorithms (black-box functionality).

Marsyas is designed with inter-operability in mind and provides various mechanisms for communicating with other software. This framework supports bindings to the run-time functionality in scripting languages (Python, Ruby), the run-time data interchange with MATLAB, the support for the Music Instrument Digital Interface (MIDI) protocol and

Open Sound Control (OSC) for communicating with controller devices, and the infrastructure for easy interfacing to the GUI components of the Qt toolkit [41].

In this work we use the *bextract* command of Marsyas(version 0.4.1) in order to extract features of audio recordings and create two datasets for our experimental results. The *bextract* command extracts the means and variances of timbral features (time-domain Zero-Crossings, Spectral Centroid, Rolloff, Flux and Mel-Frequency Cepstral Coefficients (MFCC)). The result is a 124-dimension vector, with each dimension representing an average of the extracted feature over multiple time windows.

## 2.3 High-dimensional similarity search

Similarity search in high dimensional spaces has been the focus of many works in the database community as well as related communities such as network and information retrieval during the recent years. The objective of this field is to find all items that are similar to a given query item, such as a music track, a digital image, a video, a text document or a DNA sequence. Usually items are represented in a high dimensional feature space and a distance function (d), usually an $l_2$ norm (Euclidean distance), defines the similarity of two objects.

Two types of queries are important to our interest in such systems: K-Nearest-Neighbor (KNN) query and Range query.

- *K-Nearest Neighbor (KNN) query:* Given a query point $q$ the goal is to find the $K$ closest (in terms of the distance function) points to it.
- *Range query:* Given a query point $q$ and a range $r$ the goal is to find all points within a distance r of q.

More formally we can define these queries types according to [29]:

**Definition 1.** *Given an object* $q \in D$ *and a number* $k \in N$, ***K-Nearest-Neighbor*** *query KNN(q,K) retrieves a subset of objects* $S_A \subseteq X : |S_A| = K, \forall o \in S_A, \forall o' \in X \backslash S_A : d(q, o) \leq d(q, o')$.

**Definition 2.** *Given an object* $q \in D$ *and a maximum search raduis* $r$, ***Range*** *query R(q,r) retrieves a subset of indexed objects* $S_A \in X$ *such that* $S_A = \{o | o \in X^d(q, o) \leq r\}$.

Past research on the problem with a focus on the centralized setting suggests that the approaches which address these query types in high-dimensional spaces, can be divided into two main categories.

The first category includes the *Space partitioning* methods which incorporate all tree-base approaches such as the R-tree [11] and K-D trees [5], which perform very well when data dimensionality is not high. On the other hand, the performance of these approaches degrades to linear search for high enough dimensions [9] due to the curse of dimensionality. The curse of dimensionality states that the data structures scale poorly with data dimensionality; e.g., if the number of dimensions exceeds 10 to 20, searching in K-D trees and related structures involves the inspection of a large fraction of the database; therefore, its performance would be close to a brute-force linear search. The Pyramid [6] and iDistance [42] techniques are based on mapping the high dimensional data to one dimension and then partition/cluster that space to answer queries by translating them to the one dimensional space.

Hash-based approaches form the second category which trades accuracy for efficiency. This behavior is due to the incorporation of approximation techniques to make the unavoidable sequential scan as fast as possible. As a result of this probabilistic behavior, these approaches return approximate closest neighbors of a query point. LSH [18] is an approximate method, which uses several locality preserving hash functions in order to hash the points in a database such that with high probability close points are hashed to the same bucket. While this method is very efficient in terms of time, tuning such hash functions

depends on the distance of the query point to its closest neighbor.

Several follow-ups of this LSH technique exist which try to solve the problems associated with LSH. [26, 31] suggest an intelligent probing of the LSH buckets that are likely to contain query results in a hash table, to reduce the number of required hash tables. LSH Forest [4] proposes a system which tries to address the problem of data-dependent LSH parameters which must be hand-tuned. LSH Forest [4] also improves performance guarantees for skewed data distributions while retaining the same storage and query overhead. Furthermore, [13] uses $p$-stable hash functions in order to reduce the search time and calculates an upper-bound for its scheme which strongly outperforms the structure of the K-D tree.

### 2.3.1 Distributed similarity search in high-dimensional spaces

The emergence of the P2P paradigm [27, 34, 38], has led to the improvement of computational power by distributing the computation load to a set of nodes and machines working in a network. Internet developers are constantly proposing new and visionary distributed applications. These new applications have a variety of requirements for scalability and performance. Services such as multimedia content sharing are one of the main interests in today's world which requires *similarity search* as a basic building block for its functionalities. A number of P2P approaches, such as [8, 6, 7] have been proposed for similarity search over distributed networks, but they either consider one dimensional data or data with a small number of dimensions.

Some approaches such as MCAN [16] and M-Chord [30] can perform similarity search in the metric space. They both use a pivot-based technique to map the high dimensional metric data to an N-dimensional vector space, and then respectively use CAN [16] and Chord [38] as their underlying structured P2P system. As a drawback of such systems we can mention the centralized data preprocessing phase before distributing the data on peers,

which is required in order to choose the pivots.

SWAM [3] is a family of Small World Access Methods, which aims for efficient execution of various similarity-search queries, such as Exact-Match, Range, and K-Nearest-Neighbor (KNN) queries. SWAM [3] builds a network structure that groups together peers with similar content. The problem of this structure is that each peer can hold a single data item, which is not well-suited for large data sets and real-world applications.

SkipIndex [43] and VBI-tree [22] both rely on tree-based approaches which do not scale well when data dimensions are high. Recently, SimPeer [14] was proposed, which uses the idea of iDistance [42] to provide range query capabilities in a hierarchical unstructured P2P network for high dimensional data. In that work the peers are assumed to hold and maintain their own data which is contradictory to the requirements of multimedia content sharing systems.

pSearch [39], uses the two well known information retrieval techniques Vector Space Model (VSM) and Latent Semantic Indexing (LSI) to create a semantic space. This Cartesian space is then directly mapped to a multi-dimensional CAN [16] ID space which basically has the same dimensionality of the Cartesian space (300 dimensions at most). Different overlays are needed for various data sets with different dimensionality due to the fact that the dimensionality of the underlying peer-to-peer network depends on the dimensionality of the data (or the number of reduced dimensions). Again, this dependency and centralized computation of LSI make this approach less practical in real applications.

In [36] the authors follow pSearch by employing VSM and LSI. Their approach differs from pSearch in mapping the resultant high dimensional Cartesian space to a one-dimensional Chord. Unlike pSearch this method is independent of data size and dimensionality. This is the closest work in the state-of-the-art to [19], since it considers high dimensional data over a structured peer-to-peer system. The results of both systems [36] and [19] are compared with each other in [19]. [19] considers efficient similarity search

over structured P2P networks, which guarantees a logarithmic lookup time in terms of the network size, and leverages on LSH-based approaches to provide approximate results to KNN search efficiently, even with very high dimensional data. This approach also enables efficient Range query which is very difficult and has its own pitfalls in LSH-based approaches [19]. LSH is discussed in more details in Section 2.4.

The technique proposed in [19] is adopted by our work to create a distributed high-dimensional music similarity search system. The authors in [19] show that this method is applicable to image data, which means that the technique is suitable for similarity search in Euclidean space. One of the main goals of our research is to investigate the applicability of this technique and the usage of Euclidean distance in music similarity search.

## 2.4 Locality Sensitive Hashing techniques

A similarity search problem involves a collection of objects (documents, images, etc.) that are characterized by a collection of relevant features and represented as points in a high-dimensional feature space; given queries in the form of points in this space, we are required to find the nearest (most similar) objects to the query.

A particularly interesting and well-studied instance is Euclidean space. This problem is of major importance to a variety of applications; some examples are: information retrieval, databases and data mining, data compression, image and video databases, machine learning, pattern recognition, statistics and data analysis. Typically, the features of the objects of interest (documents, images, etc) are represented as points in $\Re^d$. A distance metric is used to measure the similarity of objects in this space. The basic problem then is to perform indexing or similarity search for query objects. The number of features (i.e., the dimensionality) ranges anywhere from tens to thousands [13].

In [21, 18], the authors introduced an approximate high-dimensional similarity search

scheme with a provably sublinear dependence on the data size. Instead of using tree-like space partitioning, it relied on a new method called Locality Sensitive Hashing (LSH). The key idea is to hash the points using several hash functions so as to ensure that, for each function, the probability of collision is much higher for objects which are close to each other than for those which are far apart. Then, one can determine nearest neighbors by hashing the query point and retrieving elements stored in the buckets containing that point. In [19, 14] the authors provided such locality-sensitive hash functions for the case when the points are in binary Hamming spaces $\{0,1\}^d$. They experimentally showed that the LSH data structure achieves a large speedup over several tree-based data structures when the data is stored on disk. In addition, since the LSH is a hashing-based scheme, it can be naturally extended to the dynamic setting, i.e., when insertion and deletion operations also need to be supported. This approach avoids the complexity of dealing with tree structures when the data is dynamic. This technique is used in [36].

However, the naive LSH approach suffers from a fundamental drawback: it is fast and simple only when the input points live in the Hamming space. As mentioned in [21, 18], it is possible to extend the algorithm to the $l_2$ norm, by embedding $l_2$ space into Hamming space. However, it increases the query time and/or error by a large factor and complicates the algorithm.

In this work we use a newer version of the LSH algorithm based on $p$-stable hash functions, introduced in [13]. As with previous schemes, it works for the K-Nearest-Neighbor (KNN) and can be extended to a P2P structure in order to support Range queries as well [19]. Unlike the previously mentioned approaches, $p$-stable LSH works directly on points in Euclidean space without any embeddings.

In order to understand this technique, we first explain Locality Sensitive Hash functions and then the LSH functions based on stable distributions. Afterwards, we discuss the $p$-stable LSH scheme parameters and provide formulas to tune them according to the dataset.

### Locality Sensitive Hash functions

A family of hash functions $H = h : S \rightarrow U$ is called $(r_1, r_2, p_1, p_2)$-sensitive if the following conditions are satisfied for any two points $q_1, q_2 \in S$:

- if $dist(q_1, q_2) \leq r_1$ then $\Pr_H(h(q_1) = h(q_2)) \geq p_1$
- if $dist(q_1, q_2) > r_2$ then $\Pr_H(h(q_1) = h(q_2)) \leq p_2$

where $S$ specifies the domain of points, $dist$ is the distance measure defined in this domain and $Pr$ is the probability function.

If $r_1 < r_2$ and $p_1 > p_2$, the intrinsic property of these functions results in more similar objects being mapped to the same hash value than distant ones [13].

### $p$-stable distributions

Stable distributions are defined as limits of normalized sums of independent identically distributed variables. The most well-known example of a stable distribution is *Gaussian (or normal)* distribution. However, the class is much wider; for example it includes heavy-tailed distributions.

**Definition 3.** *A distribution $D$ over $\Re$ is called $p$-stable, if there exists $p \geq 0$ such that for any $n$ real numbers $v_1 ... v_n$ and i.i.d variables $X_1 ... X_n$ with distribution $D$, the random variable $\sum_i v_i X_i$ has the same distribution as the variable $(\sum_i |v_i|^p)^{1/p} X$, where $X$ is a random variable with distribution $D$ [13].*

According to [13], stable distributions exist for any $p \in (0, 2]$. As for $p = 1$ and $p = 2$ we have:

- a **Cauchy** distribution $D_C$, defined by the density function $c(x) = \frac{1}{\pi} \frac{1}{1+x^2}$, which is 1-stable.

- a **Gaussian** (**normal**) distribution $D_G$, defined by the density function $g(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$, which is 2-stable.

From a practical point of view, despite the lack of closed form density and distribution functions, it is known [13] that one can generate $p$-stable random variables essentially from two independent variables distributed uniformly over [0,1].

In the case of $p$-stable LSH, for each $d$-dimensional data point $v$, the hashing scheme considers $k$ independent hash functions in forms of (2.1).

$$h_{a,B}(v) = \lfloor \frac{a \cdot v + B}{W} \rfloor \tag{2.1}$$

where $a$ is a $d$-dimensional vector whose elements are chosen independently from a $p$-stable distribution, $W \in \mathbb{R}$, and $B$ is drawn uniformly from [0,$W$]. Each hash function maps a $d$-dimensional data point to an integer. With $k$ such hash functions, the final result is an integer vector of dimension $k$ in forms of (2.2).

$$g(v) = (h_{a_1,B_1}(v), ..., h_{a_k,B_k}(v)) \tag{2.2}$$

In LSH-based schemes, in order to achieve a high search accuracy, multiple hash tables need to be constructed, each with a new instance of $g(v)$ function encapsulating $k$ hash functions. Experiments [18] show that the number of hash tables needed can reach up to over a hundred [19]. While this would cause issues in centralized settings, it is not constraining in P2P settings. But on the other hand it raises other problems specific to a P2P environment. For example, in order to visit all the hash tables, a large number of peers may need to be visited by the query. This problem and its solutions are addressed in our System Architecture (Chapter 3). It should be mentioned that in this work the Normal distribution is used as our $p$-stable distribution. Appendix A discussed the LSH parameters and how to tune them according to the system requirements.

# Chapter 3

# System Architecture

In this chapter we present the design of our system architecture with more details. In Section 3.1 we start by describing the underlying Distributed Hash Table (DHT) system and its attributes on which we can build our system. In Section 3.2, we discuss the Locality Sensitive Hashing (LSH) technique that we use to index the content on the underlying DHT: i.e., how the outcome of the LSH subsystem can be mapped to the peer identifier space of the underlying DHT system. Section 3.2 also explains how the system satisfies the basic requirements for both high precision queries and query load balancing, through the creation of local DHTs and the incorporation of gateway peers. Furthermore, in Section 3.3 we explain how this technique enables us to predict the pattern of content distribution and thus the place of hotspots in the network and how an LSH-based indexing technique can be used to ensure a fair load balancing on the peers. Section 3.4 presents the query processing technique and discusses the forwarding and processing schemes used by each of the query types.

## 3.1 DHT-based overlay network

In our system, we need an underlying peer-to-peer structure that can provide us with a linear peer identifier space. One of the best choices available in this research area is the Chord [38] P2P structure. Chord network incorporates a cyclic ID space in which we have $N$ peers $P_1, ..., P_N$ . In this network each peer knows its immediate neighboring peers, namely the *predecessor* and *successor* peers. The Chord network uses a Distributed Hash Table (DHT) to store and search for the data or association. The DHT in Chord maintains the key-value pairs by mapping the key (here the content or the music track) to the peer identifier space that is responsible for the key. In Chord DHT, each peer is responsible for all the keys that fall between its peer ID (in our design an integer number) and its predecessor's peer ID. To improve lookup performance, Chord requires each node to maintain a *finger table* containing up to $m$ entries. This efficient design let us reduce the number of nodes to be contacted to $O(\log N)$ in order to find a key. Using finger tables beside the logical ring structure provides us with a tradeoff between the space complexity and the number of hops required to reach the destination peer maintaining the key-value pair that we are looking for. The details of Chord can be found in [38].

We have chosen Chord as our underlying network for the sake of the following properties:

- **Linear peer identifier space:** As described above, peers are sorted in a linear, cyclic space. Considering that the LSH technique used in this work finally maps the data points to a linear peer identifier space, we can use the Chord P2P structure in our design. The mapping would be discussed in Section 3.2.
- **Scalability:** Due to the efficient design of Chord, its lookup routing performance scales logarithmically with regards to the number of peers in the system.
- **Self organization:** As a result of decentralization in structured P2P networks, there

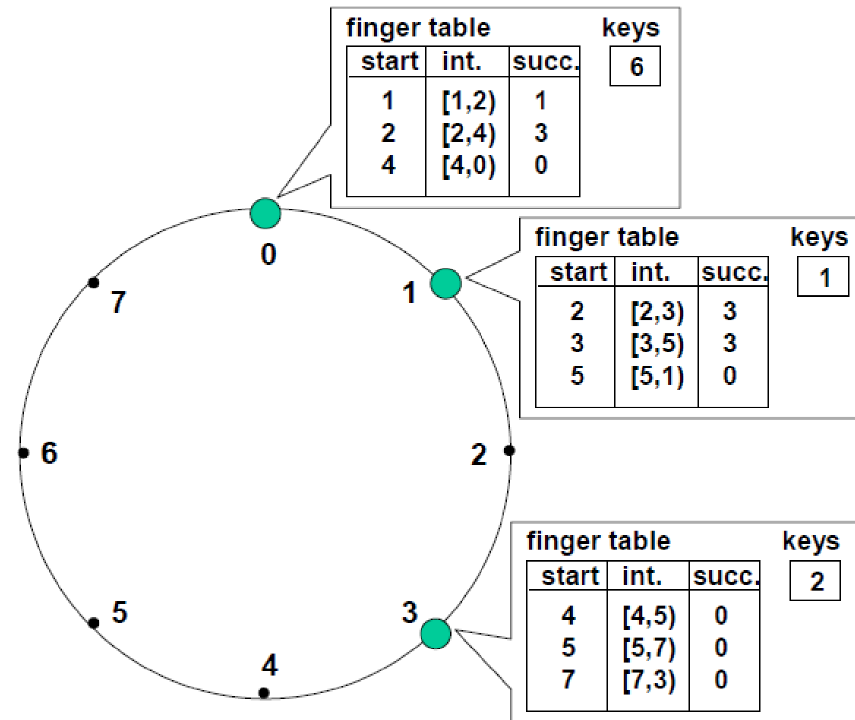


Figure 3.1: An identifier circle and three existing nodes 0, 1, and 3 and their finger tables (picture taken from [38])

is no peer that centrally coordinates the system activity or a centralized database that stores the global system information. Therefore peers can self-organize themselves and maintain the P2P network. Having such a property, each peer can decide to arbitrarily leave or join the system with a minimum number of message transfers in order to maintain the system.

- **Well investigated area:** There has been a lot of research done on the Chord network, improving its reliability in terms of fault tolerance, load balancing and security [24, 35, 37, 33]. This would allow further improvements to our system by incorporating the techniques introduced by other research. It is also worth mentioning that the Chord paper was one of the winners of the ACM SIGCOMM's Test of Time Paper Award this year (2011).

- **Ease of implementation:** Because of its simplicity, basic functionalities of the Chord DHT can be easily implemented and used as an underlying overlay.

The Chord implementation that we have used is a plug-in component for PeerSim [23] P2P simulator written in Java language. Section 3.1.1 discusses how the simulator is used in our work.

### 3.1.1 P2P Simulator

In this work we choose to simulate P2P networks with thousands of peers using the Chord protocol as their routing algorithm, while incorporating the Locality Sensitive Hashing data indexing technique. The purpose is to evaluate the LSH technique's performance when coupled with the Chord structure. Choosing a dynamic network simulator which can act as a reliable platform for our implementation is a crucial decision. The simulator should be able to easily handle a huge number of peers while not losing accuracy and be easily extensible while giving us a good set of components to work with for the performance evaluation. All of these properties can be found in a free P2P simulator called PeerSim, which has been developed in Java language, supporting extreme scalability and dynamisity. PeerSim is composed of two simulation engines, a simplified (cycle-based) and an event driven engine. Both engines are supported by many simple and extensible plug-in components with a flexible configuration mechanism [23].

The cycle-based engine provides a great scalability using some simplifying assumptions such as ignoring the transport layer messages in the communication protocol stack, while the event-based engine is less efficient but more realistic. It is also worthy mentioning that the cycle-based protocols can be adopted by the event-based engine too. In this work, we have used the event-based engine to simulate 100,000 nodes in a P2P network.

The PeerSim was initiated as a part of EU projects BISON and DELIS and now it is being well maintained by the original authors and community contributors, with lots of

extensions such as the implementations for Chord, Pastry, Kademlia, Skipnet, BitTorrent, and some gossiping algorithms.

## 3.2 Data indexing based on Locality Sensitive Hashing

### 3.2.1 Mapping to the peer identifier space

As described in the previous section, the Chord DHT system associates a Chord ID to each peer in the network. The network maps the key associated with each data point (in our case a song) to a peer's Chord ID. In the original Chord network the keys are calculated based on a consistent hash algorithm, known by all the peers. By storing the key/data on the peers it would be possible to find the location of the data point in a logarithmic time.

In our case, the Chord IDs are Java Big Integers. Chord network originally uses a uniform hash function which uniformly distributes the data among the buckets and its output is unpredictable. The problem with the uniform hash functions is that they cannot preserve the locality when the data points are being mapped to the peer identifier space. This means that there is no guarantee that the data points that are close to each other (in terms of their distance) will fall close to each other in the peer identifier space.

As we discussed in Chapter 2, an integer vector is created for each data point using $p$-stable LSH. This integer vector represents the data point in a $k$-dimensional space ($k$ is the number of hash functions used). We want to map this $k$-dimensional vector to an integer number which represents a peer ID in the Chord network.

In order to preserve the locality, we are interested in a mapping function which satisfies the following properties:

- *Property 1*: Assign buckets likely to hold similar data to the same peer.
- *Property 2*: Have a predictable output distribution which fosters fair load balancing.

Figure 3.2 shows an illustration of the overall mapping from the $d$-dimensional feature space to the $k$-dimensional bucket space and finally to 1-dimensional peer identifier space using a $\xi$ function. Also Figure 3.3 shows the overall mapping from the $d$-dimensional space to the local DHTs.

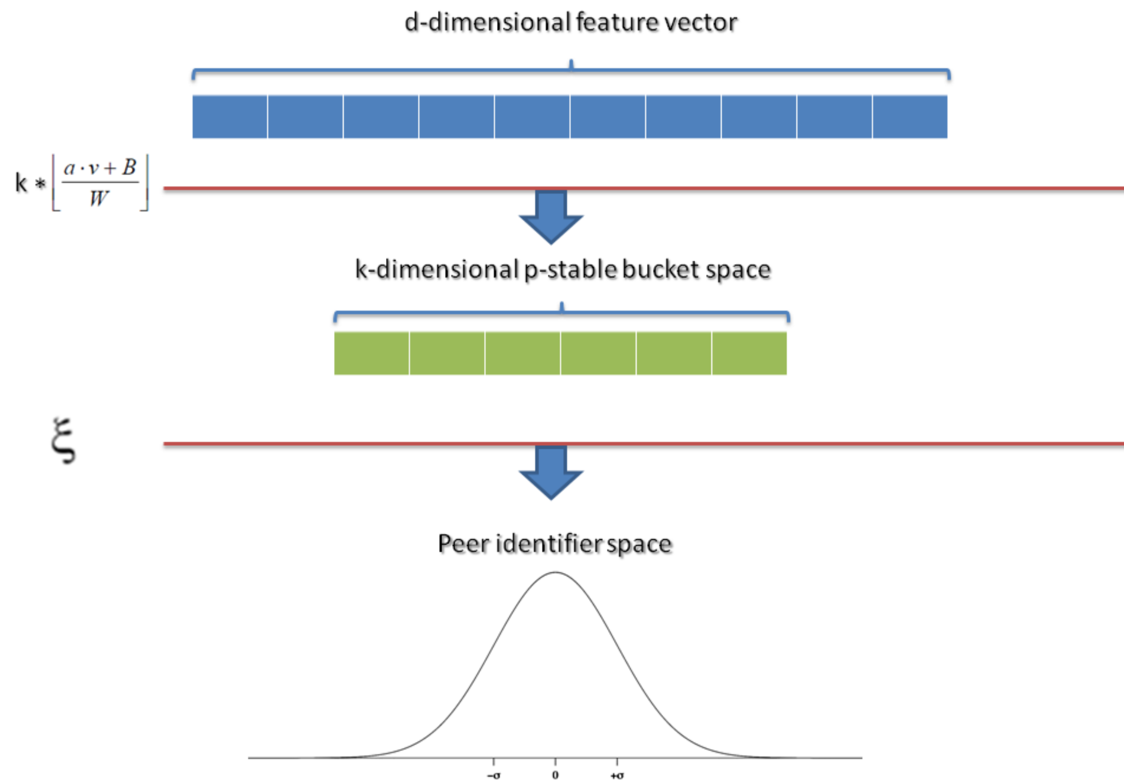


Figure 3.2: Mapping from the d-dimensional feature space to the peer identifier space

Now we discuss the actual mapping process. First let us define the meaning of similar buckets: Similar buckets are the buckets likely to hold data close to each other in the feature space.

According to Section 2.4, the first condition of LSH which describes the probabilistic behavior of it, states that close data points are more likely to be mapped to the same bucket

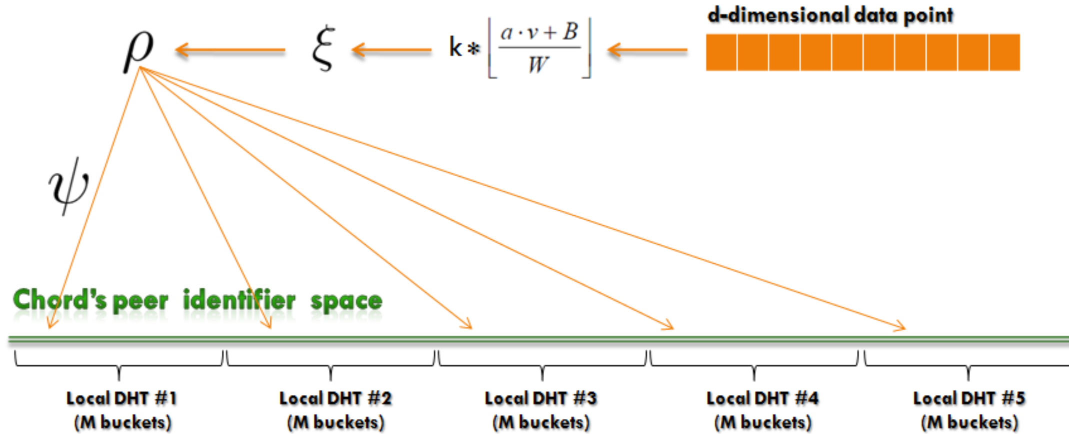


Figure 3.3: Mapping from the d-dimensional feature space to the local DHTs

but we have no clue which bucket will hold those data points. This challenge has been addressed in [26, 31] but in a query dependent way.

[19] gives us a more generalized solution, making the system independent of the queries. We may rephrase the new problem as follows:

Using the hash functions in the form of (2.1), close points should have a higher probability of being mapped to close integers, i.e., integers with a small $l_1$ distance.

In other words, we need to label buckets in a way that their $l_1$ distance can capture the distance between the buckets, recalling that *buckets likely to hold close data have small $l_1$ distance to each other*.

Let the following $g$ function be a wrapper for the $k$ hash functions in the form of (2.1) and $v$ be a $d$-dimensional feature vector in the form of (2.2)

The labeling can be done by concatenating all the hash values resulted by the aforementioned hash functions. In order to have a better insight, the reader is well advised to refer to the proofs of the *Theorem 1* and *Theorem 2* from [19]. The proofs are to show that $l_1$ distance can capture the distance between buckets in terms of the probability of holding close data: Given bucket labels $b_1$,$b_2$ and $b_3$ which are integer vectors of dimension $k$, if $||b_1 - b_2||_1 < ||b_1 - b_3||_1$, then $b_1$ and $b_2$ have a higher probability to hold similar data than

$b_1$ and $b_3$.

**Theorem 1** For any three points $v_1, v_2, q \in S$ where $\|q - v_1\|_2 = c_1$ and $\|q - v_2\|_2 = c_2$ and $c_1 < c_2$ the following inequality holds:

$$\Pr(\mid h(q) - h(v_1) \mid \leq \delta) \geq \Pr(\mid h(q) - h(v_2) \mid \leq \delta)$$

**Theorem 2** For any two points $q, v \in S$, $\Pr(\mid h(q) - h(v) \mid = \delta))$ is monotonically decreasing in terms of $\delta$.

Now it is shown that using specific mapping functions we can index the data points in a way that closer data points fall beside each other in the destination space, so we introduce the **linear mapping** scheme which maps the data from the bucket space to the peer identifier space in the next section.

**Linear mapping based on Sum**

In this work, we use a linear mapping based on the *Sum* function suggested by [19] to place (index) the data points on the peers. The function works as follows:

$$\xi_{sum} = \sum_{i=1}^{k} b_i \tag{3.1}$$

The mapping function $\xi_{sum}$ is used to map the $k$-dimensional vector of integers $b_i$ (the $k$ LSH outputs), to the 1-dimensional peer identifier space of the underlying Chord structure.

The idea behind the technique is that the *Sum* function treats all bucket label parts $b_i$ (the $k$ hash values calculated by LSH) equally and smooths out the minor differences in the $b_i$ values and finally gives us a single integer, representing the place that the data point should be indexed in, while preserving the locality of the original data points.

Here using [19] and [20] we explain using $p$-stable LSH and its properties we can satisfy the properties discussed in Section 3.2.

As discussed in Section 3.2, the first property states that the mapping function should assign buckets likely to hold similar data to the same peer. Also as discussed in the previous section, we know that the buckets which are more likely to hold similar data have small $l_1$ distance to each other. Considering $\xi_{sum} = \sum_{i=1}^{k} b_i$, $\mid \xi_{sum}(b_1) - \xi_{sum}(b_2) \mid = \mid (b_{11} - b_{21}) + ... + (b_{1k} - b_{2k}) \mid \leq \|b_1 - b_2\|_1$. This means that if the buckets $b_1$ and $b_2$ are likely to hold similar data their $\xi_{sum}(b_1)$ and $\xi_{sum}(b_2)$ will be close to each other as well. So if we use a Chord structure as our underlying network, we can maintain each bucket on a subset of closed nodes (in terms of their peer ID) and those nodes are going to have similar data to each other with a high probability.

The second property requires a predictable output distribution for the mapping function. In (2.1), $a$ and $v$ are both $d$-dimensional points, where the elements of $a$ are chosen from a Normal distribution, with mean 0 and standard deviation 1, i.e., $N(0, 1)$. This leads to a Normal distribution of $a \cdot v$ with mean 0 and variance $\|v\|_2$. This means that for small values of $W$, $h_{a,B}(v)$ is distributed according to the Normal distribution $N(\frac{W}{2W}, \frac{\|v\|_2}{W})$. Therefore, using the properties of Normal distributions, we can calculate the distribution of $\xi_{sum}(g(v))$, which is the sum of all the elements in the $k$-dimensional vector of hash values as follows:

$$\xi_{sum}(g(v)) \sim N(\frac{k}{2}, \frac{\sqrt{k}\|v\|_2}{W})$$

The above distribution describes how a single data point's $\xi_{sum}$ is distributed, but we are more interested in the global picture which includes all data points $v_1, v_2, ..., v_n$ (assuming there are $n$ data points in the system). Therefore, by first projecting the data points using the $p$-stable LSH and then mapping to $\mathbb{Z}$ by $\xi_{sum}$, the results will follow this distribution:

$$N(\frac{k}{2}, \frac{\sqrt{k \sum_i \|v_i\|_2^2}}{\sqrt{n}W})$$

The above equation suggests that we can predict the global output of the $\xi_{sum}$ if we know the mean of the data point's $l_2$ norm. It can be assumed that this mean is global knowledge, known to all the peers in the system, although this is only required by the system startup phase when the Local DHTs are being created. This phase is discussed in detail in Section 3.2.2. Having a bootstrap server, we can populate the required information to all the peers in the network, letting them know about the current mean of the dataset. Considering music tracks as our main dataset we can update the indexed data with the new mean, as we exceed a certain threshold when too many new music tracks are added to the system.

### 3.2.2 Local DHT creation

In the previous section we explained how the data points can be indexed using LSH and then a mapping function ($\xi$). Due to the fact that Locality Sensitive Hashing is a probabilistic algorithm, we can assume that with a high probability close data points will be indexed close to each other in the peer identifier space. On the other hand, a reliable system in terms of search accuracy would require a high precision in the search procedure. That is why we can create multiple indexes of the same data point in the system; in other words we may create multiple *hash tables* and maintain each one on a particular subset of peers. When we query a point looking for either the closest points (e.g., KNN query) or points in a specific range (e.g., Range query) to a reference data point, we may send the query to multiple hash tables and aggregate the results suggested by each *hash table* on the initializing peer. In this way, the probabilistic nature of the indexed data points in the system results in a more accurate search, handling the random essence of the LSH and its noise.

Now the following questions arises: *How are we supposed to choose the peers that maintain each hash table?* In order to map a particular domain of integer values (here, the results of the $\xi$ function) to a subset of peers, we are required to know the size and

distribution of the domain. According to Section 3.2.1 values generated by $\xi$ follow a known distribution, so we can use this information to distribute our index on the peers. Consider a linear bucket space of $M$ buckets, in which we want to distribute the values generated by $\xi$ mapping. 95-99.7 rule states that $68\%$ of the data lies within one standard deviation from the mean of the distribution; $95\%$ of the data is located within two standard deviations from the mean and $99.7\%$ of the data is located within three standard deviations from the mean. Knowing the $\mu$ (mean) and $\sigma$ (variance) of the results generated by $\xi$ we can choose the first bucket (at position 1) to be responsible for the values starting at $\mu - 2 * \sigma$ and the last bucket (at position M) to be responsible for values ending at $\mu + 2 * \sigma$. We can assume that the span of four standard deviations is enough to cover a broad set of values. Using the same rule we can map the remaining data to the considered range via a simple modulo operation:

$$\psi(value) := (\frac{value - (\mu - 2 * \sigma)}{4 * \sigma} * M) \mod M \tag{3.2}$$

Now we need to maintain each hash table on a subset of peers. The number of peers in each hash table should be an order of magnitude smaller than the number of peers in the global network. To do so, we create $l$ separate dynamic DHTs, each maintaining a hash table on the peers which we choose using the $\rho$ function showed in (3.3).

$$\rho(value, l) := (\psi(value) + Hash(l_i)) \mod |G| \tag{3.3}$$

The $Hash$ function used in the $\rho$ function is meant to apply a displacement for the start of each hash table in the peer identifier space and the $l_i$ is the hash table $id$. The $Hash$ function is also necessary for global load balancing. This would make sure that the subsets of peers in each table are not overlapping with each other. The $\rho$ function assigns at most $M$ peers to each local DHT. Now in order to access a local DHT, as [20] suggests, we use a

set of gateway peers to act as start-up peers, and also as the entry points to each local DHT.

Figure 3.4 depicts the distribution of the indexed data on the global network. Each bell-shaped curve is indicating a single local DHT. The network consists of 1100 peers and 10 Local DHTs, each having 100 nodes (the extra 100 peers are to make sure there is no overlap between the local DHTs)

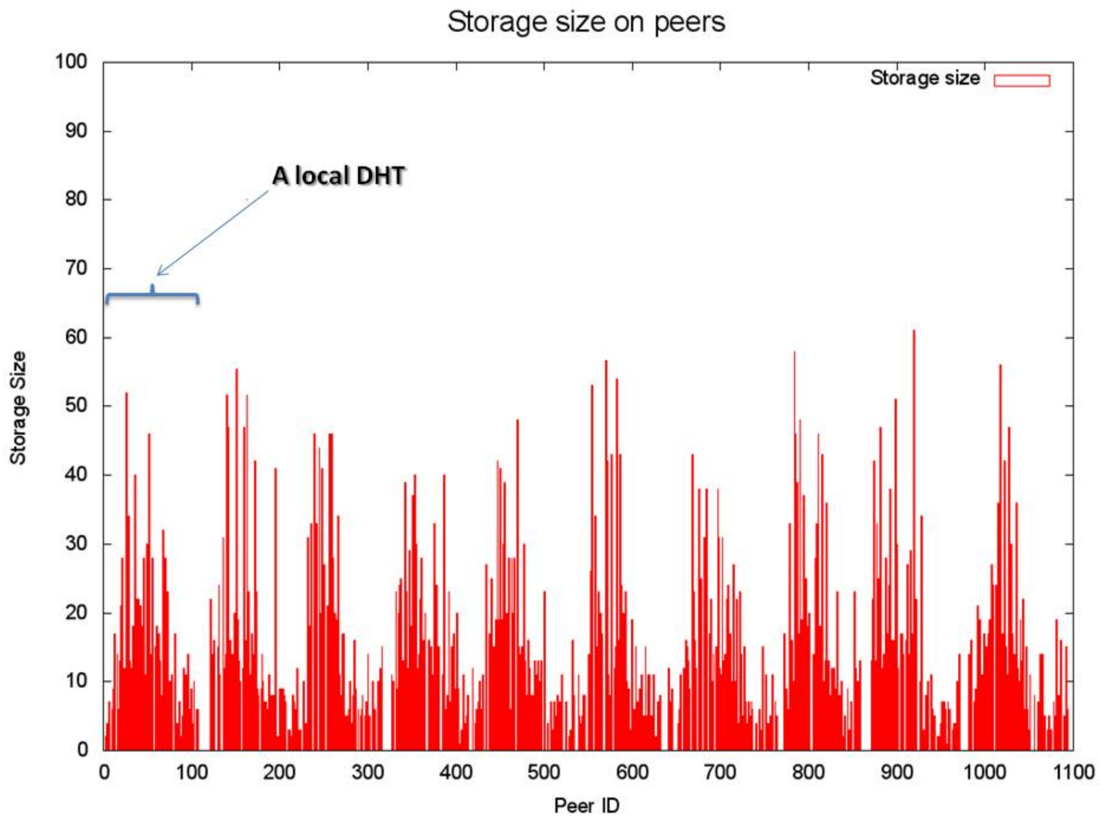


Figure 3.4: 10 Local DHTs on a Chord network with 1100 peers

### 3.2.3 Local DHT entry points

As discussed above, in order to make sure each local DHT has at least a number of start-up peers also acting as the entry points we assign $\gamma$ gateway peers for each local DHT using the algorithm from [19], listed in Table 3.1.

```
1  for (int i = 0 ; i < Gamma ; i++)
2  {
3      double Mean = BootstrapServer.Mean[tableId];
4      double Sigma = BootstrapServer.Sigma[tableId];
5      sample = Mean + rand.nextGaussian() * Sigma;
6      sampleSet.add(sample);
7
8      double targetNode = MainStorage.MainIndex.Rho(sample, tableId
           );
9
10     BigInteger targetId = BootstrapServer.maxID.multiply(
           BigInteger.valueOf((long) targetNode));
11     targetId = targetId.divide(BigInteger.valueOf((long) Network.
           size()));
12
13     Node P = (Node) findId(targetId,0,Network.size()-1);
14     if (i==0)
15     {
16         localDHT = CreateDHT(P);
17     }
18     else
19     {
20         localDHT = DHT.get(tableId);
21         Node P1 = (Node) findId(targetId,0,Network.size()-1);
22         Join(P1, localDHT,true); // the true value tells the join
               function to join the peer as a gateway node
23     }
24 }
```

Table 3.1: Gateway peer selection and local DHT initialization

The algorithm in Table 3.1, actually samples $\gamma$ values from the distribution of the values generated by $\psi$ for each table and uses them as the entry points. It is logical to maintain the list of gateway peers in each local DHT on the bootstrap server due to the limited number of gateway peers. Having a list of gateway peers as global knowledge, whenever we want to query a local DHT (either KNN or Range query), first we send the query to a randomly chosen gateway in each DHT, and then the gateway peer is responsible for finding the destination peer holding the specified $\rho$ value in that local DHT.

Incorporating gateway peers, and randomly choosing among them as the entry points is another means of load balancing in this system, but the main load balancing technique is discussed in Section 3.3

## 3.3 Load balancing

As we know, the main purpose of a distributed system is to equally distribute the processing and data load on its peers. In this section first we discuss the load balancing problem associated with the technique used in [19] and then we will introduce our new load balanced scheme based on the bucket size prediction. So first we discuss the advantages of using LSH as the indexing technique, and then we will explain how to exploit the LSH properties to achieve a fair load balancing on the peers. It should be noted that this section includes the main contribution of our work.

### 3.3.1 Load blancing with bucket width prediction

As discussed in Section 3.2.1, we see that if we distribute our data points according to the linear mapping, because of the nice properties of $p$-stable distributions and their usage in the LSH algorithm, we can predict the output data distribution. We know that the $\psi$ function as shown in (3.2) distributes the data points in $M$ buckets and the output would

follow a Normal distribution. But the problem is that we do not know how many data points will fall in each bucket. This might be problematic in a sense that the size of each bucket might grow based on the density of the indexed data that falls into that bucket. So if for example in case of highly skewed data points the output would be more skewed and we will have too many data points indexed inside the buckets close to the mean while other buckets are almost empty. Then the peer or peers that are associated with the buckets close to the mean have to hold a large number of indexed data points and of course have to handle a huge number of requests. So it can be inferred that the naive bucket assignment technique might contradict with our load balancing requirements. Figure 3.5 shows the number of indexed data points assigned to each bucket in a local DHT for a skewed dataset like the *mixed dataset* described in Section 4.1.

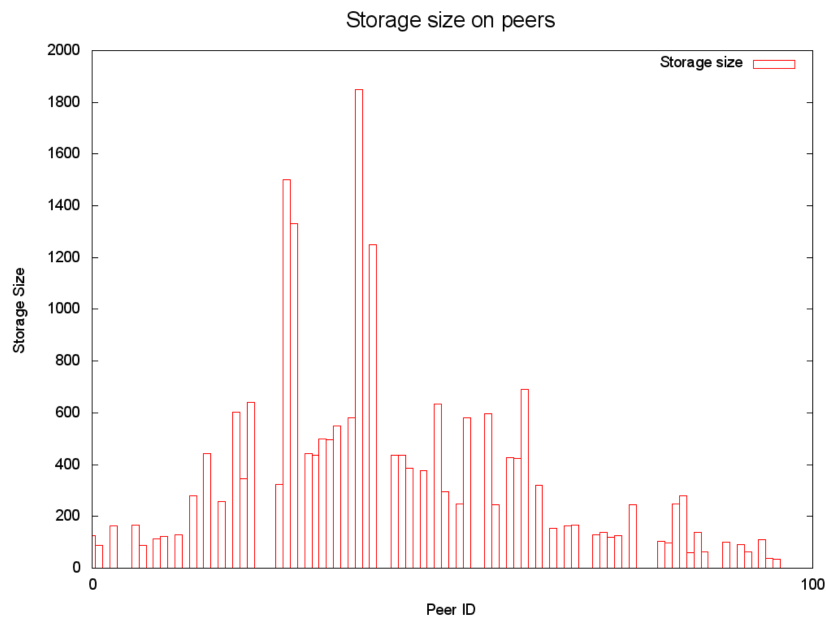


Figure 3.5: Skewed load on peers in a local DHT

The naive $\psi$ function (3.2) assigns fixed bucket sizes to all the buckets. This is only meaningful if we assume the data set is not skewed at all and its distribution is completely symmetric, which is clearly not true for our sample datasets and most of the real-world

datasets.

On the other hand, by knowing the distribution of the $\xi$ function (3.1), we can allocate different bucket widths to different bins in the $\psi$ function. This is done by manipulating the $\psi$ function in a way that it responds to the density of the indexed data in each bucket. Our technique is based on the fact that we can assign larger bucket sizes to the buckets so that they are likely to hold a larger number of indexed data points. This means that we simply assign smaller bucket widths to buckets closer to the data mean and as we go further away from the mean we increase the bucket widths.

Let $B_i$ be the width of bucket $i$, $\mu$ and $\sigma$ be the mean and standard deviation of the data generated by $\xi_{sum}$ function and $M$ be the number of buckets. Knowing the properties of the Normal distribution, again we use the 95-99.7 rule (explained in Section 3.2.2). According to this rule we can come up with the following equation:

$$CDF(\mu - 2 * \sigma + B_i) = CDF(\mu - 2 * \sigma + \sum_{j=0}^{i-1} B_j) + \frac{0.95}{M} \tag{3.4}$$

Therefore the width of the $i$'th bucket can be calculated knowing the sum of the previous bucket widths:

$$B_i = CDF^{-1}(CDF(\mu - 2 * \sigma + \sum_{j=0}^{i-1} B_j) + \frac{0.95}{M}) - (\mu - 2 * \sigma) \tag{3.5}$$

Now we know the starting point and the width of all the buckets, so we map the indexed data using the $\psi_{enhanced}$ function listed in Table 3.2.

In Table 3.2, oldPsi is also used, which is the implementation of the $\psi$ function used by the authors of [20]. Table 3.3 shows the oldPsi function implementation.

```
1  public double enhancedPsi(double value, int TableID)
2  {
3      // Find which bucket does the value belong to
4      for (int i=0 ; i<M-1 ; i++)
5      {
6          if (value>bucketStart.get(TableID)[i] &&
7              value<bucketStart.get(TableID)[i+1])
8          {
9              return i;
10         }
11     }
12     if (value>mu[TableID]+2*sigma[TableID])
13     {
14         return oldPsi(value,TableID);
15     }
16     if (value<mu[TableID]-2*sigma[TableID])
17     {
18         return oldPsi(value,TableID);
19     }
20     return 0;
21 }
```

Table 3.2: $\psi_{enhanced}$ function

```java
public double oldPsi(double value, int TableID)
{
    double numertator = value-mu[TableID]-2*sigma[TableID];
    double denominator = (4*sigma[TableID]);
    double result = numertator / denominator;

    result = ((double) result * M) % M;

    if (result <0)
        result = M + result;

    return result;
}
```

Table 3.3: $\psi_{old}$ function

The logic behind our enhanced $\psi$ function is that it distributes $95\%$ of the data among the M buckets allocated within $[\mu - 2 * \sigma, \mu + 2 * \sigma]$ range and distributes the remaining $5\%$ of the data points using the naive $\psi_{old}$ function. The idea is illustrated in Figure 3.6.

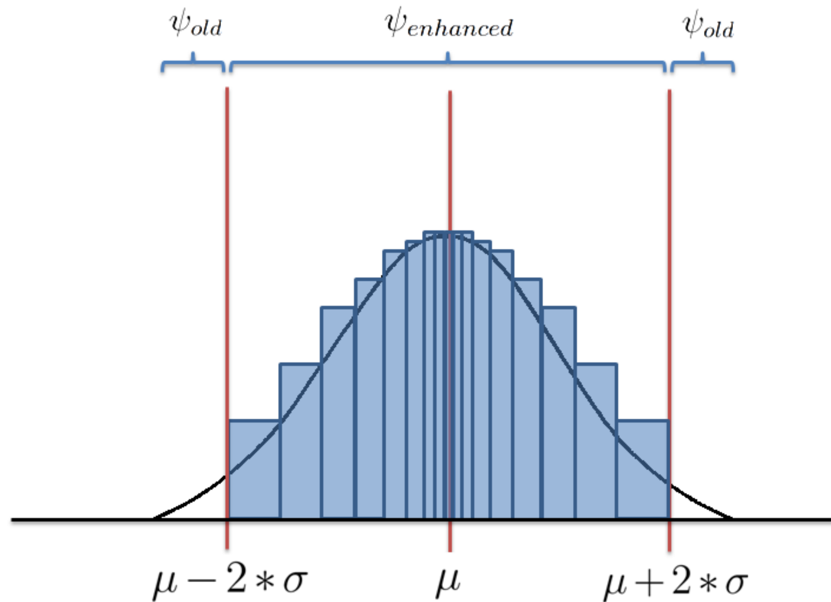


Figure 3.6: $\psi_{enhanced}$ and bucket width prediction

In Chapter 4 (Performance Evaluation) we will discuss in more details the performance of our load balancing scheme which is integrated into the indexing algorithm.

## 3.4 Queries

In any kind of database system, one of the main concerns is how to find specific data which match our criteria among others. This problem becomes a more essential part of the system when it comes to distributed architectures. This is due to the factors such as network traffic, load and delay among the peers that play an important role in planning for the query forwarding and processing strategies. A distributed system which tends to support information retrieval should be able to support two basic query types, namely K-

Nearest-Neighbor (KNN) query and Range query.

The query forwarding and processing scheme used by our system tends to efficiently address the aforementioned query types using the $p$-stable LSH indexing technique. The KNN and Range query types and their forwarding scheme are discussed in details in this section and their performance results will be presented and discussed in Chapter 4.

### 3.4.1 KNN Query processing

Let us assume that a peer is interested in finding the $K$ most similar items to a specified item among all the items in a dataset. In our system, this is analogous to finding the most similar music tracks to a music track that we are interested in; in order to find for instance five songs that sound similar to the one that we like, we can perform a 5-NN query in the network.

Let's assume that we have $l$ replicas for each data point, which means we have $l$ hash tables or $l$ local DHTs in our global Chord ring. When a peer needs to perform a KNN query, it should have two things ready: a $d$ dimensional query reference point $q = (q_1, q_2, ..., q_d)$ and a $K$. The peer first maps the query point to buckets $g_1(q), g_2(q), ..., g_l(q)$ each associated with a local DHT. Then it randomly chooses a gateway peer as an entry point to each local DHT. As mentioned in Section 3.2.3, each local DHT has a number of entry points, known to all the peers. The list of gateway peers will be available from the bootstrap server. Then the peer sends the query to the selected entry point in each hash table. The gateway peer automatically forwards the query to the peer $P$ that is responsible for the part of the global index that holds the $g_i(q)$. The responsible peer can be found using the $\xi(g_i(q))$. The following steps take place when $P$ receives the KNN query:

1. Run the query locally by performing a full scan on the local storage

2. Send the K nearest items to the initializing peer

3. Forward the query to the next peer most likely to hold related information

The local full scan is done in a naive way by calculating the distance of the query point to all the items in the local storage of the peer. In our system, the global search performance is decoupled from the local search performance. This is due to the negligible effect of the local search time on global search time. We will discuss our reasons in more details in Chapter 4. After sending the matching results (if there were any), the peer forwards the query to the next peer using the linear forwarding method.

**Linear Forwarding**

As the name suggests, Linear KNN query forwarding forwards the query to the predecessor and successor peers of the peer $P$ (responsible for $g_i(q)$). When the peer $P$ receives the query, after a full scan in its storage it finds the distance of the $K-th$ item regarding the point $q$. This distance which is noted as $\tau$ should be passed along with the query to the next peer(s). If the peer to pass the query is the first responsible peer met in the local DHT, then it passes the query to both its successor and predecessor peers otherwise the query should be passed along in one direction. The stopping conditions for the query forwarding can be listed as below:

1. If the result set of the local full scan has no items
2. If the closest item (in terms of its distance) in the result set from the local scan has a distance greater than $\tau$ (received with the query)

The algorithm is summerized in Table 3.4

```
1 public int linearKNN(Node node,LookUpMessage message)
2 {
3 // Perform a full scan on local storage
```

```
4  ArrayList<RhoToVectorClass> localResultSet = localSearch(node,
       message);
5  // If the Local Results Set is empty or its minimum distance is
       larger than the threshold then stop the search
6  if (localResultSet.isEmpty() || localResultSet.get(0).Distance >
       (message.Tau/BootstrapServer.Alpha))
7  {
8          return 0;
9  }
10 else // If the Local Results Set is not empty
11 {
12     if ( (!localResultSet.isEmpty()))
13     {   // Select the K nearest items to the query point
14         for (int i=0 ; i< localResultSet.size() && i < message.
               KNN; i++)
15         {
16             if (localResultSet.get(i).Distance < message.Tau/
                   BootstrapServer.Alpha)
17                 if (!message.resultSet.contains(localResultSet.
                       get(i)))
18                 {
19                     message.resultSet.add(localResultSet.get(i));
20                 }
21             else
22                 break;
23         }
24
25         // Sort the results in the message
```

```
26            RhoToVectorComparator rc = new RhoToVectorComparator();
27            Collections.sort(message.resultSet,rc);
28
29            int K=Math.min(message.resultSet.size()-1,message.KNN -
                  1);
30
31            if (!message.resultSet.isEmpty())
32            {
33                // Attach the value of Tau to the message
34                message.Tau = message.resultSet.get(K).Distance;
35
36                // Send the results to the initiator peer
37                sendResults(node, message,0);
38
39                // Forward the message to next peer(s) : predecessor/
                     sucessor
40                forwardMsg(node, message);
41            }
42        }
43    }
44        return 1;
45    }
```

Table 3.4: Linear KNN algorithm

So, if the peer does not have any results for a query, it stops the forwarding procedure. As mentioned earlier, each query (Lookup Message) is provided with a direction field, notifying the peer about the forwarding direction. Also a relaxing parameter $\alpha$ can be used to manipulate the stopping condition as follows: *If* $d_{best}$ *(the distance of the closest item to*

*the query point in local storage) is greater than $\tau/\alpha$ then stop the search.* As can be seen, a relaxing technique is adopted in Line 6 of the above algorithm. For a more aggressive search, we can set the $\alpha$ to be greater than 1 and for early stopping the $\alpha$ should be smaller than 1.

### 3.4.2 Range Query processing

The difference between the Range and KNN query is that, we need the Range query to return *all* the data points within an specific range $r$ (defined in terms of distance). This means that the stopping condition for linear forwarding algorithm should be manipulated in order to cover the range requirements.

At first glance, it seems that the Range query in this linear space can be performed similar to the KNN query. But counter-intuitively we will see how the problems associated with the linear space prevent us from dealing with linearly mapped data in the same way as we did for KNN queries. One of the important and limiting problems associated with the Range query in this linear space is the *Empty Ranges* problem. Suppose that the query is sent to a peer actually holding the data in itself. This means that due to the intrinsic characteristic of this space, most of the data points that are related to the query point are clustered on the same peer. So with a high probability, the follow up peers, will not have any data within the specified search radius which causes the forwarding algorithm to stop and we may miss some data points that are placed a small number of hops away from this peer.

The problem is depicted in Figure 3.7. As can be seen in the figure, the search stops somewhere close to Peer 19 and Peer 39, which is not correct and we are missing the data on the peers numbered between 40 and 60.

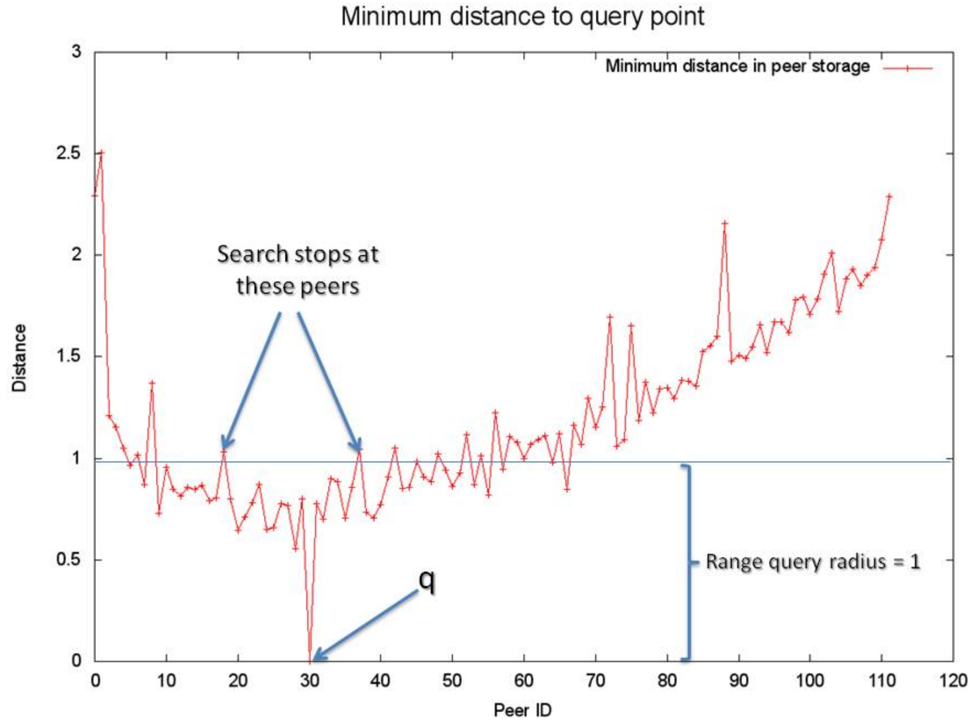


Figure 3.7: Linear Range query processing and the associated empty ranges problem

### Range Prediction and Sampling

To overcome the problem of Empty Ranges while we do not know the distribution of the data on the queried peers we use a sampling technique suggested by [19]. The idea is to divide the range $r$ into equally sized partitions and sample $s$ points within the radius $r$ from the query point $q$. We are actually extracting $s$ samples from the output of $\xi$ function within the range $lower(q, r), ..., upper(q, r)$ in a way that they produce a bucket label with maximum $l_1$ distance to the $g_{a,b}(q)$. The $P_l$ and $P_u$ which correspond to the peers holding the values $lower(q, r)$ and $upper(q, r)$ can be estimated using the following algorithm:

1. **Step 1:** Find the index of the minimum and maximum elements of $d$ dimensional vector $a$ among all $k$ hash functions within $g_i(q)$ and call them $a_i^{(-)}$ and $a_i^{(+)}$ respectively

2. **Step 2:** Manipulate the original feature vector in order to form the lower bound and the upper bound vectors that are within distance $r$ to the query point:

   - $q^{(+)} = q + j_{a_i^{(+)}} * r$
   - $q^{(-)} = q - j_{a_i^{(-)}} * r$

   where $j_i$ is the $i^{th}$ unit vector.

3. **Step 3:** Calculate the lower bound and upper bound $\xi$ values using:

   - $upper(q, r) = argmax\{\xi(g(q^{(+)}))\}$
   - $lower(q, r) = argmin\{\xi(g(q^{(-)}))\}$

4. **Step 4:** Repeat the previous steps for each hash table

After we found the $upper(q, r)$ and $lower(q, r)$, we can sample $s$ $\xi$ values within the specified range and send the queries to the peers maintaining the items that are associated with those values. This method lets us search the network more aggressively and more accurately compared to the linear range search method which is the naive version of the algorithm that starts the search from $lower(q, r)$ and ends at $upper(q, r)$.

The stopping conditions for both methods are the same, that is:

1. If a peer does not have a single item within the queried range
2. If an already queried peer is met

Linear forwarding vs. range sampling are illustrated in Figures 3.8 and 3.9.

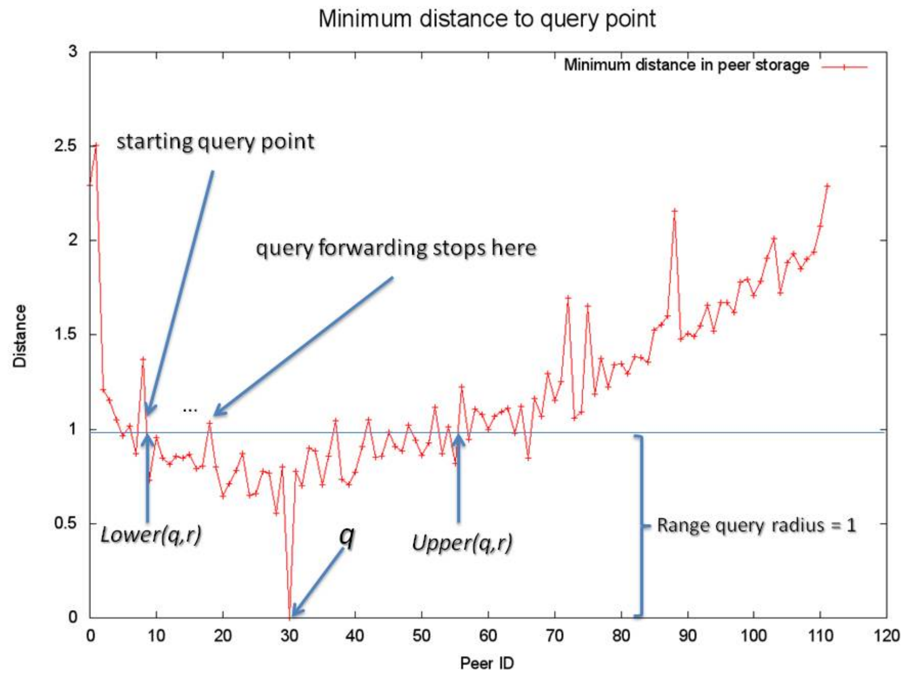


Figure 3.8: Linear Range query processing

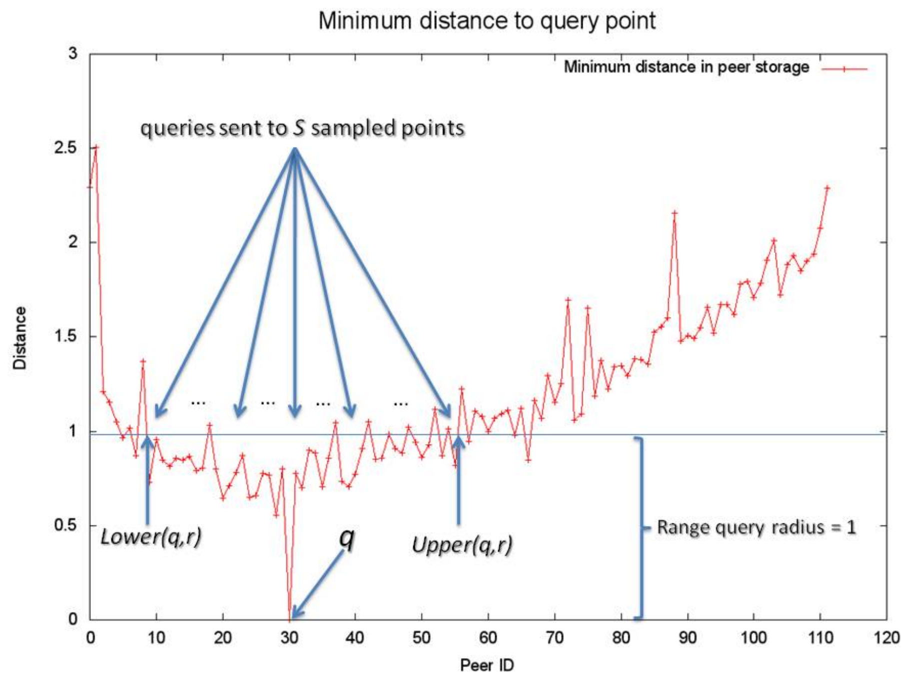


Figure 3.9: Sampling-based Range query processing

# Chapter 4

# Experimental results

In this chapter, we present and evaluate the simulation results of our proposed load balancing scheme. We explain our simulation setup and the datasets used and also explain the measures of interest. Afterwards, we compare the performance of the proposed load balancing scheme through the comparison with the method proposed in [20] and [19]. In our simulation, we consider audio features as our data points and different variations of KNN and Range queries as the query types under evaluation.

## 4.1 Experimental Setup

As discussed in 3.1.1 we use Java 1.6 to extend the PeerSim [23] P2P simulator in order to add the LSH data mapping functionality as well as query processing methods for our simulation. The following parts will describe the datasets and the overlay setup and our simulated query processing methods.

#### Datasets and overlay setup

1. **Standard dataset:** This dataset consists of 1,000 audio tracks of 30 seconds long each. It contains 10 genres, each represented by 100 tracks. The tracks are all

22,050 Hz Mono 16-bit audio files in .wav format. The dataset is provided by George Tzanetakis [40]. For each audio track, 124 features are extracted using the Marsyas framework (version 0.4.1) [41]. The feature set includes the means and variances of timberal features (e.g., time-domain Zero-Crossings, Spectral Centroid, Rolloff, Flux and Mel-Frequency Cepstral Coefficients (MFCC)) over a window of 512 milliseconds.

2. **Mixed dataset:** This dataset consists of 21,591 audio tracks of 30 seconds long each. The files were collected between $2000 - 2010$ from a variety of sources including personal CDs, microphone recordings and personal studio recordings, in order to represent a variety of recordings in a music collection while resembling an actual music collection of a user by having multiple versions of some music tracks in the dataset (e.g., live version, unplugged version and studio version). The feature set extraction uses similar configurations as the ones used in the standard dataset.

3. **Synthetic dataset:** The synthetic dataset uses the standard dataset as its seed, and consists of 1,000,000 randomly generated data points. Each feature follows the mean and variance of the features extracted from the songs in the standard dataset. This dataset is meant to test the system performance while loaded with a large number of data points.

For all datasets, Euclidean distance is used to measure the distances between the data points. We treat all dimensions equally and the features in the datasets are first normalized between 0 and 1 and then indexed onto peers. 100 query points were chosen randomly from each of the datasets and the results are averaged among 100 query runs. It should be noted that in real-world implementations we can manipulate the importance of each dimension using a weighting scheme.

### Simulated query processing methods

In order to evaluate our load balancing and data placement methods, the data is distributed on peers once with $\psi_{old}$ and once with $\psi_{enhanced}$, where $\psi_{old}$ represents the $\psi$ mapping function used in [19] and $\psi_{enhanced}$ is the enhanced $\psi$ function proposed by us to resolve the load balancing issue with $\psi_{old}$.

The LSH algorithm is configured with the following default values, unless otherwise stated in an experiment. $k = 50$, $W = 1$, $\gamma = 10$ and $L = 10$, where $k$ is the number of hash functions that the $d$-dimensional data are mapped through, $W$ is the minimum distance between two data points and $\gamma$ is the number of gateway peers in each local DHT.

For each of the mapping functions, we test the following query processing methods:

**Linear KNN query:** As described in Section 3.4.1, at query time each peer forwards the KNN queries in a linear manner, i.e., forwarding to the successor/predecessor peer.

**Linear Range query:** At query time each peer forwards the Range query to its successor/predecessor peer. This method is used as a baseline to evaluate the sampling-based Range query processing.

**Sampling-based Range query:** As described in Section 3.4.2, this query processing method uses a sampling-based method to forward the queries.

### Measures of interest

We use the following measures in order to evaluate the system:

*Gini Coefficient:* According to [32] as a measure of load imbalances we can consider the Gini coefficient, that is defined as $G = 1 - 2\int_0^1 L(x)$ where $L(x)$ is the Lorens curve of the underlying distribution. In other words, "it is defined as the relative mean difference of load, i.e. the mean of the load difference between every possible pair of peers, divided by their mean load [32]". [32] also shows that the Gini coefficient is the most appropriate

statistical metric for measuring load distribution fairness. The Gini coefficient, apart from the other three measures, is query independent and measured once to report on the peer storage load distribution.

*Number of Network Hops:* We count the number of network hops during the query execution. Network hops are one of the most important parameters in making distributed algorithms applicable in large distributed networks. Each DHT lookup causes $logN/2$ network hops (where N is the number of peers in the network, i.e. local or global). Therefore, we count the aggregate number of local and global DHT lookups and report the overall number of network hops. As mentioned in Section 3.4.1, we decouple the global search performance from the local search performance. The reason for this decision is that the cost of local query execution is considered to be negligible in our scenarios, as the DHT network lookup cost is definitely the dominating factor. According to [19], a single network hop in a wide-area network costs in average around 100ms, which is much bigger than the I/O cost, induced by a standard hard disk, with approximately 8ms for a standard hard disk seek time plus rotation latency and 100MB/s transfer rate for sequential accesses, in case of local disk access. In case we assume that we have a cluster of servers, it can be implied that the links between the computers are much more faster and reliable (resulting less than 10ms mean-hop delays) and the bottleneck would be on the local search and as a result it should be optimized as well.

*Relative recall:* This metric shows the accuracy of the search results. It is defined as the number of relevant data points among the returned data points. Also, the relevance is calculated by a full-scan run over all the data points in a dataset, considering a specific query point. We first run a full-scan on the dataset to find the real K-nearest-neighbors to a query point, and after running the simulation we calculate the fraction of retrieved points that match the full-scan results in order to calculate the recall. For Range queries, all data points in range $r$ of the query point are relevant. It should be mentioned that since we are

ranking all candidate objects and returning only the top $K$ in KNN queries and only the points within distance $r$ of the query point in Range queries, the precision is equal to the relative recall.

Table 4.1 summarizes the simulation setup parameters.

| | Standard dataset | Mixed dataset | Synthetic dataset |
|---|---|---|---|
| Number of data points | 1,000 | 21,591 | 1,000,000 |
| Number of dimensions | 124 | 124 | 124 |
| Peers in global DHT | 100,000 | 100,000 | 100,000 |
| Peers per local DHT | 100 | 100 | 100 |
| Peer ID Length | 20 | 20 | 20 |
| Peer successor list size | 20 | 20 | 20 |
| k (the number of hash functions) | 50 | 50 | 50 |
| $\alpha$ for KNN search | 1 | 1 | 1 |

Table 4.1: Simulation setup

## 4.2 Experimental Results

### 4.2.1 KNN query results

We now show the results obtained for the KNN queries. First, we should find the most appropriate number of hash tables (replicas) to be used for the next set of experiments. We should have a sufficient number of hash tables to compensate the probabilistic nature of the LSH algorithm while running a query; so we aim to find the minimum number of hash tables required to have a reasonable recall for queries by varying the number of hash tables from 1 to 10. The recall here shows the number of results relevant to the real KNN results gathered by queries sent to all hash tables. Here we show the results for both placement functions $\psi_{old}$ and $\psi_{enhanced}$ using the standard dataset which is the most realistic dataset. The number of network hops is actually the sum of the number of network hops for all the returned query responses.

Figure 4.1 shows the recall versus the number of hops in each local DHT for standard dataset, using one hash table. Figures 4.2 and 4.3 respectively illustrate the number of network hops in each local DHT required to reach a specific recall for the same dataset using five and ten hash tables.

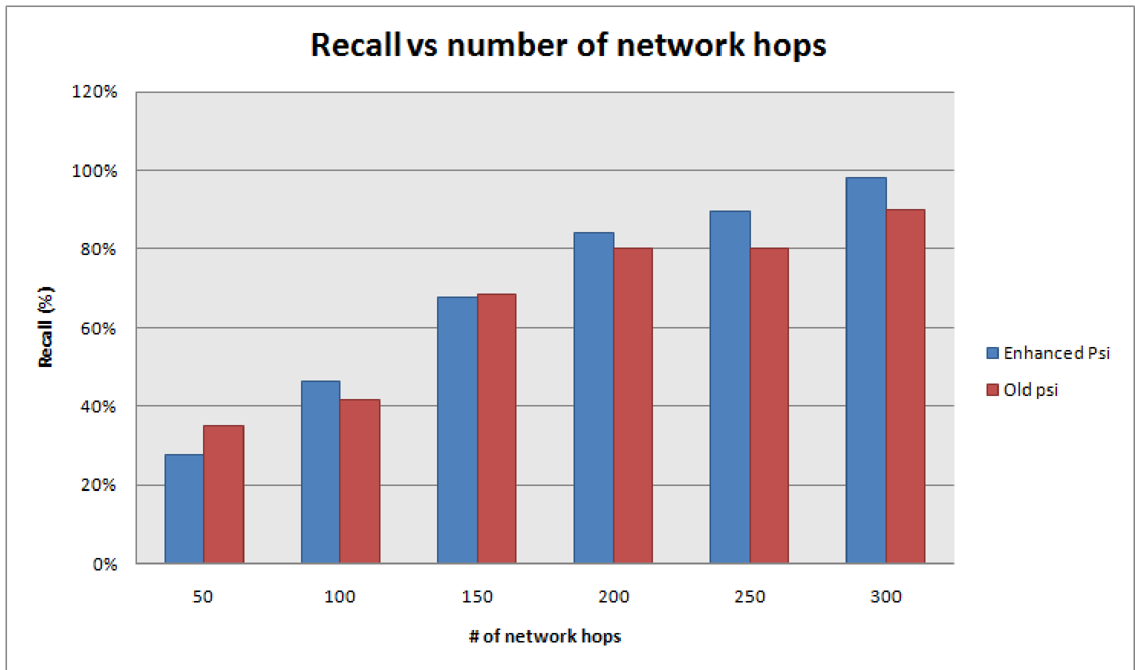


Figure 4.1: Recall vs. the number of network hops for different placement methods, employing the linear KNN query processing for the standard dataset when distributing one replica of the dataset.

As the figures illustrate, having 10 hash tables we get the best recall when comparing the aggregate number of hops that needs to incur as a result of all queries. As can be seen in the figures, the recall for one hash table (Figure 4.1) grows suddenly at the beginning and as the number of hops increases, its slope decreases; this confirms that the $\psi_{old}$ and $\psi_{enhanced}$ both preserve locality, i.e., they group similar contents to the same bucket. This observation also shows the validity of our simulator. Figure 4.1 shows the recall using both mapping functions, but we see that when our enhanced mapping is used the recall does not progress that fast. This difference in pace is because our $\psi_{enhanced}$ function more evenly

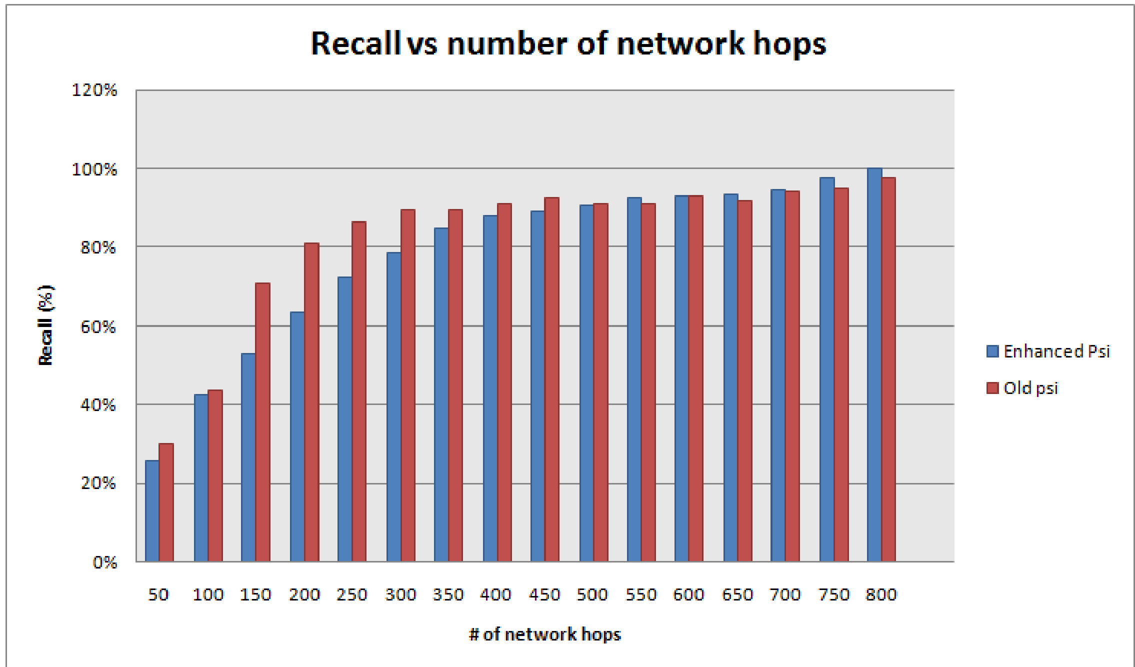


Figure 4.2: Recall vs. the number of network hops for different placement methods, employing the linear KNN query processing for the standard dataset when distributing five replicas of the dataset.

distributes the data points on the peers, and as a result the query will retrieve a more fixed fraction of relevant data points as the query is passed on.

In each figure, the number of incurred network hops for answering a query is equal to the number of hash tables (replicas) times $logN/2$. For example for $L = 1$, and 100,000 peers in the global network, we have nearly 16 hops to reach a gateway peer, and after that point the query is linearly forwarded inside a local DHT. As a result, as for the data in Figure 4.1 we need to visit 17 peers in average for each response (having around 18 responses in total) in order to reach $98\%$ recall for $\psi_{enhanced}$ and $90\%$ recall for $\psi_{old}$. Figures 4.2 and 4.3 show that having the hash tables we reach over $95\%$ recall with 10 hops inside a local DHT, and having 10 hash tables we reach $100\%$ recall with five network hops in each local DHT. This means as we increase the number of hash tables, the number of network hops required to reach a reasonable recall for queries decreases, and it is due to the probabilistic

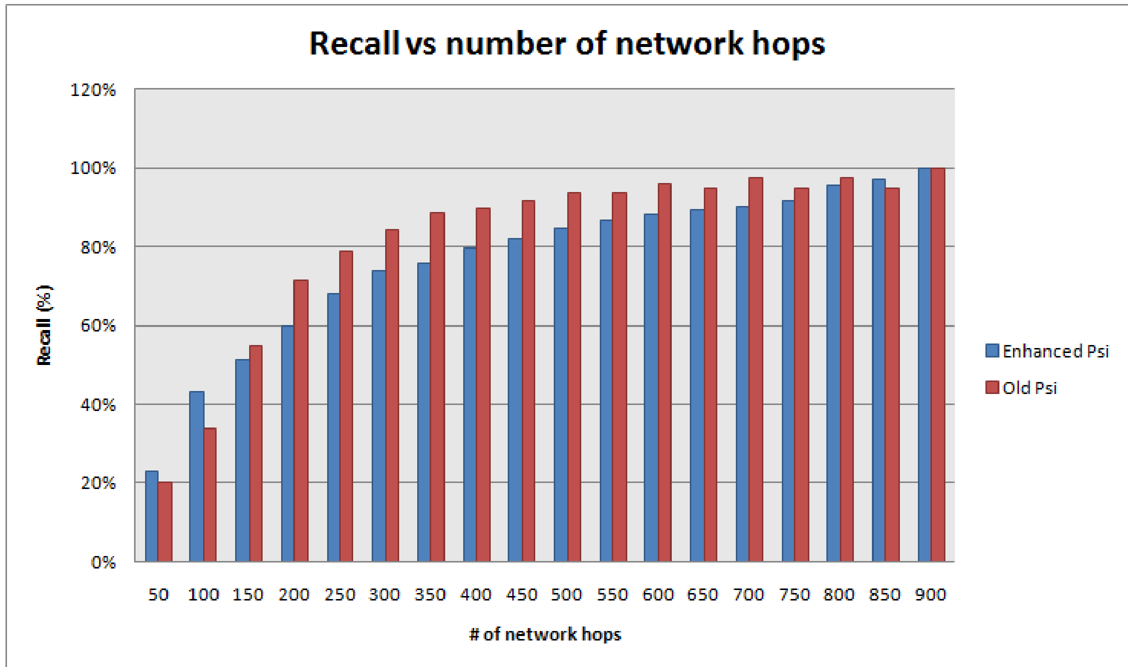


Figure 4.3: Recall vs. the number of network hops for different placement methods, employing the linear KNN query processing for the standard dataset when distributing 10 replicas of the dataset.

nature of LSH technique. Having five network hops inside a local DHT is reasonable to achieve $100\%$ recall accuracy, so from this point we always use $L = 10$ as the number of hash tables in our simulation.

Figures 4.4 and 4.5 show the recall versus the number of network hops for the mixed and synthetic datasets. It is interesting to see that for the mixed dataset, the number of hops required to achieve $100\%$ recall is drastically reduced in comparison to the standard dataset. This can be explained by having multiple versions of the same data points in the mixed dataset. Due to the fact that different versions of song are placed in the same bucket because of their small distance, the algorithm is more successful in finding similar data points to the query point in smaller hop counts.

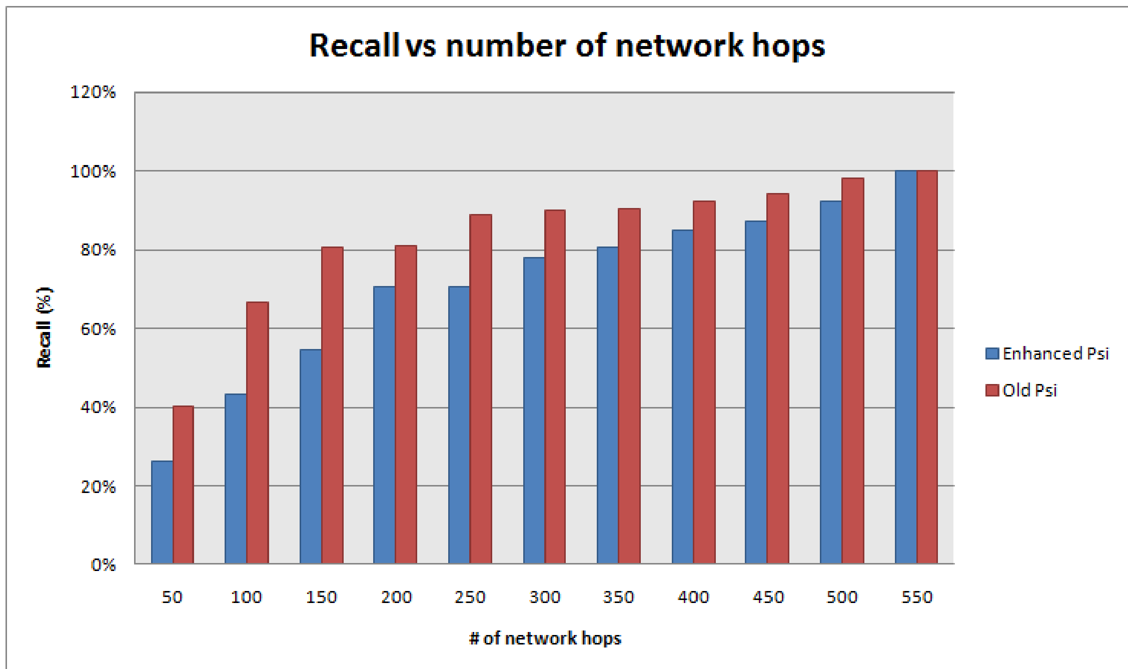


Figure 4.4: Recall vs. the number of network hops for different placement methods, employing the linear KNN query processing for the mixed dataset when distributing 10 replicas of the dataset.

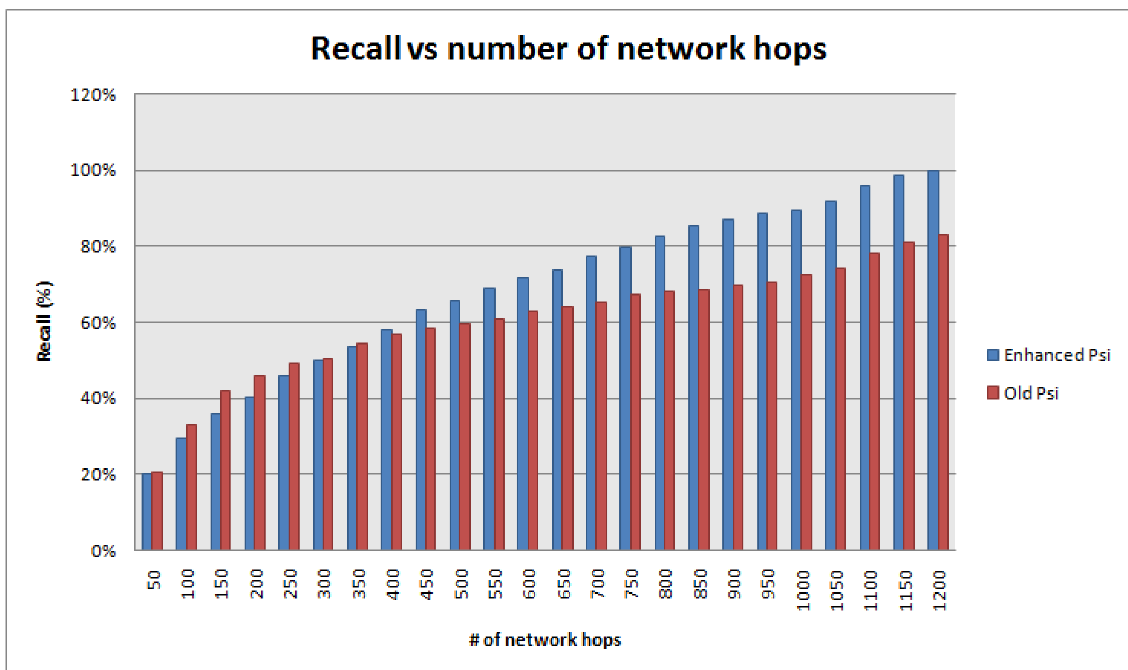


Figure 4.5: Recall vs. the number of network hops for different placement methods, employing the linear KNN query processing for the synthetic dataset when distributing 10 replicas of the dataset.

Figure 4.5 shows the recall versus the number of network hops for the synthetic dataset which follows the same distribution of data points in the standard dataset. Comparing the recall for the synthetic dataset with the recall for the other two, we can reach the following results:

1. Due to the fact that the synthetic dataset uses the standard dataset as its seed, if we have a larger dataset with equally distributed music files (in terms of genre), we reach the same recall performance with a slightly *different* number of hops.

2. Our proposed mapping function ($\psi_{enhanced}$) achieves higher recall when compared with $\psi_{old}$, when both are used for larger datasets.

3. Our proposed mapping function ($\psi_{enhanced}$) achieves almost the same recall when compared with $\psi_{old}$ when running on the mixed dataset, having a lower recall at the first few hops of a KNN query and reaching the same recall for the $\psi_{old}$ during the final hops.

4. The significant recall improvement in $\psi_{enhanced}$ using the synthetic dataset is being caused by the final steps of query forwarding. Our algorithm keeps the same pace because using the $\psi_{enhanced}$ the data load distribution on the peers is more even than $\psi_{old}$. So in the final steps $\psi_{old}$ reaches the peers with a very small number of data points which are probably irrelevant to the query point. This means that the query should have more network hops inside a local DHT to reach the peers maintaining the relevant data with a larger distance to the query point.

### 4.2.2 Range query results

We now report the results obtained for the Range queries. We run the Range query for the standard and mixed datasets. For each dataset the radius of the Range query is chosen such

that the number of possibly returned items is reasonable, i.e., for the standard dataset this number ranges from 20 to 890, while for the mixed dataset it ranges from 1 to 7,015. These numbers are found using a full scan on each dataset. Figure 4.6 illustrates a general view of the effect of varying the range on recall. As discussed in Section 3.8, recall in the linear method can decrease when the search radius of the Range query increases. This effect is fully illustrated in Figure 4.6 which shows the recall versus the search radius for both linear and sampling-based methods for the standard dataset using $\psi_{enhanced}$ mapping function as the placement method.

As what can be seen, as the search radius increases, the linear range recall drastically decreases in the standard dataset using $\psi_{enhanced}$. The results for the same dataset but using $\psi_{old}$ confirms the same issue and as we see, the difference between linear and sampling-based methods is higher when using $\psi_{old}$. Also $\psi_{old}$ shows a better performance in terms of recall using the sampling-based method and the worst performance using the linear method. For example, as the Figures 4.6 and 4.7 depict, for a search radius of $2.2$, $\psi_{old}$ has a recall of $98\%$ while $\psi_{enhanced}$ has $90\%$ recall. According to $\psi_{old}$ algorithm, with a higher probability it puts more relevant points in the same bucket, therefore when we choose a large enough radius, with a high probability we first visit the buckets that are filled with relevant data points and as a result we can retrieve more of them. On the other hand $\psi_{enhanced}$ more evenly distributes the point into buckets and this means the probability of facing the empty ranges increases. This problem can be solved by choosing more partitions when using the sampling-based method, in this way we decrease the occurrence probability of stops due to empty ranges.

We run the same tests using both placement methods for the mixed dataset and the results are illustrated in Figures 4.8 and 4.9. In both figures we see a huge difference between the overall recall of the linear Range query method. This can be again explained with the number of closely chosen data points as the main property of the mixed dataset.

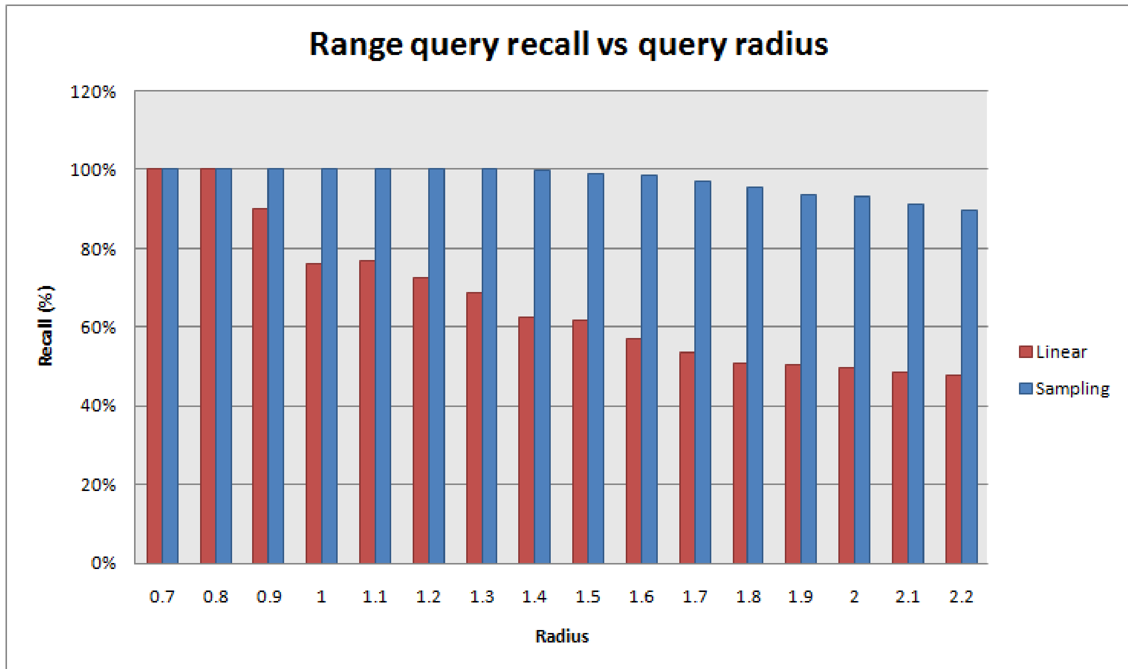


Figure 4.6: The effect of varying the range on recall. The results are shown for linear and sampling-based query processing methods using the standard dataset with 10 replicas and $\psi_{enhanced}$ placement method.

Due to the fact that with a high probability closer data points (songs) are placed on similar buckets, and we have multiple versions of the same songs (i.e., studio version, unplugged version and live version), this property reduces the effect of the empty ranges issue which is the primary obstacle in the linear Range query method.

As a result, in Figure 4.9, the recall for the linear Range query method closely follows the recall of the sampling-based method. The same behavior for both query methods repeats in the simulation with $\psi_{old}$ and it is depicted in Figure 4.9. Comparing the results for the two placement methods, again we see that both methods work better using $\psi_{old}$, but the difference is negligible as a tradeoff for load balancing which will be discussed in Section 4.2.3. We see around $9\%$ better recall performance in the linear method and $3\%$ improvement in the sampling method when using $\psi_{old}$. Referring to the idea of sampling-based Range query shown in Figure 3.9, it can be inferred that by increasing the number

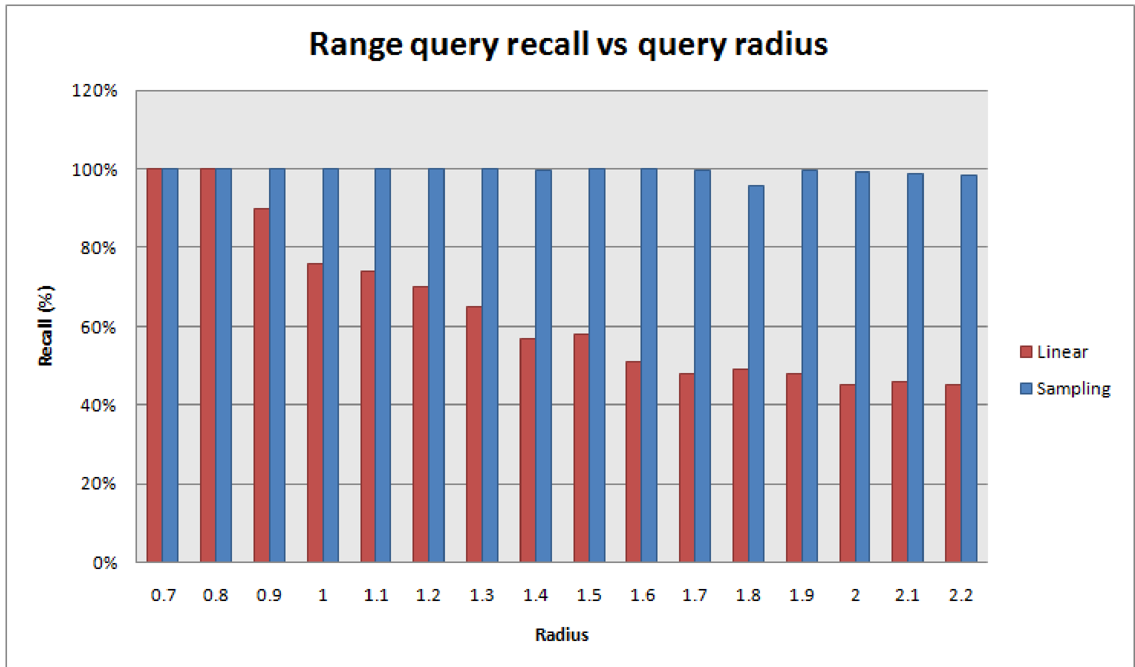


Figure 4.7: The effect of varying the range on recall. The results are shown for linear and sampling-based query processing methods using the standard dataset with 10 replicas and $\psi_{old}$ placement method.

of partitions $(s)$ we can avoid more peers that have empty ranges in between. Also the stopping condition of the linear search can be modified to use a relaxing parameter same as the linear KNN, in a way that when we visit a limited number of buckets that hold irrelevant data points we stop the forwarding. Using these two modifications, we may decrease the effect of the empty ranges on the recall at the cost of having more network hops.

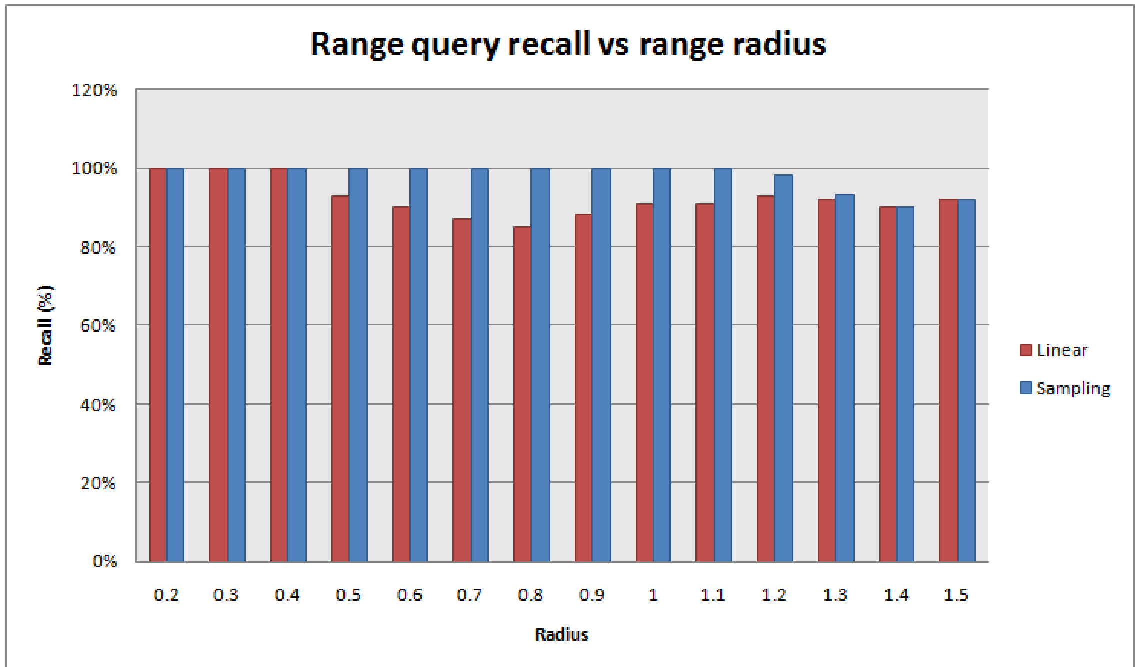


Figure 4.8: The effect of varying the range on recall. The results are shown for linear and sampling-based query processing methods using the mixed dataset with 10 replicas and $\psi_{enhanced}$ placement method.

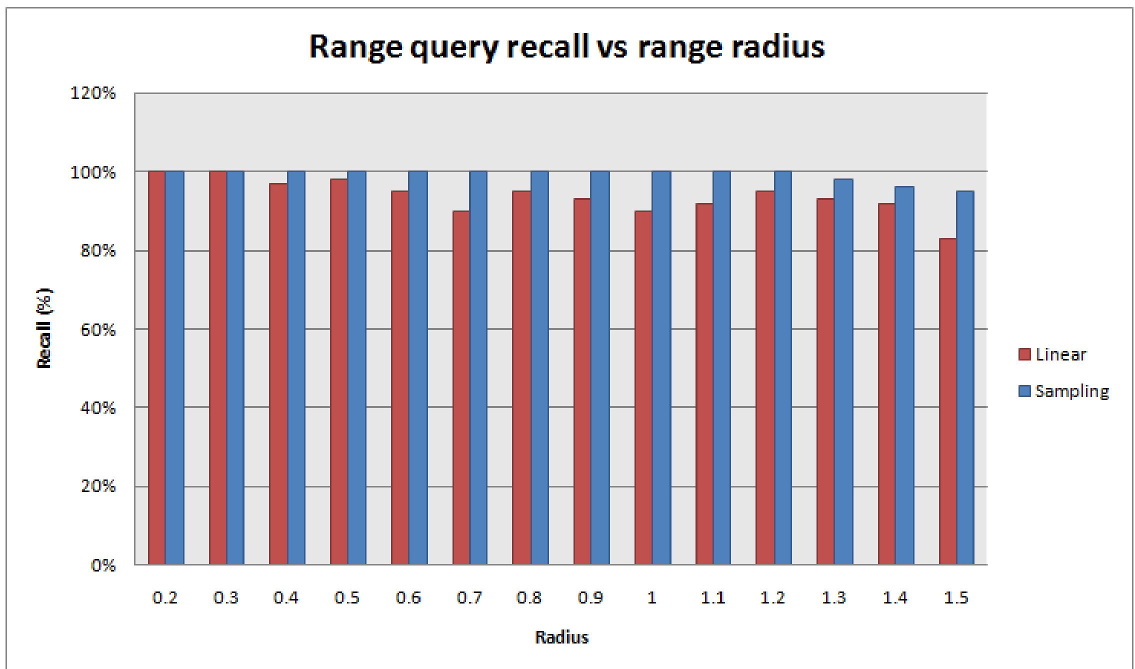


Figure 4.9: The effect of varying the range on recall. The results are shown for linear and sampling-based query processing methods using the mixed dataset with 10 replicas and $\psi_{old}$ placement method.

### 4.2.3 Load balancing results

Here we evaluate the most important aspect of our proposed mapping function $\psi_{enhanced}$, the load distribution. As we tested our proposed mapping function by varying the different parameters of LSH and overlay network, namely $k$ (the number of hash functions), $W$ and $M$ (the number of peers in each local DHT), we found that the most important parameter that affects the load balancing is $M$, the number of peers in the local DHTs and none of the others have a significant effect on load balancing.

As discussed at the beginning of this chapter in Section 4.1, using the Gini coefficient we can show the load distribution in each local DHT. As the load balancing should be evaluated using a large number of diverse data points, we use the synthetic dataset in order to measure this factor. Figure 4.10 shows the Gini coefficient for both placement methods versus $M$ while having 10 hash tables in a 100,000 peer global network. Beacuse we need to compare our results to the ones in [19], we set the $k = 20$ for this test. As Figure 4.10 shows, as the number of peers increases in local DHTs, the Gini coefficient increases.

Our results exactly match the results in [19] in their Section 7.2. The Flicker dataset in [19] is a dataset with $1,000,000$ data points, which is similar to our synthetic dataset. Also the $\xi_{sum}$ used in the implementation of our $\psi_{old}$ and $\psi_{enhanced}$ is the same function used in [19]. As we set our parameters similar to the ones in [19], using 100 peers in each local DHT, $W = 5$, $L = 2$ hash tables (replicas) and $1,000,000$ data points of the synthetic dataset, we get the Gini coefficient of $0.53$ in our simulations which exactly matches the results in Table $2$ of [19]. Table 4.2.3 summarizes our results for both mapping functions as well as the improvement in load balancing.

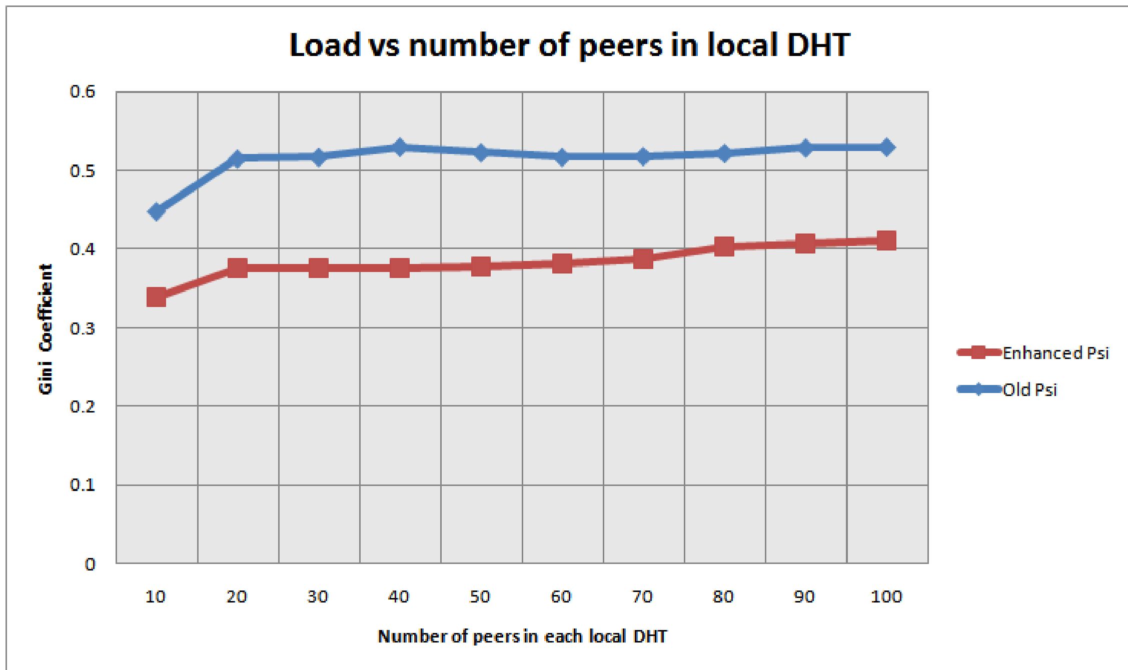


Figure 4.10: The effect of varying the number of peers inside each local DHT on load balancing factor (Gini coefficient) for $\psi_{enhanced}$ and $\psi_{old}$ placement methods.

| M | Gini coefficient for $\psi_{enhanced}$ | Gini coefficient for $\psi_{old}$ | Improvement % |
|---|---|---|---|
| 10 | 0.338 | 0.447 | 24% |
| 20 | 0.375 | 0.515 | 27% |
| 30 | 0.375 | 0.517 | 27% |
| 40 | 0.376 | 0.5301 | 29% |
| 50 | 0.378 | 0.523 | 28% |
| 60 | 0.381 | 0.517 | 26% |
| 70 | 0.387 | 0.518 | 25% |
| 80 | 0.402 | 0.522 | 23% |
| 90 | 0.407 | 0.529 | 23% |
| 100 | 0.41 | 0.53 | 23% |

Table 4.2: Gini coefficient for synthetic dataset when distributing 2 replicas of the data points

# Chapter 5

# Conclusions and future work

Distributed similarity search in high-dimensional feature spaces is one of the most important challenges in information retrieval. Locality Sensitive Hashing (LSH) techniques have been shown to have promising results in centralized and distributed setups in terms of the similarity search in high-dimensional spaces. In this work, we describe a scalable and load-balanced P2P system based on the design proposed in [19] that supports a rich set of high-dimensional similarity search techniques with a specific focus on music data.

As the main contribution of our research, we proposed a new load balanced indexing algorithm as an extension to the algorithm used in [19]. The proposed algorithms shows similar behaviors to the ones used in [19] and it also ensures an even distribution of the load on the peers in the network. If we use such a load-balanced system on a cluster of servers, we can guarantee a lower variation of processing and storage load on the servers. This means that each server has to handle a lower processing peak in order to serve in the cluster. As a consequence we can lower the costs of the server hardware and maintenance and this can help the company save more money.

We have also extended a highly scalable Java based simulator that is capable of simulating a Peer-to-Peer (P2P) system with an underlying Chord network that uses an indexing

technique based on LSH algorithm. Our simulator is capable of performing a linear K-Nearest-Neighbor (KNN) query processing and two types of Range query processing (i.e., a linear Range processing and a sampling-based Range processing) methods with configurable parameters.

In total three datasets have been used in order to evaluate the system performance on musical data. The first is a standard dataset created by George Tzanetakis [40]. We also created two musical datasets: first a mixed dataset that is based on real recordings, resembling an actual music collection, and a synthetic dataset which uses the standard dataset as its seed and is used to test the load balancing performance of the system when loaded with highly skewed data and a large number of data points.

The results from the comparison of our proposed mapping function ($\psi_{enhanced}$) and the mapping function proposed in [19] ($\psi_{old}$) can be summarized as follows:

1. Our $\psi_{enhanced}$ function shows a $25\%$ (on average) improvement on load balancing when compared with the $\psi_{old}$ function, considering the Gini coefficient as a measure of the load imbalance.

2. $\psi_{enhanced}$ shows similar behaviors to $\psi_{old}$ for KNN queries on the standard and mixed datasets and has a better performance on the synthetic dataset.

3. Sampling-based Range query processing method always shows a better performance when compared with the linear Range query processing on the standard dataset using both mapping functions.

4. Linear Range query processing closely follows the results from the sampling-based Range query processing when performed on the mixed dataset when using our $\psi_{enhanced}$ function and shows better performance in comparison to the $\psi_{old}$ function.

## 5.1 Further research issues

To have a complete and accurate distributed high-dimensional similarity search system, there are many related interesting issues that need further investigation. There are three main open issues that need to be addressed:

1. In our system we assumed that each bucket holds only one peer, but the load balancing can be further improved if multiple peers are allocated to maintain each bucket. The peers responsible for each bucket can form a local DHT, with each peer maintaining a portion of the data points that fall into the bucket.

2. Replication within particular local DHTs can be employed in order to handle the churn in the system: neighboring peers hold the index of their immediate neighbors and in case of peer failure the replicas are transmitted to the new peer joining the free spot. Also, the replication of the complete local DHTs can be used to further improve the query processing load in the system: This type of replication strategy is more straight-forward to implement and suitable for handling the concentrated query processing load problems (i.e., hot-spot peers).

3. Beside feature based item representation, Keyword-based Vector Space Model (VSM) can be adopted by the system to make the system compatible with keyword matching, and tag based searches as well.

4. The relevance of the returned items can also be improved by incorporating a weight strategy for the features to increase the weight of the features that are more important in the feature space.

Last but not the least, based on the above conclusions, our system can be employed in multipurpose machine learning libraries such as Apache Mahout [1] as the main item

indexing module, providing a load balanced search and access to multimedia content. Although many opportunities for further improvement still exist, we outlined a few key issues in this chapter, hoping to inspire further research interests in the this new area.

# Bibliography

[1] Apache. Mahout framwork: http://mahout.apache.org/.

[2] J.J. Aucouturier and Francois Pachet. Music Similarity Measures : Whats the Use ? In *Proceedings of the ISMIR*, pages 157–163. Citeseer, 2002.

[3] F. Banaei-Kashani and Cyrus Shahabi. SWAM: a family of access methods for similarity-search in peer-to-peer data networks. In *Proceedings of the thirteenth ACM international conference on Information and Knowledge Management*, pages 304–313. ACM, 2004.

[4] Mayank Bawa, Tyson Condie, and P. Ganesan. LSH forest: self-tuning indexes for similarity search. In *Proceedings of the 14th international conference on World Wide Web*, pages 651–660. ACM, 2005.

[5] Jon Louis Bentley. K-d Trees for Semidynamic Point Sets. In *Symposium on Computational Geometry*, pages 187–197, 1990.

[6] Stefan Berchtold, C. Böhm, and H.P. Kriegal. The pyramid-technique: towards breaking the curse of dimensionality. In *ACM SIGMOD Record*, volume 27, pages 142–153. ACM, 1998.

[7] A. Berenzweig, D.P.W. Ellis, and S. Lawrence. Anchor space for classification and similarity measurement of music. In *ICME*, pages 29–32. IEEE, 2003.

[8] Adam Berenzweig, Beth Logan, D.P.W. Ellis, and Brian Whitman. A large-scale evaluation of acoustic and subjective music-similarity measures. *Computer Music Journal*, 28(2):63–76, June 2004.

[9] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When Is Nearest Neighbor Meaningful? In Catriel Beeri and Peter Buneman, editors, *Database Theory ICDT99*, volume 1540 of *Lecture Notes in Computer Science*, pages 217–235. Springer Berlin / Heidelberg, 1999.

[10] T.L. Blum, D.F. Keislar, J.A. Wheaton, and E.H. Wold. Method and article of manufacture for content-based analysis, storage, retrieval, and segmentation of audio information. US Patent No. 5,918,223, June 1999.

[11] A. Crainiceanu, P. Linga, A. Machanavajjhala, J. Gehrke, and J. Shanmugasundaram. P-ring: an efficient and robust P2P range index structure. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of Data*, pages 223–234. ACM, 2007.

[12] Roger B Dannenberg, William P Birmingham, Bryan Pardo, Ning Hu, Colin Meek, and George Tzanetakis. A Comparative Evaluation of Search Techniques for Query-by-Humming Using the M USART Testbed Query Processing and Music. *Journal of the American Society for Information Science*, 58(3):1–19, 2007.

[13] M Datar, N Immorlica, P Indyk, and V.S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual Symposium on Computational Geometry*, pages 253–262. ACM, 2004.

[14] Christos Doulkeridis, Akrivi Vlachou, Yannis Kotidis, and Michalis Vazirgiannis. Peer-to-peer similarity search in metric spaces. In *Proceedings of the 33rd inter-*

*national conference on Very Large Databases*, pages 986–997. VLDB Endowment, 2007.

[15] D.P.W. Ellis, Brian Whitman, Adam Berenzweig, and Steve Lawrence. The quest for ground truth in musical artist similarity. In *Proc. ISMIR*, pages 170–177. Citeseer, 2002.

[16] Fabrizio Falchi, Claudio Gennaro, and Pavel Zezula. A contentaddressable network for similarity search in metric spaces. *Databases, Information Systems, and Peer-to-Peer Computing*, 6:98–110, 2007.

[17] J.T. Foote. Content-based retrieval of music and audio. In *Proceedings of SPIE*, volume 3229, page 138, 1997.

[18] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 518–529. Morgan Kaufmann Publishers Inc., 1999.

[19] Parisa Haghani, Sebastian Michel, and Karl Aberer. Distributed similarity search in high dimensions using locality sensitive hashing. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 744–755. ACM, 2009.

[20] Parisa Haghani, Sebastian Michel, P. Cudré-Mauroux, and Karl Aberer. LSH At Large-Distributed KNN Search in High Dimensions. In *International Workshop on Web and Databases (WebDB)*, number iii. Citeseer, 2008.

[21] P Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613. ACM, 1998.

[22] HV Jagadish, B.C. Ooi, Q.H. Vu, Rong Zhang, and Aoying Zhou. Vbi-tree: A peer-to-peer framework for supporting multi-dimensional indexing schemes. *In Proc. Intl. Conf. on Data Engineering (ICDE)*, 0:34–34, 2006.

[23] Márk Jelasity, Alberto Montresor, G.P. Jesi, and Spyros Voulgaris. The Peersim simulator, 2008.

[24] D.R. Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *Proceedings of the sixteenth annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 36–43, New York, NY, USA, 2004. ACM.

[25] B. Logan and A. Salomon. A music similarity function based on signal analysis. In *ICME 2001*, pages 745–748. Citeseer, 2001.

[26] Qin Lv, W. Josephson, Z. Wang, M. Charikar, and Kai Li. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very Large Databases*, pages 950–961. VLDB Endowment, 2007.

[27] Petar Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. *Peer-to-Peer Systems*, pages 53–65, 2002.

[28] M.F. McKinney and J. Breebaart. Features for audio and music classification. In *Proc. ISMIR*, volume 3, pages 151–158. Citeseer, 2003.

[29] D. Novàk. Similarity Search on a Very Large Scale. *is.muni.cz*, 2008.

[30] D Novak and P Zezula. M-Chord: a scalable distributed similarity search structure. *Proceedings of the 1st international conference on Scalable Information Systems*, 1:1–10, 2006.

[31] Rina Panigrahy. Entropy based nearest neighbor search in high dimensions. *Proceedings of the seventeenth annual ACM-SIAM Symposium on Discrete Algorithm - SODA '06*, pages 1186–1195, 2006.

[32] Theoni Pitoura, Nikos Ntarmos, and Peter Triantafillou. Replication, load balancing and efficient range query processing in DHTs. *Advances in Database Technology-EDBT 2006*, pages 131–148, 2006.

[33] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. Load Balancing in Structured P2P Systems. In M Kaashoek and Ion Stoica, editors, *Peer-to-Peer Systems II*, volume 2735 of *Lecture Notes in Computer Science*, pages 68–79. Springer Berlin / Heidelberg, 2003.

[34] S Ratnasamy, P Francis, M Handley, R Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 161–172. ACM, 2001.

[35] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[36] O.D. Sahin, F. Emekçi, Divyakant Agrawal, and A.E. Abbadi. Content-based similarity search over peer-to-peer systems. *Databases, Information Systems, and Peer-to-Peer Computing*, pages 61–78, 2005.

[37] Emil Sit and Robert Morris. Security Considerations for Peer-to-Peer Distributed Hash Tables. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Peer-*

*to-Peer Systems*, volume 2429 of *Lecture Notes in Computer Science*, pages 261–269. Springer Berlin / Heidelberg, 2002.

[38] I. Stoica, R. Morris, D. Liben-Nowell, D.R. Karger, M.F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, February 2003.

[39] Chunqiang Tang, Zhichen Xu, and Sandhya Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *Proceedings of the 2003 conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 175–186, New York, New York, USA, 2003. ACM.

[40] G. Tzanetakis and P. Cook. Musical genre classification of audio signals. *IEEE Transactions on Speech and Audio Processing*, 10(5):293–302, July 2002.

[41] George Tzanetakis. MARSYAS submissions to MIREX 2009. *Proceedings of the Music Information Retrieval Evaluation EXchange*, 2009.

[42] C. Yu, B.C. Ooi, K.L. Tan, and HV Jagadish. Indexing the distance: An efficient method to knn processing. In *Proc. VLDB*, pages 421–430, 2001.

[43] C. Zhang, A. Krishnamurthy, and R.Y. Wang. Skipindex: Towards a scalable peer-to-peer index service for high dimensional data. *Department of Computer Science, Princeton University, New Jersey, USA, Tech. Rep*, pages 703–04, 2004.

# Appendix A

# LSH Parameters

Here we compute the probability that two input vectors collide under a hash function drawn uniformly at random from the family of Locality Sensitive Hash functions, according to [13].

Let $f_p(t)$ denote the probability density function of the **absolute value** of the $p$-stable distribution. For the two vectors $v_1, v_2$, let $c = \|v_1 - v_2\|_p$. For a random vector **a** whose elements are drawn from a $p$-stable distribution, $a \cdot v_1 - a \cdot v_2$ is distributed as $cX$ where $X$ is a random variable drawn from a $p$-stable distribution. Since $B$ is drawn uniformly from [0,W] it can be seen that:

$$p(c) = \Pr[h_{a,B}(v_1) - h_{a,B}(v_2)] - \int_o^W \frac{1}{c} f_p(\frac{t}{c})(1 - \frac{t}{r})\mathrm{d}t \tag{A.1}$$

For a fixed parameter $W$, the probability of collision decreases monotonically with $c = \|v_1 - v_2\|_p$. Thus, as per the hash function of (2.1) the family of hash functions is $(r_1, r_2, p_1, p_2)$-sensitive for $p_1 = p(1)$ and $p_2 = p(c)$ for $r_2/r_1 = c$.

As for the space and time complexity of LSH, according to [13] we have the following:

**Theorem 1.** *Suppose there is a $(R, cR, p_1, p_2)$-sensitive family $H$ for a distance measure D. Then there exists an algorithm for (R,c)-NN under measure $D$ which uses $O(dn + n^{1+\rho})$*

*space, with query time dominated by $O(n^\rho)$ distance computations, and $O(n^\rho \log_{\frac{1}{p_2}}(n))$ evaluations of hash functions from $H$, where $\rho = \frac{\ln(1/p_1)}{\ln(1/p_2)}$*

Here we will bound the ratio $\rho = \frac{\ln 1/p_1}{\ln 1/p_2}$, which is critical to the performance when this hash family is used to perform KNN queries. For special cases $p = 1$ and $p = 2$ we can compute the probabilities $p_1$ and $p_2$, using the density functions. We can find the optimal values for $p_1$ and $p_2$ for different cases of $p$-stable hash functions:

- For $p = 1$ (Cauchy): $p_2 = 2\frac{\tan^{-1}(W/c)}{\pi} - \frac{1}{\pi(W/c)} \ln(1 + (W/c)^2)$
- For $p = 2$ (Gaussian): $p_2 = 1 - 2norm(-W/c) - \frac{2}{\sqrt{2\pi}/c}(1 - e^{-(W^2/2c^2)})$

where $norm()$ is the cumulative distribution function (CDF) for a random variable that is distributed as $N(0, 1)$. The value of $p_1$ can be calculated by substituting $c = 1$ in the formulas above. Further details about LSH parameter tuning can be found in [13].

# Glossary

**collaborative filtering** The process of filtering for information or patterns using techniques involving collaboration among multiple agents, viewpoints, data sources, etc.. 2

**collision** A situation that occurs when two distinct pieces of data have the same hash value. 14

**hot spots** Peers in a network that are frequently contacted in order to resolve or route a query. 3

**logical ring** a network topolgy in which the nodes only know a predecessor and a successor neighbor, forming a ring. 18

**Lorens curve** A graphical representation of the cumulative distribution function of the empirical probability distribution of wealth. 46

**peer identifier space** A numerical space from which the peer IDs of peers in a P2P network are selected. 17

**precision** The fraction of retrieved instaces that are relevant. 48

**recall** The fraction of relevant instaces that are retrieved. 47

**Semantic Web** A ”Web of data” that facilitates machines to understand the semantics, or meaning, of information on the World Wide Web. 1

**string-matching algorithms** An important class of string algorithms that try to find a place where one or several strings (also called patterns) are found within a larger string or text. 2