ΚΑΤΑΣΚΕΥΗ ΔΙΚΤΥΩΝ ΟΜΟΤΙΜΩΝ ΒΑΣΙΣΜΕΝΩΝ ΣΤΟ ΦΟΡΤΙΟ ΕΡΩΤΗΣΕΩΝ ΜΕ ΧΡΗΣΗ ΙΣΤΟΓΡΑΜΜΑΤΩΝ

Η
ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

Υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης
του Τμήματος Πληροφορικής
Εξεταστική Επιτροπή

από τον

Θεόδωρο Τσώτσο

ως μέρος των υποχρεώσεων

για τη λήψη

του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ ΣΤΟ ΛΟΓΙΣΜΙΚΟ

Ιούλιος 2006

ACKNOWLEDGEMENTS

I would firstly like to thank my supervisor Professor Evaggelia Pitoura for her help, support, the time spent during the elaboration of this thesis, and especially, the patience she has shown until this thesis is completed. I would also like to thank Georgia Koloniari and Yannis Petrakis for their useful advices on many issues of this work. Finally, I would like to thank my friends Kostas Stefanidis, Nikoleta Tzimpouka, Jenny Stathopoulou and Maria G. Chroni for their moral support all this time.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ΠΕΡΙΛΗΨΗ

Τσώτσος Θεόδωρος του Αθανασίου και της Αντωνίας.
MSc, Τμήμα Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Ιούλιος, 2006. Κατασκευή Δικτύων Ομοτίμων Βασισμένων στο Φορτίο Ερωτήσεων με Χρήση Ιστογραμμάτων. Επιβλέπουσα: Ευαγγελία Πιτουρά.

Τα συστήματα ομοτίμων παρέχουν έναν αποδοτικό τρόπο διαμοιρασμού δεδομένων μεταξύ ενός δυναμικού συνόλου από ανεξάρτητους κόμβους. Κάθε κόμβος σε ένα σύστημα ομοτίμων συνδέεται με ένα μικρό πλήθος άλλων κόμβων, δημιουργώντας με αυτόν τον τρόπο ένα λογικό δίκτυο επικάλυψης. Μία ερώτηση για δεδομένα μπορεί να τεθεί σε οποιοδήποτε από τους κόμβους και να δρομολογηθεί μέσω του δικτύου στους κόμβους που παρέχουν τα επιθυμητά δεδομένα. Σε αυτήν την εργασία, επικεντρωνόμαστε σε ερωτήσεις εύρους και χρησιμοποιούμε την ιδέα της κατασκευής λογικών δικτύων επικάλυψης βασιζόμενα στο φορτίο ερωτήσεων. Ο γενικός στόχος είναι να κατασκευάσουμε λογικά δίκτυα επικάλυψης που ικανοποιούν συγκεκριμένες απαιτήσεις αναφορικά με το πλήθος των συσχετιζόμενων αποτελεσμάτων των ερωτήσεων. Το κίνητρο για την δημιουργία τέτοιου είδους δικτύων είναι ότι οι κόμβοι που παρέχουν μεγάλος πλήθος αποτελεσμάτων για παρόμοιες ερωτήσεις θα πρέπει να βρίσκονται σε μικρή «απόσταση» μεταξύ τους. Η πιθανότητα εμφάνισης των ερωτήσεων λαμβάνεται υπόψιν έτσι ώστε οι δημοφιλείς ερωτήσεις να συμβάλλουν περισσότερο στην διαμόρφωση του λογικού δικτύου επικάλυψης από ότι οι λιγότερο δημοφιλείς. Η προσέγγιση μας βασίζεται στον καθορισμό «μέτρων ομοιότητας» μεταξύ των κόμβων προκειμένου να γίνει ομαδοποίηση ομοίων κόμβων.

Σε αυτήν την εργασία, εισάγουμε τρεις μετρικές υπολογισμού της ομοιότητας δύο κόμβων που λαμβάνουν υπόψιν τους το φορτίο ερωτήσεων και διαφέρουν ως προς τον τρόπο που το «μέγεθος» των κόμβων, δηλαδή το πλήθος των πλειάδων ενός κόμβου, συμβάλλει στον καθορισμό της ομοιότητας μεταξύ τους. Στην συνέχεια, αποτιμούμε την επίδοση των μετρικών αυτών με κριτήριο τον τρόπο ομαδοποίησης των κόμβων του συστήματος που επιτυγχάνουν και τις συγκρίνουμε με διάφορες άλλες δημοφιλείς μετρικές που χρησιμοποιούν αποκλειστικά τα περιεχόμενα δύο κόμβων για τον καθορισμό της ομοιότητας τους. Το κύριο

συμπέρασμα που προέκυψε είναι ότι στην γενική περίπτωση οι μετρικές που λαμβάνουν υπόψιν το φορτίο ερωτήσεων συμβάλλουν στην αποδοτικότερη ομαδοποίηση των κόμβων, που οδηγεί στην επιστροφή περισσότερων αποτελεσμάτων για μία ερώτηση, σε σύγκριση με την ομαδοποίηση που παρέχουν οι μετρικές που χρησιμοποιούν μόνο τα περιεχόμενα των κόμβων.

Προκειμένου να υλοποιηθεί αποδοτικά μία τέτοιου είδους ομαδοποίηση κόμβων σε ένα δίκτυο ομοτίμων, απαιτείται η χρησιμοποίηση δομών τέτοιων που να συνοψίζουν με ακρίβεια τόσο τα περιεχόμενα των κόμβων όσο και το φορτίο ερωτήσεων. Επιλέγουμε να χρησιμοποιήσουμε ιστογράμματα για να περιγράψουμε τόσο τα περιεχόμενα τους όσο και το τοπικό φορτίο ερωτήσεων τους. Ειδικότερα, προτείνουμε ένα είδος ιστογράμματος, το *maxdiff(v, f)* ιστογράμμα για να συνοψίσουμε τα δεδομένα των κόμβων και το συγκρίνουμε με το δημοφιλές *equi-width* ιστογράμμα. Επιπλέον, εισάγουμε ένα νέο τύπο ιστογράμματος, το *W-ST* ιστόγραμμα, για να αναπαραστήσουμε το τοπικό φορτίο ερωτήσεων κάθε κόμβου και συγκρίνουμε την απόδοση του με το *W-Equi-width* ιστόγραμμα, το οποίο κατασκευάζεται χρησιμοποιώντας τον equi-width ευριστικό αλγόριθμο. Στην συνέχεια, προτείνουμε έναν αλγόριθμο για την συγχώνευση δύο ιστογραμμάτων. Ο συγκεκριμένος αλγόριθμος χρησιμοποιείται προκειμένου να αποκτήσουμε μία εκτίμηση του φορτίου ερωτήσεων του συστήματος από τις εκτιμήσεις των τοπικών φορτίων ερωτήσεων των κόμβων. Επιπρόσθετα, εισάγουμε μία μετρική υπολογισμού της ομοιότητας δύο κόμβων χρησιμοποιώντας ιστογραμμάτα που λαμβάνει υπόψιν αφ' ενός τις συνόψεις των δεδομένων των κόμβων αφ' ετέρου την εκτίμηση του φορτίου ερωτήσεων του συστήματος. Αποδείχθηκε ότι χρησιμοποιώντας αυτή την μετρική με βάση τα ιστογράμματα επιτυγχάνουμε μία πάρα πολύ καλή προσέγγιση των μετρικών που λαμβάνουν υπόψιν τους το φορτίο ερωτήσεων.

Επιπρόσθετα, χρησιμοποιούμε ιστογράμματα και για την δρομολόγηση των ερωτήσεων. Ειδικότερα, εκτός από τα ιστογράμματα που χρησιμοποιούνται για να αναπαρασταθούν τόσο τα περιεχόμενα του κάθε κόμβου όσο και το τοπικό του φορτίο ερωτήσεων, κάθε κόμβος έχει επιπλέον και ένα ευρετήριο δρομολόγησης για κάθε μία από τις ακμές του που περιγράφει το περιεχόμενο των κόμβων που μπορούμε να επισκεφτούμε μέσω αυτής της ακμής. Για την κατασκευή των ευρετηρίων δρομολόγησης, χρησιμοποιούμε επίσης τον αλγόριθμο συγχώνευσης ιστογραμμάτων. Τέλος, αποτιμούμε πειραματικά το δίκτυο που κατασκευάζεται χρησιμοποιώντας την μετρική υπολογισμού της ομοιότητας δύο κόμβων με βάση τα ιστογράμματα και το φορτίο ερωτήσεων του συστήματος και το συγκρίνουμε με τα δίκτυα που κατασκευάζονται χρησιμοποιώντας τις μετρικές ομοιότητας που λαμβάνουν υπόψιν μόνο τα δεδομένα των κόμβων. Η αποτίμηση των δικτύων γίνεται με βάση τα αποτελέσματα που

επιστρέφονται ανά ερώτηση. Τα πειραματικά αποτελέσματα δείχνουν ότι τα δίκτυα ομοτίμων που κατασκευάζονται χρησιμοποιώντας το φορτίο ερωτήσεων αυξάνουν το ποσοστό των επιστρεφόμενων αποτελεσμάτων για κάθε ερώτηση, όταν κάθε μία από αυτές επισκέπτεται ένα συγκεκριμένο πλήθος κόμβων.

ABSTRACT

Tsotsos, Theodoros.
MSc, Computer Science Department, University of Ioannina, Greece. July, 2006. Query-Driven Overlay Network Construction Using Histograms. Thesis Supervisor: Evaggelia Pitoura.

Peer-to-peer (p2p) systems offer an efficient means of data sharing among a dynamic set of a large number of autonomous nodes. Each peer in a p2p system is connected with a small number of other peers, thus forming an overlay network. A query for data items posed at a peer is routed through the overlay network to peers that host the requested items. In this thesis, we focus on range selection queries. Our overall objective is an overlay network construction that satisfies specific requirements regarding the result sizes of queries. The motivation is to to create overlays where peers that match a large number of similar queries are nearby. Query probability is taken into account so that popular queries have a greater effect on the formation of the overlay than unpopular ones. Our approach is based on defining appropriate workload-aware similarity measures and then creating cluster of similar peers based on these measures.

In this thesis, we consider three types of workload-aware distance measures that differ on how they take into account the size of the peers. We assess their efficiency on clustering the peers of the system in comparison with several well known content-based distance measures. Our evaluation shows that in the general case the workload-aware distance measures result in creating cluster of peers that return more results than those created using the content-based ones.

To implement clustering efficiently in a p2p system, we need estimations of both the content of the peers and the query workload. We use histograms to summarize the content of the peers and their local query workloads. In particular, we propose using the *maxdiff(v, f)* histogram for summarizing the content of each peer and compare it with the traditional *equi-width*

histogram. Furthermore, we introduce a new type of histogram, the *W-ST* histogram, for summarizing the local query workload of each peer and evaluate its accuracy in comparison with the *W-Equi-width*, which uses an equi-width based heuristic for its construction. Next, we propose a general algorithm for merging two histograms. This algorithm is used to obtain the estimation of the system query workload by merging the local estimations of the peers query workloads. In addition, we introduce a workload-aware histogram distance metric that takes into account the estimation of the system query workload and the summaries of the peers content. We show that by using the workload-aware histogram distance measure we get a good approximation of the workload-aware distances. Then, peers are clustered based on this workload-aware histogram distance measure.

We also use histograms for routing queries. Specifically, besides the histograms that represent the content of the peers and their local query workloads, each peer maintains a routing index for each of its links that summarizes the content of the set of peers that we can visit by following this link. To build routing indexes, we also use the procedure of merging histograms.

Finally, we present a simulation of the networks that are constructed using the workload-aware histogram distance measure and the content-based histogram distance measures. The evaluation of these networks is done in term of the number of results returned per query. Our results show that workload-aware overlays created using the corresponding histogram-based distance measure increase the percentage of query results returned for a given number of peers visited.

CHAPTER 1. INTRODUCTION

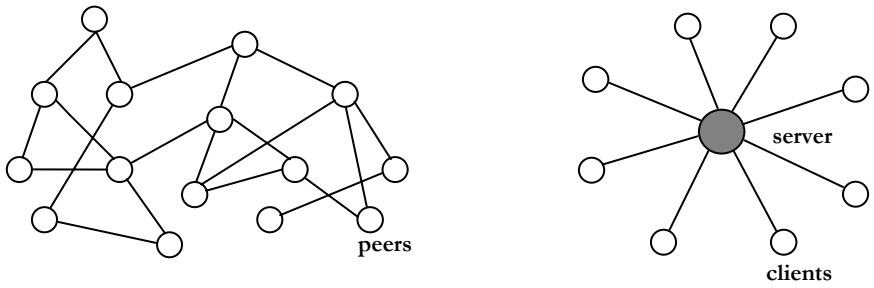
1.1. Peer-to-Peer Preliminaries

The popularity of file sharing systems such as Napster [2] and Gnutella [1] has resulted in attracting much current research in peer-to-peer (p2p) architectures as an efficient means of sharing data. In peer-to-peer computing, a large dynamic set of autonomous computing nodes (the peers) cooperate to share resources and services. The peers form logical overlay networks by establishing links to some other peers they know or discover.

There are several definitions of "peer-to-peer" that are being used by the p2p community. A widely accepted definition is the following [4]:
"Peer-to-peer systems are distributed systems consisting of interconnected nodes able to self-organize into network topologies with the purpose of sharing resources such as content, CPU cycles, storage and bandwith, capable of adapting to failures and accommodating transient population of nodes while maintaining acceptable connectivity and performance, without requiring the intermedation or support of a global centralized server or authority."

Specifically, the term "peer-to-peer" refers to a class of systems and applications that employ distributed resources to perform a critical function-application in a decentralized manner [26]. Although the best-known application of p2p systems is file sharing (for example, music files in Napster), p2p system applications go beyond data sharing. Peer-to-peer computing is also a way of implementing systems based on the notion of increasing the decentralization of systems. In particular, by leveraging vast amounts of computing power, storage and connectivity from personal computers distributed around the world, p2p systems provide a

substrate for a variety of applications such as network monitoring and routing, web search and large scale notification systems.



(a) (b)

Figure 1.1: High-Level View of (a) Peer-to-Peer versus (b) Centralized (Client-Server) Approach.

The motivation behind building applications based on p2p architectures derives to a large extent from their ability to function, scale and self-organize in the presence of a highly transient population of nodes, network and computer failures, without the need of a central server and the overhead of its administration. Some additional benefits of peer-to-peer systems include: load-balancing availability through massive replication, and the ability to pool together and harness large amounts of resources. For example, file-sharing peer-to-peer systems distribute the main cost of sharing data – bandwith and storage – across all the peers in the network, thereby allowing them to scale without the need for powerful, expensive servers.

Conceptually, peer-to-peer systems are an alternative to the centralized and traditional client-server models of computing, where there is typically a single or small cluster of servers and many clients. In its purest form, the peer-to-peer model has no concept of server; rather all participants are peers. In Figure 1.1, we demonstrate in a high level a peer-to-peer system and a centralized (client-server) approach.

The peers in a peer-to-peer system are autonomous, i.e., they are not wholly controlled by each other or by the same authority, e.g., the same user. Peers depend on each other for getting information, computing resources, forwarding requests, etc, which are essential for the functioning of the system as a whole and for the benefit of all peers. As a result of the autonomy of peers, they cannot necessarily trust each other and rely completely on the

behaviour of other peers, so issues of scale and redundancy become much more important than in traditional centralized or distributed systems.

Peer-to-peer is about sharing: giving to and obtaining from the peer community. A peer gives some resources and obtains other resources in return. For example, in the case of Napster, it was about offering music to the rest of the community and getting other music in return. An increasing number of users choose peer-to-peer systems as a means to efficiently share their resources, data and services, and exploit the resources that other participating users provide. Therefore, a central issue is discovering the appropriate data and services among the available huge, massively distributed data collections. Thus, the resource discovery mechanism that is responsible for finding the appropriate data a user desires is a challenging issue for peer-to-peer systems. Users issue queries that describe the resources they are interested in, and expect from the system to provide accurate, timely results. A single query on a peer may need results from a large number of others, hence we need a mechanism that finds the peers that contain matching to the query data efficiently and is able to scale up as more users join the system and large volumes of data become available.

1.2. Scope of the Thesis

The goal of this thesis is to design a peer-to-peer database system for resource discovery. We consider that we have a peer-to-peer system where each peer is connected with a small number of other peers, thus forming an overlay network. Each peer stores a database with the same schema. A query for data may be issued at any peer. The query is routed through the overlay network from the issuing peer to peers that have data (tuples) satisfying it (stored at various peers); we call such peers *matching peers*. We are interested in increasing the number of query results returned by propagating the query message to peers that have large number of results that satisfy this query.

One way to attain a large number of matching results for a query is by clustering peers based on their content so that peers with similar content are nearby in the overlay network. Then, once in the appropriate cluster, all relevant peers are nearby. In our previous work [22], we have proposed forming clusters based on the *query* workload. In particular, the formation of clusters relies on two basic factors, (i) *peers content*, where two peers with similar content will participate in the same cluster, and (ii) *query workload*, so that the type and probability of queries is taken into account in creating the clusters. Hence, “popular” queries affect the formation of peer groups more than unpopular ones. The motivation for taking into account

the query workload is that if some data items are queried only seldomly, we do not want them to influence clustering as much as other data items.

In particular, we have considered workload-aware overlays consisting of a number of smaller networks (clusters) of relevant peers that are rich in links between their peers (short-range links), while they are linked to each other with a few random connections (long-range links). The relevance of two peers was based on their query results to the system workload, denoted as *global query workload*. The reason for constructing such "workload-aware" overlay network is that once in the appropriate group, the most relevant to a query peers are a few short-range links apart. Long-range links are used for routing among peers.

The construction of the overlay network exploits local indexes. In particular, each peer stores a local index, which is a summarization of the local content of a peer. Besides the local index, each peer maintains one routing index for each of its links. The routing indexes are created by aggregating local indexes of neighboring peers and are used to form the cluster of peers, i.e., to navigate a new peer that joins the system to the right cluster, and to route a query message to those peers that provide enough matching results. We used *equi-width* histograms as local indexes. The main advantages of histograms over other techniques are that they incur almost no run-time overhead and, for most real-world databases, there exist histograms that produce low-error estimates while occupying reasonably small space.

To demonstrate the feasibility of workload-aware clustering we consider range queries over one attribute of the database. For instance, the attribute of the query may correspond to a single characteristic of music files (such as release-year, artist-name, etc.) in a music file-sharing system or the available resources (such as CPU, memory) in a resource sharing system.

In this work, we extend our previous work as follows: First, we introduce three workload-aware distance measures that differ on how they take into account the size of the peers. We compare them with several well known content-based distance measures from the perspective of how efficient is the clustering that they achieve. The experimental evaluation shows that in the general case the clustering that the workload-aware distance measures provide is more efficient than those created using the content-based ones, since the matching results returned for a query are more than those when we use a content-based distance measure for clustering.

In addition, we propose more sophisticated histograms to use as local indexes. In particular, we use the *maxdiff(v, f)* histogram to summarize the content of the peers and compare it with the traditional equi-width histogram. As expected, the experimental evaluation shows that the maxdiff(v, f) histogram is much more accurate than the equi-width histogram since it produces much lower estimation error in approximating the distribution of data. Furthermore, we propose a novel procedure for merging two histograms of unequal sizes that we use it to create the routing indexes of each peer.

Besides the local index and the routing indexes, each peer maintains a *local query workload synopsis* which summarizes the queries that arrive at a peer. To represent the local query workload of each peer efficiently, we propose using also histograms. We introduce a new histogram type, the *workload self-tuning* (*W-ST*) histogram. We compare the W-ST histogram with the *W-Equi-width* histogram, which uses an equi-width based heuristic for its construction. The experimental results show that the W-ST histogram is much more efficient in comparison with the W-Equi-width histogram and captures accurately the distribution of a query workload. We also show that W-ST histograms can adapt to changes in the workload fast.

In addition, we present an approach for building workload-aware overlays based on a decentralized procedure that exploits local indexes and query workload synopses. To attain workload-awareness, we define a weighted histogram distance measure between two peers that takes into account their local indexes and is weighted by using the synopsis of the global query workload, which is acquired by the local query workload synopses, so as to achieve the peers with similar results for the query workload to be a few links away in the overlay network. The estimation of the global query workload can be obtained by merging the peers local query workload synopses using the general algorithm for merging two histograms that we have developed. We also show that this workload-aware histogram distance measure is an accurate approximation of the workload-aware distance measures.

In summary, in this thesis we make the following contributions:

- We propose three workload-aware distance measures for the formation of the overlays, where the grouping of similar peers is not based solely on the content of the peers but also on the query workload, and compare the quality of clustering that they achieve with the quality of clustering that is achieved when using several well-known content based distance measures.

- We exploit the use of histograms as local indexes and as local query workload synopses for constructing workload-aware overlays. In particular, we propose using the maxdiff(v, f) histogram for summarizing the peers content. Furthermore, we introduce a new type of histogram, the W-ST histogram, for summarizing the query workload information, which is adaptive to changes of the query workload.
- We propose a general algorithm for merging two histograms. This algorithm is used for creating routing indexes by merging local indexes and for obtaining an estimation of the system query workload by merging the peers local query workload synopses.
- We introduce an appropriate workload-aware histogram distance metric that incorporates the probability and type of queries.

1.3. Thesis Outline

The remainder of this thesis is structured as follows. In Chapter 2, we present related work. In Chapter 3, we motivate the need for workload-aware overlays. We propose three workload-aware similarity measures and experimentally compare them with several well known content-based similarity measures. Chapter 4 focuses on histograms as local indexes. We consider two popular type of histograms, equi-width and maxdiff(v, f) histograms and evaluate their suitability for serving as local indexes in a p2p setting. We also present a procedure for merging histograms of different sizes. In Chapter 5, we deal with the problem of selecting an efficient histogram for representing the query workload. We introduce a new type of histogram, the Workload Self-Tuning (W-ST) histogram that can represent the query workload accurately and adapts to changes of the query workload. In Chapter 6, we propose a novel workload-aware histogram distance measure and present experiments comparing it with other content-based histogram distance measures. In Chapter 7, we describe the protocols that we use for the construction of the network, the routing of a query, and the creation and maintainance of the routing indexes. In Chapter 8, an experimental evaluation is presented showing the performance of the workload-aware network in comparison with other networks including those that are constructed using content-based similarity measures. Finally, in Chapter 9, we offer conclusions and directions for future work.

CHAPTER 2. RELATED WORK

We propose an approach for building a peer-to-peer overlay network based on both the content of the peers and the query workload using *routing indexes* and *query workload synopsis*, respectively. Routing indexes are stored for each link of a peer and summarize the content of a number of peers that are reached through this link. In addition, query workload synopsis summarizes the query workload, i.e., the queries that are posed by all the users that participate in the p2p network. We focus on "range" queries and use histograms to summarize the content of peers and the query workload. In this chapter, related work is presented. Consequently, we distinguish the research related to our work into two areas. First, we present the research that was done on resource discovery in p2p systems and then we provide several techniques for creating histograms that can efficiently summarize the content of relations over one or more than one attributes.

2.1. Resource Discovery in P2P Systems

Peer-to-peer systems have become one of the fastest growing applications in recent years since they offer a lot of potentials at the domain of sharing resources among dynamic sets of users while providing autonomy, robustness in failures, self-organization, load balancing, privacy, etc. In p2p computing, distributed nodes (peers) across the Internet form an overlay network and exchange information directly with each other.

The first widely used system of this kind of distributed computing was Napster [2]. Napster relied on a centralized architecture, where a central server, e.g., Napster's web site, stored the index of all the files available from the nodes that formed the overlay network. In order to find and download a file, a node had to issue a query to the central Napster site and find which other nodes stored the requested file. The file was then downloaded directly from one of these nodes. The main disadvantage of Napster was that the file location method used was "centralized", thus the system was not scalable. In addition, due to the centralized nature of Napster, legal issues forced it to shut down. To avoid this kind of problems, the research community turned to unstructured architectures for peer-to-peer systems. Gnutella [1] is one of the most representative systems that uses unstructured content location and relies on flooding to answer nodes queries.

In general, peer-to-peer systems can be classified based on their architecture. The centralized p2p systems, like Napster, rely on indexing all the shared data of all nodes in a central directory. Queries are issued to this central directory to find the nodes that have the desired files. In decentralized p2p systems, the information is shared among the nodes of the network without having any centralized structure, such as a central directory. We can further distinguish the decentralized p2p systems in structured and unstructured ones. In structured p2p systems, data items are not placed at random nodes but at specific nodes determined by using distributed hashing (DHT). In more details, each data item is assigned a key, by using DHT, and each node is assigned a range of keys. Thus a data item with an associative key will be placed at the node that includes this key in its range. In this kind of systems search is very efficient, requiring a small number of hops to answer a query. CHORD [37] and CAN [33] are the most popular structured p2p systems. In unstructured p2p systems, there is no assumption about how the data items are placed at the nodes. The location of each data item is unknown. Thus, in order to find a file, the most common method is flooding which induces a lot of communication overhead in the p2p network. The most popular unstructured p2p system is Gnutella that uses flooding as a search method. In addition, p2p systems can be classified based on the degree of decentralization. Thus, p2p systems can be categorized either as *pure*, where all nodes have equal roles, i.e., playing both the role of a client and a server, or as *hybrid* where some nodes, denoted as *super-peers,* have different roles from the rest of the nodes, denoted as *leaf-nodes*. Each super-peer acts like a proxy for its neighboring leaf-nodes by indexing their data-items and servicing their requests. Figure 2.1 demonstrates the p2p classification described above.

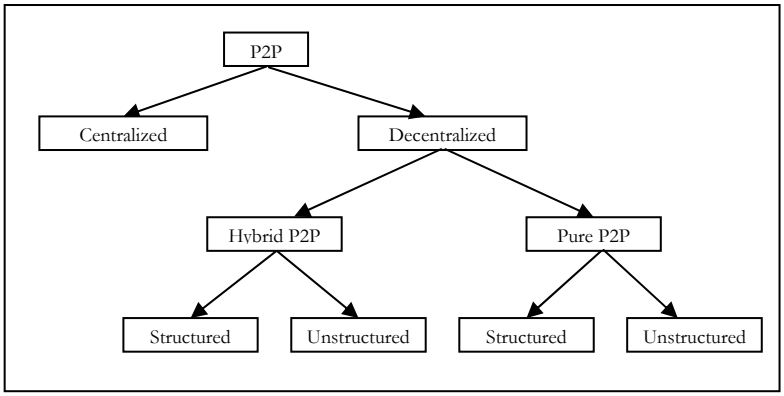


Figure 2.1: Classification of P2P Systems.

In the following sections, we summarize recent work on several issues that have risen with the arrival of p2p systems. In Section 2.1.1, we describe several methods that have been adopted by p2p systems to cluster nodes or data items in the overlay network and Section 2.1.2 refers to the most important search techniques. In Section 2.1.3, we describe several p2p systems that support range queries from the perspective of clustering and routing.

2.1.1. Clustering in P2P Systems

In peer-to-peer systems there are two categories of methods on how clustering can be achieved [20]. Both of them have the intention to place together data that have similar properties. The first category clusters similar data or indexes of similar data. Thus, similar data (or indexes of similar data) are placed at the same or neighboring nodes. In contrast, the second category groups nodes that have similar data. By clustering nodes with relevant data, when a query is routed and finds a node with the desired data, then with high probability this node must be at the appropriate cluster, thus, all the other nodes that have similar data can be found within a short distance. In centralized p2p systems, no clustering is applied. Hence, a node joins the p2p network in a random fashion. In the following two sections, we describe several clustering methods for structured and unstructured p2p systems respectively.

2.1.1.1. Clustering in Structured P2P Systems

Structured p2p systems use the first category to achieve clustering. As mentioned before, at this kind of p2p systems, a key derived from a hash function is assigned to each data item.

CHORD [37] assigns (using a hash function) to each node of the overlay network an identifier so as each node to maintain a small fraction of (key, data) pairs. In more details, the nodes identifier space is represented as a ring and each data item's associative key k is assigned to the node, denoted as *successor* of key k, whose identifier is equal or follows in the identifier space the key value k. Figure 2.2 shows an example with four nodes and in which node each key will be stored. When a new node n enters the system then the appropriate keys stored at n's successor must be reassigned to n. To implement this procedure and for better routing performance, each node maintains information about a small fraction, O(logN), of the other N system's nodes in a structure called *finger table*. The i-th entry of node k's finger table includes the identity of the first node that succeeds node k a distance at least 2^{i-1}, i=1, …, m on the circle. Hence, it has the information about the exact node location of the data keys that intuitively must be placed at nodes with distance 2^{i-1}, i=1, …, m far from node k. In Figure 2.2 we represent the finger table of node 0 that points to nodes 1, 2 and 4 respectively. Since nodes 2 and 4 are not in the system, node 0 points to nodes 3 and 6 that immediately follow nodes 2 and 4 respectively. Thus, when a new node n joins the system the three steps that must be followed are: firstly, n must connect with an existing node n' in the system and initialize its finger table using n' support, secondly the finger tables of the existing nodes in the system must be updated to include node n, and finally the appropriate keys that is responsible for must be transferred to node n from its successor, i.e., the first node clockwise in the ring from n.

In CAN [33] the hash table is represented as a d-dimensional Cartesian coordinate space and is partitioned among all the CAN nodes of the system. The associative keys of data items are mapped onto points of the coordinate space. Hence, in each node an individual chunk (zone) of the coordinate space is assigned. Figure 2.3 illustrates a 2-dimensional coordinate space with 6 nodes. The (key, data) pairs stored in a node are those whose key values are contained in the zone of that node. Thus, similar keys are laid in the same node. Furthermore, two nodes in the overlay network are immediate neighbors if their zones are adjacent. This means that relevant keys, hence relevant data, are either in the same node or at adjacent nodes. The clustering of the data items is more clear when a new node n_i joins the CAN network. This node must allocate its own chunk in the coordinate space. That can be achieved by first connecting randomly the new node with a CAN node currently in the system. Then, the new node randomly selects a point P from the coordinate space and sends a message in the overlay CAN network to find the node n_j whose zone contains the point P. The occupant node n_j splits its own zone in two equal chunks and assigns one of them to the new node. Finally, the node n_i connects immediately with node n_j and with a subset of n_j's neighbors, whose zones are

adjacent with n_i's zone, while node n_j updates its neighbor set in order to delete the immediate connections it had with the neighbor nodes that are no longer its neighbors after the join of the new node n_i in the CAN network. From the join procedure it is obvious that when a new node enters the CAN network and "selects" a chunk of the coordinate space, the data items whose keys are included in this chunk, which are similar data, will be stored in the new node. Furthermore this node will have neighbors with adjacent zones with its zone, thus the data stored to its neighbor nodes will be similar to its stored data.

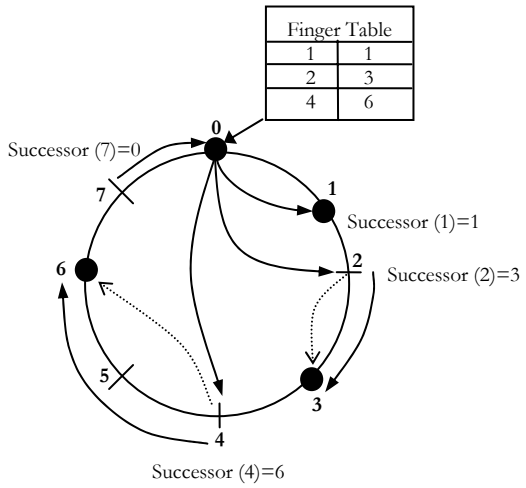


Figure 2.2: An Identifier Circle Consisting of the four Nodes 0, 1, 3 and 6. Finger Table for Node 0 and Key Locations for Keys 1, 2 and 7. In this Example, Key 1 is Located at Node 1, Key 2 at Node 3, and Key 7 at Node 0.

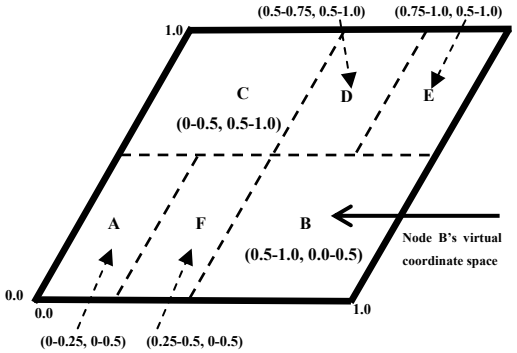


Figure 2.3: Example 2-d Space with 6 Nodes.

In [39], the notion of semantic overlay networks is introduced where indices of documents are distributed across the network based on their semantics. Thus, documents with similar semantics are clustered either at the same node or at nodes with small distance between them, creating a semantic overlay network. In particular, each document is represented by a semantic vector in a Cartesian space, based on its content. These vectors are derived using either the vector space model (VSM) or latent semantic indexing (LSI). Hence, the similarity of two documents is proportional to the similarity of their semantic vectors. The Cartesian space is used along with CAN, so as the location, i.e., point, of a document in the CAN's coordinate space is derived from its semantic vector. So, similar documents are placed in a short distance at the Cartesian space. Furthermore, as mentioned in CAN, for each node of the

overlay network is assigned a zone of the space. Thus, each node will store the indices of semantically similar documents.

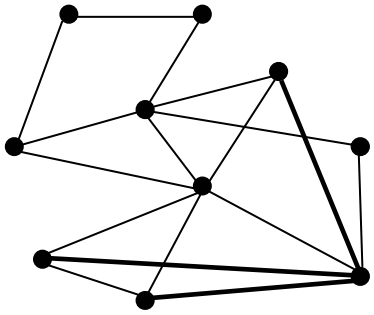


Figure 2.4: A Gnutella Overlay Network with three Shortcut-links for the Bottom Right Node.

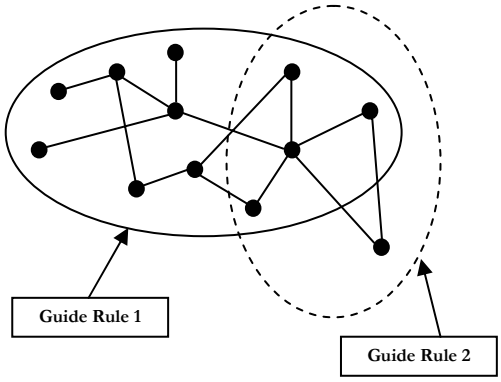


Figure 2.5: Overlay Network with two Overlapping Guide Rules.

2.1.1.2. Clustering in Unstructured P2P Systems

Gnutella does not provide any kind of clustering, i.e., when a new node joins the network connects with a small set of random nodes of the network. In [36], some kind of clustering for Gnutella is proposed based on grouping nodes that share the same interests. More specifically, this mechanism organizes the Gnutella nodes into a clustered network on top of the Gnutella network. Thus, the basic Gnutella topology is retained and in addition a new network of nodes is created on top of Gnutella consisting of *shortcut* links. Figure 2.4 illustrates a Gnutella overlay network with three shortcut links for the bottom-right node. The criterion for creating a shortcut link between two nodes based on *interest-based-locality*, e.g., if a node n_i has a data that another node n_j requested, then node n_i is very likely to have other data items that node n_j is interested in. Hence, these two nodes share the same interests. Each node maintains a shortcut list with the nodes to which it is connected through shortcuts. Initially, a node joins the system in a Gnutella-like fashion since it has no information about the other nodes interests. Then, when it looks up for a data item, a set of nodes is returned that stores the desired data. The newly joined node will create shortcuts with these nodes updating its shortcut list. Hence, the node is connected, clustered, with nodes that share similar interests.

In Semantic Overlay Networks (SONs) [10], nodes that have similar documents are clustered at the same group, denoted as SON. In particular, a classification hierarchy is used to classify the nodes documents, which is defined a priori. Thus, two nodes belong to the same overlay network (SON) if some of their documents are relevant, i.e., they are classified under the same concept. This way, a set of overlay networks are formed; the number of these networks is predefined. All nodes that belong to the same overlay network have documents that belong to the same concept. In addition, nodes can belong to more than one SON. Thus, when a node wishes to join the p2p network, it initially floods the network to obtain the classification hierarchy. It then decides which SONs to join. This can be done by classifying its documents to their associative concepts. The next step is to find nodes for each SON that it belongs to. This can be done again by flooding the network.

Another unstructured p2p system, which provides clustering of nodes with similar data, is the Associative Overlays [8]. The overlay network is constituted by *guide rules*. Each guide rule is a set of nodes that satisfy some predicate. Thus, all the nodes that belong to the same guide rule contain similar data. The connectivity of nodes inside a guide rule is similar to the connectivity of unstructured network. Figure 2.5 illustrates a set of nodes with two overlapping guide rules. Several kinds of guide rules can be proposed. *Possession rules* is the guide rule applied in [8], where each possession rule has an associative data item. The predicate a node must satisfy to be included in a possession rule is the presence of the possession's rule associative data item to its local index. Note that a node can participate in more than one guide rules.

2.1.2. Searching in P2P Systems

Several search techniques have been proposed in the literature in order to accomplish discovery in a small number of hops and getting as much query results as possible. In centralized systems, such as Napster, a centralized index is storing information about the contents of all nodes in the network. Queries are issued to this central index to find the nodes that have the desired files. The files are then downloaded directly from these nodes. The drawback of this approach is that the central index server becomes a bottleneck and a single point of failure. In decentralized systems, the search methods can be categorized in two main domains. The first domain refers to those methods applied to structured p2p systems and the second domain to methods applied to unstructured p2p systems. The categorization is done due to the morphology of these two kinds of p2p systems. More specifically, at the first one

data items are placed at specific nodes in contrast with the other kind where there is no assumption about the location, i.e., the nodes, the data items will be placed at.

2.1.2.1. Searching in Structured P2P Systems

In general, searching in structured p2p systems is very efficient because specific data items are placed at specific nodes, thus all lookups are resolved successfully.

In Chord [37], an efficient method for routing can be implemented using the nodes finger tables. When a requesting node asks for a key k, it must check its finger table to see whether one of the nodes for which has information about is the successor of the key. If so, it can contact immediately the successor node of the key, as it knows its identity, hence the lookup is resolved in one hop. Otherwise, when the requesting node does not know about the successor of key k it must find another node j in its finger table, whose identifier is closer and precedes the key. Node j repeats this procedure. Hence, at each step the query is forwarded to nodes that are closer and closer to the key. Finally, the query message finds the immediate predecessor node of the node whose identifier is associated with key k. The successor of that node holds the key k, thus the lookup procedure is resolved correctly. It has been shown that a query request is resolved in an N-node network in O(logN) hops.

In CAN [33], each node maintains the IP address and the coordinate zones of its neighbors. When a query, generated in a CAN node i, requires a data item then the node i can learn data item's associative key, using the hash function, and the point P of the coordinate space corresponding to the key. If the point P is not within the requesting node's zone routing is deployed. The routing works by following the straight line of the coordinate space that connects the requesting node's coordinates with the associative coordinates of the data item (point P). In more details, a CAN node forwards the query to one of its neighbors whose coordinates are closest to the destination coordinates. It has shown that for a d-dimensional space, the average path length is $(d/4)(n^{1/d})$ hops which means that increasing the number of dimensions of the coordinate space the average path length grows by $O(n^{1/d})$.

Finally, *pSearch* procedure was introduced in [39] in order to achieve routing in semantic overlay networks. When a query q is created at a node, initially its semantic vector is generated, i.e., a point of the Cartesian space, and the query message is forwarded to the overlay network, as described in CAN, in order to find the destination, i.e., the node that contains this point in its zone. Upon reaching the destination, the query is flooded to all nodes

within a certain radius. Finally, all nodes that get the query message execute a search to their local index so as to find query matches.

2.1.2.2. Searching in Unstructured P2P Systems

In [40] the search methods in unstructured systems are classified in two categories, blind and informed search. In blind methods, there is no information about the location of a document. In contrast, in informed methods a distributed directory service contributes in discovering the requested data location.

Blind search methods are very popular due to their simplicity. The most simple blind search mechanism is *flooding*, used by Gnutella. When a node initiates a query, in order to find results, it sends the query message to its neighbors that forward the query message to their neighbors and so on. Thus, the whole network or a subset of it, if we use the *TTL* (time-to-live) parameter, is flooded in order to find as many matches as possible matches for the query. The TTL parameter represents the maximum number of hops the query message can travel before it gets discarded. The main disadvantages of this method are the large network overhead and the TTL selection problem. In order to eliminate this huge cost, several search methods, variations of flooding have been proposed. The first one is *Modified-BFS* [40] where each node instead of forwarding the query message to all of its neighbors, it chooses randomly a fraction of its neighbors. This algorithm reduces the network overhead of flooding but not as much as we wanted. In order to eliminate further the network overhead and resolve the TTL selection problem, the *expanding ring* [25] search method was proposed. This technique uses iterative flooding searches with increasing TTLs. The requesting node begins the search by flooding the network with a small TTL and if the search is not successful then the requesting node repeats the same procedure using an increased value for TTL and so on. The advantage of this method is that it is possible to find relative to the query data using a small TTL, hence incurring a small network overhead. More specifically, “popular” data that spread across a large fraction of the network nodes can be found with little network overhead. Instead, rare data can bring larger network overhead than flooding. Furthermore, expanding ring does not solve the problem of message duplication where a node receives the same query message from different neighbors. *Random walks* [25] try to solve this issue. The requesting node forwards the query message to a fraction k of its neighbors set. Then, each of the intermediate nodes forwards the query message to a randomly selected node. Hence, the query message follows k different paths, denoted as walks, in order to find the query results.

To terminate a walk either a specified TTL is used or “checking”, i.e., the walker asks first the requesting node before deciding to visit a new node.

For hybrid p2p systems, two blind search techniques have been proposed. In hybrid p2p systems each super-peer is “responsible” for a number of other peers, denoted as “leaf-nodes”. In GUESS [40], the super-peers are fully connected and each super-peer is connected with a fraction of other nodes of the network. The search is done by repeatedly forwarding the query message to a different super-peer and furthermore forwarding the query to the super-peer’s leaf nodes. The search is terminated when a specified number of results has been found. In Gnutella2 [40], when a super-peer gets a query message from a leaf-node, it forwards the message to the other relevant leaf-nodes and also to its neighboring super-peers.

Also, several informed search techniques have been proposed. The most important are described below. In order to eliminate the weakness of Gnutella flooding, that is scalability, an informed search mechanism is proposed in [36] that uses the *shortcut* links mentioned in previous section. In particular, when a node requests for a data item, firstly it uses its shortcut list to select the nodes to which will forward the query message. If the data item cannot be located through shortcuts then the node uses the underlying Gnutella network and performs flooding, otherwise its shortcut list must be updated with the nodes storing the returned data item.

Another informed search technique is the *Intelligent-BFS* [40] where each node stores for each of its neighbors the number of results returned from recently answered queries. Thus, when a node receives a query message, identifies all the previous queries that are similar with this query and forwards the query message to the set of neighbors that returned the most results for the similar previously answered queries.

An alternative informed search method uses *local-indices* [40] where each node has information about the files stored at nodes within a specific distance from it. Thus, a node returns query results on behalf of a set of other nodes. The query message can be forwarded only at the nodes that are within a specified distance from the requesting node. A very promising informed search technique, based on distributed indexing, is proposed in [9]. Each node of the network maintains *routing indices* (RI), one for each of its links. Each node’s routing index summarizes the content of other nodes of the network that can be reached through the selected node’s link. When a node receives a query, it firstly evaluates it using its local data and secondly if not enough results have been found it forwards this query to other

nodes of the network. Instead of routing the query message to all of the node's neighbors, as flooding does, with routing indices a node selects some of its neighbors to forward the query. Hence, the query is routed through the paths that the more relevant data items can be found. Several routing indices have been proposed. *Compound Routing Indices* (CRI) of node n for a specified link l summarize information about the content of other nodes that are reachable from node n through link l. The main drawback of the CRIs is that they don't take into account the cost of the number of "hops" required to find the desired data. The *hop-count RIs* overcome this problem by storing for link l a routing index for each number of hops up to a maximum number of hops, denoted as *horizon*. The drawback of this index is that it requires higher storage than CRIs and does not have any information for nodes beyond the horizon. The *exponential RI* is a combination of both CRIs and hop-count RIs, eliminating their drawbacks and generally outperforming the other two kinds of RIs.

The idea of routing indexes is also used in [29], [22]. In particular, the authors propose using histograms as local and routing indexes. Each peer maintains a local index of all data available locally. It also maintains for each of its links, one histogram as routing index that summarizes the content of all peers reachable through this link within a given number of hops. Such histograms are used to route range queries and maximize the number of results returned for a given number of peers visited.

A similar approach is presented in [21], [27]. Bloom filters are used as indexes. Specifically, in [21] the authors use multi-level Bloom filters to route queries in a p2p system. Each peer maintains a summary of its content in the form of a multi-level Bloom filter and it also maintains one merged multi-level Bloom filter for each of its links summarizing the content of all peers that can be reached through this link. Using such merged filters, each peer decides to direct a query only through links that may lead to peers with documents that match a query. In [27], each peer maintains a local Bloom filter that represents the object in the local repository, and a remote Bloom filter for each link obtained from its neighbors. During the query routing, each node propagates the query to the best k links based on the semantic similarity with the query. When a node discovers that a peer frequently produces good results to its request, it attempts to move closer to it by connecting directly to that peer.

Furthermore, a query routing technique that relies on *routing filters* is introduced in [15]. In particular, the authors present an adaptive maintenance technique based on query feedback for keeping routing filters up-to-date. These routing filters are used for optimizing routing, which prevents from flooding the network. Routing filters are local index structures established at

each peer and are based on *path trees*, an adaptable summary structure for XML data. They compound information about data obtainable by the connections to all neighboring peers, each peer is restricted to limited knowledge about global data location and other participating peers by limiting the filters to a maximal hop count. In addition, each peer holds one separate filter instance for each neighbor. The basic query processing procedure takes place in four main steps, which are: 1. query forwarding, where each peer that receives a query message forwards it to all those neighboring peers that, according to its routing filters, can contribute to the final result, 2. local query processing, 3. updating the cache for intermediate results, and 4. sending an answer message to that peer that has sent the message in the first place. The central issue is how to obtain and maintain the knowledge contained in the routing filters. They utilize a dynamic and adaptive maintenance approach based on query feedback, where the results of executed queries are used in order to refine the filters. Thus, the routing filters adapt to the actual workload and changing network situations.

A search method that cannot be categorized neither as blind nor as informed, is the one used in *Overlay Semantic Networks* [10]. When a node creates the query, first it classifies it and then forwards the query message to the appropriate SON. Then, the query message is propagated at the nodes of the SON, in order to find matching documents, using flooding. As mentioned before, this method cannot be categorized neither as blind nor as informed, due to the reason that in [10], it is not determined how the query is forwarded to the appropriate SON.

The *Guided search* method [8] can be viewed as a combination between blind search and routed search used in structured p2p systems. In more details, when a node originates a query, decides in which possession rules the query message is forwarded. Thus, the queries are directed to nodes that are likely to have the most results. Two possible search algorithms have been proposed about deciding in which possession rule the query must be forwarded. In *Rapier* (Random Possession Rule) algorithm a node selects randomly a possession rule from a set of possession rules that have been selected to answer previous node's queries. In contrast in *GAS* (Greedy Guide Rule) algorithm each node creates its own strategy consisting by possession rules that have been more effective in previous node's queries. It has been shown that GAS performance is better than Rapier's but with higher overhead. After the selection of the possession rule, a blind search is performed inside this possession rule.

2.1.3. P2P Systems Supporting Range Queries

In the previous sections, we discussed about p2p systems that support only simple lookup queries over a single attribute. However, many new p2p applications, such as p2p photo-sharing applications and multi-player online games, require support for range queries. In the following sections we describe several p2p systems that support multi-attribute range queries from the perspective of how these systems achieve clustering and the search techniques they use to resolve range queries.

2.1.3.1. Systems Overview

Mercury [5] is a structured p2p system that supports multi-attribute range queries, i.e., each query is a conjunction of ranges in one or more attributes. Its basic architecture relies on creating a structure, denoted as *routing hub,* for each attribute in the application schema consisting of a subset of system nodes. Note that a node can be part of multiple hubs. In addition, each routing hub is organized into a circular overlay of nodes and each node within a routing hub takes charge of a contiguous range of values for the equivalent attribute. Thus, two data items that have contiguous values for a specific attribute will be placed at the same node or at neighboring nodes in the routing hub that "represents" this attribute. This kind of structure is similar with CHORD's but the main difference is that Mercury doesn't use randomized hash functions for placing data due to the choice of supporting range queries.

In [34], a method for efficiently evaluating multi-attribute range queries is proposed. This kind of method uses a 2d-dimensional coordinate space in a way similar to CAN system. In more details, considering a relation with d attributes the system creates a 2d coordinate space, i.e., 2 dimensions of the space correspond to a single attribute. The coordinate space is constructed as follows: Assuming that the domain of a single attribute is [a, b], the boundaries of the two-dimensional coordinate space are (a, a), (b, a), (b, b) and (a, b). In addition, the space is partitioned into multiple rectangular sub-regions, denoted as *zones*, where each zone is assigned to a system node. Note that the zones are assigned to only some of the system nodes, denoted as *active* nodes, whereas the remaining nodes that do not own a zone are called *passive* nodes. Furthermore, each active node keeps a list of passive nodes and links to its neighbors, i.e., the nodes that are owners of adjacent zones. The partitioning of the coordinate space is done dynamically by splitting existing zones and new zones are assigned to passive nodes that become active. The decision of splitting a zone is taken by the owner of the zone. For splitting a zone one of the following two conditions must hold. The first condition is met when the owner of the zone answers too many queries; hence, it splits its

corresponding zone into two chunks of even distribution of stored answers. The second condition is satisfied when a node is overloaded because of too many query routing messages. Then, this node splits its owned zone into two equal chunks. Each range query corresponds to a single point, denoted as *target point*, in the coordinate space therefore each node is responsible for a number of range queries that their points are included to its owned zone. The zone in which a target point lays and the corresponding node are called *target zone* and *target node*, respectively. Thus, each target node stores the data items that are associated with the range queries for which the target node is responsible. When a node generates a range query and wishes to publish the query results to the network, it caches the query results and the target node creates a pointer to it. The target node also caches the results of the range query.

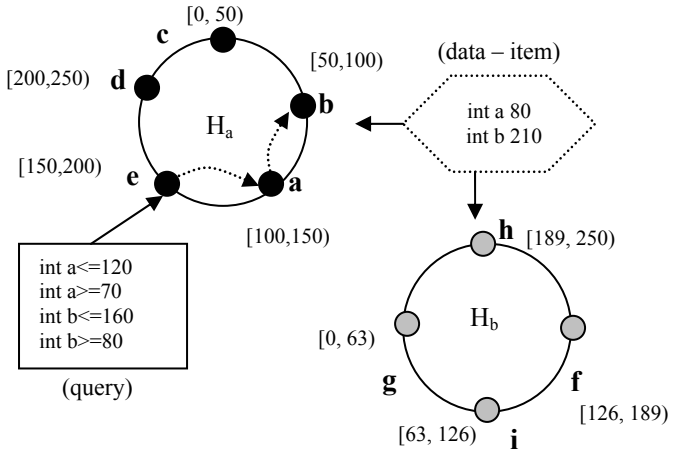


Figure 2.6: Example of Data Item Insertion and Query Routing in Mercury.

Space-Filling curves with range Partitioning (SCRAP) [12] and Multi-Dimensional Rectangulation with KD-Trees (MURK) [12] are two alternative approaches for supporting multi-dimensional range queries. In SCRAP, the multi-dimensional data items, i.e., the data items having more than one attribute, are mapped into a single dimension by using a space-filling curve. These one-dimensional data items are then partitioned among the system's nodes where each node is responsible for a contiguous range of values for the single dimension. In MURK, the multi-dimensional data space, i.e., the space where each multi-attribute data item is represented as a point, is partitioned into sub-regions, denoted as "rectangles", and each rectangle is assigned to one of the system nodes. To achieve this partitioning, MURK uses Kd-trees, in which each leaf corresponds to a rectangle. This kind of partitioning is similar with the partitioning of the CAN's coordinate space. The main

difference is that CAN tries to partition the coordinate space into equal sub-regions since the data items are expected to be uniformly distributed. In contrast, MURK splits its data space considering that each rectangle must have the same load. Another difference between CAN and MURK is that in CAN the number of dimensions for the coordinate space is determined by the routing efficiency, rather than the dimensionality of the data items, as MURK does.

2.1.3.2. Query Routing

In Mercury [5], the first step for routing a query is to select one of the hubs, let's say H_a, which corresponds to the queried attributes. Hence, to guarantee that the query will find all the relevant data items, each data item is placed at all hubs that are associated with the attributes for which the data item has value. After the selection of the hub H_a, the next step is routing the query within this hub, which is done by forwarding the query to the node that is responsible for the first value of the query range for attribute a. Then, using the property of contiguity that Mercury has, the query is spread along the nodes of the ring to find all relevant data items.

Example 2.1*: Consider that we have two hubs H_a and H_b that correspond to attributes a and b respectively. The value range of both attributes is [0, 250]. When a new data item, with values 80 and 210 for the attributes a and b, is inserted into the system it is sent to both hubs H_a and H_b and it is stored at nodes b and h. In addition, a query is initiated at a system node and wants to find all the data items with $70 \le a \le 120$ and $80 \le b \le 160$. It selects the hub H_a to execute the routing and enters at node e. Then, the query is routed within H_a and finds the associated results at nodes a and b. The whole process is illustrated in Figure 2.6.*

The implementation of routing requires each node that participates in a routing hub to maintain a link to each of the other routing hubs, denoted as *cross-hub links,* so as to route the query to another hub, and to have links to its predecessor and successor nodes within its own hub for routing the query within the chosen hub. Using only predecessor and successor links query routing is not efficient since in the worst case a query can get flooded to all nodes within a hub. Thus, Mercury adopts a routing optimization, denoted as *k long-distance links*. In particular, besides the predecessor and successor links, each node maintains k links to other nodes of the same hub. It has been shown that with k long-distance links the number of hops required for routing is logarithmic.

In [34], when a node generates a range query, the query is routed to its target zone. In particular, when a query is initiated at a zone, the requesting node forwards the query message to one of its neighbors, i.e., to a node that is responsible for an adjacent zone, that its corresponding zone coordinates are the closest to the target zone. All the nodes, which get the query message, follow this procedure until the target node is reached. The routing of the query message is done by using each node its neighbor lists and the target point of the query. Note that a passive node can generate a range query. Hence, the passive node must forward the query message to any of the active nodes. The query is then routed to the target zone by following the above procedure. It has been shown that the routing path is $O(\sqrt{n})$, where n is the number of zones in the system. When the query message reaches the target zone, the target node checks if it has stored results from previous range queries that contain the query range. If so, then these results are forwarded directly to the requesting node. Furthermore, if the target node has a pointer to another node, let's say n_j, that contains results from a superset range, then the IP address of n_j is returned as an answer to the requesting node, which can contact immediately with n_j. If query results cannot be found locally to the target node, the query is forwarded to the top and left neighbors, which potentially contain results for the query. These neighbors check for local results and can also forward the query to their top-left neighbors recursively.

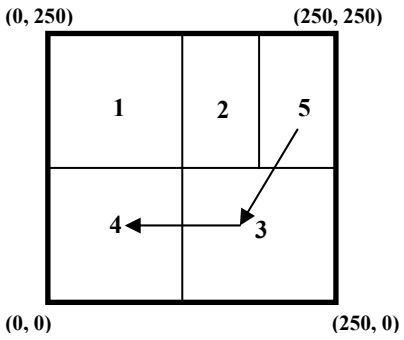


Figure 2.7: Partitioning of the Coordinate Space and Routing of the Range Query <70, 120>.

Example 2.2*: Consider that we have data items with one attribute and the value domain of the attribute is [0, 250], as in Example 2.1. Then, the boundaries of the coordinate space are (0, 0), (250, 0), (0, 250) and (250, 250). In addition, we assume that we have five active nodes. The five zones of the coordinate space are: zone-1 <(0, 125), (125, 250)>, zone-2 <(125, 125), (188, 250)>, zone-3 <(125, 0), (250, 125)>, zone-4 <(0, 0), (125, 125)>, zone-5 <(188, 125), (250, 250)>. Assuming that the range query <70, 120> is initiated at zone-5*

then the query is routed through zone-3 to its target zone, which is zone-4. Figure 2.7 illustrates the query routing of the range query <70, 120>.

In SCRAP [12], multi-dimensional query routing is executed by following two steps. At the first step, the multi-dimensional range query is divided into a set of one-dimensional range queries and at the next step each of the unidimensional queries are routed to the nodes whose range of values intersects with the query range. The second step is efficiently performed by using a circular list of nodes, denoted as skip graph. Recall, that each node in the skip graph is responsible for a range of contiguous values for the single dimension that the multi-dimensional data items are mapped to. Instead of using only links to its neighbors, that will result in O(n) messages (n is the number of nodes) to locate the node having the query results, each node keeps additional O(logn) *skip pointers* to other nodes of the skip graph. It has been shown that with skip pointers query routing is achieved in O(logn) hops.

In MURK [12] each node n_i creates *grid pointers* to other nodes of the system that store adjacent rectangles of the multi-dimensional data space with the one that node n_i stores. The query routing is executed in a very similar way with CAN. Consider a query generated at node n which requires data items that their corresponding data points are laid within a rectangle Q. If the rectangle Q does not belong to the requesting node, then the routing protocol forwards the query message from the node n to one of its neighbor nodes whose corresponding rectangle reduces the distance to Q by the largest amount. When the query message reaches a node m with relevant to the query data items, node m forwards the query to its neighbors that also contain relevant data items. That requires that each node has information about the corresponding rectangles boundaries of its neighboring nodes. This procedure is done recursively until the query message reaches all relevant to the query nodes. Furthermore, every MURK node uses additionally *skip pointers* to a few other nodes to improve query routing, especially when dimensionality is low. Thus, in the routing protocol when a node forwards the query message to one of its neighbors that is closest to the destination, i.e., the node that is responsible for rectangle Q, this neighbor might be either a "grid" neighbor or a neighbor through a skip pointer.

2.1.3.3. Clustering - Node Join

In Mercury [5], clustering is achieved in each routing hub by placing data items at the node whose range of values includes the data item's value for the hub's attribute. Thus, data items with equal or contiguous attribute values will be placed in the same or contiguous nodes at the

corresponding to the attribute hub. In particular, when a new node n_i joins the system, it initially communicates with a node that is already part of the system and gets information about the system's hubs and a list of representative nodes for each hub. Then, it selects randomly one of the hubs and contacts a node n_j that is already part of the chosen hub. The new node becomes a predecessor of node n_j, takes over half of n_j's values range and finally becomes a part of the hub. Furthermore, the new node must create its own cross-hub links and k-long distance links. To achieve this, it firstly copies these kind of links from its successor n_j and then starts the process of setting up its new k-long distance links and obtaining new cross-hub neighbors, by starting random-walks to each of the other hubs, distinct from those stored by its successor.

MURK [12] achieves clustering by placing similar data items, i.e., data items that their corresponding data points are very close in the data space, to the same or neighboring system nodes. When a node joins the system, the rectangle that is managed by a participant node is split into two parts of equal load and one of them is assigned to the new node.

2.2. Histograms Supporting Selectivity Estimation of Queries

Histogram construction techniques and their use in selectivity estimation of range queries as well as their relationship to approximate query answering have a long research history in the database literature. In databases, several modules of a database system require estimates of query result sizes. For example, query optimizers select the most efficient plan for a query based on the estimation of competing plans. These costs are in turn based on estimates of intermediate result sizes. To estimate query result sizes an accurate approximation of the distribution of data values in attributes of the relations of a database must be maintained. A variety of statistics have been proposed in the literature to approximate the distribution of data values, e.g., histograms, sampling and parametric techniques. Probably the most common technique used in practice (e.g., DB2, Informix, Ingres) is maintaining *histograms*, where a histogram contains the number of tuples in a relation for each of several subsets of values in an attribute. Thus, histograms are used as a mechanism for compression and approximation of data distributions and play an important role in estimating query result sizes. In particular, *histograms* approximate the frequency distribution of a single (or more) attribute (attributes) by grouping attribute values into subsets, called *"buckets"*, and approximating true attribute values and their frequencies based on summary statistics maintained in each bucket. The main advantages of histograms over other techniques are that they incur almost no run-time overhead and they do not require the data to fit a probability distribution. In addition, there

exist histograms that produce low-error estimates while occupying reasonably small space [18].

2.2.1. One-dimensional Histograms

Several one dimensional histograms, i.e., histograms that approximate the data distribution of a single attribute, have been proposed in the literature. A histogram on an attribute *x* is constructed by partitioning the attribute's data distribution into a number of mutually disjoint subsets called *buckets* and approximating the frequencies and values in each bucket.

Earlier work deals with histograms in the context of single operations, primarily selection. In particular, in [30] the authors consider the problem of reducing the error for selection queries using histograms. They study the *equi-width* histograms and propose an alternative kind of histograms, the e*qui-depth* (or *equi-height*) histograms. Specifically, an *equi-width* histogram on an attribute, e.g., *x*, is specified by partitioning the value set of *x* into contiguous values ranges of equal width (buckets). Thus, the number of attribute values associated with each bucket is the same. For each bucket, the sum of the frequencies of the values that lie within the bucket, i.e., the total number of tuples with values for *x* within the value range of the bucket, is kept. In contrast, in an *equi-depth* histogram the value set of *x* is divided into buckets so as the sum of the total number of tuples having the attribute values associated with each bucket is the same for each bucket. Their main result shows that an equi-width histogram is not very good for estimating selectivity, since it frequently leads to estimates not much better than those obtained by random guessing. In contrast, the equi-depth histograms have a much lower worst-case and average error, compare to equi-width histograms, in estimating queries results for a variety of selection queries.

The notion of variance optimality of histograms for limiting the errors in the estimates of query result sizes is introduced in [16], [17] and the *v-optimal* histogram is defined to be that which minimizes the error for estimating *tree equality-join* queries for a given number of buckets. For example, if R_0, R_1, R_2 are relation names and a, b are attributes of both, the authors do not deal with queries whose qualifications contain ($R_0.a = R_1.a \; and \; R_0.b = R_1.b$). Instead, they deal with *chain* join queries, i.e., ones whose qualification is of the generic form $Q = (R_0.a_1 = R_1.a_1 \; and \; R_1.a_2 = R_2.a_2 \; and \; ... \; and \; R_{N-1}.a_N = R_N.a_N)$, where $R_0, ..., R_N$ are relations and $a_1, ..., a_N$ are appropriate attributes. In particular, in [16], the tree equality-join queries are studied for which the result size reaches some extreme that essentially represent

the worst case of error. It is shown that optimality for error reduction is achieved by the class of *serial* histograms, which are introduced, when the query result size is maximized, whether or not the attribute independence assumption holds, and when the query result size is minimized and the attribute independence assumption holds. Since then, traditional histograms are usually constructed in such a way that each bucket stores attribute values that belong in a certain range in the natural order of the attribute domain. In contrast, serial histograms are constructed so that attribute values are grouped in buckets based on proximity in their corresponding frequencies and not in their actual values. Thus, this class of histograms group similar frequencies together, thereby reducing the variances of frequencies in the buckets.

In addition, in [17], the authors investigate the trade-offs between histogram optimality and practicality. They prove that the class of serial histograms are optimal for tree equality-join queries either when the joint-frequency distribution is available or in the case that the frequency distribution of the relations of a query are individually available but are examined in isolation, which is the most common situation in practice. Furthermore, the optimal histogram on a join attribute of a query relation is proved to be independent of the rest of the query and is equal to the optimal histogram for the query that joins the relation with itself on that attribute. Thus, optimal histograms can be identified independently for each relation. They introduce the *v-optimal serial* histograms or *v-optimal* histograms and as in serial histograms they group contiguous sets of frequencies into buckets so as to minimize the variance of the overall frequency approximation. Unfortunately, the construction of a v-optimal histogram is too expensive; its construction complexity is exponential. Due to the high complexity for constructing an optimal serial histogram, a subclass of serial histograms is studied, the *end-biased* histograms class, that accurately maintain the frequencies of some attribute values and assume the uniform distribution for the rest. A comparison between the optimal histogram in that class, i.e., v-optimal end-biased histogram, and the overall optimal (serial) histogram, i.e., the v-optimal serial histogram, is done and shows that the histograms in this subclass are effective in error reduction. Furthermore, the histograms of this subclass can be constructed much more efficiently than the v-optimal serial histograms.

In [31], several key properties that characterize histograms are identified and a formal taxonomy of existing histograms is formed based on these properties. After placing all existing histogram types, such as equi-width, equi-depth, v-optimal and v-optimal end-biased histograms in the appropriate places in the taxonomy, new histogram types are proposed such as *maxdiff* and *compressed* histograms. Let β be the number of buckets in a histogram. In

general, the *maxdiff* histogram is constructed by a heuristic technique that places the bucket boundaries between those β pairs of adjacent values that differ the most in their frequencies. In contrast, in a *compressed* histogram the *n* highest frequencies are stored separately in *n* singleton buckets, i.e., they contain a single value and its corresponding frequency, and the rest are partitioned as in an equi-width or equi-depth histogram. The accuracy of both old and new histogram was determined using a large set of data distribution and range selection predicates. The v-optimal histograms have been shown to minimize the average error for several selectivity estimation problems, thus achieving good accuracy for those predicates as well, but as we mentioned before no efficient algorithm for constructing them has been proposed. In addition, the maxdiff histograms are very close to the best histograms on construction time and generated error issues.

In [19], several algorithms are proposed for constructing v-optimal histograms, i.e., algorithms that attempt to minimize the error for a given number of buckets. The authors propose three algorithms for the problem, all of which found provably optimal or close to optimal solutions. At first, a basic optimal algorithm is proposed based on dynamic programming, which takes time that is quadratic in the number of distinct values of the attribute being considered and linear in the number of buckets being used. Then, an improved and more sophisticated method of the basic algorithm is presented that can compute optimal histograms in a short time for data distributions over ten of thousands of values. Finally, an approximation algorithm that determines a provably close to optimal histogram is introduced that is significantly faster than the optimal algorithms.

Previous work on computing optimal histograms considers only equality queries when computing the error incurred by a particular choice of a histogram bucket boundaries. In particular, the v-optimal histograms defined in [16], [17] minimize the error for estimating equality join queries. Heuristically, constructed v-optimal histograms are evaluated in [31] for range selection predicates along with several other histograms and are shown to achieve a good accuracy for those predicates as well. However, the evaluation of v-optimal histograms for range queries was performed on v-optimal histograms constructed by taking only equality queries into account. In [23], the authors address this problem and focus on efficiently computing optimal histograms for the case of *hierarchical range* queries. A range query q_{ij} over an attribute *x* asks for the sum of the tuples that have value for the attribute *x* between the range [i, j]. Thus, a set *S* of range queries is said to be *hierarchical*, if for any two queries q_{ij} and q_{kl} in *S*, either the range [i, j] and [k, l] are disjoint, or one is contained in the other. They show that "optimal" histograms for equality queries are sub-optimal for hierarchical range

queries. In addition, they present polynomial-time, dynamic programming algorithms for computing optimal histograms that provably minimize expected error for a given amount of space, for the special cases of one-sided ranges and balanced binary trees, as well as for the general case of arbitrary hierarchical range queries. They prove that the algorithm for the case of one-sided ranges is as efficient in running time as the v-optimal algorithm proposed in [19], which computes optimal histograms for equality queries, and experimentally demonstrate that the histograms produced by their algorithm have substantially lower error.

The histogram types mentioned before have the common characteristic of examining exclusively the data set or a sample of it so as to infer data distribution. Thus, the cost of building them and maintaining or rebuilding them when the data set is modified could become very large. In particular, building a histogram involves scanning or sampling the data, sorting them and finally partitioning them into buckets. For large databases, the cost is significant enough to prevent from building all the histograms that might be useful for estimating query result sizes. A novel approach that helps reduce the cost of building and maintaining histograms for large tables is introduced in [3], and the *Self-tuning (ST)* histograms are proposed. This type of histogram is build not by examining the data but by using feedback information about the execution of the queries on the database *(query workload)*. Specifically, the construction of a Self-tuning histogram is done by firstly building an initial histogram with whatever information we might have about the distribution of the histogram attribute. As queries are issued on the database, the query optimizer uses the histogram to estimate selectivities in the process of finding the best query execution plan. Whenever a plan is executed, the query execution engine can count the actual result size produced by each operator. Thus, the feedback information can be used to refine the histogram. In other words, whenever a query is issued and uses the histogram, the estimated selectivity and the actual selectivity are compared and the histogram is refined based on the selectivity estimation error. Summarizing, an ST-histogram greedily partitions the data domain into disjoint buckets and refines their frequencies using query feedback. After a predetermined number of queries, the histogram is restructured by merging and splitting buckets at a time.

2.2.2. Multi-dimensional Histograms

Several multi-dimensional histograms have been proposed in the literature to compute selectivity estimators of multidimensional data sets. A multidimensional version of the equi-depth histogram is presented in [28] that recursively partitions the data domain, one

dimension at a time, into buckets enclosing the same number of tuples. It is shown that the cost for building a D-dimensional equi-depth histogram is significantly less compared to D times, at least, the cost for creating an equi-depth histogram on a single attribute, as one might expect.

MHist histograms are introduced in [32], based on underlying maxdiff histograms, which at each step choose and partition the most "critical" attribute. In more details, assume that we have a number, let's say n, of attributes x_i for which we would like to create the MHist histogram. The *joint data distribution* $T_{1...n}$ of $x_1,...,x_n$ is the entire set of *(value combination, joint frequency*) pairs, where *value combination* is the set of all possible combinations of the attributes' values and the joint frequency for a specific value combination is the number of tuples in the relation that for each one of the attributes $x_1,...,x_n$ have the corresponding value of the value combination. In addition, the individual data distributions of each one of the attributes are referred as *marginal distributions*. The main idea is to iteratively partition the data domain using a greedy procedure. At each step, the algorithm deals with a set of *partial data* distributions that are subsets of the entire joint data distribution. In other words, each partial data distribution corresponds to a bucket. Initially this set contains the entire joint data distribution, hence one bucket. At each step, from the set of partial data distributions, we choose a partial data distribution, i.e., a bucket, that contains an attribute x_i that is the most in need for partitioning. Such a bucket will have the largest "area gap" between two consecutive values along the x_i*'s* dimension. Thus, we split the bucket that corresponds to the selected partial data distribution along the dimension that corresponds to the x_i attribute. Using this information, MHist iteratively splits buckets until it reaches the desired number of buckets.

All the above histogram techniques are *static* in the sense that after histograms are built, their buckets and frequencies remain fixed regardless of any changes in the data distribution. The *Self-tuning* histogram [3], described in the previous section, is also used in the case of multiple attributes. We refer to Self-tuning histograms for multiple attributes as *STGrid* histograms [3]. STGrid is the first multi-dimensional histogram that uses query feedback to refine buckets. A STGrid histogram greedily partitions the data domain into disjoint buckets that form a grid, and refines their frequencies using query feedback. After a predetermined number of queries, the histogram is restructured by merging and splitting *rows* of buckets at a time (in order to preserve the grid structure).

Recently, *STHoles* [6] is presented, a novel workload-aware multidimensional histogram that allow bucket nesting to capture data regions with reasonably uniform tuple density. This

histogram identifies a novel partitioning strategy that is especially well suited to exploit workload information and query feedback. Intuitively, STHoles exploits query workload to zoom in and spend more resources, i.e., more buckets, in heavily accessed areas, thus allowing some inaccuracy in the rest. Hence, this technique uses information about both the workload (range selection queries) and the data distribution itself, through statistics collected from query result streams. The authors present algorithms that show how to exploit result of queries in the workload and gather associated statistics to progressively build and refine a STHoles histogram. An important consequence of this refinement procedure is that the STHoles histograms can gracefully adapt to changes in the data distribution they approximates without the need to periodically rebuild them.

In [24], *SASH* is presented, a Self-Adaptive Set of Histograms that addresses the problem of building and tuning a set of histograms collectively for multidimensional queries in a self-managed manner based only on query feedback information. SASH is a two-phase method for the online construction and maintenance of a set of histograms. In the online tuning phase, SASH uses the *delta rule* to tune the current set of histograms in response to the estimation error of each query. The estimation error is computed from the true selectivity of a query obtained from the query execution engine, i.e., the query feedback. In the restructuring phase, SASH searches for a new and more accurate set of histograms to replace the current set of histograms. To model the set of histograms they used graphical statistical models and the best model found by SASH includes both the optimal set of histograms and the corresponding optimal memory allocation for each histogram. Thus, SASH addresses both the problem of finding the best attribute sets to build histograms on and the problem of finding the best memory distribution (of a given amount of memory) among the histograms.

CHAPTER 3. WORKLOAD - AWARE CLUSTERING

Peer-to-peer (p2p) systems have become a popular medium to share huge amounts of data. P2p systems distribute the main costs of sharing data, disk space for storing data items and bandwidth for transferring them, across the peers in the network, thus enabling applications to scale without the need for powerful, expensive servers. Their ability to build a resource-rich system by aggregating resources enables them to dwarf the capabilities of many centralized systems for little cost.

We assume a p2p system with a set of autonomous computing nodes (the peers). Each peer is connected to other peers thus forming an overlay network and stores data items or resources that shares with the other peers of the system. This kind of system is dynamic, meaning that the set of peers changes as peers may join and leave the system. A query may be posed at any of the peers, while data items that satisfy the query may be stored at various peers of the system; we call such peers *matching peers*. The query is routed through the overlay network from the peer that posed the query to such matching peers. Our goal is for each query to maximize the number of results returned for a given number of peers visited. We would like

for each query to return more than one result due to several reasons. For example, consider a p2p system where each peer shares music files. Ideally, when a user poses a query, i.e., asks for a song, he would expect to receive the IP addresses of all the peers that store this song. The reason that a user would like to know all the peers that have the desired song, hence as many results as possible, is that he might want to choose from which of the candidates peers to download the desired song based on specific criteria like their bandwidth, e.g., if the peer has dial-up connection to the Internet or is connected via cable-modem or DSL. Another example is when a user poses a query not for a song but for an artist. Hence, he expects to receive all the songs of this artist that are stored at the peers.

3.1. Motivation

Ideally, when a query q is issued at a peer, we would like to route the query through the overlay network from the issuing peer only through peers that have data items satisfying it. The role of the overlay p2p network topology, i.e., how the peers are connected to each other, is crucial for efficient query routing. Specifically, in a random p2p network where each peer is connected with a small set of peers selected randomly, e.g., Gnutella [1], when a query is initiated at a peer, we have to perform flooding or a variation of it in order to find matching results for the query. Thus, either the whole network is flooded or a subset of it, if we use the *TTL* (time-to-live) parameter. Both of these two approaches have serious drawbacks. The first one finds all the matching results for the query but it brings a large communication overhead due to visiting all the peers of the network. The second one may eliminate this huge network cost but the effectiveness in finding a satisfactory number of query results is very low. The main problem both for low performance in finding query results and high network overhead using flooding with or without the TTL parameter respectively, is owed to visiting peers that may not have relevant to the query data items due to the random network topology. Hence, we would like to create a network topology so as for each query to visit only a specified number of peers that have the most results for the query. This way, on one hand, we eliminate the network overhead, since we do not forward the query to all the peers of the network, and on the other hand we get the majority of data items that satisfy the query. Thus, for each query we are interested in maximizing the number of matching results returned for a given number of peers visited.

One way to attain a large number of results is by clustering peers based on their content so that peers with similar content are nearby in the overlay network. The motivation for clustering is that once in the appropriate group, i.e., when a query finds a peer with a large

number of results, relevant peers that also have many results are nearby in the overlay network. We propose building a p2p network topology based on *workload-aware* clustering. In particular, the formation of clusters relies on two basic factors, (i) *peers content*, where two peers with similar contents will participate in the same cluster, and (ii) *query workload*, so that the type and probability of queries is taken into account in creating the clusters. Hence, "popular" queries affect the formation of peer groups more than unpopular ones. The motivation for taking into account the query workload is that if some data items are queried only seldomly, we do not want them to influence clustering as much as other data items.

3.2. System Model

Assume that we have a p2p system consisting of a set N of peers n_i. Each peer is connected to a small number of other peers called its *neighbors*, thus forming an overlay network. We focus on p2p systems where each peer stores a relation R with a numeric attribute x, which is a non-negative integer-valued attribute, and on routing queries on x.

3.2.1. Definitions and Query Result Set Requirements

In general, the domain D of x is the set of all possible values of x and the (finite) value set $V_n (\subseteq D)$ for peer n is the set of values of x that are actually present in its relation R. Without loss of generality, we assume a finite numerical domain with discrete values for attribute x, $D = \{0, \ldots, M-1\}$. For peer n, the *frequency* of value i for the attribute x, $0 \le i \le M-1$, denoted as f_i, is the number of tuples (data items) $t \in R$ of peer n with $t.x = i$. In addition, the *probability* of value i, denoted as p_i, $0 \le p_i \le 1$, is a measure of how likely the value i occurs for the attribute x, i.e., $p(t.x = i)$. In general, let S be a set or a bag of items, by $|S|$ we denote the number of items that are included in S. For peer n, its *data distribution* for the attribute x is defined as follows:

Definition 3.1 (Data Distribution) *Let n be a peer which stores a relation R with a numeric attribute x. Let D be the domain of x. The data distribution of x for peer n is the set of pairs* $T = \{(i, p_i)\}, i \in D$.

Our approach is based on the query workload W that consists of a set of queries. Each query is posed at a peer and is routed through other peers of the system to find matching results. For a query q we define as f_q its corresponding frequency, i.e., the number of its occurrences from a

sample of queries, while with p_q we define the probability of the appearance of q. At this point we introduce the notion of the *query workload*:

Definition 3.2 (Query Workload) *A query workload W is a set of pairs (q, f_q) where q denotes a distinct query and f_q its associated frequency.*

As mentioned in Section 3.1, our overall goal is to route each query q only through those peers that have the largest number of results for q, i.e., to maximize the number of results returned for a given number of peers visited. Consider a routing procedure that visits k peers. A routing procedure is optimal for a query q if it visits the best k peers, which is the peers that contain the largest number of results for q. *Definition 3.3* expresses the optimal query routing procedure. Let K be a set of k peers $(K \subseteq N)$ that the routing procedure visits and *results(n, q)* be the number of results of peer n that match query q.

Definition 3.3 (Optimal Query Routing) *A routing procedure visiting only a set K of k peers is optimal for a query q if and only if* $\forall$ *peer* $n \in K \Leftrightarrow \neg\exists$ *peer n'* $\notin K$ *such that results(n', q)* $>$ *results(n, q). We denote the set K as Optimal(q, k).*

In general, we are looking for building networks and designing routing procedures that are optimal for a given query workload W. In more detail, a routing procedure that visits k peers for each query is optimal for a given query workload W, if for each query q of W, an optimal query routing is accomplished. *Optimal Workload Routing* expresses the optimal performance that each routing protocol should try to achieve in a p2p system.

Definition 3.4 (Optimal Workload Routing) *A routing procedure visiting k peers is optimal for a query workload W if* $\forall\, q \in W$*, the procedure visits only the peers in Optimal(q, k).*

When a routing protocol is used in a p2p system, we have to measure how far from the optimal workload routing, the routing protocol performs. To this end, we define *PeerRecall* as our performance measure. Let K be a set of k peers $(K \subseteq N)$, with *Sresults(K, q)* we denote the sum of the number of results (matching tuples) for query q returned by all the peers that belong to K. Thus, $Sresults(K, q) = \sum_{n \in K} results(n, q)$.

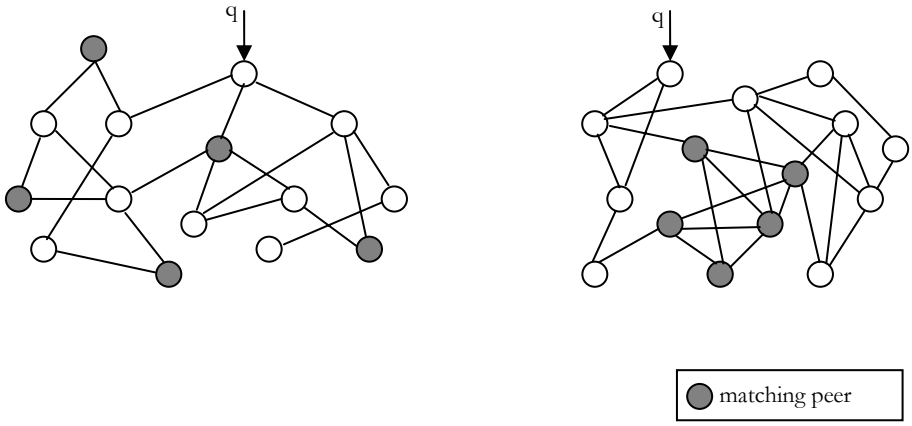
Definition 3.5 (PeerRecall) *Let W be a query workload. For each* $q \in W$*, let Visited(q, k)* $\subseteq N$ *be the set of k peers visited during the routing of a query q and Optimal(q, k)* $\subseteq N$ *be the set of k peers for the optimal routing of query q. We define* $PeerRecall_{(W)}$ *for the entire workload W as:*

$$Peer\ Recall_{(W)} = \left(\sum_{q \in W} f_q \frac{Sresults(Visited(q,k),q)}{Sresults(Optimal(q,k),q)} \right) \frac{1}{\sum_{q \in W} f_q}.$$

PeerRecall expresses the average percentage of results that the routing protocol can achieve, when visiting *k* peers, for each query *q* of the workload with respect to the optimal results of each *q*. Thus, each distinct query contributes equally to the evaluation of *PeerRecall* independently of the number of its results, meaning that a query which has a large number of data items satisfying it does not influence the estimation of *PeerRecall* more than a query with a small number of results. Intuitively, to increase *PeerRecall*, for the entire workload *W*, peers that have the most results for most queries of *W* should be close to each other in the overlay network.

3.3. Workload-Aware Clustering in Peer-to-Peer Systems

We are interested in creating a topology for a p2p system in the form of clusters. Intuitively, this kind of topology represents a number of smaller networks (clusters) of relevant peers that are rich in links between their peers. These smaller networks (clusters) are linked to each other through a few connections. Figure 3.1 illustrates a random p2p network and a clustered p2p network. To increase *PeerRecall* for a given workload *W*, groups of peers should be formed based on whether the peers match similar queries and have the most results for each one of them. Hence, peers that match a query *q* and have a large number of results for the query will be a few links away. Thus, once in the right group, i.e., when a query visits a peer with a large number of results, all relevant peers that also have many results are nearby. Besides the content of each peer, the type of the query workload should also be taken into account in forming the clusters. For example, consider keyword-value queries and the query workload $W = \{(x = 'A', 3), (x = 'B', 1)\}$. Consider three peers n_1, n_2 and n_3. Assume that peer n_1 has data items with values ‘A’ and ‘B’, peer n_2 with values ‘A’ and ‘C’ and peer n_3 with values ‘B’ and ‘D’. Most probably a non workload-aware clustering approach would consider peers n_2 and n_3 equally similar to peer n_1, since both share one common value with it. However, when taking into account the workload, peer n_2 is more “similar” to n_1, since n_1 and n_2 match a larger part of the workload than n_1 and n_3.



(a) (b)

Figure 3.1: (a) Random and (b) Clustered P2P Network.

In addition, a central issue in p2p systems is how the load induced by the queries is distributed across the peers of the network. In particular, when a query *q* is posed at a peer *n*, it travels through the p2p network to find matching results from the issuing peer *n* and all the other peers that this query message visits. Thus, each peer that *q* visits queries its content to find data items satisfying it. For a query workload *W*, the total number of queries that are issued to the p2p network is the same as the total number of query frequencies of the workload set. For example, assume that we have a query workload $W = \{(q, f_q)\}$, as defined in Definition 3.2, the total number of queries that are issued in the p2p network is $\sum_{q \in W} f_q$. We define the load for a peer *n*, denoted as *PeerLoad(n)*, as a performance measure that expresses the percentage of the total queries of the query workload set that peer *n* serves.

Definition 3.6 (PeerLoad) *Let* $W = \{(q, f_q)\}$ *be a query workload and N be a set of peers. For each peer* $n_i \in N$, *let* Q_i *be the bag of queries that arrive at* n_i. *We define the load for peer* n_i, *denoted PeerLoad*(n_i), *as:* $PeerLoad(n_i) = \frac{|Q_i|}{\sum_{q \in W} f_q}$.

In this thesis, we focus on measuring the performance of a p2p system based solely on the percentage of number of matching results returned for each query of the query workload (*PeerRecall*). Summarizing, an efficient clustering topology in a p2p system that is based both on the content of peers and the query workload must satisfy a number of properties so as to achieve an efficient grouping of similar peers, hence an efficient query routing. These properties are defined below:

Efficient Workload-Aware Clustering Properties

1. *Two peers, n_1 and n_2, which offer a similar number of matching results for a query q of the query workload must belong to the same cluster.*
2. *The clustering of peers must take into account the whole query workload. Hence, queries which are frequent influence further the formation of clusters compare to other queries that are issued seldomly.*
3. *Two peers that belong to the same cluster must be either immediately connected or nearby in the overlay network, i.e., the length of the shortest path that connects these two peers in the network must be very small. Thus, when a query q finds the appropriate cluster all the matching peers that contribute to the answer of the query will be a few links far away. In other words, any two peers lying in the same cluster tend to have a large number of paths connecting them, i.e., each cluster is rich in links between its peers.*
4. *The load that the queries bring must be uniformly spread among the peers of the system. Thus, each peer must serve roughly the same amount of queries. Load balancing is a desirable property that an efficient clustering should hold.*

3.4. Content-Based and Workload-Aware Distances

To create workload-aware overlays, we create clusters of peers that have similar content taking also into account a given query workload. Hence, the similarity between two peers depends both on their content and the query workload. We define the *overlay network distance* between two peers n_i and n_j, $dist(n_i, n_j)$, as the length of the shortest path from n_i to n_j in the p2p network. Ideally, to achieve an optimal query routing for each query of the workload, the desirable procedure would be for each query to re-cluster the peers so as to get an optimal clustering for the issuing query. Obviously, this procedure is non feasible for several reasons. The central issue for creating an efficient clustering is to define an efficient distance measure *(d)* between pairs of peers to represent how related two peers are, since the similarity of two peers will be descriptive of the overlay network distance between these two peers. Note that the workload-aware distance is different from the overlay network distance. At this point, we denote by *S(n)* the *size* of peer *n*, i.e., the total number of tuples (data items) that peer *n* stores.

As we have mentioned in Section 3.2.1, without loss of generality, the domain of the attribute *x* is $D = \{0, ..., M-1\}$. Thus, the relation *R* with a numeric attribute *x* of each peer *n* can be represented as an *M*-dimensional vector (index) with features $\vec{x}_n = \{x_{n,0}, x_{n,1}, ..., x_{n,M-1}\}$,

where the feature $x_{n,i}$ represents the frequency of value *i*, of peer *n* for the attribute *x*. Hence, each feature has a non-negative integer value. In the following sections, we discuss about several content-based distance measures that have been proposed and we consider alternative workload-aware distance measures to define the similarity of two peers for a given query workload *W*.

3.4.1. Content-Based Distance Measures

Several similarity (distance) measures have been used in the literature [7], [14], [38], [41] to capture the similarity (distance) between two vectors. A similarity measure $s \in [0,1]$ captures how related two vectors are and it can be mapped to a distance measure $d \in [0,\infty]$. In general, the distance between two vectors $\vec{x}_1$ and $\vec{x}_2$, $d(\vec{x}_1, \vec{x}_2)$, is a non-negative number that is close to 0 when the two vectors are highly similar or "near" each other and becomes larger the more they differ. We experimented with the most popular distance measures, which are described in this section. Assume that we have two peers n_1 and n_2 with vectors $\vec{x}_1$ and $\vec{x}_2$ of size $|D| = M$ respectively.

Distance Metrics

The *Minkowski distance*, also known as the L_p norm, is a very popular distance measure. Equation 3.1 suggests a definition of similarity between two peers, n_1 and n_2, based on Minkowski distance. For *p=1 (p=2)* we obtain the *Manhattan-City Block (Euclidean)* based similarity normalized to the [0, 1] interval, which is defined in equation 3.2 (3.3).

$$s_{Minkowski}(n_1, n_2) = \frac{1}{1 + \|\vec{x}_1 - \vec{x}_2\|_p} = \frac{1}{1 + \left(\sum_{i=0}^{M-1} |x_{1,i} - x_{2,i}|^p\right)^{1/p}} \tag{3.1}$$

$$s_{L_1}(n_1, n_2) = \frac{1}{1 + \|\vec{x}_1 - \vec{x}_2\|} = \frac{1}{1 + \sum_{i=0}^{M-1} |x_{1,i} - x_{2,i}|} \tag{3.2}$$

$$s_{L_2}(n_1, n_2) = \frac{1}{1 + \|\vec{x}_1 - \vec{x}_2\|_2} = \frac{1}{1 + \left(\sum_{i=0}^{M-1} |x_{1,i} - x_{2,i}|^2\right)^{1/2}} \tag{3.3}$$

The Minkowski, the Manhattan and the Euclidean distance measures are defined as follows:

$$d_{Minkowski}(n_1, n_2) = \|\vec{x}_1 - \vec{x}_2\|_p = \left(\sum_{i=0}^{M-1} |x_{1,i} - x_{2,i}|^p\right)^{1/p} \tag{3.4}$$

$$d_{L_1}(n_1, n_2) = \|\vec{x}_1 - \vec{x}_2\| = \sum_{i=0}^{M-1} |x_{1,i} - x_{2,i}| \tag{3.5}$$

$$d_{L_2}(n_1, n_2) = \left\| \vec{x}_1 - \vec{x}_2 \right\|_2 = \left(\sum_{i=0}^{M-1} \left| x_{1,i} - x_{2,i} \right|^2 \right)^{1/2} \quad (3.6)$$

In addition, an alternative distance measure is the *edit* distance. It has been shown that for *ordinal* measurements, as in our case, the edit distance is captured by the equation 3.7 [7]. In an ordinal measurement the values are ordered, e.g., the value of attribute *x* that each tuple has can be quantized into *M* integer values between 0 and *M-1*.

$$d_{edit}(n_1, n_2) = \sum_{i=0}^{M-1} \left| \sum_{j=0}^{i} \left(x_{1,j} - x_{2,j} \right) \right| \quad (3.7)$$

All the distance measures mentioned above satisfy the following mathematic requirements of a distance metric.

1. $d(n_1, n_2) \geq 0$: *The distance between two peers is a nonnegative number (Non-negativity).*
2. $d(n_1, n_1) = 0$: *The distance of a peer to itself is 0 (Reflexivity).*
3. $d(n_1, n_2) = d(n_2, n_1)$: *Distance is a symmetric function (Commutativity).*
4. $d(n_1, n_3) \leq d(n_1, n_2) + d(n_2, n_3)$: *The distance between peer n_1 and peer n_3 is no more than the sum of distances between (n_1, n_2) and (n_2, n_3) (Triangular inequality).*

Jaccard and Dice Similarity Measures

Many similarity functions are based on the inner product. In what follows, we refer to the most popular of this kind, the *Jaccard* and *Dice* similarity measures. In general, for binary features the *Jaccard coefficient* measures the ratio of the number of commonly active features of $\vec{x}_1$ and $\vec{x}_2$ to the number active in either $\vec{x}_1$ or $\vec{x}_2$. For example, given two vectors $\vec{x}_1 = (0,1,1,0)$ and $\vec{x}_2 = (1,1,0,0)$, the cardinality of their intersection is 1 and the cardinality of their union is 3, rendering their Jaccard coefficient 1/3. When used with real-valued features, as in our case, the *extended Jaccard coefficient*, also known as the *Tanimoto coefficient*, captures a length-dependent measure of similarity. The definition of the extended Jaccard coefficient in vector notation is given by Equation 3.8.

$$s_{Jaccard}(n_1, n_2) = \frac{\vec{x}_1^T \vec{x}_2}{\vec{x}_1^T \vec{x}_1 + \vec{x}_2^T \vec{x}_2 - \vec{x}_1^T \vec{x}_2} \quad (3.8)$$

Thus, the *Jaccard* distance measure for real-valued features, based on the extended Jaccard coefficient, is defined as:

$$d_{Jaccard}(n_1, n_2) = 1 - s_{Jaccard}(n_1, n_2) = 1 - \frac{\vec{x}_1^T \vec{x}_2}{\vec{x}_1^T \vec{x}_1 + \vec{x}_2^T \vec{x}_2 - \vec{x}_1^T \vec{x}_2} \quad (3.9)$$

It has been proven in [35] that the Jaccard distance also satisfies the mathematic requirements of a distance metric. Another similarity measure highly related to the extended Jaccard coefficient is the *Dice coefficient* which is defined in equation 3.10.

$$s_{Dice}(n_1, n_2) = \frac{2(\vec{x}_1^T \vec{x}_2)}{\vec{x}_1^T \vec{x}_1 + \vec{x}_2^T \vec{x}_2} \quad (3.10)$$

The Dice coefficient can be obtained from the extended Jaccard coefficient by adding $\vec{x}_1^T \vec{x}_2$ to both the numerator and denominator. Hence, it behaves very similar to the extended Jaccard coefficient. The *Dice* distance measure, based on Dice coefficient, is defined as:

$$d_{Dice}(n_1, n_2) = 1 - s_{Dice}(n_1, n_2) = 1 - \frac{2(\vec{x}_1^T \vec{x}_2)}{\vec{x}_1^T \vec{x}_1 + \vec{x}_2^T \vec{x}_2} \quad (3.11)$$

3.4.2. Workload-Aware Distance Measures

The distances mentioned above measure the similarity between two peers taking into account only their content. We would like to also take into account the query workload. Thus, we propose alternative distance measures so as the similarity between two peers depends both on their content and on the query workload. Consider a query workload set $W=\{(q, f_q)\}$ with $|W|$ distinct queries. The distribution of the query workload can be represented as a vector of size $|W|$ with features $\vec{w} = \{ p_{q_1}, p_{q_2}, ..., p_{q_{|W|}} \}$, where the feature $p_{q_i} = f_{q_i} / \sum_{j=1}^{|W|} f_{q_j}$ represents the probability of the distinct query q_i in W. In addition, as mentioned in Section 3.2.1, we symbolize with *results(n, q)* the number of matching results that peer n provides for the query q. Thus, for a given workload W, the matching results that a peer n provides for each distinct query q_i of W can be represented as a vector of size $|W|$ of the form:

$\vec{r}(n, W) = \{ results(n, q_1), results(n, q_2), ..., results(n, q_{|W|}) \}$.

The first two workload-aware distances are variations of the Manhattan-(L_1) distance and are based on the absolute difference of the number of matching results that two peers provide for a given query q. In particular, the *Manhattan Workload-Aware* distance measure between two peers, e.g., n_1 and n_2, computes for each distinct query q of the workload W, the absolute

difference between the number of matching results that the two peers provide and multiplies this factor with the probability p_q corresponding to q. Then all these quantities that are arisen by the distinct queries of the workload are added up. Definition 3.7 formulates the Manhattan Workload-Aware distance measure.

Definition 3.7 (Manhattan Workload-Aware Distance) *Given two peers n_1 and n_2 and a query workload $W = \{(q, f_q)\}$, we define the Manhattan Workload-Aware Distance as:*

$$wd_{L_1}(n_1, n_2, W) = \sum_{q \in W} p_q \left| results(n_1, q) - results(n_2, q) \right| = \vec{w}^T \left| \vec{r}(n_1, W) - \vec{r}(n_2, W) \right|$$

Another proposed distance that is based on the Manhattan distance is the *Size-weighted Manhattan Workload-Aware* distance. The Size-weighted Manhattan Workload-Aware distance between two peers, e.g., n_1 and n_2, arises in a similar way with the Manhattan Workload-Aware distance. The only difference is that the quantity produced by the Manhattan Workload-Aware distance is additionally weighted by dividing it with the sum of the *sizes* of the two peers n_1 and n_2. Thus, the number of tuples (data items) that each peer has is also taken into account for the computation of the distance between the two peers. Definition 3.8 formulates the Size-weighted Manhattan Workload-Aware distance.

Definition 3.8 (Size-weighted Manhattan Workload-Aware Distance) *Given two peers n_1 and n_2 and a query workload $W = \{(q, f_q)\}$, we define the Size-weighted Manhattan Workload-Aware Distance as:*

$$wd_{S\text{-}L_1}(n_1, n_2, W) = \frac{\sum_{q \in W} p_q \left| results(n_1, q) - results(n_2, q) \right|}{S(n_1) + S(n_2)} = \frac{\vec{w}^T \left| \vec{r}(n_1, W) - \vec{r}(n_2, W) \right|}{S(n_1) + S(n_2)}$$

Both of the previous mentioned distances are based on the number of results that each peer n provides for a query q, i.e., the number of results of peer n that match query q. An alternative distance that factors out the size of the query results, thus making the distance between two peers depends on the distribution of their results is the *Distribution-Based Manhattan Workload-Aware distance*. In particular, this kind of distance computes the sum of the absolute differences of the *normalized results* that the two peers have for each query q of the workload. By *normalized results*, we mean that the number of data items that a peer n provides for a query q are divided by the size $S(n)$ of peer n. Definition 3.9 formulates the Distribution-Based Manhattan Workload-Aware distance.

Definition 3.9 (Distribution-Based Manhattan Workload-Aware Distance) *Given two peers n_1 and n_2 and a query workload $W = \{(q, f_q)\}$, we define the Distribution-Based Manhattan Workload-Aware Distance as:*

$$wd_{D\text{-}L_1}(n_1, n_2, W) = \sum_{q \in W} p_q \left| \frac{results(n_1, q)}{S(n_1)} - \frac{results(n_2, q)}{S(n_2)} \right| = \vec{w}^T \left| \frac{\vec{r}(n_1, W)}{S(n_1)} - \frac{\vec{r}(n_2, W)}{S(n_2)} \right|$$

The Manhattan workload-aware distance measure and the Distribution-based Manhattan workload-aware distance measure satisfy the mathematical requirements of a distance metric. (Proofs in the Appendix).

Query Workload

We consider a query workload of range selection queries over an attribute *x*: $W = \{(q_{ij}, f_{q_{ij}})\}$, where $q_{ij} = \{x \in D : i \le x \le i + j\}$ and $f_{q_{ij}}$ its associated frequency. We denote by *i*, $0 \le i \le M - 1$, the value where the query starts and *j*, $0 \le j \le M - 1$, the query range, i.e., the number of contiguous values the query includes. Thus, a range query q_{ij} over an attribute *x* asks for data items that have value for *x* within the range [i, i+j]. Note that for a query q_{ij} if $i + j > M - 1$ then $q_{ij} \equiv q_{i(M-1-i)}$, i.e., the range of values that the query asks for is [i, M-1].

3.5. Discussion

In this section we discuss about which type of cluster is created by each of the distance measures, i.e., for each distance measure which peers are grouped together to form a cluster and how efficient is this kind of clustering.

3.5.1. Content-Based Distance Measures

As we have mentioned before, we use a numerical domain *D* for the attribute *x*. Thus, there exists an ordering among the values of the domain *D*. For example, for our value domain $D = \{0, ..., M-1\}$ the values 0, 1, 2, …, M-1 correspond to the 1st, 2nd, …, Mth dimension of the vector-index that represents the relation of a peer. In finding the distance between two vectors of ordinal type measurements, the ordering of the values plays a crucial role when the query workload consists of range queries. Thus, we want a distance measure to take into account this ordering. We say that a distance that takes into account the ordering satisfies the *shuffling*

dependence property. The importance for a distance measure to satisfy the shuffling dependence property becomes clear by the following example:

Example 3.1**:** *Consider three vectors* $\vec{x}_1$, $\vec{x}_2$ *and* $\vec{x}_3$ *that represent the relations of the peers* n_1, n_2 *and* n_3 *respectively. We assume a value domain D = {0, ..., 7} of size 8 and that the number of tuples in each relation is 5. For all tuples of relations* $\vec{x}_1$, $\vec{x}_2$ *and* $\vec{x}_3$, *the attribute x has the value 1, 2 and 7, respectively.*

$\vec{x}_1 = [0\ 5\ 0\ 0\ 0\ 0\ 0\ 0]$,
$\vec{x}_2 = [0\ 0\ 5\ 0\ 0\ 0\ 0\ 0]$,
$\vec{x}_3 = [0\ 0\ 0\ 0\ 0\ 0\ 0\ 5]$.

The distance between n_1 *and* n_2, *that have all their values at adjacent dimensions in* $\vec{x}_1$ *and* $\vec{x}_2$ *(dimensions 1 and 2 respectively), should be smaller than the distance between* n_1 *and* n_3, *that have all their values at dimensions further apart in* $\vec{x}_1$ *and* $\vec{x}_3$ *(dimensions 1 and 8 respectively). This is because, the difference between the number of results provided by peer* n_1 *and the number of results provided by peer* n_2 *is smaller for a large number of range queries than for peers* n_1 *and* n_3.

As shown, the shuffling dependence property is not satisfied by the Manhattan, Euclidean and Jaccard distance measures because they are sums of individual distances for each value of the value domain. In particular, for our example the three vectors have the same pair-wise distance, $d(n_1, n_2) = d(n_1, n_3)$, for each of the three metrics that mentioned before. Hence, these three distance measures are appropriate only for keyword-value queries, i.e., queries of the form: $q_i = \{ x \in D : x = i \}$.

In contrast, for the edit distance, the shuffling dependence property holds. In particular, the edit distance between two vectors, $\vec{x}_1$ and $\vec{x}_2$ is the total number of necessary minimum movements for transforming $\vec{x}_1$ to $\vec{x}_2$ by moving elements to the left or right. Edit distance takes the sum of absolute values of prefix sums of difference for each value of the value domain. In our example, using the edit distance measure, the pair-wise distances between peers n_1, n_2 and n_3 are:

$$d_{edit}(n_1, n_2) = 0 + 5 + 0 + 0 + 0 + 0 + 0 + 0 = 5$$
$$d_{edit}(n_1, n_3) = 0 + 5 + 5 + 5 + 5 + 5 + 5 + 0 = 30$$

Hence, $d_{edit}(n_1, n_2) < d_{edit}(n_1, n_3)$. Thus, using the edit distance the network distance between n_1 and n_2 must be smaller than the network distance between n_1 and n_3, since the similarity of two peers is descriptive of the overlay network distance, which is the ideal.

3.5.2. Workload-Aware Distance Measures

The Workload-Aware distances measures that introduced before (Definitions 3.7-3.9) create the same kind of clustering when the size of all peers, i.e., the total number of tuples (data items) that each peer stores, is the same. For non-equal sized peers, the workload-aware distance measures, defined in Section 3.4.2, achieve different kinds of clustering due to the way they measure the similarity between pairs of peers. In what follows, we analyze the kind of clustering that each distance metric creates. At this point, we introduce the notion of *"fat"* and *"thin"* peers. In particular, we denote as *"fat"* peers those peers having a larger number of data items compare to the size of the rest of the system peers, which we denote them as *"thin"* peers.

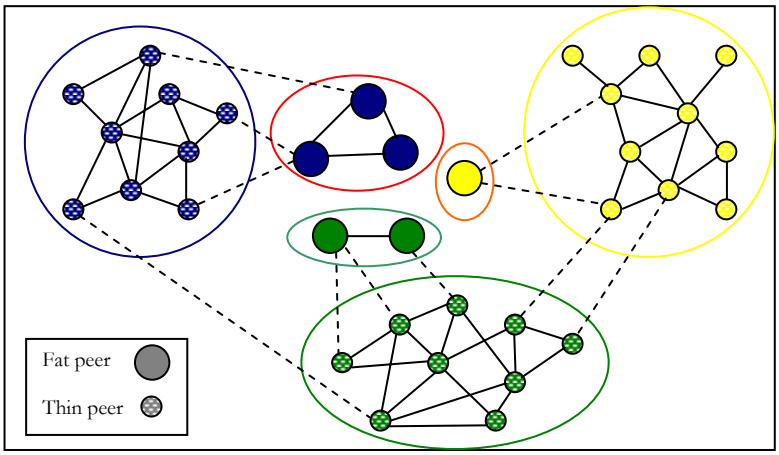


Figure 3.2: Example of Clustering Using the Manhattan Workload-Aware Distance Measure.

Consider that we have a p2p system and three different groups of peers where all peers in the same group follow the same distribution for the attribute *x*. In addition, the peers in each group do not have the same size, i.e., we have "fat" and "thin" peers in the same group. Also, we assume a query workload *W* that consists of range queries. In Figures 3.2 - 3.4, we depict graphically how these peers are clustered using the three workload-aware distance metrics. In particular, the peers that follow the same data distribution for *x* have the same color density, while we distinguish the fat peers from the thin peers by their sizes, i.e., the fat peers are illustrated larger than the thin peers.

As we have mentioned before, the Manhattan workload-aware distance measures the absolute difference between the number of matching results that a pair of peers provides for each query of the workload *W*. Thus, the distance between two peers is very small if they have roughly the same size of matching results for the entire query workload and is independent of the size of the peers. In Figure 3.2, we depict the kind of clustering that this distance measure achieves for the p2p system that mentioned before. In particular, all the thin peers that follow the same data distribution for *x* belong to the same cluster (group of peers) that is rich in links between their peers. Similarly, all fat peers with the same data distribution are also grouped together. Thus, each cluster consists of peers that follow the same distribution for their data and also have the same size. Furthermore, these clusters are linked to each other through a few connections.

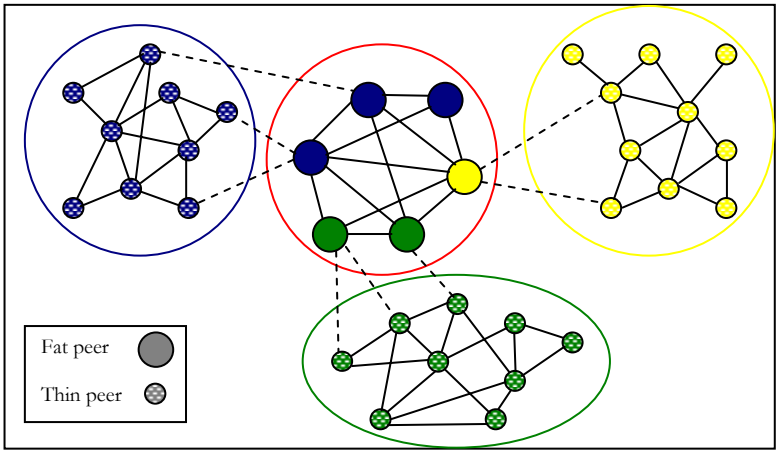


Figure 3.3: Example of Clustering Using the Size-Weighted Manhattan Workload-Aware Distance Measure.

In contrast, the size-weighted Manhattan workload-aware distance favors fat peers since the absolute difference of results that two peers provide for a query *q* is divided by the sum of the sizes of the two peers. In Figure 3.3, we depict the kind of clustering that this distance measure achieves for our example. In particular, all the fat peers are clustered in the same group, while all the thin peers that follow the same data distribution are also grouped together. The main drawback of this distance, which makes it inefficient, is that even if two peers have a large difference in the number of results for a workload *W,* hence we would expect having a large distance, by dividing them by their sum of their sizes their distance could become very small.

Example 3.2: *Consider three peers n_1, n_2 and n_3 with sizes 10000, 1000 and 100 respectively and a query workload W consisting of a single query q. In addition, assume that peers n_1, n_2 and n_3 provide 5, 500 and 50 results for the query q, respectively and the query visits two of the peers. Suppose that initially the query message finds the peer with the most results for the query, i.e., peer n_2. Ideally, the query message should visit peer n_3, since it provides the next best number of results for query q. Clearly, the optimal results for the query are 550 results. In contrast, in the clustering based on the size-weighted manhattan workload-aware distance measure, the network distance between peers n_2 and n_1 is smaller than the network distance between n_2 and n_3. Hence, the query message visits peers n_2 and n_1 and gets only 505 results.*

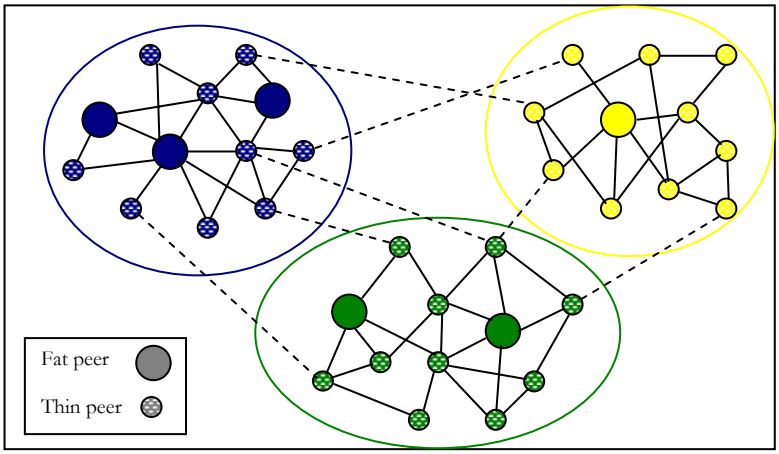


Figure 3.4: Example of Clustering Using the Distribution-Based Manhattan Workload-Aware Distance Measure.

In addition, with the distribution-based Manhattan workload-aware distance measure, the clustering of peers is based solely on their data distributions; hence, the size of matching results that each peer *n* provides for each query *q* of the workload is factored out. In Figure 3.4, we depict the kind of clustering that this distance measure achieves. In particular, all thin and fat peers that follow the same distribution belong to the same cluster. The main drawback of the distribution-based workload-aware distance measure is that each cluster contains peers that follow the same distribution. Thus, the distance between a thin and a fat peer which follow the same distribution is 0 although the fat peer provides a larger number of results for each kind of query workload.

Example 3.3: *Consider three peers n_1, n_2 and n_3 with sizes 10000, 1000 and 100 respectively and a query workload W consisting of a single query q. The three peers n_1, n_2 and n_3, follow the same distribution for their data items and they provide 5000, 500 and 50 results for the*

query q respectively. Assume that the query visits two of the peers and initially the query message finds the peer with the most results for the query, i.e., peer n_1. Ideally, the query message should visit peer n_2, since it provides the next best number of results for query q. Clearly, the optimal results for the query are 5500 results. In contrast, in the clustering based on the distribution-based manhattan workload-aware distance measure the network distance between the three peers is the same. Hence, the query message might visit peer n_3 and get only 5050 results.

3.6. Experimental Evaluation

In this section, we evaluate the performance of the distance measures experimentally. In particular, we examine the quality of clustering that these distance measures achieve from the perspective of *PeerRecall* for different kinds of query workloads. We consider a number $|N|$ of peers. Each peer stores a relation *R* with a numeric attribute *x* that follows a data distribution. In addition, we consider a query workload *W* consisting of queries on *x* that follows its own distribution. Initially, we select one of the distance measures and calculate the distances that each peer *n* has from all other peers of the network. Then, we simulate a peer-to-peer overlay network with a directed graph based on the distances that we have found before. For example, if we have three peers, n_1, n_2 and n_3, and the distance between n_1 and n_2 is smaller that the distance between n_1 and n_3, ($d(n_1, n_2) < d(n_1, n_3)$), then the overlay network distance between n_1 and n_2 is smaller that the overlay network distance between n_1 and n_3 ($dist(n_1, n_2) < dist(n_1, n_3)$).

After the construction of the peer-to-peer network, we pose to it the queries of *W*. In particular, for each query *q* of *W* we initially select the peer, e.g., *n*, that provides the most results for *q*. We make this strong assumption so as to focus on the effect of clustering that if a query message starts from the peer that has the largest number of results for *q* in the network, then all other peers that also provide enough results for *q* are few links away. After posing the query to peer *n*, the query message is forwarded to a specified number (*k-1*) of other peers according to the topology of the p2p network that we have created. For each query *q*, we measure the total number of results returned and the optimal number of results returned visiting *k* peers, in the case of optimal query routing. Finally, we measure *PeerRecall* for the entire query workload *W*. The same experiment is also carried out when the p2p system is randomly constructed, i.e., each query of the query workload is forwarded to *k* randomly selected peers of the network.

3.6.1. Experimental Parameters

We run a set of experiments to evaluate the performance of the distance measures. The value domain for the attribute x is D = [0, 999] and the parameter k varies from 5% to 80% of the network peers. Furthermore, we distinguish the set of experiments into two categories. In the first category, we assume that all peers have equal size, whereas in the second category all peers do not have equal size. In particular, a percentage of the network size, $|N|$, are "fat" peers and all the other peers are "thin". We make this distinction because when all peers have the same size the workload-aware distances create the same clustering, thus the p2p network topology is the same and thus the estimation of *PeerRecall* for each kind of query workload is the same. In contrast, when all peers do not have the same size, the clustering of peers are not identical because some of the workload-aware distances take into account the size of the peers.

3.6.1.1. Peer Data Distribution

We consider that each peer in the network follows a data distribution for the attribute x. In particular, we divide the value domain D of x into a number of disjoint regions of equal width, denoted as Dr. For each peer a fraction of its tuples, denoted as DC (*Data Concentration*), falls into two of the Dr regions of the value domain, which are selected randomly, and the rest of the tuples are distributed uniformly among the rest of the values. We assume that each one of the two regions has ($DC/2$) of the tuples.

The intuition behind our choice for this data distribution is that typically a user in a p2p system is not interested in all the available data. Usually, each user is interested just only in one or two topics. For example, a user may be interested in songs that were released or popular in the 80s or 70s and not earlier. Thus, in our distribution we simulate the concentration (DC) of a user's interest in particular values (Dr) of the data domain.

In the case of peers having the same size, we assume that each peer has different data distribution from all the other peers of the network. We make this strong assumption because the quality of clustering that each distance measure provides becomes much more evident in this way. For example, if we have a set N_i of peers that follow the same data distribution, then the distance between them is 0, hence all these peers in N_i are nearby in the overlay network. Thus, for a query of the workload and for a small number of peers visited when we initially select the peer $n_i \in N_i$ that has the largest number of results, all other peers in N_i will be

immediately visited and the *PeerRecall* will be high. Hence, we cannot clearly investigate where all the other peers in the network with different data distributions, e.g., $n_j \notin N_i$, and probably with similar number of query results are placed in the network. Note, that we are especially interested for each query to visit only a small fraction of the network. In contrast, in the case when each peer has different data distribution, after visiting the peer with the largest matching results we can see directly which other peers with different data distributions are nearby in the network and if they also provide large number of results for the query. Figure 3.5 depicts an example of this data distribution with 6 disjoint regions and 3 peers, n_1, n_2 and n_3. In our experiments, we use 100 peers to create the p2p network and we assume that each peer *n* has 10000 tuples. Furthermore, we set the parameters *Dr* and *DC* equal to 200 and 0.8, respectively. To create 100 different data distributions we select each peer $n \in N$ to have different regions, where the majority of its tuples falls into, from all the other peers of the system.

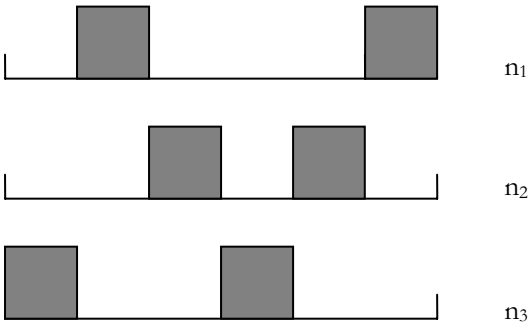


Figure 3.5: Example of our Data Distribution with 6 Disjoint Regions and 3 Peers.

We also deal with the case where the peers do not have equal size. In this case, we assume that the network has 500 peers and more peers than one follow the same distribution. The reason for this is that two peers with the same distribution but different sizes may give fewer results for a query workload when compared with two peers that follow different data distributions. Thus, the distance between the two peers with different data distributions must be smaller than the distance of the peers that follow the same one. In other words, the size of a peer plays a crucial role on how the peers will be clustered. In our experiments, we create 50 different data distributions in a similar way that we have mentioned before and for each one of them, 10 peers follow the same distribution. Hence, we created 50 clusters of peers. Recall that in each cluster the selected regions, where the majority of its tuples falls into, are different from all the other clusters. In addition, in each cluster 10% of the peers are “fat”,

while the rest of them are "thin". We experimented with two cases. In the first one, the thin peers have 1000 tuples and in the second one 10000 tuples, while in both cases the fat peers have 100000 tuples.

3.6.1.2. Query Workload Distribution

As we have mentioned before, we deal with query workloads, *W*, that consist of range selection queries. In addition, we consider that the starting point of the queries is chosen uniformly from the value domain *D* whereas their ranges vary according to a *Zipf* [13], [42] distribution. The main characteristic of the Zipf distribution is that there are a few values with high frequencies and many with low frequencies. In general, in Zipf distribution there is a ranking of the values of the data domain and the probability of a value *i* with rank *r* is analogous to $p_i = 1/r^z$, where *z* is a parameter which determines the skewness of the distribution. If the parameter *z* is set to 0, the Zipf is the same with the uniform distribution but as *z* increases, the skewness of the distribution increases accordingly. For a relation that has *T* tuples and a value domain *D* of size $|D|$, the frequency of value *i* with rank *r*, generated by the Zipf distribution is $f_r = T \frac{1/r^z}{\sum_{r=1}^{|D|} 1/r^z}$ for $1 \le r \le |D|$.

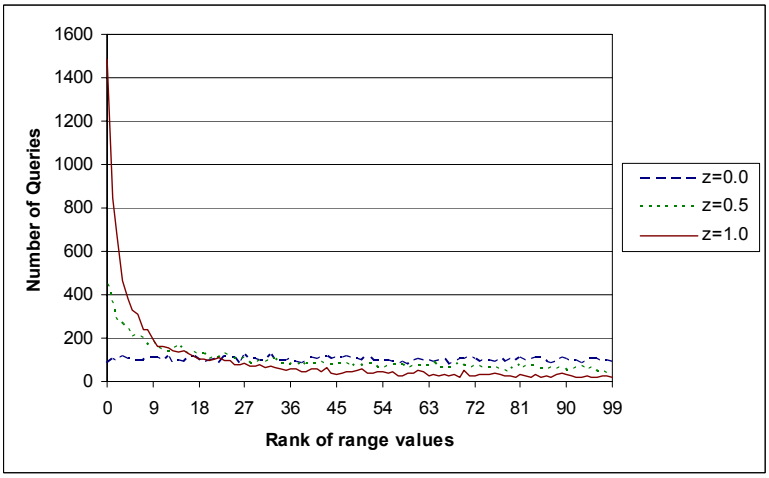


Figure 3.6: Zipf Frequency Distribution.

In our experiments, the ranking of the ranges is done as follows: Assume that the range with value *i* is the most popular one, denoted as *hot query range* (*Hqr*). Then, the popularity reduces as the distance between the range *i* and the other ranges increases. Hence, in our case,

the query workload *W* can be characterized by two parameters: the hot query range (*Hqr*) which determines which range is the most popular in *W* and the *z* parameter that determines for each range the frequency of the queries in the workload that have the specified range.

Figure 3.6 is a graphical representation of the frequency distribution of the query range, which follows the Zipf distribution with 0 being the most popular range, i.e., keyword-value queries, and for several values for the parameter *z*. The x-axis represents the rank of the query range value, which in this case coincides with the query range value, with respect to its associated frequency of queries. In this example, the query workload consists of 10000 queries and the domain of the query range is [0, 99]. In our experiments, we consider a query workload which consists of 10000 queries and the *z* parameter that determines the frequency of a query, i.e., how many queries in *W* have the specified range, varies from 1.0 to 3.0.

The input parameters for the first set of experiments are summarized in Table 3.1. We divide the input parameters into three categories: *Peer-to-Peer* parameters, *Data distribution* parameters and *Query workload* parameters.

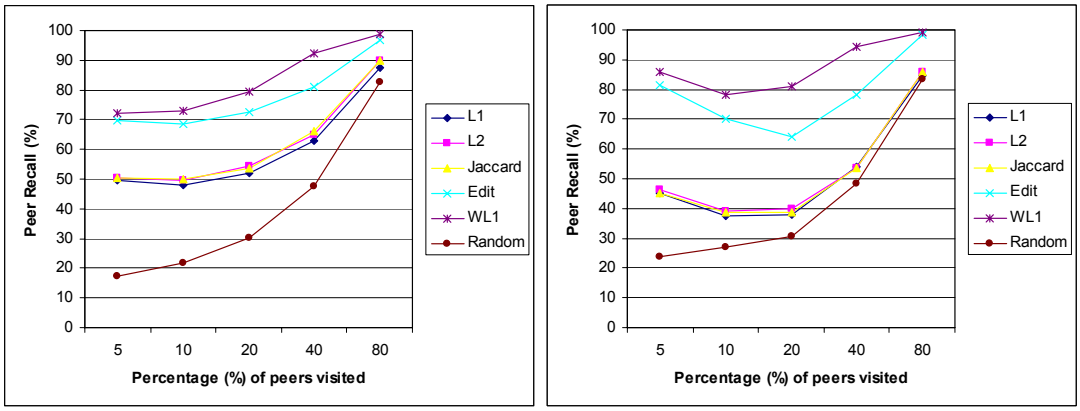
Table 3.1: Input Parameters for the P2P Network and the Query Workload *W*.

Parameter	**Default Value**	**Range**
Peer-to-Peer Parameters		
Number of peers ($\|N\|$)	100	100 - 500
Percentage (%) of peers visited during routing		5 - 80
Percentage (%) of "fat" peers		0 - 10
Data Distribution Parameters		
Domain of *x*	[0, 999]	
Tuples per node	10000	1000 - 100000
Data Concentration (*DC*)	0.8	
Number of disjoint regions (*Dr*)	200	100 - 200
Query Workload Parameters		
Number of queries	10000	
Range of queries	[0, 999]	
Zipf parameter (*z*)		1.0 - 3.0
Hot query range (*Hqr*)		10 - 100

3.6.2. Performance Evaluation of Distance Measures in the Case of Equal Peer Sizes

In this section, we present experiments in the case of equally-sized peers. We evaluate the performance of clustering that all the distance measures achieve, using as our performance

measure *PeerRecall* (as defined in Definition 3.5), for several kinds of query workloads, i.e., varying the "hot" query range and the parameter z. In Figure 3.7, we demonstrate the *PeerRecall* we achieve when each query q in W visits a specified number of peers in the network when for the query workload W the parameter z takes the value 1.0 and the "hot" query range is 10 (Fig. 3.7(a)) and 100 (Fig.3.7(b)) respectively. In addition, in Figure 3.8 we demonstrate the same experiment but for $z = 3.0$ while the "hot" query range takes value 10 (Fig. 3.8(a)) and 100 (Fig.3.8(b)). Recall that increasing the parameter z, the frequency of queries in the workload that have as range the "hot" query range increases too. Thus, when the "hot" query range takes the value 10 and as z increases the majority of the queries in W have small ranges, while setting the "hot" query range to 100 the most frequent queries are those with large ranges. Since all peers have equal size, all workload-aware distance measures result in the same clustering of peers for each kind of query workload; hence *PeerRecall* is the same. Thus, we demonstrate only the performance of the Manhattan workload-aware distance measure, denoted as WL_1. Furthermore, we symbolize with L_1 and L_2 the Manhattan and the Euclidean distance measures, respectively.



(a) (b)

Figure 3.7: Comparison of Distance Metrics when Varying the Number of Peers Visited with $z = 1.0$ for (a) $Hqr = 10$ and (b) $Hqr = 100$.

Obviously, for all the distance measures, when each query visits a large number of peers in the network, i.e., similar with flooding the network, the *PeerRecall* is high, since for each query, there is a high probability to find all relevant peers with large number of matching results. Thus, we are more interested in measuring *PeerRecall*, when each query visits a small fraction of the network.

As expected, the *Manhattan* (L_1), the *Euclidean* (L_2) and the *Jaccard* distance measures do not perform well, since they compare only the absolute difference between individual values without taking into account the neighboring values which however influence the behavior of queries with ranges larger than 0. In particular, because of the nature of the data distribution, the L_1 (L_2, *Jaccard*) distance between a peer *n* and all peers of the network is the same since L_1 (L_2, *Jaccard*) is not shuffling dependent and considers only individual values. Hence, these distance measures perform well only for queries with range 0, i.e., keyword-value queries. In contrast, the edit distance performs better than the L_1 (L_2, *Jaccard*) distance, since it is shuffling dependent.

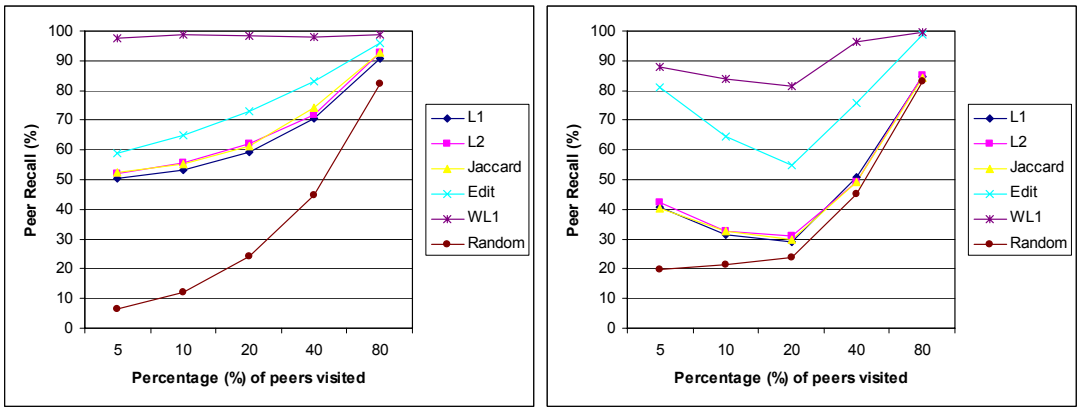


Figure 3.8: Comparison of Distance Metrics when Varying the Number of Peers Visited with $z = 3.0$ for (a) $Hqr = 10$ and (b) $Hqr = 100$.

However, the edit distance between two peers takes into account the ordering of all values, while a query with range *j* involves only (*j+1*) values and thus it should not depend on the difference that the two peers may have in the rest of their values. Hence, we expect the edit distance to perform well for query workloads where the most popular queries are those with large ranges. Indeed, as we can see in Figures 3.7(b) and 3.8(b) where the most popular queries in the workload are those whose range is 100, the *PeerRecall* achieved by the edit distance is much higher than when the majority of the queries have low ranges, e.g., Figures 3.7(a) and 3.8(a) where the most popular query range is 10. This observation is much more evident in Figure 3.8 where the parameter *z* takes the value 3.0, hence the skewness of the range distribution is very high. In this case, when the *Hqr* is small, e.g., 10, the performance of the edit distance is not as good and is roughly similar to the L_1, L_2 and *Jaccard* distances (Fig. 3.8(a)). In contrast, the *PeerRecall* is very high when the *Hqr* is large (Fig. 3.8(b)).

For the workload-aware distance measures (WL_1) the *PeerRecall* they achieve is very high for all kinds of query workloads, hence the clustering that these measures provide is efficient and the average number of results returned for each query in W is close to optimal.

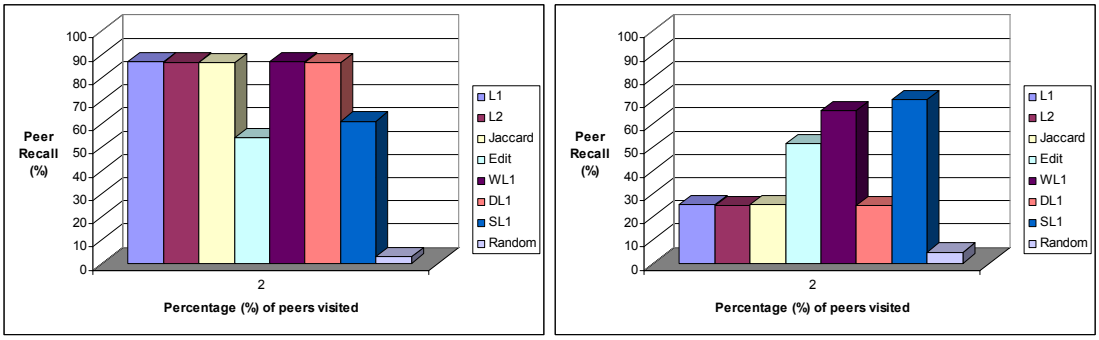
Finally, as we expected when we have a randomly constructed p2p network, the performance of the *PeerRecall* is very low for each kind of query workload. This happens because each peer is connected with a randomly selected peer and not with a peer that provides similar results for the query workload. Hence, the peers that a have large number of results for a query q might have a large network distance from the peer that has the most results for q, which we initially select to start the query routing.

3.6.3. Performance Evaluation of Distance Measures in the Case of Non Equal Peer Sizes

In this section, we investigate how the distance measures perform when the peers do not have equal sizes. In particular, we are especially interested in comparing the clustering achieved when using the Manhattan Workload-Aware Distance (WL_1), the Size-weighted Manhattan Workload-Aware Distance (SL_1) and the Distribution-based Manhattan Workload-Aware Distance (DL_1) for different kinds of query workloads. We measure *PeerRecall* for each of the distance measures when for each query q of W, we visit 2% of the peers. We set the parameter k equal to 2% of the network size since what is most important in this experiment is to see whether all relevant peers that provide the largest number of results for W are placed nearby in the network or not.

As we have mentioned in Section 3.5.2, the WL_1 distance measure clusters the peers based on the difference of their results for a given workload W. Thus, two peers that provide a similar number of results for W will be placed nearby in the network, whereas two peers with large difference in their results will be far away. Note, that this distance metric measures the difference of results between pairs of peers for all the queries of W. Hence, the clustering that WL_1 provides might be not optimal for each query $q \in W$, meaning that the difference of results for W might be very small for a pair of peers but not necessarily so for each distinct query q of the workload. The DL_1 distance clusters peers based on their data distributions, i.e., all peers that have similar data distributions for x will be placed nearby in the network. Finally, the SL_1 distance measure leads in placing fat peers nearby in the network even if they have large difference in their results for W.

In this set of experiments, we deal with two cases. In the first case, we consider that the size of the thin peers is 10000 and in the second one that this size is 1000, while in both cases the size of the fat peers is 100000 tuples. Furthermore, we set the skewness of the query range distribution (z) equal to 3.0, i.e., the query workload almost consists of queries with a specified *Hqr*.



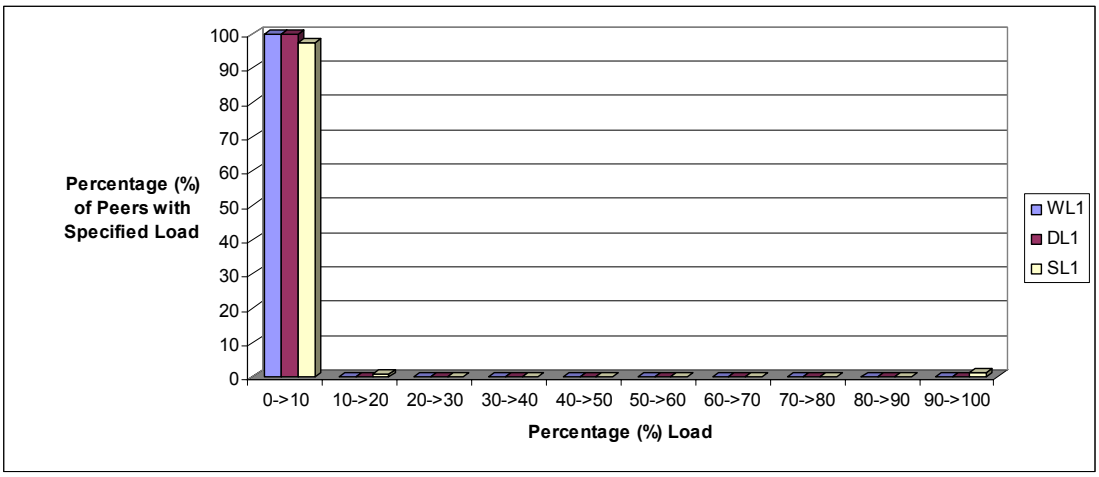
(a) (b)

Figure 3.9: Comparison of Distance Metrics in the Case of Non-Equal Peer Sizes when the Number of Peers Visited is Set to 2% of the Network's size and the Query Workload consists of Range Queries with (a) *Hqr* = 10 and (b) *Hqr* = 100, while z = 3.0. The Size of the "thin" and the "fat" Peers is 10000 and 100000 tuples, respectively.

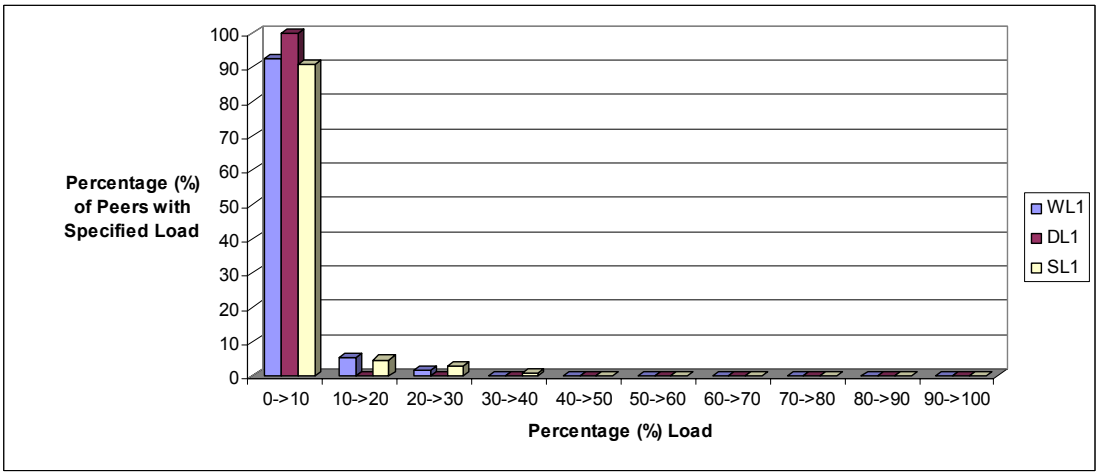
Initially, we deal with the case where the thin peers have 10000 tuples. In Figure 3.9, we demonstrate the performance of all the distance measures when the majority of queries have low ranges (Fig. 3.9(a)) and high ranges (Fig.3.9(b)) respectively. As we can see in the case when the most popular queries are those with low range, i.e., *Hqr = 10*, the DL_1 and WL_1 distances performs well, while SL_1 does not. This happens when we have a query workload with small range queries, the query q initially selects the fat peer, e.g. n, which provides the most results for q, all the thin peers that follow the same data distribution with n provide also the most results for q in comparison with all other peers. Hence, placing peers with similar data distributions nearby in the network is ideal in this case. In contrast, when the majority of W's queries have a large range (Fig.3.9(b)), i.e., asking for a larger number of contiguous values, the fat peers of the network provide larger number of results, hence placing them nearby in the network is desirable. That is why in this occasion the DL_1 performs badly and SL_1 performs efficiently. Note that in both cases the WL_1 distance performs well.

Figure 3.11 demonstrates the performance of the distance measures when the thin peers have 1000 tuples while the fat peers have 100000 tuples. In particular, when the query workload nearly consists of either queries with small ranges or queries with large ranges, the fat peers

provide a larger number of results than the thin peers, independently of the data distribution that each peer follows, due to the small size of the thin peers. Hence, the SL_1 performs well, while the DL_1 has low performance especially when the query workload consists of queries with high ranges. The WL_1 performs well in this case too.



(a)

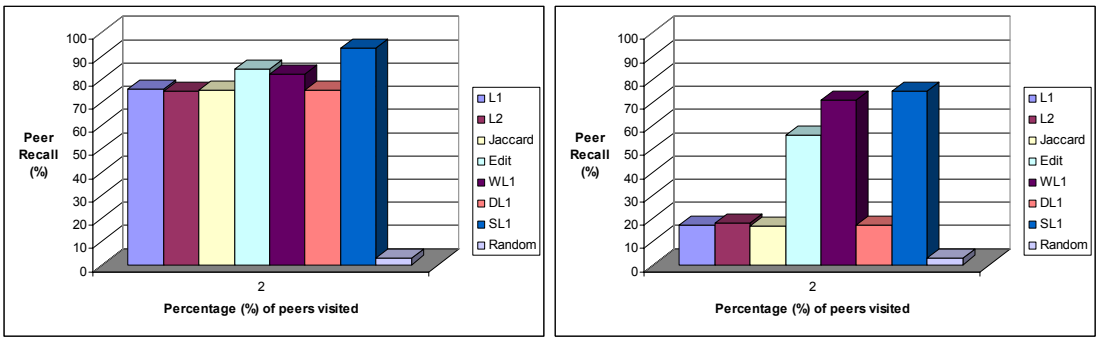


(b)

Figure 3.10: Peers Load when the Clustering is Done based on the three Workload-Aware Distance Measures and the Query Workload consists of Range Queries with (a) *Hqr* = 10 and (b) *Hqr* = 100, while *z* = 3.0. The Size of the "thin" and the "fat" Peers is 10000 and 100000 tuples, respectively.

We also examine the influence of the clustering that the three workload-aware distance measures achieve in the load that the peers receive. In particular, the load of each peer, i.e., the number of queries that a peer receives, depends on the way that the peers are clustered, according to the workload-aware distance measure that we use. Ideally, we would like the

load that the queries bring to be uniformly spread among the peers of the system. In Figure 3.10, we demonstrate the percentage of network peers that receive a percentage of queries of the query workload (*PeerLoad*) for each one of the three workload-aware distance measures when the size of the thin peers is 10000 tuples and for query workloads that consists of queries with most frequent range 10 (Fig.3.10(a)) and 100 (Fig.3.10(b)), respectively. Similarly, we conducted the same experiment also for the case when the size of the thin peers is 1000 tuples (Figure 3.12). As we expected, using the DL_1 distance measure for clustering the peers leads in a unifom spread of the queries to the network (all the peers serves from 0 to 10% of the workload's queries), since this kind of distance metric is idependent of the size of the peers; hence it clusters peers with similar data distributions. In contrast, using the SL_1 workload-aware distance, some peers of the network receive a large ratio of query messages. We expect that these peers are the fat peers. This is not desired at all because on the one hand the fat peers receive a very large number of query messages and on the other hand, as we discussed in a previous section, these peers are very likely to not have a large number of matching results for a query. Finally, the load that the WL_1 distance measure brings to the peers is an intermediate situation between these two that we described before, i.e., the load is not distributed equally among peers but it is not accumulated to some peers either. Some peers might receive a large number of queries but also they provide a large number of matching results for the query workload.

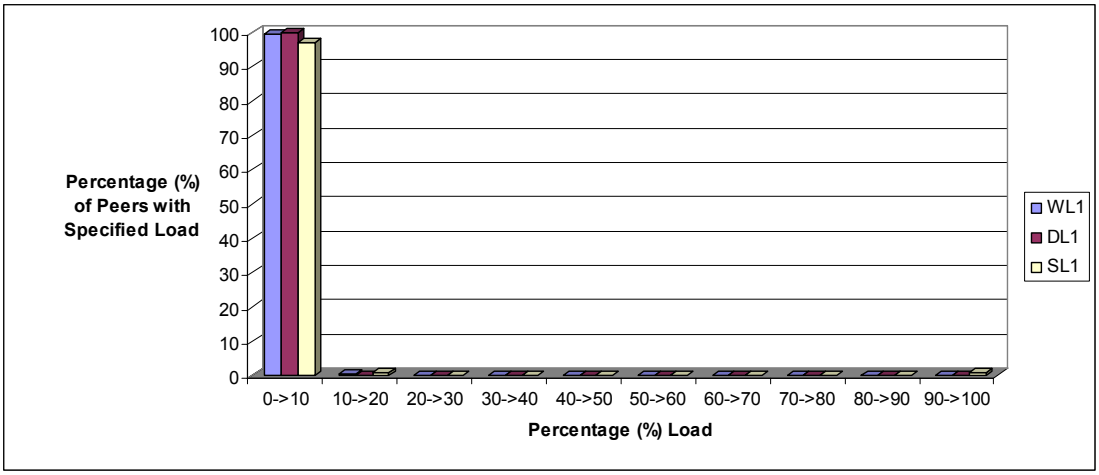


(a) (b)

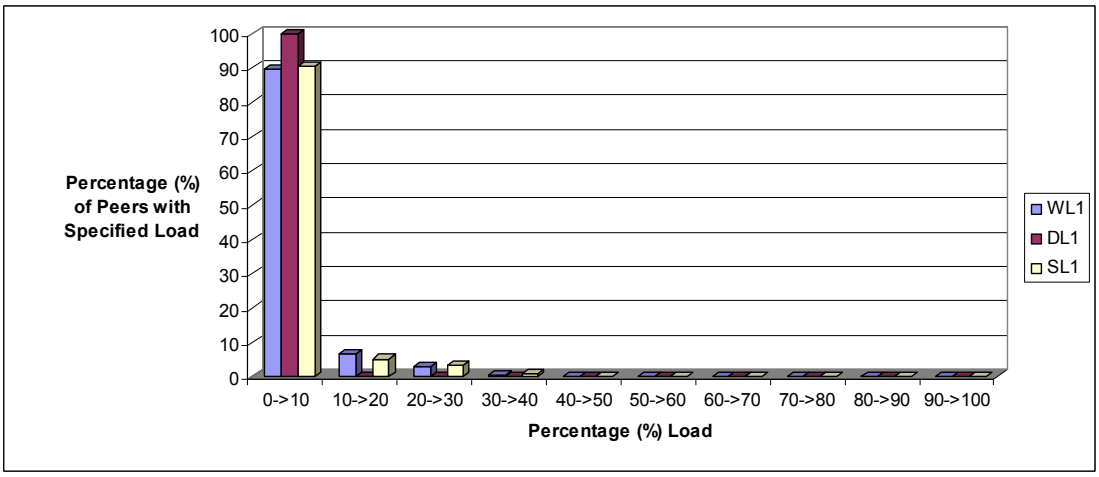
Figure 3.11: Comparison of Distance Metrics in the Case of Non-Equal Peer Sizes when the Number of Peers Visited is Set to 2% of the Network's size and the Query Workload consists of Range Queries with (a) *Hqr* = 10 and (b) *Hqr* = 100, while *z* = 3.0. The Size of the "thin" and the "fat" Peers is 1000 and 100000 tuples, respectively.

Summarizing, there are occasions where the DL_1 performs well and others where its performance is very low. The same conclusions also hold for the SL_1 distance. In contrast, the WL_1 seems to perform well in every case. Hence, the conclusions of this experimental

evaluation leads us to select the WL_1 distance metric as the most efficient distance metric for clustering the peers. Furthermore, for simplicity of presentation in the following sections we consider equal-sized peers. This assumption does not influence the system's behaviour since the clustering that the WL_1 distance metric provide is efficient in both cases when we have equal-sized peers or non-equal ones.



(a)



(b)

Figure 3.12: Peers Load when the Clustering is Done based on the three workload-aware distance measures and the Query Workload consists of Range Queries with (a) Hqr = 10 and (b) Hqr = 100, while z = 3.0. The Size of the "thin" and the "fat" Peers is 1000 and 100000 tuples, respectively.

Finally, we have to note that using the whole information of peers content and the query workload for estimating the similarity between pairs of peers leads in high computational cost. In the following sections, we propose more lightweight procedures, based on the

summarization of both the content of the peers and the query workload, to measure the similarity of peers.

3.7. Summary

To conclude, in this chapter we propose building workload-aware overlays where the grouping of similar peers is not based solely on the content of the peers but also on the query workload. Hence, the central issue for creating an efficient clustering is to define an efficient distance measure between pairs of peers to represent how related the two peers are. Initially, we present several popular content-based distance measures that have been proposed in the literature, e.g., Manhattan, Euclidean, Edit and Jaccard distance measures, and then we propose three workload-aware distance measures, the Manhattan workload-aware distance, the Size-weighted Manhattan workload-aware distance and the Distribution-based Manhattan workload-aware distance. In addition, we make an extensive discussion about how these distance measures create clusters of peers, i.e., which peers are grouped together to form a cluster, and we refer to the main drawbacks of each one of them.

Furthermore, we experimentally evaluate the efficiency of the distance measures. In particular, we simulate a p2p network as a directed graph and we cluster the peers using each time one of these distance measures. We characterize their performance by measuring the PeerRecall they achieve for a given query workload. The main conclusions from our experiments show that the workload-aware distance measures perform better than the content-based ones. In the extreme case, the average results returned for a query (PeerRecall), using the workload-aware distance measures are 50% more than those using the content-based ones. In addition, when the size of the peers is not equal, the workload-aware distance measures do not provide the same kind of clustering. Thus, we evaluate the three workload-aware distance measures also for this case. The results lead us to select the Manhattan workload-aware as the most efficient distance measure for clustering the peers in a p2p network.

CHAPTER 4. HISTOGRAM-BASED P2P INDEXES

To build workload-aware overlays, each peer maintains a summary of the data values stored locally; this is called a *local index*. Let *LI(n)* denote the local index and *S(n)* the number of data items of peer *n*. Besides its local index, each peer *n* maintains one routing index *RI(n, e)* for each of its links *e*, that summarizes the content of all the peers that are reachable from *n* using link *e* at a distance at most *r*, called *radius*. We use histograms to summarize the content of each peer.

In the database community, histograms are widely used as a mechanism for compression and approximation of data distributions used in selectivity estimation and approximate query answering [18]. The main advantages of histograms over other techniques are that they incur almost no run-time overhead and, for most real-world databases, there exist histograms that produce low-error estimates while occupying reasonably small space. As mentioned in Chapter 3, we focus on p2p systems where each peer stores a relation *R* with a numeric attribute *x*. Hence, we are interested on constructing *unidimensional* histograms for the approximation of the data distribution of *x* in each peer. Intuitively, a histogram on an attribute *x* is constructed by partitioning the data distribution of *x* into $b\ (\geq 1)$ mutually disjoint subsets called *buckets* and approximating the frequencies and values in each bucket.

4.1. Histograms as Local Indexes

One requirement in our context is using histograms as local indexes that can efficiently approximate the data distribution of the attribute x with a low cost; that is, given the data items stored locally at a peer, we are interested in a low-cost procedure for constructing a histogram that summarizes the content of this peer. To this end, we consider *equi-width* [30] and *maxdiff(v, f)* [31] histograms. The equi-width histogram is chosen due to its simplicity and efficiency in construction cost [31], while the more sophisticated maxdiff(v, f) histogram is selected because it is very close to the best histogram regarding both, construction time and generated errors [31]. As we have mentioned in Section 3.2.1, assume that the numerical domain for the attribute x is $D = \{0, \ldots, M\text{-}1\}$ with M distinct values.

Equi-width histograms group ranges of contiguous attribute values into buckets where the number of attribute values associated with each bucket is the same. The equi-width histogram with b buckets over an attribute x is constructed using the following heuristic. Initially, the value set of attribute x is sorted and then it is divided into b mutually disjoint ranges of equal width w called buckets. Thus, each bucket includes M/b attribute values.

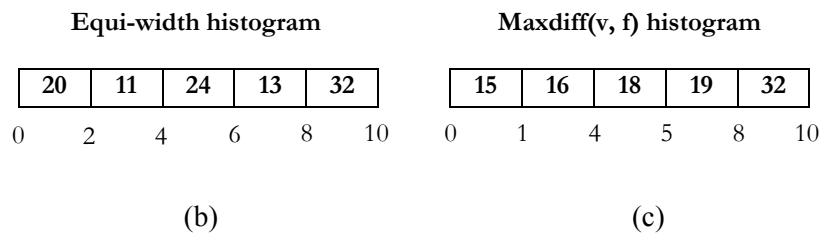
The maxdiff(v, f) histogram groups contiguous sets of attribute values into buckets and places buckets boundaries so as to avoid grouping vastly different frequencies of values into one bucket. In particular, the maxdiff(v, f) histogram with b buckets over an attribute x is constructed using the following heuristic. At first, the value set of attribute x is sorted and then a bucket boundary is placed between two frequencies of values, which are adjacent in attribute value order, if the difference between these frequencies is one of the $b\text{-}1$ largest such differences.

For each bucket $i, 0 \leq i \leq b - 1$, we keep information about the sum of frequencies of values that lie within it, that is the number of tuples (data items) with values for x within the value range of bucket i, as well as the lowest and highest value. In addition, for each histogram we maintain the total number of tuples, denoted as *histogram size*. We shall use the notation $H(n)$ to denote a histogram used as local index for peer n. In addition, considering a histogram $H(n)$ with b buckets, with $H_i(n)$ and $[l_i(n), r_i(n)]$ we denote the frequency and the range of the values within the i-th bucket, $0 \leq i \leq b - 1$, and with $S(H(n))$ its size. In Example 4.1 we demonstrate the construction of both equi-width and maxdiff(v, f) histograms with five buckets over an attribute x.

Example 4.1*: Consider an attribute x of a relation R with value domain D = {0, ..., 9} and M = 10 distinct values. The associated frequencies for each one of the attribute values are shown in Fig. 4.1 (a). For the equi-width histogram with five buckets (b = 5), each bucket keeps the total frequency of two (M/b = 2) attribute values. Hence, the first bucket holds information about the sum of frequencies for the attribute values 0 and 1, i.e., the total number of tuples that belong to the first bucket is 20. Similarly, the total number of tuples that belong to the second bucket (the value of x is included within range [2, 3]) is 11, and so on. The equi-width histogram is depicted in Fig. 4.1 (b).*

Relation R

Attribute x values	Values Frequencies
0	15
1	5
2	8
3	3
4	18
5	6
6	8
7	5
8	15
9	17



(a) (b) (c)

Figure 4.1: (a) Relation *R* with a Numeric Attribute *x*. Construction of (b) an Equi-width and (c) a Maxdiff(v, f) Histogram over the Attribute *x* of *R*.

For the construction of the maxdiff(v, f) histogram with five buckets we perform a single pass over the distinct attribute values and we locate the 4 (b-1) maximum differences between frequencies of successive distinct attribute values as well as an index at which attribute values each difference occurs. In our case, the four maximum differences in frequencies are between the attribute values pairs (0, 1), (3, 4), (4, 5) and (7, 8). Thus, the bucket boundaries are placed between the values of each of the above value pairs. The first bucket is a singleton, i.e., it holds the frequency for the single attribute value 0 which is 15, the second bucket keeps information for the attribute values 1, 2, 3, i.e., the number of tuples have value x within range [1, 3] is 16, and so on. The maxdiff(v, f) histogram is depicted in Fig. 4.1(c).

In addition, for both the equi-width and the maxdiff histograms, we maintain the histogram size, which is 100.

Estimation of Query Results Using Histograms

For a query *q*, we denote by *results(n, q)* the actual number of matching results of peer *n* and by *hresults(H(n), q)* the number of matching results estimated using the histogram *H(n)* of peer *n*. As usual, we make the *uniform frequency* assumption [31] and approximate all frequencies in a bucket by their average. In addition, to approximate the set of attribute values within a bucket, we make the *continuous values* assumption [31], where all possible values in *D* that lie in the range of a bucket are assumed to be present. Assume that we have a range selection query of the form $q_{ij} = \{ x \in D : i \le x \le i + j \}$. The number of results for query q_{ij} using the histogram *H(n)* of peer *n* is estimated as follows: Consider the bucket *l* of *H(n)* with boundaries $[l_l(n), r_l(n)]$ that "straddles" *i*, i.e., $l_l(n) \le i \le r_l(n)$. Likewise, the bucket *r* with boundaries $[l_r(n), r_r(n)]$ is defined as the one that straddles *i+j*. Thus, the range of the query q_{ij} contains portions of the *l* and *r* buckets and every bucket between these two in its entirety. Note that the *l* and *r* buckets may coincide, and there may be no buckets in between, but our discussion here is not seriously affected. For the portions within the *l* and *r* buckets, we use the uniform frequency assumption, i.e., we estimate the total frequency in the interval $[i, i + j] \cap [l_l(n), r_l(n)]$ as $H_l(n)\left[\left([i, i + j] \cap [l_l(n), r_l(n)]\right) / [l_l(n), r_l(n)]\right]$, and likewise for the *r* bucket. The total estimate for the query q_{ij} is the sum of the estimates of frequencies for the *l* and *r* buckets, and the exact sum of frequencies for the buckets in between. More formally, there are two cases:

- If $l \neq r$, then

$$hresults(H(n), q_{ij}) = H_l(n)\frac{[i, i + j] \cap [l_l(n), r_l(n)]}{r_l(n) - l_l(n) + 1} + \sum_{k=l+1}^{r-1} H_k(n) + \\ + H_r(n)\frac{[i, i + j] \cap [l_r(n), r_r(n)]}{r_r(n) - l_r(n) + 1} \tag{4.1}$$

- If $l = r$, then

$$hresults(H(n), q_{ij}) = H_l(n)\frac{[i, i + j] \cap [l_l(n), r_l(n)]}{r_l(n) - l_l(n) + 1} \tag{4.2}$$

Example 4.2: *Consider the equi-width histogram of Figure 4.1 (b) with domain D = {0, …, 9}. We want to estimate the result size for the range predicate* $q_{3,5} = \{x \in D : 3 \le x \le 8\}$. *The values 3 and 8 belong to the* 2^{nd} *and the* 5^{th} *buckets respectively. Thus, the result size of* q_{35} *is:*

$$hresults(H(n), q_{3,5}) = H_2(n)\frac{[3,8] \cap [2,3]}{3-2+1} + H_3(n) + H_4(n) + H_5(n)\frac{[3,8] \cap [8,9]}{9-8+1} \Leftrightarrow$$

$$\Leftrightarrow hresults(H(n), q_{3,5}) = 11\frac{1}{2} + 24 + 13 + 32\frac{1}{2} \approx 58$$

4.2. Histograms as Routing Indexes

Besides its local index, each peer *n* maintains one routing index *RI(n, e)* for each of its links *e*, that summarizes the content of all peers that are reachable from *n* using link *e* at a distance at most *r*, called *radius.* The set of peers within distance *r* of *n* is called the *horizon* of radius *r* of *n*. How routing indexes are used is described in detail in Section 7. In our context, the central issue in creating a routing index for the link *e* of a peer *n* is to efficiently aggregate the histograms of the peers, which represents the local indexes, that are reachable through *e* within *r*. Thus, the routing index *RI(n, e)* is also a histogram.

The problem of merging several histograms into one so as to create the routing index is non trivial in the case we use histograms that their bucket boundaries are not the same. In particular, consider that we have two histograms, e.g., $H(n_1)$ and $H(n_2)$, as local indexes for peers n_1 and n_2 with the same number of *b* buckets and we want to create a routing index for peer *n*, (*RI(n)*), with also *b* buckets that includes the information of both histograms. If $H(n_1)$ and $H(n_2)$ are equi-width histograms, then simply for the *i*-th bucket of *RI(n)*, its bucket boundaries will be the same with the *i*-th bucket boundaries of $H(n_1)$, $H(n_2)$ and its frequency will be the sum of the frequencies of the *i*-th buckets of $H(n_1)$ and $H(n_2)$. In the general case, when we do not use equi-width histograms as local indexes, it is possible that the bucket boundaries of the *i*-th bucket of $H(n_1)$ and $H(n_2)$ are different. Hence, we cannot straightforwardly merge these two buckets into one so as to include the information of both $H_i(n_1)$ and $H_i(n_2)$. In the following section, we provide an algorithm for merging two histograms with *b* buckets.

algorithm *MergeHist*
Inputs: $H(n_1)$, $H(n_2)$, b
Outputs: $MH(H(n_1), H(n_2))$, b
begin
1. /* Find all the intervals that are intersections of the buckets regions of $H(n_1)$ and $H(n_2)$ and compute the total frequency of values within each interval. */
2. $i = 1$, $j = 1$ and $m = 1$;
3. **while**$((i \leq b)$ and $(j \leq b))$
4. $[l_m, r_m] = [l_i(n_1), r_i(n_1)] \cap [l_j(n_2), r_j(n_2)]$;
5. /* Compute the frequency of the $[l_m, r_m]$ at $H(n_1)$ and $H(n_2)$ using the uniform frequency assumption and get the sum of these frequencies.*/
6. $F(MH(H(n_1), H(n_2)), [l_m, r_m]) = F(H(n_1), [l_m, r_m]) + F(H(n_2), [l_m, r_m])$;
7. /* If the right boundary of the intersection is the same with the right interval of $H_i(n_1)$ then go to next bucket of $H(n_1)$. */
8. **if** $r_m = r_i(n_1)$ **then**
9. $i = i + 1$;
10. /* If the right boundary of the intersection is the same with the right interval of $H_i(n_2)$ then go to next bucket of $H(n_2)$. */
11. **else if** $r_m = r_j(n_2)$ **then**
12. $j = j + 1$;
13. **endif**
14. $m = m + 1$;
17. **end while**
18. /* If the number of intersection intervals is larger than b. */
19. **if** $m > b$ **then**
20. /* Find the differences in average frequencies of all adjacent intervals. */
21. **for** $k = 1$ to $(m-1)$
22. $diff_k([l_k, r_k], [l_{k+1}, r_{k+1}]) = F(MH(H(n_1), H(n_2)), [l_k, r_k]) / (r_k - l_k + 1)$ $- F(MH(H(n_1), H(n_2)), [l_{k+1}, r_{k+1}]) / (r_{k+1} - l_{k+1} + 1)$;
23. **endfor**
24. **while(m = b)**
25. /* Find the pair of adjacent intervals with the lowest difference in their average frequencies that has not been selected yet. */
26. $diff_p([l_p, r_p], [l_{p+1}, r_{p+1}]) = min\{diff_1, diff_2, \ldots, diff_k\}$;
27. /* Merge the adjacent intervals and add their corresponding frequencies. */
28. $[l_p, r_p] \cup [l_{p+1}, r_{p+1}]$;
29. $F(MH(H(n_1), H(n_2)), [l_p, r_p]) + F(MH(H(n_1), H(n_2)), [l_{p+1}, r_{p+1}])$;
30. /* Decrease at one the number of intervals. */
31. $m = m - 1$;
32. **end while**
33. **endif**
34. /* Create the Merged Histogram $MH(H(n_1), H(n_2))$. */
35. **for** $i = 1$ **to** b
36. $[l_i(MH(H(n_1), H(n_2))), r_i(MH(H(n_1), H(n_2)))] = [l_i, r_i]$;
37. $MH_i(H(n_1), H(n_2)) = F(MH(H(n_1), H(n_2)), [l_i, r_i])$;
38. **endfor**
end *MergeHist*

Figure 4.2: Algorithm for Merging Two Histograms.

4.2.1. Merging two Histograms

We propose a general algorithm for merging two histograms. This algorithm takes as input two histograms $H(n_1)$ and $H(n_2)$ of b buckets each and gives as output a merged histogram $MH(H(n_1), H(n_2))$ with also b buckets. In Figure 4.2, we present this algorithm.

Initially, we find all possible intervals that are intersections of the two histograms bucket regions and calculate the frequency of each interval as the sum of frequencies of the two histograms for this interval, using the uniform frequency assumption and continuous values assumption (Steps 3-17 in Figure 4.2). In particular, to estimate the frequency of an interval of values, which does not necessarily coincide with the boundaries of a bucket, we use the equations (4.1) and (4.2) introduced for the estimation of range selection queries. The number of these intervals is between $[b, 2b-1]$, considering that each histogram has b buckets.

Since we want to construct routing indexes of size b, if the number of intervals produced previously is larger than b, we want to decrease it to b. We propose to merge all adjacent pairs of intervals that have the lowest differences in their average frequencies until the number of regions that remain is b (Steps 18-33 in Figure 4.2). By average frequency we mean the total frequency of the interval divided by the number of values that are included within this interval. For example, if we want to merge two adjacent regions i and j, where $i < j$, with bucket boundaries $[l_i, r_i]$ and $[l_j, r_j]$ respectively, where $l_j = r_i + 1$, and frequencies $F[l_i, r_i]$ and $F[l_j, r_j]$ respectively, then the merged region will have bucket boundaries $[l_i, r_j]$ and total frequency $F[l_i, r_i] + F[l_j, r_j]$.

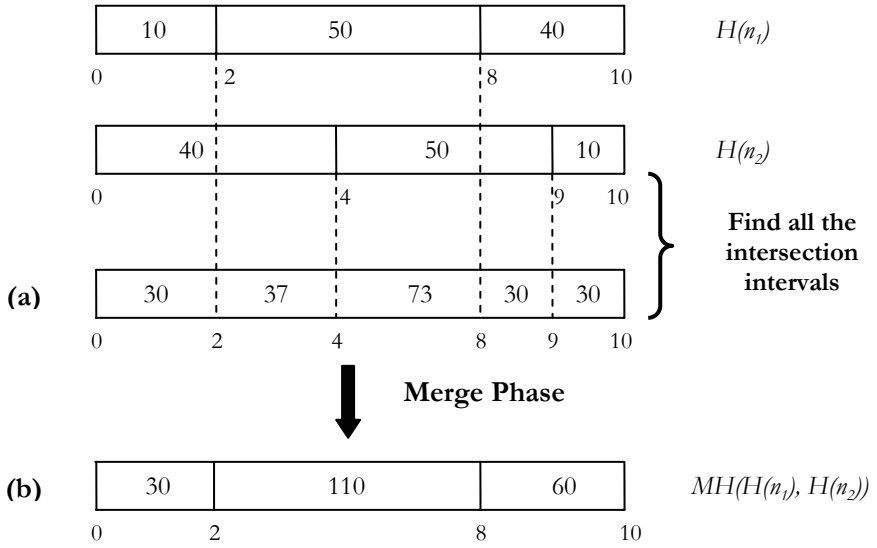


Figure 4.3: An Example of Merging Two Histograms.

Example 4.3: *Consider that we have two histograms, $H(n_1)$ and $H(n_2)$, with 3 buckets over an attribute x with value domain $0 \le x < 10$ and we want to merge these two histograms so as to create the $MH(H(n_1), H(n_2))$, which will summarize the content of both $H(n_1)$ and $H(n_2)$. In Figure 4.3, we demonstrate how the merge process is achieved. In Step (a), we find all the intersection intervals based on the bucket boundaries of $H(n_1)$ and $H(n_2)$ and calculate the frequency of each interval. The number of these intervals is 5, thus to reduce the number of buckets to 3 we find the minimum differences in the average frequencies of adjacent intervals. In our case the 2^{nd} and 3^{rd} intervals and the 4^{th} and 5^{th} intervals have the lowest differences in their average frequencies ($37/2 \approx 73/4$ and $30/1 = 30/1$). Hence, we merge these two pairs of adjacent buckets, we calculate their frequencies and finally get the $MH(H(n_1), H(n_2))$.*

```
algorithm SubtractHist
Inputs: H(n1), MH(H(n1), H(n2), …, H(nk)), b
Outputs: MH(H(n2), …, H(nk)) with updated bucket frequencies
begin
1. for i = 1 to b do
2. /* Define the range of values that the i-th bucket of the H(n1) covers. */
3.        [lm, rm] = [li(n1), ri(n1)];
3. /* Find the set S of ki contiguous buckets of the MH(H(n1), H(n2), …, H(nk)) overlapping the range [lm,
      rm], S = {b1, b2,..., bki}. */
4. /* For each one of the buckets ∈ S. */
5.        for j = 1 to ki do
6. /* Decrease the frequency of the selected bucket bj ∈ S in proportion to the percentage of overlap between the i-
      th bucket of the H(n1) and the selected bucket bj of the MH(H(n1), H(n2), …, H(nk)). */
7.                frac = ([lm, rm] ∩ [lj(MH), rj(MH)])/[lm, rm];
8.                MHj(H(n2), …, H(nk)) = MHj(H(n1), H(n2), …, H(nk))) - frac* Hj(n1);
9.        endfor
10. endfor
end SubtractHist
```

Figure 4.4: Algorithm for Subtraction of a Histogram $H(n_1)$ from the Merged Histogram $MH(H(n_1), H(n_2), ..., H(n_k))$.

4.2.2. Subtraction of a Local Index from a Routing Index

To subtract a histogram $H(n_1)$ from a merged histogram $MH(H(n_1), H(n_2), ..., H(n_k))$ that summarizes the information of several histograms, e.g., $H(n_1)$, $H(n_2)$, …, $H(n_k)$ histograms, we propose an algorithm that is depicted in Figure 4.4. This algorithm is used for subtracting a local index from a routing index since a routing index summarizes the information of several local indexes. The algorithm takes as input a histogram $H(n_1)$ and a merged histogram $MH(H(n_1), H(n_2), ..., H(n_k))$ and gives as output the merged histogram without the information of $H(n_1)$, $MH(H(n_2), ..., H(n_k))$. Recall that both histograms have b buckets. The algorithm

first determines for each bucket i of $H(n_1)$ the set of buckets, $S = \{b_1, b_2, ..., b_{k_i}\}$, of the merged histogram that overlap the range of values that the bucket i covers (lines 1-3 in Figure 4.4). Next, the algorithm decreases the frequency of each one, b_j, of these buckets in proportion to the fraction of overlap between its range of values and the one of the i-th bucket. This fraction is the length of interval where the bucket b_j overlaps the bucket i, divided by the length of the bucket i (line 7). To distribute the frequency of the bucket i, $H_i(n_1)$, among the selected buckets of the merged histogram, for each bucket $b_j \in S$ its frequency is decreased by a portion of $H_i(n_1)$ equal to its contribution to the overlap, i.e., $frac * H_i(n_1)$ (line 8).

4.3. Experimental Evaluation

In this section, we present an experimental evaluation of the equi-width and maxdiff(v, f) histograms and of the accuracy of the merging procedure between two histograms, defined in Section 4.2.1, to construct routing indexes. In particular, in Section 4.3.1 we assess the accuracy of these two types of histograms for query workloads with varying degrees of skewness and investigate their effectiveness for estimating range query result sizes. In Section 4.3.2, we evaluate the algorithm we propose for merging two histograms for several types of query workloads by measuring its accuracy for estimating range query result sizes in comparison with the case when we have two individual histograms.

4.3.1. Accuracy of Equi-width and Maxdiff(v, f) Histograms

To compare these two types of histograms, we first construct a data set for an attribute x and then we create the corresponding equi-width and maxdiff(v, f) histograms for x. Next, we generate a query workload, which consists of range selection queries over x, and measure the performance of the histograms by their estimation errors on this query workload. A common metric for measuring the accuracy of the histograms is the 1-norm of the absolute errors, or *average absolute errors*. We choose average absolute errors as the accuracy metric, since relative errors tend to be less robust when the actual number of results for some queries is zero or near zero. Thus, given a data set Ds, a histogram H and a query workload W of N queries, the *average absolute error*, $E(Ds, H, W)$, is calculated as follows:

$$E(Ds, H, W) = \frac{1}{N} \sum_{q \in W} f_q \left| est(H, q) - act(Ds, q) \right| \qquad (4.3)$$

where $est(H, q)$ is the estimate of the number of data items in the result of q, using histogram H for the estimation, and $act(Ds, q)$ is the actual number of Ds data items in the result of q. To

make the results more meaningful, we normalize 1-norm absolute errors using the largest selectivity in the query workload. The data sets and the query workloads used for the experiments are described below. The number of buckets, *b*, that we use to construct a histogram is 10, 20, 50 and 100.

Table 4.1: Parameters for the Experimental Evaluation of the Equi-width and Maxdiff(v, f) Histograms.

Histogram-Related Parameters	**Default Value**	**Range**
Number of buckets (*b*)		10 - 100
Data Set Parameters	**Default Value**	**Range**
Data Distribution Type	*Zipf*	
Domain of *x*	[0, 999]	
Number of data items	10000	
Skew (*Dz*) of data distribution		0.0 - 3.0
Hot value	0	
Query Workload Parameters	**Default Value**	**Range**
Query Distribution Type	*Zipf*	
Number of queries	10000	
Range of queries		0 - 999
Skew (*Qz*) of query distribution	1.0	
Hot query range	0, 10	

Data Sets: The data sets that are used for the experiments follow data distributions with 10000 data items, while the value domain of the attribute *x* values (*D)* is [0, 999]. The frequency sets were generated with frequencies following the *Zipf* distribution, and the distribution skewness, denoted as *Dz*, varied between 0 (uniform) and 3 (highly skewed). In addition, the correlation between the frequency sets and the values is done as follows: Considering that a value v_i is the most popular, i.e., it has the highest frequency, the popularity reduces as the distance between the value v_i and the other values increases. In our experiments, we assume that the most popular value is value 0.

Query Workloads: Experiments were conducted using different query sets. All queries are of the form $a \le x \le b$ and each set contains 10000 queries. In particular, we dealt with query workloads that consist of either range or prefix-range queries, that is for queries with one sided ranges: $W = \{(q_{om}, f_{q_{om}})\}$ and for any pair q_{oj} and q_{0l} in W one is contained in the other.

The ranges of the queries follow the *Zipf* distribution and for the range queries the starting point of a query, i.e., *a*, is chosen uniformly from the value domain. In addition, the correlation between a query range value and its frequency is similar with the one mentioned for the data sets. In particular, if a query range value is the most popular then the popularity

reduces as the distance between this value and the other values increases. Due to the large number of combinations of the parameter choices, i.e., which query range is the most popular and the skewness (*Qz*) of the distribution, we present results from two experiments that illustrate the typical behavior of the histograms' errors. In the first set *A*, we consider prefix-range queries, i.e., $a = 0$ and $b \in [0, 999]$, while the query ranges follow the *Zipf* distribution with 0 being the most popular range, i.e., keyword-value queries. Set *B* also contains prefix-range queries with 10 being the most popular range value. In both sets, the parameter *Qz* is set equal to 1.0.

Table 4.1 summarizes the parameters that are used in the experiments. We distinguish the input parameters into three categories: *Histogram-related* parameters, *Data set* parameters and *Query workload* parameters.

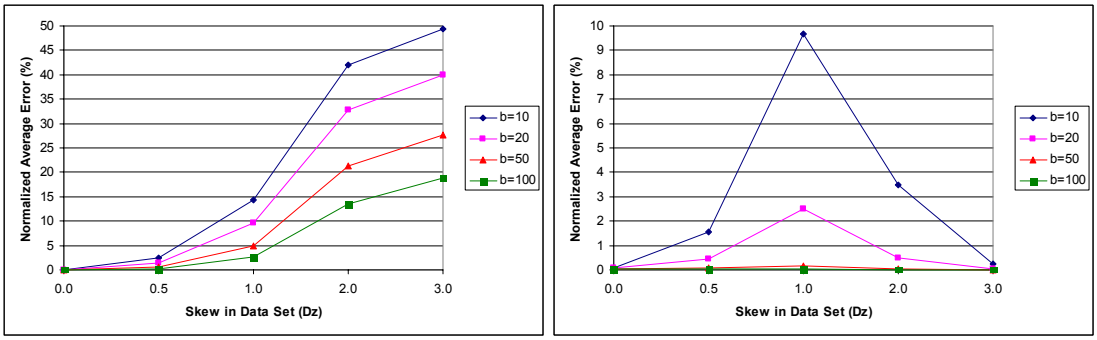


Figure 4.5: (a) *Equi-width* and (b) *Maxdiff(v, f)* Normalized Average Error on Query Set *A* when Varying the Data Set Skew (*Dz*) and the Number of Buckets.

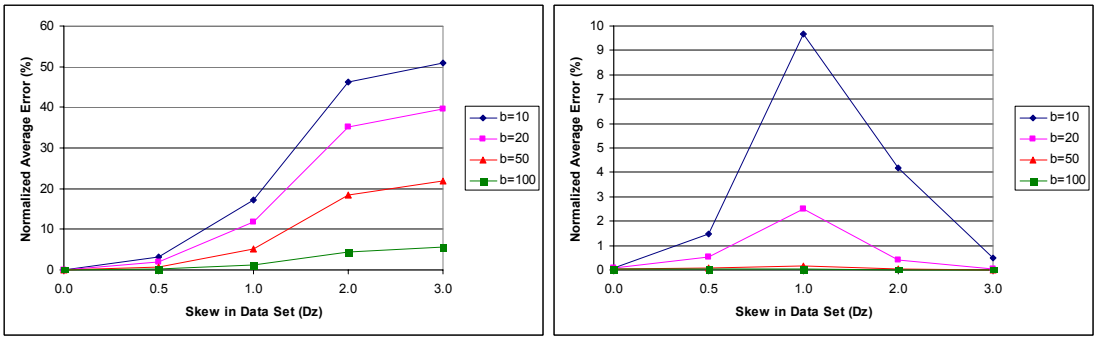


Figure 4.6: (a) *Equi-width* and (b) *Maxdiff(v, f)* Normalized Average Error on Query Set *B* when Varying the Data Set Skew (*Dz*) and the Number of Buckets.

Performance: To study the accuracy of the equi-width and maxdiff(v, f) histograms, we present results that show the errors generated by these two types of histograms when the skew of the data set changes. In particular, in Figures 4.5 and 4.6, we demonstrate the error produced by equi-width (Fig. 4.5(a) and 4.6(a)) and maxdiff(v, f) histograms (Fig. 4.5(b) and 4.6(b)) on the query sets *A* and *B* respectively, when the skew in the data set (*Dz*) varies from 0.0 to 3.0 and the number of buckets (*b*) varies from 10 to 100. In addition, in Figure 4.7 we compare the equi-width and maxdiff(v, f) histograms on the query set *A* for several degrees of skewness in the data set and when the number of histogram buckets varies from 10 to 100.

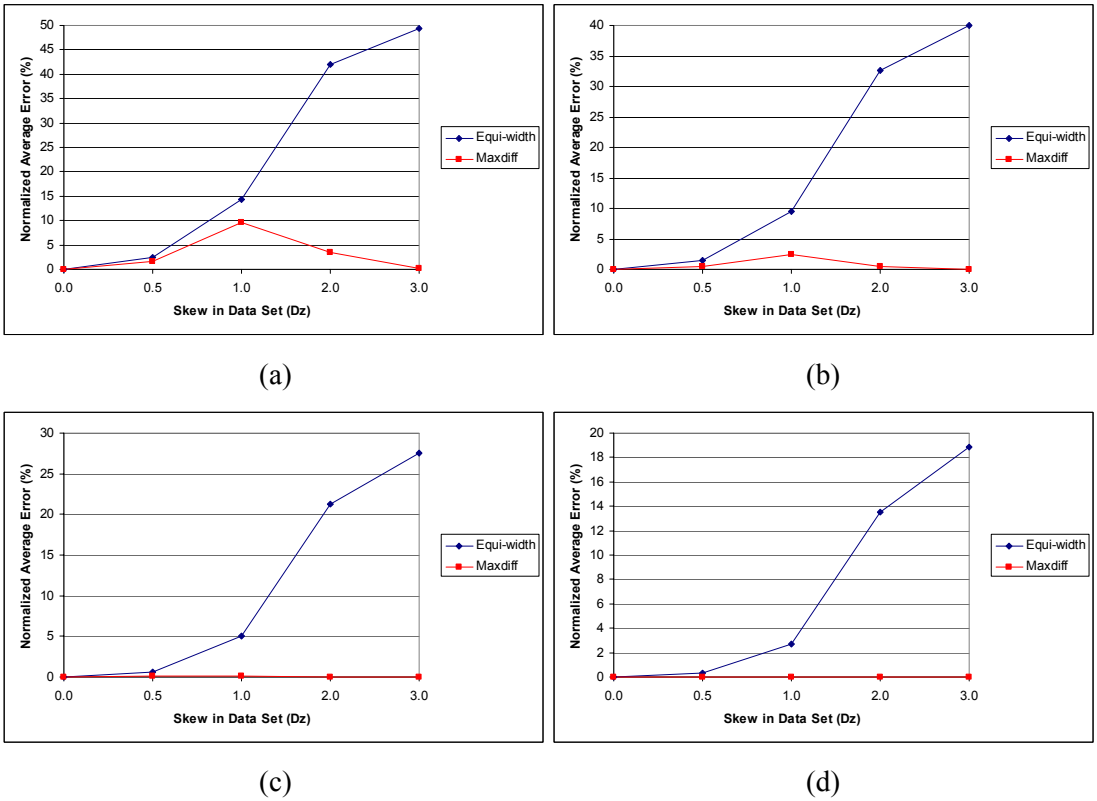


Figure 4.7: Normalized Average Error of *Equi-width* and *Maxdiff(v, f)* Histograms on Query Set *A* when Varying the Data Set Skew (*Dz*) and the Number of Buckets Takes the Values (a) 10, (b) 20, (c) 50 and (d) 100.

The behavior of the histograms in both query sets, *A* and *B*, is similar. Clearly, for very low skewness in the data set (*Dz* = 0.0), both histograms generate essentially low error. In contrast, when the skew is very high (*Dz* = 3.0), the maxdiff(v, f) histogram produces very low error, while the equi-width histogram does not perform well. This happens because the maxdiff(v, f) histogram tries to avoid grouping attribute values with vastly different frequencies into a bucket, while the equi-width does not do so since its bucket boundaries are

fixed from the beginning. Hence, in the limit, i.e., when the skew is very high, the maxdiff(v, f) histogram separates the high frequency values in singleton buckets of their own, a common objective for histograms. In addition, for the maxdiff histogram the hardest to deal with is the intermediate levels of the data set skewness, e.g., Dz = 1.0, since there are many frequencies that are quite different in these distributions. Thus, grouping them into only 10 (Fig.4.7(a)) or 20 (Fig.4.7(b)) buckets was inevitable to avoid errors, i.e., grouping high frequency and low frequency values in the same buckets.

Overall, the equi-width histogram performs well only when the skew in the data set is very low. As the skew increases, the accuracy of the equi-width histogram decreases dramatically. In contrast, the maxdiff(v, f) histogram performs better than the equi-width histogram especially when the data set skew is very high. For intermediate levels of skew, the accuracy of the maxdiff(v, f) is not so high, when the histogram is constructed with a small number of buckets.

Hence, we select the maxdiff(v, f) histogram for using it as peer local index in our p2p system and set the number of buckets to 100. In that case, the maxdiff(v, f) generates negligible errors for every kind of query set and for several degrees of skewness in the data set. The selected number of buckets may seems high but we have to note that for each histogram bucket, we just need to store information about the lowest and highest attribute value in the bucket and its frequency. Thus, the total amount of space that the histogram requires is only a few hundreds of bytes. In particular, the storage cost of a histogram with 100 buckets is only the 10% of an index that keeps for each value of the data domain the corresponding frequency.

4.3.2. Effectiveness of Merge Procedure

To study the effectiveness of the merge procedure between two histograms, H_1 and H_2, we conduct an experiment which measures the accuracy of the histogram that results from merging H_1 and H_2. In particular, we first construct two data sets for the attribute x and we create their corresponding maxdiff(v, f) histograms, H_1 and H_2. Then, we apply the merge procedure on these two histograms and create the merged histogram $MH(H_1, H_2)$. Finally, we generate a query workload and measure the performance of the merged histogram by comparing for each query q of the workload, the result size that is estimated based on the $MH(H_1, H_2)$ histogram with the sum of the result sizes estimated using the original H_1 and H_2. As in the case of measuring the histogram performance, we select the average absolute errors

as accuracy metric to measure the effectiveness of $MH(H_1, H_2)$. Hence, given two data sets Ds_1 and Ds_2, their corresponding histograms H_1 and H_2 and a query workload W of N queries, the *average absolute error*, $E(H_1, H_2, MH(H_1, H_2), W)$, is calculated as follows:

$$E(H_1, H_2, MH(H_1, H_2), W) = \frac{1}{N} \sum_{q \in W} f_q \left| est(MH(H_1, H_2), q) - est(H_1, H_2, q) \right| \quad (4.4)$$

where $est(MH(H_1, H_2), q)$ is the estimate of the number of data items in the result of q, using the merged histogram $MH(H_1, H_2)$ for the estimation, while $est(H_1, H_2, q)$ is the sum of the number of data items for q using the H_1 and H_2 histograms for the estimation, i.e., $est(H_1, H_2, q) = \sum_{i=1}^{2} est(H_i, q)$. In addition, we normalize the average absolute error using the largest estimated selectivity in the query workload.

Note that for the construction of H_1 and H_2 we use 100 buckets, since this is the selected value in the previous section for the construction of the maxdiff(v, f) histograms. The merged histogram $MH(H_1, H_2)$ will also have 100 buckets. The data sets and the query workloads used in our experiment are described below.

Table 4.2: Parameters for the Experimental Evaluation of the Merged Histogram, $MH(H_1, H_2)$.

Histogram-Related Parameters	**Default Value**	**Range**
Histograms' (H_1, H_2) type	*Maxdiff(v,f)*	
Number of buckets (b)	100	
Data Set Parameters	**Default Value**	**Range**
Data Distribution type	*Zipf*	
Domain of x	[0, 999]	
Number of data items	10000	
Skew (Dz) of data distribution		0.0 - 3.0
Hot values for the pair (Ds_1, Ds_2) of data sets	(0, 999), (0, 100)	
Query Workload Parameters	**Default Value**	**Range**
Query Distribution type	*Zipf*	
Number of queries	10000	
Range of queries		0 - 999
Skew (Qz) of query distribution	1.0	
Hot query range	10	

Data Sets: For the experiment, we create a pair of data sets, Ds_1 and Ds_2, over an attribute x with value domain (D) [0, 999]. Each one of the two data sets consists of 10000 data items and their frequency sets follows the Zipf distribution. The skewness, Dz, of the two distributions is the same and varies from 0.0 to 3.0. In addition, the correlation between the frequencies, produced by the Zipf distribution, and the values is similar with the one

mentioned in Section 4.3.1. We consider two different pairs of data sets. In the first pair, DP_A, the data set Ds_1 has the value 0 as the most popular value, while data set Ds_2 has the value 999 as the most popular value, i.e., we consider two totally different data distributions. In the second pair, DP_B, the data set Ds_1 has the value 0 as the most popular value while data set Ds_2 has the value 100 as the most popular value, i.e., the overlap between these two data sets is much larger than the overlapping of DP_A's data sets.

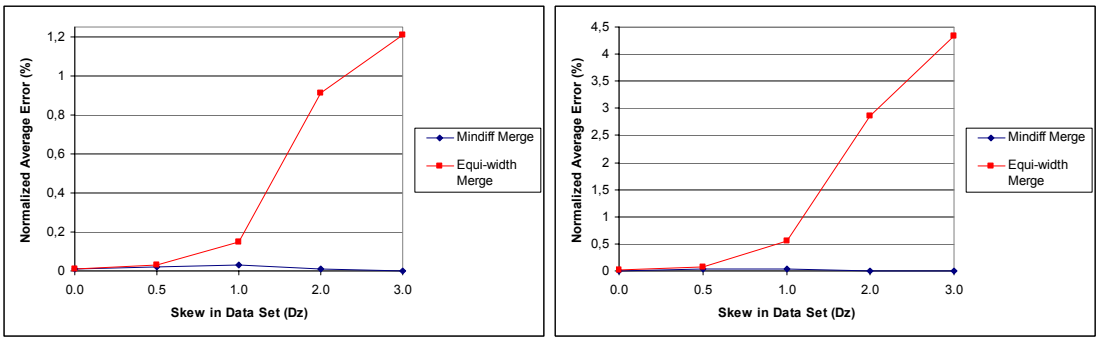
Query Workloads: We considered several kinds of query workloads for the evaluation of the merged histogram. In particular, we deal with query sets that consists of 10000 range queries of the form $a \leq x \leq b$, where the query ranges follow the Zipf distribution. The results of our experiment did not vary significantly for different query sets, so we only present those obtained for two query sets, Qs_A and Qs_B. In the first query set Qs_A, we consider range queries where the starting point of each query, i.e., a, is chosen uniformly, while in the second query set Qs_B, we consider prefix-range queries, i.e., $a = 0$. In addition, in both query sets the skewness (Qz) of the distribution is set to 1.0. The correlation between a query range value and a frequency is similar with the one mentioned for the data sets.

Table 4.2 summarizes the parameters that we use to conduct the experiment of merging two histograms.

Performance: To study the effectiveness of our merging procedure for merging two histograms, H_1 and H_2, into one, $MH(H_1, H_2)$, we present results that show the errors generated by the merged histogram, when the skew of the data set changes and for several kinds of query workloads. In addition, we compare our merging procedure with the *straightforward* or *equi-width* merge, i.e., for the merged histogram we divide the value domain of x into b disjoint regions-buckets of equal width and calculate the frequency in each bucket as the sum of the corresponding frequencies of H_1 and H_2 for the value domain that each bucket of the merged histogram covers, using the uniform frequency and contiguous values assumptions. For simplicity, we denote our merging procedure as *Mindiff* merge procedure.

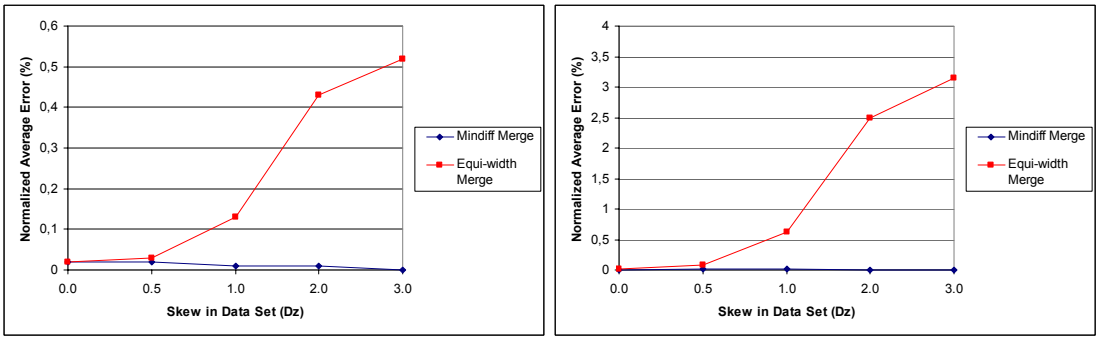
In particular, in Figure 4.8 we demonstrate the normalized average absolute error produced from the $MH(H_1, H_2)$ histogram, which is constructed using the Mindiff and the equi-width merge procedures, for the pair DP_A of data sets, when the data skew (Dz) varies from 0.0. to 3.0 and using the Qs_A (Fig. 4.8(a)) and Qs_B (Fig. 4.8(b)) query sets for the estimation of the error. Similarly, in Figure 4.9 we conduct the same experiment but using the DP_B pair of data

sets. As expected, in all cases the merged histograms that are produced by the Mindiff and equi-width merge procedures perform similarly for low degrees of skew of the data sets but as the skew increases, e.g., when Dz = 2.0 or 3.0, the Mindiff merged histogram is much more efficient than the equi-width merged histogram.



(a) (b)

Figure 4.8: Comparison of the *Mindiff* and *Equi-width* Merge Procedures Using the (a) Qs_A and (b) Qs_B Query Sets for the Pair DP_A of Data Sets when Varying the Data Set Skew (Dz).



(a) (b)

Figure 4.9: Comparison of the *Mindiff* and *Equi-width* Merge Procedures Using the (a) Qs_A and (b) Qs_B Query Sets for the Pair DP_B of Data Sets when Varying the Data Set Skew (Dz).

In addition, we observe that for data sets with high skew, e.g., Dz = 3.0, the error produced by the Mindiff merged histogram is lower than for intermediate levels of data sets skewness, e.g., Dz = 0.5 or 1.0. This happens because in each data set with high skew the values that have frequency different from 0 are very few; hence it is easy for the Mindiff merge procedure to capture these values into singleton buckets. In contrast, when we have intermediate levels of data sets skewness, the overlap between the two different data sets is significant; hence it is

hardest to capture the distributions of both data sets especially when the two data sets follow totally different data distributions, as in the case of DP_A pair of data sets.

4.4. Summary

To conclude, in this chapter based on the idea of using histograms as local indexes to summarize the data values stored locally at each peer, we study the performance of different types of histograms to determine the appropriate type for this setting. In particular, we select the equi-width and the maxdiff(v, f) histogram types and present the corresponding algorithms for their construction. The equi-width histogram is chosen due to its simplicity and efficiency in construction cost [31], while the more sophisticated maxdiff(v, f) histogram is selected because it is very close to the best histogram regarding both, construction time and generated errors [31]. We experimentally evaluate the accuracy of these two histogram types for several kinds of data sets and query workloads in the cases when the number of buckets varies. The main conclusions from our experiments show that the maxdiff(v, f) histogram performs better than the equi-width histogram, especially in cases when the skewness of the data distribution is high. Thus, we select the maxdiff(v, f) histogram as the appropriate one for summarizing the content of the peers.

In addition, we use the notion of the routing index that each peer *n* maintains for each of its links *e*, *RI(n, e)*, that summarizes the content of all peers that are reachable from *n* using link *e* at a distance at most *r*, called radius. We propose a general algorithm for merging two histograms into one, denoted as *Mindiff* merge, so as to create the routing indexes. Especially, we study the effectiveness of our *Mindiff* merge, by measuring the accuracy of the corresponding constructed merged histogram in comparison with the merged histogram that is produced using an equi-width procedure for several kinds of pairs of data sets and for several kinds of query workloads. The results show that the Mindiff merge is much more efficient, i.e., its corresponding merged histogram generates very small errors, than the equi-width one especially when the skew of the data sets is very high.

CHAPTER 5. HISTOGRAM-BASED QUERY WORKLOAD SYNOPSIS

For taking into account the query workload for the formation of clusters, we must maintain a summary of it, which is denoted as *query workload synopsis*. We also propose using histograms to summarize the query workload. To this end, we introduce the notion of the global query workload and the local query workload. The *global query workload*, denoted as *W*, represents the set of queries along with their corresponding frequencies that are issued in the p2p system. In contrast, the *local query workload* for a peer *n*, denoted as *LW(n)*, represents the set of queries along with their corresponding frequencies that arrived to the peer. Besides the local index and the routing indexes, a peer *n* also maintains a summary of its local query workload. Since, we are interested on routing queries on an attribute *x*, we focus on *unidimensional* histograms for the summarization of the query workload. We shall use the notations *WS* and *LWS(n)* to denote the global query workload synopsis and the local query workload synopsis of the peer *n*, respectively. Since we are using histograms as synopsis for the query workload we shall also use the notations *H(W)* and *HW(n)* to denote the histograms that summarize the global query workload *W* and the local query workload of a peer *n*, *LW(n)*, respectively.

5.1. Histograms as Query Workload Synopsis

To represent a query workload *QW* efficiently, we propose using histograms as query workload synopsis to summarize the query workload over an attribute *x* with low cost. Notice that building a histogram over an attribute *x* that is used as local index for a peer is based on initially scanning the entire data set of this peer, sorting the data on *x* and partitioning them into buckets. Thus, the total information required for the construction of a peer histogram, i.e., the data items of a peer, is available. In contrast, we do not know from the beginning the entire query workload information, since the query workload set changes dynamically as new queries are initiated. Hence, for building a histogram that efficiently represents the query workload, several requirements must be satisfied.

The first requirement is that the histogram be adaptive to query workload changes so as to represent efficiently the query workload set at each time. In particular, as mentioned before, the query workload dynamically changes as new queries are initiated. Thus, restructuring the histogram is necessary so as to include this change. An additional requirement is that the cost for building and restructuring the histogram that summarizes the query workload at each time must be very low. Since queries are issued very frequently in a p2p system, we expect that the need for restructuring the query workload histogram arises often. Thus, the overhead that is incurred by the reconstruction must be negligible. Finally, the histogram that summarizes the query workload must be efficient, meaning that the estimation of the frequency that each distinct query is issued must be accurate.

Traditional types of histograms, such as *maxdiff*, *v-optimal*, *equi-width*, *equi-depth* histograms, do not have the potential of dynamic restructuring. These kinds of histograms are widely used for capturing data distributions of available data items. Thus, they are not suitable for summarizing query workload because they cannot be modified efficiently as new queries are issued.

We propose two types of histograms for representing the query workload. Suppose we have a query workload consisting of a set of queries over an attribute *x*. We would like to construct a histogram over the attribute *x* that summarizes efficiently the query workload. The first type of histogram that we propose is based on the traditional *equi-width* histogram [30], denoted as *Workload-equi-width (W-Equi-width)* histogram. The main idea for building a W-equi-width histogram with *b* buckets is to initially create an equi-width histogram built with whatever information we have about the query workload. Whenever a new query is issued, we update the frequency of the corresponding bucket to include the information of the new query.

In addition, we propose a novel approach based on *Self-tuning (ST) histograms* [3], denoted as *Workload Self-tuning (W-ST) histogram*, that helps reduce the cost of building and maintaining histogram for the query workload. We start with an initial histogram built with whatever information we have about the query workload. As queries are issued, we refine the histogram bucket frequencies, i.e., update the bucket frequencies, and then we rebuild the histogram, i.e., restructure the buckets by moving the bucket boundaries to get a better partitioning that avoids grouping queries with high frequency and queries with low frequency in the same buckets. In general, the W-ST histogram is adaptive; meaning that as new information arrives, the W-ST histogram adjusts its bucket boundaries and generally is restructured to include the new information.

In Section 3.6.1.2, we considered that we deal with query workloads consisting of range selection queries where the starting point of each query is chosen uniformly and the query ranges vary according to a distribution. Hence, to represent the query workload *QW* we have to approximate accurately, via histograms, the distribution that the query ranges follow. We describe next in detail how the W-Equi-width and W-ST histograms are built. Each one of them consists of *b* buckets. In each bucket *i*, we store the value range of the query ranges that it represents, i.e., the query range domain that the bucket *i* covers, and the number of queries in this domain, i.e., the frequency of the bucket. We shall use the notation *H(QW)* to denote the query workload histogram. In addition, for the query workload histogram *H(QW)* with *b* buckets, with $H_i(QW)$ we denote the frequency of the values within the *i*-th bucket, $0 \le i \le b-1$, and with *S(H(QW))* its size. Note, that the range of a query can take values from the domain *D* of the attribute *x*.

5.1.1. Workload Equi-Width Histograms

We describe how to construct a W-Equi-Width histogram for a workload *QW* consisting of a set of *b* buckets. In general, the construction of a W-Equi-Width histogram consists of two stages. First, the histogram is initialized and then it is refined. The refinement process is driven by the query workload. The bucket frequencies are updated with every query on the histogram attribute. We describe each of these steps in the rest of this section.

5.1.1.1. Initial Histogram

The *b* bucket boundaries for the initial histogram are specified using the equi-width heuristic mentioned in Section 4.1. Thus, the *b* buckets of the initial histogram are evenly spaced

between 0 and *M*-1. At the time of initializing the histogram structure, we have no information about the query workload. Thus, initially we set the frequency to zero for each one of the histogram buckets.

5.1.1.2. Refining Bucket Frequencies

The bucket frequencies of a W-Equi-Width histogram are refined (updated) using the information from the queries of the workload. In particular, for every issuing query on the histogram attribute, we increase by one the frequency of the appropriate bucket, i.e., the bucket that contains the value of the query range in its range. Figure 5.1 presents the algorithm for updating the bucket frequencies of a W-Equi-Width histogram in response to a range selection query q_{ij}.

algorithm *UpdateFreq*
Inputs: H(QW), b, q_{ij}
Output: H with updated bucket frequencies
begin
1. *Find the bucket k that contains within its range the value j.*
2. *Increase by one the frequency of the bucket k.*

end *UpdateFreq*

Figure 5.1: Algorithm for Updating Bucket Frequencies in W-Equi-Width Histograms.

5.1.2. W-ST Histograms

Refining bucket frequencies, as W-Equi-Width histogram does, is not enough to get an accurate histogram. The frequency of a specified range within a bucket is approximated by the average frequency of the bucket, i.e., the frequency of the bucket divided by the number of values that are contained within the value range of the bucket. Hence, if there is a large variation in frequency within a bucket, the average frequency is a poor approximation of the individual frequencies. Specifically, high frequency values will be contained in high frequency buckets, but they may be grouped with low frequency values in these buckets. Thus, in addition to refining the bucket frequencies, we must also restructure the buckets, i.e., move the bucket boundaries to get a better partitioning that avoids grouping high frequency and low frequency values in the same buckets.

Hence, we propose the W-ST histogram that tries to avoid grouping query ranges which are much more frequent than others in the same bucket. In general, the lifecycle of a W-ST histogram consists of two stages. First, it is initialized and then it is refined. The process of

refinement can be broken further into two parts: (a) refining individual bucket frequencies, and (b) restructuring the histogram, i.e., moving the bucket boundaries. The initialization and the refinement of the individual bucket frequencies are identical with those described for the W-Equi-Width histogram in Sections 5.1.1.1 and 5.1.1.2, respectively. Thus, the bucket frequencies are updated with every query, while the bucket boundaries are updated by periodically restructuring the histogram. We describe the restructuring process in the following section.

5.1.2.1. Restructuring

In the restructuring process, we restructure the buckets, i.e., move the bucket boundaries so as to get a better partitioning that avoids grouping high frequency and low frequency query ranges in the same buckets. Therefore, we choose buckets that currently have high frequency and split them into several buckets. *Splitting* induces the separation of high frequency and low frequency values into different buckets. The frequency refinement process later adjusts the frequencies of these new buckets. Furthermore, to ensure that the number of buckets assigned to the W-ST histogram does not increase due to splitting, we need a mechanism to reclaim buckets as well. To this end, we use a step of *merging* that groups a run of consecutive buckets with similar frequencies into one bucket.

Summarizing, we restructure the W-ST histogram periodically by merging buckets and using the freed buckets to split high frequency buckets. The restructuring process is invoked after every *ri* issuing queries. The parameter *ri* is called the *restructuring interval*. The merge and split procedures are described below.

Merge Process

To merge buckets with similar frequencies, we first have to decide how to quantify “similar frequencies”. We assume that two bucket frequencies are similar if the difference between them is less than *mt* percent of the sum of frequencies of all buckets, i.e., the total number of queries that are summarized from the histogram at the time we begin the restructure phase for the histogram. *mt* is a parameter that we call the *merge threshold*. We use a greedy strategy to form a run of adjacent buckets with similar frequencies and collapse them into a single bucket. We repeat this procedure until no further merging that satisfies the merge threshold condition is possible.

```
algorithm RestructureHist
Inputs: H(QW), b, mt, st, Q
Outputs: restructured H
begin
1.  /* Find buckets with similar frequencies to merge. */
2.  Initialize b runs of buckets such that each run contains one histogram bucket;
3.  For every two consecutive runs of buckets, find the maximum difference in frequency between a bucket in the
    first run and a bucket in the second run;
4.  Find the minimum of all these maximum difference, mindiff;
5.  if mindiff ≤ mt*Q then
6.          Merge the two runs of buckets corresponding to mindiff into one run;
7.          Look for other runs to merge. Goto line 3;
8.  endif
9.
10. /* Assign the extra buckets freed by merging to the high frequency buckets. */
11. k=st*b
12. Find the set, {b1, b2,..., bk} of buckets with the k highest frequencies that were not chosen to be merged with
    other buckets in the merging step;
13. Assign the buckets freed by merging to the buckets of this set in proportion to their frequencies;
14.
15. /* Construct the restructured histogram by merging and splitting. */
16. Merge each previously formed run of buckets into one bucket spanning the range represented by all the buckets
    in the run and having a frequency equal to the sum of their frequencies;
17. Split the k buckets chosen for splitting, giving each one the number of extra buckets assigned to it earlier.
18. The new buckets are evenly spaced in the range spanned by the old bucket and the frequency of the old bucket
    is equally distributed among them;
end RestructureHist
```

Figure 5.2: Algorithm for Restructuring W-ST Histogram.

Figure 5.2 (Steps 2-9) depicts the merge process for a W-ST histogram, *H(QW)*, of *b* buckets for a query workload *QW*. The first step in histogram restructuring is greedily finding runs of consecutive buckets with similar frequencies to merge. The algorithm repeatedly finds the pair of adjacent runs of buckets such that the maximum difference in frequency between a bucket in the first run and a bucket in the second run is the minimum over all pair of adjacent buckets. The two runs are merged into one, if this difference is less than the threshold *mt*Q*, where *Q* is the total number of queries that are summarized from the histogram at the time we begin the construction of the histogram, otherwise we stop looking for runs to merge. This process results in a number of runs of several consecutive buckets. Each run is replaced with one bucket spanning its entire range, and with a frequency equal to the total frequency of all the buckets in the run. This frees a number of buckets to allocate to high frequency buckets during splitting.

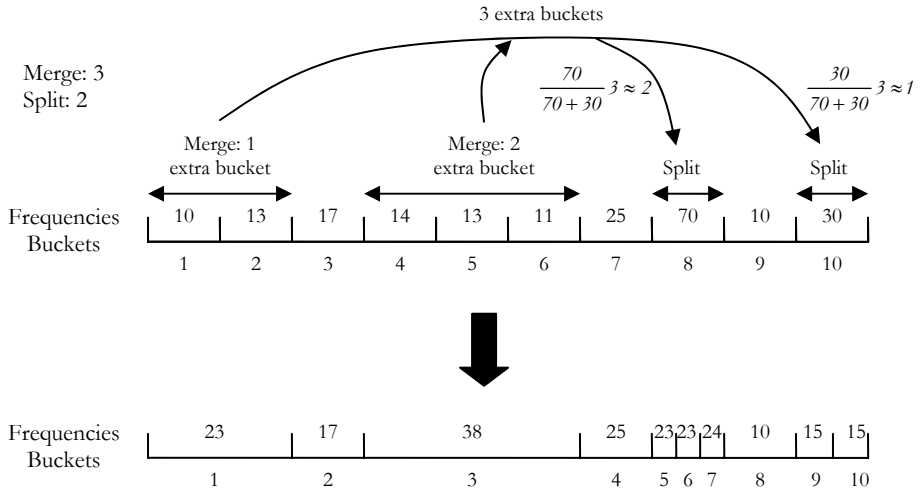


Figure 5.3: Example of Histogram Restructuring.

Split Process

In the split process, we have to decide which "high frequency" buckets to split. We choose to split the *st* percent of the buckets with the highest frequencies. *st* is a parameter that we call the *split* threshold. The heuristic that is used distributes the reclaimed buckets among the high frequency buckets in proportion to their frequency. The higher the frequency of a bucket, the more extra buckets it gets. Specifically, splitting starts by identifying the *st* percent of the buckets that have the highest frequencies and are not singleton buckets. We avoid splitting buckets that have been chosen for merging since their selection indicates that they have similar frequencies to their neighbors. The extra buckets freed by merging are distributed among the buckets being split in proportion to their frequencies. In particular, a bucket *j* being split gets $H_j(QW)/totalfreq$ of the extra buckets, where $H_j(QW)$ is the frequency of bucket *j* and *totalfreq* is the total frequency of the buckets being split. To split a bucket, it is replaced with itself plus the extra buckets assigned to it. These new buckets evenly divide the range of the old bucket, and the frequency of the old bucket is evenly distributed among them. Figure 5.2 (Steps 10-18) depicts the split process for restructuring a W-ST histogram of *b* buckets for a query workload *QW*. Note that splitting may unnecessarily separate values with similar low frequencies into different buckets. Such runs of buckets with similar low frequencies would be merged during subsequent restructuring.

Example 5.1: *An example of histogram restructuring is depicted in Figure 5.3. In this example, the merge threshold is such that the merge procedure merges buckets if their difference between their frequencies is within 3. The algorithm identifies two runs of buckets to be merged 1 and 2, and buckets 4 to 6. Merging these runs frees three buckets to assign to high frequency buckets. The split threshold is such that we split the two buckets with the highest frequencies, buckets 8 and 10. Assigning the extra buckets to these two buckets in proportion to frequency means that bucket 8 gets two extra buckets and bucket 10 gets one extra bucket.*

5.2. Adaptivity of the W-ST Histogram

Up to now, we have proposed the W-ST histogram to capture the distribution that the query workload follows. The main issue is what happens when the distribution of the query workload changes. In particular, we are interested in approximating the query distribution of the current queries and not of the queries that were issued in the past. However, the W-ST histogram, as introduced, keeps all the information of the past queries. This is not considered a drawback, when the query workload does not change. But in the case when the query workload changes this is not desirable since we want the W-ST histogram to approximate the current distribution that the queries follow. In other words, the W-ST histogram does not adapt to changes in the workload fast.

In general, the problem of capturing the change of a query distribution is hard to deal with. To address this problem, we propose using an aging technique. In particular, after a restructuring process takes place we multiply the frequency of each bucket by a factor ag, $0 \leq ag \leq 1$, which we call *aging factor*. This way, we degrade the importance of the queries that were issued in the past. The closer to zero we set the aging factor, the more we degrade the information of the past queries. In contrast, the closer to one we set the value of ag, the longer the histogram keeps the past information about the query distribution. An additional requirement for the W-ST histogram to become adaptive is to change the merge criterion. Recall that, we merge a pair of adjacent runs of buckets, if the maximum difference between a bucket in the first run and a bucket in the second run is less than the threshold $mt*Q$, where Q is the total number of queries, i.e., the sum of bucket frequencies, that are summarized from the histogram. Because of the aging technique, after each restructuring process, we set the $Q = ag*Q$. In this way, the W-ST histogram becomes adaptive.

The aging factor depends on two parameters. The restructuring interval, *ri*, and the query distribution update, i.e., how many queries follow a specified distribution before this distribution changes, denoted as *qd_up*. Specifically, if $ri \approx qd\_up$, i.e., the query distribution changes almost in every restructuring interval then we have to set the aging factor close to 0 so as to not take into account the previous distribution that the queries follow. In contrast, if $qd\_up >> ri$, i.e., the query distribution changes rarely, then we have to set the aging factor close to 1.0.

5.3. Experimental Evaluation of Histograms that Are Used as Query Workload Synopsis

In this section, we evaluate the performance of the W-Equi-Width and W-ST histograms for representing the distribution of the query ranges of a workload *QW*. Initially, we make an extensive discussion about the effect of the refinement parameters in the accuracy of the W-ST histograms and select the appropriate values for each one of them. Then, we compare the W-Equi-Width and W-ST histograms for several query workloads with varying degrees of skew for the distribution of the query ranges.

Consider that we have a query workload $QW = \{(q, f_q)\}$, with range selection queries on an attribute *x*, in which the ranges of the queries follow a distribution and use it to construct the two types of histograms. Assuming that the value domain of attribute *x* is *D*, then the range of a query takes values from *D*. We used several metrics for measuring the accuracy of the W-Equi-Width and W-ST histograms (*H)* for a query workload *QW*. A common measure is the *average absolute error*, which is defined as follows:

$$E(D, H, QW) = \frac{1}{|D|} \sum_{i \in D} |est(H, i) - act(QW, i)| \qquad (5.1)$$

where *est(H, i)* is the estimate of the number of queries in *QW* with range equal to *i*, using histogram *H* for the estimation, and *act(QW, i)* is the actual number of queries with value range *i*. Note that the estimation of the number of queries in *QW* with range equal to *i*, using histogram *H* is done using the uniform frequency assumption and the continuous values assumption. As in the case of measuring the effectiveness of equi-width and maxdiff histograms, we choose again the 1-norm of absolute errors as the accuracy metric, since relative errors tend to be less robust when the actual number of queries with a specified range is zero or near to zero. In general, however, absolute errors vary over different query sets, i.e., query workloads with different number of queries, making it difficult to report results for

different query sets. Therefore, we *normalize* the average absolute error by dividing it by $E_{unif}(D, QW) = \frac{1}{|D|}\sum_{i \in D}\left|est_{unif}(QW, i) - act(QW, i)\right|$, where $est_{unif}(QW, i)$ is the number of queries in QW with range i obtained by assuming uniformity, i.e., in the case where no histograms are available. We refer to the resulting measure as *Normalized Absolute Error*, which is given by the following equation:

$$NAE(D, H, QW) = \frac{\sum_{i \in D}\left|est(H, i) - act(QW, i)\right|}{\sum_{i \in D}\left|est_{unif}(QW, i) - act(QW, i)\right|} \tag{5.2}$$

An alternative measure is the *Normalized weighted absolute error*, which is calculated as follows:

$$NWAE(D, H, QW) = \frac{1}{\sum_{q \in QW} f_q}\sum_{i \in D} p_i \left|est(H, i) - act(QW, i)\right| \tag{5.3}$$

where p_i is the probability of query with range equal to i occurrences in QW. This metric favors the range values with high frequencies. For example, consider that the actual and the estimated number of queries with range j is $act(QW, j) = 100$ and $est(H, j) = 101$ respectively while the actual and the estimated number of queries with range k is $act(QW, k) = 1000$ and $est(H, k) = 1001$. In both ranges, the absolute difference between the actual and the estimated results is the same but the absolute difference that occurs for range k contributes more to the error than the absolute difference that occurs for range j. Hence, the error that is produced by the most frequent queries is much more significant than the error produced by rare queries. In addition, we normalize the weighted absolute error by dividing it with the sum of query frequencies of the workload.

The number of buckets that we used to conduct the experiments varies from 10 to 100. In addition, the query workloads and the distribution that the query ranges follow are described below.

Query Workload Distribution: We use workloads consisting of 10000 range queries on an attribute x; the domain D of x is [0, 999]. Furthermore, the query ranges follow the *Zipf* distribution with several degrees of skewness. We set 0 to be the most popular range, i.e., keyword-value queries. The correlation between a query range value and a frequency is identical with the one used for the creation of the data sets and query workloads of Section 4.3. The parameters we use for evaluating the W-Equi-Width and W-ST histograms are summarized in Table 5.1.

Table 5.1: Parameters for the Experimental Evaluation of the W-Equi-Width and W-ST Histograms.

Histogram-Related Parameters	**Default Value**	**Range**
Number of buckets (*b*)		10 - 100
Query Workload Parameters	**Default Value**	**Range**
Query Distribution type	*Zipf*	
Domain of *x*	[0, 999]	
Number of queries	10000	
Range of queries		0 - 999
Skew (*Qz*) of query distribution		0.0 - 3.0
Hot query range	0	

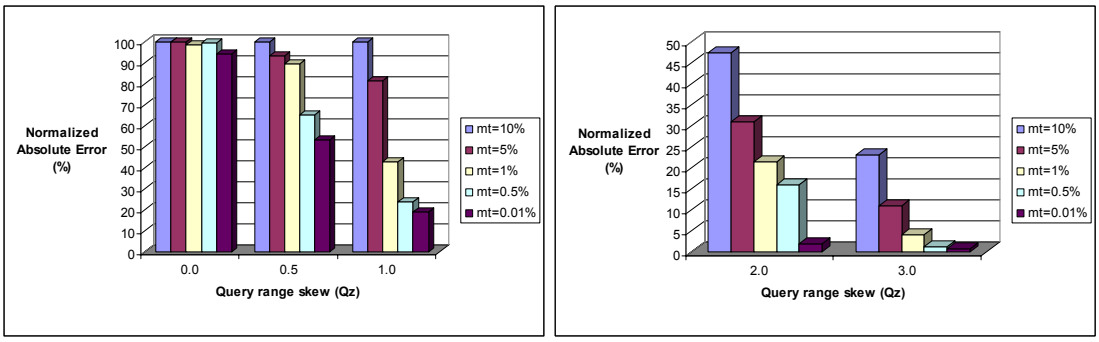
5.3.1. Effect of the Refinement Parameters

In this section, we investigate the effect of the refinement parameters: merge threshold (*mt*), split threshold (*st*) and restructuring interval (*ri*), in the accuracy of the *W-ST* histogram. For each one of these parameters, we select multiple values and we conduct experiments in which we measure the Normalized absolute error and the Normalized weighted absolute error, for several kinds of query workloads with varying degrees of the query range skew. We select the appropriate value for each parameter that minimizes both metrics.

Effect of the Merge threshold parameter (*mt*): The selection of the appropriate merge threshold value is an important factor for efficiency of the *W-ST* histogram. Recall that the merge threshold determines which adjacent buckets have similar frequencies, hence we can merge them. If we select a large value for *mt*, then two adjacent buckets with large variance in their frequencies will be assumed similar and will be merged into one bucket. Hence, values with large difference in their frequencies will be placed in the same bucket, which is not desirable.

In Figure 5.4, we demonstrate the estimation errors, using as measure the Normalized absolute error (Eq. 5.2), which the W-ST histogram achieves when we vary the query range skew (*Qz*) of the query workload and the merge threshold takes the values *mt* = 10%, 5%, 1%, 0.5% and 0.01%. The number of buckets, the split threshold and the restructuring interval that we selected for the conduction of the experiment are 100, 10% and 100, respectively. As expected, when the skew is very low, i.e., 0.0, the error that the W-ST histogram generates is the same for every selected value of *mt*. This happens because in this case we have uniform distribution; hence the frequency of all the values is nearly the same. Thus, the problem of placing values with similar frequencies into a single bucket becomes trivial and independent of the *mt* value. In addition, a nearly 100% error occurs because the absolute error generated

by the W-ST histogram is almost the same with the error occurred when we assume uniformity, which is the case.



(a) (b)

Figure 5.4: Effect of the Merge Threshold (*mt*) in the Accuracy of the W-ST Histogram for (a) Low Degrees and (b) High Degrees of Range Skew Using the *Normalized Absolute Error*.

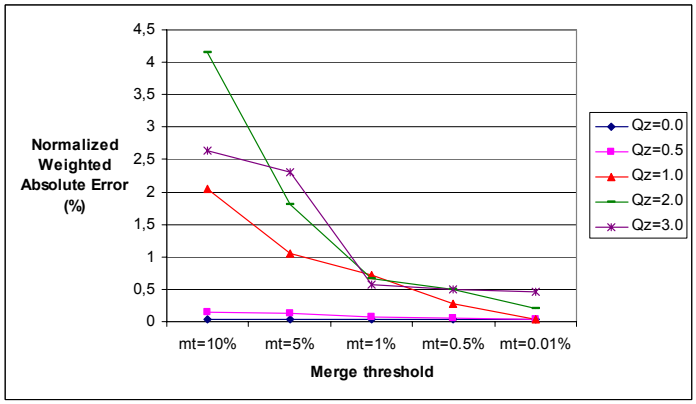
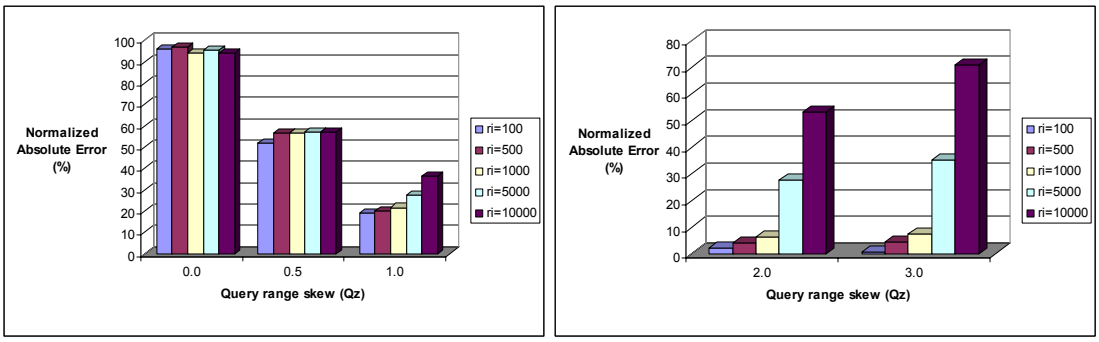


Figure 5.5: Effect of the Merge Threshold (*mt*) in the Accuracy of the W-ST Histogram Using the *Normalized Weighted Absolute Error*.

As the degree of skewness increases, i.e., the distribution becomes narrower; the selection of the appropriate value for the *mt* parameter becomes crucial. As we can see in Fig.5.4(b), where *Qz* takes the values 2.0 and 3.0, when *mt* takes the value 10% the estimation error is very high because it merges buckets with dissimilar frequencies. Hence, the W-ST histogram cannot isolate the high frequency values into singleton buckets. In contrast, the 0.01% value for the merge threshold leads to satisfactory results with low estimation error. We are lead to the same conclusions when we use as measure the Normalized weighted absolute error (Eq. 5.3). Figure 5.5 depicts the effect of the merge threshold in the accuracy of the W-ST histogram for several degrees of range skew.

Effect of the Restructuring Interval parameter (*ri*): The restructuring interval, i.e., how often the restructuring phase must be done so as to restore the histogram in a consistent state, is an important factor for the accuracy of the W-ST histogram and depends upon the degree of skew of the query range distribution.



(a) (b)

Figure 5.6: Effect of the Restructuring Interval (*ri*) in the Accuracy of the W-ST Histogram for (a) Low degrees and (b) High Degrees of Range Skew Using the *Normalized Absolute Error*.

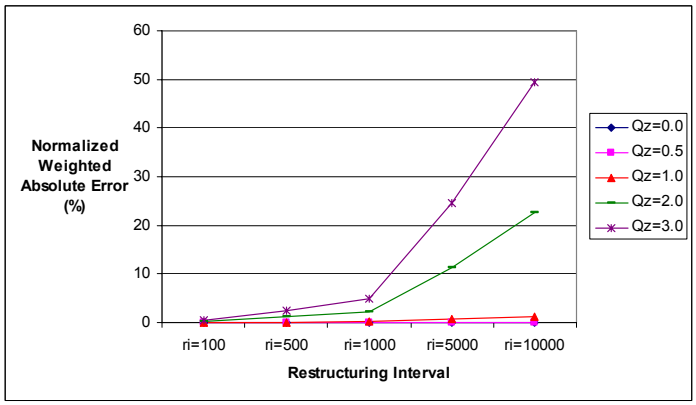


Figure 5.7: Effect of the Restructuring Interval (*ri*) in the Accuracy of the W-ST Histogram Using the *Normalized Weighted Absolute Error*.

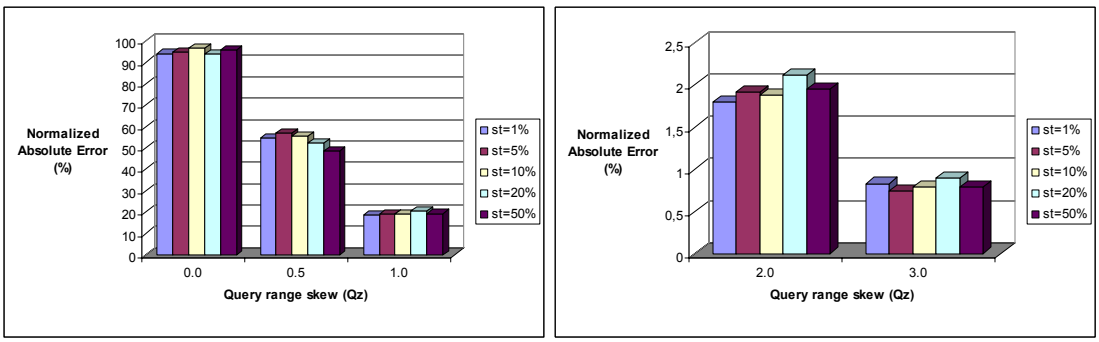
In Figure 5.6 we present the efficiency of the W-ST histogram, using the Normalized absolute error, when the query range skew varies from 0.0. to 3.0 and the restructuring interval takes the values *ri* = 100, 500, 1000, 5000 and 10000 queries. Furthermore, we set to 100, 0.01% and 10% the number of buckets, the merge threshold and the split threshold respectively. Obviously, for query workloads with low skew (Fig.5.6(a)) the restructuring phase has no benefits but as the skew increases (Fig.5.6(b)), the need for restructuring becomes evident. In

particular, when the restructuring phase takes place rarely, e.g., every 5000 or 10000 queries, and the degree of skewness is very high, e.g., 3.0, the values with high frequencies cannot be efficiently isolated into singleton buckets. Hence, values with large variance in their frequencies are placed in the same bucket, which causes the performance of the histogram to be low. In contrast, when the restructuring process is invoked after 100 queries, the W-ST histogram leads to efficient results. We lead to the same conclusions when we use the Normalized weighted absolute error. In particular, in Figure 5.7 we demonstrate the effect of the restructuring interval in the accuracy of the W-ST histogram for several degrees of range skew.

Effect of the Split threshold parameter (*st*): We also investigate the influence of the split threshold value (*st*) in the accuracy of the W-ST histogram. Recall that the *st* parameter determines the percentage of histogram buckets with the highest frequencies that we choose to split and distribute to them, in proportion to their frequencies, the free buckets that arose from the merge phase. In this experiment we choose the merge threshold and the restructuring interval to be 0.01% and 100 respectively, while the number of buckets is 100.

In Figure 5.8, we present the effectiveness of the split threshold for several degrees of skew of the query range distribution. As we can see, for this type of distribution the split threshold parameter does not play a crucial role even if the degree of skew is very high. In the case where we have low degrees of skew (Fig.5.8(a)), the split threshold does not play a crucial role, because all the value frequencies are nearly the same. Thus, in this case, the splitting of buckets does not have a particular importance in the histogram accuracy. In addition, when we have high degrees of skew (Fig.5.8(b)), the selection of the *st* is also not crucial. This happens because we choose the correlation between a query range value and a frequency to be such that the values with high frequencies are adjacent. Hence, the values with high frequencies are either in the same bucket or in a small fraction of the buckets. We are lead to the same conclusions when we use the Normalized weighted absolute error as error measure. Figure 5.9 depicts the effect of the split threshold in the accuracy of the W-ST histogram for several degrees of range skew. Thus, we choose a split threshold value of 10% of the W-ST buckets.

Overall, the values we select for the merge threshold, the split threshold and the restructuring interval that leads in increasing the W-ST histogram's efficiency is 0.01%, 100 and 10% respectively.



(a) (b)

Figure 5.8: Effect of the Split Threshold (*st*) in the Accuracy of the W-ST Histogram for (a) Low Degrees and (b) High Degrees of Range Skew Using the *Normalized Absolute Error*.

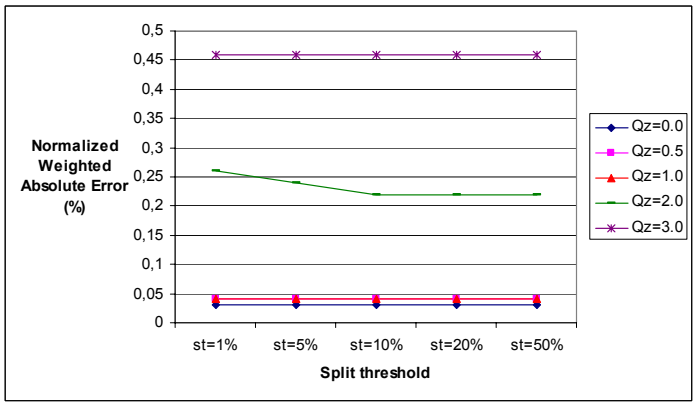


Figure 5.9: Effect of the Split Threshold (*st*) in the Accuracy of the W-ST Histogram Using the *Normalized Weighted Absolute Error*.

5.3.2. Comparison of W-Equi-Width and W-ST Histograms

In this section we investigate the effect of the number of histogram buckets in the accuracy of both W-Equi-Width and W-ST histograms for query ranges distributions with varying degrees of skew. The values that we choose for the refinement parameters are those that we selected in the previous section, i.e., *mt* = 0.01%, *ri* = 100 and *st* = 10%. In Figures 5.10 and 5.11, we demonstrate the errors produced, using the normalized absolute error measure, by these two types of histograms when the number of buckets (*b*) varies from 10 to 100 buckets and for several query range skews. Similarly, in Figures 5.12 (5.14) and 5.13 (5.15) we compare the W-Equi-Width and W-ST histograms as in Figures 5.10 and 5.11 respectively but using the normalized weighted absolute error (*sum absolute error* (*SAE*)) as accuracy measure. The sum absolute error is defined as:

$$SAE(D,H,QW) = \sum_{i \in D} \left| est(H,i) - act(QW,i) \right| \qquad (5.4)$$

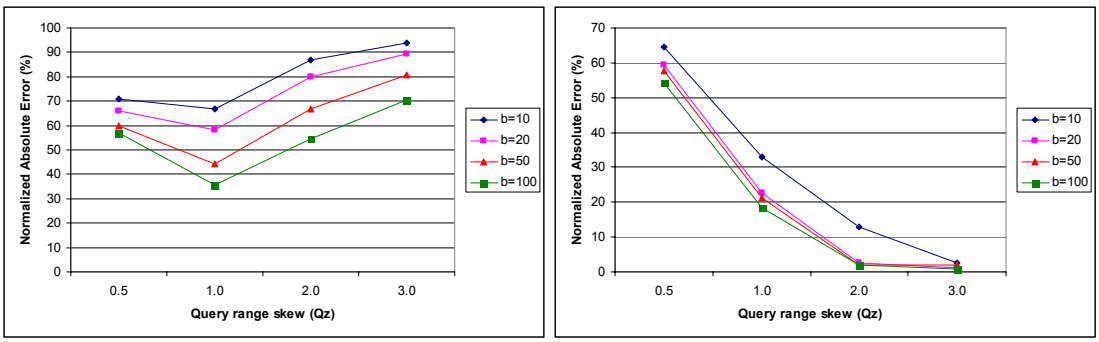
The results are similar for all the error measures that we use. In particular, both W-Equi-Width and *W-ST* histograms perform similarly for low degrees of skew, e.g., *Qz* = 0.5, because the variance between the frequencies of the values is not high. In contrast, the W-Equi-Width histogram does not perform well for high degrees of skew, e.g., *Qz* = 2.0 or 3.0. This happens because the bucket boundaries of the W-Equi-Width are predetermined, hence it is not possible to restructure the buckets to avoid grouping high frequency and low frequency values in the same buckets. In particular, when the number of buckets is small, e.g., 10, grouping high frequency with low frequency values in the same bucket is inevitable. In contrast, the W-ST histogram moves the bucket boundaries to get a better partitioning; hence it performs much better than the W-Equi-Width histogram.

Table 5.2: Parameters of the *W-ST* Histogram.

W-ST Histogram Parameters	**Default Value**
Number of buckets (*b*)	100
Merge threshold (*mt*)	0.01%
Split threshold (*st*)	10%
Restructuring Interval (*ri*)	100

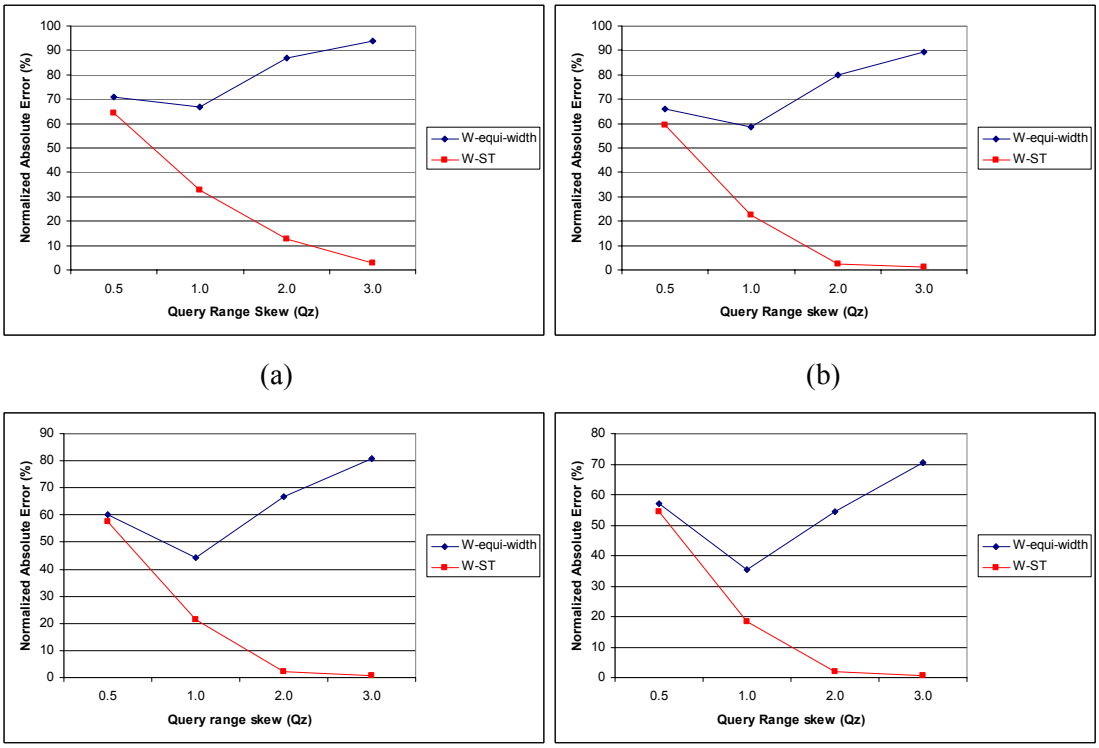
In addition, in both histograms when we increase the number of buckets, the accuracy gets better as expected. For the W-ST histogram, this happens until some point; meaning that even if we increase the number of buckets, the generated errors are roughly similar. This is much more evident for high degrees of skew in the query range distribution and happens because the high frequency values, which cause the largest amount in the error, are already isolated into singleton buckets. Overall, the W-ST histogram is much more efficient than the W-Equi-Width histogram.

We select the W-ST histogram with 100 buckets to be the histogram that is going to be used as query workload synopsis because we want to have high accuracy in approximating each kind of query workload. The selected values of the parameters, which determine the W-ST histogram, are summarized in Table 5.2. Thus, the selected number of buckets for each local index, its local query workload synopsis and its routing indexes is the same, i.e., 100.



(a) (b)

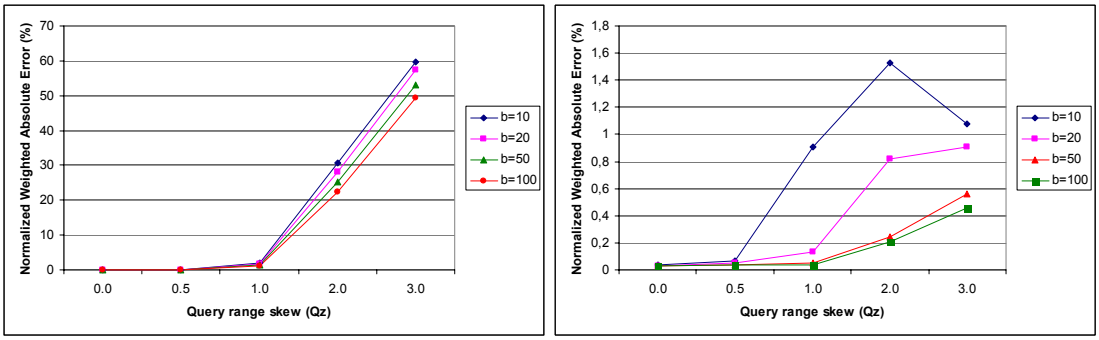
Figure 5.10: (a) *W-Equi-width* and (b) *W-ST* Accuracy when Varying the Query Range Skew (*Qz*) and the Number of Buckets Using the *Normalized Absolute Error* as Accuracy Measure.



(a) (b)

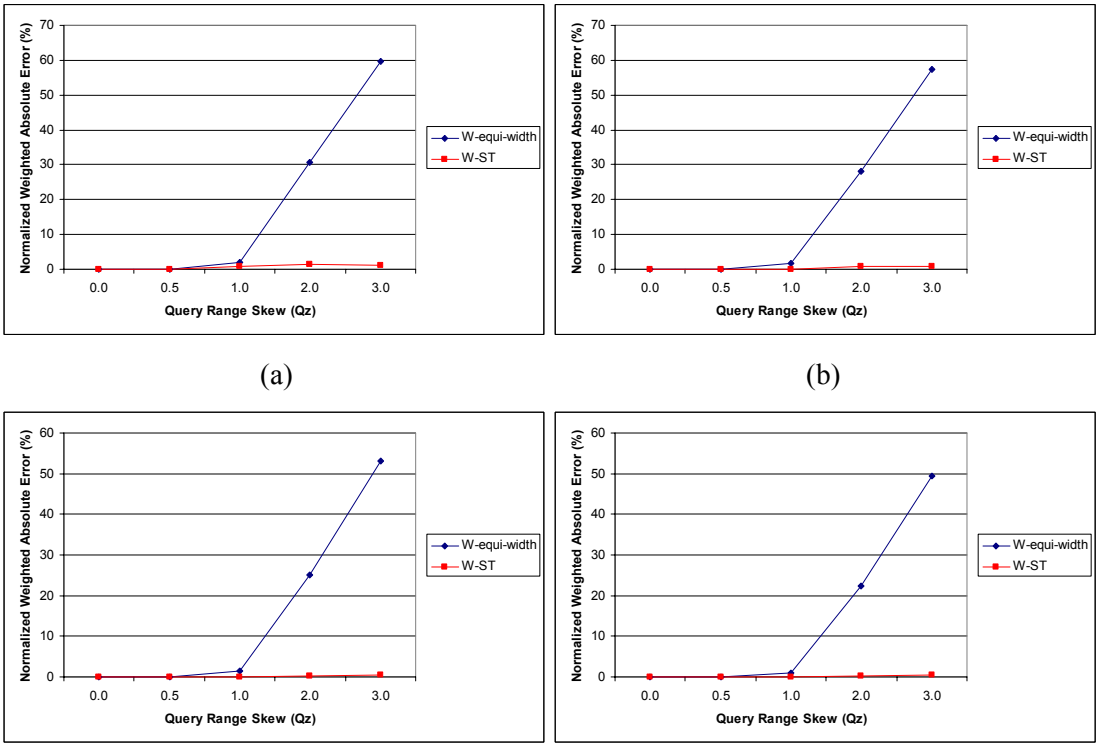
(c) (d)

Figure 5.11: *W-Equi-Width* and *W-ST* Accuracy, Using the *Normalized Absolute Error*, when Varying the Query Range Skew (*Qz*) and the Number of Buckets Takes the Values (a) 10, (b) 20, (c) 50 and (d) 100.



(a) (b)

Figure 5.12: (a) *W-Equi-Width* and (b) *W-ST* Accuracy when Varying the Query Range Skew (*Qz*) and the Number of Buckets Using the *Normalized Weighted Absolute Error* as Accuracy Measure.



(a) (b)

(c) (d)

Figure 5.13: *W-Equi-Width* and *W-ST* Accuracy, Using the *Normalized Weighted Absolute Error*, when Varying the Query Range Skew (*Qz*) and the Number of Buckets Takes the Values (a) 10, (b) 20, (c) 50 and (d) 100.

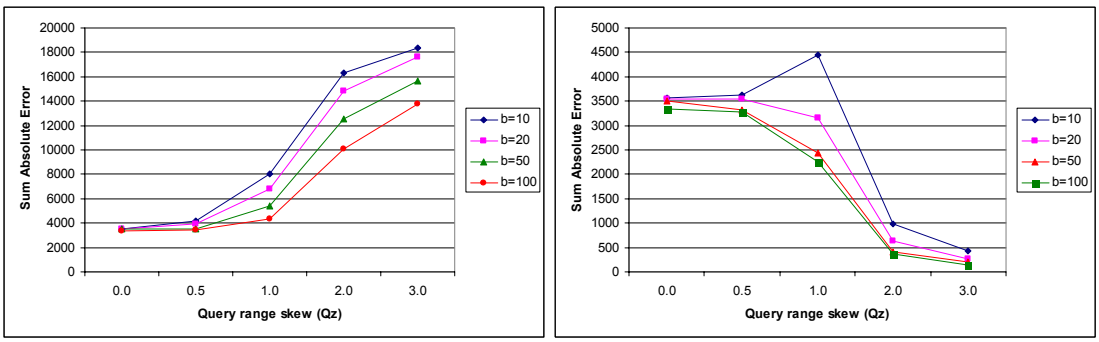


Figure 5.14: (a) *W-Equi-Width* and (b) *W-ST* Accuracy when Varying the Query Range Skew (*Qz*) and the Number of Buckets Using the *Sum Absolute Error* as Accuracy Measure.

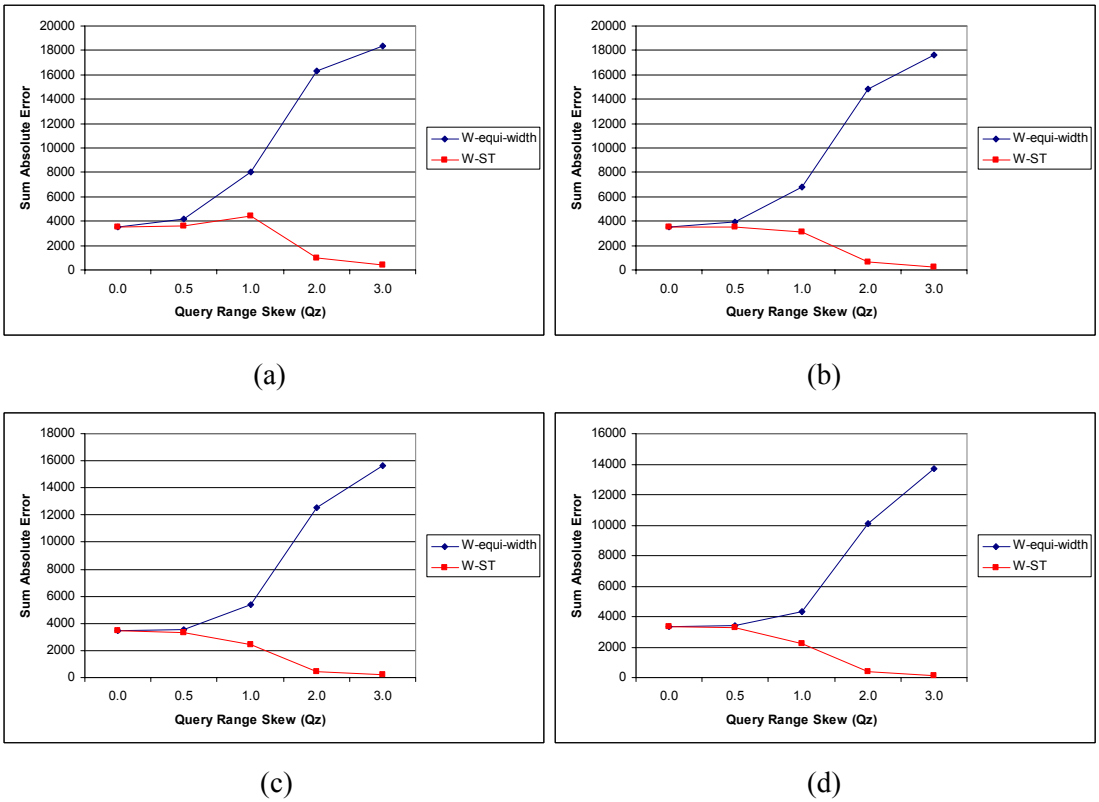


Figure 5.15: *W-Equi-Width* and *W-ST* Accuracy, Using the *Sum Absolute Error*, when Varying the Query Range Skew (*Qz*) and the Number of Buckets Takes the Values (a) 10, (b) 20, (c) 50 and (d) 100.

5.3.3. Effect of the Restructuring Process

As we have mentioned in Section 5.1.2, refining only the histogram frequencies is not enough to get an accurate histogram. Thus, the process of restructuring, i.e., moving the bucket boundaries to get a better partitioning that avoids grouping high frequency and low frequency

values in the same buckets, plays a crucial role in the accuracy of the W-ST histogram. In this section, we study the importance of the restructuring process for "capturing" the distribution that the query ranges follow.

In particular, we issue a query workload of 10000 queries, one query at a time, and create the W-ST histogram. Furthermore, for every set of M queries we measure, independently from the other query sets, its absolute estimation error. Our goal is to see the extend of the contribution of the restructuring process to the reduction of the absolute error. Note, that the values for the number of buckets, the merge threshold and the split threshold are those mentioned in Table 5.2. In addition, we also set the restructuring interval equal to M queries, i.e., for every set of M queries we restructure the W-ST histogram. We set the number of queries in each set equal to the restructuring interval, i.e., measuring the sum absolute error each time the restructuring process starts, so as to see more clearly the effect of the restructuring process on approximating the query workload. To verify that the restructuring process does indeed reduce the error, we also compute the sum absolute error of the query sets in the cases when assuming uniformity and using a W-Equi-Width histogram. These errors are plotted in Figures 5.16(a) - (e) and 5.17(a) - (e) for several degrees of query ranges skew when each set of queries consists of $M = 1000$ and $M = 100$ queries, respectively.

There are several conclusions that arise. Initially, we deal with the errors produced when the restructuring process is invoked in every 1000 queries. Figures 5.16(a) - (e) show that the W-ST histogram restructuring converges fairly rapidly. Especially, for the first set of queries when the restructuring process has not been invoked yet, the absolute error is high and the same with the one of the W-Equi-Width histogram. This happens because initially the bucket boundaries are fixed according to the equi-width heuristic; hence they are the same with those of the W-Equi-Width histogram. Recall that the difference between a W-Equi-Width and a W-ST histogram is the restructuring process. After the first restructuring process, the absolute error is sufficiently reduced and is nearly the same for all the query sets that follow. This means that query workload distribution is almost "captured" after the first restructuring process and shows the importance of this process. This is much more evident when the skewness of the query workload is high, e.g., $Qz = 2.0$ or 3.0. For low degrees of skew, e.g., $Qz = 0.0$ or 0.5, the absolute errors that are produced by the query sets are nearly the same due to the nature of the distribution; hence the restructuring process is not critical. Thus, in general the restructuring process has to be performed only a small number of times before the histogram becomes sufficiently accurate, i.e., "learns" the distribution.

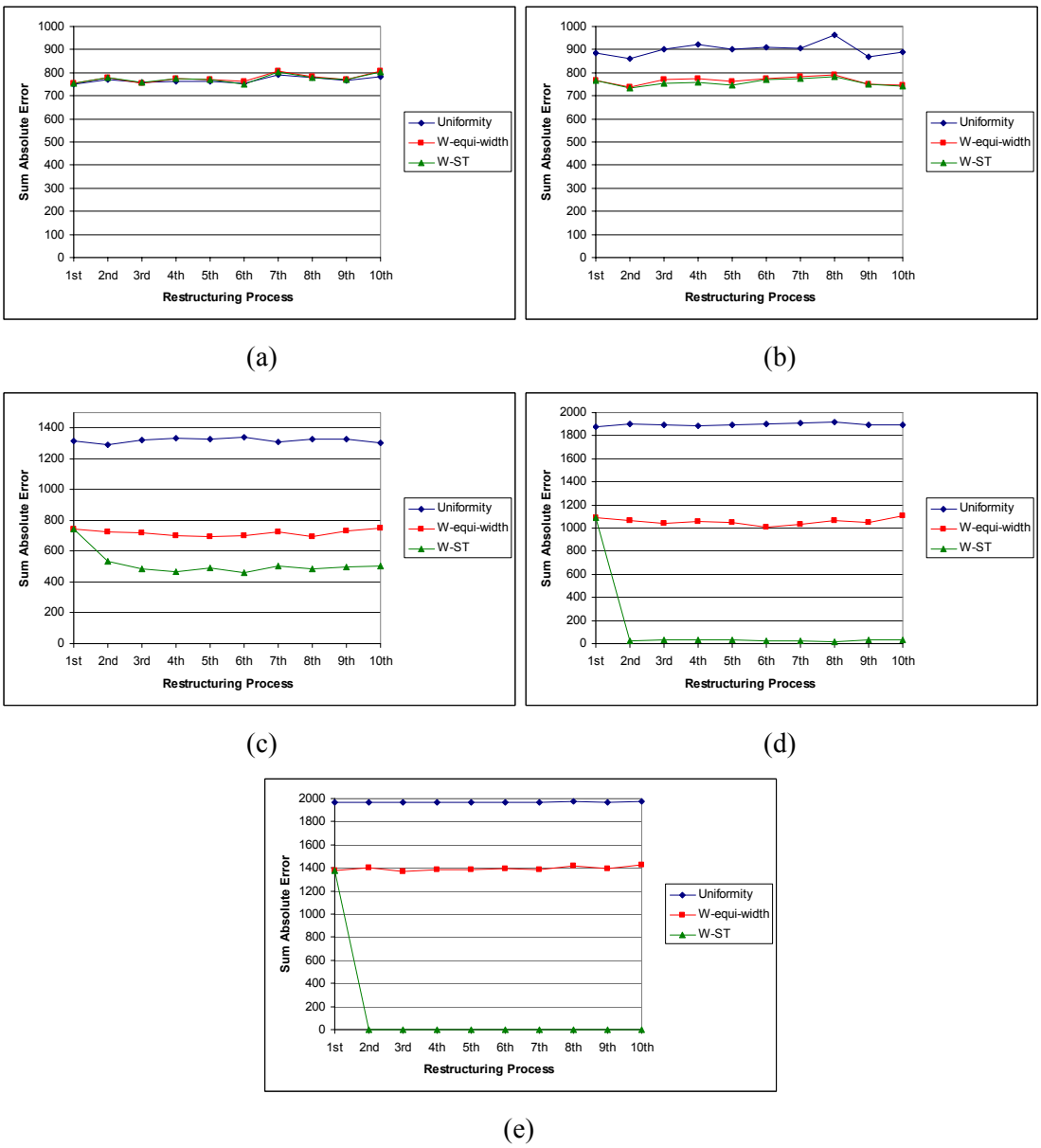


Figure 5.16: Effect of Restructuring Process when the Query Range Skew Takes the Values (a) $Qz = 0.0$, (b) $Qz = 0.5$, (c) $Qz = 1.0$, (d) $Qz = 2.0$ and (e) $Qz = 3.0$ and the Restructuring Process is Invoked in every 1000 Queries.

The previous experiment showed that the W-ST histogram almost captures the query workload distribution from the first restructuring process. This happens because the number of queries, $M = 1000$, between two restructuring processes is large enough to approximate the distribution of the query workload. The main issue that arises is what happens when the restructuring process is invoked after a small number of queries; since it is much more difficult to approximate the distribution using a small set of queries. The frequency of invoking the restructuring process is important. The higher the restructuring interval between

two restructuring processes, the larger the error that is produced before the first restructuring process is invoked in order to capture the distribution. Therefore, the more often the restructuring process takes place, the faster the W-ST histogram is going to capture the query workload distribution; hence it will become more accurate. Furthermore, if the restructuring process takes place rarely, then it is possible that the query workload distribution changes between two restructuring processes. Thus, the W-ST histogram will become inaccurate.

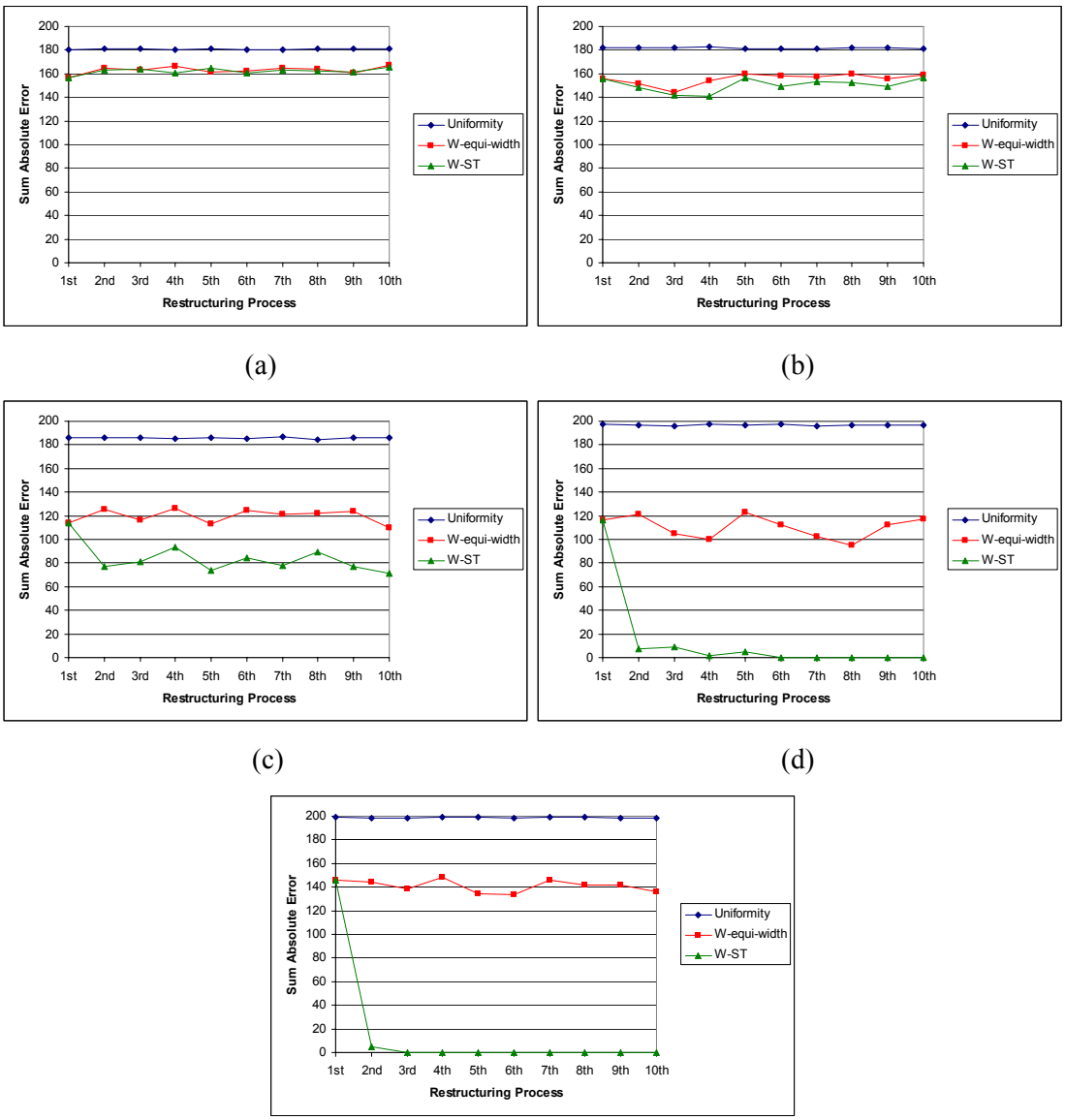


Figure 5.17: Effect of Restructuring Process when the Query Range Skew Takes the Values (a) $Qz = 0.0$, (b) $Qz = 0.5$, (c) $Qz = 1.0$, (d) $Qz = 2.0$ and (e) $Qz = 3.0$ and the Restructuring Process is Invoked in every 100 Queries.

Therefore, we set the restructuring interval and the number of queries in each set equal to 100. In Figures 5.17(a) - (e), where the restructuring process is invoked in every 100 queries, we demonstrate the absolute errors for the first 10 of 100 restructuring processes, since the errors for the rest of the restructuring processes is nearly the same. The conclusions are the same with those mentioned before. Specifically, the W-ST histogram may not "capture" the query distribution from the first restructuring process but this happens after a few restructuring processes. This happens because the queries that are issued are very few so as to represent the distribution.

Summarizing, the restructuring process is essential for capturing the query workload distribution especially when its skewness is very high. Furthermore, in order to capture the distribution as quick as possible the restructuring interval between two restructuring processes must not be very long.

5.3.4. Evaluation of the W-ST Histogram Using the Aging Technique

In this section, we experimentally evaluate how fast the W-ST histogram adapts to the query workload changes using the aging technique. In particular, we create a query workload that consists of 10000 queries and change its distribution every *qd_up* queries. For simplicity, we consider that the value of the *qd_up* factor is a multiple of the value of the restructuring interval parameter, *ri*, which is 100; hence, distribution changes after a restructuring process takes place. In addition, we assume that the distribution of the query workload changes randomly. Recall, that our distribution for the query workload follows the Zipfian distribution, hence it depends on two parameters: the query range that is the most frequent one, *Hqr*, and the skewness of the distribution, *Qz*. By saying that the distribution of the query workload changes randomly, we mean that a new query distribution is created by selecting randomly the most frequent range value and its skewness. In our experiments, skewness takes values from the domain [0.0, 3.0] and *Hqr* from the value domain *D*.

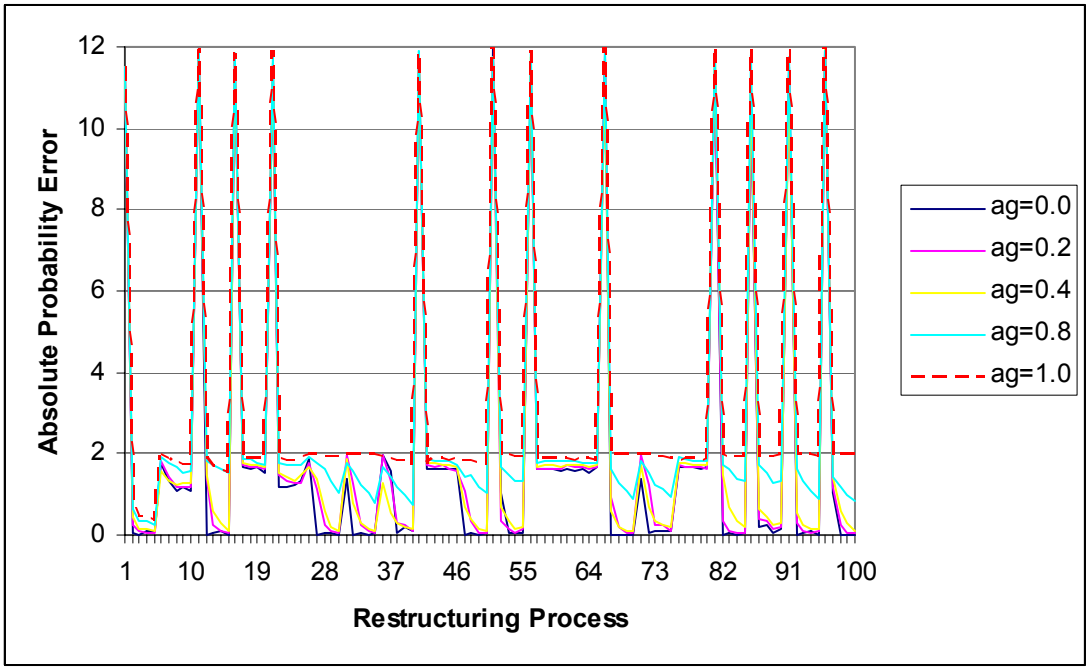
Ideally, we would like the W-ST histogram to approximate efficiently the current distribution that the query workload follows. To measure the accuracy of our histogram we calculate the error produced after each restructuring process. A common error measure is the *sum absolute probability error* (*SAPE*)), which is defined as follows:

$$SAPE(D, H, CQW) = \sum_{i \in D} \left| \tilde{p}(H, i) - p(CQW, i) \right| \qquad (5.5)$$

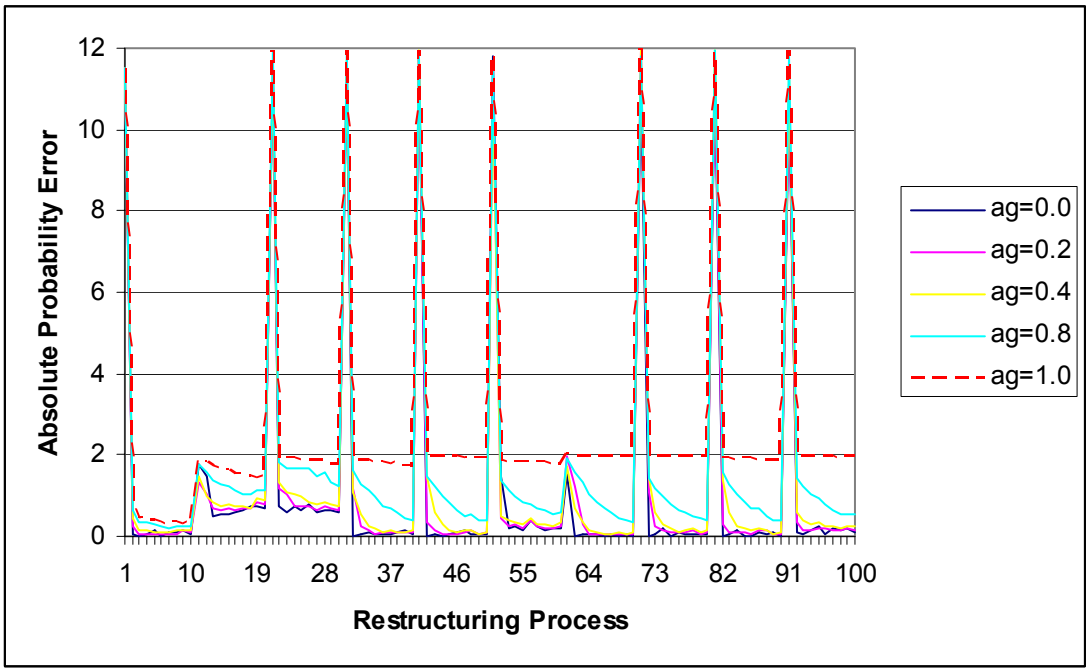
where *CQW* is the query workload between two restructuring processes, i.e., the current query workload, $\tilde{p}(H,i)$ is the estimated probability of the query with range *i*, using the histogram, and $p(CQW,i)$ is the actual probability of the query in the current query workload *CQW*.

In Figure 5.18, we demonstrate the effect of the aging factor in capturing the distribution that the queries follow. In particular, we depict the errors produced when the aging factor takes several values from 0.0 to 1.0 and when the distribution changes in every qd_up queries. In our experiments, we set the qd_up parameter equal to $5*ri$ Fig. 5.18(a), $10*ri$ Fig. 5.18(b) and $50*ri$ Fig. 5.18(c), i.e., the query distribution changes in every 500, 1000 and 5000 queries respectively. The results are similar for all values of the qd_up parameter. Initially, the W-ST histogram captures the first query distribution very fast and performs similarly for all degrees of aging, i.e., using the aging technique or not the produced errors are similar. This shows that even if we set the aging factor close to 0.0, i.e., we drop all information about past queries that follow the same query distribution, the histogram still performs well. This happens for two reasons. The first one is that even if we have a small sample of queries between two restructuring processes, this sample seems to be capable of approximating the current distribution. Secondly, after a restructuring process the aging technique may reduce the frequency of the buckets but the bucket boundaries remain unchanged. Thus, the bucket boundaries are fixed in that way so as to capture efficiently the distribution of the future queries. This assumes that the future queries, i.e., the queries that are initiated in the next restructuring interval, follow the same distribution with those that were initiated in the previous restructuring interval.

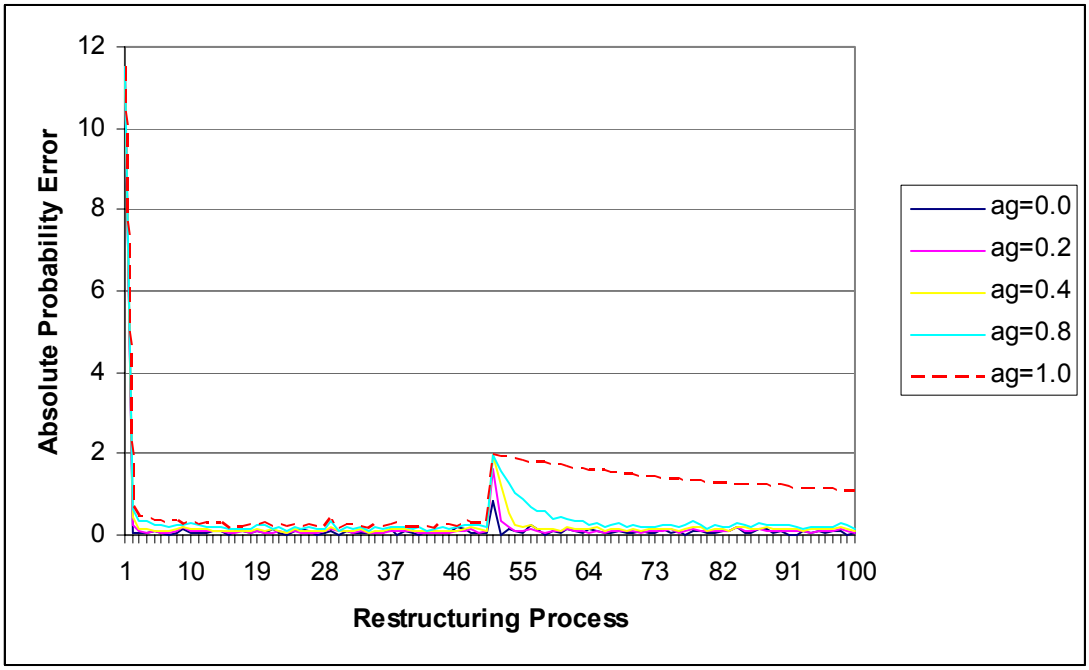
In addition, when the distribution of the queries changes the errors that are produced by the W-ST histogram increase for all degrees of aging, since the histogram cannot capture immediately this change. However, when the aging factor is close to 0.0 the histogram captures faster this change. This happens because the histogram reduces a lot the information about past queries; hence the previous distribution of the queries. In contrast, when we do not use the aging technique, ag = 1.0, the learning rate of the new distribution is much lower and leads to larger errors in comparison with the case we use aging.


Absolute Probability Error
0
2
4
6
8
10
12
1
10
19
28
37
46
55
64
73
82
91
100
Restructuring Process
ag=0.0
ag=0.2
ag=0.4
ag=0.8
ag=1.0

(a)


Absolute Probability Error
0
2
4
6
8
10
12
1
10
19
28
37
46
55
64
73
82
91
100
Restructuring Process
ag=0.0
ag=0.2
ag=0.4
ag=0.8
ag=1.0

(b)



(c)

Figure 5.18: Accuracy of the W-ST Histogram when Varying the Aging Factor (*ag*) and the Query Distribution Changes in every (a) 500, (b)1000 and (c) 5000 Queries.

Overall, the W-ST histogram performs better when we use an aging technique since it is adaptive to changes of the query workload distribution. Furthermore, we select 0.0 to be the appropriate value for the aging factor since the histogram does not seem to behave worse in comparison with the case when we do not use aging and the query distribution does not change while when the distribution changes the histograms adapts much more quickly to this change.

5.4. Summary

To conclude, in this chapter to take into account the query workload for the formation of clusters we consider using histograms for summarizing the query workload. In particular, we initially propose the W-Equi-Width histogram. The main drawback of the W-Equi-Width histogram is that it does not have the potential to avoid grouping in the same bucket query ranges which are frequent with ranges that are less frequent. Thus, we introduce a novel approach, the W-ST histograms and present an algorithm for their construction. In addition, we extend the W-ST construction procedure by using an aging technique so as to adapt to changes of the query workload.

We make an extensive experimental evaluation about the effect of the refinement parameters in the accuracy, i.e., the merge threshold, the split threshold and the restructuring interval, of the W-ST histogram and we select the appropriate values for each one of them that lead in increasing the accuracy of the histogram. We experimentally evaluate the accuracy of these two histogram types for several kinds of query workloads in the cases when the number of buckets varies. The main conclusions from our experiments show that the W-ST histogram performs better than the W-Equi-Width histogram, especially in cases when the skewness of the query distribution is high. Furthermore, we study the importance of the restructuring process for "capturing" the distribution that the query ranges follow. The experiments show that this process is critical for the accuracy of the W-ST histogram since after a few restructuring processes the W-ST histogram "captures" the query distribution and becomes accurate. Finally, we assess the usage of the aging technique to adapt the changes of the query workload distribution. The experiments show that using the aging technique in conjuction with the W-ST histogram leads in much lower estimation errors in comparison with the case when we do not use this technique; hence the rate of learning the new distribution is much higher.

CHAPTER 6. HISTOGRAM DISTANCE METRICS

As discussed in Chapters 4 and 5, histograms over one attribute are used to summarize the peer content and their local query workload. In addition, in Chapter 3 we conclude that the distance measure used to define the similarity between pairs of peers plays a crucial role in clustering, hence in the performance of the p2p network. In this chapter, we focus on creating clusters of peers using their local indexes and query workload synopsis. We introduce histogram-based distance measures and we evaluate their performance.

6.1. Histogram-Based Workload-Aware Property

Assume that we have a query workload *QW*. *QW* may be either the global query workload *W* or the local query workload of a peer *n*, *LW(n)*. To construct workload-aware overlays, we propose using local indexes and the synopsis of the query workload *QW*. In particular, we create clusters of peers that have similar local indexes taking into account a query workload *QW*. For this to work, the similarity between the local indexes of two peers, which is provided by a distance measure, must be descriptive of "how much" the two peers match the workload *QW*. In addition, as discussed in the previous chapters, we use histograms as local indexes

and local query workload synopses. Hence, we need to introduce histogram-based workload-aware distance measures (*hd*) to define the similarity between pairs of peers. Furthermore, in Chapter 3, we concluded that when we have the whole information about the content of the peers and the query workload, the *Manhattan Workload-Aware* distance measure achieves the best clustering which in many occasions is close to optimal. Thus, we consider that the histogram-based workload-aware distance measures must be an approximation of this distance measure. In particular, the distance (*hd*) between two histograms must be descriptive of the difference in the number of results for a query workload *QW*.

Property 6.1 (Histogram-Based Workload-Aware Property) *A histogram distance metric hd between two histograms, $H(n_1)$ and $H(n_2)$, is said to be workload-aware for a query workload QW, if for any three peers n_1, n_2, n_3,*

$$\frac{hd(H(n_1),H(n_2),QW)}{hd(H(n_1),H(n_3),QW)} \propto \frac{wd_{L_1}(n_1,n_2,QW)}{wd_{L_1}(n_1,n_3,QW)} = \frac{\sum_{q\in QW} p_q \left|results(n_1,q)-results(n_2,q)\right|}{\sum_{q\in QW} p_q \left|results(n_1,q)-results(n_3,q)\right|}$$

If the distance between the histograms $H(n_1)$ and $H(n_2)$ of peers n_1 and n_2 is smaller than the distance between the histograms $H(n_1)$ *and* $H(n_3)$ of peers n_1 and n_3 for a query workload *QW*, we want also the difference in the number of results provided by n_1 and n_2 for *QW* to be smaller than the corresponding difference in the number of results provided by n_1 and n_3.

6.2. Histogram Distance Metrics

We consider two well-known distance metrics (namely the L_1 and the *edit* distance) between two histograms and propose a weighted version of the edit distance, which satisfies the workload-aware property. Recall that, we assume that the histograms we use as local indexes and as local query workload synopsis have the same number of *b* buckets. In general, a histogram *H(n)* can be represented as a vector where each vector's feature *i* represents the total frequency of the values that lie within the value range of the corresponding bucket *i*, i.e., $H_i(n)$. To compare two histograms (vectors) with the same number *b* of buckets, the *i*-th bucket (feature), $0 \le i \le b-1$, for both histograms must be responsible for the same range of values. In particular, consider two histograms, $H(n_1)$ and $H(n_2)$, with *b* buckets where for each bucket *i* the value range for which it is responsible is the same in both histograms. Then, the L_1 and the edit distance between these two histograms can be defined as follows:

Definition 6.1 (L_1 distance between histograms)

Let $H(n_1)$ and $H(n_2)$ be two histograms with b buckets, their L_1 distance, $hd_{L_1}(H(n_1), H(n_2))$, is defined as: $hd_{L_1}(H(n_1), H(n_2)) = \sum_{i=0}^{b-1} |H_i(n_1) - H_i(n_2)|$.

Definition 6.2 (*Edit* distance between histograms)

Let $H(n_1)$ and $H(n_2)$ be two histograms with b buckets, their edit distance, $hd_{edit}(H(n_1), H(n_2))$, is defined as:

$$hd_{edit}(H(n_1), H(n_2)) = \sum_{i=0}^{b-1} \left| \sum_{j=0}^{i} H_j(n_1) - \sum_{j=0}^{i} H_j(n_2) \right|.$$

Let us define as:

$$pref(l) = \begin{cases} \sum_{i=0}^{l} H_i(n_1) - \sum_{i=0}^{l} H_i(n_2) & if\ 0 \le l < b \\ 0 & otherwise \end{cases}$$

Then, the edit distance can be written as:

$$hd_{edit}(H(n_1), H(n_2)) = \sum_{i=0}^{b-1} |pref(i)|.$$

The L_1 and the edit distance, as introduced before, can be directly applied to equi-width histograms. However, they cannot be applied to other types of histograms, e.g., maxdiff histograms, because the bucket boundaries of two histograms might be different. Thus, we have to extend them so that they can be used for these types of histograms as well.

6.2.1. Extended Histogram Distance Metrics

Consider that we have two histograms with *b* buckets on an attribute *x* and for each bucket *i* the bucket boundaries of the two histograms might be different. As we discussed in Section 4.2.1 the number of all possible regions that are intersections of the bucket ranges of the two histograms is between [*b*, *2b-1*]. Thus, to apply the L_1 and the edit distance between two histograms, we follow the same strategy used for merging two histograms. In particular, we find all the *intervals*, which are intersection of the two histogram bucket regions. Let *C* be the number of these intervals and a histogram *H(n)*. The frequency of each interval *i*, $0 \le i < C$, is calculated based on *H(n)* using the *uniform frequency* assumption and the *continuous values* assumption. We denote by $F_i(n)$ the frequency of the interval *i* for peer *n*. Assume that interval *i* is responsible for a range *k* of values. Then, the interval *i* is responsible for the value domain $[v_i, v_i+k-1]$, where v_i is the first ordered value that is contained in this interval. Hence,

$F_i(n)$ is defined as: $F_i(n) = \sum_{j=v_i}^{v_i+(k-1)} \tilde{f}_j(n)$, where $\tilde{f}_j(n)$ is the approximated frequency of value *j* provided by *H(n)*. Thus, the *Extended-L1* and the *Extended-edit* distance are defined as:

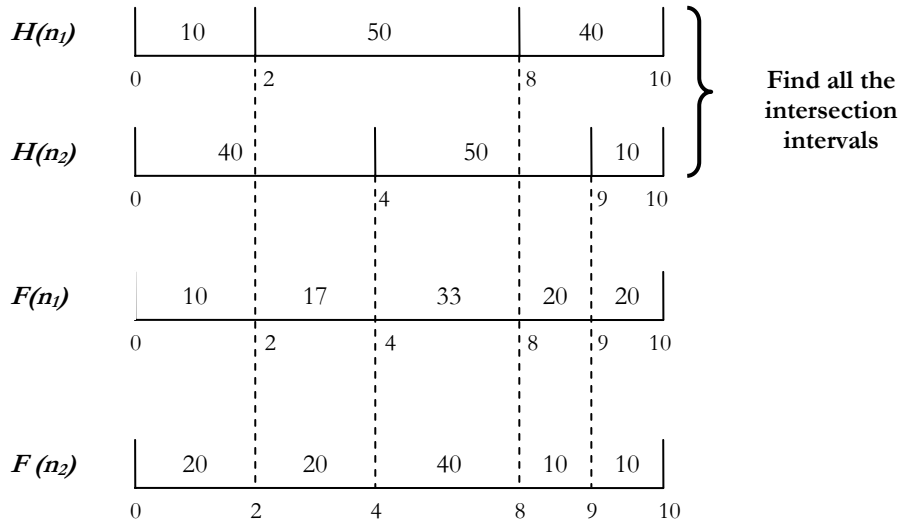


Figure 6.1: Example of Dividing into five Intervals each one of the Histograms $H(n_1)$, $H(n_2)$ with three Buckets over an Attribute $x \in [0,10)$.

Definition 6.3 (*Extended-L1* distance between histograms)

Let $H(n_1)$ and $H(n_2)$ be two histograms with b buckets, and C be the number of their intersection intervals. Their Extended-L1 distance, $hd_{EL_1}(H(n_1), H(n_2))$, is defined as:

$$hd_{EL_1}(H(n_1), H(n_2)) = \sum_{i=0}^{C-1} |F_i(n_1) - F_i(n_2)|.$$

Definition 6.4 (*Extended-Edit* distance between histograms)

Let $H(n_1)$ and $H(n_2)$ be two histograms with b buckets, and C be the number of their intersection intervals. Their Extended-Edit distance, $hd_{Eedit}(H(n_1), H(n_2))$, is defined as:

$$hd_{Eedit}(H(n_1), H(n_2)) = \sum_{i=0}^{C-1} \left| \sum_{j=0}^{i} F_j(n_1) - \sum_{j=0}^{i} F_j(n_2) \right|.$$

Let us define as:

$$pref\_F(l) = \begin{cases} \sum_{i=0}^{l} F_i(n_1) - \sum_{i=0}^{l} F_i(n_2) & if\ 0 \le l \le C\text{-}1 \\ 0 & otherwise \end{cases}$$

Then, the Extended-edit distance can be written as:

$$hd_{Eedit}(H(n_1), H(n_2)) = \sum_{i=0}^{C-1} |pref\_F(i)|.$$

The *Extended-Edit* distance takes the prefix sums of the intervals of the histograms $H(n_1)$ and $H(n_2)$.

Example 6.1: *Figure 6.1 depicts an example of two histograms, $H(n_1)$ and $H(n_2)$, with three buckets and with five intersection intervals. The distance between $H(n_1)$ and $H(n_2)$ using the Extended-L_1 and the Extended-Edit distance measures is:*

$$hd_{EL_1}(H(n_1), H(n_2)) = |10-20| + |17-20| + |33-40| + |20-10| + |20-10| = 40 \text{ and}$$

$$hd_{Eedit}(H(n_1), H(n_2)) = |10-20| + |27-40| + |60-80| + |80-90| + |100-100| = 53,$$

respectively.

From now on, we will refer to the *Extended-L_1* and the *Extended-edit* distances as L_1 and e*dit* distances, respectively. The L_1 and the edit distances depend only on the local indexes without taking into account the query workload *QW*. In the following section, we introduce a workload-aware histogram distance measure, based on the edit distance, which takes into account the query workload.

6.2.2. Workload-Aware Histogram Distance Metrics

To take into account the query workload *QW*, based on Definition 6.4 we define a workload-aware variation of the *edit* distance metric. Consider that the query workload *QW* is summarized by the histogram *H(QW)* with *b* buckets, as the histograms used as local indexes. To measure the distance between $H(n_1)$ and $H(n_2)$, we want also to use *H(QW)*. Recall that all three histograms are very likely to have different bucket boundaries. Thus, the main issue is how to define common intervals between $H(n_1)$ and $H(n_2)$ and *H(QW)* so as to measure the histogram distance. There are several approaches to address this problem. The first one is the strategy we followed in defining the Extended-L_1 and Extended-edit distances. This approach is not convenient in this case because selecting the intervals based on $H(n_1)$ and $H(n_2)$ might lead in high loss of information of *H(QW)*. Another approach is to define the intervals based solely on the *H(QW)* bucket boundaries. Also this approach has some drawbacks. The first one is that it might lead to high inaccuracy of $H(n_1)$ and $H(n_2)$. Furthermore, if we pose the same query workload twice the bucket boundaries of *H(QW)* might be different because the construction of *H(QW)* depends on the sequence of the queries. Hence, the similarity between two peers might be different for the same *QW*. To avoid these problems, we follow an

intermediary approach. We propose to divide the value domain of each histogram into (*2b*), if $2b < |D|$ otherwise $2b = |D|$, intervals of equal width. In addition, we denote by $p_i(QW)$ the probability of the interval *i* of the query workload *QW* synopsis, i.e., $p_i(QW) = F_i(QW) / S(H(QW))$ where *S(H(QW))* denotes the size of the histogram *H(QW)*. Recall that, we assumed the query workload consists of range queries whose ranges follow a distribution and the starting point of each query is chosen at random. Hence, the histogram *H(QW)* approximates the query range distribution. In this case, the *Workload-Aware Edit* distance between histograms (*wedit*) is defined as follows:

Definition 6.5 (Workload-Aware Edit (*wedit*) distance between histograms)
Let $H(n_1)$ and $H(n_2)$ be two histograms of peers n_1 and n_2 respectively and H(QW) be the histogram for the query workload QW. All the histograms have b buckets. The Workload-Aware Edit distance (wedit), $hd_{wedit}(H(n_1), H(n_2), H(QW))$, is defined as:

$$hd_{wedit}(H(n_1), H(n_2), H(QW)) = \sum_{i=0}^{2b-1} p_i(QW) \sum_{j=0}^{2b-1} \left| \sum_{k=0}^{i} F_{k+j}(n_1) - \sum_{k=0}^{i} F_{k+j}(n_2) \right|.$$

In addition, we claim that for a given j, the following equation holds.

$$\sum_{k=0}^{i} F_{k+j}(n_1) - \sum_{k=0}^{i} F_{k+j}(n_2) = pref\_F(i+j) - pref\_F(j-1).$$

<u>*Proof*</u>

$$pref\_F(i+j) - pref\_F(j-1) = \sum_{l=0}^{i+j} \left(F_l(n_1) - F_l(n_2)\right) - \sum_{l=0}^{j-1} \left(F_l(n_1) - F_l(n_2)\right) =$$
$$= \sum_{l=0}^{j-1} \left(F_l(n_1) - F_l(n_2)\right) + \sum_{l=j}^{i+j} \left(F_l(n_1) - F_l(n_2)\right) - \sum_{l=0}^{j-1} \left(F_l(n_1) - F_l(n_2)\right) =$$
$$= \sum_{l=j}^{i+j} \left(F_l(n_1) - F_l(n_2)\right).$$

Also, let $k + j = l$. Then,

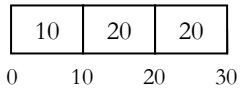
$$\sum_{k=0}^{i} F_{k+j}(n_1) - \sum_{k=0}^{i} F_{k+j}(n_2) = \sum_{k=0}^{i} \left(F_{k+j}(n_1) - F_{k+j}(n_2)\right) =$$
$$= \sum_{l=j}^{i+j} \left(F_l(n_1) - F_l(n_2)\right).$$

Hence, *the wedit* distance can be written as:

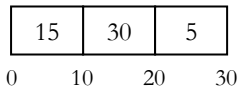
$$hd_{wedit}(H(n_1), H(n_2), H(QW)) = \sum_{i=0}^{2b-1} p_i(QW) \sum_{j=0}^{2b-1} \left| pref\_F(i+j) - pref\_F(j-1) \right|.$$

This metric computes for each range *i*, $0 \le i \le 2b-1$, of intervals the distance of each possible region, which starts from interval *j*, $0 \le j \le 2b-1$, and consists of *i* contiguous intervals in $H(n_1)$ from the corresponding region in $H(n_2)$ weighted by the *i*-th interval probability of *H(QW)*. We select each possible region for each range *i* because we have assumed that the starting point of each query is chosen uniformly.

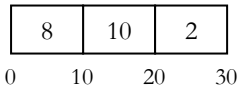
Example 6.2: *Consider that we have two peers, n_1 and n_2, and a query workload QW. In addition, we assume that the attribute's x value domain is split into three intervals and the frequency of each interval i, $0 \le i \le 2$, for peers n_1, n_2 and the workload QW results from $H(n_1)$* and *$H(n_2)$ and H(QW), respectively. Then, the similarity of n_1 and n_2 using the wedit distance measure is computed as follows:*



$F(n_1)$



$F(n_2)$



$F(QW)$

$$hd_{wedit}(H(n_1),H(n_2),H(QW)) = p_0(QW)(|F_0(n_1)-F_0(n_2)|+|F_1(n_1)-F_1(n_2)|+|F_2(n_1)-F_2(n_2)|)$$
$$+ p_1(QW)(|(F_0(n_1)+F_1(n_1))-(F_0(n_2)+F_1(n_2))|+|(F_1(n_1)+F_2(n_1))-(F_1(n_2)+F_2(n_2))|+$$
$$+|F_2(n_1)-F_2(n_2)|)+p_2(QW)(|(F_0(n_1)+F_1(n_1)+F_2(n_1))-(F_0(n_2)+F_1(n_2)+F_2(n_2))|+$$
$$+|(F_1(n_1)+F_2(n_1))-(F_1(n_2)+F_2(n_2))|+|F_2(n_1)-F_2(n_2)|) \Leftrightarrow$$
$$hd_{wedit}(H(n_1),H(n_2),H(QW)) = \frac{8}{20}*(|10-15|+|20-30|+|20-5|)+\frac{10}{20}*(|(10+20)-(15+30)|$$
$$+|(20+20)-(30+5)|+|20-5|)+\frac{2}{20}*(|(10+20+20)-(15+30+5)|+|(20+20)-(30+5)|+$$
$$+|20-5|) = \frac{8}{20}*(5+10+15)+\frac{10}{20}*(15+5+15)+\frac{2}{20}*(0+5+15) =$$
$$= \frac{8}{20}*30+\frac{10}{20}*35+\frac{2}{20}*20 = 31.5$$

In addition, wedit also holds for a special case of range queries denoted as *prefix-range* queries; that is for queries with one sided ranges: $QW = \{(q_{0j}, f_{q_{0j}})\}$ and for any pair of queries q_{0j} and q_{0l} in *QW* one is contained in the other. In particular, since the starting point of each prefix-range query is the 0, by setting *j* equal to 0, i.e., the starting interval of each region is always the first one, the *wedit* distance for prefix-range queries takes the form:

$$hd_{wedit}(H(n_1),H(n_2),H(QW)) = \sum_{i=0}^{2b-1} p_i(QW)\left|\sum_{k=0}^{i} F_k(n_1) - \sum_{k=0}^{i} F_k(n_2)\right| \Leftrightarrow$$
$$\Leftrightarrow hd_{wedit}(H(n_1),H(n_2),H(QW)) = \sum_{i=0}^{2b-1} p_i(QW)|pref\_F(i)|.$$

It has been shown that the *wedit* distance measure satisfies the requirements of a distance metric. The proof is given in the Appendix.

6.3. Discussion

The histograms that we study are *ordinal*, i.e., there exists an ordering among their buckets, since they are built on a numeric attribute. Similarly with the content based distance metrics that we have discussed in Section 3.5.1, for ordinal histograms the position of the buckets is important and thus, we want the definition of the histogram distance to satisfy the *shuffling dependence* property, i.e., to take into account this ordering. For example, consider the 3 equi-width histograms $H(n_1)$, $H(n_2)$ and $H(n_3)$. The histogram distance between $H(n_1)$ and $H(n_2)$ that have all their values at adjacent buckets (buckets 0 and 1 respectively) should be smaller than the distance between histograms $H(n_1)$ and $H(n_3)$ that have their values at buckets further apart (buckets 0 and 3 respectively). This is because, the difference between the number of results provided by peer n_1 and the number of results provided by peer n_2 is smaller for a larger number of different range queries than for peers n_1 and n_3. As we have discussed in Chapter 3, the shuffling dependence property does not hold for the L_1 distance, i.e., the three histograms have the same pair-wise distance, while it holds for the *edit* distance. In particular, the *edit* distance between two histograms $H(n_1)$ and $H(n_2)$ is the total number of all necessary minimum movements for transforming $H(n_1)$ to $H(n_2)$ by moving elements to the left or right.

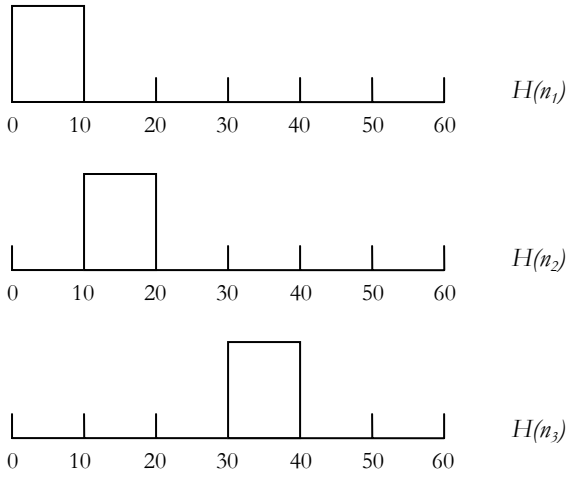


Figure 6.2: The Distance between $H(n_1)$ and $H(n_2)$ should be Smaller than the Distance between $H(n_1)$ and $H(n_3)$.

However, although, the *edit* distance is shuffling dependent still it does not take into account the probability of queries, i.e., the query workload. Overall, the L_1 distance does not satisfy the workload-aware property (Property 6.1), while the *edit* distance satisfies it in special cases when the query workload does not influence the pair-wise distance of peers by examining

only their contents. Recall that the workload-aware property requires that the distance between the histograms of two peers be descriptive of the difference in the number of results the peers provide for a workload *W*. In the ideal case, we want the distance between two histograms $H(n_1)$ and $H(n_2)$ of peers n_1 and n_2 to correspond to the difference of matching results provided by n_1 and n_2. In the Appendix, we show that the *wedit* distance satisfies the Property 6.1 for a workload $W = \{(q_{ij}, f_{q_{ij}})\}$, when *i* is uniformly distributed between the values of the attributes domain and the query range *j* follows a distribution. In addition, it also satisfies Property 6.1 in the special case when the query workload consists of prefix-range queries, i.e., $W = \{(q_{0j}, f_{q_{0j}})\}$.

6.4. Experimental Evaluation

In this section, we run a set of experiments to evaluate the performance of the histogram distance measures. In particular, in Section 6.4.1 we investigate experimentally if the histogram distance measures satisfy the workload-aware property for several kinds of query workload while in Section 6.4.2 we conduct the same experiment with that in Section 3.6 but this time we use histograms for summarizing the peer content and the global query workload *W*. Furthermore, in Sections 6.4.3 and 6.4.4, we study how the global query workload can be estimated and propose to cluster the peers using their local query workloads, respectively. The experimental parameters for the peer data distribution, the query workload distribution and the network size are the same with those used for experiments of Chapter 3 in the case we have equal peer sizes, while the histogram parameters are the ones selected in Chapters 4 and 5. These parameters are summarized in Table 6.1.

6.4.1. Histogram Similarity

In this set of experiments, we evaluate the three histogram distance metrics from the perspective of satisfying the workload-aware property for several query workloads. Consider that we have a-priori knowledge of the global query workload *W*. Initially, we create the histogram *H(n)* of each peer *n*, $0 \le n < 100$, that summarizes its content and the corresponding to *W* histogram *H(W)*. We select a peer $n_i \in N$ and we compute the distance of each histogram *H(n)*, $0 \le n < 100$ and $n \ne n_i$, with $H(n_i)$ using the three distance metrics. Our performance measure is the difference between the number of results provided by each *n* and the number of results provided by n_i for the workload *W*, that is:

$\sum_{q \in W} p_q \left| results(n, q) - results(n_i, q) \right|$ with respect to their distance *hd(H(n), H(n_i), H(W))*. The desired behavior (as expressed by Property 6.1) is for the difference in the number of results provided by a pair of peers for *W* to be analogous to their distance, i.e., the smaller the distance between the two histograms, the smaller the difference in their result size.

Table 6.1: Input Parameters for the P2P Network Using Histograms.

Parameter	Default Value	Range
Peer-to-Peer Parameters		
Number of peers ($\|N\|$)	100	
Percentage (%) of peers visited during routing		5 - 80
Data Distribution Parameters		
Domain of *x*	[0, 999]	
Tuples per node	10000	
Data Concentration (*DC*)	0.8	
Number of disjoint regions (*Dr*)	200	
Query Workload Parameters		
Query Workload Distribution	*Zipf*	
Number of queries	10000	
Range of queries	[0, 999]	
Zipf parameter (*Qz*)	3.0	
Hot query range (*Hqr*)		10 - 800
Histogram-Related Parameters		
Peer Histogram Parameters (H(n))		
Type of Histogram:	*Maxdiff(v, f)*	
Number of buckets (*b*)	100	
Query Workload Histogram Parameters		
Type of Histogram:	*W-Self-Tuning*	
Number of buckets (*b*)	100	
Merge threshold (*mt*)	0.01%	
Split threshold (*st*)	10%	
Restructuring Interval (*ri*)	100	

6.4.1.1. Histogram Similarity Taking into Account the Query Workload

We investigate if the histogram similarity metrics satisfy the workload-aware property for several range query workloads. We consider query workloads that consist of range queries and their ranges follow the Zipf distribution with skew equal to 3.0 (*Qz* = 3.0) and the most popular query range varying from 10 (covering 10 values) to 800 (covering 800 values). We selected the skew parameter to have the value 3.0, i.e., high skewnesss in query range distribution. In Figures 6.3, 6.4 and 6.5 we demonstrate the behavior of the L_1, edit and wedit histogram distances respectively.

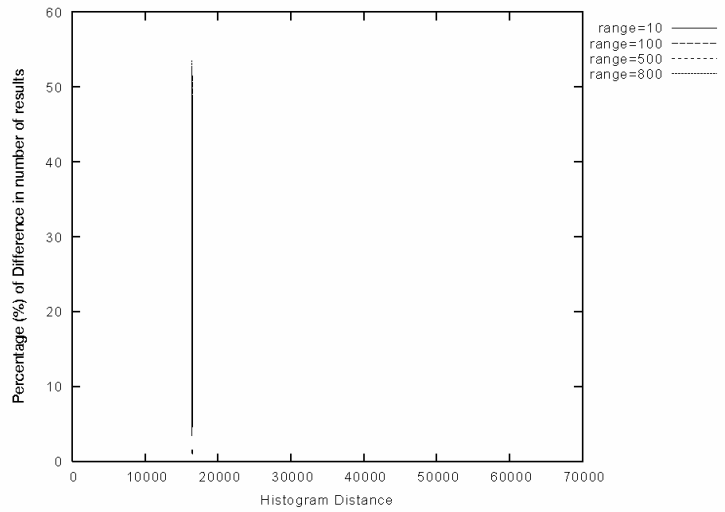


Figure 6.3: Clustering Quality for L_1 Distance when Varying the Popular Query Range Value of the Workload and the Range Skew is Set to 3.0.

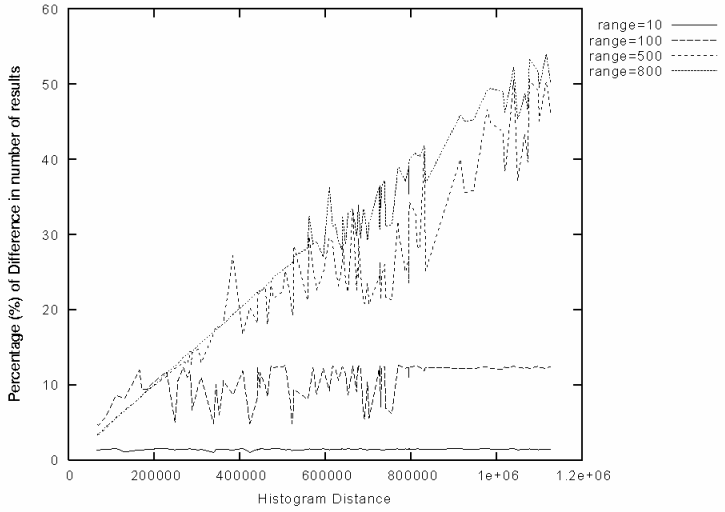


Figure 6.4: Clustering Quality for *Edit* Distance when Varying the Popular Query Range Value of the Workload and the Range Skew is Set to 3.0.

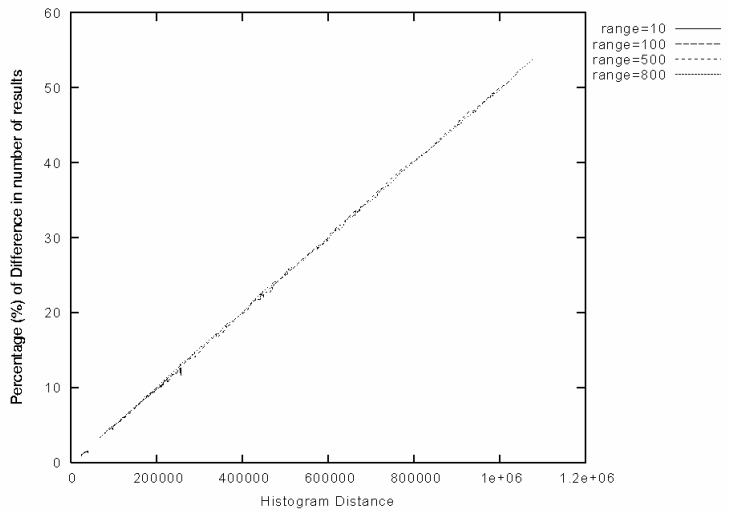


Figure 6.5: Clustering Quality for *Wedit* Distance when Varying the Popular Query Range Value of the Workload and the Range Skew is Set to 3.0.

The conclusions confirm the statements that we have made in the experimental evaluation of Chapter 3. In particular, because of the nature of the peer data distribution the L_1 distance assumes that all histograms *H(n)* have nearly the same distance with $H(n_i)$, since L_1 is not shuffling dependent and considers only individual intervals. However, the absolute difference of results for *W* between each peer and peer n_i, $n_i \neq n$, are different. The edit distance performs better than L_1, especially for query workloads that almost consist of queries with large ranges, since it takes into account the order of intervals; it is shuffling dependent. In particular, the edit distance between two histograms, $H(n_1)$ and $H(n_2)$, takes into account the ordering of all common intervals, while a query with range k involves only (k+1) values. Hence, the query does not depend on the difference that the two peers, n_1 and n_2, may have in the rest of their values. This is evident in Figure 6.2 where for workloads with low range, e.g. 10, the edit distance does not perform well. As the popular query range value increases, e.g., 800, the performance of the edit distance improves. Finally, as we expected, the wedit distance performs well for all kind of query workloads, i.e., the difference in the results increases analogously to the histogram distances.

6.4.2. Clustering of Peers Using Global Query Workload

In this section, we conduct the same experiment as described in Section 3.6 but using histograms for summarizing the peer content and the global query workload *W*. In particular, we initially create for each peer *n* its associated histogram, *H(n)*, that summarizes its content and for a given workload *W*, i.e., we know from the beginning the necessary information that

characterize the workload, its associated histogram *H(W)*. Then, for each one of the three histogram distance metrics we construct the p2p network. Finally, we pose each query *q* of *W* to the network and we measure the *PeerRecall* that the network achieves when each $q \in W$ visits a specified number of peers.

In Figure 6.6, we depict the performance of the three histogram distance metrics in comparison with the corresponding distance metrics, defined in Chapter 3, when we have the whole information about the peer content and the query workload. For *W*, we set the parameter *Qz* equal to 3.0 while varying the value of the most popular query range (*Hqr*). We define by *H-L1* and *H-edit* the L_1 and edit histogram distances respectively so as to differentiate them from the L_1 (Eq. 3.5) and edit (Eq. 3.7) distances defined when we do not use histograms. As we can see, all the corresponding distances, e.g., *L1* and *H-L1*, achieve the same *PeerRecall*, i.e., the clustering of the peers is identical. This means, on the one hand that the selected histograms summarize efficiently the peer content and the global query workload respectively and on the other hand the histogram distance metrics approximates well the corresponding distance metrics, which use the whole information about peers content and the query workload, since we do not loose enough information capable to lead in *PeerRecall*'s reduction. Furthermore, as we expected the wedit distance performs better than the H-L_1 and H-edit distances.

6.4.3. Global Query Workload Estimation

In the previous section, we have shown that the wedit distance performs efficiently when we have a-priory knowledge about the global query workload. The assumption of having a-priory estimation of the global query workload is very strong since in p2p systems the query workload distribution may change during time. The results of this experiment show that having somehow an estimation of the global query workload and taking it into account to calculate the similarity between pairs of peers, leads us in efficient clustering.

Thus, an important issue is the estimation of the global query workload when we do not have a-priori knowledge of its distribution. Recall that, each peer *n* keeps statistics, a histogram *HW(n)*, that summarizes its local query workload, i.e., the queries that arrive to it. Hence, we propose to acquire the global query workload synopsis, *H(W)*, by merging the local query workload histograms of the peers. The merging process that we follow is the same as the one described in Section 4.2.1, hence the merged global query workload synopsis, denoted as *MH(W)*, that arises from this process has also *b* buckets. To evaluate the clustering of peers

using their local indexes and the global query workload synopsis, which arises from merging all the local query workload synopses, we conduct an additional experiment using the wedit distance.

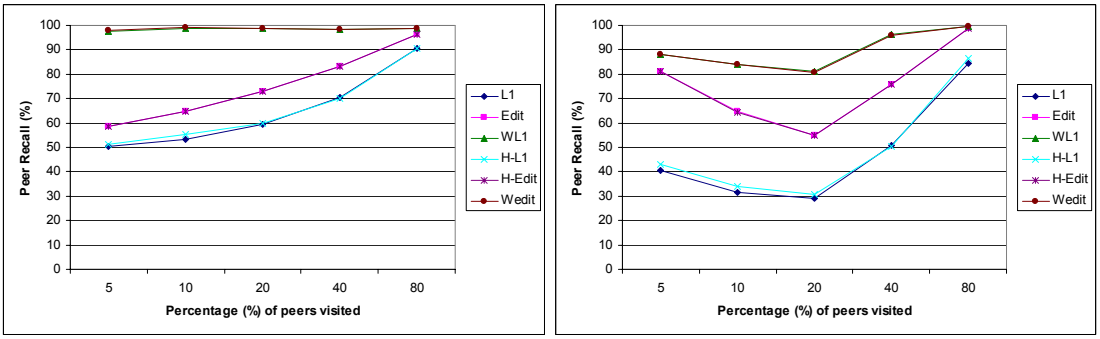


Figure 6.6: Comparison of Histogram-Based Distance Metrics with the Corresponding Distances, which Use the Whole Information about Peer Content and Query Workload, when Varying the Number of Peers Visited with Qz = 3.0 for (a) Hqr = 10 and (b) Hqr = 100.

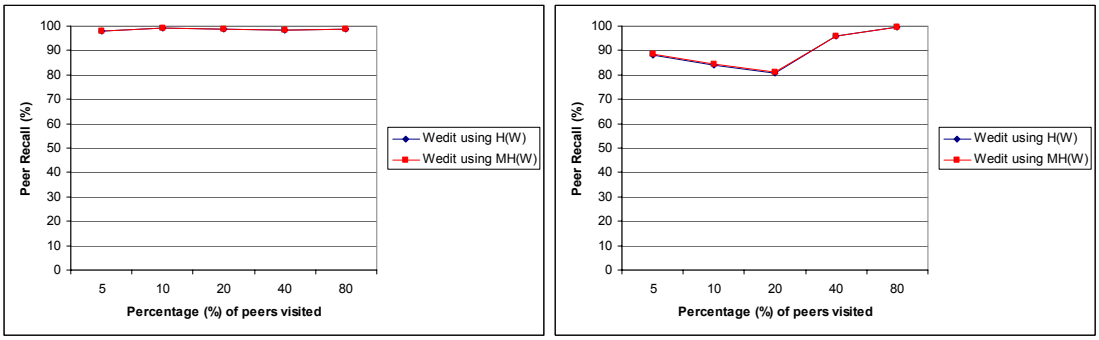
In particular, the experiment is conducted as follows: Assume that we have a set N of peers and a given global query workload W. For each peer $n \in N$, we maintain two structures. The first one concerns the summarization of its content, i.e., its local index $H(n)$, and the other, $HW(n)$, summarizes its local query workload $LW(n)$. Initially, we create the p2p network using the wedit distance based on $H(n)$ and $H(W)$. Then, we pose each query q of W to the constructed network and update the local query workload synopses of all the peers that q visits. After the construction of each $HW(n)$, we re-cluster the peers based on the wedit distance but using the merged global query workload synopsis, $MH(W)$, instead of $H(W)$. Finally, we pose again each $q \in W$ to the reconstructed network and measure the corresponding *PeerRecall* to see how efficient is the clustering of peers using their local query workloads synopses.

In Figure 6.7, we show the *PeerRecall* achieved for several query workloads when for the formation of clusters we use the wedit distance when we use $H(W)$ and when we use the merged global query workload synopsis, $MH(W)$. As we can see, acquiring the global query workload synopsis by merging the local query workload synopsis is efficient since the *PeerRecall* we achieve in this case is similar with the one we get when we have a-priory knowledge of $H(W)$. This was expected since in Section 4.3.2 we showed that the merge

procedure for merging two histograms is very efficient. Hence, the *H(W)* and *MH(W)* histograms are very similar.

6.4.4. Clustering of Peers Using Local Query Workload

The question that arises is if it would be efficient to create a clustering of peers based on their local query workload synopses instead of using the global query workload synopsis. To answer this question, we conduct an additional experiment where we create clusters of peers using the *wedit* distance based on local query workloads. The experiment is carried out in a similar way as before but we re-cluster the p2p network based on the local indexes and the local query workload synopses. In particular, for each pair of peers, n_1 and n_2, we find their distance, based on *wedit*, using $H(n_1)$, $H(n_2)$ and one of the local query workload histograms of these two peers, $HW(n_1)$ or $HW(n_2)$, selected randomly. Finally, we pose again each $q \in W$ to the reconstructed network and measure the corresponding *PeerRecall* so as to see how efficient is the clustering of peers using their local query workloads synopsis.

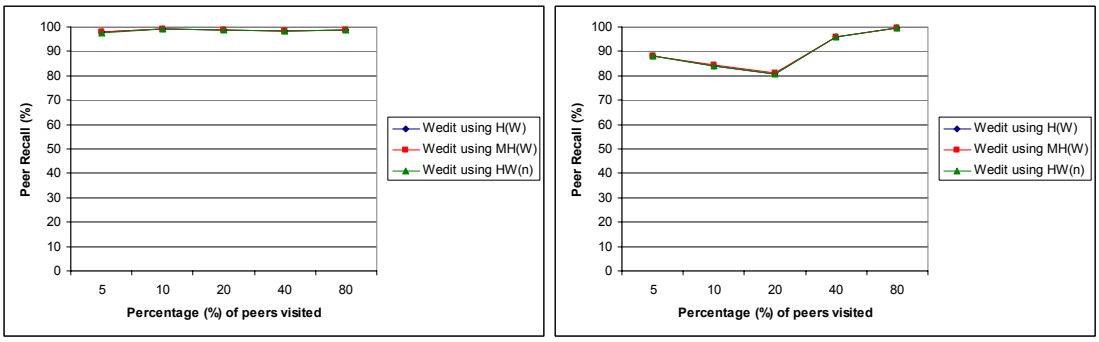


(a) (b)

Figure 6.7: Comparing the *Wedit* Distance when it Uses the A-priori Global Query Workload Synopsis, *H(W)*, and the Merged Global Query Workload Synopsis, *MH(W)* while Varying the Number of Peers Visited with $Qz = 3.0$ for (a) $Hqr = 10$ and (b) $Hqr = 100$.

In Figure 6.8, we depict the *PeerRecall* for a given workload *W* when the clustering of peers is performed based on local query workload synopsis, *HW(n)*, and when the clustering of peers is done based on the merged global query workload synopsis, *MH(W)*. We compare these two methods with the case when we have a-priory knowledge of *W*. As we can see, the *PeerRecall* we achieve when we cluster the peers using the wedit distance based on local query workload synopses is similar with that when we use the global query workload synopsis. This happens because peers that have similar results for a given query workload

also have similar distribution of their local query workloads, i.e., the same queries arrive to them. Therefore, this experiment shows that if the appropriate queries for a peer arrive to it, i.e., a query visits the peers with the most matching results, the clustering of peers can be done efficiently taking into account their local query workloads.



(a) (b)

Figure 6.8: Comparing the *Wedit* Distance when we Use the Global Query Workload Synopsis, *H(W)*, the Local Query Workload Synopsis, *HW(n)*, and the Global Query Workload Resulted by Merging Peers Local Query Workload Synopses, *MH(W)*, respectively while Varying the Number of Peers Visited with $Qz = 3.0$ for (a) $Hqr = 10$ and (b) $Hqr = 100$.

6.5. Summary

To conclude, in this chapter we make an extensive discussion on histogram-based distance measures that are used for the formation of peer clusters. Initially, we introduce a histogram-based workload-aware property that the histogram distance measures must follow so as to create an efficient clustering. Then, we introduce a workload-aware histogram distance measure, the *wedit* distance, and we compare it with two well-known histogram distance measures, the L_1 and the *edit* distance. The experiments show that the wedit histogram distance satisfies the workload-aware property while the L_1 and the edit histogram distances do not.

In addition, we conduct several experiments where the clustering of peers is done based on the global query workload, which obtained by merging the peers' local query workloads, and when the clustering of peers was done based on the local query workloads. In both cases the clustering of peers is efficient. Summarizing, there are many ways to create workload-aware clustering of peers. At one extreme, the global query workload is used while at the other extreme only the workload at a single peer is used. In the first case, an estimation of the global query workload can be obtained by merging the local query workloads of the peers.

Hence, some form of gossiping or epidemic propagation [11] has to be used, i.e., each peer in the network has to send its local query workload to the other peers of the network, so as to eliminate the communication overhead in the network. This issue is beyond the scope of this thesis.

In the next chapter, we describe how the protocols for join, leave and lookup behave when the clustering of peers is done taking into account the global query workload and the local query workload.

CHAPTER 7. BUILDING WORKLOAD-AWARE OVERLAYS USING HISTOGRAMS

In this chapter, we describe how histograms are used to construct a workload-aware p2p network and how a query is routed through this network. Recall that each peer *n* maintains the following data structures: its local index *LI(n)* that summarizes the data values stored locally, one routing index *RI(n, e)* for each of its links *e*, which summarizes the content of all peers that are reachable from *n* using link *e* at a distance at most *r*, called *radius*, and its local query workload synopsis *LW(n)* that summarizes the set of queries arrive to *n*. In addition, for the connection between two peers we make a distinction between two types of links: *short-range* links (or short links) and *long-range* links (or long links). Short-range and long-range links are used to connect peers that are similar and non-similar, respectively. The degree of similarity between two peers depends on the similarity measure that we decide to use for the construction of the overlay network. In the following sections, we describe the procedures that we follow for building the p2p network and routing a query through it, based on those

described in [22], when the clustering of peers is done based on the global query workload and when the clustering of peers is based on local query workloads.

7.1. Building Workload-Aware Overlays Based on Global Query Workload

In this section, we describe the protocols that we use when a new peer wishes to join the p2p network, for routing a query on it and when a peer decides to leave the system in the case when we use the global query workload for the formation of clusters, i.e., each peer has knowledge of the global query workload.

7.1.1. Workload-Aware Overlay Construction

In this section, we describe how the overlay network can be constructed based on the routing indexes. The main issue when a new peer *n* enters the system is to find and link *n* with a set of peers similar to *n*. In general, the idea is to set the local index of peer *n* as a join message which is routed, via routing indexes, towards the peers that are most similar with *n*. Assume that the join message at some point has visited a group *J* of peers. The peers in *J* are sorted based on their similarity with *n*. Peer *n* is linked with the *SL* most similar to it peers and with probability P_l to one of the remaining peers in *J*. Both *SL* and P_l are tuning parameters whose values affect the quality of clusters formed. The motivation for distinguishing the links into two types is that short links are inserted so as to connect peer *n* with the most similar peers that the join message has visited. Hence, relevant peers are located nearby in the network. In contrast, long links are inserted so as when the join message is not in a "relevant" group of peers, i.e., the join message of peer *n* is located in a peer that is not similar with *n*, to be easy to navigate the message to another group of relevant peers that might be more relevant with *n*. In addition, long links are also useful for the routing of a query as we will discuss in the next section.

In particular, when a new peer *n* wishes to join the system, initially it poses its local index *LI(n)* as a join message to a well known peer of the system. In addition, this message also keeps a list *L* (initially empty) with all the peers that it will visit during routing. Each peer *p* that receives the join message performs the following steps:

1. The distance *d(WS, LI(p), LI(n))* between the local indexes *LI(p)* and *LI(n)* of peers *p* and *n* respectively is calculated taking into account the global query workload synopsis *WS*.
2. Node *p* and the corresponding distance are added to list *L*.

3. For each one of the links *e* of peer *p*, the distance *d(WS, RI(p, e), LI(n))* between the local index *LI(n)* and the routing index *RI(p, e)* is calculated taking into account the global query workload synopsis *WS*.
4. The join message is propagated through the link *l* whose routing *RI(p, l)* is the most similar with *LI(n)* (*d(WS, RI(p, l), LI(n)) < d(WS, RI(p, e), LI(n))* $\forall$ link $e \neq l$) and has not been followed yet. When the join message reaches a peer with no more unvisited links to follow, backtracking is used.

The routing of the join message stops when *JMaxVisited* peers are visited. Then, peer *n* creates short links with the *SL* peers in the list *L* of visited peers whose distances are the smallest ones. Also, with probability P_l, peer *n* creates a long link with another peer from the list different from the *SL* peers previously selected.

Note that if we use a content-based similarity measure that does not take into account the query workload, in Steps 1 and 3 of the join procedure, the distance between the local indexes *LI(p)* and *LI(n)* of peers *p* and *n* takes the form *d(LI(p), LI(n))* and the distance between the local index *LI(n)* and the routing index *RI(p, e)* takes the form *d(RI(p, e), LI(n))*, respectively.

An important issue is how to select the peer that will be attached through long link with peer *n*. One simple approach is to select the peer from the list *L* that has the largest distance and does not belong to the *SL* peers selected to be linked through short links. Another approach is to select randomly one of the rest of the peers within the list *L*, which also does not belong to the *SL* peers selected to be linked through short links. In this thesis we follow the second approach.

Finally, as we have mentioned before the routing index of peer *p* for its link *e*, *RI(p, e),* summarizes the content of all peers that are reachable from *p* using link *e* within radius *r*. Hence, to calculate in Step 3 the distance *d(WS, RI(p, e), LI(n))*, we normalize the frequency of each bucket of the *RI(p, e)* by dividing it with the number of the peers that are included in the routing index. By using normalization, the *RI(p, e)* portrays the average frequency distribution of the peers, which are within the radius *r* and are reachable from *p* using link *e*.

7.1.2. Query Routing

A query *q* may be posed at any peer *n*. Our goal is to route the query *q* through a set of peers that gives a large number of results for *q*, that is, we want to maximize *PeerRecall*. The

routing of a query stops when a predefined number of peers is visited, denoted as *QMaxVisited*. The routing procedure we propose relies on a depth-first traversal of the network. In particular, each peer *p* that receives a query *q* executes the following steps:

1. For each one of its links *e*, peer *p* estimates the number of results that we will be found for *q*, by using the *RI(p, e)*, i.e., *hresults(RI(p, e), q)*, if link *e* is followed.
2. The query is propagated through the link *l* whose routing *RI(p, l)* gives the most matches (*hresults(RI(p, l), q)* $\geq$ *hresults(RI(p, e), q)* $\forall$ link $e \neq l$ and *hresults(RI(p, l), q)*$\neq$*0*) and has not been followed yet. When the query reaches a peer with no more unvisited links to follow, backtracking is used.

By following this link, the query is propagated towards the peers that are estimated to provide the most results and thus *PeerRecall* is increased. An occasion that can be occurred is when the query *q* visits peer *p* and there are no matching peers within the *horizon* of this peer, i.e., *hresults(RI(p, e), q)=0* $\forall$ link *e* of *p*. Recall that, *horizon* is the set of peers within distance *r* of *n*. Hence, the matching peers are outside the radius *r* of *p*. To handle this situation, we follow the long-range link of peer *p* (even if it does not match the query). The idea is that we want to move to another region of the network, since the current region (bounded by the *horizon*) has no matching peers. In the case that peer *p* has no long-range link or we have already followed its long-range link, the query is propagated through a short link to a direct neighbor peer and so on until a long-range link is found.

The main drawback of the above procedure is that the query is propagated to only one peer each time. Hence, this procedure is not so efficient in terms of response time; it is desirable to retrieve the query matching results as quick as possible. A variation of the above procedure can be used for faster retrieval of query results. In particular, when the query *q* is initiated at a peer *n*, we follow steps 1 and 2 of the above procedure to route the query to the appropriate region of the network that is rich with matching results. Consider that peer *p* belongs to the appropriate group of peers and is visited by *q*. Then, to exploit the grouping of similar peers, which are nearby in the network, we can use flooding to propagate the query to all the neighbor peers that are connected with short-links.

The main issue in this procedure is to determine whether the query has reached the appropriate region in the network, i.e., a peer with a large number of matching results. A simple approach is the query message to keep information about the number of matching results that each visited peer provides. If the query visits a peer that provides a sufficient larger number of results compared to those provided by the already visited peers, then we

assume that we are in the right group of peers. However, this approach of detecting the appropriate cluster of peers for the query does not guarantee that we are in the right cluster of peers. In addition, when using this variation of query routing, after the flooding starts, the propagation of the query is limited within one group of peers only. This might be a drawback since it is possible that a range query can be answered by peers of more than one group depending on the similarity measure we use when a peer joins the system. In particular, if we use a content-based similarity measure, each peer that joins the system links to the most similar of the existing peers according to their contents. Furthermore, a range query can be answered efficiently by more than one group of peers. Hence, to retrieve a large number of results, the query must visit all the appropriate groups and not only one of them. Thus, using content-based similarity measures for clustering the peers does not ensure that many results will be found. In contrast, if we use a workload-aware similarity measure, all the peers that are nearby in the network answer similar the same set of queries; thus, they are in the same group. Hence, by limiting the query propagation into only one group of peers we expect the performance of *PeerRecall* not to be influenced negatively. The variation of the routing procedure needs further investigation and is left as future work.

Another important issue is the kind of routing indexes [9] that we use for routing the query. In particular, the routing indexes that we proposed so far summarize the content of all the peers that are reachable from a peer n using its link e at a distance at most r. The main limitation of this kind of routing indexes is that they do not take into account the number of hops required to find the query results. For instance, consider that we want to create *RI(n, e)* and we have two peers, p and g, which must be included in the routing index. The first one is a direct neighbor of n while the second is many hops away from n. In this case the local indexes of both peers, *LI(p)* and *LI(g)*, will participate "equally" in the creation of the routing index of peer n, even though the probability of the query visiting peer g is much lower when compared to that of visiting peer p. An alternative approach is to keep separate routing indexes for the procedures of join and routing. For the routing of a query, we propose the routing indexes to take into account the number of hops: *hop-count* routing indexes. In particular, if we want to create *RI(n, e)* and peer p is reachable from peer n through link e at $m<r$ hops, then the *LI(p)* will be divided by a factor m and then it is going to be included in the *RI(n, e)*. Hence, the furthest peer p is located from n, the less the *LI(p)* contributes in the creation of *RI(n, e)*. The hop-count routing indexes are left as future work.

7.1.3. Peer Leave

When a peer *n* decides to leave the system, it must decide about the new connections that will be established among peers so as on the one hand to keep the network connected and on the other hand the peer departure to not affect the grouping of peers. To achieve this, we propose to link all the neighbors of peer *n* in a path.

In particular, when a peer *n* wishes to leave the system it first asks its neighbors about their local indexes and their local query workload synopsis. Hence, for each of its neighbors, *p*, peer *n* stores the local index of *p*, the local query workload synopsis of *p* and the type of link, short or long, which is connected with *p*. In addition, peer *n* keeps in a list *NL* the identifiers of its own neighbors. Before peer *n* leaves the system, does the following steps in order to decide about the new connections that will be created after its departure.

While *NL* is not empty

1. Peer *n* selects a neighbor *p* from *NL*. $NL = NL - \{p\}$.
2. If the link that connects peers *n* and *p* is short, *link(n, p) = short*, then we select another peer $m \in NL$, which is also connected through short link with *n*, *link(n, m) = short*, to link through short link with *p*. In addition, peer *m* must be the one with the smallest histogram distance from peer *p*, i.e., $d(WS, LI(p), LI(m)) < d(WS, LI(p), LI(k)) \ \forall$ peer $k \neq m$ and $k \in NL$, if we use a workload-aware distance metric or $d(LI(p), LI(m)) < d(LI(p), LI(m)) \ \forall$ peer $k \neq m$ and $k \in NL$, if we use a content-based distance metric.
3. If the link that connects peers *n* and *p* is long, *link(n, p) = long*, then similarly with step 2 we select another peer $m \in NL$, which is also connected through short link with *n*, to link with *p* through a long link.

End While

By following the above procedure we ensure that the network will remain connected after the departure of peer *n* since there is a path between all the neighbors of *n*. Furthermore, the grouping of the peers remains unaffected since each neighbor *p* that connects with short link with *n* will be linked with the most similar of *n*'s neighbor, *m*, which is also linked with *n* through a short link. As we have mentioned in section 7.1, when a peer joins the system it connects with the most similar existing peers. Hence, we expect peers *n*, *p*, *m* to be similar to each other. After determining about the new connections, peer *n* sends a message to each of its neighbors, *p*, which contains the new connections that *p* is going to create. Then, peer *n*

leaves the system. Note that the creation of new links for *n*'s neighbors changes the routing indexes of peers that are within the horizon of *n*. Thus, the routing indexes of those peers must be updated.

7.1.4. Creating and Maintaining Routing Indexes

Let us now turn our attention on how RIs are either created or maintained [22]. The creation of a routing index is necessary when a peer joins the system while the routing index update is essential for existing peers in cases when a peer joins the system, when the data changes locally at a peer; hence its local index is updated, and when a peer leaves the system.

In particular, when a new peer n_i joins the system must construct its own routing indexes and must inform the peers within radius *r* about its stored data, in order to update their routing indexes and keep them in a consistent state. Thus, after peer n_i joins the system and links with its selected as neighbors peers, then it sends a message *New(LI(n_i), Counter)* to all the peers that are within its horizon, in order to propagate them its local index. Initially, the message *New(LI(n_i), Counter)* is propagated to all n_i's neighbors and the *Counter* is set to *r*. Each peer n_j that receives the *New* message, from its link *e*, merges the *LI(n_i)* with its routing index of the corresponding link, *RI(n_j, e)*, it reduces the *Counter* by one and then it sends the message *New(LI(n_i), Counter-1)* to all of its neighbors.

In addition, peer n_i must construct its own routing indexes. Hence, it must receive the local indexes of all the peers that are within its radius *r* for each of its links and create the corresponding routing indexes as we described in Section 4.2.1. This is achieved through a sequence of *FW(Local Index, Counter, Flag)* messages. In particular, each peer n_j that receives the *New* message from a peer n_k, it replies to n_k with a *FW(LI(n_j), r, False)* message. Upon receipt the *FW(LI(n_j), Counter, False)* message, each peer decrements the *Counter* by one and if *Counter* is non zero then it forwards the *FW(LI(n_j), Counter-1, False)* message back to the peer that sent the *New* message to it. Thus, all the peers that receive the *New* message, i.e., all the peers that belong in the horizon of n_i, send their local indexes through the *FW* messages to n_i and its routing indexes are constructed by merging the corresponding local indexes.

The usage of the *Flag* parameter is important in the case when the insertion of the new peer changes the horizons of the existing peers. For example, consider the network depicted in Figure 7.1 where the radius of the routing indexes is set to $r = 2$. Assume that peer 10 enters

the system and links with peers 1 and 4. Hence, the local index of peer 10 must be propagated to peers 1, 2, 3, 4, 5. Furthermore, the routing indexes of the new peer must be constructed. However, note that the insertion of peer 10 changes the relative distance of some peers. In particular, now peer 4 belongs to the horizon of peer 1 and vice versa, since their distance through peer 10 is now 2. Thus, the local index of peer 4 (1) must now be merged with the corresponding routing index of peer 1 (4). To deal with this situation, we use the *Flag* parameter.

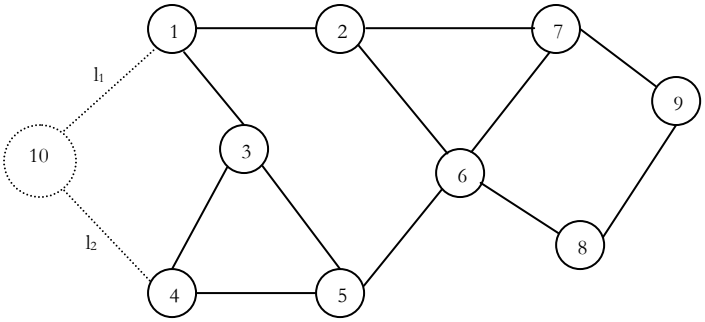


Figure 7.1: Example of Index Update

Specifically, the *Flag* parameter is used as follows: Initially, *Flag* is set to *False* for each new *FW(Local Index, Counter, Flag)* message. When the new peer n_i receives a *FW(Local Index, Counter, Flag)* message, it changes the *Flag* parameter from *False* to *True* and decrements the *Counter* by one. If the *Counter* is non-zero, it propagates the *FW(Local Index, Counter-1, True)* message to all of its neighbors except from the peer that has received the message. Each peer that receives a *FW(Local Index, Counter, True)* message merges the *Local Index* with its corresponding routing index, decrements the *Counter* by one and if the *Counter* is non-zero it forwards the *FW(Local Index, Counter-1, True)* message to its neighbors, except from the one that has received the message. Hence, in our example peer 10 sends the *New(LI(10),2)* message through the links l_1 and l_2 to all the peers, i.e., 1, 2, 3, 4, 5. Upon receipt of the *New* message, peer 1 sends a *FW(LI(1), 2, False)* message back to 10. When peer 10 receives this message, changes the *Flag* parameter from *False* to *True*, decrements the *Counter* by one and propagates the *FW(LI(1), 1, True)* message to peer 4. When peer 4 receives this message, merges the *LI(1)* with its *RI(4, l_2)*; hence the horizon of peer 4 and the routing index of link l_2 are maintained up-to-date.

In the case when the data change locally at a peer n_i, its local index must be updated. In addition, the routing indexes of all the peers within n_i's horizon must be updated too. Hence, each peer, in order to detect the changes in its data, periodically reconstructs its local index. Furthermore, a peer n_i that updates its local index informs the peers within its horizon to update their routing indexes through a sequence of *New_Up(LI(n_i), LI'(n_i), Counter)* messages. The *New_Up* messages are similar with the *New* messages and the only difference is that they contain additional information about the old "version" of the n_i's local index, *LI(n_i)*, and the more recent version of n_i's local index, *LI'(n_i)*. When a peer n_j receives the *New_Up* message, then it excludes from the corresponding routing index the old "version" of n_i's local index, *LI(n_i)*, and updates the routing index by merging to it the *LI'(n_i)*.

In addition, when a peer n_i wishes to leave the system, it must inform all the peers within its horizon to exclude the information of the *LI(n_i)* from their routing indexes. To achieve this, peer n_i sends a *New* message to all its neighbors with a *Counter* set to the radius *r*, *New(LI(n_i), r)*. When the message reaches a peer n_j through link *l*, the peer updates its routing index, reduces the *Counter* by one and then sends the message *New(LI(n_i), Counter-1)* further to all its neighbors until the *Counter* reaches 0. Furthermore, peer n_j sends a *New(LI(n_j), r)* message through link *l* to inform the peers that are now included in its horizon about its local index, since the departure of peer n_i has resulted in the decrease of the distance between other peers.

In the following sections, we briefly discuss how the global workload synopsis can be kept up-to-date and the conditions that we must follow for efficiently maintenance of peer clustering.

7.1.5. Updating the Global Query Workload

In this section, we discuss how we can keep the synopsis that approximates the global query workload up-to-date. In particular, the workload distribution in a p2p system may change during time. Hence, the corresponding synopsis, *WS*, must be kept up-to-date so as to represent each time the distribution that the global query workload follows accurately. In addition, as we have mentioned in Section 6.4.3 we create the global query workload synopsis by merging the local peer workload synopses. Although, the issue of acquiring the global query workload synopsis from the local ones is beyond the scope of this thesis, i.e., which form of gossiping or epidemic propagation has to be used, we propose some ideas to accomplish this.

A very simple approach is using a periodic strategy where a peer *n* periodically broadcasts-pushes an update message $W\_Up(LWS_{t_k}(n), LWS_{t_{k+1}}(n))$ to the network. In particular, each peer *n* keeps two snapshots of its local query workload synopsis. The first one, $LWS_{t_k}(n)$, corresponds to the local workload synopsis of peer *n* at time t_k while the second one, $LWS_{t_{k+1}}(n)$, represents the workload synopsis of peer *n* at time t_{k+1}, where (t_{k+1} - t_k) is the broadcast period between two messages. Upon receiving this message, each peer *p* excludes from its *WS* the information of the $LWS_{t_k}(n)$ and merges the *WS* with the $LWS_{t_{k+1}}(n)$. The period of sending the update message might be different for each peer. This approach has two serious drawbacks. The first one is that it incurs a large communication overhead to the network, especially when the time period is too small, because each peer is forced to broadcast the update message even when the distribution of its local query workload does not change; hence it is meaningless to send this message. Secondly, the distribution of the query workload for a peer might change between two consequent messages. This will lead in inaccuracy of the estimation of the global query workload especially when the period between two consecutive update messages is large enough. Overall, it is difficult to determine the time period between two concecutive broadcasts of the update message.

A more sophisticated approach is for a peer *n* to broadcast its local query workload synopsis when it detects that a change to its workload distribution has occurred. In particular, this approach is similar with the previous one with the difference that the peer *n* broadcasts the update message $W\_Up(LWS_{t_k}(n), LWS_{t_l}(n))$ when it detects a change on its local query workload distribution. t_k corresponds to the last time unit that peer *n* sent an update message while t_l is the time unit that the peer detects the change in its local workload distribution. In that way, we eliminate the cost of sending useless messages and furthermore at each point the global query workload is consistent.

7.1.6. Detection of Changes in the Local Query Workload Distribution

In this section, we discuss how we can detect a change in the query workload distribution of a peer. Although this issue is hard to deal with, we propose a simple approach to overcome this problem. Recall that in Section 5.2, we proposed an aging technique in order to enable the W-ST histogram to capture the current distribution of the queries that arrive at a peer. Similarly, to detect the change of the query workload, exactly before each time the restructuring process of the W-ST histogram takes place, e.g., t_l, we compare the two local query workload

synopses $LWS_{t_k}(n)$ and $LWS_{t_l}(n)$ of peer *n*, where t_k corresponds to the last time unit that peer *n* sent an update message. To achieve this, we propose to use a content-based similarity measure, e.g., the edit histogram distance metric, and to measure the ratio of dissimilarity between these two histograms. If the dissimilarity is larger than a threshold then we assume that the query workload distribution has changed.

7.1.7. Re-clustering

When the global query workload distribution changes, we have to re-cluster the peers since their clustering is done based on the previous distribution. By the term re-cluster we mean that a peer leaves and joins the system in the way that we have described in Sections 7.1.3 and 7.1.1, respectively. Obviously, the procedure of re-clustering all the peers will cause a large cost for reconstructing the network especially when the distribution changes frequently. To reduce this cost, we propose a peer to leave and re-join the system only when the distribution changes considerably. In particular, for a peer *n*, we measure the distance between two snapshots of the global query workload synopsis, whenever an update of the global query workload is occurred, *d(WS', WS)* where *WS'* and *WS* is the previous and the current-updated version of the global query workload synopsis. If this distance is larger than a threshold then the peer is re-clustered.

The main drawback of the above procedure is that each peer just examines when a change to the global query workload happens and not whether the current clustering of peers is still or remains appropriate for the new "version" of the global workload distribution; hence the peer does not have to leave and re-join the network. Thus, an extension of the above procedure can be used. In particular, when a peer *n* detects a change to the global query workload distribution, then it sends a message to its neighbors and asks for their local indexes. Upon receiving their local indexes, for each neighbor *p* it measures the distance *d(WS, LI(n), LI(p))* and if this distance is close to zero or lower than a threshold, which means that peer *n* and its neighbor *p* provide similar results even for the new global workload distribution, then peer *n* does not have to leave and re-join the system.

7.2. Building Workload-Aware Overlays Based on Local Query Workloads

As we saw from the experimental evaluation of Section 6, taking into account the local query workload for the formation of clusters can lead to an efficient clustering in the case when a

query visits the peers with the most matching results. This kind of clustering does not have the drawback that each peer must know the global workload distribution. In what follows, we describe the procedures for constructing the p2p network and when a peer decides to leave the system. These procedures are similar with those mentioned in Section 7.1 but with few differences. The protocols for routing a query and keeping the routing indexes up-to-date are exactly the same with those described in Sections 7.1.2 and 7.1.4, respectively.

7.2.1. Workload-Aware Overlay Construction

The procedure that we propose to follow when a peer *n* wishes to join the system is similar with the one mentioned in Section 7.1.1 but with the difference that instead of using the global query workload synopsis, *WS*, for measuring the distance between itself and the candidate peer *p* that is going to be linked, it uses the local query workload synopsis of peer *p*, *LWS(p)*. The idea is that if the two peers provide similar number of results for a given query workload, which in our occasion is the local query workload of peer *p*, then they must be immediately linked.

Thus, as before when a new peer *n* wishes to join the system, initially it poses its local index *LI(n)* as a join message to a well known peer of the system. In addition, this message also keeps a list *L* (initially empty) with all the peers that it visits during routing. Each peer *p* that receives the join message executes the following steps:

1. The distance *d(LWS(p), LI(p), LI(n))* between the local indexes *LI(p)* and *LI(n)* of peers *p* and *n* respectively is calculated taking also into account the local query workload synopsis *LWS(p)* of peer *p*.
2. Node *p* and the corresponding distance are added to list *L*.
3. For each one of the links *e* of peer *p* the distance *d(LWS(p), RI(p, e), LI(n))* between the local index *LI(n)* and the routing index *RI(p, e)* is calculated taking into account the local query workload synopsis *LWS(p)* of peer *p*.
4. The join message is propagated through the link *l* whose routing *RI(p, l)* is the most similar with *LI(n)* (*d(LWS(p), RI(p, l), LI(n))* < *d(LWS(p), RI(p, e), LI(n))* $\forall$ link $e \neq l$) and has not been followed yet. When the join message reaches a peer with no more unvisited links to follow, backtracking is used.

After peer *n* joins to the system, i.e., it is linked with the peers that it chooses as neighbors, initially it has no information about its local query workload since no queries has arrived at it yet. Its local query workload synopsis will be created when several queries will visit it.

An important issue is how to calculate the distance *d(LWS(p), LI(p), LI(n))* between the local indexes of peers *p* and *n* and the distance *d(LWS(p), RI(p, e), LI(n))* between the local index of peer *n* and the routing index of the link *e* of peer *p* in the case when we use a workload-aware distance measure and no queries have arrived to peer *p*, i.e., we have no information about *LWS(p)*. In that case, we overcome this problem by using a content-based similarity measure.

7.2.2. Peer Leave

The procedure that we must follow when a peer *n* leaves the system is similar with the one mentioned in Section 7.1.3 with the only difference that the new connections that will be established between the neighbors of *n*, in order to keep the network connected, is based on the local query workload synopsis of *n*. In particular, before peer *n* leaves the system, it performs the following steps:

While *NL* is not empty

1. Peer *n* selects a neighbor *p* from *NL*. $NL = NL - \{p\}$.
2. If the link that connects peers *n* and *p* is short, *link(n, p) = short*, then we select another peer $m \in NL$, which is also connected through short link with *n*, *link(n, m) = short*, to link through short link with *p*. In addition, peer *m* must be the one with the smallest histogram distance from peer *p*, i.e., *d(LWS(p), LI(p), LI(m)) < d(LWS(p), LI(p), LI(k))* $\forall$ peer $k \neq m$ and $k \in NL$, if we use a workload-aware distance metric or *d(LI(p), LI(m)) < d(LI(p), LI(m))* $\forall$ peer $k \neq m$ and $k \in NL$, if we use a content-based distance metric.
3. If the link that connects peers *n* and *p* is long, *link(n, p) = long*, then similarly with step 2 we select another peer $m \in NL$, which is also connected through short link with *n*, to link through long link with *p*.

End While

Finally, when a peer detects a change in its local query workload distribution then the peer must leave and re-join the system since the clustering of peers might be inefficient. The detection of changes in the local query workload distribution was discussed in Section 7.1.6.

7.3. Summary

Summarizing, in this chapter we describe the protocols that a peer follows when it wishes to join and leave the system as well as how a query is routed through the network in the case when we use the global query workload and when we use the local query workload to create

the clustering of peers. In addition, we described how to create and maintain the routing indexes when a new peer joins the system, when an existing peer wishes to leave the system and when the data changes locally at a peer. Finally, we made a brief discussion on how the global query workload synopsis can be updated, how a peer can detect a change on its local query workload distribution and the conditions that we must follow in order to decide when a peer must be re-clustered.

As we have mentioned before, taking into account the local query workload for the formation of clusters can lead to an efficient clustering in the case when a query visits the peers with the most matching results. This approach does not have the drawback that each peer must know the global workload distribution. However, it does not guarantee that all the queries that arrive at a peer are answered efficiently, i.e., a query might visit several peers that do not provide enough matching results. Thus, the clustering of peers might not be efficient because the local query workload of the peers is not the ideal.

In the following chapter, we conduct an experimental evaluation where we follow the first approach, i.e., the clustering of peers takes into account the global query workload.

CHAPTER 8. EXPERIMENTAL EVALUATION

In this chapter, we experimentally evaluate the performance of the network built based on the global query workload. The network construction and the routing of a query through the network are done by following the procedures that we described in Sections 7.1.1 and 7.1.2, respectively. In particular, we evaluate the network that is constructed using the workload-aware edit histogram distance mesure (*wedit*) and compare it with the networks that are constructed using the content based, the L_1 and the *edit* histogram distance measures. We also compare those p2p networks with a randomly constructed p2p network (*random*), i.e., when both the join of a peer and the routing of a query is done randomly, and with a network that uses the routing indexes only for the routing of a query message (*random_join*), i.e., when only the construction of the network is done randomly. The evaluation of the performance of answering range-queries is done by measuring the *PeerRecall*. In what follows, we initially describe the parameters that we use for the conduction of the experiments. Furthermore, we study the influence of the radius of the routing indexes and the long links in the performance of the network and finally we discuss "cycle" avoidance.

8.1. Experimental Parameters and Distributions

We simulated the peer-to-peer network as a graph and the size, $|N|$, of the network is set to 500 peers. When a new peer wishes to enter the system, the routing of the join message is propagated until *JMaxVisited* peers are visited. In our experiments, we set this parameter equal to $log|N|$ peers so as the routing of the join message increase logarithmically with the

network. Furthermore, each peer creates 1 to 2 short links (*SL* = 1 or 2) and one long link with probability P_l that varies from 0.0 to 1.0. As far as routing indexes are concerned, we set the radius of the horizon from 1 to 4. In addition, when a query is initiated at a peer, we set the number of peers that the query message visits, *QMaxVisited*, equal to 5% of the network size, since we are interested in visiting only a small fraction of the network.

Each peer stores a relation with an integer attribute $x \in [0, 999]$ that contains 10000 tuples. The parameters that we use for the histograms that are used as local indexes are those defined in Chapter 4. In particular, the tuples are summarized by a maxdiff(v, f) histogram with 100 buckets. Note that the histograms that are used as routing indexes also have the same number of buckets. The data distribution that the content of each peer follows is the same with the one described in Section 3.6.1.1 in the case when the peers have equal size but with the difference that more than one peer follows the same distribution. Specifically, for each peer, 80% of the tuples falls into 2 of 200 equal disjoint regions that we have divided the value domain, which are selected randomly, and the rest of the tuples are uniformly distributed among the rest of the values. Each one of the selected regions has 40% of the tuples.

Furthermore, for simplicity, in our experiments we consider that we know the distribution that the global query workload follows. Thus, instead of acquiring the global query workload synopsis from the local ones by merging the local workload synopses we create the histogram-synopsis that represents the distribution that the global workload follows by passing a set of queries. The W-ST histogram that represents the global query workload synopsis also has 100 buckets, while the rest of the parameters, e.g., merge threshold e.t.c., that describe the histogram are those defined in Chapter 5. For these experiments, we assume two different kinds of query workloads consisting of range queries. In both query workloads, we assume that the starting point of the queries is chosen uniformly, while the query ranges follow a Zipf distribution. For the first query workload, we consider that the most frequent queries are those with range 10 (*Hqr* = 10), while in the second query workload the most frequent queries are those with range 100 (*Hqr* = 100). In both query workloads the skewness of the Zipf distribution is set to 3.0 (*Qz* = 3.0). The input parameters are summarized in Table 8.1.

8.2. Performance of Constructed Networks

In this section, we evaluate the clustered networks that are constructed using the *wedit*, the *edit* and the L_1 histogram distance measures. We compare them with a randomly constructed

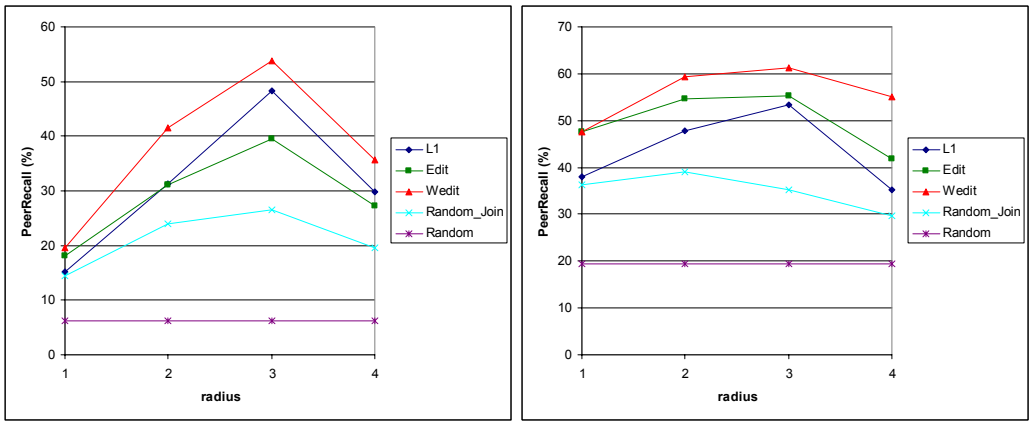
network (*random*), where a new peer that joins the system connects randomly to existing peers and the routing of a query is also done randomly, meaning that when a peer gets a query message it forwards it randomly to one of its neighbors, and with a randomly constructed network that uses the routing indexes only for routing a query (*random_join*). The evaluation of the constructed networks is done using *PeerRecall* as our performance measure.

Table 8.1: Input Parameters for the P2P Network.

Parameter	Default Value	Range
Peer-to-Peer Parameters		
Number of peers ($\|N\|$)	500	
Radius of the horizon (r)	2	1 - 4
Number of short links (SL)	2	1 - 2
Probability of long link (P_l)	1.0	0.0 - 1.0
Percentage (%) of peers visited during join procedure routing (*JMaxVisited*)	$log\|N\|$	
Percentage (%) of peers visited during query routing (*QMaxVisited*)	5	
Data Distribution Parameters		
Domain of x	[0, 999]	
Tuples per peer	10000	
Data Concentration (DC)	0.8	
Number of disjoint regions (Dr)	200	
Query Workload Parameters		
Query Workload Distribution	*Zipf*	
Number of queries	1000	
Range of queries	[0, 999]	
Zipf parameter (Qz)	3.0	
Hot query range (Hqr)	10, 100	
Histogram-Related Parameters		
Peer Histogram Parameters (H(n))		
Type of Histogram	*Maxdiff(v, f)*	
Number of buckets (b)	100	
Query Workload Histogram Parameters		
Type of Histogram	*W-Self-Tuning*	
Number of buckets (b)	100	
Merge threshold (mt)	0.01%	
Split threshold (st)	10%	
Restructuring Interval (ri)	100	

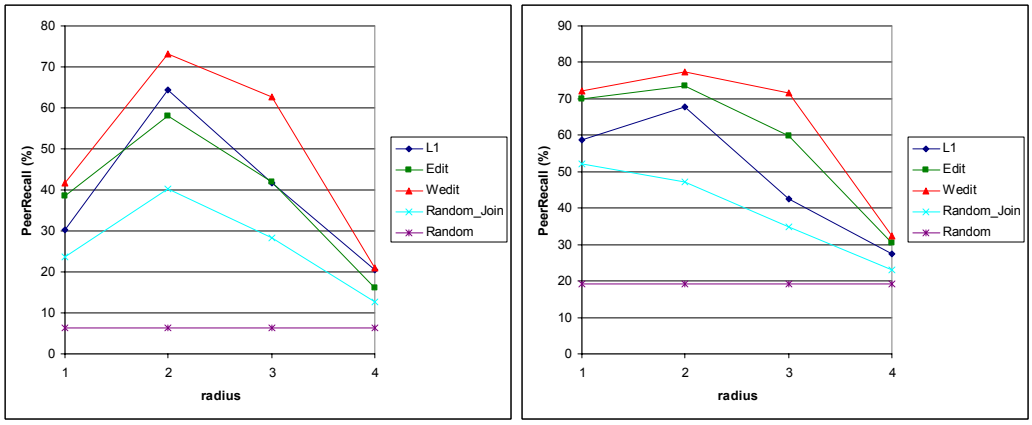
In Figures 8.1 and 8.2, we demonstrate the influence of the radius of the routing indexes in the performance of the constructed networks when the number of short links that are used for connecting a new peer that joins the system with the existing peers is SL = 1 and SL = 2, respectively and when the query workload mainly consists of queries with ranges 10 (Fig.

8.1(a) and 8.2(a)) and 100 (Fig. 8.1(b) and 8.2(b)). The radius of the horizon varies from 1 to 4 while the probability of a long link is fixed to 1.0.



(a) (b)

Figure 8.1: Performance of Constructed Networks for different Values of the Radius when (a) *Hqr* = 10 and (b) *Hqr* = 100 and when *Qz* = 3.0 and *SL* = 1.



(a) (b)

Figure 8.2: Performance of Constructed Networks for different Values of the Radius when (a) *Hqr* = 10 and (b) *Hqr* = 100 and when *Qz* = 3.0 and *SL* = 2.

Our first observation is that using routing indexes for both the construction of the p2p network, i.e., when a new peer joins the system we try to link it with the most similar peers as determined by the similarity distance measure (L_1, edit or wedit) used, and for the routing of a query results in much better performance in comparison with the networks that do not use routing indexes at all (random) or use the routing indexes only for the routing of a query (random_join). Thus, the clustering of similar peers using a distance measure plays a crucial

role in the performance of the network. In addition, in the case of the clustered constructed networks when we use 2 short links instead of 1 for connecting a new peer results in better performance, since when a new peer enters the system, it has the potential to be linked with more similar peers.

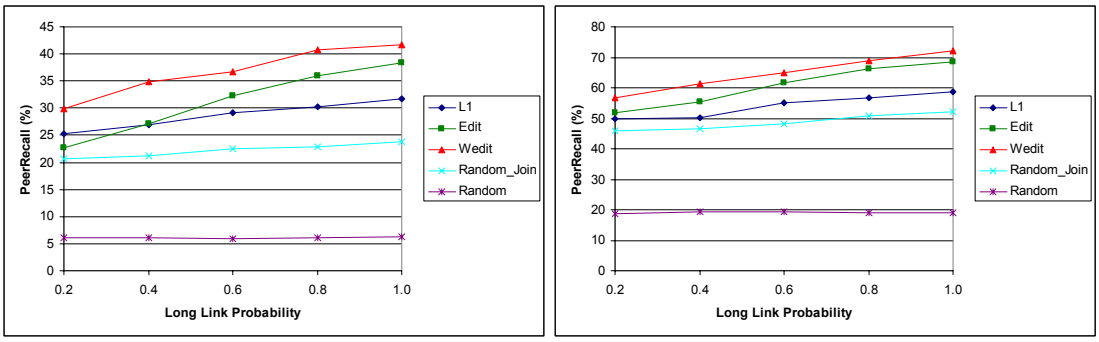
Comparing now the constructed clustered networks we see that creating a p2p network using our wedit distance measure leads to better performance in comparison with the networks that are constructed using the content-based, which are the L_1 and the edit distance measures for all kinds of query workloads. In particular, for query workloads that consist of queries with small range, *Hqr* = 10, the network constructed based on the edit distance measure performs worse than the one constructed based on the L_1 distance, in the case when the radius is larger than 1, and even worse than the one created using the wedit distance (Fig.8.1(a) and 8.2(a)). This happens because as we have discussed in Section 6.4.1, the clustering of peers that the edit distance achieves is not efficient in the case when the query workload consists of queries with small ranges.

Although we expected the network constructed by the edit distance to perform better than the one created using the L_1, since the L_1 distance is not shuffling dependent, this does not happen in the case when the radius of the routing indexes is larger than 1. Specifically, as we have discussed, for the data distribution that the peers follow in our experiments the L_1 distance metric clusters the peers that follow the same distribution and it considers the distance between a peer that follows a distribution with all the other peers that follow different distributions equal. Hence, when the radius of the routing indexes is 1, i.e., a peer has information about the local indexes of only its immediate neighbors that likely follow the same distribution, then indeed the edit distance performs better than the L_1. In other words, when the radius is 1 the performance of the query routing depends on how efficient is the similarity that we use for clustering the peers. But as the radius increases, then it is possible the routing index of a peer to include information of the content of peers that do not follow the same distribution with it. Hence, during the routing of a query message the routing indexes, with radius larger than 1, allow the query to navigate efficiently to other clusters of peers that provide enough results for the query.

In the case when the query workload consists of queries with large range (*Hqr* = 100) (Fig.8.1(b) and 8.2(b)) the network that is constructed using the edit distance performs better than the one that is created by using the L_1 distance since the clustering of peers that the edit distance creates is much more efficient for query workloads with large ranges. Also in this

case, the network that is constructed using the wedit distance is much more efficient in comparison with the other two constructed clustered networks.

In addition, for all distance metrics, PeerRecall increases until some point and then it decreases. In particular, in the case when $SL = 2$, PeerRecall decreases for radius greater than 2 while in the case when $SL = 1$ the performance of the constructed networks increases until the radius takes the value 3. In general, this happens because the larger the radius of a routing index, the more peers with different distributions are included within the horizon of a peer; hence, a large number of local indexes are summarized by a routing index of a peer. This leads the routing indexes to become useless since a query message cannot "select" efficiently which link to follow from a peer so as to find enough matching results for the query. Furthermore, for 1 short link, the PeerRecall increases even for radius 3 which does not happen when we use 2 short links. This happens because in this case the number of peers that are wihin the horizon of a routing index is smaller; hence, the inaccuracy of the routing index is not so high. For radius 2 and for 2 short links, we achieve the best performance.



(a) (b)

Figure 8.3: Effect of Long Link in the Performance of the Constructed Networks for Different Values of P_l when (a) $Hqr = 10$ and (b) $Hqr = 100$ and when $Qz = 3.0$, $SL = 2$ and $r = 1$.

We also examine the influence of creating each peer a long link in the performance of the network. Recall that, the long links are inserted so as when the join message of a new peer n is not in the right group of peers, i.e., the join message of the new peer is located in a peer that is not similar with n, to be easy to navigate the message to another group of relevant peers that might be more relevant with n. For the same reason, long links are also useful for the routing of a query message. When the query message is not in the right group of peers, i.e., these peers do not provide enough results for the query then the long links can be used to forward the query message more easily to another group of peers that provide larger matching

results for the query. Specifically, we vary the probability of creating a long link for a peer when it joins the system and we measure the PeerRecall that the constructed networks achieve for each value that the probability takes. The probability of creating a long link takes values between 0.2 and 1.0. PeerRecall with respect to the long link probability is presented in Figure 8.3 for query workloads where the majority of queries have range 10 (Fig.8.3(a)) and 100 (Fig.8.3(b)). From the experimental results, we conclude that as the probability of creating a long link for each peer increases the performance of the constructed networks increases too. This was expected because the creation of long links contributes considerably to the navigation of the join and query messages among the groups of peers. We achieve the best performance when this probability takes the value 1.0.

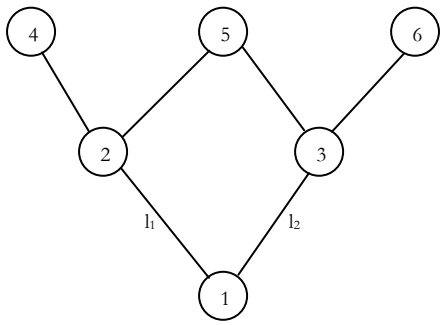


Figure 8.4: Example of a Case when more than one Routing Indexes Include the same Local Index.

8.3. Cycles in Query Routing

In this section, we analyze how cycles in query routing affect performance. In particular, it is possible that the local index of a peer n_i is included in more than one routing index of a peer n_j. For example, consider the network depicted in Figure 8.4 and that the radius of the routing indexes is set to $r = 2$. Hence, for peer 1 the routing index of link l_1 contains the local indexes of peers 2, 4 and 5, while the routing index of link l_2 contains the local indexes of peers 3, 5 and 6. This is not desirable because if the routing indexes of two links, e.g., l_1 and l_2, of peer n_j contain the local indexes of almost the same peers, then during query routing it is highly possible that two different paths from the same peer n_j, starting from l_1 and l_2 respectively, may lead us to the same peer n_i. The query message will detect that it is routed to the same peer n_i only one hop far away of it. Note that the query message visits the same peer n_j when backtracking is used, which we expect to happen occasionally, and link from n_j that has not been followed yet is selected so as to route the message.

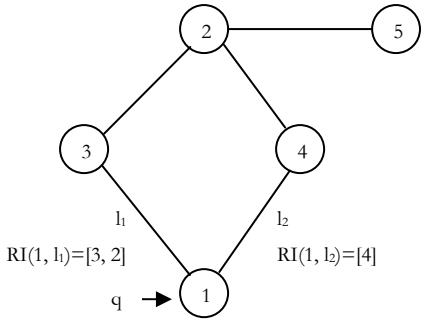
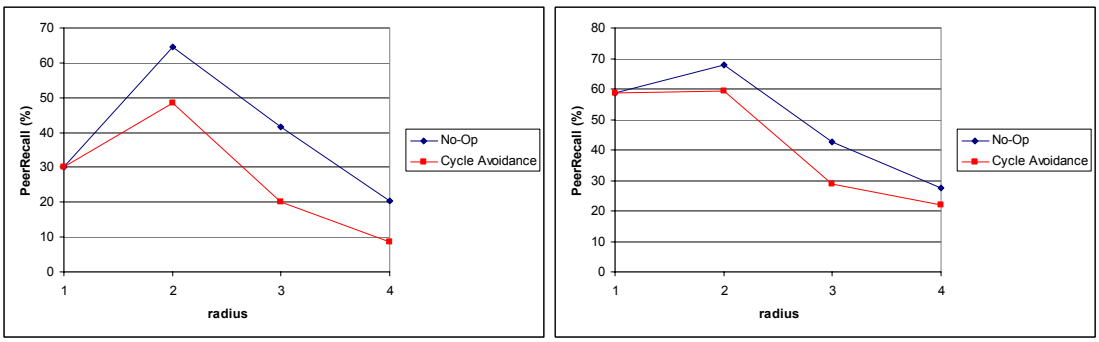


Figure 8.5: Example of a Mistaken Routing of the Query Message in the Case when we Use the Cycle Avoidance Approach.

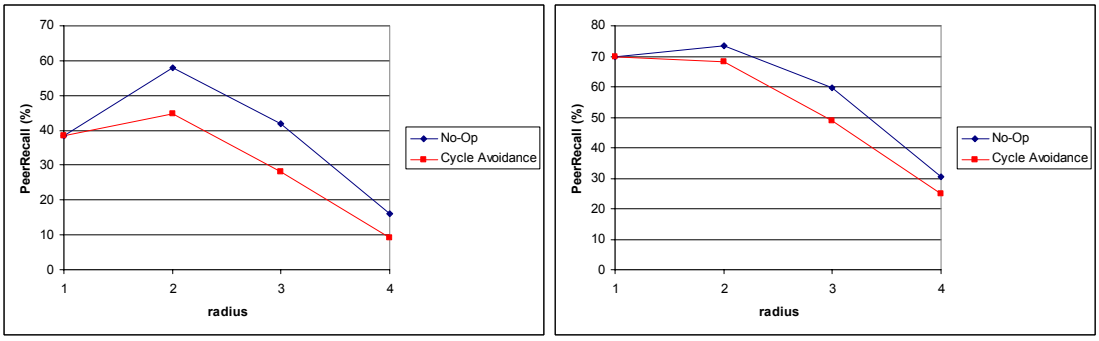
To avoid this situation, there are several approaches. In the first approach, denoted as *No-op* solution, no changes are made to the algorithms. An alternative approach, proposed in [22], is *cycle avoidance solution*. In particular, we considered that a local index is stored in only one of a peer's routing indexes. To achieve this, each peer stores the identifiers of the peers that are included in each of its routing indexes. Hence, when a peer n_j that receives the *New(LI(n_i), Counter)* message from its link *e*, that informs the peer n_j that the local index of peer n_i must be included in its routing of link *e*, before it starts the merging of the *LI(n_i)* with its routing index of the corresponding link, it first checks if this local index has already stored at the routing index of another of its links. If so, then the merge procedure is not executed.

Although this approach seems to be promising it has a serious drawback. Excluding one or more local indexes from the routing indexes of a peer n_j can lead us to take a wrong decision about which link a query message must follow. For example, consider the network depicted in Figure 8.5. Let us assume that a query *q* is initiated at peer 1 and that peers 2, 3, 4 and 5 have 1000, 0, 400 and 600 matching results for *q*, respectively. Moreover, the radius of the horizon is set to 2. Thus, in the case when we do not use the No-op solution, the routing indexes of links l_1 and l_2 must summarize the information of the local indexes of peers 3, 2 and 4, 2, respectively. When we use the Cycle Avoidance solution, the routing index of link l_1 summarizes the information of the local indexes of peers 3 and 2. Then, the routing index of link l_2 has information about the local index of peer 2 only. Thus, in this case the path that the query message follows is 1 - 3 - 2 - 5 and 1600 matching results are found. In the case when we follow the No-op solution the query follows the path 1 - 4 - 2 - 5 and finds 2000 matching results. In our implementation, we follow the No-op solution.



(a) (b)

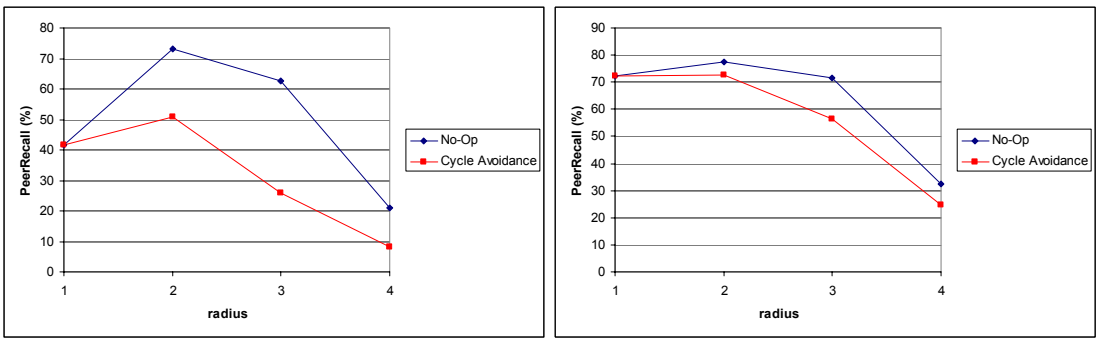
Figure 8.6: Performance of the Constructed Network Using the L_1 Distance Measure when the No-op and the Cycle Avoidance Solutions are Applied for different Values of the Radius when (a) Hqr = 10 and (b) Hqr = 100, when Qz = 3.0 and SL = 2.



(a) (b)

Figure 8.7: Performance of the Constructed Network Using the *Edit* Distance Measure when the No-op and the Cycle Avoidance Solutions are Applied for different Values of the Radius when (a) Hqr = 10 and (b) Hqr = 100, when Qz = 3.0 and SL = 2.

We conducted an additional experiment where we examine the influence of the No-op solution and the cycle avoidance solution in the performance of query routing of the clustered constructed networks. In particular, in Figures 8.6, 8.7 and 8.8 we show the PeerRecall we achieve with respect to the radius when the L_1, the edit and the wedit histogram distance measures are used for the construction of the network, respectively, and for query workloads where the most frequent queries are those with range 10 (Fig. 8.6(a), 8.7(a) and 8.8(a)) and 100 (Fig.8.6(b), 8.7(b) and 8.8(b)). The results show that the performance of all clustered networks is better when we use the No-op approach instead of the cycle avoidance approach.



(a) (b)

Figure 8.8: Performance of the Constructed Network Using the *Wedit* Distance Measure when the No-op and the Cycle Avoidance Solutions are Applied for different Values of the Radius when (a) Hqr = 10 and (b) Hqr = 100, when Qz = 3.0 and SL = 2.

8.4. Summary

To conclude, the networks that are constructed using a distance measure, L_1, edit or wedit, and routing indexes both for the construction of the p2p network and for the routing of query perform better than the *random* and the *random_join* networks. Furthermore, the networks that are constructed using the workload-aware edit (wedit) distance perform better than the clustered networks that use other distance measures, e.g., L_1 and edit, for all kinds of query workloads. Finally, using the No-op approach leads to better performance than using the cycle avoidance approach, which tries to avoid cycles in the routing of a query message.

CHAPTER 9. CONCLUSIONS AND FUTURE WORK

9.1. Summary

In this thesis, we use the idea of creating a peer-to-peer network based on the query workload [22]. In particular, the formation of peer clusters relies on two basic factors, (i) *peers content*, where two peers with similar content will participate in the same cluster, and (ii) *query workload*, so that the type and probability of queries is taken into account in creating the clusters. Hence, "popular" queries affect the formation of peer groups more than unpopular ones. The motivation for taking into account the query workload is that if some data items are queried only seldomly, we do not want them to influence clustering as much as other data items.

Initially, we introduce three workload-aware distance measures that differ on how they take into account the size of the peers and we compare them with several well known content-based distance measures so as to see how efficient is the clustering of peers that each one of them achieves. The experimental evaluation shows that in the general case the clustering that the workload-aware distance measures provide is more efficient than those created using the content-based ones. In addition, we select one of the three workload-aware distance measures, the *Manhattan workload-aware* distance measure, as the most appropriate to cluster the peers since the quality of clustering that it provides, from the perspective of PeerRecall, in compare with the quality of clustering that the other two workload-aware distance measures provide is more efficient in most of the cases.

Furthermore, based on the idea of using histograms as local indexes to summarize the content of each peer, we study the performance of different types of histograms. In particular, we select the maxdiff(v, f) and traditional equi-width histogram types and we make an extensive discussion on how these histograms are constructed. The experimental evaluation shows that the maxdiff(v, f) histogram performs much better than the equi-width; hence, we selected the maxdiff(v, f) histogram as the appropriate one for summarizing the content of the peers. Moreover, we also use histograms as routing indexes. Routing indexes are stored for each link of a peer, summarizing the content of the peers within the horizon, reachable through this link. These indexes are used in order to route a join or a query message through the link that we expect to find similar peers or enough matching results for the query, respectively. For creating routing indexes, we propose a novel procedure for merging two independent histograms into one that summarizes accurately the information of both of them.

To take into account the query workload for the formation of clusters we also propose using histograms for summarizing the peers local query workload, i.e., the queries that arrive at each peer. We introduce the W-ST histogram and we make an extensive discussion on how these histograms are constructed. We experimentally evaluate the accuracy of this histogram in compare with the W-Equi-Width histogram for several kinds of query workloads. The main conclusion from our experiments shows that the W-ST histogram can capture accurately the distribution that the query workload follows and performs much better than the W-Equi-Width histogram. Moreover, by using an aging technique the W-ST acquires the possibility to adapt to changes of the query workload distribution.

In order to construct the peer-to-peer network based on the content of the peers and the query workload, the distance between the histograms that represent the content of pair of peers should be calculated. The peers with small distance will be grouped together. The criterion based on which peers will be grouped together is the difference in the number of results that a pair of peers provide for a query workload, denoted as Histogram-Based Workload-Aware Property. Based on this property, we introduced a workload-aware histogram distance metric, denoted as wedit, that takes also into account the query workload in the calculation of the distance between two histograms. We also considered two well known content based histogram distance measures, the L_1 and the edit. It is shown experimentally and mathematically that the wedit satisfies this property, i.e., the distance between two histograms is an accurate approximation to the difference in the number of results that the two corresponding peers provide for a given query workload, while the content-based histogram distance metrics does not. In addition, we conduct several experiments where the clustering of

peers is done based on wedit distance measure and in the first case it is taken into account the global query workload, which obtained by merging the peers local query workload histograms, while in the second case it is taken into account the peers local query workloads. In both cases the clustering of peers was efficient. This experiment gives us a motivation that there are several ways to create an efficient workload-aware clustering of peers.

Furthermore, we present the corresponding procedures for the construction of the network, for updating and maintaining the routing indexes, for routing a query through the network and when a peer wishes to leave the system in the cases when we take into account the global query workload and the peers' local query workloads to form the clusters of peers.

Finally, we construct a clustered p2p network using the wedit histogram distance measure that takes into account the global query workload and we compared it with the clustered p2p networks that are constructed using the content-based histogram distance measures. We also compare those networks with a randomly constructed p2p network (random) and with a randomly constructed network that uses routing indexes only for the routng of a query message (random_join). Our experimental results show that the clustered constructed networks perform much better than the random and random_join networks. Moreover, the network which is constructed using the wedit histogram distance measure performs well and higher from all the other clustered constructed networks, which use content-based histogram distance measures for the clustering of peers, for several kinds of query workloads.

9.2. Future Work

This work takes into account the query workload for the formation of clusters in a peer-to-peer network using histograms. There are many issues that need further investigation. So far, we used the global query workload for the clustering of peers. An interesting issue is to create the peer-to-peer network based on local query workloads, as described in Chapter 7. This approach does not have the "drawback" of estimating the global query workload distribution which is a hard to deal problem.

In addition, in our thesis we considered that the query workload consists of range queries where the starting point of each query is chosen randomly while the range of queries follows a skewed distribution. Hence, the query workload can be efficiently estimated by constructing a histogram that represents accurately the distribution that the query ranges follow. An interesting issue is to include more general types of query workloads. In particular, instead of

selecting randomly the starting point of each query, we can select it following a skewed distribution. Thus, the query workload will be described by the combination of two parameters: the distribution of the query ranges and the distribution of selecting the starting point of each query. In this case, we have to construct a two-dimensional histogram to summarize the query workload and to define new histogram distance measures.

Another interesting issue is how to select the peer that will be attached through a long link with a new peer that joins the system. In our implementation, we assume that this peer is selected randomly from the list of the visited peers and does not belong to the set of peers that are selected to be linked through short links. For this case, more sophisticated methods can be used that will affect the topology of the constructed network.

Another important issue is the kind of routing indexes that we use for routing the query. In particular, the routing indexes that we use in our implementation summarize the content of all peers that are reachable from a peer *n* at a distance at most *r*. The main limitation of this kind of routing indexes is that they do not take into account the number of hops required to find the query results. For instance, consider that we want to create the *RI(n, e)* and we have two peers, *p* and *g*, which must be included in the routing index. The first one is a neighbor of *n* while the second is many hops away from *n*. In that case, the local indexes of both peers, *LI(p)* and *LI(g)*, will participate "equally" in the creation of the routing index of peer *n*, even though the probability that the query visits peer *g* is much smaller than visiting peer *p*. An alternative approach is to keep separate routing indexes for the procedures of join and routing. For the routing of a query, we propose the routing indexes to take into account the number of hops: *hop-count* routing indexes. In particular, if we want to create the *RI(n, e)* and peer *p* is reachable from peer *n* through link *e* at $m < r$ hops, then the *LI(p)* will be divided by a factor *m* and then it is going to be included in the *RI(n, e)*. Hence, as far away is peer *p* from *n*, the less the *LI(p)* contributes in the creation of the *RI(n, e)*.

Finally, we can further extend our model to support also range queries over multiple attributes. So far, we considered queries that involve a single attribute. Histogram-based local and routing indexes can be used also in the case of queries that involve more than one attribute. Let us assume *m* attributes x_i, $1 \le i \le m$. There is a number of different approaches to exploiting histograms in this case. The most simple approach is to select one of the attributes and create cluster based on this. The attribute selected may be the one present in most queries or the one with the largest skew among peers. Another straightforward extension

is to select c $(1 \leq c \leq m)$ attributes and build one histogram per each. Then, we can define the distance between two peers as the weighted sum of the c single attribute workload-aware distances. The weight for each of the distances should be based on the popularity of the respective attribute. For calculating the c single attribute workload-aware distances it is necessary to have c independent histograms, one for each attribute, that summarize the query workload over each attribute. In this approach, the workload-aware histogram distance measure that we proposed in this thesis can directly be applied. Such approaches, however, ignore any value dependencies that may exist among the attributes. To deal with this issue, we may also consider a multi-dimensional histogram built on all or a subset c of the m attributes [22]. Furthermore, it is required to create a sophisticated multi-dimensional histogram so as to summarize the query workload and define new workload-aware distance measures.

BIBLIOGRAPHY

[1] Gnutella. http://www.gnutella.com.

[2] Napster. http://www.napster.com.

[3] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In Proceedings of the ACM SIGMOD Conference on Management of Data, pages 181-192, 1999.

[4] S. Androutselis-Theotokis and D. Spinellis: A Survey of Peer-to-Peer Content Distribution Technologies. ACM Computing Surveys, Vol. 36, No. 4, December 2004, pp. 335-371.

[5] A. Bharambe, M. Agrawal and S. Seshan. Mercury: Supporting Multi-Attribute Range Queries. In Proceedings of ACM SIGCOMM '04, August 2004.

[6] N. Bruno, S. Chaudhuri, L. Gravano: STHoles: A Multidimensional Workload-Aware Histogram. ACM SIGMOD 21-24 May 2001.

[7] S-H Cha and S. N. Sribari. On Measuring the Distance Between Histograms. Pattern Recognition, 35:1355-1370, 2002.

[8] E. Cohen, A. Fiat and H. Kaplan. Associative Search in Peer-to-Peer Networks: Harnessing Latent Semantics. IEEE INFOCOM, March 2003.

[9] A. Crespo and H. Garcia Molina. Routing Indices For Peer-to-Peer Systems. In Proceedings of the 22nd IEEE International Conference on Distributed Computing Systems (ICDCS), July 2002.

[10] A. Crespo and H. Garcia-Molina. Semantic Overlay Networks for P2P Systems. In Proceedings of the 29th VLDB Conference, 2003.

[11] A. J. Demers, D. H. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. E. Sturgis, D. C. Swinehart and D. B. Terry. Epidemic Algorithms for Replicated Database Maintenance. In PODC, 1987.

[12] P. Ganesan, B. Yang and H. Garcia-Molina. One Torus to Rule them All: Multi-dimensional Queries in P2P Systems. In Proceedings of the 7th International Workshop on the Web and Databases (WebDB '04).

[13] J. Gray, P. Sundaresan, S. Eggert, K. Baclawski and P. Weinberger. Quickly generating billion-record synthetic databases. Proc. ACM SIGMOD, pp. 243-252, 1994.

[14] J. Han and M. Kamber. Data Mining - Concepts and Techniques. Morgan Kaufmann, 2001, pp. 335 – 348.

[15] K. Hose, M. Karnstedt, K. Sattler and E. Stehr. Adaptive Routing Filters for Robust Query Processing in Schema-Based P2P Systems. In Proceedings of the 9th International Database Engineering & Application Symposium (IDEAS '05), July 2005.

[16] Y. Ioannidis. Universality of serial histograms. In Proceedings of the International Conference on Very Large Databases, pages 256-267, 1993.

[17] Y. Ioannidis and V. Poosala. Balancing histogram optimality and practicality for query result size estimation. In Proceedings of the ACM SIGMOD Conference on Management of data, pages 233-244, 1995.

[18] Y. Ioannidis. The History of Histograms. In Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003.

[19] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In Proceedings of the International Conference on very large Databases, pages 275-286, 1998.

[20] G. Koloniari and E. Pitoura. Peer-to-Peer Management of XML Data: Issues and Research Challenges. SIGMOD Record, June 2005.

[21] G. Koloniari, Y. Petrakis and E. Pitoura. Content-Based Overlay Networks for XML Peers Based on Multi-Level Bloom Filters. In International Workshop on Databases, Information Systems and Peer-to-Peer Computing, 2003.

[22] G. Koloniari, Y. Petrakis, E. Pitoura and T. Tsotsos. Workload-Aware Overlay Construction Using Histograms. ACM Fourteenth Conference on Information and Knowledge Management (CIKM) , Bremen, Germany, 2005.

[23] N. Koudas, S. Muthukrishnan, D. Srivastava. Optimal Histograms for Hierarchical Range Queries. In Proceedings of the 19th ACM SIGMOD symposium on Principles of database systems, May 2000.

[24] L. Lim, M. Wang and J. S. Vitter. SASH: A Self-Adaptive Histogram Set for Dynamically Changing Workloads. In Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003.

[25] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and Replication in Unstructured Peer-to-Peer Networks. In Proceedings of ACM ICS ’02, June 2002.

[26] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, Zhichen Xu. Peer-to-Peer Computing, HP Laboratories Palo Alto, HPL-2002-57.

[27] A. Mohan and V. Kalogeraki. Speculative Routing and Update Propagation: A Kundali Centic Approach. In ICC, Alaska, USA, 2003.

[28] M. Muralikrishna and D. J. DeWitt. Equi-Depth Histograms for Estimating Selectivity Factors for Multi-Dimensional Queries. In Proceedings of the 1988 ACM International Conference on Management of Data (SIGMOD ’88), 1988.

[29] Y. Petrakis, G. Koloniari and E. Pitoura. On Using Histograms as Routing Indexes in Peer-to-Peer System. In conjunction with VLDB 2004, 2nd International Workshop on Databases, Information Systems and Peer-to-Peer Computing, Toronto, Canada.

[30] G. Piatetsky-Shapiro and C. Cornell. Accurate estimation of the number of tuples satisfying a condition. In Proc. of the 1984 ACM-SIGMOD Conf., pages 256-276, Boston, MA, June 1984.

[31] V. Poosala, Y. Ioannidis, P. Haas, and E. Shekita. Improved histograms for selectivity estimation of range predicates. In Proceedings of the ACM SIGMOD, pages 294-305, June 1996.

[32] V. Poosala and Y. Ioannidis. Selectivity Estimation Without the Attribute Value Independence Assumption. In Proceedings of the 23rd VLDB Conference, Athens, Greece, 1997.

[33] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In Proceedings of ACM SIGCOMM ’01, Aug. 2001.

[34] O. D. Sahin, A. Gupta, D. Agrawal and A. Abbadi. A Peer-to-Peer Framework for Caching Range Queries. In Proceedings of the 20th International Conference on Data Engineering (ICDE), 2004.

[35] H. Spath, Cluster Analysis Algorithms. (Ellis Horwood, 1980) pp. 27-28.

[36] K. Sripanidkulchai, B. Maggs, and H. Zhang. Efficient Content Location Using Interest-Based Locality in Peer-to-Peer Systems. In Proceedings of IEEE INFOCOM, 2003.

[37] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In Proceedings of ACM SIGCOMM ’01, August 2001.

[38] A. Strehl and J. Ghosh. Value-based Customer Grouping from Large Retail Data-Sets. In Proceedings of the SPIE Conference on Data Mining and Knowledge Discovery: Theory, Tools, and Technology II, 24-25 April 2000, Orlando, Florida, USA.

[39] C. Tang, Z. Xu and S. Dwarkadas. Peer-to-Peer Information Retrieval Using Self-Organizing Semantic Overlay Networks. In Proceedings of ACM SIGCOMM, August 2003.

[40] D. Tsoumakos and N. Roussopoulos. A Comparison of Peer-to-Peer Search Methods. International Workshop on the Web and Databases (WebDB), June 2003.

[41] R. Xu and D. Wunch. Survey of Clustering Algorithms. IEEE Transactions on Neural Networks, Vol. 16, No. 3, May 2005 pp: 645 - 678.

[42] G. K. Zipf. Human behaviour and the principle of least effort. Addison-Wesley, Reading, MA, 1949.

APPENDIX

Proof that the Manhattan Workload-Aware Distance is a metric. (page 42)

We show next that the wd_{L_1} distance measure is a metric, by proving that it satisfies the metric properties (reflexivity, non-negativity, commutativity and the triangle inequality).

The wd_{L_1} distance measure is given by the following equation

$$wd_{L_1}(n_1, n_2, W) = \sum_{q \in W} p_q \left| results(n_1, q) - results(n_2, q) \right|$$

- reflexivity: $wd_{L_1}(n_1, n_1, W) = 0$

$$wd_{L_1}(n_1, n_1, W) = \sum_{q \in W} p_q \left| results(n_1, q) - results(n_1, q) \right|$$

Since $\left| results(n_1, q) - results(n_1, q) \right| = 0$, then $wd_{L_1}(n_1, n_1, W) = 0$

- non-negativity: $wd_{L_1}(n_1, n_2, W) \geq 0$

Since $0 \leq p_q \leq 1$ and wd_{L_1} is the sum of absolute values, $\left| results(n_1, q) - results(n_2, q) \right| \geq 0$, the property holds.

- commutativity: $wd_{L_1}(n_1, n_2, W) = wd_{L_1}(n_2, n_1, W)$

$$wd_{L_1}(n_1, n_2, W) = \sum_{q \in W} p_q \left| results(n_1, q) - results(n_2, q) \right| =$$
$$= \sum_{q \in W} p_q \left| results(n_2, q) - results(n_1, q) \right| = wd_{L_1}(n_2, n_1, W)$$

- triangle inequality

$$wd_{L_1}(n_1, n_3, W) \leq wd_{L_1}(n_1, n_2, W) + wd_{L_1}(n_2, n_3, W)$$

$$wd_{L_1}(n_1, n_3, W) = \sum_{q \in W} p_q \left| results(n_1, q) - results(n_3, q) \right| =$$
$$= \sum_{q \in W} p_q \left| results(n_1, q) - results(n_2, q) + results(n_2, q) - results(n_3, q) \right| \leq$$
$$\leq \sum_{q \in W} p_q \left| results(n_1, q) - results(n_2, q) \right| + \sum_{q \in W} p_q \left| results(n_2, q) - results(n_3, q) \right| =$$
$$= wd_{L_1}(n_1, n_2, W) + wd_{L_1}(n_2, n_3, W)$$

Proof that the Distribution-Based Manhattan Workload-Aware Distance is a metric. (page 42)

We show next that the wd_{D-L_1} distance measure is a metric, by proving that it satisfies the metric properties (reflexivity, non-negativity, commutativity and the triangle inequality).

The wd_{D-L_1} distance measure is given by the following equation

$$wd_{D\text{-}L_1}(n_1, n_2, W) = \sum_{q \in W} p_q \left| \frac{results(n_1, q)}{S(n_1)} - \frac{results(n_2, q)}{S(n_2)} \right|$$

- reflexivity: $wd_{D-L_1}(n_1, n_1, W) = 0$

$$wd_{D\text{-}L_1}(n_1, n_1, W) = \sum_{q \in W} p_q \left| \frac{results(n_1, q)}{S(n_1)} - \frac{results(n_1, q)}{S(n_1)} \right|$$

Since $\left| \frac{results(n_1, q)}{S(n_1)} - \frac{results(n_1, q)}{S(n_1)} \right| = 0$, then $wd_{D-L_1}(n_1, n_1, W) = 0$

- non-negativity: $wd_{D-L_1}(n_1, n_2, W) \geq 0$

Since $0 \leq p_q \leq 1$ and wd_{L_1} is the sum of absolute values, $\left| \frac{results(n_1, q)}{S(n_1)} - \frac{results(n_2, q)}{S(n_2)} \right| \geq 0$, the property holds.

- commutativity: $wd_{D-L_1}(n_1, n_2, W) = wd_{D-L_1}(n_2, n_1, W)$

$$wd_{D\text{-}L_1}(n_1, n_2, W) = \sum_{q \in W} p_q \left| \frac{results(n_1, q)}{S(n_1)} - \frac{results(n_2, q)}{S(n_2)} \right| =$$

$$= \sum_{q \in W} p_q \left| \frac{results(n_2, q)}{S(n_2)} - \frac{results(n_1, q)}{S(n_1)} \right| = wd_{D-L_1}(n_2, n_1, W)$$

- triangle inequality:

$$wd_{D-L_1}(n_1, n_3, W) \leq wd_{D-L_1}(n_1, n_2, W) + wd_{D-L_1}(n_2, n_3, W)$$

$$wd_{D-L_1}(n_1, n_3, W) = \sum_{q \in W} p_q \left| \frac{results(n_1, q)}{S(n_1)} - \frac{results(n_3, q)}{S(n_3)} \right| =$$

$$= \sum_{q \in W} p_q \left| \frac{results(n_1, q)}{S(n_1)} - \frac{results(n_2, q)}{S(n_2)} + \frac{results(n_2, q)}{S(n_2)} - \frac{results(n_3, q)}{S(n_3)} \right| \leq$$

$$\leq \sum_{q \in W} p_q \left| \frac{results(n_1, q)}{S(n_1)} - \frac{results(n_2, q)}{S(n_2)} \right| + \sum_{q \in W} p_q \left| \frac{results(n_2, q)}{S(n_2)} - \frac{results(n_3, q)}{S(n_3)} \right| =$$

$$= wd_{D-L_1}(n_1, n_2, W) + wd_{D-L_1}(n_2, n_3, W)$$

Proof that the workload-aware edit distance (*wedit*) is a metric. (page 110)

We show next that the *wedit* histogram distance measure is a metric, by proving that it satisfies the metric properties (reflexivity, non-negativity, commutativity and the triangle inequality).

The *wedit* histogram distance measure is given by the equation

$$hd_{wedit}(H(n_1),H(n_2),H(QW)) = \sum_{j=0}^{2b-1} p_j(QW) \sum_{i=0}^{2b-1} \left| \sum_{k=0}^{j} F_{k+i}(n_1) - \sum_{k=0}^{j} F_{k+i}(n_2) \right|$$

- reflexivity: $hd_{wedit}(H(n_1),H(n_1),H(QW)) = 0$

$$hd_{wedit}(H(n_1),H(n_1),H(QW)) = \sum_{j=0}^{2b-1} p_j(QW) \sum_{i=0}^{2b-1} \left| \sum_{k=0}^{j} F_{k+i}(n_1) - \sum_{k=0}^{j} F_{k+i}(n_1) \right|$$

Since $\sum_{i=0}^{2b-1} \left| \sum_{k=0}^{j} F_{k+i}(n_1) - \sum_{k=0}^{j} F_{k+i}(n_1) \right| = 0$, then $hd_{wedit}(H(n_1),H(n_1),H(QW)) = 0$

- non-negativity: $hd_{wedit}(H(n_1),H(n_2),H(QW)) \geq 0$

Since $0 \leq p_j(QW) \leq 1$ and hd_{wedit} is the sum of absolute values, $\left| \sum_{k=0}^{j} F_{k+i}(n_1) - \sum_{k=0}^{j} F_{k+i}(n_2) \right| \geq 0$, the property holds.

- commutativity: $hd_{wedit}(H(n_1),H(n_2),H(QW)) = hd_{wedit}(H(n_2),H(n_1),H(QW))$

$$hd_{wedit}(H(n_1),H(n_2),H(QW)) = \sum_{j=0}^{2b-1} p_j(QW) \sum_{i=0}^{2b-1} \left| \sum_{k=0}^{j} F_{k+i}(n_1) - \sum_{k=0}^{j} F_{k+i}(n_2) \right| =$$

$$= \sum_{j=0}^{2b-1} p_j(QW) \sum_{i=0}^{2b-1} \left| \sum_{k=0}^{j} F_{k+i}(n_2) - \sum_{k=0}^{j} F_{k+i}(n_1) \right| = hd_{wedit}(H(n_2),H(n_1),H(QW))$$

- triangle inequality:

$$hd_{wedit}(H(n_1),H(n_3),H(QW)) \leq hd_{wedit}(H(n_1),H(n_2),H(QW)) +$$
$$+ hd_{wedit}(H(n_2),H(n_3),H(QW))$$

$$hd_{wedit}(H(n_1),H(n_3),H(QW)) = \sum_{j=0}^{2b-1} p_j(QW) \sum_{i=0}^{2b-1} \left| \sum_{k=0}^{j} F_{k+i}(n_1) - \sum_{k=0}^{j} F_{k+i}(n_3) \right| =$$

$$= \sum_{j=0}^{2b-1} p_j(QW) \sum_{i=0}^{2b-1} \left| \sum_{k=0}^{j} F_{k+i}(n_1) - \sum_{k=0}^{j} F_{k+i}(n_2) + \sum_{k=0}^{j} F_{k+i}(n_2) - \sum_{k=0}^{j} F_{k+i}(n_3) \right| \leq$$

$$\leq \sum_{j=0}^{2b-1} p_j(QW) \sum_{i=0}^{2b-1} \left| \sum_{k=0}^{j} F_{k+i}(n_1) - \sum_{k=0}^{j} F_{k+i}(n_2) \right| +$$

$$+ \sum_{j=0}^{2b-1} p_j(QW) \sum_{i=0}^{2b-1} \left| \sum_{k=0}^{j} F_{k+i}(n_2) - \sum_{k=0}^{j} F_{k+i}(n_3) \right| =$$

$$= hd_{wedit}(H(n_1),H(n_2),H(QW)) + hd_{wedit}(H(n_2),H(n_3),H(QW))$$

The workload-aware edit distance (*wedit*) satisfies the histogram-based workload-aware property. (page 112)

Proof.

To prove that the *wedit* distance satisfies the workload-aware property, it suffices to show that *wedit* is proportional to the difference in the number of results two peers, n_1 and n_2, provide for a query workload QW, i.e., $hd_{wedit}(H(n_1), H(n_2), H(QW)) \propto wd_{L_1}(n_1, n_2, QW)$, where

$$wd_{L_1}(n_1, n_2, QW) = \sum_{q \in QW} p_q \left| results(n_1, q) - results(n_2, q) \right| \quad (1)$$

Assume a value domain $D = \{0, ..., M-1\}$ of size $|D| = M$ for an attribute x.

A) Consider that we have a query workload $QW = \{(q_{ij}, f_{q_{ij}})\}$, which consist of range queries on x. The starting point of each query is chosen uniformly from the value domain D whereas the query ranges vary according to a distribution. The *wedit* distance is given by the following equation:

$$hd_{wedit}(H(n_1), H(n_2), H(QW)) = \sum_{j=0}^{2b-1} p_j(QW) \sum_{i=0}^{2b-1} \left| \sum_{k=0}^{j} F_{k+i}(n_1) - \sum_{k=0}^{j} F_{k+i}(n_2) \right| \quad (2)$$

We denote by p_{q_j} the probability of queries in the workload with range j. Since we have range queries with uniform starting point, then it holds that $p_{q_{ij}} = \frac{p_{q_j}}{|D|} = \frac{p_{q_j}}{M}$. Hence, equation (1) can be written as:

$$wd_{L_1}(n_1, n_2, QW) = \sum_{j=0}^{M-1} \sum_{i=0}^{M-1} p_{q_{ij}} \left| \sum_{k=i}^{i+j} f_k(n_1) - \sum_{k=i}^{i+j} f_k(n_2) \right| =$$

$$= \sum_{j=0}^{M-1} \frac{p_{q_j}}{M} \sum_{i=0}^{M-1} \left| \sum_{k=i}^{i+j} f_k(n_1) - \sum_{k=i}^{i+j} f_k(n_2) \right| =$$

$$= \frac{1}{M} \sum_{j=0}^{M-1} p_{q_j} \sum_{i=0}^{M-1} \left| \sum_{k=i}^{i+j} f_k(n_1) - \sum_{k=i}^{i+j} f_k(n_2) \right| \quad (I)$$

where $f_k(n)$ is the actual frequency of value k in peer n. In addition, assuming that the number of buckets for $H(n_1)$, $H(n_2)$ and $H(QW)$ is equal with the size of x's domain, i.e., $b = M$, then $2b > M$. Hence, we set $2b = M$. Equation (2) takes the form:

$$hd_{wedit}(H(n_1), H(n_2), H(QW)) = \sum_{j=0}^{M-1} p_j(W) \sum_{i=0}^{M-1} \left| \sum_{k=0}^{j} F_{k+i}(n_1) - \sum_{k=0}^{j} F_{k+i}(n_2) \right| =$$

$$= \sum_{j=0}^{M-1} \tilde{p}_{q_j} \sum_{i=0}^{M-1} \left| \sum_{k=0}^{j} \tilde{f}_{k+i}(n_1) - \sum_{k=0}^{j} \tilde{f}_{k+i}(n_2) \right| \quad (3)$$

where $\widetilde{p}_{q_j}$ is the approximated probability of queries with ranges within interval *j* resulted by $H(QW)$ and $\widetilde{f}_k(n)$ is the approximated frequency of values, using $H(n)$ of peers *n*, within interval *k*. We set $k+i=w$. Hence, equation (3) is done as follows:

$$hd_{wedit}(H(n_1),H(n_2),H(QW)) = \sum_{j=0}^{M-1} \widetilde{p}_{q_j} \sum_{i=0}^{M-1} \left| \sum_{w=i}^{i+j} \widetilde{f}_w(n_1) - \sum_{w=i}^{i+j} \widetilde{f}_w(n_2) \right| \quad (4)$$

Since, $2b=M$, then $\widetilde{f}_w(n) = f_w(n)$. Thus,

$$hd_{wedit}(H(n_1),H(n_2),H(QW)) = \sum_{j=0}^{M-1} p_{q_j} \sum_{i=0}^{M-1} \left| \sum_{w=i}^{i+j} f_w(n_1) - \sum_{w=i}^{i+j} f_w(n_2) \right| \quad (II)$$

From equations (I) and (II) we can see that $hd_{wedit}(H(n_1),H(n_2),H(QW)) \propto d_{WL_1}(n_1,n_2,QW)$ holds.

B) Assume that the query workload consists of prefix-range queries on *x*; $QW = \{(q_{0j}, f_{q_{0j}})\}$. In this case, the *wedit* distance is given by the following equation:

$$hd_{wedit}(H(n_1),H(n_2),H(QW)) = \sum_{j=0}^{2b-1} p_j(W) \left| \sum_{k=0}^{j} F_k(n_1) - \sum_{k=0}^{j} F_k(n_2) \right| \quad (5)$$

We denote by p_{q_j} the frequency of prefix-range queries in the workload with range *j*. Hence, equation (1) can be written as:

$$wd_{L_1}(n_1,n_2,QW) = \sum_{j=0}^{M-1} p_{q_j} \left| \sum_{k=0}^{j} f_k(n_1) - \sum_{k=0}^{j} f_k(n_2) \right| \quad (III).$$

In addition, assuming that the number of buckets for $H(n_1)$, $H(n_2)$ and $H(QW)$ is equal with the size of *x*'s domain, i.e., $b=M$, then $2b>M$. Hence, we set $2b=M$. Equation (5) takes the form:

$$hd_{wedit}(H(n_1),H(n_2),H(QW)) = \sum_{j=0}^{M-1} p_j(QW) \left| \sum_{k=0}^{j} F_k(n_1) - \sum_{k=0}^{j} F_k(n_2) \right| =$$
$$= \sum_{j=0}^{M-1} \widetilde{p}_{q_j} \left| \sum_{k=0}^{j} \widetilde{f}_k(n_1) - \sum_{k=0}^{j} \widetilde{f}_k(n_2) \right| \quad (6)$$

where $\widetilde{p}_{q_j}$ is the approximated probability of queries with ranges within interval *j* resulted by $H(W)$ and $\widetilde{f}_k(n)$ is the approximated frequency of values, using $H(n)$ of peer *n*, within interval *k*.

Since, $2b=M$, then $\widetilde{f}_k(n) = f_k(n)$ and $\widetilde{p}_{q_j} = f_{q_j}$. Thus,

$$hd_{wedit}(H(n_1),H(n_2),H(QW)) = \sum_{j=0}^{M-1} p_{q_j} \left| \sum_{k=0}^{j} f_k(n_1) - \sum_{k=0}^{j} f_k(n_2) \right| \quad (IV)$$

From equations (III) and (IV) we can see that $hd_{wedit}(H(n_1),H(n_2),H(QW)) = d_{WL_1}(n_1,n_2,QW)$ holds.

SORT VITAE

Theodoros Tsotsos was born in Athens on the 26th of February, 1979. He is a MSc degree student in the Department of Computer Science at the University of Ioannina since November 2003. He received his BSc degree in 2003 from the same Department. During his Bachelor thesis assignment he developed and implemented algorithms about scheduling multiple data streams in a common communication channel under the supervision of Prof. Evaggelia Pitoura. From 2002 until now, he is a member of Distributed Management Laboratory. His research interests include Distributed Systems and Peer-to-Peer Systems' design.