A Distributed and Scalable Peer-to-Peer Content Discovery System Supporting Complex Queries

Jun Gao

October 2004
CMU-CS-04-170

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Thesis Committee:
Peter Steenkiste, Chair
Christos Faloutsos
Srinivasan Seshan
Ellen W. Zegura, Georgia Institute of Technology

This research was sponsored in part by the Defense Advanced Research Project Agency and monitored by AFRL/IFGA, Rome NY 13441-4505, under contract F30602-99-1-0518, and in part by the NSF under award number CCR-0205266. Additional support was provided by Intel and a Siebel Scholar Award. The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any sponsoring party, Carnegie Mellon University, or the U.S. Government.

Keywords: Content Discovery, Peer-to-Peer, Distributed Hash Table, Rendezvous Point, Load Balancing, Range Query, Similarity Query

Abstract

A Content Discovery System (CDS) is a system that allows nodes in the system to discover content published by other nodes. CDSes are an essential component of many distributed applications including wide-area service discovery systems, peer-to-peer (P2P) systems, and sensor networks. However, existing applications have difficulties in achieving both rich searchability and good scalability. For example, applications built directly on top of Distributed Hash Tables (DHTs) are scalable but only allow exact name lookup. Peer-to-peer file sharing systems such as Gnutella and KaZaA, on the other hand, offer searching capability, but their flooding-based searching mechanism does not scale with the number of queries.

In this thesis, we present the design, implementation and evaluation of a distributed CDS that meets the scalability and searchability requirements simultaneously. Nodes in the CDS organize themselves into a structured P2P overlay network in a distributed fashion using a DHT. The CDS supports the search of highly dynamic contents represented using a descriptive attribute-value based naming scheme. We ensure scalability by deploying efficient registration and query algorithms using Rendezvous Points (RPs). To maintain high system throughput under realistic skewed load, such as flash crowds, we designed a novel mechanism that uses Load Balancing Matrices (LBMs) to eliminate hot-spots by balancing both registration and query load in a fully distributed fashion. To support complex queries, we developed two new distributed data structures, namely the Range Search Tree and Distributed KD-Tree. These data structures coupled with a set of lightweight tree-based protocols, allow the CDS to support range and similarity queries efficiently without creating bottlenecks.

To evaluate the CDS, we implemented a comprehensive simulator. Our extensive simulation results confirmed the effectiveness of the CDS design. In addition, we implemented a prototype of the CDS that includes the RP-based registration and query mechanisms with distributed load balancing. Our deployment of the prototype on the Internet (Planet Lab testbed) and its integration with a content-based music information retrieval system verified the system's feasibility and its applicability to a large category of real world applications.

To my parents

Acknowledgments

First of all, I would like to thank my advisor, Professor Peter Steenkiste, without whom this thesis would not be possible. Peter is patient and extremely responsible, and he always expects the best from me. I can't remember the number of times that I thought my design was good enough, and he challenged me to do it better. It's Peter who introduced me to the networking and distributed systems area; it's Peter who taught me what is research, and how to do good research; and it's Peter who helped me to become a researcher, and most importantly, an independent thinker.

I would also like to thank other members of my thesis committee, Professors Christos Faloutsos, Srini Seshan and Ellen Zegura. Christos is a source of inspiration, and I want to thank him for his encouragement, enthusiasm and warmth. Every time I had a database related question, he always had a comprehensive list of references ready for me. I want to thank Srini for some valuable discussions we had when I was defining my thesis, and for his continued support in helping me set up accounts on the Planet Lab testbed. I want to thank Ellen for agreeing to serve on my committee under a very short notice. I am also grateful to Ellen and Christos for writing various reference letters for me.

I had the privilege to interact with many faculty members at CMU. In particular, I want to thank Professor Hui Zhang, who has been an informal advisor to me. I was fortunate to have a chance to collaborate with him in the Darwin project. I also want to thank Professor Garth Gibson, even though I did not have a chance to work with him. I took Garth's software systems course the first semester, and the most important thing I learned from him is that when you design a system, you must know how to evaluate it. I also want to thank Professors Bruce Maggs and Greg Ganger, who have being supportive to me and my wife over the years. Special thanks go to Sharon Burks, who takes care of all the graduate students in the department, like a "den mother". I am also thankful to Barbara Grandillo, Kathy McNiff, Jennifer Lucas, and Catherine Copetas, who have all helped me in various occasions.

Many colleagues helped me in many aspects of my thesis. I enjoyed the collaboration and they all became my good friends. Particularly, I would like to thank Umair Shah, with whom I had numerous thought-provoking discussions when I started working on my thesis. Those meetings helped me through a very difficult time. I would like to thank Dr. George Tzanetakis for our successful collaboration on applying my content discovery system to his content-based music search work. I would also like to thank Adam Kushner, who worked with me on the system prototyping in the last year. His part of the work, which became his undergraduate honor thesis, strengthened my thesis, and together, we demonstrated the feasibility of my system. I would also like to thank other fellow students in Peter's group, An-Cheng Huang, Ningning Hu, Urs Hengartner and Glenn Judd, who provided many feedback to my papers and presentations in the last few years.

I would like extend my appreciation to other members of the CMCL lab over the years. In particular, I want to thank Eduardo Takahasi, Keng Lim, Prashant Chandra and Eugene Ng for our earlier collaboration in the Darwin Project. Eduardo helped me to get started with the Darwin delegate work. Keng and I worked side by side for a couple of semesters on the fruitful virtual private network project. I would also like to thank our lab manager, Nancy Miller, who did a great job in managing the machines in the lab, so that we could focus one hundred percent on research.

I have made a lot of friends over the years at CMU who made my life as a graduate student enjoyable. I want to thank them all for being my friend: Yanghua Chu, Qifa Ke, Yinglian Xie, Sanjay Rao, Umut Acar, Ted Wong, Antonia Zhai, Julio Lopez, and Peter Venable. I was fortunate to have two great officemates, Hal Burch and Leejay Wu. Hal is a wizard who knows just about everything. I turned to him whenever I had a subtle bug in my program, and he usually can figure it out in a few seconds! His help to me ranged from the meaning of a field in the IP header to the difference between two types of stock options. Leejay is a year behind me and Hal. Since Hal left our office after three years, Leejay and I kept each other accompanied for a good part of the last four years. The occasional *xevil* games kept us entertained. Leejay has become a good photographer, and I especially enjoyed his amusing squirrel shots. Leejay, being an excellent English speaker and writer, proofread almost every paper I had published, and listened to every talk I gave. For that I thank him.

I am blessed to have many truly loving friends ever since I landed in this US. I want to thank Dr. Jack Brenizer, my former advisor from the University of Virginia, and his wife Diana. The Brenizers have become me and my wife's "American parents". They helped us to organize our wedding and hosted the wedding reception at their house. Diana has always been praying for us and sending us gifts like a mom, and Dr. Brenizer offered me wise career advice whenever I called him up. I also want to thank Carl Stebbings, my long-time friend from the days I was at UVA. He even made a 7-hour trip to my defense. I would like to express my special thanks to the wonderful Mrs. Gerda Pirsch, who was my international host when I first came to the States. Mrs. Pirsch's kindness, love, and care not only made my life transition much easier, but also had a tremendous impact on my view of the world and the people everywhere. I am also very grateful to Professor Gabriel Robins at the University of Virginia. I took a couple of algorithms courses from him, and he re-ignited my interests in computer science, and encouraged me to pursue a career in this wonderful field. I am also thankful to Drs. Dimitrios Pendarakis, Debanjan Saha, Raj Yavatkar, and Professor Andrew Campbell for their support.

I want to thank my family in China. My parents are both great educators, and they started my education at a very early age and helped me to set up a life-long goal in pursuing knowledge. It's their love, encouragement and belief in me that made it possible for me to overcome every seemingly insurmountable difficulty, and to continue my next dream. I dedicate this thesis to them. My two older brothers always believed in me, and they supported me to come to the US to realize a dream that they and my parents didn't have a chance to pursue. Being so far away from home, the only reason that I can focus on my study is because that my brothers are taking good care of my parents at home. Without them, I would not have the luxury of pursuing my dream. I want to give my special thanks to my parents-in-law for taking me as their new child, and their love and support. I also want to thank my two uncles and their families for their love and support for me over the years.

Finally, I want to thank my wonderful wife, Shuheng Zhou. It is still magical to me that we found each other. Life in general, and a graduate student's life in particular, has many ups and downs. With Shuheng by my side, the amount of frustration is halved and joy doubled, and everything becomes so much more meaningful. Shuheng also has great intuition in networking research, and her critical thinking and many invaluable discussions with me directly helped me in improving many parts of this thesis. I want to thank her for her love, encouragement and being my best friend.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Internet has become an integral part of our society. People are becoming increasingly dependent on Internet applications such as the World Wide Web and Email to conduct their life and business. Continued advances in networking technology and computer hardware have enabled more and more inexpensive PCs, various devices such as PDAs, sensors, and cameras to link to the Internet with better and better connectivity. In recent years, we have witnessed the emergence of several important classes of distributed applications that are transcending how the Internet may be used beyond today's applications.

As a first example, the abundance of devices and sensors connecting on the Internet is making ubiquitous computing all but reality. Being able to efficiently discover and make use of the large amount of information gathered by these devices has become increasingly important. Consider a nationwide highway traffic monitoring service, where devices such as cameras and sensors are installed along the roadside of highways or mounted on patrol cars, to monitor traffic status, e.g., the speed they observe, road and weather conditions. These devices frequently send updates to the service to accurately reflect the current status of the highways. A driver on the road may use his PDA with a wireless connection to the Internet to query the service to get traffic information. For example, he may issue a query such as "*What is the speed at the Fort Pitt Tunnel?*" to get the speed information at a particular location, or to plan his route by issuing a query such as "*Identify the highway sections to the airport that are icy*", so that he can avoid those sections.

As another example, in the last few years, Peer-to-Peer (P2P) applications, have become one of the fastest growing applications in the history of computing. Despite the copyright issues raised by some popular P2P file swapping applications, such as Napster [47], the peer-to-peer computation model pioneered by these applications opened a new chapter of the continuing revolution of distributed computing on the Internet. Unlike the traditional client-server model, the P2P paradigm makes it possible to harness the vast computing and storage resources available on the Internet in a decentralized fashion, with little deployment cost. For example, SETI@home [62] is using Internet volunteers' home PCs' idle time to find extraterrestrial life by analyzing data collected from outer space. The collected power of these PCs is larger than any supercomputers that a group of scientists can find and afford.

A fundamental functionality that a P2P application must provide is the ability for a peer to locate resources, be it computation power or a set of files, that are available on

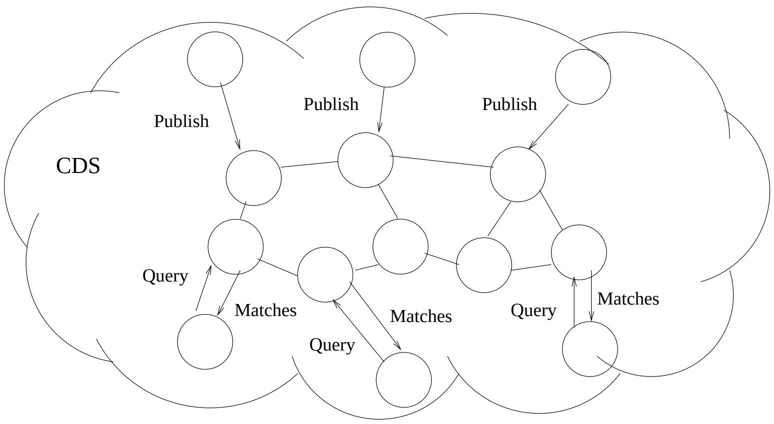


Figure 1.1: A distributed Content Discovery System. Nodes may *publish* contents, or issue *queries* to search for matched contents.

other peers. Directory services such as Napster [47], and primitive searching methods based on limited flooding such as Gnutella [30] and KaZaA [41], have become popular among Internet users. Query expressions used for the search are also rather primitive, in that they are often represented using simple metadata and keywords. Content-based searching in P2P applications is starting to draw a lot of attention, since it has the potential to make Internet-scale information retrieval possible [65].

1.1 The Content Discovery Problem

The above applications share one common functionality, *content discovery*. We use a generic term, the Content Discovery System, or CDS, to represent a system that supports the discovery of contents. Depending on where the contents are stored, a CDS may have a centralized or a distributed architecture. A web search engine can be considered as a centralized CDS for web contents, since it creates a centralized index and resolves all users' queries. In a distributed CDS, nodes form an overlay network, and a node can publish and provide contents, issue queries looking for contents, store contents or contents' metadata published by other nodes, and resolve other nodes' queries. Throughout this thesis, we sometimes refer to a node that publishes content or issue queries as a *client* or an *endpoint*.

Figure 1.1 shows an example of a distributed content discovery system. We use the term "content" in a broad sense and its meaning may differ from application to application. Content is often represented using a short description, or metadata. We also refer to this description as the *name* of the content, or *content name*. For example, in the traffic monitoring service, a piece of "content" refers to the description of a sensor or a camera, which may include attributes such as `location, speed, view,` etc. We also note that a node in this system refers to a computer and it may connect to multiple devices. In a P2P music sharing application, "content" refers to the metadata of a music file and it may include attributes such as `artist, year, album name` etc.

Besides the service discovery services and peer-to-peer object sharing systems, there exist a wide variety of distributed applications that either themselves are CDS systems or use a CDS as one of their major components. For example, publication subscription systems (pub/sub) also employ CDS. Publishers advertise the descriptions of their publications. Subscribers are clients that submit their subscriptions or queries. The CDS in a pub/sub system must match subscriptions with advertisements. Some other conventional Internet applications, e.g., Internet search engines and the DNS service, can also be viewed as CDS systems.

The primary task of a CDS is to efficiently locate the set of contents that match a client's query. As with the wide range of applications that use CDS systems, there are a variety of solutions to the content discovery problem. However, these existing solutions often display one of the following deficiencies: (1) The CDS may provide powerful searching capabilities, but the mechanism does not scale, in that when the amount of content and queries increases, the computation, storage and network bandwidth requirements grow quickly to exceed the capacity of the system; (2) Some systems may scale well, but they typically are designed with a specific application in mind, and therefore their functionality is limited and their techniques do not apply to other applications. For example, search engines deal with static and inter-linked web pages. DNS deals with hierarchical and well-administered domain names; (3) Some systems may be both scalable and general, but they only provide primitive searching capability. Systems using a basic Distributed Hash Table (DHT) to support exact name lookup belong to this category.

In this thesis, we address the wide-area content discovery problem. Our goal is to design a system that is highly scalable while also providing rich searching capability. We describe the two aspects of our proposed system in more detail as follows.

- **Rich Searchability.**

  Contents in the CDS must be searchable, in that a node can locate contents using partial information rather than having to specify a complete and exact name. More formally, searching is defined as a subset matching, i.e., a query matches a content name as long as the information specified in the query is a subset of the information specified in the matched name. For example, a user may submit a query such as "*find all songs by U2 released in 1985*" to retrieve the collection of songs without having to know each song's name.

  For discovery and exploratory purposes, a user may issue queries that are more complex than an exact query. Our proposed CDS supports complex queries such as range queries and similarity queries, which are typical operations in traditional databases and information retrieval systems. For example, a driver may issue a range query such as "*find the speed information between Exit 1 and 20*". A user using a P2P content-based music information retrieval system may issue a similarity query "*find the 10 songs most similar to x.mp3*".

  The type of content that can be searched by the CDS is general and can be dynamic:

  Flat content representation. In the applications we are targeting, the descriptions of content do not have to be hierarchical in nature like domain names or directories. Content names in these applications often display a flat structure. Examples include the description of a service, the metadata of a file, and the feature vector of an mp3 file. Furthermore, contents are independent of each other, and they do not link to each other like web pages.

Dynamic content. The description of a piece of content may change over time. For example, when a camera observes a different speed, it must change its description and announces it to the CDS system. As such, it is possible for the CDS to answer queries such as "*find all the road sections that are congested*". This is in contrast to a typical search engine, which builds its database based on static content information. For example, a search engine may be able to return a list of links of classical music stations, but it typically can not give the user an answer to "*find the station that is currently playing Beethoven's Concerto No. 9*", and the user may have to click each of the potentially many links to find the desired station.

- **Good Scalability.**

  We are interested in applications that have many nodes running on the wide area Internet. The number of nodes in the system may be on the order of tens of thousands. As such, both the amount of content and the number of queries are large. By scalability we mean that as the load (e.g., the registration and query rate) to the system increases, the performance of the CDS, such as throughput and response time, must not degrade significantly before the system as a whole reaches its capacity. More specifically, the CDS must meet the following scalability challenges.

  Scale with queries. The issue is that how many messages will be involved in resolving a query. For example, a system that must broadcast a query to a large portion of the network is inefficient, and the system as a whole does not scale with the number of queries.

  Scale with registrations. The CDS may observe high registration load resulting from possibly frequent dynamic content updates. Existing systems such as search engines typically do not concern themselves with registration load, since they are targeting contents that are mostly static such as web pages.

  Scale with the load regardless the load distribution. Realistic load is often skewed. For example, the distribution of query interests are often skewed and their popularity may be due to a certain event, the so-called "flash crowd" effect. As an example, the word "Olympics" becomes a heavily queried term while the Olympic Games is going on. This is true for both registration and query load. The proposed CDS must perform well under this type of skewed load.

  Scale for complex queries. Queries such as range queries and similarity queries are complex, and the CDS must be able to handle them efficiently. For example, a user may specify a query that covers a large range, the cost of naively sending the query to a number of nodes proportional to the range length may be prohibitive.

While we are designing a searchable and scalable CDS system that meet the above challenges, the system we present in this thesis does not address the following.

- Our system does not replace an Internet search engine or a full-fledged information retrieval system. In particular, our system does not analyze the semantic meaning of a piece of content, or the relationship between pieces of content (e.g., web pages), and their relevances to a query. However, it is possible to add such analysis to nodes in our system as a complementary functionality.

- Our system does not enforce strict content consistency. Our system is designed for large-scale loosely controlled peer-to-peer applications. We use soft-state protocols and do not require content providers to actively invalidate their registered contents when they leave the system. This is a realistic assumption in that in peer-to-peer systems, content providers may simply leave without notifying the system. In our system, when a provider registers a piece of content, it must associate an expiration timer with the content, and the node that receives the content will purge the content when the timer is up. Before the timer expires, it is possible that the content metadata a client retrieves is invalid, if the content provider had left. By using a timer for each registered content, we make sure that this type of inconsistency is not persistent and it is bound by the timer. For applications that require strict consistency such as a bank transaction system, it is conceivable that more sophisticated protocols can be added to our system to achieve the consistency goal.

- Our system does not address content delivery, which, in addition to content discovery, is a functionality often needed in the applications we are targeting. In our system, we implement the content discovery functionality, in that the system returns a set of matching names for each query. This may be sufficient for some applications, e.g., when a user tries to retrieve the speed information in the traffic monitoring service. However, in some other applications, post-content discovery actions, such as retrieving the discovered content, may take place. For example, a user may want to connect to the radio station he found to listen to the live music. Similarly, after discovering the IP address of the peer that owns a particular file, a peer may then connect to that peer and request the file.

1.2 Related Work

Existing CDS systems have difficulties in achieving both rich searchability and scalability. We survey the CDS solutions in existing applications. Based on how the CDS network is organized, we classify the systems into two categories: centralized and distributed. For each solution, we first describe the general mechanisms and then present specific systems that use this solution. We also discuss why these systems do not meet our goals set for a CDS.

1.2.1 Centralized Solutions

In centralized solutions, the resolver or the cluster of resolvers form a central data repository. The resolvers are typically administered by one organization and are co-located in the same geographical location. The data repository may be built using several mechanisms: (1) content providers actively register with the central resolver, e.g., Elvin [22] in pub/sub system, and Napster [47] in peer-to-peer file sharing; (2) Central resolvers actively search for contents, e.g., search engines [31] typically find web contents by periodically crawling the web. Users of a centralized system submit their queries to the central resolver(s), which subsequently resolves queries by examining its database.

The centralized system allows flexible searches. With access to the complete information of the contents in the system, a centralized system can conduct complex analysis in answering a query. For example, search engines use sophisticated "ranking" algorithms (e.g., PageRank [13] in Google [31]) to return a user the set of matches corresponding to a query in the order of their relevancy to the query.

There are several problems associated with the centralized solution. It typically does not scale well as the amount of content and the number of queries increase, since the central server must maintain all the content information in the system, and process every content registration and query request. The need to support frequent dynamic updates makes this problem even more prominent. Techniques, such as deploying a cluster of well-connected machines as the resolvers and performing dynamic load balancing across these resolvers, can improve the scalability.

Another issue is the robustness of the system. The centralized infrastructure forms a single point of failure, and makes the system vulnerable to attacks. If the resolver(s) is down or is attacked (e.g. by a DDoS attack), the system will not be available to the users. Many such attacks in recent years have temporarily but successfully taken down some of the major Internet search engines. The most recent instance occurred while writing this thesis on 7/26/2004, when the MyDoom worm partially knocked out the service of Google and some other search engines [1].

Finally, the deployment cost of a centralized service that can scale up is very high, and corporate support is typically required to create and maintain such a service. This is in sharp contrast with the P2P paradigm, where Internet hosts can create a powerful online service quickly at almost no extra cost.

1.2.2 Distributed Solutions

In a distributed solution, nodes in the CDS form an application level overlay network for content registration and query resolution. We further classify systems using a distributed solution into three categories based on their overlay network topology: (1) Hierarchical tree based system, (2) Unstructured overlay, and (3) Structured overlay.

Hierarchical Tree-based systems

In this type of systems, resolvers are organized into a hierarchical infrastructure. The main advantage of the tree topology is that system-wide content broadcasting and query flooding can be avoided. Specifically, it is possible for each node to maintain a relatively small content database to avoid the full replication of all the content, thus limiting the scope of registration and searching. Instead of broadcasting a content registration to all nodes in the system, it is only populated in a certain subtree based on some aggregation mechanism. A user first sends its query to the node it connects to. If the query can not be resolved there, it will be sent up the tree. As a result, query flooding can be avoided. Many existing systems use a tree-based infrastructure for content discovery, e.g., the DNS [45] service, the pub/sub system designed by Yu *et al.* [74], service discovery systems such as SDS [17] and the Service Location Protocol(SLP) [34].

The hierarchical organization scales well for applications in which content names follow a hierarchical format, e.g., domain names and hierarchically named service descriptions. However, the hierarchical topology has some fundamental shortcomings.

First, the amount of information that must be propagated up the tree may be high. In DNS [45], the hierarchical names can be aggregated well due to their hierarchical nature, thus it limits the amount of information that must be propagated among servers. The DNS also scales well with query load by using caching, since the hostname-IP address binding is fairly static for most of the hosts on the Internet. For content names that are not naturally hierarchical, such as the services we are targeting, nodes high in the tree are likely to be overloaded. SDS [17] addresses this problem by using Bloom filters to compress the amount of data to be transferred.

Second, nodes high in the tree, the root node in particular, may experience high query load since all queries that can not be resolved will have to go up the hierarchy.

Third, the tree topology may not be robust, in that each node could become a potential single point-of-failure, and the crash of any node will cause the partition of the tree.

Systems based on unstructured overlays

In this type of systems, nodes are organized into a general graph. Based on the way content is distributed/replicated in the CDS network, we examine two types of mechanisms.

- Content broadcasting solution

  In this type of system, a node registers its content (names) to the nodes that it is connecting to in the overlay network. A node that receives a registration store the name in its local database, and then broadcasts it to all of its neighbors. Multicast protocols such as DVMRP [71] are typically used to avoid message looping. In this solution, a content name is replicated at all nodes in the system. With this full replication, query resolution can be done efficiently: a client sends its query to the node that it connects to, and the node compares the query with entries in its database and resolves the query. This type of mechanism is used in many systems, e.g, INS [6] for service discovery, and Siena [14] for pub/sub systems.

  This approach does not scale with the amount of content, because (1) each resolver must store all the contents in the system; and (2) each new content name will cause waves of broadcast messages throughout the network.

  Siena [14] tries to improve scalability by exploring the relationship between different registrations (subscriptions), e.g., a resolver that receives a new content name, broadcasts it only if the new name is not “covered” by some previous content name that has been broadcast before. However, the degree of aggregation that can be done is often limited, since the names (i) do not necessarily relate to one another, and (ii) may not display a hierarchical nature. Gryphon [7] improves the resolver’s local matching algorithm by constructing a matching tree off-line.

- Query flooding solution

  In this solution, nodes do not actively register their content names with other nodes in the network. Instead, they only maintain their own contents. When a node receives a query, it checks its local set of content names, and if it can not find a match for the query, it will forward the query to its neighbors in the system. As such, a query may have to traverse a large portion of the network before a match can be found. The propagation of the query may follow mechanisms such as Breadth First Search (BFS) and Depth First Search (DFS). Systems using this type solution include P2P applications Gnutella [30] and Freenet [25].

The sheer number of messages incurred by each query clearly results in poor scalability with the number of queries. Consequently, a querying node may experience long delay in resolving one query, and may not get results even if there exists a match in the system.

There have been some recent work that tries to improve Gnutella's scalability. KaZaa [41] improves upon Gnutella by deploying supernodes, but the system wide flooding may still be used in the worst case. To improve scalability, caching is often used [63]. However, caching works well only if queries in the system display temporal and spatial correlation, and it does not work well for dynamic content. In [16], several techniques have been proposed to make Gnutella more scalable, and it is shown that aggressive content replication mechanisms can make the system perform well for popular queries.

Systems based on structured overlays

One reason causing the scalability problems in above systems is that content names generally do not have any relationship with the nodes that host them, and therefore broadcasting or query flooding is often inevitable.

In recent years, Distributed Hash Tables (DHTs), e.g., Chord [64], CAN [55], Pastry [58], and Tapestry [75], have been proposed as a scalable and robust layer for building large scale distributed applications. In a DHT, a data item can be associated with the node that will host it. This is typically done by applying a uniform hash function to the data item, and then storing it on the node whose ID in the DHT's key space is closest to the hash. This primitive makes searches based on exact name lookups efficient, and system-wide flooding or broadcasting is avoided.

There have been many recent activities in building a variety of applications on top of DHTs. However, these systems have various limitations, and do not meet the design goals of the CDS. Here, we discuss a few example work in this domain, and we will present more detailed comparison in individual chapters, since our system is also built on top of a DHT. CFS [18] and PAST [59] are two distributed file systems built on top of Chord [64] and Pastry [58] respectively. Scribe [15] is a pub/sub system built on top of Pastry [58]. One significant problem with these systems is that they work only for fixed content names, and do not allow searching. For example, to use the DHT, a client must know the exact name of a file it is looking for before hand. Twine [10] is a service discovery service built on top of Chord, and it allows search, but it does not address load imbalance issues, which are common in realistic applications. PIER [38] is a distributed database built on top of Chord, and handles traditional database operations such as "distributed join". However, it does not support rich queries such as range and similarity.

1.3 Thesis Statement

In this thesis, *we present the design, implementation and evaluation of a distributed Content Discovery System that meets both the searchability and scalability goals.*

Here we briefly highlight some of the key techniques we used in supporting this thesis.

- **Distributed DHT-based P2P system.** Nodes in our CDS organize themselves into a structured P2P overlay network using DHT. The P2P infrastructure provides a decentralized and robust system, and avoids problems such as single-point-of-failure and concentration of load encountered by the centralized and tree-based systems. The DHT layer ensures a scalable overlay network management and routing. Nodes in the CDS share equal responsibilities: a node can publish contents, store contents published by other nodes, issue queries, and resolve others' queries.

- **Rendezvous Points based scalable search algorithms.** We support search in our system by representing contents and queries with descriptive attribute-value pairs. While enabling search, we ensure scalability through the use of Rendezvous Points (RPs) to avoid system-wide message flooding at both content registration and query time. Each content name is registered with a small set of nodes in the system, known as the name's RP set, and queries are directed to proper RP nodes depending the query itself. Queries are resolved locally on RP nodes to minimize the amount of network traffic.

- **Distributed load balancing to ensure system's scalability under skewed load.** The system's performance will degrade quickly when RPs are overloaded. We design a novel mechanism that uses Load Balancing Matrices (LBMs) to dynamically balance both registration and query load in a truly distributed fashion to ensure the system's throughput, even under extremely skewed load, such as flash crowds. As a result, the load balancing mechanism allows the system to scale further.

- **Supporting complex queries efficiently using distributed tree-based protocols.** The basic searching mechanism is based on exact matching between the query and a subset of the content description. Clients use a CDS for exploration often prefer open-ended queries, as they do not know what contents are available. Straightforward extension of the basic search mechanisms to handle complex queries such as range and similarity queries is inefficient.

  We developed a set of dynamic and adaptive distributed tree-based protocols to support complex queries. In particular, we design a distributed search tree structure that automatically aggregates registrations to handle range queries efficiently, and we design a distributed kd-tree structure to cut down the search space for similarity queries. While supporting complex queries efficiently, these protocols are executed in a fully distributed fashion, and do not create any bottlenecks in the system.

1.4 Contributions

The design, implementation and evaluation of a distributed Content Discovery System that is scalable while supporting complex searching paradigms is the subject of this thesis. The primary contributions of this thesis are as follows:

- **System architectural contribution.**

  We contribute to the architectural design of large scale, scalable distributed Internet applications by identifying *content discovery* as a fundamental building block for these applications.

  The CDS layer is completely separated from another building block, the DHT, and runs on top of it. The CDS layer exports a simple API, *registration* and *query* to allow a wide range of applications such as wide area service discovery and P2P information retrieval systems to be built upon.

- **Distributed protocol design contribution.**

  We make contributions in advancing distributed protocols design. We demonstrated that some traditionally difficult tasks in distributed systems that are done in centralized systems, can indeed be done in a fully distributed fashion.

  In particular, we contribute a distributed load balancing protocol that uses a structure called Load Balancing Matrix (LBM) to dynamically balance both registration and query load in a fully distributed fashion.

  We contribute two tree based protocols, namely the Path Maintenance Protocol (PMP) and the Tree Maintenance Protocol (TMP), that convert efficient centralized spatial indexing data structures, the Range search Tree (RST) and the distributed kd-tree (DKDT), into a distributed system that is efficient to support distributed range queries and similarity queries.

- **Software system contribution.**

  We contributed a comprehensive simulation implementation of the designed CDS. Our evaluation results validated the effectiveness of the system.

  We contributed a prototype implementation of the CDS that is built on top of a real DHT, and runs on the Internet. The prototype implements the core registration and query protocols and the distributed load balancing mechanisms. The prototype and its deployment demonstrate the feasibility of the proposed CDS system.

  We contributed a proof-of-concept implementation of a P2P content-based music information retrieval application that utilizes the CDS.

1.5 Thesis Overview

The remainder of this thesis is organized as follows.

In Chapter 2, we present the CDS system architecture. We discuss how the CDS is layered on top of a DHT. We detail the basic RP-based registration and query mechanism, and show that these operations are supported in an efficient and scalable way.

In Chapter 3, we present our distributed load balancing mechanism that dynamically balances registration and query load when RP nodes are overloaded. We analyze the protocol's performance and present comprehensive evaluation results that demonstrate the effectiveness of the mechanism.

In Chapter 4, we show how we provide efficient support for range queries in the CDS. We start by introducing a search structure, the Range Search Tree, and then go on to present a set of protocols and adaptive mechanisms that support range queries efficiently. We present both quantitative analysis and extensive simulation results to verify our design.

In Chapter 5, we describe how the CDS supports similarity queries. Our design is based on Distributed KD-Tree (DKDT), a distributed version of a kd-tree. We extend the protocols for range queries to work for the DKDT. We also present several optimizations that try to overcome the dimension curse problem. Our simulation results again showed the effectiveness of the system.

In Chapter 6, we describe a prototype implementation of the system, called Camel. Camel is built on top of Chord, and as a separate library for distributed applications to link to. Camel implements the basic system design and the distributed load balancing mechanisms. We also show some experimental results obtained from a deployment of Camel on Planet Lab [51], a planetary scale overlay network testbed on the Internet.

We conclude the thesis and present some future research directions in Chapter 7.

Chapter 2

CDS System Architecture

In this chapter, we present the CDS system architecture. In Section 2.1, we start by giving a formal description of how contents are represented in our system, and what we mean by search. In Section 2.2, we discuss in detail a three-layered software architecture on each CDS node. In particular, we explain how CDS utilizes the underlying DHT substrate to form a peer-to-peer overlay network, and how it enables applications to register content names and to conduct searches. In Section 2.3, we describe an example application that uses the CDS to conduct content-based music information retrieval.

Section 2.4 presents the Rendezvous Point (RP) based registration and query mechanism in the CDS. This basic RP based mechanisms allow the CDS to achieve the searchability and scalability design goals. We discuss two limitations of this basic design in Section 2.5, namely, it does not handle skewed load and it does not support complex queries efficiently. We will present solutions to these problems in the next three chapters. We present work related to the basic system in Section 2.6 and summarize the chapter in Section 2.7.

2.1 Content Naming Scheme

To provide content searchability, applications built on top of the CDS layer use a flexible attribute-value based naming scheme, similar to what is used in [6], [14]. Contents are represented using attribute-value pairs (AV-pairs). For example, in a service discovery system, a device may be described with attributes such as `Type`, `Location`, `Model`, etc. In multimedia applications, such as the P2P music information retrieval system described in [28], attributes that are used to describe an mp3 file include not only manually configured ones such as `Artist` and `Song Name`, but also features extracted from the audio signals, such as `Tempo` and `Strength`. Attributes may also include information such as the `IP address` of the node that owns the content for content delivery purpose after the content is discovered.

We refer to the collection of the AV-pairs as the "content name (CN)", or "content description". In our terminology, "content discovery" means the discovery of the "content name", not the actual content. We consider mechanisms such as contacting the device or retrieving the actual file after the "content discovery" step as a separate function, known as the "content delivery".

In a content name, an AV-pair is specified using an equality predicate in the form of

$$
\begin{array}{|lll|}
\hline
a_1 = v_1 & & \\
& & a_{11} = v_{11} \\
& & \quad \ldots \\
& & a_{12} = v_{12} \\
& & \quad \ldots \\
& & \ldots \\
a_2 = v_2 & & \\
\ldots & & \\
a_n = v_n & & \\
\hline
\end{array}
$$

Figure 2.1: An example content name.

$\{a_i = v_i\}$, where a_i is an attribute, and v_i is its value. In contrast, as we will describe shortly, AV-pairs in a query may contain inequality predicates. Figure 2.1 is an example CN. It is represented using the following format:

$$CN : \{\{a_1 = v_1, a_{11} = v_{11}, a_{12} = v_{12}\}, a_2 = v_2, ..., a_n = v_n\},$$

or

$$CN : \{\{a_1v_1, a_{11}v_{11}, a_{12}v_{12}\}, a_2v_2, ..., a_nv_n\},$$

where a_iv_i is the short form for $a_i = v_i$. Languages such as XML may be used to describe content names. Figure 2.2 is an example name for a highway monitoring camera.

```
Camera ID = 5562
Camera Type = Q-cam
Highway Number = I-279
    Exit Number = 4
City = Pittsburgh
Speed Measured = 45MPH
Road Condition = dry
Connection availability = yes
```

Figure 2.2: The content name for a highway camera.

There are two types of attributes: *independent* and *dependent*. As its name implies, an independent attribute does not depend on other attributes. In Figure 2.1, a_1 and a_2 are independent attributes. In comparison, dependent attributes must co-exist with some other attributes. For instance, the `Exit Number` attribute in Figure 2.2 is a dependent attribute of `Highway Number`, since it is not meaningful to search for cameras based on the exit number only. On the other hand, if the application allows searches based on exit number alone, then `Exit Number` can be promoted to an independent attribute.

An attribute can be *static* or *dynamic*. The value of a static attribute does not change over the lifetime of a content name, whereas a dynamic attribute may take on a different value at different times for the same content. In Figure 2.2, for instance, `Camera ID` and `Camera Type` are static attributes, and `Speed Measured` and `Road Condition` are dynamic attributes, since the speed observed at a section of the road is constantly changing, and the road condition may be dry or wet. The attributes related to location information such as `Highway Number` may also be dynamic if we consider cameras that are mounted on vehicles. When the value of a dynamic attribute changes, the content name changes, and this may require content names to be frequently updated.

Searching based on Subset Matching

To search for contents, a user may submit a query to the system. A query is also comprised of a set of AV-pairs, e.g., $Q : \{a_1v_1, a_2v_2, ..., a_mv_m\}$ contains m AV-pairs. Note that we allow inequality signs such as "$<, >, \leq$" and "$\geq$" in queries. In this and the next chapter, we consider queries that only contain equality predicates, and in Chapter 4, we extend the system's functionality and allow queries to contain inequality predicates. If a query contains dependent attributes, it must also contain the attributes that they are dependent on. For example, $Q : \{\{a_1v_1, a_{11}v_{11}\}, a_2v_2\}$ is a valid query, but $Q : \{a_{11}v_{11}, a_2v_2\}$ is not a valid query.

Content names registered in the CDS are searched via *subset matching*. More specifically, a content name matches a query as long as the set of AV-pairs in the query is a subset of the set of AV-pairs in the content name. The content name may contain extra AV-pairs that are not in the query. For example, for query $Q : \{a_1v_1, a_2v_2\}$, the matched content names must contain both $a_1 = v_1$ and $a_2 = v_2$.

Many queries can match a given content name. In particular, for a content name consisting of n independent AV-pairs, the number of matched queries, M_Q, is exponential in n. More specifically, it is equal to the number of subsets (excluding the empty set) of the content name:

$$M_Q = \binom{n}{1} + \binom{n}{2} + ... + \binom{n}{n-1} + \binom{n}{n} = 2^n - 1. \tag{2.1}$$

For names that contain dependent AV-pairs, the actual number of matched subsets will be smaller, since we consider a dependent AV-pair and its parent AV-pair(s) together as one "combined" AV-pair. For example, we would not have $\{a_1v_1, \{a_1v_1, a_{11}v_{11}\}\}$ as a possible subset, instead we would only have $\{a_1v_1, a_{11}v_{11}\}$. The exponential relationship between M_Q and n demands a careful design of the content registration and query scheme that is scalable.

2.2 System Architecture

Nodes participating in the CDS connect to each other in a peer-to-peer fashion to form a CDS overlay network. Figure 2.3 shows the software architecture on a node. It consists of three layers above the network (TCP/IP) layer. The CDS layer is designed as a common communication layer on which higher level applications, such as service discovery and file sharing, can be built. The CDS layer is in turn built on top of a scalable distributed hash table (DHT), such as Chord[64], CAN [55], Pastry [58], or Tapestry [75]. We now describe the functionalities of each layer in detail.

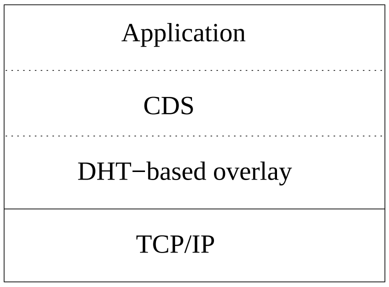


Figure 2.3: CDS node architecture.

2.2.1 DHT-based Overlay Substrate

The CDS system uses the DHT layer [3] for two purposes: (1) constructing and managing the overlay network, and (2) delivering messages within the overlay network.

In a DHT, when a node first joins the system, it obtains a numerical identifier, known as its node ID, from an m-bit key space (e.g., in Chord [64], $m = 128$). The node ID may be computed locally, e.g., by applying a system-wide hash function to some local information such as the node's IP address. During the joining process, the node must first contact some nodes that are already in the system, and then according to its node ID, "insert" itself into the proper place in the overlay network. The node ID serves as this node's overlay network address, and the node is responsible for a contiguous region around its node ID in the key space. Overlay networks built using a DHT are structured in such a way that node IDs encode overlay network topology information: The node ID determines the set of nodes that this node will be neighboring with, and which neighbor to use when forwarding a message in the overlay network.

The CDS network formed using a DHT has the following important properties:

- **Self-organizing.** The network is formed in a fully decentralized fashion, and no centralized entity is needed.
- **Scalable.** DHT-based systems are scalable with the number of nodes in the system, by keeping both the number of routing table entries on a node and the number of overlay hops between any two nodes small, e.g., both are $O(\log N_c)$ in Chord[64], where N_c is the number of nodes in the network.
- **Robust.** The system is robust in that nodes can freely join or leave the network, and a node join or departure/crash has a localized effect: only a small number of nodes need to change their routing tables. For example, when a node leaves the system, the segment that it was responsible for will be taken over by other live nodes that are close to this node in the key space.

All DHTs export a common simple API to the upper layer, `lookup(key)` [3], which performs a well-defined task: given a key, find the node in the system responsible for the key. In the CDS, similar to other DHT-based system such as CFS [18], we use the extended DHT API, `put(key, message)` [3]: when given a `key` and a `message`, the DHT not only finds the node responsible for the `key`, but also delivers the `message` to it. The DHT layer does not dictate how the CDS chooses the `key` for `message`. The DHT layer on the receiving node does not examine the message, but it simply passes the message to the CDS layer. The CDS layer processes the message and takes different actions, such as accepting registrations and resolving queries, based on the type of the message. As such, the CDS layer uses the DHT purely as a transport vehicle.

2.2.2 Applications

Applications built on top of the CDS can perform two operations, register a content name or issue a query to search for contents. The application carries out these operations by invoking one of the two methods provided by the CDS API:

- `register(content_name)`. When this function is invoked, the CDS layer will send the content name to a set of selected nodes in the network, using the DHT primitive `put()`.
- `search(query)`. When this function is invoked, the CDS layer will send the query to proper nodes in the network to retrieve the set of content names that match the query. The message passing again is done by invoking the underlying DHT's `put()` function.

The parameters `content_name` and `query` are the AV-pair representations of a content name or a query as we defined earlier.

2.2.3 CDS Functionality

The main task of the CDS layer is to determine where to register a content name and where to send a query for resolution, once it receives a content name or query. The primary design goal is to meet the scalability and content searchability requirements. We first briefly examine the design space for the CDS. In particular, we have two concerns:

1. **Registration cost.**

   How many DHT nodes does a client node have to register a content name with?

2. **Query cost.**

   How many DHT nodes does a client node have to contact to resolve a query?

We illustrate these two metrics in Figure 2.4 along with several CDS design options. In a centralized design, all names and queries are sent to one central location. While this may look attractive since the individual cost of a registration and query is minimized, the central location constitutes the system's single point-of-failure and bottleneck. In the registration broadcasting design, a content name is sent to every node in the system, and the cost of registration is thus high. However, the query cost is low, as any query can be resolved locally. Similarly, in the query flooding design, the registration cost is 0, in that all nodes keep their content names to themselves. However, it is expensive to resolve queries, since a query may have to be sent to all nodes in the system to get resolved. Neither of these approaches is scalable due to the prohibitive number of registration or query messages.

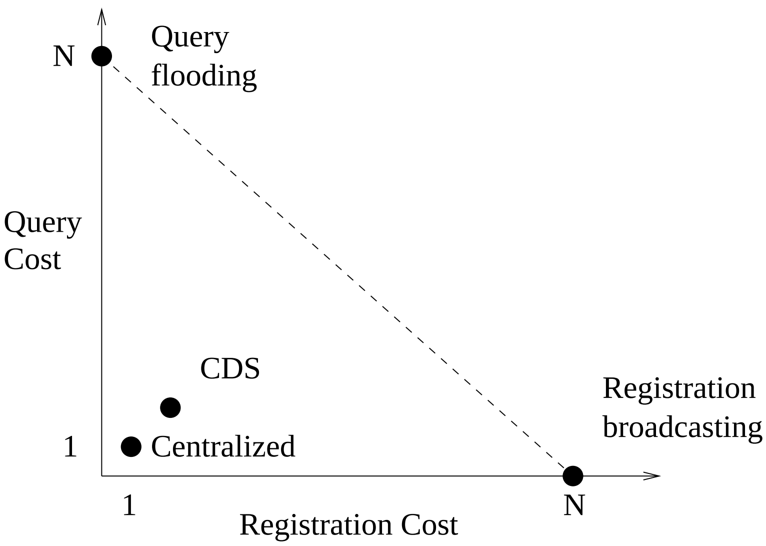


Figure 2.4: CDS design space.

In our system, we introduce a scalable approach based on *Rendezvous Points (RPs)*. In this design, a content name is only registered with a small number of nodes in the system, the RPs; thus the full duplication of content names at all nodes is avoided. Queries are directly sent to the proper RPs for resolution according to the AV-pairs in the queries, and no network-wide searching is needed. The term "rendezvous" is used because the RPs are where queries and the matched names meet.

2.3 CDS Application Example: A P2P Music Information Retrieval System

In this section, we describe an example CDS application that we built for content-based music information retrieval [28, 69].

A substantial percentage of the Internet traffic today consists of music files exchanged by Internet users over P2P networks. In such a system, each peer may contribute to the music collection by making a set of files on the local machine available to others. The P2P protocol allows a peer to discover files stored on other peers. Unlike traditional client-server based system, the decentralized and self-organizing nature of P2P networks makes them a more suitable and powerful platform for resource sharing. However, the usefulness of existing P2P systems is often limited by their searching capability. For example, they only provide primitive searching methods, such as keyword searching, or searching based on a combination of simple metadata. Users of a P2P system may desire more powerful searching capabilities such as searching based on the music content itself. As an example, they may want to search for the album that contains an unknown song recorded from radio, or they may want to find more songs that are "similar" to the unknown song in terms of *tempo*.

We designed a scalable P2P system that supports content-based retrieval of music files

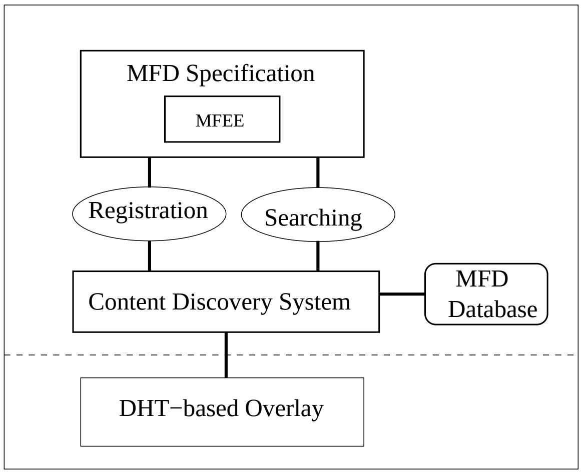


Figure 2.5: Software architecture on a peer node in the music information retrieval system.

as well as the traditional attribute-value based search using simple metadata. The system is built on top of the CDS. Figure 2.5 shows the software architecture on each peer node. A user may perform two types of operations: registering shared files and initiating searches. For registration, a shared audio file is represented using a content name, which consists of a set of AV-pairs. A content name here is also known as a Music File Description (MFD). The MFDs are either specified by the user or automatically generated by the Music Feature Extraction Engine (MFEE). The criteria of a search is formulated as a query, which is also in the form of an MFD.

The MFEE component takes as input an audio file in compressed format, such as MP3, the MPEG audio compression standard, and outputs a feature vector of AV-pairs that characterizes the particular musical content of the file. In our system, we use the feature set proposed in [68] for the purpose of musical genre classification. This feature vector captures aspects of instrumentation and sound texture (what instruments are playing and their density distribution over time), rhythm (fast-slow, strong-weak), and pitch content (harmony) and has been shown to be an effective representation for the purposes of classification and retrieval of music. Examples of such features include *tempo, beat strength* and *degree of harmonic change*. The different types of information represented by the feature vector combined with the query flexibility of the system supports a rich variety of queries. For example, a user can search only on the basis of rhythmic content while ignoring other similarity aspects.

Registrations and queries are carried out using the CDS. Based on the AV-pairs in the MFD, the CDS chooses a set of peers in the system that should receive the registration or query, computes their node IDs, and then uses the DHT layer to deliver the registration or query messages to them. We use Figure 2.6 to illustrate the steps involved in registering an MFD. The steps for issuing a query are similar. In Step 1, the application passes the MFD to the CDS, and the CDS then determines a destination node ID based on the name, and passes it to the DHT layer in Step 2. The DHT layer sends the message through the overlay network to the destination node in Step 3. At the remote node, the DHT layer passes the message to the CDS layer (Step 4), and then the CDS layer stores it in the local

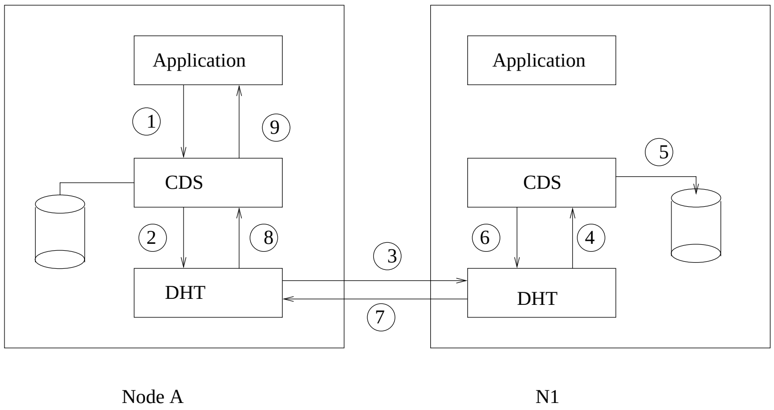


Figure 2.6: The steps involved in registering a content name.

MFD database (Step 5). Step 6-9 are confirmation messages to let the registering node know whether the registration is completed successfully or not.

Upon the arrival of a query, the peer examines its database and returns the set of MFDs that match the query to the query initiator. The matched MFDs may contain sufficient information that the query initiator is looking for, e.g., the song name of an unknown piece of music. The query initiator may further download the actual music file from the peer that owns it.

2.4 Rendezvous Points-based Design

We now describe the algorithms the CDS layer uses to select nodes in the system to handle registrations and queries, and explain how a node matches queries to registered contents.

2.4.1 Registration

To register a content name, $CN_1 : \{a_1v_1, a_2v_2, ..., a_nv_n\}$, that has n independent AV-pairs, the CDS layer on the provider node must determine the set of nodes that should receive this name. It chooses the set of nodes by applying a system-wide hash function, $\mathcal{H}$, to each AV-pair in the content name. $\mathcal{H}$ can be any uniform hash function such as SHA-1 [4]. The *complete* content name is then sent to each of the nodes in the DHT whose IDs are numerically closest to each of the hash value using the underlying DHT's `put()` primitive. We call each selected node a Rendezvous Point (RP) for this content name, and together these nodes (n of them assuming no hash collision) form the RP set for this content name. Figure 2.8 is an illustrative example showing the registration of two content names.

The pseudo code for the registration algorithm is listed in Figure 2.7. It first applies $\mathcal{H}$ to each AV-pair (Line 3), and the result of the hash is used as the *key* argument to call the DHT function PUT. Line 4 builds a REGISTRATION message that contains the complete content name. The message is then supplied to PUT as the second argument (Line 5). Subsequently, the DHT layer will deliver the message to the node whose ID is closest to N_i. For clarity, we refer to this node as node N_i.

```
Register(CN) {
  /* CN is the content name to be registered */
  foreach AVpair a_i v_i in CN {
      N_i ← H(a_i v_i);
      message ← Make-message(REGISTRATION, CN);
      Put(N_i, message); /* call the DHT API function */
  }
}
```

Figure 2.7: The algorithm for registering a content name.

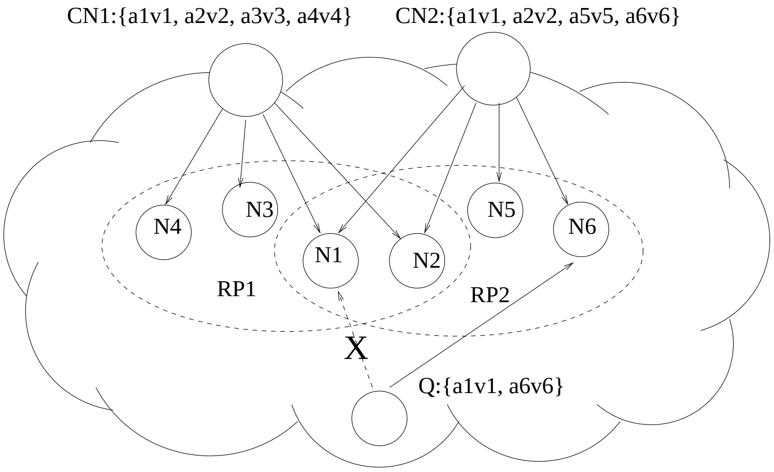


Figure 2.8: Example registration and query processing with RP set.

For a dependent attribute, the hash function is applied to both the dependent AV-pair and all of its parent AV-pairs together. In the example in Figure 2.2, attribute a_1 has two dependent attributes a_{11} and a_{12}, which take on values v_{11} and v_{12} respectively. Two nodes are determined with the following computation:

$$N_{11} \leftarrow \mathcal{H}(\{a_1 = v_1; a_{11} = v_{11}\}),$$

$$N_{12} \leftarrow \mathcal{H}(\{a_1 = v_2; a_{12} = v_{12}\}).$$

Upon receiving a registration, the node stores the content name in a local *name database*. The node then sends a confirmation message back to the registering node. From an RP node's point of view, it becomes a specialized node for the AV-pairs that are mapped onto it. For example, N_1 contains all the names in the system that contain $\{a_1 v_1\}$.

Nodes maintain content names in a soft state fashion. An expiration timer is set for each registered content name depending on the type of application. For example, for mostly static contents such as file descriptions, the period may be on the order of hours or days, but for dynamic service descriptions, a content name may expire in minutes. When a content name expires, the node will remove it from the name database. As such, content providers must periodically refresh the names they have registered before. The refreshing mechanism provides protection against certain types of failures. For example, when an RP node leaves or crashes, the refresh messages will be delivered to a live node whose ID is now closest to the hashed value. For example, in Figure 2.8, suppose N_1 leaves/crashes, now in the DHT, node N_1' who is closest to N_1 will receive the refresh of CN_1 and CN_2. It is worth pointing out that when a name contains dynamic attributes, the refresh messages will typically register the name at a different set of RPs when attribute values change.

2.4.2 Query

Applications may search for contents by issuing queries. The CDS layer on a querying node must determine the set of RPs that may contain matching content names. Since all content names that contain the pair $\{a_i v_i\}$ are stored in the node $N_i (= \mathcal{H}(a_i v_i))$, a query $Q : \{a_1 v_1, a_2 v_2, ..., a_m v_m\}$ can be sent to any one of the m nodes, $N_1, ..., N_m$ (Figure 2.8). If the query contains dependent attributes, similar to the registration mechanism, the querying node determines the node corresponding to the dependent attribute by applying $\mathcal{H}$ to the dependent AV-pair and all of its parent AV-pairs.

Given these m candidate nodes, the querying node may pick one randomly and send one query message to that node. It may also choose a node that has better performance based on previous experience. In the example shown in Figure 2.8, to resolve $Q : \{a_1 v_1, a_6 v_6\}$, either N_1 or N_6 may be used. If the querying node had previous experience that N_6 gave a shorter response time, e.g., due to a smaller name database or closer network proximity, it will send the query to N_6 rather than N_1. In the next chapter, when we introduce a load balancing mechanism to handle skewed loads, the importance of carefully selecting which AV-pair to use to resolve a query will become clearer.

Once an RP node receives a query, it simply examines its name database, and determines the set of names that match the query by comparing each name's AV-pair list with that of the query's. In Figure 2.8, suppose N_6 is chosen to receive the query. N_6 will return CN_2 to the querying node as it matches both $a_1 v_1$ and $a_6 v_6$.

Since a node can resolve queries locally, no communication between RP nodes is needed. An alternative to having queries fully resolved at one RP node is to have a client send its query to multiple nodes, each of which resolves the query partially and returns any matches. The client then performs a "join" operation to determine the final set of matched names. While this approach reduces the computational load on resolver nodes, it adds potentially significant communication overhead due to the exchange of large sets of partial matches across the network. Given that exact matching for AV-pairs is a relatively lightweight operation, it is more efficient to do the complete matching on the selected RP node.

2.4.3 System Properties

We summarize the basic RP-based system's properties here.

- **Registration and query efficiency.** The RP-based CDS design is efficient for both registrations and queries. Hashing each AV-pair individually for registration yields an RP set of size n for a name that has n AV-pairs, thus requiring $O(n)$ messages per registration. In real-world applications, n is typically a small number (e.g., < 50), and registrations can be done efficiently. The system is also efficient for queries, since typically only one node is needed to resolve any query.

Being efficient for endpoints also implies that the system is scalable with the number of registrations and queries, since only a small number of messages is needed to handle each registration and query.

- **Searchability.** The design ensures content searchability. Any query can find all the content names that fully contain the query. For example, the query $Q : \{a_1v_1\}$, which has only one AV-pair, can discover all content names that contain $\{a_1v_1\}$ by visiting node N_1.

  An alternative approach that would also ensure that a content name can be found by any query that is the subset of the content name is to register the name with nodes corresponding to each of its $2^n - 1$ subsets. This approach is clearly inefficient, since the number of registration messages needed is exponential to n.

- **CDS network scalability and robustness.** Since the CDS uses a DHT to build and manage the CDS overlay network, and route messages within this network, the system inherits the good scalability and robustness offered by the DHT. Most notably, the amount of routing state maintained on each node is small, and the number of hops between any two nodes is also small.

  The use of a DHT as the substrate also simplifies the RP discovery step, which is typically required for endpoints to locate the RP points in similar systems. For example, the IP multicast protocol PIM-SM [20] uses bootstrap configuration mechanism. Since both registrations and queries select proper RP nodes based on their own AV-pairs, there is no need to establish a separate discovery mechanism for them to find the appropriate RPs.

2.5 Challenges Faced by the Basic CDS

While the CDS's RP-based registration and query design is scalable and enables search, we observe that the system will perform poorly under skewed load, and that it does not support complex queries efficiently.

2.5.1 Handling Skewed Load

In the basic RP-based design, each AV-pair is used as the argument by the hash function and is mapped onto a node. Hashing attribute and value together to determine the set of RP nodes provides a natural way of spreading registrations and queries to more nodes in the system. Since each node in the DHT is responsible for an equal-sized segment, the number of AV-pairs each node receives is fairly uniform. However, the registration and query loads observed by the nodes are determined by the AV-pair distributions in content names and queries, i.e., how many content names or queries contain this AV-pair and how frequently they arrive at this node. The basic design performs well when the distributions are even, in which case nodes in the system observe similar load.

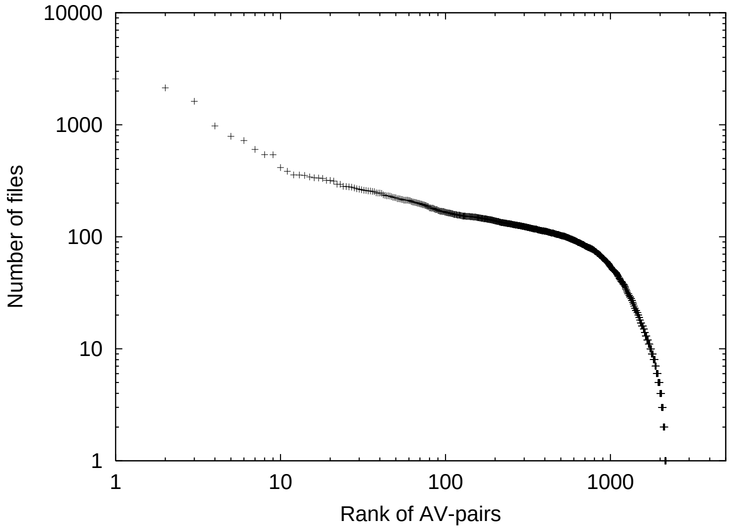


Figure 2.9: Popularity distribution of feature attributes.

In real-world applications, the distribution of AV-pairs in registrations and queries are likely to be skewed as some AV-pairs are common or significantly more popular than others. As a specific example, while evaluating the music information retrieval system presented in Section 2.3, we analyzed a set of 5,000 mp3 audio files representing a variety of genres and styles. For each file, we extracted 30 feature attributes that characterizes the musical content of the file. Figure 2.9 is a log-log plot of the feature AV-pair distribution in these files. There are 2,178 distinct AV-pairs, and the distribution is highly skewed: the most common AV-pair (ranked 1) appears in 53% of the files and 41 AV-pairs only appear in 1 file.

The skewed distribution also occurs in many other applications. For instance, it has been observed that the popularity of keyword search strings in both traditional web searches [56] and Gnutella peer-to-peer networks [63] follows a Zipf-like distribution, which is highly skewed. Note that keyword-based search is a special case of AV-pair based search where attribute names are omitted. The skewed distribution implies that some nodes in the CDS system may be overloaded while others are underutilized. More specifically, consider the case where the number of names that contain $\{a_i v_i\}$, $N_{a_i v_i}$, follows a Zipf distribution:

$$N_{a_i v_i} = N_s \cdot k \cdot \frac{1}{i^\alpha}, \tag{2.2}$$

for $i = 1 ... N_d$, where N_d is the number of different AV-pairs in the system. k and α are two parameters, where α is close to 1. N_s is the total number of names in the system and i is the rank of AV-pair $\{a_i v_i\}$ in terms of its frequency of occurring in names; $i = 1$ corresponds to the AV-pair that is contained in the most number of names. As an example, suppose an application has $N_s = 10^5$ names, and $k = 0.5, \alpha = 1$. Half of the 10^5 names would contain the most popular AV-pair, which would be sent to one node. In the meantime, for nodes that correspond to AV-pairs ranked from 10^3 to 10^4, each would receive fewer than 50 names. Clearly, a few nodes would be inundated by registrations, while the majority of the nodes in the system would be rarely used.

For the CDS to work well for real applications, it must handle the load concentration problem. In the next chapter, we will explain in detail a set of load balancing mechanisms that enable the system to perform well under skewed load.

2.5.2 Complex Queries

The basic design works well for searching when the comparison is based on exact matching of the AV-pairs, e.g., "*return the list of cameras that observes a speed of 25mph*", or {`speed=25`}. However, this simple hashing mechanism makes it difficult to support queries where the search criteria is less specific, e.g., a query such as "*return the list of cameras for which the observed speed is less than 25mph*" may be used to find all the highway sections where the traffic flow is heavy. This query may be represented as {`speed <= 25`}. We call this type of queries, range queries.

A naive of way of handling range queries is to break up a range query into many point queries and resolve them one by one. For example, if we know *a priori* that the smallest granularity for the attribute `speed` is 1 mph, we can decompose {`speed <= 25`} into 25 point queries, {`speed=1`}, {`speed=2`}, ..., {`speed=25`}. This algorithm becomes very expensive for queries with large ranges, which are common for exploration and discovery purposes.

Another type of complex query is the similarity query, where the user specifies a search criteria and asks the system to return 1 or more content names that are "closest" to the search criteria. As an example, suppose that each camera in the traffic monitoring service has a longitude and latitude attribute. A similarity query may be "*find the camera that is closest to the accident location at* `Longitude = 30, Latitude = 30`". This type of query is even more expensive to resolve, since the user does not know where the cameras are, and thus does not know how many sub-queries he may have to issue to find the nearest camera.

In Chapter 4, we present a set of distributed protocols and algorithms that allow the CDS to handle range queries efficiently. Furthermore, in Chapter 5, we extend the range search mechanisms to support similarity queries.

2.6 Related Work

One of the fundamental problems in designing the CDS is where to store the content names so that they can be discovered efficiently. As we discussed in Section 1.2, many systems can be viewed as CDS systems, ranging from web search engines and directory services to peer-to-peer file sharing systems. We classify these systems based on how the content resolvers are organized and compare them with our system. Centralized systems, such as Napster[47] and Google[31], use a set of central servers to index contents and resolve queries. These servers may become the bottleneck as load increases, and form the single point-of-failure of the system, thus making it vulnerable to censors and attacks such as Denial-of-Service. Our CDS is distributed and uses a more robust overlay resolver network.

Content resolvers may be organized hierarchically into a tree structure, e.g., in DNS [45] and SDS [17]. In general, these systems are designed for hierarchical content names, such as domain names and directories [70]. To prevent overloading resolvers high in the tree, DNS relies on caching to scale to the Internet level and SDS uses bloom filters to reduce load propagated up the tree. In contrast, our system is designed to handle more general content names that do not necessarily have a hierarchical nature.

Systems based on an unstructured general resolver network such as INS [6], Siena [14], Gnutella[30], and Freenet [25] require flooding the network at either content registration time or query resolution time. Hence these systems do not scale with the number of content names and queries. More recent systems such as KaZaA [41] scale better by leveraging a two-tier infrastructure and relying on "supernodes" to suppress the flooding. In our system, we eliminate network-wide flooding at both registration and query time by establishing Rendezvous Points.

Using RPs to improve a distributed system's scalability is not a new concept. For example, in the original design of the IP multicast protocol [19], the group information is pushed to every router so that any host connecting to that router can join the group. This leads to the explosion of group state that must be maintained on every router. The PIM-SM [20] wide area extension to the original protocol distributes the group information only to a small set of routers, called Rendezvous Points, and the assumption is that most clients are not interested in the groups, and a client that wants to join a group must do so by explicitly contacting the RP points. As a result, the amount of state must be maintained on routers is dramatically reduced.

A hash-based peer-to-peer system such as Chord [64], CAN [55], Pastry [58], and Tapestry [75], uses a scalable protocol to form a self-organizing structured overlay network. While not directly supporting general content searchability, these systems provide an efficient solution to content name lookup by binding a complete content name, such as a file name, to a specific node in the system using a hash function. These systems relate to our work in two ways. First, the DHT abstraction [3] in these mechanisms provides the CDS system a scalable and robust substrate for building the CDS overlay network and for routing CDS messages. Second, our CDS system extends the basic lookup functionality and supports content searchability by using AV-pairs.

More recently, many systems have been built on top of DHTs to leverage DHTs' decentralized, scalable and robust properties. We briefly discuss some of these works, where the notion of Rendezvous Point appears.

Scribe [15] is a large-scale event notification infrastructure for topic-based pub/sub applications. Scribe shares some of the design choices made in our CDS. It is also layered on top of a DHT, namely Pastry [58]. A Rendezvous Point in Scribe refers to the node that corresponds to the hash of a topic in the pub/sub system. The RP acts as the root of the multicast tree for the subscribers to this topic. The main difference between our work and Scribe is that we use RPs to enable efficient content search, while Scribe focuses on how to build efficient multicast trees for event dissemination.

Several projects built systems on top of DHTs to enable content search. In [56], the focus is on efficient keyword-based searching. Unlike our system, a query is sent to each node that is responsible for one of the keywords in the query, and partially matched results are first collected over the network and then "join" operations are performed to get the final matches. Techniques such as bloom filters and caching are used to reduce the network bandwidth consumption. We avoid the transmission of potentially large number of partially matched results by storing complete content names (all keywords of a document in [56]'s context) on RP nodes to allow full resolution locally.

Twine [10] is a resource discovery system built on top of Chord. Resource descriptions are separated into "strands", which are essenstially the combination of AV-pairs in our system, and then mapped onto nodes in the resolver overlay network, similar to our basic system. A resolver that corresponds to a random strand in the query is used to resolve the query. However, Twine does not address the critical load balancing problem, which we will discuss in the next chapter.

2.7 Chapter Summary

In this chapter, we presented the design of a distributed and scalable Content Discovery System. The CDS is fully distributed in that no central entity is needed for both registrations and queries. Registrations and queries are carried out efficiently by using RPs, and we avoid network wide message flooding for both registrations and queries. The efficiency for individual registrations and queries means that the CDS scales well as the registration or query load increases. While maintaining efficiency, the CDS's AV-pair based content representation coupled with subset matching allows flexible searching. Finally, we must note that the CDS system uses a DHT to form the CDS overlay network and deliver messages, and as such, the network inherits the scalability, robustness, and self-organizing properties from the DHT.

Chapter 3

Distributed Load Balancing

As we mentioned in the last chapter, the CDS's performance will degrade quickly when the RPs are overloaded due to the skewed AV-pair distribution in realistic registrations or queries. In distributed systems, dynamic load balancing is a technique that is often used to increase a system's throughput under skewed load. However, most systems use some type of centralized entity to coordinate the balancing of load. For example, a popular ecommerce site may use a front-end web server to direct requests to a cluster of back-end database servers in a load balanced way. In our decentralized self-organizing peer-to-peer based CDS system, load balancing based on a centralized "load balancer" is no longer applicable, since the load balancer will become a single point-of-failure and bottleneck of the system, and thus defeat the decentralized property of the CDS.

In this chapter, we present a distributed load balancing solution that allows the CDS to dynamically discover and utilize lightly loaded nodes to share the registration and query load of heavily loaded nodes. We first introduce a distributed data structure, the load balancing matrix (Section 3.1), and then show how it is used and managed (Section 3.2) in a distributed fashion to eliminate hot spots in the system and thus improve the system's throughput. We describe a simulation implementation of the CDS in Section 3.4. Extensive simulation results are presented in Section 3.5 that validate the effectiveness of the load balancing mechanism. We then present work related to distributed load balancing in Section 3.6 and summarize the chapter in Section 3.7.

3.1 Load Balancing Matrix (LBM)

For a popular AV-pair, the CDS system uses a set of nodes instead of one node to share the registration and query load. To ensure that this set of nodes can be addressed directly by endpoints in a distributed fashion, we organize this set of nodes into a logical matrix, the Load Balancing Matrix (LBM). Figure 3.1 shows the layout of the matrix for AV-pair $\{a_iv_i\}$. Each node in the matrix has a column and row index, (p, r), and node IDs are determined by applying the hash function, $\mathcal{H}$, to the AV-pair, and the column and row indices together:

$$N_i^{(p,r)} \leftarrow \mathcal{H}(a_iv_i, p, r). \tag{3.1}$$

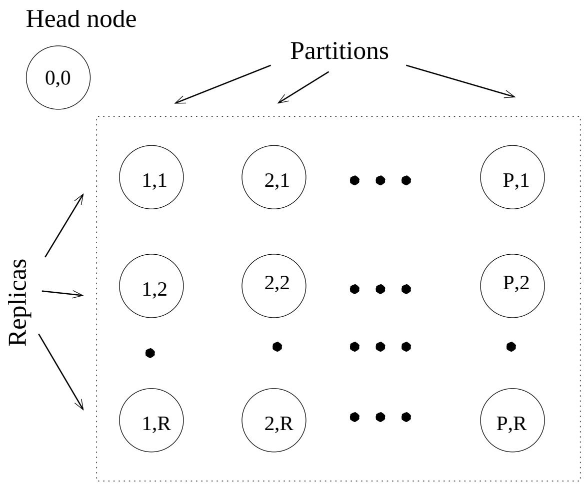


Figure 3.1: Load balancing matrix for $\{a_i v_i\}$.

Each column in the matrix stores one subset, or *partition*, of the content names that contain $\{a_i v_i\}$. Nodes in the same column are *replicas* of each other: they host the same set of names.

The matrix dynamically expands or shrinks along its two dimensions depending on the load it receives. It starts with one node when the registration and query load are low; this corresponds to the basic system. New partitions are added to the matrix when the registration load of the pair $\{a_i v_i\}$ increases, and new replicas are added when the query load increases. Matrices may end up in different shapes. For example, a matrix may have only one row, when only the registration load is high, or one column, when only the query load is high. Each matrix uses a node, called the *head node*, with ID $N_i^{(0,0)} \leftarrow \mathcal{H}(a_i v_i, 0, 0)$, to store its current size and to coordinate the expansion and shrinking of the matrix.

To expand matrices, each node in the system maintains three thresholds: T_{CN}, the maximum number of content names a node can hold, T_{reg}, the maximum rate of registration it can sustain, and T_q, the maximum query rate the node can sustain. Three corresponding low thresholds are also set for shrinking purpose. Note that a node may belong to multiple matrices when multiple AV-pairs are mapped onto it, and the thresholds are used to regulate the aggregated load from all of these pairs. In the following discussions, for simplicity, we assume all nodes are homogeneous in that they have the same computation power and network connectivity.

Next we describe the registration and query operations when LBMs are present in the system.

3.1.1 Registration with LBM

In the basic system, a content provider registers its content name with RP nodes that corresponds to each of the AV-pairs in the name. In contrast, with LBMs, the provider must register its content name with one column of nodes in each matrix that corresponds to an AV-pair (Figure 3.2).

The pseudo code for registration is listed in Figure 3.3. To register with matrix LBM_i, the content provider must first discover its size: the number of partitions, P, and the

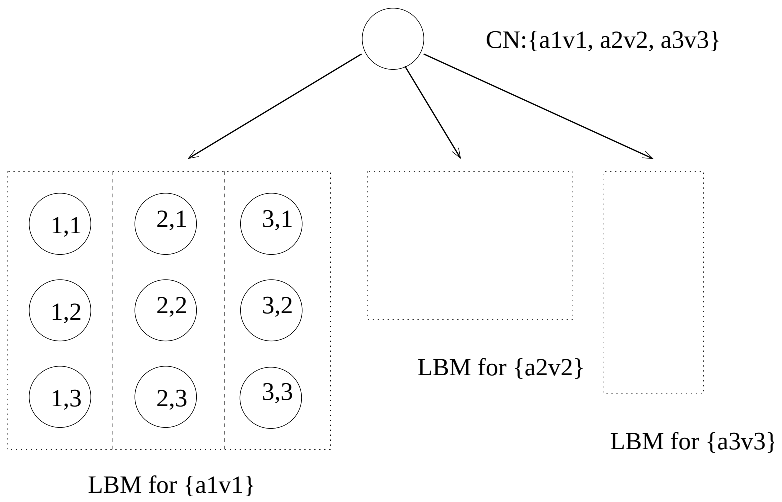


Figure 3.2: Registration with load balancing matrices.

number of replicas, R. It can do so in several ways. For example, it may be able to retrieve the size from the pair's corresponding head node (Line 4 in Figure 3.3). We will discuss below how the size may be discovered efficiently if the head node is busy. Once the size of the matrix is found, the content provider selects a *random* partition between 1 and P (Line 5). It then computes the node IDs in this partition and registers with each of the replicas within the partition (Lines 6-9). Since the partition within the matrix is randomly selected, the registration load within the matrix is distributed evenly.

Matrix size discovery via probing

We now elaborate on how the size discovery is carried out in the event that the head node is down or becomes a bottleneck. Under such scenarios, the provider may find out the matrix size by directly sending probe messages to nodes that are potentially in the matrix. For example, to discover P, the provider may first estimate a maximum number P_0, and probe a node in the P_0th partition, e.g., $N_i^{(P_0,1)}$. Node $N_i^{(P_0,1)}$ can determine whether it belongs to LBM_i by checking its database to see if it has seen $\{a_i v_i\}$ before. Since partitions are indexed contiguously, the current number of partitions can be efficiently discovered in $O(\log P_0)$ steps via binary probing between partition 1 and P_0. As a further optimization, content providers may cache an AV-pair's matrix size and use it without re-discovering it. This is for example useful when refreshing a previously registered name.

3.1.2 Query

Similar to the basic system, to resolve a query, a client can send it to the the matrix that corresponds to any AV-pair in the query. The pseudo code of the query resolution algorithm is listed in Figure 3.4. There are two types of queries depending on how many AV-pairs a query may contain.

When multiple AV-pairs are present in a query, we use a two-pass *query optimization* algorithm to determine which pair a client should use for its query, since the cost of resolving

REGISTER(`name`) {
    foreach AVpair $a_i v_i$ **in** `name` {
        $N_i^{(0,0)} \leftarrow \mathcal{H}(a_i v_i, 0, 0);$
        (P, R) $\leftarrow$ RETRIEVE-MATRIX-SIZE($N_i^{(0,0)}, a_i v_i$);
        p $\leftarrow$ GENERATE-RANDOM-NUMBER(1, P);
        foreach r **in** [1, R] {
            $N_i^{(p,r)} \leftarrow \mathcal{H}(a_i v_i, p, r);$
            PUT($N_i^{(p,r)}$, `name`);
        }
    }
}

Figure 3.3: The algorithm for content providers to register with LBM.

a query is determined by the number of partitions in the selected matrix. First, the client probes the sizes of all the matrices corresponding to each AV-pair in the query using one of the mechanisms presented above (Lines 2-5), and it then selects the one with the fewest partitions (Line 6). In practice, since the matrix sizes can be cached, the cost of the probing phase can be amortized when the client issues multiple queries.

In the second step after a matrix is selected, the client must send the query to all the partitions in the matrix (Line 7), if it needs to collect all possible matches. In reality, sending to a subset of the partitions may return the client sufficient number of results. Since nodes in the same column are replicas of each other, the query needs only to be sent to one node in each column, and the client chooses a *random* node (Lines 8-10). This random selection mechanism ensures the query load is distributed evenly within a matrix.

If the query contains only one AV-pair, the query optimization mechanism will not be applicable, and the query will be sent to the matrix corresponding to that pair. When this matrix has a lot of partitions, the query cost will be high if we must get all matches and thus send the query to all partitions. To reduce cost in this case, the client can contact a small subset of the partitions to receive enough matches, e.g., 100. The client may then refine its query based on the returned names by adding more AV-pairs. In fact, this is the behavior of Internet users when using a search engine. A study conducted in [56] shows that 71.5% of the searches found in one large web cache contains more than two keywords. With more than one AV-pairs in the query, the CDS can again use the query optimization algorithm described above.

3.2 Matrix Management

Given the registration and query algorithms, we know that the cost of registering an AV-pair with an LBM and querying an LBM are determined by the number of rows and columns in the matrix respectively. Since both the registration and query load are dynamic, the matrix must change its size accordingly to ensure the efficiency of the system. More specifically,

SEARCH(query) {
    foreach AVpair $a_i v_i$ **in** query {
        $N_i^{(0,0)} \leftarrow \mathcal{H}(a_i v_i, 0, 0);$
        $(P_i, R_i) \leftarrow$ RETRIEVE-MATRIX-SIZE($N_i^{(0,0)}, a_i v_i$);
    }
    $k \leftarrow$ index of minimum P_i for all i
    foreach p **in** [1, P_k] {
        r $\leftarrow$ GENERATE-RANDOM-NUMBER($1, R_k$);
        $N_k^{(p,r)} \leftarrow \mathcal{H}(a_k v_k, p, r);$
        send_to($N_k^{(p,r)}$, query);
    }
}

Figure 3.4: The search algorithm with LBM.

when a matrix receives a high load, it must expand itself quickly to accommodate the excessive load. When the load decreases, the matrix should shrink itself to reduce registration and query cost.

In this section, we present the matrix expansion and shrinking mechanisms. The key feature in our design is that both operations are done in a fully distributed fashion. This is made possible by the even load distribution in a matrix. We describe the detailed expansion and shrinking mechanisms using matrix LBM_i as an example. LBM_i corresponds to $\{a_i v_i\}$, and its head node is $N_i^{(0,0)}$. We assume that there are currently P_i partitions and R_i replicas in LBM_i.

3.2.1 Partition Expansion

New partitions are added to LBM_i when the existing partitions in the matrix receive high registration load. We define an LBM's *expansion region (ER)* as the set of partitions that were last added to the matrix. We first describe the expansion algorithm and then explain our design decisions.

- When the registration load on a node in the matrix reaches the threshold T_{CN} or T_{reg}, it will send an `INC_P_REQ` request to the head node, $N_i^{(0,0)}$ (Step ①in Figure 3.5).
- The head node doubles the number of partitions to $2P_i$, upon receiving the first such request from a node in the current ER (Step ②).
- The head node then sends an `INC_P_CMD` command to the nodes with partition index from $P_i + 1$ to $2P_i$, informing them that they are now in the matrix and will be responsible for $\{a_i v_i\}$ (Step ③). Partitions $P_i + 1$ to $2P_i$ become the new ER.

After the partition expansion, when a node needs to register a new content name that contains $\{a_i v_i\}$, the registering node will discover LBM_i has $2P_i$ partitions, and then select one to register this name. Hence, the registration load is shared by the expanded matrix.

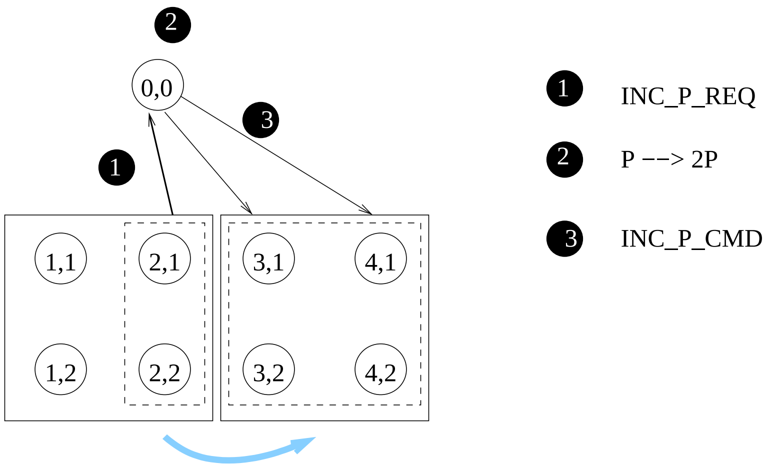


Figure 3.5: Partition expansion example. The LBM initially has 2 partitions and 1 replica. Partition 2 is the ER. After expansion, the matrix has 4 partitions, and the last two partitions becomes the new ER, annotated using a dotted box.

It is important to explain two design choices in the above algorithm. First, the reason that the head node ignores requests from non-ER partitions and other ER partitions that may arrive later is to avoid unnecessary expansion. Ideally, a matrix should expand only if all partitions in the matrix reach their threshold, i.e., are saturated. We use the first `INC_P_REQ` from an ER partition as the signal that a matrix is near saturation. Requests from non-ER partitions are not representative since, although the load on them may have reached the threshold, the load on ER partitions may still be below the threshold. This is especially true when new partitions are first added. The head node acts upon only one request from the ER partitions and suppresses others, because it needs to wait until the new partitions are being used.

Second, we increase the number of partitions in an overloaded matrix aggressively by doubling P_i every time. The reason is that the cost of having a matrix that is too small is significant, since registrations will be rejected. On the other hand, the cost of having a matrix that is temporarily larger than necessary is modest, in that it does not add extra registration cost (no data replication over the network involved), and only causes a linear increase of the query overhead (each partition must be visited). However, as we will explain next, the matrix will reduce its size through shrinking if it is too large for the corresponding registration load, thus minimizing the query overhead.

3.2.2 Partition Shrinking

A matrix should decreases the number of partitions if its registration load becomes low, since more partitions means more query messages are needed for a query that is sent to this matrix. Unlike in the expansion mechanism, where a node may issue an expansion request when it experiences high load, shrinking is done periodically.

Suppose LBM_i now has P_i' partitions, and before the last expansion, it had P_i partitions ($P_i' = 2P_i$). P_i is also equal to the number of partitions in the ER immediately after the

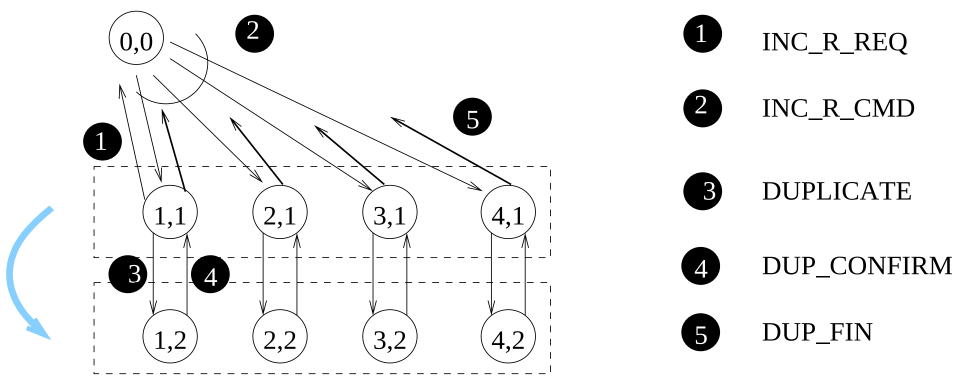


Figure 3.6: Replication expansion example. The LBM initially has 4 partitions and 1 replica, which is also the ER. The LBM establishes 2 replicas after the expansion.

last expansion. The following steps are involved in partition shrinking.

- Each node in the *last partition* of the matrix checks its average registration rate and the number of content names it has every T seconds, if the registration rate or the number of names drops below a low threshold, it will issue a `DEC_P_REQ` request to the head node. The low threshold is typically a small fraction of T_{reg} or T_{CN}.
- The head node again acts on only one such request: it sends a `SHRINK_P` command to all the nodes in the last partition and asks them to transfer their names containing $\{a_i v_i\}$ to the nodes in partition $P_i' - P_i$. For example, $N_i^{(P_i',1)}$ sends its names to $N_i^{(P_i'-P_i,1)}$.
- After all the transfers are confirmed, the head node will inform nodes in partition P_i' that they have been removed from the matrix. Now partition $P_i' - 1$ becomes the last partition, and the head node decreases the size by 1, $P_i' \leftarrow P_i' - 1$. Correspondingly, the size of the current ER is also reduced by 1.

When all the partitions in the current ER are removed, and the number of partitions drop back to P_i, the head node informs the partitions from $\lceil \frac{P_i}{2} \rceil$ through P_i that they now belong to the new ER. By collapsing the matrix one partition at a time, we try to keep the matrix load balanced, and the linear decrease coupled with the multiplicative increase prevents the matrix size from oscillating.

3.2.3 Replication Expansion

New replicas are added to the matrix when the query load of the matrix increases, similar to how partitions are added. The expansion region here refers to the replicas that were last added. The replication expansion is done as follows; Figure 3.6 is an example showing the steps involved.

- When a node in the ER observes its query rate reaches T_q, it sends an `INC_R_REQ` request to the head node (Step ①in Figure 3.6).

- Upon receiving such a message, the head node issues an `INC_R_CMD` command to each node in the last row of the ER, asking them to replicate themselves (Step ②). We call these nodes the replica initiating nodes.

- The replication is also done multiplicatively to allow the matrix expand to a large size to accommodate query load. A node that receives the `INC_R_CMD` message sends a copy of the names corresponding to $\{a_i v_i\}$ in its name database to the newly added nodes in its column. For example, node $N_i^{(1,R_i)}$ will send its names to nodes $N_i^{(1,R_i+1)}$ through $N_i^{(1,2R_i)}$ (Step ③). The new nodes then send a confirmation message back to the initiating nodes after they receive the names (Step ④).

- Once confirmed, the replica initiating nodes send `DUP_FIN` messages back to the head node to indicate all the replicas are in place (Step ⑤). The head node then doubles R_i. Correspondingly, the nodes in row $R_i + 1$ to row $2R_i$ become the new ER.

Once new replicas are added to the matrix, future queries that contain $\{a_i v_i\}$ will be served by the expanded matrix.

3.2.4 Replication Shrinking

More replicas in the matrix means providers must register with more nodes. Thus matrices should shrink along the R dimension when the query load to this matrix drops. The replication shrinking mechanism is similar to the partition shrinking mechanism, but no data transfer is needed. It is done periodically as well. When any node in the *last row* observes a low query rate, it will issue a `DEC_R_REQ` request to the head node. When it receives such a request, the head node will send a `SHRINK_R` command to all the nodes in the last row so that they can remove themselves from the matrix. The head node then informs nodes in the new last row that they now become the last row, and finally decreases the number of replicas by 1.

The shrinking mechanism is important specially under a "flash-crowd" type of load: when an AV-pair becomes popular due to, for example, a current event, its corresponding matrix will replicate quickly to accommodate the sudden surge in load. When clients lose interest in this pair, the matrix will shrink and eventually may become just one row.

3.2.5 Head Node Mechanism and LBM Maintenance

The primary job of the head node is to coordinate the matrix expansion and shrinking. The expansion and shrinking requests may come to the head node in an arbitrary order. While a matrix is in a dynamic state, i.e., expanding or shrinking, if the corresponding head node receives additional requests, it will buffer these requests and process them when the current operation completes. By serializing the operations, we ensure data consistency within the matrix.

A head node is only responsible for its own matrix, and different matrices will likely have different head nodes. Therefore, head nodes should not become the bottleneck of the system. However, when a head node leaves or crashes, vital information about its matrix, such as the size, will be lost. To prevent this from happening, live nodes in the matrix send infrequent periodical messages with their indices (p, r) to the head node. Due to the routing properties of DHT, a new node whose ID is close to the old head node's ID will receive these messages and become the new head node. It can then recover the matrix's size based on the information it receives. In addition, the matrix expansion or shrinking requests will also reach the new head node, and they can be used to recover the matrix's dimensions as well.

3.3 System Properties with LBM

3.3.1 Registration and Query Efficiency

When LBMs are deployed in the system, both the registration and query cost will increase compared to the basic system. To register a name that has n pairs, the number of registration messages needed is:

$$M_r = \sum_{i=1}^{n} R_i, \tag{3.2}$$

where R_i is the number of replicas in matrix LBM_i. The basic RP-based system presented in Chapter 2 is a special case: $\forall i$, $R_i = 1$, since LBM_i has no extra replicas, and $M_r = n$. M_r is determined by the number of replicas each matrix has, and does not depend on the number of partitions.

To resolve a query that has m pairs, when using the query optimization mechanism, the number of query messages needed excluding the probing messages, is

$$M_q = min(P_i), \tag{3.3}$$

where $i = 1..m$, and P_i is the number of partitions in matrix LBM_i. The query cost is not affected by the number of replicas in these matrices, but depends solely on the number of partitions.

3.3.2 LBM Maintenance Cost

As we mentioned above, to ensure that an LBM can work properly, even if its current head node leaves/crashes, nodes in the matrix must periodically send beacon messages to the head node. Therefore the maintenance cost of an LBM is proportional to the total number of nodes in the matrix, which equals the product of its number of columns (P) and rows (R). What this implies is that to minimize the matrix maintenance cost, it is important to ensure the matrix does not have a large P and R at the same time.

In addition to minimizing the cost of resolving a query, the query optimization algorithm described in Section 3.1.2 helps to achieve this goal in our system. For matrices that have many partitions, i.e., with a large P, the query optimization algorithm will ensure that queries avoid them whenever possible. By doing so, the query load on these matrices is reduced; thus it naturally limits the R-dimension expansion of these matrices. In other words, a matrix with a large P typically has a small R. On the other hand, matrices that have fewer partitions, i.e., with a small P, may be visited often by queries, also due to the query optimization algorithm. This in turn may cause these matrices to replicate often and have a large R. From the system's point of view, it will not greatly affect the average number of registration messages needed: small P implies only a small number of content names will be affected by having to register with more replicas.

3.3.3 CDS and Churn

It has been suggested [16] that a DHT's performance may degrade significantly in the face of high node join and departure rate, also known as node churn rate. The CDS is layered on top of a DHT, and therefore the node churn rate inevitably affects the performance of our system. Here we discuss the relationship between DHT churn and CDS performance.

First, the applications that the CDS is targeting are primarily infrastructure services such as distributed monitoring. These applications typically have a fairly stable core overlay network, and the churn rate is low. As such, the concern of churn on CDS is not a significant issue.

Second, in the CDS, content provider must periodically refresh their registrations, and as such, in the event of a node departure, contents will still be searchable, albeit the node hosting the same content is different. However, we must note that if the churn rate is higher than the content refresh rate, contents maintained by a node will be lost after the node leaves the system and before the next refresh time. In other words, the performance of the CDS is upper-bounded by the churn rate. It is important to point out that throughout our design in this thesis, we use soft state protocols (e.g., the tree-based protocols in Chapter 4 and 5). By maintaining soft states, we make sure that the system is resilient under dynamic node join and leave scenarios.

Third, recent work [57] shows that by adding network and system level optimizations, DHTs can perform well even under high churn rate. Our conscious decision of separating the CDS functions from the DHT layer makes it possible for the CDS to take advantage of any DHT design improvements.

3.4 Evaluation Methodology

We implemented the CDS system in an event-driven simulator. The software package consists of about 15,000 lines of C code. The simulator allows us to conduct comprehensive evaluation of the system. In this section, we describe the simulator and our evaluation methodology.

3.4.1 Simulator Implementation

The event-driven simulation technique [23] has been used in many large scale network simulation tools, such as *ns-2* [2] and the Chord simulator [52]. Since our focus is on the effectiveness of the CDS mechanisms, we find it more convenient to write our own simulator than using existing simulators that were designed for other purposes, such as evaluating lower level network protocols.

The CDS simulator program's structure is similar to the Chord simulator [52]. It maintains a global event queue and a global variable *current_time*. Each event in the simulator is associated with a node and a time stamp that indicates when this event must be handled. The types of events include all the messages we discussed earlier, e.g., registration, query, and matrix management messages. Each node processes registrations and queries with exponentially distributed service rates. Events are inserted into the queue according to the time stamp. The simulator sits in a loop, removes events from the queue in the current time slot according to the time they occur, and then processes them accordingly. The processing of an event may further generate new events, and the new events will subsequently be inserted into the queue. The program then advances the *current_time* variable, and goes back to the beginning of the loop. It stops if a specified termination time is reached, or there are no events left on the queue to be processed.

The key data structure in the simulator is the *node* data structure, which represents a node in the CDS system. The simulator allows one to create a CDS system with configurable number of nodes, N_c, to form a DHT-based overlay network. What the program does is to create N_c instances of the *node* structure.

The simulator assigns node IDs uniformly in that each node occupies an equal segment in the key identifier space. In practice, this may be achieved by using techniques such as assigning multiple "virtual" node IDs to one node [64]. The simulator uses an exponential distribution with a mean value of 50 ms [64] to model the one-way network delay between any two nodes. In DHT systems such as Pastry [58], by employing proximity metric into the routing rules, the overlay delay between two nodes can be limited to within 1.4 times of the physical network delay. In our simulation, we conservatively set the average overlay delay between two overlay nodes to be twice the physical network delay between them, which results in a mean of 100 ms. It is worth pointing out that the simulator does not implement DHT level functionalities. As such, messages are not "routed" through the network. Instead, once the source and destination node IDs are determined, the simulator simply computes a delay that it will take for the message to get to the destination.

The hash function used by the CDS system must generate values uniformly distributed in the name space and be insensitive to the input. In our implementation, we use the cryptographic function SHA-1 [4] as the system-wide hash function, and a 24-bit name space.

To simulate the application that is running on top of the CDS, a node may be assigned a set of content names and queries. This is typically done by reading data from a load file. We will discuss the application load shortly. To implement the CDS layer functionality, each node maintains a local content name database and resolves queries. For load balancing purpose, each node also maintains statistics such as the registration and query rate that it observes. The rates are measured using a sliding window of a certain number of registrations or queries over the last measuring time interval. It conducts load balancing related actions when thresholds are crossed.

We use a simple registration example to explain how the simulator works. For example, suppose the input load file specifies that Node 1 registers content name $\{a_1 = v_1; a_2 = v_2\}$ at time 100ms. This is translated into a registration event associated with Node 1; its time stamp is 100 and it will then be inserted to the event queue. This event will be processed when *current_time* becomes 100. Suppose the hash of $\{a_1 = v_1\}$ corresponds to Node 235,

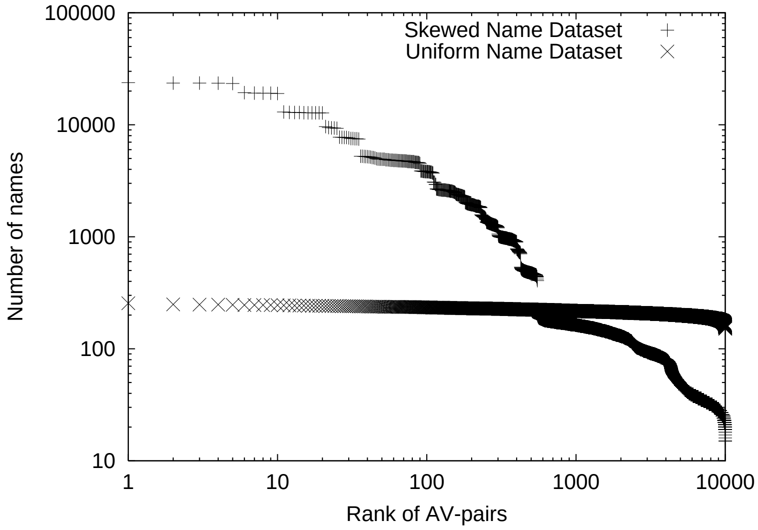


Figure 3.7: AV-pair distribution in two sets of content names.

and suppose it takes 100ms for a message to reach Node 235 from Node 1, then a new registration event will be associated with Node 235 with a time stamp of 200. This event will then be added to the global event queue. Subsequently, it will be processed when *current_time* reaches 200ms. Other CDS events are created and processed similarly.

3.4.2 Experimental Setup

To run a simulation experiment, we must provide two input files to the simulator: the configuration file and the workload file.

Configuration

The configuration file contains parameters that we need to set up an experiment. In the experiments conducted in this chapter, we use the following configuration.

The CDS network consists of 10,000 (N_c) nodes. We specify that each node has approximately $500Kbps$ available link bandwidth (DSL level) dedicated to content name registrations and queries. Corresponding to this bandwidth, assuming a 1000-byte registration packets size and a 250-byte query packet size, each node sets up a threshold of $T_{reg} = 50reg/sec$ as the maximum sustainable registration rate and $T_q = 200q/sec$ as the maximum sustainable query rate. The maximum number of content names a node is willing to receive, T_{CN}, is set to be 4000.

The processing of registrations and queries on a node is exponentially distributed with a mean rate of $1000reg(query)/sec$, which can easily be achieved by modern PCs on a database with a size on the order of 10^5 entries. With these assumptions, a node's performance is limited by its available link bandwidth.

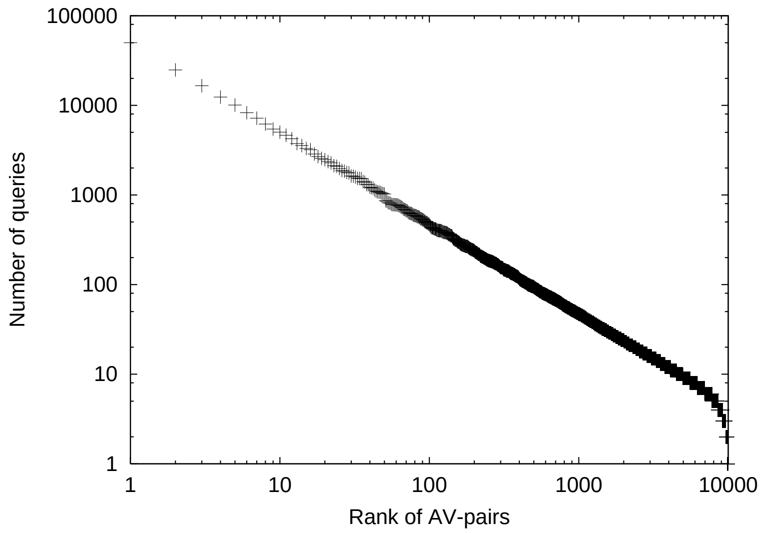


Figure 3.8: AV-pair distribution in queries.

Workload

We use both synthetic workload generated based on realistic distributions and a realistic load obtained from a large collection of mp3 music files to drive the simulations. We now describe the workloads in more detail.

- **Synthetic Workload**

  We generate two sets of content names for registration and one set of queries as workloads. There exist 50 attributes in the workloads, each of which can take on 200 values; this results in 10,000 (N_d) distinct AV-pairs. As such, each node on average is responsible for 1 ($= N_d/N_c$) AV-pair.

  Each content name dataset contains $100,000$ names and each name is comprised of $n = 20$ AV-pairs. The AV-pair distributions in names are shown in Figure 3.7. In the uniform dataset, each AV-pair is equally likely to appear in a name, and on average each AV-pair occurs in about 200 names. The uniform dataset is primarily used for comparison. In the skewed case, some AV-pairs are assigned higher weights, and the overall distribution of AV-pairs is Zipf-like, as it is close to a straight line in the log-log plot($\alpha \sim 0.9$). The top 5 most popular AV-pairs are contained in about 24,000 names. The query dataset (Figure 3.8) contains $99,473$ queries and is generated based on a Zipf distribution with $k = 0.5$ and $\alpha = 1$ in Equation 2.2. The number of AV-pairs in a query ranges from 1 to 10, and on average each query consists of 4 AV-pairs. The most popular AV-pair occurs in about 50,000 queries.

  The sender of a name or a query is selected randomly from the nodes in the system and both the arrival times for names and queries are modeled with a Poisson distribution.

- **Music Workload**

Our realistic workload is based on a collection of 5,000 mp3 files representing a variety of genres and styles, as we descried in Section 2.5.1. For each file, we use 30 feature attributes to represent its musical content. We use a standard linear quantization and normalization to transform the dynamic ranges of the continuous features into discrete values necessary for searching based on AV-pairs. Linear quantization was chosen so that the statistics of the distribution of the features do not change. In our system, each feature is quantized to 100 discrete values. Experiments comparing automatic classification of the original features and the quantized features showed no significant differences. The AV-pair distribution in these files is plotted in Figure 2.9, and it is fairly skewed. To test the system's scalability with respect to registrations, we generated 100,000 content names by replicating each of the 5,000 files 20 times, and assigned them to random nodes.

For query load, 100,000 queries were generated following a Zipf distribution independent from the above distribution. Each query corresponds to the features of one music file. We do so to emulate the behavior of a user who submits a music clip and looks for similar music. The most popular content name occurs in over 10% of the queries, and the majority of the MFDs only occur in a few queries. A query's initiator is randomly picked from all nodes, and for simplicity, only exact matches are returned.

3.4.3 Performance Metrics

Once the configuration and load files are read in, and the simulation starts, a node registers a name by sending registration messages to the RP nodes corresponding to the name's AV-pairs concurrently. The registration succeeds when *all* the pairs registered successfully. Upon receiving a registration message, the RP node either inserts the name into its local database and replies the registering node with a success, or rejects the name and replies with a failure. A registration may fail at a node for two reasons: (1) the registration rate this node observes, r_{node}, exceeds the set threshold, i.e., $r_{node} > T_{reg}$, or the number of names it is hosting exceeds T_{CN}; (2) the corresponding matrix is in a dynamic state such as expanding. For instance, a node has sent a replica to a new node, but the success of the replication has not been confirmed, and during this time period, any registration arriving at the replicating node will be rejected.

Similarly, a query is sent to one RP node in each partition of the chosen LBM concurrently. The RP node rejects the query if the query rate this node observes, q_{node}, exceeds the set query rate threshold, i.e., $q_{node} > T_q$, by replying to the query node with a failure message. Otherwise, it accepts the query, examines its database and sends the querying node the set of content names that match the query. Note that the set may be empty. From the querying node's point of view, a query succeeds when *all* the corresponding RP nodes accept the query.

We evaluate the CDS system using the following metrics: the *registration* and *query success rates* and the *registration* and *query response time*. The success rate is defined as the percentage of successful registrations or queries in one simulation run. Since the *system throughput* equals the product of the *system load* (registration or query rate) and the *success rate*, the success rate is used as an indicator of the system's throughput: the throughput increases as load increases, if the success rate remains high. For a successful registration

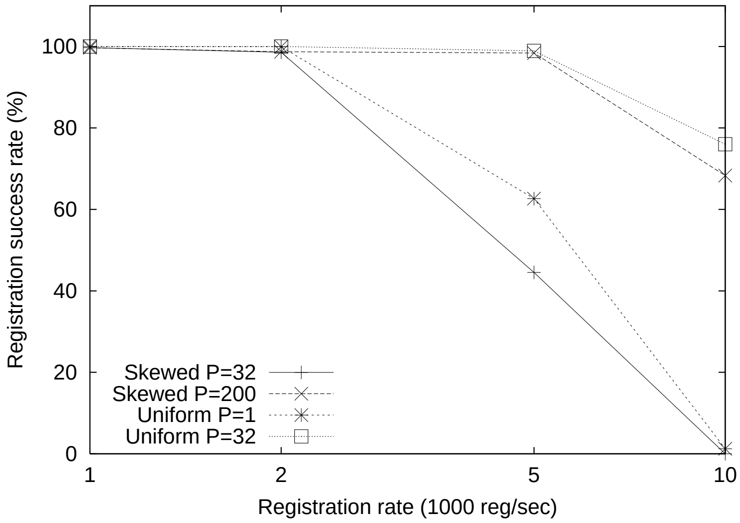


Figure 3.9: Registration success rate comparison.

or query, we define the response time as the time between when the registration or query messages (probe messages, when we must probe the matrix size) are first sent and when the last reply message is received.

3.5 Simulation Results

We conducted extensive simulations to validate the design of the CDS system. In particular, we present the following results to verify the system's properties.

- We show the load balancing mechanism can ensure high system throughput under skewed load in Section 3.5.1 and Section 3.5.2.
- We study the load distribution from the system's point of view by examining load received on each node in Section 3.5.3.
- We examine how the endpoint registration and query performance is affected due to load balancing in Section 3.5.4. We show that the endpoints's performance under skewed load is comparable to a system where load is uniformly distributed.
- Finally, we demonstrate the system's dynamic behavior under a flash crowd type of load, and show the load balancing mechanism in action in Section 3.5.5.

3.5.1 Registration Success Rate

We first examine how the system behaves as the registration rate increases. For each experiment, we inject either the skewed dataset or the uniform dataset into the system with a certain arrival rate r_{system}. Each experiment is carried out with a different P value, the maximum number of partitions a matrix may use.

Figure 3.9 compares the success rate in these experiments after all the matrices stop expanding. We observe that for a given P value, when the registration rate is low, the registrations succeed on the existing set of partitions, and the success rate is 100%. As the load increases, the success rate starts to drop, because without further expanding, nodes in the matrices become saturated and start to reject registrations. By increasing P, for the same registration load, the success rate is improved significantly. As the load is further increased, all curves eventually drop.

For the uniform dataset, since AV-pairs are distributed evenly in content names, the registration load is distributed fairly evenly among nodes in the system. Compared with the skewed load, fewer partitions are needed for the same registration load to maintain the same success rate. The basic system as we described in Chapter 2 ($P = 1$) performs well until r_{system} reaches $2000reg/sec$, but after that the success rate drops quickly to nearly 0%. The reason is that the hash function may map multiple AV-pairs onto the same node, and when r_{system} increases, the registration rate on such nodes will reach T_{reg} earlier than others, and cause registration failures.

We study the data points corresponding to the highest registration load, where $r_{system} = 10^4 reg/sec$. In these experiments, since there are no queries, and thus no replications in the system, each name is registered at $n = 20$ nodes, the average registration rate observed on a node is:

$$r_{node} = \frac{r_{system} \cdot n}{N_c} = 20reg/sec.$$

The success rate under this registration load is 76% for the uniform load when $P = 32$, and 68% for the skewed load when $P = 200$. What it means is that on average each node in the system operates at 40% of its link capacity while maintaining a fairly high success rate. These experiments show that the system can be scaled to near its capacity even for skewed load: the load balancing mechanism effectively spreads the excessive load to underutilized nodes in the system.

Effectiveness of Partitions

From the above experiments, we know that for a given load, once enough partitions are added to each matrix, the system can achieve high success rate. In this section, we study more carefully how the success rate is improved as more partitions are added, and what factors contribute to the failure of a registration. We classify registration failures into four categories:

1. *Capacity failure.* Failures due to not having enough partitions allocated to a matrix to accommodate a pair's registration load.

2. *Compulsory failure.* In the simulation, it takes one RTT to add new partitions to a matrix, and registrations arriving during that time period are rejected.

3. *Conflict failure.* Since multiple AV-pairs may be mapped onto one node, a registration may fail at a node because some other pair introduces high registration load there.

4. *Statistical failure.* Failures due to statistical variations, e.g., failures caused by bursty arrival of registrations of the same pair on one node.

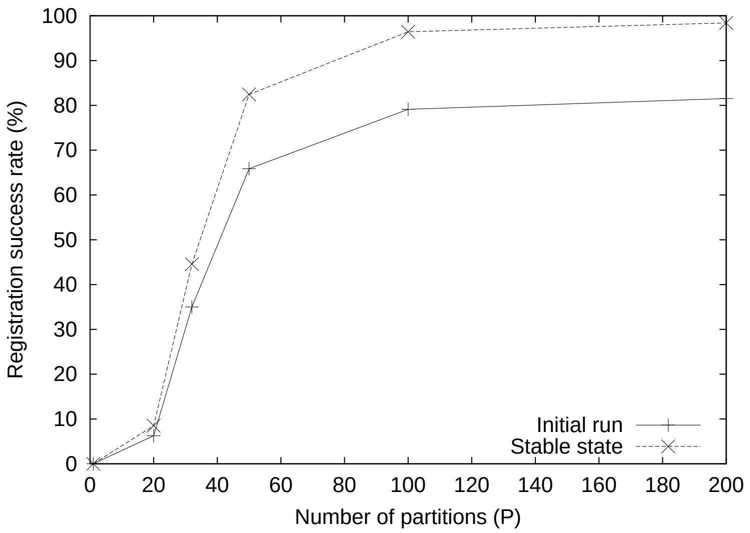


Figure 3.10: Effect of number of partitions. Skewed dataset with $r_{system} = 5000reg/sec$.

In the experiments conducted here, we inject the skewed content name dataset into the system with an arrival rate of $r_{system} = 5000reg/sec$. We vary the configured P value in each experiment, and for each P we run the experiment twice: (1) during the *initial run*, as names arrive, partitions are created when needed, and (2) the same dataset is sent to the system again in the *stable state*, when all the partitions have been created, and no new partitions are needed. The number of partitions needed, P_i, for a pair $\{a_i v_i\}$ under a certain registration rate can be analytically computed as follows:

$$P_i = \frac{r_{a_i v_i}}{T_{reg}} = \frac{r_{system} \cdot p_i}{T_{reg}} \tag{3.4}$$

where $r_{a_i v_i}$ is the arrival rate of $\{a_i v_i\}$, and p_i is the pair's probability of occurring in names. In the skewed name dataset, for the top 5 most popular pairs, $p_i = 0.24$. With $r_{system} = 5000reg/sec$ and $T_{reg} = 50reg/sec$, from Equation 3.4, we know to accommodate names that contain these pairs, each of the corresponding matrices needs at least $P = 24$ partitions.

Figure 3.10 shows the registration success rate under different P values. When $P \leq 20$, the success rate is very low primarily due to the large number of capacity and conflict failures caused by the popular pairs. In particular, $P = 1$ corresponds to our basic system and the poor performance shows that using one RP node for each AV-pair can not handle highly skewed load. When $P = 32$, the success rate is still below 50% though seemingly there should be enough partitions. The failures come mainly from conflicts: since we have 10,000 distinct AV-pairs and 10,000 nodes, it is possible that two AV-pairs are mapped onto the same node. As the system allows more partitions to be used by a matrix, the conflict failures are overcome and the success rate increases significantly. The reason is that when a node observes high registration load caused by two different pairs, it will prompt the expansion of both of their corresponding matrices (at different times), thus reducing the load observed by partitions within each of the two matrices.

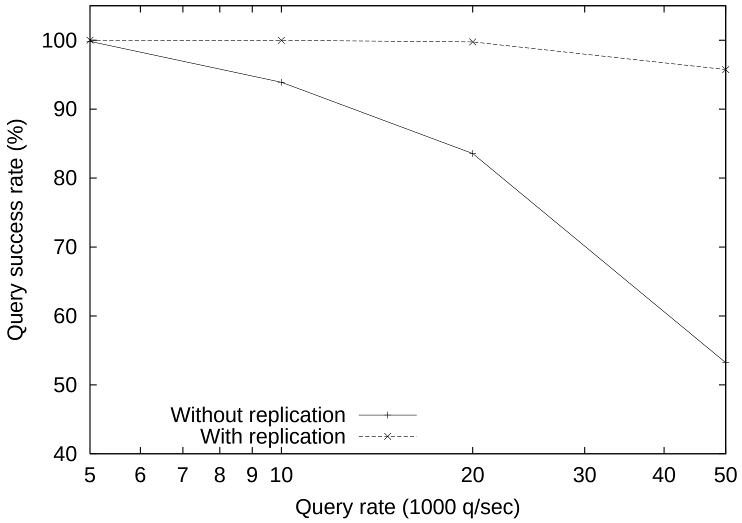


Figure 3.11: Query success rate comparison.

The gap between the initial curve and the stable curve represents the percentage of compulsory failures. Note here that for evaluation purpose, a node issues an expansion request only after its threshold is crossed, and at that time it can not accommodate any more registrations. In practice, a node should react before its absolute threshold is reached, e.g., when the observed rate is $0.5T_{reg}$, and thus it can continue to accept registrations while the matrix is expanding. In fact, this is what we do in our prototype implementation to be presented in Chapter 6. When enough partitions are allowed, the success rates in the stable run are substantially higher than those in the initial run since there are no compulsory failures. We observe that the success rates stay above 95% for $P > 100$ under this load.

In summary, by expanding the matrix along the P dimension, the system can successfully recruit lightly loaded nodes to share concentrated registration load, thus increasing registrations' success rate. In practice, if a registration fails at a node due to the concentration of load, the sender of the registration will try to register it again at a later time, and by then, the matrix should have expanded and be able to accommodate the registration.

3.5.2 Query Success Rate

We evaluate the system's performance for queries using both the music workload and the synthetic workload. We evaluate the effect of both the dynamic replication (matrix R dimension expansion), and the query optimization mechanisms in improving the system's query success rate.

Evaluation Using Music Workload

In the experiments carried out here, we use the Music workload as described in Section 3.4.2. We first inject the registration load into the network, and then inject the query load. Due to the skewed feature distribution, registrations of common AV-pairs result in multiple partitions.

Figure 3.11 compares the query success rate as a function of query arrival rate under two scenarios. In the first scenario, "Without replication", when reaching its query rate threshold, a node simply rejects new queries arriving at it without replicating its content at other nodes. This is how it is done in the basic system. Since for each query the CDS has 30 candidate AV-pairs, the query load is spread well among nodes even without replication. Therefore, the system can achieve a high success rate under fairly high load. For example, the success rate is 94% for a query rate of $10,000q/sec$. However, as the load increases further, nodes corresponding to popular queries will be saturated, and the success rate drops quickly. In the second scenario, "With replication", we enable the dynamic replication mechanism, so nodes that observe high query load will replicate their databases at other nodes to dissipate the concentrated query load. As a result, we observe that with replication, the system can sustain a much higher query rate while keeping the success rate above 95%.

Evaluation Using Synthetic Workload

When using the Music Workload, the system works fairly well even without replication, since the query has many (30) AV-pairs for the system to choose from and to spread the load. However, in many applications, queries may only contain a few AV-pairs. In this section, we study how the system scales using the synthetic load, in which the queries contain 4 AV-pairs on average.

In the following experiments, we first inject into the system the skewed name dataset with $r_{system} = 2000reg/sec$, and then issue the Zipf queries with different arrival rate q_{system}. With this registration load, the matrices corresponding to the most popular AV-pairs have 32 partitions. We compare the following three scenarios:

1. **No replication**. This is the basic design. A query is sent to a matrix corresponding to a random pair in the query. The matrices do not replicate when the query load they observe exceeds the query threshed.

2. **With replication but no query optimization**, or **Random**. A query is sent to a matrix that corresponds to a random pair in the query. Matrices will replicate when the query load they observe exceeds threshold T_Q.

3. **With replication and optimization**. Use the query optimization technique and send a query to the matrix that has the fewest partitions. A matrix may also replicate if it observes high query load.

Figure 3.12 shows the query success rates. Without replication, by selecting a random pair for each query, the system tries to spread load to different matrices, and the success rate is high (97%) when the query rate is fairly low ($1000q/sec$). The success rate drops quickly as query load increases. This is due to the skewness in the queries: the matrices corresponding to the popular pairs in queries are quickly overloaded, and without replication, many queries will fail. When compared with the results with the Music workload, the success rate is lower for the same query rate due to fewer choices within a query.

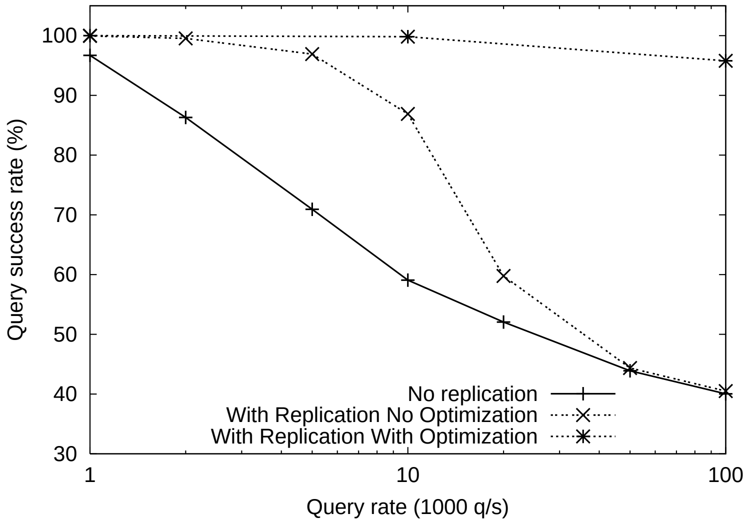


Figure 3.12: Query success rate comparison.

In the second scenario, we allow matrices to replicate under high query load. For the same query load, the success rate is increased significantly: it stays around 90% for rates as high as $10,000q/sec$. However, when q_{system} is further increased, the success rate starts to drop sharply. The reason is that since popular AV-pairs appear in many queries, and each query contains only a few pairs, it is possible that many queries select the same AV-pair and are sent to the same matrix, which will cause compulsory failures and the replication of these matrices.

In addition, in our workload, the pairs that are popular in queries are also common in registrations, which means their corresponding matrices have many partitions. The time it takes to replicate a large matrix is high, since the new replicas can be used only after all the partitions replicate successfully. Queries arriving during the replication time period are likely to be rejected, since they must be sent to the existing replicas, which have already being saturated. This phenomenon is displayed most clearly when the arrival rate is extremely high ($q_{system} > 50,000q/sec$), where all the queries arrive before a matrix can complete two rounds of replications. As a result, the performance becomes the same as the scheme without replication. Similar to the observation we made in the previous section, in a real system, a node should also start the replication expansion action before its query threshold is reached, so it can minimize this type of query failures.

In the third scenario, the query optimization mechanism successfully spreads query load to matrices with few partitions. This is especially important for high query loads, where using the load balancing mechanism alone is not effective. Figure 3.12 shows that even under the highest load, $q_{system} = 10^5q/sec$, avoiding large matrices and thus long replication time allows the system to maintain a query success rate of above 95%. Most matrices do not need to replicate at all, and the maximum number of replicas in a matrix is 4.

In summary, the basic system without load balancing works well if the query load is low and queries contain many AV-pairs. The system's performance is significantly improved for skewed load via dynamic replication. The query optimization mechanism further increases

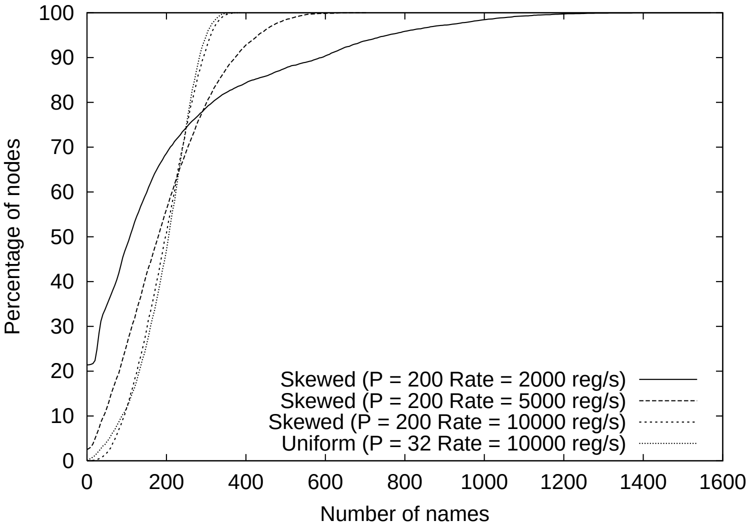


Figure 3.13: Comparison of the Cumulative Distribution Function of the number of registered names on nodes.

the system's scalability for very high query rate by avoiding matrices with many partitions.

3.5.3 System Load Distribution

In this section, we evaluate the system's load balancing property by examining the name distribution and observed load on RP nodes. In particular, we report registration load distribution and the query load distribution is similar.

Content Name Distribution

After each experiment conducted in Figure 3.9, we tally the total number of names each node received. Figure 3.13 shows the Cumulative Distribution Function (CDF) of the number of registered names on nodes. The first three curves correspond to the experiments where the skewed dataset is used with $P = 200$. The fourth curve is from the uniform dataset with $P = 32$. The registration rate in each experiment is fixed. Since the success rates are different in these experiments, for comparison purpose, we normalize the number of successfully registered names in each experiment to 10^5. Hence on average each node should receive 200 names.

Since T_{reg} is always reached before $T_{CN}(= 4000)$ in our experiments, matrix expansions are therefore caused by the high registration rates observed on nodes, and not the high number of names. At the end of each experiment, the average registration rate on each node can be simply computed by dividing the final number of names it has by the simulation time. Thus we use the number of names to represent the registration load on a node.

With low registration rate, the system can accommodate the registrations successfully using a small number of partitions for each matrix, which means many nodes in the system may receive none or a small number of names. For example, when $r_{system} = 2000 reg/sec$, 21% of nodes receive no registrations. In the mean time, some nodes in the system accumulate large number of names, as exhibited by the long tail in the distribution. Note the maximum number of names on a node is still less than T_{CN}. As registration rate increases, names are spread to more nodes due to the expansion of matrices. In Figure 3.13, when $r_{system} = 10^4 reg/sec$, we observe that the CDF grows very quickly and no nodes receive more than twice of the average number of names. A distribution that is "more vertical" represents a more load balanced system.

More quantitatively, we use the metric *Coefficient of Variance (CV)* [67] to evaluate the load balancing property. In our context, CV is defined as:

$$CV\left[n_i\right] = \frac{\sigma\left[n_i\right]}{E\left[n_i\right]} \tag{3.5}$$

where $i = 1..N_c$, n_i is the number of names node N_i holds, and σ and E are the standard deviation and mean of n_i. A smaller CV indicates a more load balanced system. As load increases, the load balancing mechanism successfully balances load across all nodes across the system. The CV decreases from 1.242 to 0.369 as r_{system} increases from $2000 reg/sec$ to $10^4 reg/sec$. As a reference, when $r_{system} = 10^4 reg/sec$, the CV in the skewed load case matches the $CV (= 0.366)$ in the uniform load case.

This set of results demonstrate that the load balancing mechanisms ensure that the load is distributed evenly under high and skewed load by shifting load from overloaded nodes to lightly loaded nodes in the system.

Observed Load on RP Nodes

We now take a closer look at the load distribution within different partitions of a matrix. Figure 3.14 shows the observed registration rate as time progresses in three different partitions of a matrix that corresponds to one of the most popular AV-pairs. In this experiment, the skewed name set with $r_{system} = 2000 reg/sec$ is used. Initially there is only one partition in the system, and it receives the entire registration load corresponding to this pair. The maximum observed registration rate on Partition 1 approaches $450 reg/sec$. As partitions are added to the matrix to share the registration load, the rate observed by the first partition begins to drop quickly, as shown in the figure. The 16th and 32nd partitions are introduced around time 2000 ms and 3700 ms respectively. Once all the partitions are in place, as expected, the load on each partition stays under the set threshold of $T_{reg} = 50 reg/sec$. In fact, since the load is shared by 32 partitions, each node observes about $15 reg/sec$.

The results here verify that the load within a matrix is indeed distributed evenly. This property makes sure that the load observed on an individual node is representative of the load observed by the matrix. This is the fundamental reason that we can conduct load balancing in a distributed fashion, based solely on local information.

3.5.4 Registration and Query Cost

In this section, we evaluate the system from the viewpoint of a content provider or query issuer. In particular, we examine the response times, and the number of messages needed for registrations and queries. In this set of experiments, registrations and queries arrive simultaneously with the arrival rates of $r_{system} = 1000 reg/sec$ and $q_{system} = 5000 q/sec$.

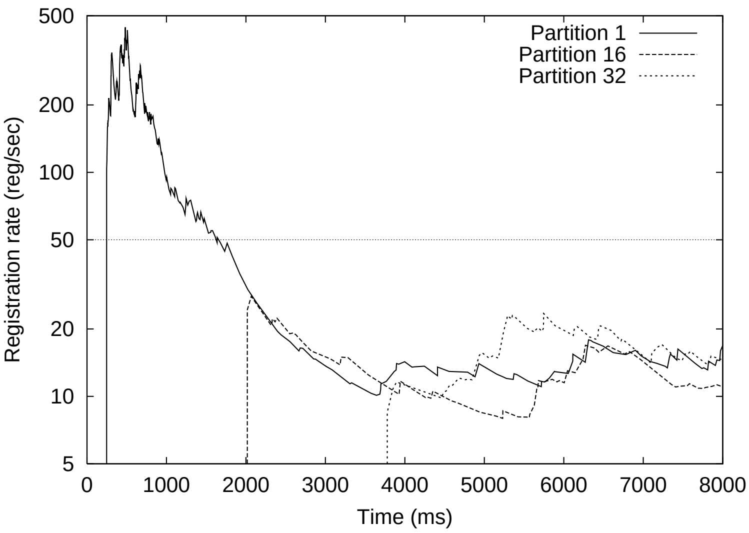


Figure 3.14: Load within a matrix. Skewed dataset with $r_{system} = 2000reg/sec$.

The workload consists of about 17,000 skewed names and 83,000 Zipf queries. Instead of devoting its full bandwidth to serve either registrations or queries, each node allocates 50% of the bandwidth to queries and 50% to registrations. Correspondingly, the thresholds are set as follows: $T_{reg} = 25reg/sec$ and $T_q = 100q/sec$. All simulations use load balancing mechanisms, and we consider two scenarios: **Random** and **With Query Optimization**.

Matrix Size Distribution

As discussed in Section 3.3, the sizes of the load balancing matrices determine the cost of registrations and queries. We first examine the matrix size distribution after each experiment. Figure 3.15 is a 3-D plot of the distribution of the matrix sizes after each simulation run. Each bar represents the number of matrices that have that particular size (P, R). Since there are 10,000 distinct AV-pairs in the system, there are 10,000 LBMs in total. All the results fall on the vertical planes that correspond to powers of two, because the dimensions are increased multiplicatively and there is no matrix shrinking in the experiments.

In the random scenario, 89.2% of the matrices have 1 partition and 1 replica, $(P = 1, R = 1)$. As we discussed earlier, matrices that have large P may still get many queries, which means they must replicate themselves frequently. Figure 3.15 confirms our analysis in that some matrices with large P, e.g., $P = 32$, also have a large R. The largest matrix has a size of $(P = 32, R = 32)$.

With query optimization, more matrices (94.3%) have the minimal size, $(P = 1, R = 1)$. The maximum number of replicas in a matrix 4 as opposed to 32 in the random scenario. It is worth noting that the matrix that has 4 replicas also has 32 partitions. The explanation is that the AV-pair corresponding to this matrix is also popular in queries. In particular, it appears frequently by itself in queries, which makes query optimization not applicable and replication necessary.

The results shown here confirmed our analysis about the importance of the query opti-

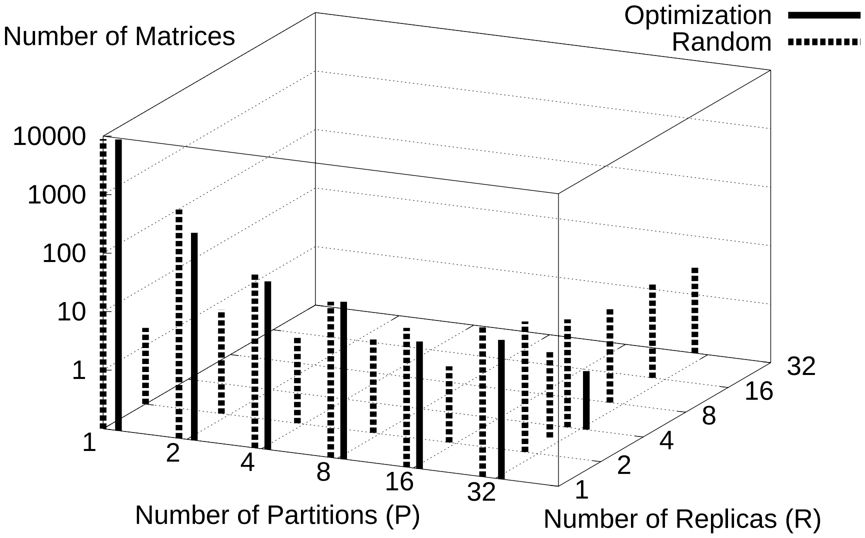


Figure 3.15: Matrix size distribution. All the axes are in logarithmic scale.

mization mechanism in Section 3.3. A matrix may have many partitions (large P) due to the amount of contents containing the corresponding AV-pair. But the query optimization algorithm tries to avoid these matrices when issue queries to the system, and therefore avoids the high replication cost of them. The matrix size distribution in Figure 3.15 are close to the $R = 1$ plane under query optimization.

Registration Cost and Response Time

Figure 3.16 shows the CDFs of the number of registration and query messages needed for registrations or queries under the two scenarios.

With query optimization, 77% of the content names need to register with only 20 nodes because the corresponding matrices have only 1 replica. This is the absolute minimal cost to register a content name, since it has 20 AV-pairs. The maximum number of registration messages is 23, and the average is 20.3, i.e., an increase of less than 1 message over the minimal registration requirement of the system. However, in the random case, 93% of the registrations need more than 20 messages, which means they involve at least one matrix that has multiple replicas. The average number of registration messages goes up to 32.3, and the maximum is 88.

The two curves on the right side in Figure 3.17 compare the registration response time of the two scenarios. Sending more registration messages in the random scenario results in a longer response time: the average is 901 ms, whereas the average is 859 ms in the optimization scenario.

Note that the average response time is greater than two RTTs (400 ms), which is how

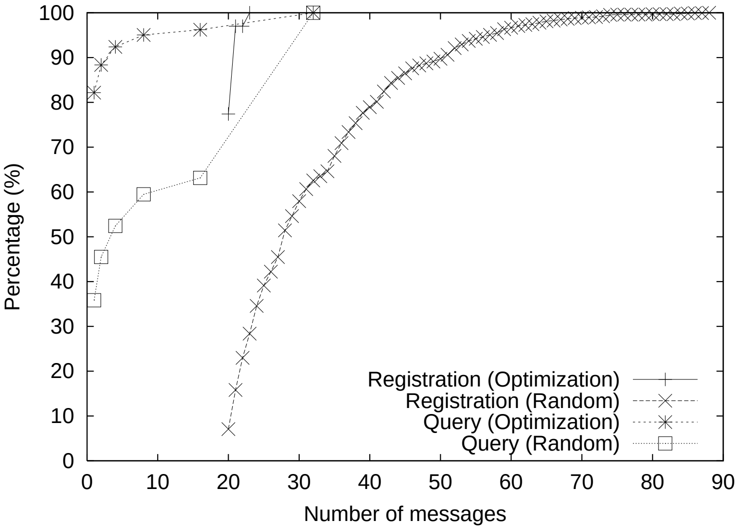


Figure 3.16: CDF of registration and query messages.

long it would take to register one AV-pair (matrix size probing and the actual registration). The main reason is that the response time is computed only when all the 20 AV-pairs' registrations are confirmed. More formally, this is equivalent to sampling an exponential distribution 20 times and take the maximum value instead of the average value.

Query Cost and Response Time

From query issuers' point of view, using query optimization, the average number of query messages (excluding the probing messages) is 2.7. This means on average a query is sent to matrices that have less than 3 partitions. In particular, from Figure 3.16, we observe that 82% of the queries are sent to matrices that have 1 partition, and only 3.7% of queries use the maximum number (32) of partitions. In comparison, in the random scheme, the average number of query messages is 13.8, only 36% of the queries are sent to matrices that have 1 partition, and 36.8% of queries are sent to matrices that have 32 partitions.

The cost of query optimization is the larger number of probing messages: instead of probing one matrix to get the size, the querying node has to probe all the matrices corresponding to the pairs in the query. This results in a slightly longer average response time for the query optimization scheme (597 ms vs. 594 ms). The two curves on the left in Figure 3.17 compare the CDF of the response times with and without query optimization. The CDF of the optimization scheme initially lies on the right side of the random scheme, but it has a steeper slope owing to a more uniform distribution. In practice, a query initiator can often cache the size of different matrices to reduce the number of probing messages for its future queries.

In summary, by using query optimization, while the system is accommodating high skewed load, both registration and query costs are kept near the minimum cost as defined by the basic RP-based system with no partitions and replications.

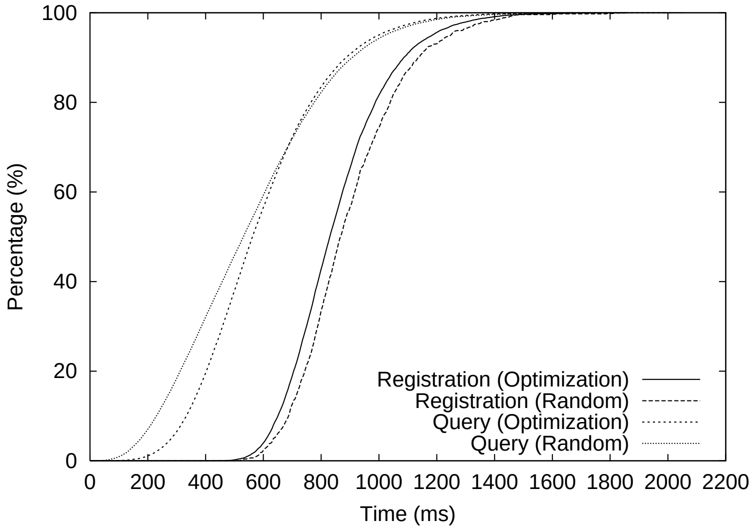


Figure 3.17: CDF of registration and query response time.

3.5.5 System under Flash Crowd

In this section, we examine the system's dynamic behavior under "flash crowd" type of workload. In particular, we show how an LBM expands under high query load and shrinks under low load.

As a reminder, the replication expansion is triggered whenever the observed query load by a matrix exceeds the query threshold $T_q = 200q/sec$. The replication shrinking is done periodically: a node checks the query rate it observes every T seconds (we set $T = 100$), and if it is lower than a low threshed T_q', then the node will issue a DEC_R_REQ to the head node and try to remove itself from the matrix. The head node removes the last row of the matrix by decreasing R, if it receives requests from the last row. We set $T_q' = \frac{1}{8}T_q$.

In the experiment, we first inject a registration load into the system and we examine one particular AV-pair, whose matrix has 8 partitions due to the registrations containing this pair. We then issue a query load to the system with varying query rate. All the queries in the query load contain this AV-pair and are sent to this matrix. The query load changes as the simulation time proceeds as follows:

- **T1: (No Query Load)** $0 < t < 60$. From 0 sec to 60th sec, there are no queries.
- **T2: (High Query Load)** $60 < t < 95$. At 60th sec, we start to issue a query load that consists of 20,000 queries, and the query rate is set to be 800 q/sec.
- **T3: (No Query Load)** $95 < t < 600$. During this time period, again there are no queries to this matrix.
- **T4: (Low Query Load)** $600 < t < 750$. At 600th sec, we issue another query load that consists of 20,000 queries with rate 200 q/sec.
- **T5: (No Query Load)** $t > 750$. There is again no query load.

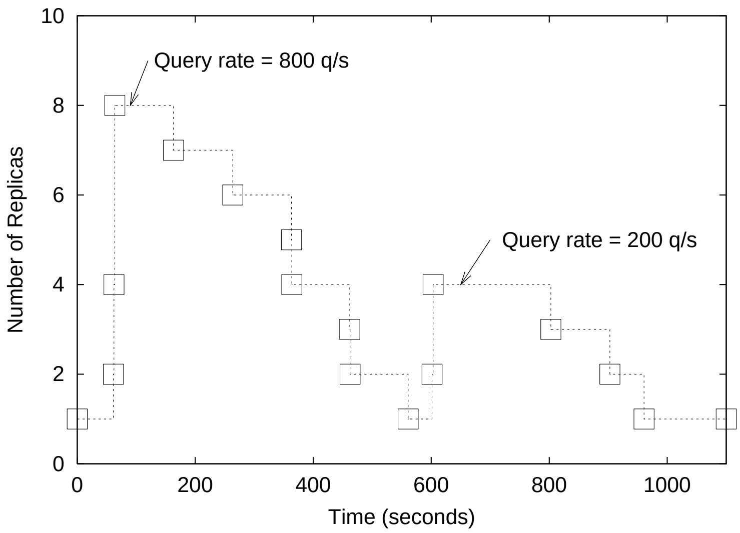


Figure 3.18: Matrix replication expansion and shrinking under changing query load.

The experiment is run for 1200 seconds of simulated time. Figure 3.18 plots how the number of rows changes in the matrix as time progresses. Initially during T1, since the matrix does not receive any query load, the matrix has only 1 row (1 replica). During T2, since the load is much higher than the query threshold $T_q(200q/sec)$, the matrix replicates itself multiplicatively and expands to 8 replicas. With these many replicas, the query success rate becomes 100%. During T3, since the load drops to 0, the matrix shrinks one row at each checking period, and eventually becomes one row only. With low query load, sometimes two rows may be removed within a short time interval (e.g., around the 400th second). Note that in this experiment, the last row in the matrix is kept because the content names did not expire during the simulation, which is why the minimum number of replicas is 1. During T4, again, the matrix ramps up to contain 4 replicas to accommodate the low query load. Finally in T5, as the query load drops again, the matrix shrinks back to one row.

3.6 Related Work

3.6.1 Load Balancing in Distributed Systems

Load balancing using partitions and replicas can trace its roots to early work in parallel databases, e.g., Gamma [21]. DDS [32] explores these ideas further in the domain of designing backend for Internet services in a server cluster setting. Upon receiving a request, the front end server selects a replica within a partition to best serve the request. Our system works in a peer-to-peer setting, and the selection of which node serves a request (query or registration) is done by the end points locally.

In the context of Content Distribution Networks (CDN), [72] proposes schemes where a request redirector can select a server replica from a dynamic list of servers to serve a URL request. The selection is based on the load of the servers, and the redirector may decide to grow the list of servers if the number of requests increases. This scheme is similar to one dimension of our load balancing mechanism, the replication expansion. However, in our system, the expansion is done in a distributed fashion by using high local query load to indicate the need of expansion, and no centralized entity like the redirector is needed. In addition, we also consider load balancing for registration.

3.6.2 Load balancing in DHT-based Systems

As we mentioned in Section 2.6, recently many systems have been built on top of DHTs. These applications, like the CDS, also face the load concentration problem. We now discuss how some representative work handle this problem, and compare with the CDS.

The resource discovery system Twine [10] observes that in the two applications that it studied, one set of bibliography files and one set of mp3 files, some resource descriptions are extremely popular. Since Twine uses a similar mapping scheme as our basic system, these popular descriptions will easily overload their corresponding nodes. Instead of conducting load balancing like our system, Twine simply sets a threshold on each node, and once the threshold is reached, the node will reject any new registrations. Another difference between our system and Twine lies in the query mechanism. To resolve a query in Twine, a random AV-pair is used to query the system. In CDS, we show that the query optimization mechanism, where it chooses the least popular AV-pair for query resolution, is important to ensure the system's performance under skewed load.

In the Chord based file system CFS [18], each file block is actively replicated at a fixed number of nodes in the system. These nodes occupy a consecutive section in the Chord key identifier space. Any one of these nodes can be used to serve this block. Doing so increases the system's availability and may also reduce the latency in retrieving a block by choosing a node that is close by. This mechanism is similar to the CDS in that it uses replicas to share query load. However, since the query load is dynamic, a fixed number of replicas do not work well: if the number is chosen too high, then resources may be wasted, and if it is set too low, then these replicas may not be enough to support a high query load. In our system, this is solved by adjusting the number of replicas depending on the current query load. Unlike in our system where we also partition registrations if the registration load is high, CFS does not address registration load balancing.

There also exist some work that address the load balancing problem by modifying the underlying DHT protocol. For example, in the information discovery system [61] proposed by Schmidt et al, to alleviate load on nodes caused by popular keywords, when a node joins the system, it will try to join a section in the DHT identifier space that has high load concentration and thus share the load. This mechanism will not work if the application load changes with time, since a previously heavily loaded section may become lightly loaded if the data distribution changes. This is why the authors also suggested nodes at runtime must exchange periodic messages to constantly re-adjust the load. Techniques similar to these are also used in [66].

Compared to the CDS, these systems have some fundamental shortcomings. First of all, these systems do not work well if application load changes often, since to maintain a balanced system, the nodes must frequently exchange load information and shift data around. In our system, nodes do not need to exchange load information, and the load balancing occurs automatically depending on the load, and is conducted in a fully distributed fashion. Second of all, the CDS does not require any modification to the underlying DHT protocol for load balancing. For example, how a node joins the DHT network is completely independent of the distribution of application load. This clean separation between the CDS and the underlying DHT allows the CDS to run on any existing DHTs as is. In contrast, the above systems must modify and add complexity to the already sophisticated DHT protocol, and this makes their deployment difficult.

3.7 Chapter Summary

One of the fundamental problem the basic CDS system faces is the rapid performance degradation due to uneven registration or query load. In this chapter, we presented a fully distributed load balancing mechanism that improves the system's throughput by eliminating hot-spots. The load balancing is based on a novel data structure, the load balancing matrix (LBM). Columns in an LBM are used to share high registration load, and rows are used to share high query load. Each LBM dynamically adjusts its own size in a distributed fashion based on the local load it observes. No centralized entity such as a load balancer is needed. The LBM coupled with the query optimization mechanism enables the system to perform well even under extremely skewed load, such as Zipf-like workload. In the mean time, the extra cost introduced to registrations and queries by load balancing remains low. Our extensive simulation results based on realistic load distribution validated the system's scalability and load balancing properties.

Chapter 4

Supporting Range Queries

The CDS we described so far supports searches based on subset matching by representing contents and queries with descriptive names. By deploying the distributed load balancing mechanism described in the previous chapter, the system works well even under skewed distribution of registration and query load. However, the matching of a query and content name is based on exact matching. For example, to find out the highway sections where speed equals 25mph, the query {`speed = 25`} will only match all cameras whose current description also has the AV-pair {`speed = 25`}. We call a query that is based on exact matching a *point query*. Besides point queries, users of a CDS may also pose *range queries* by specifying a range for one or more attributes. This type of query is especially useful for exploration purposes, when the user does not know what values are available within a range. For example, a user may want to find out the highway sections with heavy traffic by issuing the following query: "*return sections with observed speed less than 25mph*", or {`speed <= 25`}. Directly using the CDS "as is" to support range queries means we must break up a range query into a set of point queries, but this approach is inefficient for large ranges. Supporting range queries in other DHT-based systems such as distributed databases [38] has been posed as a challenging problem in the research community [35, 38].

In this chapter, we extend our CDS to support range queries efficiently. The support for range queries is centered around a distributed tree data structure, the Range Search Tree (RST), which partitions registrations based on their values with different levels of granularity. We first describe in Section 4.2 a set of algorithms that use a static RST to facilitate range queries by decomposing a range query into $O(\log R_q)$ sub-queries with R_q being the range length. This serves as a basis for the adaptive algorithms described in Section 4.4 that further enhance the system's efficiency. Our system is self-tuning: it optimizes itself based on the type of queries and load it observes to achieve efficiency for both queries and registrations. The system operates in a fully distributed fashion since all decisions are made locally. We present comprehensive simulation results to demonstrate the effectiveness of the system in Section 4.6. We discuss work related to range search in Section 4.7, and summarize in Section 4.8.

4.1 The Range Query Problem

Range queries are common and important for discovery and exploration purposes, as users may not know exactly what they are looking for. For example, a driver may issue the query "*return the speed observed by cameras that are between Exit 10 and Exit 50* ($10 \leq$ `exit` ≤ 50)", so that he may choose to get off the highway early to avoid congestion down the road. A police patrolling a highway section with speed limit of 55 mph may ask the system to "*return the list of cameras that observe speed higher than 75 mph* (`speed` ≥ 75)".

Formally, we define a range query as a query that contains at least 1 AV-pair that is specified using inequality signs such as "$<, >, \leq$" and "$\geq$", e.g., {`speed` $\geq$ `75` } and {`10` $\leq$ `exit` $\leq$ `50`}". We call these AV-pairs range pairs, and attributes in these pairs range attributes. A range query may also contain AV-pairs with equality predicates, and we call these pairs exact pairs. Similar to the definition of subset matching, a content name matches a range query, if it contains all the exact pairs in the query, and the values the range attributes take on fall into the corresponding range pairs in the query. For example a camera located at {`exit = 50`} observing {`speed = 80`} matches both of the above queries.

To resolve a range query, if it contains exact AV-pairs, we may choose one of them for resolving the query, and the inequality comparison is done at the corresponding RP node. This way we essentially treat the range query as a point query. However, this may not always be applicable. For example, if all AV-pairs in the query contain ranges, we have to deal with range pairs. It may also be the case that the exact AV-pairs are common and correspond to many partitions, and we do not want to query those matrices for efficiency reason. In the rest of the chapter, we focus on the scenario where a range AV-pair is used for query resolution. We often represent a range query as $Q : \{s \leq a \leq e\}$, or $Q : [s, e]$ while omitting the attribute a when no confusion will be caused. The length of the query is $R_q = e - s + 1$ assuming integer values.

4.1.1 Two Basic Approaches

Depending on how a content name is registered, there are two straightforward ways of resolving range queries.

Approach 1: To register a content name, we apply the hash function to each of its AV-pairs' attribute and value together as we did before. This is efficient for point queries, but to resolve a range query Q, we must break it up into R_q sub-queries, and send them to each node that corresponds to a value in the range. This approach works well if registrations are dense and queries cover relatively small ranges. However, it is an $O(R_q)$ approach and the number of query messages will become prohibitive when the query range increases. Moreover, if the registrations within this range are sparse, most of the query messages will be wasted.

Approach 2: Alternatively, we can apply the hash function to the attribute only for registration. This way all the content names that share the same attribute will end up registering at the same node, irrespective of the value; at query time, all point queries and range queries will also be sent to the same node for resolution. This approach performs well under light load, in that no matter what the range size is, all queries will have the same overhead. However, this node will become overloaded as the load increases. Fortunately, the load balancing mechanism will help by using more nodes to distribute the load. Still this solution is not efficient: each partition contains registrations with random values, so every query, including point queries, will have to visit all the partitions.

We observe that both approaches work well in some cases, but perform poorly in others. The problem is that neither solution takes the range of queries and the registration and query load into consideration. An ideal solution would behave similarly to the first approach for attributes that experience mostly point queries or queries over small ranges, but it would behave similarly to the second approach for attributes that experience light registration load and large query ranges. In the next two sections, we will present a system that uses the Range Search Tree data structure, and exhibits this adaptive behavior.

4.2 Static Range Query Mechanisms

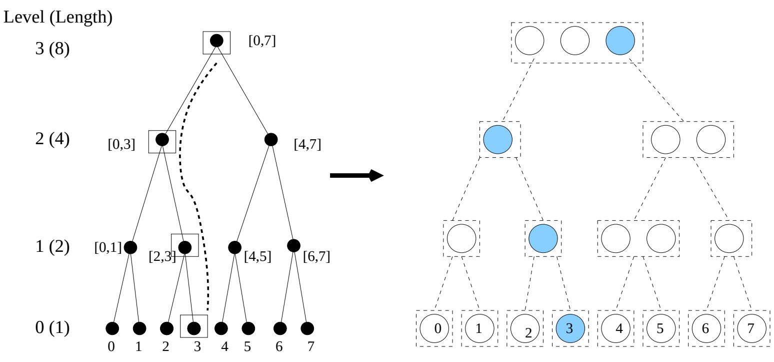
Our design to support range search is based on a distributed data structure, called the range search tree (RST). In this section we introduce the RST's organization, and describe the registration and query algorithms that use a static RST to support range searches. The static mechanisms are efficient when the query load is high, and they serve as a basis for the optimized dynamic algorithms to be presented in Section 4.4.

4.2.1 Range Search Tree (RST)

We consider an attribute a that takes on numerical values and may be searched using range queries. Suppose the domain of a is D_a, and D_a is bounded; values can be continuous or discrete. We break up D_a into n sub-ranges and represent each sub-range by its lower bound. D_a is thus the union of the sub-ranges $\{v_0, v_1, ..., v_{n-1}\}$, where $v_i < v_j$, if $i < j$. For example, if the attribute is `speed` in mph, we could break up the speed range into sub-ranges of 5 mph, and the value 35 would represent the range 35 $\leq$`speed`$<$ 40. Note that the sub-ranges may not be equal sized if we have prior knowledge of the distribution of the values of a.

The RST is a complete and balanced binary tree with n leaf nodes and $\lceil \log n \rceil + 1$ levels. (We assume n is a power of 2; otherwise, we round it up to the next power of 2 by filling in extra values.) Levels are labeled consecutively with the leaf level being level 0. Each node in the tree represents a different range. Leaf nodes correspond to the smallest sub-ranges, and each non-leaf node corresponds to the union of its two children. At level l, the range of the ith node from the left represents the range $[v_i, v_{i+2^l-1}]$. The union of all ranges at each level covers the full domain. The RST structure is similar to the segment tree data structure [60] used in computational geometry and spatial databases. A special case of an RST is a "unit RST" in which the domain is integer numbers and each leaf node represents one integer value. Figure 4.1(a) is an example unit RST with domain $[0, 7]$. In the rest of this chapter, we will present algorithms for a unit RST, but they can be generalized easily to a general RST.

We assume that each attribute's domain is known to all nodes in the system. As such, the logical structure of an attribute's RST is also known in the system. We map each node in the RST onto the underlying DHT-based overlay network by extending the techniques used in Chapter 3. Recall that an AV-pair may be mapped onto an LBM by applying



(a) Logical RST

(b) Physical overlay nodes

Figure 4.1: (a) A logical RST. The dotted curve illustrates $Path(3)$. (b) Overlay network nodes this RST is mapped onto. A circle represents a physical node and a dotted rectangle represents an LBM. Filled nodes are selected by the registration algorithm to receive $\{a = 3\}$.

the hash function to the AV-pair, and the LBM may have multiple partitions and replicas depending on the load corresponding to this AV-pair. Similarly, we map an RST node onto an LBM in the overlay network by treating the RST node's range as a parameter to the hash function. More specifically, given a range $[s, e]$, the node IDs in the overlay network that this range is mapped onto correspond to the hash of the following 4-tuple: the attribute, the range, the column and row indices in its LBM:

$$N \leftarrow \mathcal{H}(a, [s, e], p, r). \tag{4.1}$$

Figure 4.1 is an example that shows the mapping of an RST with domain of $[0, 7]$ onto the overlay network. The root RST node is mapped onto an LBM that has 3 partitions, and the node ID of the filled node (in the 3rd partition) corresponds to $\mathcal{H}(a, [0, 7], p = 3, r = 1)$. The LBMs may have different number of partitions and replicas depending on the load they receive. In Figure 4.1(b), for each LBM, we only show 1 replica for clarity. It is important to note that the parent-child relationships between nodes do not need to be actively maintained, and a node can infer its parent or children's range based on its own range.

4.2.2 Registration

We now describe the registration algorithm used by an endpoint to conduct registrations. Figure 4.2 lists the pseudo code of the registration algorithm.

To register a name, CN, that contains AV-pair $\{a = v\}$, the algorithm first computes the height of a's RST based on the domain size (Line 2, where m is the length of D_a). For value v, since each node in the RST covers a unique range, v is thus within exactly one node's range at each level. This set of nodes forms a **path** from leaf node $N[v, v]$ to the root. We name it $Path(v)$. Lines 4-6 determine the node at level l in the RST whose range covers v, in particular, the starting point, s, is determined by dividing v with the length of the level l nodes. Once the path is determined, we register $\{a = v\}$ with each LBM

1: REGISTER-RST($\{a = v\}, CN$) {
2: $max\_levels \leftarrow \lfloor \log(m) \rfloor$;
3: **foreach** $0 \leq l \leq max\_levels$ {
4: $s \leftarrow \lfloor \frac{v}{2^l} \rfloor$; // 2^l: node length at level l
5: $e \leftarrow s + 2^l - 1$;
6: $range \leftarrow [s, e]$;
7: $(P, R) \leftarrow find\_matrix\_size(a, range)$;
8: $p \leftarrow random(P)$;
9: **foreach** $1 \leq r \leq R$ {
10: $N_{range}^{(p,r)} \leftarrow \mathcal{H}(a, range, p, r)$;
11: $register(N_{range}^{(p,r)}, \{a = v\}, CN)$;
12: }
13: }
14:}

Figure 4.2: Endpoint registration algorithm using a static RST.

corresponding to each node in $Path(v)$. The actual registration algorithm with an LBM is identical to the one described in Figure 3.2: the algorithm determines the corresponding LBM's size (Line 7), picks a random partition (Line 8), and sends the name to each node in the selected partition (Lines 9-12). As an example, Figure 4.1 illustrates the registration of $\{a = 3\}$: it is registered with each physical node within a selected partition of each level's LBM.

This registration algorithm automatically aggregates AV-pairs at different levels of granularity. As the level increases, since there are fewer nodes in the RST, the LBM corresponding to one RST node may have more partitions. For example, in Figure 4.1(b), each LBM at the leaf level has one partition, and the root level LBM consists of 3 partitions. Since the structure of an RST is determined by the domain of the corresponding attribute, registrations are carried out in a fully distributed fashion: based on the value in an AV-pair, an endpoint can locally determine the set of nodes in the network that it should register with and it does not need to consult any other node or traverse the tree. As a result, no bottlenecks are created in the system.

4.2.3 Query

Given the registration mechanism, there are many ways to decompose and resolve a range query using the RST. The efficiency of a query algorithm is determined by how the range is decomposed. To find an efficient algorithm, we introduce the *relevance* metric to guide our design. Formally, suppose a query algorithm decomposes a query $Q : [s, e]$ into k subqueries, corresponding to k nodes in the RST, $N_1, ..., N_k$. The relevance r of this algorithm is defined as:

$$r = \frac{R_q}{\sum_{i=1}^{k} R_i}, \tag{4.2}$$

where R_i is node N_i's range length, and R_q is the query's length. Clearly, $0 < r \leq 1$.

Intuitively, the relevance indicates how well the query range matches the RST nodes that are being queried. Low relevance algorithms, such as sending a point query to the root level, may be inefficient for both queries (visit nodes with a low concentration of relevant registrations) and registrations (the query load concentration may cause the root level to replicate often). In contrast, decomposing a query to leaf level nodes has a relevance of 1. From the registration's point of view, there will be no unnecessary replications, but from the query's point of view, this may require too many sub-queries, so it is not a desirable algorithm either.

We now present our efficient query algorithm. Our algorithm minimizes the registration cost by maximizing the relevance ($r = 1$), and in the meantime, it is also efficient for queries by decomposing query ranges into small number of sub-queries. We first establish the **Range Decomposition Theorem**, which serves as the foundation of the query algorithm, and then describe the query algorithm itself. To facilitate our discussion, we define the following terms.

Definition 1 (Relevant Node). *Given a query $Q : [s, e]$ with range length $R_q = e - s + 1$, a Relevant Node (RN) is a node in the RST whose range is a subset of Q's range.*

Definition 2 (Cover). *A Cover is a set of relevant nodes,*

$$Cover(Q : [s, e]) = \{RN_1, RN_2, ..., RN_n\},$$

that satisfies the following two conditions:

(1) $RN_1 \cup RN_2 \cup ... \cup RN_n = [s, e]$

(2) $\forall i, j, RN_i \cap RN_j = \phi$.

Definition 3 (Height of a Cover). *The Height of a Cover is defined as the level of the node in the Cover that has the largest level number.*

Definition 4 (Minimum Cover). *The Minimum Cover(MC) is the Cover that has the smallest size among all possible covers.*

We introduce the following three Lemmas to facilitate the proof of the decomposition theorem.

Lemma 1. *The* **MC** *has the largest Height, H, among all possible covers, and $H \leq \lfloor \log_2 R_q \rfloor$. In other words, the MC does not contain nodes with level higher than $\lfloor \log_2 R_q \rfloor$.*

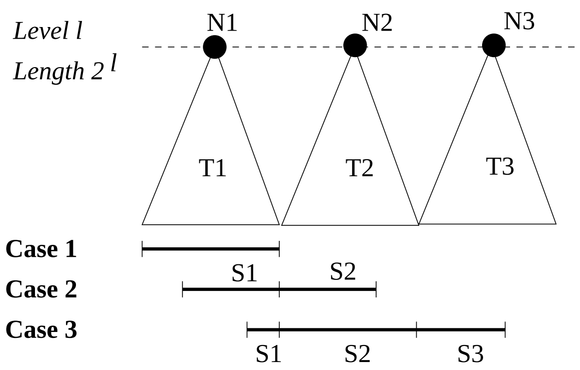


Figure 4.3: Illustration for proof of the Range Decomposition Theorem.

Proof. Prove by contradiction. Suppose MC's height H is not the largest, then there must exist a different Cover C' with height, $H' > H$. It follows that there must exist at least one RN, N', in C' at level H', and N' is not in MC. Now consider the sub-tree rooted at N'. Suppose the range that corresponds to this sub-tree is $[s', e']$. The MC must contain at least two nodes from this sub-tree to cover this range. It is clear that we can substitute these two nodes in the MC with N' to reduce the size of the MC by 1. This contradicts the MC's definition.

Now we show $H \leq \lfloor \log_2 R_q \rfloor$. Let $h = \lfloor \log_2 R_q \rfloor$. $2^h \leq R_q < 2^{h+1}$. We show any node at level $h+1$ or higher can not be a RN for R_q. Consider a node N, at level $h+1$. Suppose its range is R_N with length 2^{h+1}. There are only the following possible relationships between R_N and R_q: (1) $R_q \subset R_N$, (2) $R_q \cap R_N \neq \phi$ and (3) $R_q \cap R_N = \phi$. In any case, node N can not be a RN, since it is never the case that $R_N \subset R_q$. □

Lemma 2. *For range $[s, e]$, if s is the starting point, or e is the ending point of a node at level l, then $|MC| \leq l + 1$, where $l = \lfloor \log R_q \rfloor$.*

Proof. Prove by induction. We show the statement holds when s is the starting point. When e is the ending point, it can be shown by symmetry.

As the base case, for point query with range length 1, $l = 0$, and $|MC| = 1$. As the induction hypothesis, suppose $|MC| \leq (l - 1) + 1$ for range with length $R_q/2$, where $l = \lfloor \log R_q \rfloor$.

For the induction step, we have range $[s, e]$ with length R_q and s is the starting point of a node at level l, where $l = \lfloor \log R_q \rfloor$. We also have $2^l \leq R_q < 2^{l+1}$.

We break the range into two sub-ranges: $[s, s + 2^l - 1]$ and $[s + 2^l, e]$. The first sub-range corresponds to exactly one node at level l. For the second sub-range, its length $R'_q = R_q - 2^l < 2^l$, and $s + 2^l$ is the starting point of a node at level $l' = \lfloor \log R'_q \rfloor \leq l - 1$. Now use the induction hypothesis, we know for this sub-range, its MC, $|MC'| \leq l' + 1 \leq (l-1) + 1 = l$.

Thus, the MC for $[s, e]$ is the union of the two MCs for the two sub-ranges. It's size is thus: $|MC| \leq 1 + |MC'| = l + 1$. □

Lemma 3. *A range $[s, e]$ intersects with at most 3 consecutive nodes at level $l = \lfloor \log R_q \rfloor$ in the RST.*

Proof. Figure 4.3 illustrates the three possible relationships between the range $[s, e]$ and the nodes at level l. Based on the definition of the RST, the starting value of the node at level l that contains the starting value s is $p = \lfloor \frac{s}{2^l}$. Suppose node N_1 starts with p, and node N_2 and N_3 are the nodes after N_1. These three nodes' ranges are: $N_1[p2^l, (p+1)2^l - 1], N_2[(p+1)2^l, (p+2)2^l - 1], N_3[(p+2)2^l, (p+3)2^l - 1]$

If $e = (p+1)2^l - 1$, then $[s, e]$ intersects with one node N_1 (Case 1). If $e \leq (p+2)2^l - 1$, then $[s, e]$ intersects with two nodes N_1 and N_2 (Case 2). If $e \geq (p+2)2^l$, then $[s, e]$ intersects with three nodes N_1, N_2 and N_3 (Case 3). □

Theorem 1 (Range Decomposition Theorem). *For a given $Q : [s, e]$, the size of its MC satisfies:*

$$|MC| \leq 2 \cdot \lfloor \log_2 R_q \rfloor.$$

Proof. From Lemma 3, we know that range $[s, e]$ intersects with at most 3 consecutive level $l = \lfloor \log R_q \rfloor$ nodes in the RST. We name their subtrees of these three nodes T_1, T_2, T_3. There are only three possibilities how the range $[s, e]$ may intersect with the three subtrees. (1) intersect with T_1 only; (2) intersect with T_1 and T_2; (3) intersect with T_1, T_2 and T_3. Now we examine each case.

Case 1: This occurs only when $s = p2^l$ and $R_q = 2^l$. We use node N1. Thus $|MC| = 1$.

Case 2: We know from Lemma 1 that MC includes nodes only from level l and below. We divide the range into two segments: $S_1 = [s, (p+1)2^l - 1]$, and $S_2 = [(p+1)2^l, e]$. Use Lemma 2, $|MC_{S_1}| \leq (l-1) + 1 = l$. Similarly $|MC_{S_2}| \leq l$.

MC_{S_1} and MC_{S_2} are disjoint, hence $|MC| = |MC_{S_1}| + |MC_{S_2}| \leq 2l$.

Case 3: We divide the range into three segments:

$S_1 : [s, (p+1)2^l - 1]$, $S_2 : [(p+1)2^l, (p+2)2^l - 1]$, and $S_3 : [(p+2)2^l, e]$.

We know the length of $S_2 = 2^l$, and at most one of S_1 and S_3 may have length longer than 2^{l-1}. Suppose $S_3 > 2^{l-1}$ and $S_1 < 2^{l-1}$.

$|MC_{S_1}| \leq (l-2) + 1 = l - 1$; $|MC_{S_2}| = 1$, and $|MC_{S_3}| \leq (l-1) + 1 = l$.

Hence $|MC| = |MC_{S_1}| + |MC_{S_2}| + |MC_{S_3}| \leq l - 1 + 1 + l = 2l$.

□

Query algorithm

There are two main steps in the query algorithm. Figure 4.4 lists the pseudo code used by an endpoint.

First, to resolve a query $Q : [s, e]$ with length R_q, the endpoint must decompose the query range into a set of sub-ranges. For this purpose, the querying node locally calls a function Range-decomp to decompose Q to a list of sub-queries (Line 3, Figure 4.4). Figure 4.5 lists the pseudo code for the Range-decomp function, which recursively computes the MC for a given range $[s, e]$. It determines the MC by finding one sub-range at a time. It first makes sure the range is valid (Line 2, Figure 4.5), and then computes its length (Line 3,

1: QUERY-RST($Q : [s, e]$) {
2: $\quad init\_list(query\_list * ql)$;
3: $\quad$ RANGE-DECOMP($Q : [s, e], ql$);
4: $\quad$ **foreach** $Q : [s', e'] \in *ql$ {
5: $\quad\quad range \leftarrow [s', e']$;
6: $\quad\quad (P, R) \leftarrow find\_matrix\_size(a, range)$;
7: $\quad\quad$ **foreach** $1 \leq p \leq P$ {
8: $\quad\quad\quad r \leftarrow random(R)$;
9: $\quad\quad\quad N_{range}^{(p,r)} \leftarrow \mathcal{H}(a, range, p, r)$;
10: $\quad\quad\quad send\_query(N_{range}^{(p,r)}, Q : [s', e'])$;
11: $\quad\quad$ }
12: $\quad$ }
13: }

Figure 4.4: Endpoint query algorithm using a static RST.

Figure 4.5) and the maximum level that this range's MC corresponds to (Line 4, Figure 4.5). In the MC, the first node must be the highest node with a starting value s. Lines 5-13 in Figure 4.5 find this node by examining the tree in a top-down fashion. Once that node is found, its range is inserted to the output list of sub-queries (Line 7, Figure 4.5), and the function recursively decomposes the rest of the range (Line 8, Figure 4.5).

Second, once the list of sub-ranges, or the MC is computed, the querying node then sends each sub-query to the LBMs in the overlay network that correspond to each MC node. The querying algorithm again is the same as before: For each sub-query, based on its range, the query issuer retrieves the corresponding LBM's size (Line 6, Figure 4.4), and then sends it to a random node in each partition (Lines 7-11, Figure 4.4).

Figure 4.6 is a decomposition example for query $Q : [1, 7]$. Like the registration algorithm, the query algorithm is also fully distributed and deterministic, in that the decomposition is done by the querying node itself based on its query range, and no traversal of the tree is needed. Since the sub-ranges are computed based on the Range Decomposition Theorem, the number of sub-ranges is $O(\log R_q)$, which ensures the efficiency of the query algorithm. In addition, the query algorithm separates queries with different ranges, thus avoids creating bottlenecks. For example, a point query will be sent to the leaf level, and a large query will use nodes higher up in the tree.

4.3 Analysis of the Static Mechanisms

In this section, we analyze the cost of the registration and query algorithms using a static RST described above. The cost of a registration is determined by the number of messages need to be sent by an endpoint. Similarly, the cost of resolving a query is equal to the number of query messages that must be sent to the system. In the following analysis, we assume that the domain for attribute a is $D_a = [0, m-1]$; this results in $T = \lceil \log m \rceil + 1$

```
RANGE-DECOMP(Q : [s, e], query_list * ql) {
    if (s > e) return;
    R_q ← e − s + 1; // range length
    l ← ⌊log(R_q)⌋; // maximum level
    while (l ≥ 0) {
        if (remainder(s/2^l) == 0){
            insert_to_query_list(Q : [s, s + 2^l − 1], ql);
            RANGE-DECOMP(Q : [s + 2^l, e], ql);
            break;
        } else {
            l ← l − 1;
        }
    }
    return;
}
```

Figure 4.5: The local range decomposition algorithm.

levels in the RST.

4.3.1 Number of Registration Messages

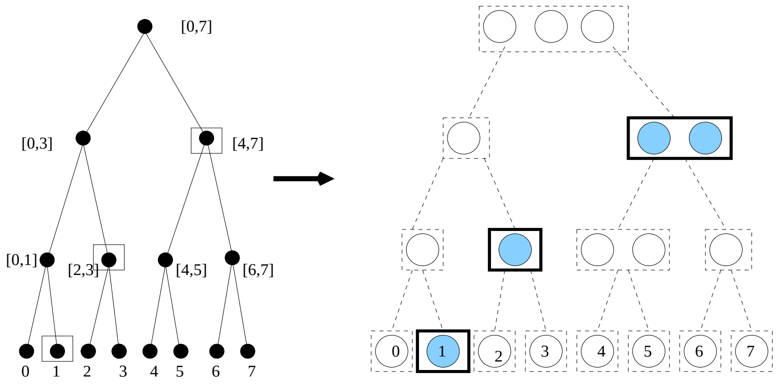
From Section 4.2.2, we know that the number of registration messages needed to register pair $\{a = v\}$, N_{R_v}, is determined by the query load that comes to this RST. More specifically, it equals the sum of the number of replicas (rows) in the LBMs that are on $Path(v)$ (See Figure 4.1 for an example). Assume L^Q is the total query load for attribute a. If the query load on the RST node at level t in $Path(v)$ is L_t^Q, then the number of replicas in its LBM is $\lceil \frac{L_t^Q}{C_Q} \rceil$, with C_Q being the maximum query capacity of a node, e.g., the query rate that this node can sustain. The total number of registration messages needed to register $\{a = v\}$ is then:

$$N_{R_v} = \sum_{t=1}^{T} \lceil \frac{L_t^Q}{C_Q} \rceil. \tag{4.3}$$

Proposition 1 gives an upper and lower bounds for the number of messages needed to register a value $a = v$ under a query load L^Q.

Proposition 1

$$T \leq N_{R_v} \leq T + \lceil \frac{L^Q}{C_Q} \rceil \tag{4.4}$$



(a) Logical RST

(b) Physical overlay nodes

Figure 4.6: Range [1, 7] is decomposed into 3 sub-ranges indicated by the nodes with a box. Filled nodes will receive the query.

Proof. The property of the ceiling function gives,

$$1 \leq \lceil \frac{L_t^Q}{C_Q} \rceil \leq \frac{L_t^Q}{C_Q} + 1 \tag{4.5}$$

Apply sum to all terms,

$$T \leq N_{R_v} = \sum_{t=1}^{T} \lceil \frac{L_t^Q}{C_Q} \rceil \leq T + \frac{\sum_{t=1}^{T} L_t^Q}{C_Q} \tag{4.6}$$

We divide L^Q into two sets, L^{Q_v}, which consists of all the queries that include v, and $L^Q - L^{Q_v}$, which is the rest of the queries. Queries in $L^Q - L^{Q_v}$ do not contribute load to the nodes in $Path(v)$. Each query in L^{Q_v}, after decomposition, contributes exactly one sub-query to one node in $Path(v)$, since nodes in the MC have disjoint ranges (by definition). Therefore,

$$\sum_{t=1}^{T} L_t^Q = L^{Q_v} \leq L^Q. \tag{4.7}$$

Substitute Eq. 4.7 into Eq. 4.6, we get Proposition 1. □

N_{R_v} reaches its maximum value, when v is contained in every query in L^Q. It reaches the minimum value T, when v is not contained in any query. In Figure 4.7, the number of registration messages, N_{R_v}, lies between the two solid lines.

We compare the cost of using a static RST with the approach where we only use the root (Approach 2 in Section 4.1). In this approach, the number of messages needed for *any* registration, N'_R, is determined by the number of replicas in the root level LBM: $N'_R = \lceil \frac{L^Q}{C_Q} \rceil$. The cost grows linearly with the query load, as is shown by the dotted line in Figure 4.7. The Root Only case is more efficient when the query load is low, specially, when the number of replicas at the root level is smaller than the height of the RST, i.e., $L^Q/C_Q \leq T$. As the load increases beyond that point, the registration cost using RST is generally lower than using only the root.

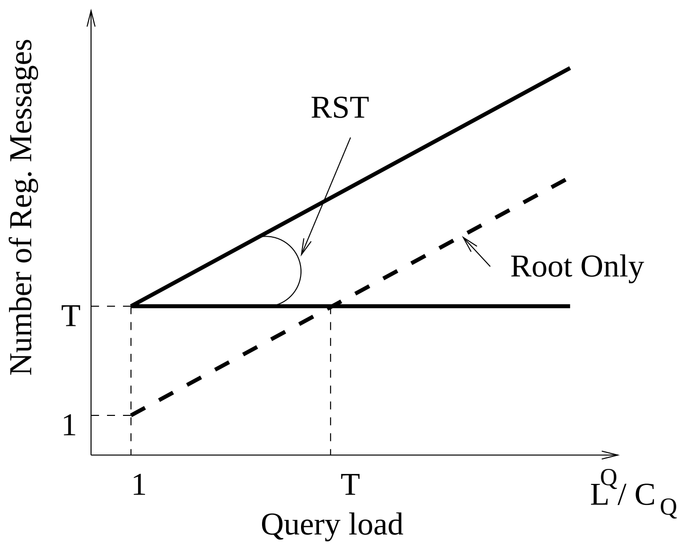


Figure 4.7: Number of registration messages needed for a value v as a function of query load.

4.3.2 Number of Query Messages

Similarly we can compute the number of query messages needed to resolve a given query, $Q : [s, e]$, with $R_q \leq m$. Assume the size of Q's MC is K, from the decomposition theorem, we know $1 \leq K \leq 2\lfloor \log R_q \rfloor \leq 2\lfloor \log m \rfloor$.

Consider an arbitrary registration load that consists of L^R registrations. The number of query messages needed to resolve Q equals the sum of the number of partitions in the LBM that corresponds to each node in the MC:

$$N_Q = \sum_{k=1}^{K} \lceil \frac{L_k^R}{C_R} \rceil \tag{4.8}$$

where C_R is the registration capacity of each node, and L_k^R is the registration load observed on the kth node. Since nodes in the MC are disjoint from each other, each registration in L^R occurs in L_k^R at most once, thus,

$$N_Q = \sum_{k=1}^{K} L_k^R \leq L^R, \tag{4.9}$$

Similar to the previous derivation, we can derive the following bounds for N_Q:

$$1 \leq K \leq N_Q \leq K + \lceil \frac{L^R}{C_R} \rceil \tag{4.10}$$

N_Q's lower bound is K, or $O(\log R_q)$, if each LBM of an MC node has only 1 partition. N_Q becomes larger if some of the LBMs have more than 1 partition. For example, in Figure 4.6, $N_Q = 4$, as query $Q : [1, 7]$ will be sent to the 4 filled nodes. N_Q takes on the

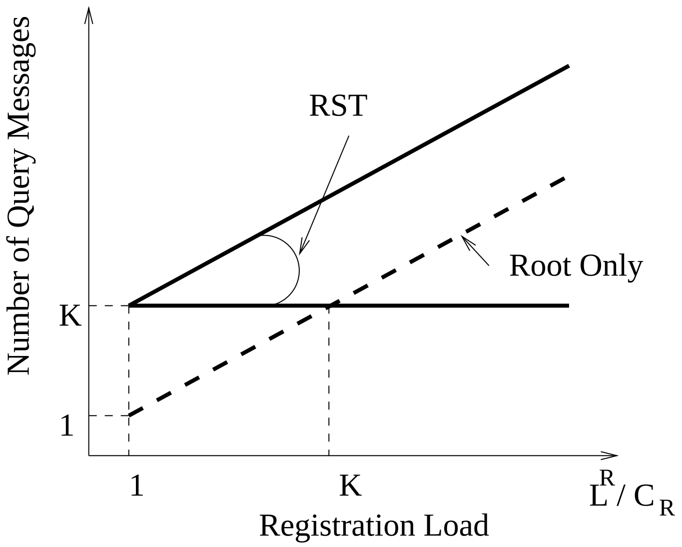


Figure 4.8: Number of query messages needed for a query with covering set of size K as a function of registration load.

maximum value when all the registrations in the registration load are within the query's range. Figure 4.8 shows the two bounds.

We again compare our algorithm with the root only approach. If we use the root level nodes to resolve a query, then the number of query messages needed equals the number of partitions of the root level matrix: $N'_Q = \lceil \frac{L^R}{C_R} \rceil$, as shown by the dotted line in Figure 4.8. For low registration load, $L^R/C_R \leq K$, the Root only case performs better than RST. Our algorithm becomes more efficient when the registration load increases beyond that point.

4.3.3 Discussion

The above cost analysis points out the following deficiencies in the static algorithms: (1) Proactively registering with all levels of the RST without considering query ranges can be wasteful. For example, if all query ranges are smaller than R_q, then registering with levels higher than $\lfloor \log R_q \rfloor$ is unnecessary, since no query will be sent there. (2) Decomposing a range solely based on its length while ignoring the registration and query load information is suboptimal. In particular, if both the registration and query load in a subtree of the RST are low, we do not need to decompose the query in the subtree and should just use the subtree root. Subsequently, we do not need to register with levels other than the root in the subtree.

4.4 Dynamic Range Query Mechanisms

To overcome the above shortcomings of the static range search algorithms, we now present our dynamic range query design that further optimizes the system's performance.

While in the static mechanisms, registrations go to every level of the RST regardless of

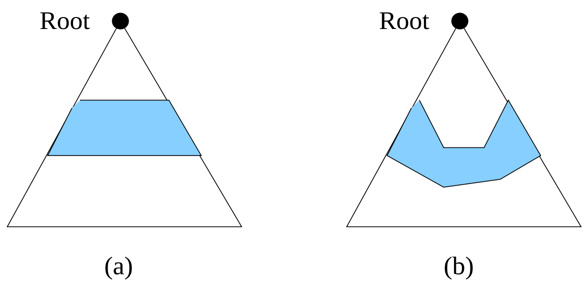


Figure 4.9: RST with (a) a flat band, and (b) a ragged band. The shaded area indicates a band.

the queries, the idea here is to only register with the nodes that are needed based on the query ranges and the load information observed by the system. We call this set of nodes the *band* (Figure 4.9): only nodes in the band will accept registrations and will be able to resolve queries. As such, only the LBMs corresponding to the band nodes will have a non-zero size. The shape of the band is not necessarily flat (Figure 4.9(b)), and it changes depending on the registrations and queries for this attribute. For example, if the registration load is low and the query ranges are large, the band will migrate upwards toward the root.

In this section, we first present the Path Maintenance Protocol, which allows endpoints to discover the band. We then describe how endpoints use the band information to conduct registrations and issue queries in a distributed fashion. Finally, we present the local algorithms executed on nodes that are located on the top and bottom edges of the band to adapt the band to the load, since a fixed band is not efficient for load that changes over time.

4.4.1 Path Maintenance Protocol (PMP)

When the static RST is used, endpoints know of the band (the full tree) by default, and as we described earlier, they can use purely local algorithms to determine which nodes a registration or a query should be sent to. In the dynamic case, only a band of nodes is used, and therefore endpoints must discover what nodes are in the band before they can issue a registration or query. Clearly an endpoint may traverse the RST starting from the root to find out the band information. The obvious problem with this approach is that the root node will become a bottleneck when the registration or query load increases. In addition, endpoints may suffer a long delay if the band is located away from the root near the bottom of the tree.

To ensure that endpoints can efficiently retrieve the band information in a distributed fashion without traversing the tree, we designed a lightweight protocol, the Path Maintenance Protocol (PMP). The goal of the PMP is to propagate RST's band information to nodes in the tree, so that endpoints can find the band efficiently.

Recall from Chapter 3 that for load balancing purposes, the size of an LBM is stored in its head node. To facilitate range search, a head node must also maintain matrix sizes of some other nodes in the tree. We call the collection of the matrix size information the Path Information Base, or PIB. The PIB on a node consists of two components: the *path component* contains the matrix size of nodes in the path from this node to the root of the

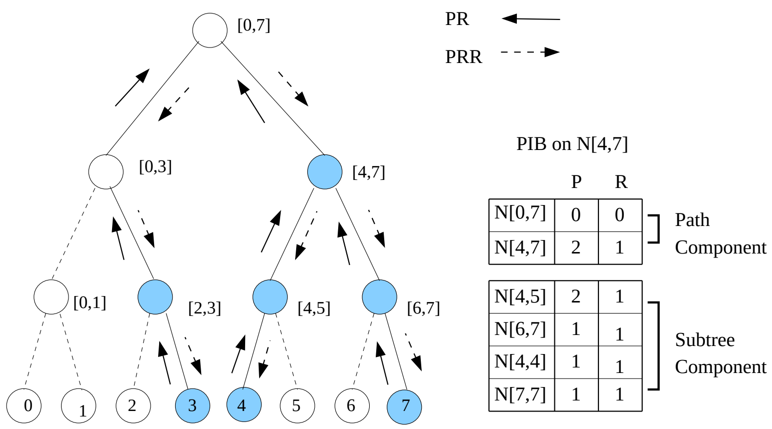


Figure 4.10: PMP message exchange among head nodes. Filled nodes are in the band. Matrix sizes are from Figure 4.1(b).

RST, and the *subtree component* contains the matrix size of nodes in this node's subtree.

The PIB is established through the exchange of two types of periodic messages: Path Refreshing (PR) and Path Refreshing Reply (PRR) messages (Figure 4.10). Each head node in the band periodically sends a PR message to its parent; the message contains the node's current subtree component of its PIB. When a node receives a PR message, it updates the corresponding entries in its PIB. When it is time for this node to send its own PR message, it will propagate the updated PIB to its parent. Through the PR messages, each node collects up-to-date subtree status. The periodical PRR messages are issued down the tree along the reverse paths of the PR messages. In particular, a node sends a PRR message to each of its children from which it receives PR messages before. The PRR message includes this node's PIB's path component. Upon receiving a PRR message, a node updates its PIB's path component accordingly. Subsequently, in its own PRR message, the node will send the updated path component to its children. Through the PRR messages, each node's path component is updated.

In addition to allowing nodes to establish their PIBs, the PMP messages also act as a fault tolerance mechanism to re-establish the PIB if nodes fail or leave the system. For example, if a parent node fails, the PR messages from its children will be forwarded to a current live node that is close to the original parent node in the DHT key identifier space. This node will then become the new parent. Using PIB, endpoints can issue registrations and queries in a fully distributed fashion, as we describe next.

4.4.2 Registration

To register an AV-pair $\{a = v\}$, an endpoint first discovers the band by probing any head node in $Path(v)$. Nodes in lower levels are preferred for probing, since collisions with probes for other values are less likely to occur at those levels. If the node being probed has an established PIB, it can provide the matrix size information for all nodes in $Path(v)$ to the probing endpoint. The endpoint then registers only with the nodes that are in the band. Endpoints may cache the retrieved path information for future use, thus avoiding repeated probing.

If the node being probed does not have a PIB because it is too low in the tree and the PMP messages did not reach it, the node replies with a NULL. The endpoint can then probe a node higher in the path, e.g., the node located halfway between the first probed node and the root in $Path(v)$. This "binary probing" method ensures the maximum number of probe messages needed to find a node with an established PIB is $O(\log T)$, where T is the height of the RST.

The first registration is a special case, because all the probes will return NULL. To handle this, we define a default band, whose level is known to the endpoints. The first registration will go to the default level, and the head node of the default band will then start the PMP message exchanges. One option is to use the root level as the default band.

4.4.3 Query

To resolve a query $Q : [s, e]$ with range length R_q, the querying node also needs to retrieve the band information first. Ideally, probing the root node suffices since its PIB contains the whole tree. However, the root node may become a bottleneck if all probes go there. It is also possible to probe the lowest node in the RST whose range contains the range $[s, e]$. The problem with this approach is that even for a small range, the node that contains it may be a node high in the tree, which must handle probes from large ranges. As an example, in Figure 4.10, the node that covers range $[3, 4]$ is the root node.

A good probing algorithm should minimize the possible collisions among probes. As we showed in Lemma 3, that for range $[s, e]$ there exist at most 3 consecutive RST nodes at level $\lfloor \log R_q \rfloor$ that cover the range. In our algorithm, we choose to probe these nodes to minimize collisions with probes from queries with different lengths and ranges. Each probed node returns its complete PIB to the querying node.

Based on the returned PIB information, the querying node reconstructs the logical RST and annotates each RST node with its LBM's size, which reflects the load status. It then executes a local algorithm FIND-MIN-DECOMP to decompose the query range (Figure 4.11). The FIND-MIN-DECOMP function takes as input the query range and the root of the RST data structure, and returns a list of nodes corresponding to the decomposition. FIND-MIN-DECOMP differs from the static RANGE-DECOMP decomposition function in two aspects. First, it only considers nodes within the band. Second, rather than ignoring load information, FIND-MIN-DECOMP compares the cost of using a subtree root or nodes within the subtree, and chooses the more efficient option, e.g., the one that will result in fewer query messages.

FIND-MIN-DECOMP is a recursive function. Lines 2-5 are terminating conditions. The function first stores the LBM size of the current node (Lines 6-7). Lines 9-15 recursively retrieve the list of decomposition results using each of the child nodes if they are in the band. Lines 16-18 computes the total cost of using the children lists. If the cost of using the children is less than using the parent, then the function will return the children lists (Lines 19,20). If the parent LBM has fewer partitions and the number of its replicas is smaller than a threshold $T_{replica}$, then the function will return the parent (Lines 22-23). Otherwise, the function will return the children (Line 25). The reason that we use the children when the parent has many replicas (received many queries already) is to avoid overloading the parent.

Once the list of nodes is determined by FIND-MIN-DECOMP, the querying node will then send the query to the LBMs that correspond to each node in the list. This step is the same as the algorithm presented in Figure 4.4. Figure 4.12 shows the same example as in Figure 4.6, but at the subtree rooted at $N[0,3]$, we use it rather than $N[0,1]$ and $N[2,3]$ to reduce the query cost from 4 to 3 for query range $[1,7]$.

```
node_list *FIND-MIN-DECOMP(node, range) {
    if (range == NULL)
        return NULL;
    if (node == range) // exact query
        return node;

    P_parent ← number of partitions in LBM_node;
    R_parent ← number of replicas in LBM_node;

    left_range ← range ∩ node.leftchild.range;
    right_range ← range ∩ node.rightchild.range;
    if (node.leftchild in the band) {
        left_child_list ← FIND-MIN-DECOMP (node.leftchild, left_range);
    }
    if (node.rightchild in the band) {
        right_child_list ← FIND-MIN-DECOMP (node.rightchild, right_range);
    }

    P_left ← ∑ P in left_child_list;
    P_right ← ∑ P in right_child_list;
    P_children = P_left + P_right;

    if (P_parent > P_children)
        return CONCATENATE(left_child_list, right_child_list));

    if (P_parent ≤ P_children) {
        if (R_parent ≤ T_replica)
            return node;
        else
            return CONCATENATE(left_child_list, right_child_list));
    }
}
```

Figure 4.11: The optimized range decomposition algorithm.

4.4.4 Distributed Band Adaptation

The band allows us to minimize the cost of registrations and queries for a given load. However, the load may change, and if the band does not adapt, the system may perform poorly under the new load. Consider the example in Figure 4.12. At a given time, due to a high frequency of point queries, suppose the band only contains nodes from the lowest level. Now many queries come in with range $[0,3]$. Using the current band, each such query

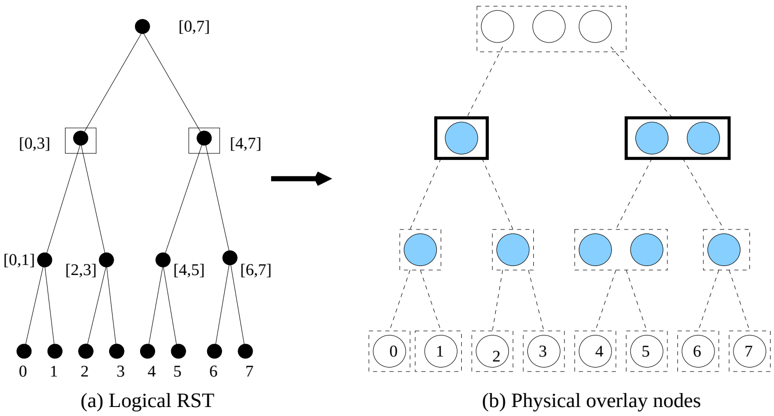


Figure 4.12: Query range $[1, 7]$ is decomposed into 2 sub-ranges indicated by the solid boxes. Filled nodes are in the band.

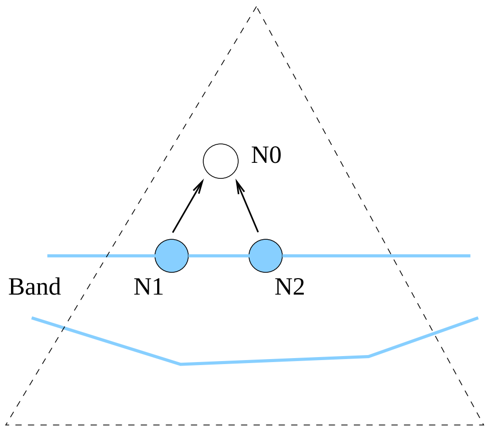


Figure 4.13: Band top expansion. Nodes N_1 and N_2 are at the top edge of the current band. N_0 is recruited to the band.

would be decomposed into into 4 sub-queries. However, if node $N[0, 3]$ were in the band, we could resolve such a query with only one query message by sending the query to node $N[0, 3]$, since $N[0, 3]$'s LBM has just one partition in the overlay network.

Band adaptation is controlled by the nodes at the top or bottom edge of the band. Note that a node knows its position in the band from its PIB. Each such node periodically performs a cost/benefit analysis of whether it should include its parent or children in the band, or whether it should remove itself from the band. The node will take such an action if by doing so, it can reduce the total number of query and registration messages received by the system. A key property of the adaptation algorithms is that adaptation decisions can be made based on local information. There are four possible adaptation actions: Top Expansion, Bottom Expansion, Top Reduction, and Bottom Reduction.

Table 4.1: Classification of $Q : [s, e]$ for Top Expansion

Type of query	Range property
Large query	$[s, e] \supseteq [s_0, e_0]$
Left query	$[s, e] \subseteq [s_1, e_1]$
Right query	$[s, e] \subseteq [s_2, e_2]$
Middle query	$[s, e] \cap [s_1, e_1] \neq \phi$ AND $[s, e] \cap [s_2, e_2] \neq \phi$

Top Expansion (TE)

A node at the top of the band periodically evaluates whether including its parent node in the band will reduce the overall cost. With the parent in the band, the cost of future queries with a large range will be reduced, since they will be sent to the parent by the query algorithm. However, including the parent also means an increase in the registration cost, since future registrations must be sent to both the child and the parent level. When the percentage of large queries received by this node exceeds a threshold T_{large}, the decrease of query cost will outweigh the increase of registration cost, and the node will expand the band to include the parent by duplicating its contents at the parent. To ensure consistency, the node will request its sibling to duplicate as well.

There are two steps involved in choosing a proper T_{large}. We consider two sibling nodes $N_1[s_1, e_1]$ and $N_2[s_2, e_2]$ on the top edge of a band as shown in Figure 4.13. First, N_1 and N_2 must gather query and registration load information. Second, they calculate locally whether including the parent N_0 will reduce the overall query and registration cost. We now describe these two steps in more detail.

1. Gather load information

To help make adaptation decisions, each node maintains statistics on the type of queries it receives. Each query arriving at N_1 or N_2 is classified into one of the following categories: *Large Query, Left Query, Right Query and Middle query* (Table 4.1). Suppose a query's range is $[s, e]$. If it is fully contained within $[s_1, e_1]$, then it is a left query; if it is fully contained within $[s_2, e_2]$, then it is a right query; if it intersects with both, then it is a middle query; if it covers both, then it is a large query. N_1 and N_2 also maintain statistics on the number of registrations they receive.

In addition to the local information, N_1 and N_2 periodically exchange their load information. In particular, N_1 must inform N_2 of the number of left queries it receives, and N_2 must inform N_1 of the number of right queries it receives. Similarly, they also exchange the number of registrations they received over the last time interval. With this information, N_1 or N_2 can independently compute the percentage of each type of queries and registrations they receive over the last time interval Δ. The variables are summarized in Table 4.2.

2. Determine T_{large}

Suppose during a time interval Δ, queries and registrations arrive at node N_1 and N_2 with rates r_q and r_{reg} respectively. Assume the rates remain the same in the next time period, then nodes N_1 and N_2 can compare the cost difference with and without the TE. With top expansion, the parent node N_0's LBM would have a size of (P_0, R_0), and N_1 and N_2's LBM would have a size of (P_1, R_1') and (P_2, R_2') respectively. N_1 (or N_2) can infer the sizes of these three nodes based on the query algorithm as follows.

Table 4.2: Variables for Top Expansion

Variable	Meaning
n_{left}	percentage of registrations on N1
n_{right}	percentage of registrations on N2
p_{large}	percentage of large queries
p_{left}	percentage of left queries
p_{right}	percentage of right queries
p_{middle}	percentage of middle queries
P_1	The number of partitions of N1
P_2	The number of partitions of N2
P_0	The number of partitions of N0 after TE
R_1	The number of replicas of N1
R_2	The number of replicas of N2
R_0	The number of replicas of N0 after TE
R'_1	The number of replicas of N1 after TE
R'_2	The number of replicas of N2 after TE

With and without the expansion, the cost of left, right and middle queries do not change. For example, a query belongs to N_1 will always be sent to N_1 and that takes P_1 messages. However, for large queries, with expansion, they will be sent to the parent node N_0, since $P_0 \leq P_1 + P_2$. We summarize the cost with and without TE for each type of query in Table 4.3. For the registration cost, each registration, besides being sent to either N_1 and N_2's matrix, will also be sent to N_0. We summarize registration costs in Table 4.4.

Table 4.3: The cost of different queries with and without Top Expansion

	Without TE	With TE
Left query	P_1	P_1
Right query	P_2	P_2
Middle query	$P_1 + P_2$	$P_1 + P_2$
Large query	$P_1 + P_2$	P_0

Table 4.4: The cost of different registrations with and without Top Expansion

	Without TE	With TE
Left registrations	R_1	$R'_1 + R_0$
Right registrations	R_2	$R'_2 + R_0$

Due to the redistribution of load after TE, the corresponding matrix sizes of N_0, N_1 and N_2 would also change. The relationships between the sizes are determined as follows. We know

$$P_1 = \lceil \frac{n_{left} r_{reg}}{C_{reg}} \rceil, P_2 = \lceil \frac{n_{right} r_{reg}}{C_{reg}} \rceil, P_0 = \lceil \frac{r_{reg}}{C_{reg}} \rceil$$

and thus,

$$P_1 + P_2 - P_0 = 1 \tag{4.11}$$

Similarly,

$$R_1 = \lceil \frac{(p_{left} + p_{middle} + p_{large})r_q}{C_q} \rceil, R_1' = \lceil \frac{(p_{left} + p_{middle})r_q}{C_q} \rceil, R_0 = \lceil \frac{p_{large} r_q}{C_q} \rceil$$

and thus,

$$R_1' + R_0 - R_1 = 1 \tag{4.12}$$

For the same reason, we have:

$$R_2' + R_0 - R_2 = 1 \tag{4.13}$$

N_1 can now compute the total cost for all registrations and queries that arrive at these three nodes. The total number of messages needed equals the sum of query messages and registration messages:

$$M = M(query) + M(reg) \tag{4.14}$$

- Without TE, the number of query messages:

$$\begin{aligned} M(query) &= M(left - query) + M(right - query) + M(middle - query) + M(large - query) \\ &= r_q \Delta p_{left} P_1 + r_q \Delta p_{right} P_2 + r_q \Delta p_{middle}(P_1 + P_2) + r_q \Delta p_{large}(P_1 + P_2) \end{aligned}$$

  The number of registration messages:

$$\begin{aligned} M(reg) &= M(left - reg) + M(right - reg) \\ &= r_{reg} \Delta n_{left} R_1 + r_{reg} \Delta n_{right} R_2 \end{aligned}$$

- With TE, the total number of messages $M' = M'(query) + M'(reg)$, where,

$$M'(query) = r_q \Delta p_{left} P_1 + r_q \Delta p_{right} P_2 + r_q \Delta p_{middle}(P_1 + P_2) + r_q \Delta p_{large}(P_0)$$
$$M'(reg) = r_{reg} \Delta n_{left}(R_1' + R_0) + r_{reg} \Delta n_{right}(R_2' + R_0)$$

TE is beneficial, when $M' < M$ holds, or:

$$r_q \Delta p_{large}(P_0) + r_{reg} \Delta n_{left}(R_1' + R_0) + r_{reg} \Delta n_{right}(R_2' + R_0) <$$
$$r_q \Delta p_{large}(P_1 + P_2) + r_{reg} \Delta n_{left} R_1 + r_{reg} \Delta n_{right} R_2$$

Simplify and we have

$$r_q p_{large}(P_1 + P_2 - P_0) > r_{reg} n_{left}(R_1' + R_0 - R_1) + r_{reg} n_{right}(R_2' + R_0 - R_2) \tag{4.15}$$

Substitute Eq. 4.11 4.12 4.13 into above, we have,

$$r_q p_{large} > r_{reg} n_{left} + r_{reg} n_{right} = r_{reg}. \tag{4.16}$$

or $p_{large} > \frac{r_{reg}}{r_q}$. This means that TE will reduce the overall number of messages coming to this subtree, when the rate of Large Queries exceeds the registration rate. Therefore, N_1 can set $T_{large} = r_{reg}/r_q$. As an example, if the query rate is 5 times the registration rate, then $T_{large} = 20\%$, and when N_1 observes the percentage of large queries exceeds 20%, N_1 will expand the band upwards to include N_0.

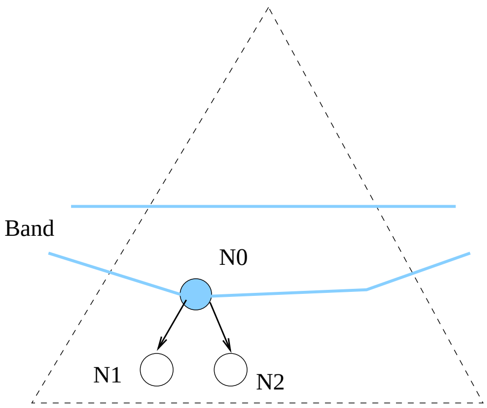


Figure 4.14: Band bottom expansion action. Node N_0 in at the bottom edge of the current band. Expand to include N_1 or N_2.

Top Reduction (TR)

This is the reverse action of TE. A node at the top of the band may remove itself from the band to reduce future registration cost. Future queries destined to it will have to go to its children, and from a child node's point of view, these queries are Large Queries. If the percentage of these queries becomes smaller than the threshold T_{large}, the parent node would leave the band by informing its corresponding matrix's head node to set the matrix size to zero. This information will be propagated to other nodes during the next round of PMP messages.

The way that a parent node sets the threshold T_{large} for top reduction is similar to how T_{large} is set for top expansion. We again use the example in Figure 4.13, but now N_0 is in the top edge of the band, and N_1 and N_2 are also in the band. N_0 considers the load received by these three nodes. To compute the variables in Table 4.2, N_0 must periodically get load information on N_1 and N_2. Once the load information is collected, the derivation is the same as Step 2 in Top Expansion, and $T_{large} = r_{reg}/r_q$. N_0 removes itself from the band if the percentage of the large queries becomes smaller than T_{large}.

Bottom Expansion (BE)

Similarly, a node at the bottom of the band evaluates the benefit of including its children into the band. By doing so, the cost of future partial queries (queries that belong to one child's range) will be reduced. For example, a query whose range belongs to the left child's range would be sent to the left child, which typically has fewer partitions than the parent. Of course, the cost of expansion is an increase in registration cost. When the percentage of partial queries exceeds certain thresholds, the decrease of query cost will outweigh the increase of registration cost, and the node will expand the band to include its left child, right child or both. The node will then send the corresponding children part of its own registrations, so that they can answer future queries. It is important to note that the BE occurs only if the node receives high load. In other words, it corresponds to a large matrix (e.g., more than 2 partitions or replicas), since only then, expanding to include children can

Table 4.5: Query classification for Bottom Expansion

Type of query	Range property
Left partial query	$[s_i, e_i] \subseteq [s_1, e_1]$
Right partial query	$[s_i, e_i] \subseteq [s_2, e_2]$
Middle partial query	$[s, e] \cap [s_1, e_1] \neq \phi$ AND $[s, e] \cap [s_2, e_2] \neq \phi$
Exact query	$[s, e] = [s_0, e_0]$

Table 4.6: Variables for Bottom Expansion

Variable	Meaning
n_{left}	percentage of registrations belong to N1
n_{right}	percentage of registrations belong to N2
p_{left}	percentage of left partial queries
p_{right}	percentage of right partial queries
p_{middle}	percentage of middle queries
p_{exact}	percentage of exact queries
P_0	The number of partitions of N0
R_0	The number of replicas of N0
P_1	The number of partitions of N1 after BE
R_1	The number of replicas of N1 after BE
R'_0	The number of replicas of N0 after BE

reduce the cost.

The way a node chooses the proper thresholds to conduct bottom expansion is similar to how the threshold is chosen in top expansion. There are also two steps. We consider node $N_0[s_0, e_0]$ on the bottom edge of a band as shown in Figure 4.14. N_0 evaluates whether it needs to add its two children $N_1[s_1, e_1]$ and $N_2[s_2, e_2]$ to the band.

1. Gather load information

N_0 classifies the queries it receives into the categories listed in Table 4.5. Note that this classification is different from the one we discussed in the Top Expansion. N_0 also maintains statistics on the types of registrations it receives, e.g., how many registrations belong to its left child's range. Unlike in top expansion, N_0 does not need to exchange load information with other nodes. For any time interval Δ, it can compute all the variables in Table 4.6.

2. Determine thresholds.

Suppose during a time interval Δ, the query rate N_0 observes is r_q and the registration rate is r_{reg}, and assume the rates remain the same in the next time interval. With BE, N_0, N_1 and N_2's LBMs would have a size of $(P_0, R'_0), P_1, R'_1)$ and (P_2, R'_2) respectively. N_0 can infer these matrix sizes based on the query algorithm, and can then compare the cost difference before and after the BE.

In particular, we look at how T_{left} is set. Choosing T_{right} and T_{middle} is similar. Suppose N_1 is added to the band, and based on the query algorithm, left partial queries will be sent to N_1, and all other queries will remain on N_0. Registrations that belong to the N_1 will be sent to both N_0 and N_1, and after the expansion, the size of N_0 becomes P_0, R'_0, and N_1's

Table 4.7: Query cost with and without Bottom Expansion

	Without BE	With BE
Left partial query	P_0	P_1
Right partial query	P_0	P_0
Middle partial query	P_0	P_0
Exact query	P_0	P_0

Table 4.8: Registration cost with and without Bottom Expansion

	Without BE	With BE
Left registrations	R_0	$R_1 + R_0'$
Right registrations	R_0	R_0'

size becomes P_1, R_1. The cost change is listed in Table 4.7 and Table 4.8.

The relationship between the matrix sizes are as follows. We know

$$P_1 = \lceil \frac{n_{left} r_{reg}}{C_{reg}} \rceil, P_0 = \lceil \frac{r_{reg}}{C_{reg}} \rceil$$

and thus, $P_0 - P_1 \geq 1$ (assuming $n_{left} \neq 1$). Similarly,

$$R_0 = \lceil \frac{r_q}{C_q} \rceil, R_1 = \lceil \frac{p_{left} r_q}{C_q} \rceil, R_0' = \lceil \frac{(1 - p_{left}) r_q}{C_q} \rceil$$

and thus, $R_1 + R_0' - R_0 = 1$.

N_0 computes the cost change as follows. Without N_1, the total number of messages is:

$$\begin{aligned} M &= M(query) + M(reg) && (4.17) \\ &= r_q \Delta P_0 + r_{reg} \Delta R_0 && (4.18) \end{aligned}$$

After $N1$ is added, the total number of messages is $M' = M'(query) + M'(reg)$, where the number of query messages:

$$M'(query) = r_q \Delta p_{left} P_1 + r_q \Delta (1 - p_{left}) P_0$$

and the number of registration messages is:

$$M'(reg) = r_{reg} \Delta n_{left} (R_1 + R_0') + r_{reg} \Delta n_{right} R_0'$$

To make sure BE is beneficial, the condition $M' < M$ must be met, or

$$r_q \Delta p_{left} P_1 + r_{reg} \Delta n_{left} (R_1 + R_0') + r_{reg} \Delta n_{right} R_0' <$$

$$r_q \Delta p_{left} P_0 + r_{reg} \Delta n_{left} R_0 + r_{reg} \Delta n_{right} R_0$$

Simplify, we have,

$$\begin{aligned} r_q p_{left} (P_0 - P_1) &> r_{reg} n_{left} (R_1 + R_0' - R_0) + r_{reg} n_{right} (R_0' - R_0) \\ &> r_{reg} n_{left} (R_1 + R_0' - R_0) + r_{reg} n_{right} (R_1 + R_0' - R_0) \\ &= r_{reg} (R_1 + R_0' - R_0) \end{aligned}$$

Finally, substitute, we have,

$$p_{left} > \frac{r_{reg}}{r_q(P_0 - P_1)} \tag{4.19}$$

N_0 then set the threshold $T_{left} = \frac{r_{reg}}{r_q(P_0-P_1)}$. As an example, if $P_0 - P_1 = 1$, the above becomes $p_{left} > \frac{r_{reg}}{r_q}$. Intuitively, if only a small portion of the registrations belong to the left child, with a small percentage of left queries, it is beneficial to add N_1. Using the same deriving method, thresholds can be set for adding N_2.

Bottom Reduction (BR)

Bottom reduction is the reverse action of bottom expansion. A node at the bottom edge of the band may remove itself to reduce registration cost. Partial queries' cost may increase due to bottom reduction, since they now have to be sent to the corresponding parent node. To ensure the BR is beneficial, the percentage of partial queries must become smaller than thresholds T_{left}, T_{right} and T_{middle}.

We again use the example shown in Figure 4.14, but now along with N_0, N_1 and N_2 are also in the band. In order for N_1 to choose a proper T_{left}, it must exchange load information with N_0 and N_2 to compute the variables in Table 4.6. Once these variables are computed, the following derivation is the same as above, and $T_{left} = \frac{r_{reg}}{r_q(P_0-P_1)}$. N_1 removes itself from the band, if the percentage of left partial queries becomes smaller than T_{left}. T_{right} and T_{middle} can be similarly set.

4.5 Analysis of the Dynamic Mechanisms

4.5.1 Overhead of PMP

The PMP protocol plays a crucial role in supporting range queries efficiently. We now examine the overhead more carefully. From a node's point of view, in each round of PMP message exchange, it sends at most 1 PR and 2 PRR messages, and receives at most 2 PR and 1 PRR messages. Hence, no node will be overwhelmed and no system bottleneck is created. From the system's point of view, the overhead of the protocol is determined by two factors: (1) PMP message size and (2) PMP message exchange frequency. We next examine these two factors.

The PMP messages are reasonably small. In the first round of message exchange, a PR message carries the LBM's size (2 integers) for each node in its subtree, so the message size is $O(n)$, for a subtree that has n leaves. As an example, suppose an attribute can take on 200 distinct values, and its RST will have at most 200 leaves, the largest PR message is the one sent to the root, and it has a size of $\sim$ 1600 bytes (= 200 nodes * 8 bytes/node). For the same tree, the PRR messages are even smaller with a size of $O(\log n)$ corresponding to the height of the tree, since it contains only the path component. In the above example, the largest PRR messages are the ones to the leaves; they have a size of $\sim$ 70 bytes (= $\log 200$ nodes * 8 bytes/node). As an optimization, the PR and PRR message sizes can be further reduced in future PMP rounds by only including matrices whose sizes have changed.

The frequency of the periodic PMP message exchange can be set fairly low. When the load in the system is stable, the band does not change. The band may change when the matrices' size change due to significant load change, or when adaptation actions are needed due to changes in query ranges. Both of these will typically occur on a much larger time scale in comparison to the registration or query rate.

In conclusion, since the PMP message size is small and the PMP message frequency is set low, the overhead incurred by the PMP protocol is reasonably small.

4.5.2 Overhead of Band Adaptation

Band adaptation actions are carried out in a distributed fashion, since each decision is made by a node based on local information. To set the proper thresholds for the various actions, a node may need additional load information such as query and registration rate from its sibling or parent, but this remains a localized operation. By combining expansion and reduction actions, the band can move up and down the RST depending on the load. An additional advantage that band adaptation brings is that we no longer need a well-defined leaf level to support range queries. This is important because in many applications it may not be possible to predefine a fixed smallest granularity. For example, when the domain is real numbers, query ranges may be arbitrarily small. With band adaptation, the tree can grow downwards as deep as is needed to handle small range queries.

The overhead of band adaptation is minimal since the decisions are made locally and no network wide message exchange is required. During band adaptation, an endpoint may get stale information about the tree. The impact of this transient state is small and temporary. For example, stale information may result in some additional, unnecessary registrations, or cause an endpoint to repeat its query if the node it originally contacted left the band. The system will go back to normal state after next round of PMP message exchange.

4.5.3 Band Stability Analysis

The band adaptation mechanism coupled with the PMP protocol ensures that the band may shift up and down in the RST depending on the load and query ranges. However, it is important to carefully examine the band stability property, since a band that is constantly changing its shape will introduce significant overhead due to frequent data replication in the overlay network.

The main requirement for stability is that given a stable load, the edges of the band should remain fixed. To show that our algorithms meet this requirement, we assume that after a band adaptation action, the load becomes stable. We examine the following four cases: (i) a node is added via Top Expansion will not be removed via Top Reduction; (ii) a node is removed via Top Reduction will not be added via Top Expansion; (iii) a node is added via Bottom Expansion will not be removed via Bottom Reduction; and (iv) a node is removed via Bottom Reduction will not be added via Bottom Expansion.

We first look at cases (i) and (ii). From the previous section, we know that the conditions for top expansion ($p_{large} > T_{large}$) and top reduction ($p_{large} < T_{large}$) are exactly the opposite of each other. For any given load, p_{large} is either larger than or smaller than T_{large}, and it cannot be both. Therefore once a node is added to the top, it will not be removed if the load does not change. Similarly, if it is removed due to top reduction, the criteria of adding it back will not be met if the load does not change. The same argument can be applied to cases (iii) and (iv).

In practice, we do not want the band to change due to small changes in the load. This can be easily achieved by slightly modifying the conditions for expansion and reduction to introduce some hysteresis. For example, in the case of top expansion and top reduction, we can change the thresholds as follows. A node is added to the top if the rate of large queries exceeds the threshold $T_{large} + \delta$; and a node is removed if the rate of large queries drops below $T_{large} - \delta$, where δ is a configurable value.

In addition to the above scenario, we note that if the load itself oscillates around a threshold, it is possible that a node may be added or removed frequently. To prevent this from happening, we can add addition rules to the band adaptation actions. For example, after a top expansion, a top reduction can occur only if the percentage of large queries drop below T_{large} for a certain period of time.

In summary, our band adaptation algorithms, on the one hand, can adapt the band to load changes, and on the other hand, also ensures a stable band if the load changes are not significant or become stable.

4.6 Evaluation

We now evaluate the effectiveness of the RST-based range search mechanism. We implemented both the static and dynamic mechanisms in the CDS simulator we described in Section 3.4.1.

4.6.1 Methodology

The CDS configuration for the simulations conducted in this section is the same as what was used in Section 3.4.2. To test the range search mechanisms, the workloads we used here are different from before. Each registration load is comprised of a set of content names, each consisting of one AV-pair. The AV-pairs share the same attribute, a, which can take on 200 different values. We chose 200 since it is a typical range for attributes such as speed, temperature, etc. This attribute's RST has 9 levels ($\lceil \log 200 \rceil + 1$); recall that the leaf level is Level 0, and the root level here is Level 8. The sender of a name is selected randomly from the nodes in the system and the names' arrival times are modeled with a Poisson distribution. Query loads are similar, except that instead of one value, a range may be specified. In addition to the **Static RST** and **Dynamic RST** designs described in Sections 4.2 and 4.4, the evaluation also considers the following algorithms:

- **Root Only**: All registrations and queries are sent to the root level nodes, i.e., there is no RST.
- **Leaf Only**: Registrations are sent to leaf nodes only, and queries are decomposed into point queries.
- **RST(3)**: Similar to Static RST, but we use Lemma 3 and decompose a range with length R_q into three adjacent ranges at level $\lfloor \log R_q \rfloor$ in the RST.

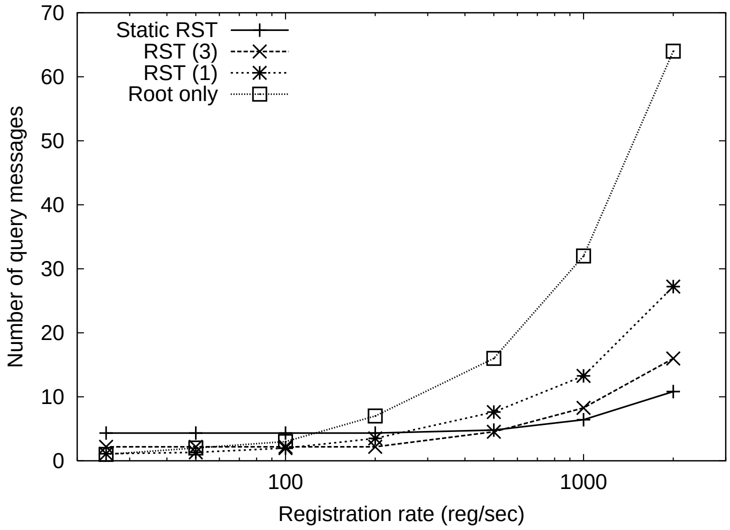


Figure 4.15: Query cost comparison.

- **RST(1)**: A query is always sent to the node in the RST that corresponds to the common prefix of the query range's two end values. In this case, even a small range can translate to a high level node, e.g., range $[3, 4]$'s prefix node is the root $[0, 7]$ in Figure 4.1.

We use the number of messages needed for registrations and queries as the primary performance metric.

4.6.2 Performance of Static RST

We start by examining the query and registration performance of the static RST design.

Query Performance

We first evaluate the query performance. In each experiment, we inject a registration load into the system with a certain rate, and then inject a query load with rate $200q/sec$. The registration rate varies from $20reg/sec$ to $2000reg/sec$. In the query load, there are 10,000 random queries but all have a range of $R_q = 20$. We compute the average number of query messages needed for a query after each run. We plot the results in Figure 4.15 as a function of the registration rate.

When the registration load is low, Static RST uses the most query messages due to its logarithmic decomposition. As the registration load increases, nodes higher in the tree will start to create partitions. As expected, the cost in Root Only grows linearly as partitions are created proportionally to the registration load. Since RST(1) also may use levels higher in the tree, its cost grows fast as well and it becomes more expensive than Static RST. Static RST also performs better than RST(3) under high load, since its decomposition is finer and uses more lower level nodes.

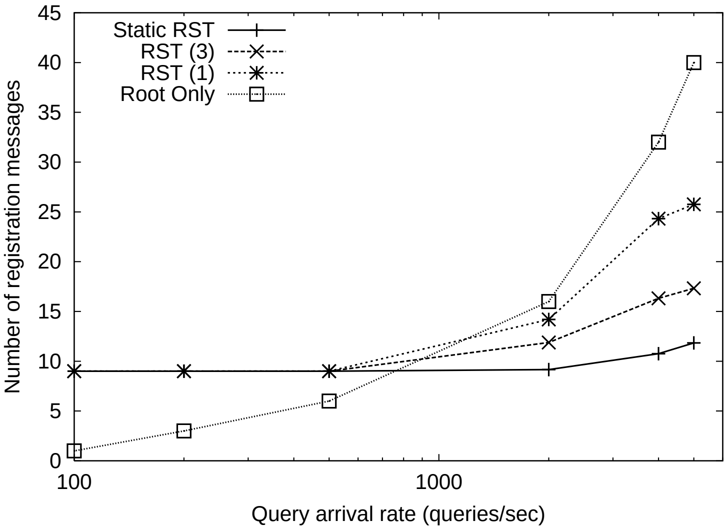


Figure 4.16: Registration cost comparison.

Registration Performance

Next we examine the registration performance. In this set of experiments, we fix the registration rate and vary the query rate from $100q/sec$ to $5000q/sec$, and the query range $R_q = 20$. Figure 4.16 shows the average number of registration messages needed for each registration as the query rate increases.

For low query load, all the RST cases use more registration messages than Root Only, because they have to register with all 9 levels. As the query load increases, the cost of Root Only grows linearly and it becomes the worst performer. Static RST performs the best. The reason is that for high query load, the registration cost of $\{a = v\}$ is dominated by the number of replicas along $Path(v)$. Recall from Section 4.2.3 that the *relevance* of an algorithm indicates how well a query matches the nodes that are being queried. The *relevance* of these algorithms are in the following order:

$$r_{RootOnly} \le r_{RST(1)} \le r_{RST(3)} \le r_{LeafOnly} = r_{StaticRST} = 1.$$

For low relevance algorithms such as Root Only and RST(1), queries are concentrated at a small number of nodes with large ranges and this causes them to replicate often. As a result, the registration cost increases rapidly, since a registration must go to all replicas. In comparison, in Static RST ($r = 1$), the query load is spread out to nodes in many lower levels, and the number of replications in the system is minimized, which translates into a low registration cost. Between RST(3) and Static RST, RST(3) has a smaller relevance and results in more replications under high load, and as show in Figure 4.16, its performance is also worse than Static RST.

In summary, the Static RST mechanism provides the best performance for both queries and registrations under high load. However, when the load is low, it is suboptimal in that it is worse than the Root Only algorithm. These results are consistent with our analysis in Section 4.3.

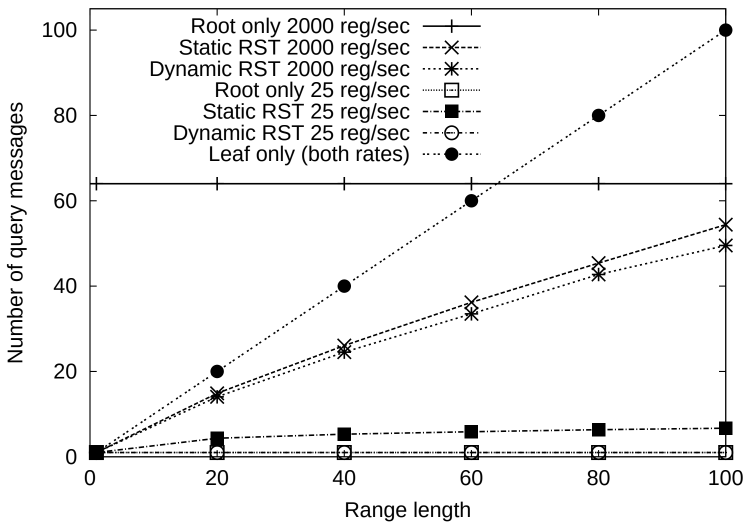


Figure 4.17: Query cost vs. Range length.

4.6.3 Performance of Dynamic RST

Next, we evaluate the performance of the Dynamic RST design. For the results in this section, we use the root level as the default band, and we take measurements after the necessary band adaptations have completed. We first evaluate the query performance, which depends on two factors: the query range length and the registration load. We then evaluate the registration performance, which depends on the query load.

Query Performance

1. Query Performance for queries with different ranges

Figure 4.17 shows the average number of query messages as a function of the query range. In this set of experiments, we first inject a registration load into the system and then issue a query load with rate $200q/sec$. In each experiment, the range lengths are the same, and across experiments, the range varies from from 1 (point query) to 100 (50% of the domain). We use two registration loads with low ($25reg/sec$) and high arrival rates ($2000reg/sec$).

With low registration load, the root level (and all other levels, if they are used, e.g., in the Static RST) has only 1 partition. The Root Only and Dynamic RST designs perform the best, since they will just use the root for all queries. The Static RST design ignores the load status and always decomposes the query into logarithmic number of sub-queries, so the number of query messages grow logarithmically with the range length. The Leaf Only is the worst algorithm, and the cost grows linearly with the range.

Under high registration load, Root Only performs poorly, since irrespective of the range length, 64 query messages are needed for all queries (the root level has 64 partitions under this load). In the Leaf Only case, the number of query messages again grows linearly with the range length, since no partitions were created at the leaf level. The Static RST approach grows faster than logarithmic due to the partitions created at higher levels, but it

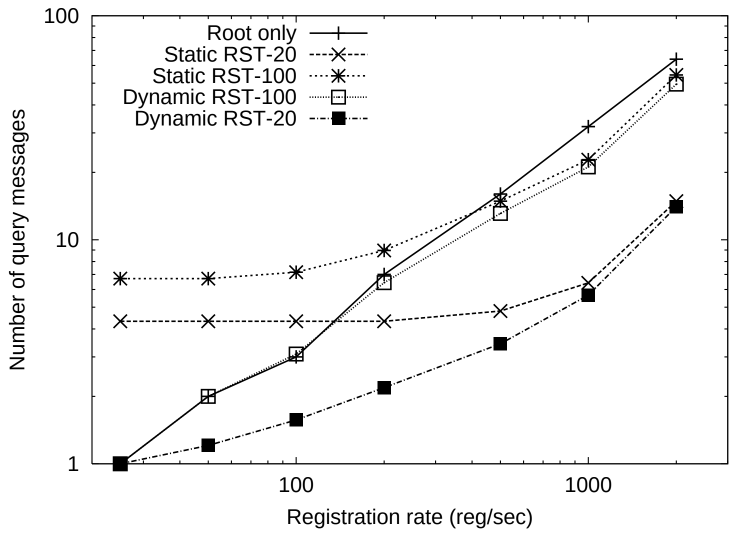


Figure 4.18: Query cost vs. Registration load.

still requires far fewer query messages than Root Only. The Dynamic RST design improves performance further since it does not need to decompose the query all the way to the leaf level. Of course, if the query range is the full domain, both Static RST and Dynamic RST will degenerate into the Root Only, since using the root level nodes to resolve a full domain query is the most efficient way.

2. Query performance vs. registration load

Using the same setup, Figure 4.18 compares the query performance as the registration load increases. We use two query loads, with a range length of 20 and 100 respectively. This plot is in log-log scale to amplify the differences for low loads. Dynamic RST does the best in all cases. In particular, it tracks the Root Only case when the load is low by using high level nodes and by avoiding unnecessary decomposition. It migrates towards and stays under the Static RST curve as the load increases.

Registration Performance

Figure 4.19 compares the average number registration messages needed as query load increases. The experimental setup is the same as what was used for Figure 4.16 in the previous section, and the query range is 20. The two curves corresponding to Static RST and Root Only are taken directly from Figure 4.16. Dynamic RST performs the best for both low and high query load. When the query load is low ($100q/s$), the root node can resolve all queries without having to replicate, and the band stays at the root level. As a result the cost for a registration is 1 and it is sent to the root level. Dynamic RST behaves the same as the Root Only case. In contrast, in the Static RST, a registration has to go to all 9 levels regardless of the query load. As the query load increases, the root level node starts to replicate, and the cost of Root Only increases linearly. However, for the Dynamic RST, the band grows downwards when top level node observes high load, and as a result, queries

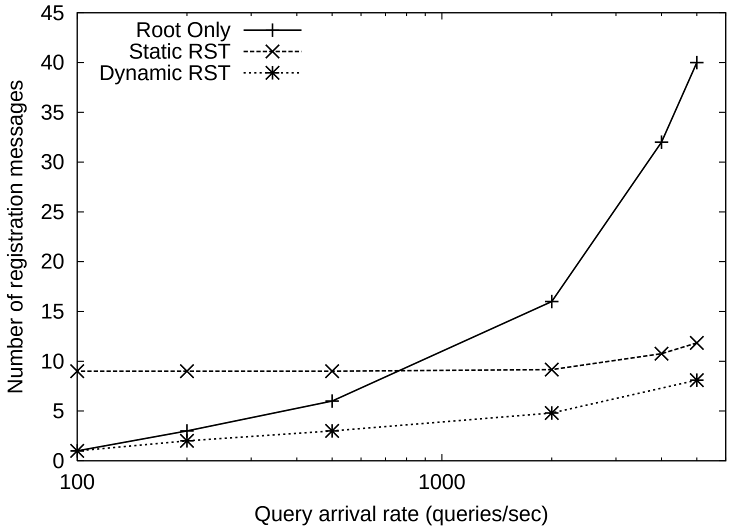


Figure 4.19: Registration cost comparison. Query range = 20

will choose lower level nodes in the band. Top level nodes will leave the band when they do not receive enough queries (Top Reduction). For example, when query rate is $2000q/s$, only Levels 2, 3 and 4 are in the band, and the registrations are sent only to these levels. For this reason, Dynamic RST performs better than Static RST for high query load as well. If the query load is further increased and query ranges are distributed over the full domain, all the levels will be required for the Dynamic RST, and at that time, it will perform the same as Static RST.

4.6.4 System Optimization with Band Adaptation

We examine in more detail how band adaptation improves the system's performance. We first show how bottom expansion helps to reduce query cost, where we also examine how different PMP update periods affect the performance. We then use "flash crowd" type of query load to show different band adaptation actions work in concert to optimize the band shape depending on the load. We further show that the band of an RST can become stabilized under load that consists of various ranges.

Effect of Bottom Expansion

In this set of experiments, the root level is chosen as the default band. The registration load is high with an arrival rate of 500 reg/sec. All the registrations are initially sent to the root level (Level 8), and it creates 16 partitions to accommodate this registration rate. We use two query loads with rates $r_q = 200q/s$ and $r_q = 20q/s$. Within each load, all queries have the same range length ($R_q = 20$).

We run experiments with different PMP update period T. "$T = 0ms$" means that when a node is included in the band, it sends a *triggered* PR/PRR message to its parent or children right away without waiting for the next update period. When T is non-zero, the

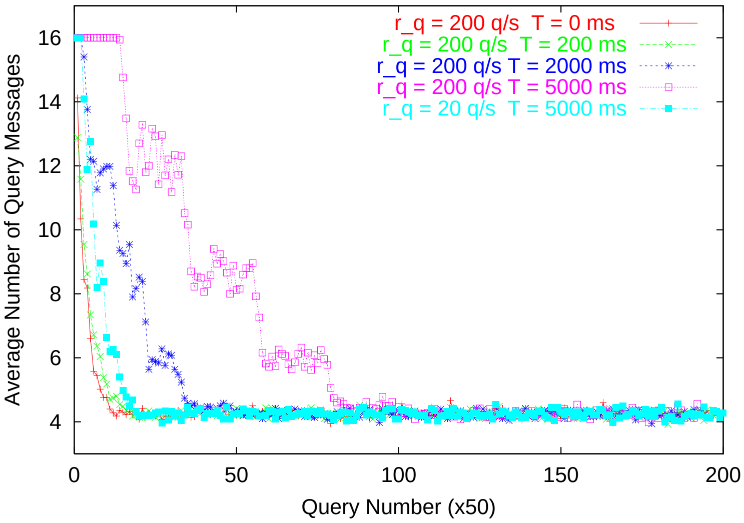


Figure 4.20: Band bottom expansion reduces query cost.

node waits until the next update cycle to propagate this information. We plot the average number of query messages needed to resolve a query versus the query index in Figure 4.20. Each data point corresponds to the average cost of 50 queries.

We first compare the four scenarios corresponding to $r_q = 200q/s$. At the beginning, since only the root level is in the band, all queries are sent to all partitions at the root level, and the query cost is 16. Since the root level has many partitions and it receives many small queries ($R_q << 200$), the root level expands the band downwards.

As the band grows downwards, the local query algorithm on query initiating nodes decide it is beneficial to send queries to lower level nodes in the band. As shown in Figure 4.20, for all scenarios, the query cost decreases significantly. For this load and query range, eventually Level 4 is included in the band and the query cost stabilizes at around 4, i.e., 4 query messages to Level 4 nodes are needed to resolve each query. The difference between these scenarios is the time when queries start to benefit from band adaptation. The triggered update scenario ($T = 0$) propagates the dynamic band information the fastest, and queries start to benefit early on. When T increases, band changes at a slower pace, and the cost of queries arriving before the update cycle does not decrease.

We examine the scenario of $r_q = 200q/s$, $T = 5000ms$ more closely: it has 5 distinct steps. During the first $5000ms$, about 1000 queries arrive at the system. Since Level 7 will not be included into the band until after $5000ms$, the first 1000 queries do not benefit from the band expansion and their cost remains at 16. After Level 7 is included in the band, queries are sent to Level 7 and their cost drops. Similarly, only queries arrive after another $5000ms$ can benefit from the inclusion of Level 6. Eventually after Level 4 is included, the band adaptation stops and the query cost is optimized to the lowest value. As a comparison, for $r_q = 20q/s$, with a low arrival rate, the long update cycle has less impact on reducing query's cost. The 5 adaptation steps are close to each other in Figure 4.20, since the band adaptation completes after about 1000 queries. From each query's point of view, the performance is comparable to the $r_q = 20q/s$, $T = 200ms$ case.

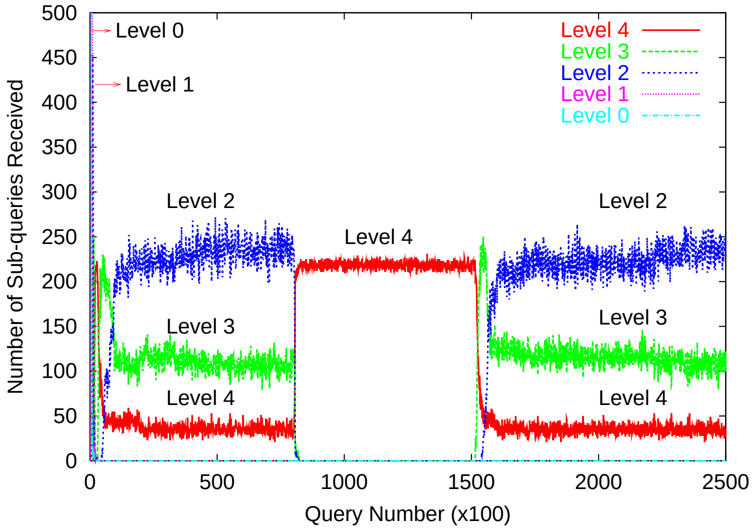


Figure 4.21: Band adaptation under flash crowd.

In summary, bottom expansion can help the system to improve its performance significantly. One factor that must be considered is how to select a proper PMP update cycle. With a small update cycle, the system reacts to the load change quickly, and thus can benefit queries quickly if the load is high. However, the cost of a small update cycle is that more PMP messages are required. If queries arrive at the system with a low rate, a larger update cycle may be sufficient for reducing the query cost, while incurring a low PMP overhead.

System Performance under Flash Crowd

In this experiment, the band initially only contains the lowest level (Level 0) due to previous point queries. The registration load is low in that even if all registrations are sent to the root level, it will only have one partition. We inject a query load with range $R_q = 20$ into the system. The arrival rate of queries changes from high ($2000q/sec$) to low ($200q/sec$) and finally to high ($2000q/sec$) again. This query load is used to emulate the "flash crowd" type of workload, where users may shift their query interest rapidly. The thresholds we used to trigger expansions are 20%.

Figure 4.21 shows the number of sub-queries received at each level for every 100 queries that are issued by endpoints. Correspondingly, Figure 4.22 shows the average number of query messages needed for every 100 queries. Both figures have three primary sections.

In the first section, Level 0 receives all the queries, since it is the only level in the band, and each query is resolved using 20 sub-queries. As more queries arrive, since these queries are Large Queries with respect to Level 0, the band expands to higher levels, and eventually Level 4 nodes (with a length of 16) are added to the band. Figure 4.23 zooms in on this section of Figure 4.21 to show when different levels are added to the band. As a result of top expansion, the query cost first drops to around 2 and then stabilizes at 3.8. This is because when Level 4 is first added, all queries are decomposed into 2 Level 4 level sub-queries, since this is the most efficient decomposition scheme. However, with the high load ($2000q/sec$), Level 4 nodes' LBMs have to replicate to handle all queries. Our query algorithm then directs the queries to use level 2 and 3 nodes in addition to level 4 nodes to avoid further overloading level 4 nodes. As a result the query cost increases logarithmically, but it prevents the registration cost from growing linearly.

In the second section, the query rate drops to $20q/sec$, and all LBMs shrink to only 1 replica. The query decomposition algorithm directs queries to Level 4, rather than further using Level 2 and 3. As shown in the figure, nodes in Levels 0-3 do not see enough queries, and they eventually drop out of the band. The band reduces to Level 4 only. At this time, the query cost drops to ~ 2 (using 2 Level 4 nodes for each query).

In the third section, the query rate increases to $200q/sec$ again, and the band expands downwards through bottom expansion to recruit Levels 3 and 2 back.

Band Stability

In this experiment, the registration load is the same as above and stays low, and the initial band consists of only the leaf level. The query load contains 10,000 queries, and the query ranges vary from 1 to 100. More specifically 80% of the queries have a range of 20 and the rest 20% of the query ranges are distributed uniformly between 1 and 100.

Figure 4.24 shows the band adapatation process. Similar to before, the large ranges cause the band to expand upwards. However, we observe that the band becomes stable and consists of only levels 2,3 and 4 in the end. Even though there are ranges that are much larger than the range of Level 4 nodes (length equals 16), the band does not further expand upwards due to the small number of these ranges, since doing so would increase the overall cost registrations and queries.

Figure 4.25 shows the average cost of queries of different ranges. It can be seen that for queries with small ranges (< 20), the cost is low. For larger queries, since the band is not optimized for them, their cost is higher than the ideal case where the levels corresponding to their lengths are in the band. For example, for queries with range larger than 90, they need 8 messages rather than 3 messages, which would be the case were Level 6 in the band.

The experiments in this section show that the band adaptation algorithms can successfully adapt the band based on load changes to reduce query and registration cost.

4.7 Related Work

Efficiently supporting range queries in DHT-based systems was posed as an open question in [35, 38]. There have been some recent efforts in addressing this problem. In [54], the Prefix Hash Tree (PHT) is proposed to support range queries. While the PHT is conceptually similar to the RST, there are important differences between the two systems. Unlike our system, where we store contents and resolve queries using multiple levels depending on the load and query ranges, the PHT is a trie, and only leaf nodes store contents. Queries are first sent to the node corresponding to the common prefix of the range and traverse down to the leaves. In our system no tree traversal is needed, and our evaluation shows that the logarithmic query decomposition is more efficient than using the common prefix node.

In [8], the authors use space filling curves as hash functions in CAN-based DHT systems [55] to support range queries. In this work, a query is first sent to a node within the range; that node then locally broadcasts the query. Our system makes no assumption on the type of DHT used. Queries are sent to nodes that contain potential matches, and no broadcasting is involved. In [33], a mechanism based on locality sensitive hashing function is used for range queries in the context of relational databases. However, unlike our system, only approximate answers that are similar to users' range queries are returned.

SkipNet [36] proposed a lexicographic order preserving DHT, and thus allows data items with similar values to be placed on contiguous nodes. This facilitates range search, but the number of nodes must be visited is still linear to the query range due to lack of aggregation. P-Grid [5] is a DHT in which nodes are organized based on a virtual distributed search tree similar to our RST structure. The critical difference is that in P-Grid, the tree structure is used for DHT routing purpose to locate a node that holds a given key in the identifier space, and as such, like other DHTs, it does not directly support range queries.

In a different but similar context, Li *et al.* [44] proposed a distributed mechanism to partition a multi-dimensional space using a data structure similar to kd-trees to support range queries in sensor networks. Our system can be readily extended to high dimensions to support multi-dimensional range queries.

In traditional parallel databases, a large relation is often partitioned among multiple disks [21]. A partitioning technique that works well for both point and range queries is to partition the relation based on data values. A centralized partition vector must be consulted before a query may be issued. In our system, if we consider the collection of all content names as a large relation, the RST based mechanism mimics the value-based mechanism by partitioning the relation multiple times with different granularity. The advantage of our design is that queries can be resolved in a fully distributed fashion without using a centralized partition vector.

4.8 Chapter Summary

In this chapter, we presented an adaptive protocol to support range queries efficiently in the CDS. Our algorithm is based on a distributed data structure, the Range Search Tree (RST), for content registration and query resolution. Nodes at different levels of the RST represent different level of aggregation, and may be used to resolve queries with different range lengths efficiently. To ensure efficiency, we do not instantiate every node in the tree by default, instead, we only use a part of the tree, known as the *band* to accept registrations and resolve queries. The band changes its shape depending on the load it observes: A node may be recruited to join the band if its inclusion to the band can lower the overall registration and query cost. The band adaptation is conducted in a fully distributed fashion, since only local information is needed. To ensure endpoints can discover the band efficiently, we designed a lightweight protocol, the Path Maintenance Protocol, to update the band information in the tree. As a result, registrations and queries can still be carried out efficiently without having to traverse the tree, and thus no bottlenecks are created. Our extensive simulation demonstrates that the system can support range queries efficiently using an adaptive RST under both high and low load, while incurring low overhead.

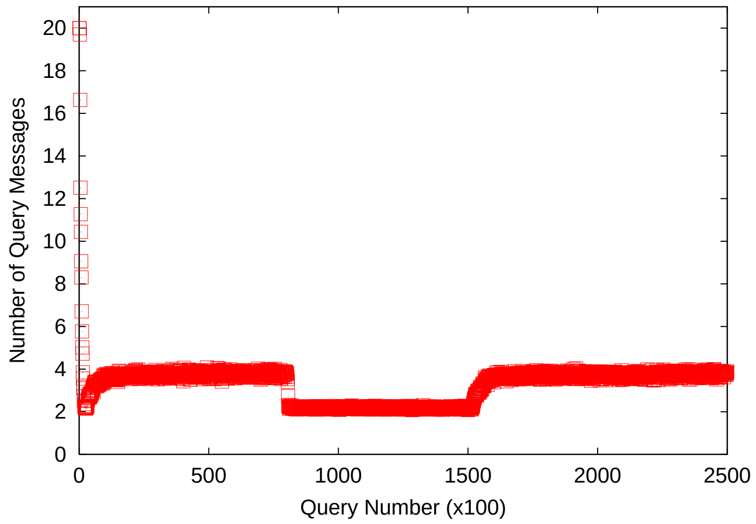


Figure 4.22: The query cost change as band adaptation occurs.

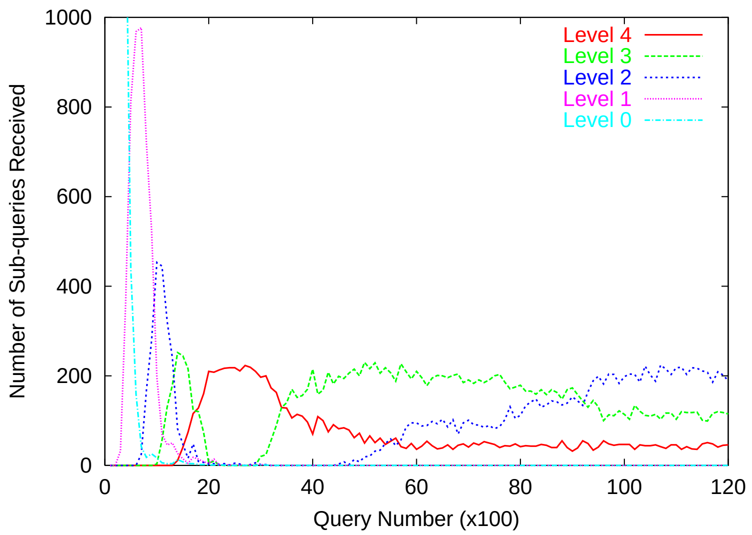


Figure 4.23: Zoom in on the top expansion effect.

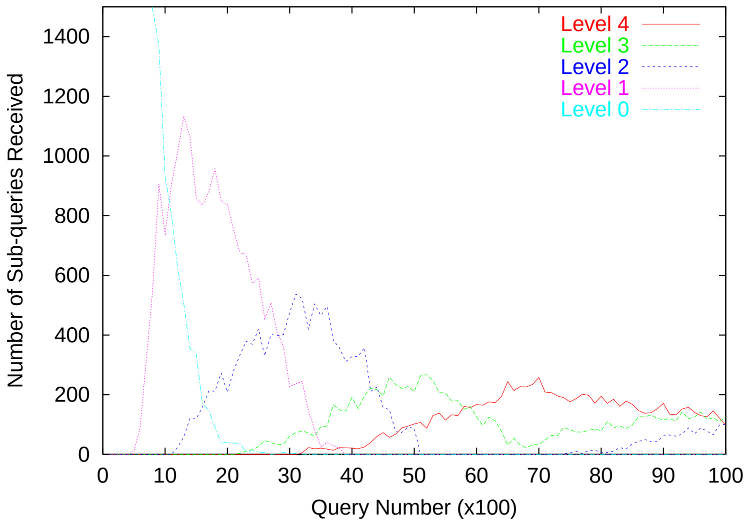


Figure 4.24: Band adaptation for mixed range lengths.

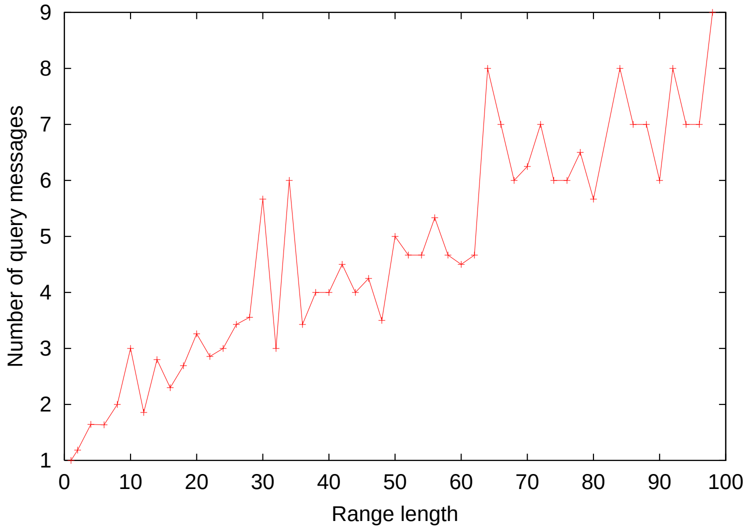


Figure 4.25: The average cost for queries with different range lengths.

Chapter 5

Supporting Similarity Queries

The CDS system we described so far supports range searches on individual attributes efficiently. For example, in the traffic monitoring service, it is possible to allow users to find all cameras that observe a low speed to identify jammed sections on highways. However, in many CDS applications, data are distributed in a multi-dimensional space, and queries in these applications are often more complex than one-dimensional range queries. For example,

- In pervasive computing, the geographical location of devices, such as cameras and sensors, may be represented using 3-d GPS coordinates.
- In a network service discovery systems such as NSSS [37], a network node on the Internet may represent its location in the network using a multi-dimensional coordinate (e.g., 5-d or 7-d) computed based on services such as GNP [48].
- In content-based information retrieval systems, multimedia contents such as images [42] or audio files (mp3) [28] are often represented using feature vectors that captures the key properties of the contents. A feature vector typically consist of multiple dimensions. In [28], a 30 dimensional vector is used to represent an mp3 file.

In these applications, *similarity queries* are widely used. For example, in the P2P music sharing system described in Section 2.3, each peer contributes a set of mp3 files. A user may issue a query to find songs that are "similar to a song that he recorded from the radio" [28]. The user would specify the feature vector of the recorded song and formulate that as a query to the CDS system. The CDS system will then try to find songs registered in the system whose feature vectors are close to the specified vector in the multidimensional vector space. Similarly, in a service discovery system, a user may want to find the closest proxy server for video transcoding [37] using GNP coordinates. The user will supply its own GNP coordinates to the CDS, and the system will then carry out the query and find servers that are close to the user's node in the coordinate space.

Conducting similarity searches on a large multidimensional dataset in a decentralized system, such as the CDS, is a difficult problem. In this chapter, we present efficient mechanisms employed by the CDS to support similarity searching in multidimensional data sets. The rest of the chapter is organized as follows. Next, we formally define the similarity search problem in the CDS environment and motivate our distributed kd-tree based design. In

Section 5.2, we describe a set of distributed protocols to build and maintain the distributed kd-tree data structure. In Section 5.3, we describe how endpoints carry out registrations and queries in our system. In Section 5.4, we present optimizations based on virtual node shrinking to improve the system's performance for high dimensional datasets. We present a simulation-based performance evaluation in Section 5.5. We discuss related work and summarize in Sections 5.6 and 5.7 respectively.

5.1 System Design for Similarity Search

Formally, suppose in an application, data are distributed in a k dimensional space and $d_1, d_2, ..., d_k$ are the k dimensions. The data points are in the set $S = \{p_i \in \mathcal{R}^k | i = 1, n\}$. Queries also have k dimensions, and can be considered as data points in the k-dimensional space. The similarity search problem is to find a data point $p \in S$ that is closest to a given query q, i.e.,

$$dist(p, q) = \min_{i=1,n} dist(p_i, q). \tag{5.1}$$

The $dist$ function can be defined using any Minkowski metric such as L_1 (Manhattan) or L_2 (Euclidean). A similarity search is also known as a nearest neighbor search (NN) in the literature, and we use both terms interchangeably. A generalization of the NN search is the KNN search, where K nearest neighbors instead of 1 are sought.

Nearest neighbor search in multidimensional databases has been a heavily studied area in the database community for more than thirty years [27]. For small data sets, the most efficient way to support similarity search is to use a central database that incorporates a tree-based indexing data structure to limit the amount of data that must be examined to resolve a query. Many indexing structures have been proposed, including the kd-tree [12], R^*-tree [11], the SR-tree [40] and many others. For larger datasets, the data may be partitioned across multiple databases running on different nodes. While this distributes the storage and registration overhead, queries have to be forwarded to each database. As such, the main drawback of this approach is that it does not scale with respect to query load, since each database has to handle all queries. In a distributed system such as the CDS, both the amount of available data and the number of queries can be very large. Supporting similarity searches in a fully distributed fashion in such a system is very challenging.

In Chapter 4, we presented how the CDS supports range searches efficiently. In particular, it facilitates range queries by deploying a distributed Range Search Tree, which partitions the 1-d data set at different levels of granularity. However, simply extending the RST to a multi-dimensional tree indexing structure will not work well because the size of the indexing data structure may become prohibitive without a careful design.

Additionally, the various tree structures generally perform well only for for data sets with a relatively small number of dimensions (< 10). For higher dimensions, however, due to the well-known "curse of dimensionality" problem, the performance of any tree-based indexing structures degrades significantly and the query overhead effectively becomes linear in the size of the database [73].

In this chapter, we present a distributed search architecture that is based on a kd-tree data structure: nodes in a kd-tree are mapped to physical overlay network nodes in the

CDS to form a distributed kd-tree (DKDT). The distributed tree indexing structure makes it possible for a similarity query to visit only a small subset of nodes to get resolved. To address the challenges associated with high dimensionality, we incorporate several mechanisms to compress the DKDT's size when possible, and we also design a virtual node shrinking algorithm to battle the dimensionality curse.

Next, we briefly review the centralized kd-tree data structure, which is used to index multi-dimensional data, and we then present the DKDT design.

5.1.1 Background on Centralized kd-tree

Over the last several decades, many data structures have been proposed to index multi-dimensional point data, such as geographical data and image vectors. Refer to [27] for a comprehensive survey on these structures. The kd-tree [12] is one of the earliest data structure proposed and it is still widely used because of its simplicity and effectiveness.

A kd-tree is in essence a binary search tree extended to support multidimensional search. At each node, a dimension and a value (known as the *discriminating* dimension and value respectively) are chosen to divide the space, and data points whose values along the discriminating dimension are smaller than the discriminating value are assigned to the left child, while points with higher value are assigned to the right child. The children are typically created when a specified bucket size, e.g., the number of data points on a node is exceeded.

Many kd-tree variants have been designed and they differ in how the discriminating dimension and value are chosen at each level. The most common variants are based on data or space partitioning. With data partitioning, e.g., the "optimized" kd-tree [12], at each level, the dimension which has the largest spread is chosen as the discriminating dimension, and the value that can divide the data set in half is chosen as the discriminating value. With space partitioning, e.g., the PR kd-tree [50], a different dimension is chosen at each level according to an *a priori* ordering and at each level, the space is divided in half,

Resolving an NN query involves two steps. First, we traverse the tree from the root, following a path to the leaf node that contains the At the leaf node, the data point nearest to the query, called the candidate NN, is identified and its distance to the query, r, is recorded. Second, we must visit any nodes in the kd-tree that may contain data points that are closer than r to the query. As we visit these nodes, we replace the candidate NN when we find a data point that is closer to the query than the current candidate NN and we also recalculate r; this may reduce the number kd-tree nodes that have to be checked.

This approach has been shown to be effective for data sets with relatively low dimensionality (up to 6-10 dimensions), e.g., the expected cost for queries in the optimized kd-tree is $O(\log N)$ [26], where N is the database size. As the dimensionality grows, the number of kd-tree nodes that has to be checked also grows, up to a point where a growing number of queries have to check all leaf nodes. This in effect corresponds to an exhaustive search of the data. To battle this "dimensionality curse", some researchers have proposed to use data structures that efficiently summarize the data, thus making it possible to quickly eliminate some nodes from consideration [73]. Alternatively, it is possible to reduce overhead by performing an approximate nearest neighbor search [9], where the returned neighbor's distance to the query point is within a given bound of the nearest neighbor's distance to the query.

5.1.2 Design Rationale

In our design, we extend the centralized kd-tree data structure into a distributed system. In particular, we map the nodes in the tree onto the physical overlay network nodes: the mapping is done by applying a hash function to the hyper-rectangle represented by each node in the kd-tree to get a corresponding node ID in the DHT network (Section 5.2.1). This set of nodes form a distributed kd-tree (DKDT). The efficiency of the DKDT infrastructure will depend on a number of important design decisions. The primary performance metric to evaluate a design is the average number of messages needed to resolve queries and perform registrations.

The first design question is what variant of the kd-tree we should use. We decided to use a space partitioned kd-tree for several reasons. First, in the applications that the CDS is targeting, data can arrive in an on-line fashion, so we do not have the full set of data *a priori*. This makes good data partitioning schemes such as the optimal kd-tree not applicable. Second, our goal is to have a fully distributed system, where endpoints can make independent decisions about where to register and query in the tree without having to go through a central location. This is only possible in a space partitioned kd-tree, since the hyper-rectangle a node corresponds to does not change and is independent of the data distribution. Finally, space partitioning also makes it efficient to maintain the parent-child relationship, which is important for recovering the tree topology if the relationship is lost due to a machine crash or leave.

One problem associated with space partitioning is that the DKDT can become very large, especially for clustered data with high dimensionality, where only a small number of dimensions are useful in separating the data. The maintenance cost of a large DKDT is high and may become prohibitive in extreme cases. To ensure manageability, the CDS compresses a DKDT using *compact splitting* and *branch coalescing* mechanisms that ensure each internal node has two children by eliminating nodes with only one child. We present the DKDT construction algorithms next in Section 5.2.2.

The second design question is how to enable end points to find the relevant DKDT nodes that may contain the nearest neighbor efficiently. In a centralized system, registrations and queries can traverse the tree starting at the root to find the leaf nodes of interest, but such a design is not desirable in a distributed system. There are several reasons: (i) the root node of the tree would become a bottleneck if every registration and query must go through it; and (ii) the end points would suffer a long delay, since the traversal of every level in the tree involves at least one network round trip time. To avoid these problems, we extend the ideas presented in the previous chapter, where a lightweight protocol is used to propagate information within the RST. This results in two sets of protocols and algorithms:

- A *distributed tree protocol*, the tree maintenance protocol (TMP), that propagates information about the tree structure along the path between each leaf node and the root of the DKDT. As a result, endpoints can learn about the structure of the tree without traversing the tree. The distributed tree protocol is described in Section 5.2.3.

- A set of *endpoint algorithms* that allow nodes to issue registrations and queries in a fully distributed fashion. We present the endpoint algorithms in Section 5.3.

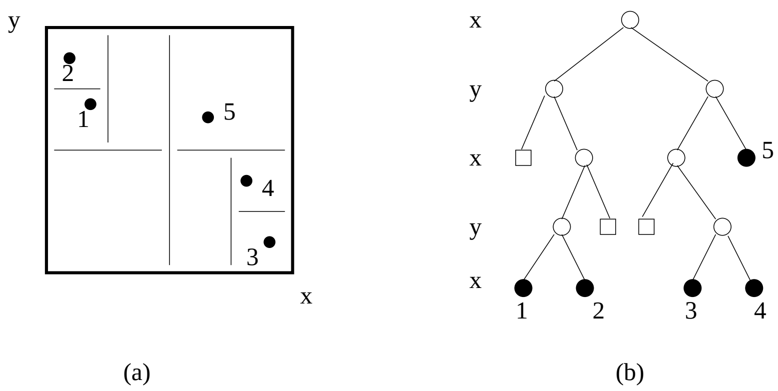


Figure 5.1: (Left child corresponds to the space that has smaller value than the division line; The ordering of dimension is x, y) (a) The locations of 5 data points in a 2-d space. (b) The logical kd-tree with bucket size = 1. Squares denote empty cells.

A final design question is how to handle data sets with high dimensionality, where the number of nodes that queries have to visit increases rapidly with the dimensionality. Given the nature of high dimensional similarity searches, this problem is unavoidable [73]. To reduce the number of DKDT nodes that has to be visited for a given query, we introduce a technique called *virtual node shrinking*. The technique allows endpoints to eliminate as many nodes as possible from the search space, by inspecting a compact representation of the data points each node stores. We discuss the virtual node shrinking mechanism in Section 5.4.

5.2 Distributed Kd-tree

We form a distributed kd-tree (DKDT) by embedding a logical kd-tree structure onto the overlay network. To avoid terminology confusion, from here on, we refer to a node in the logical kd-tree as a *cell*, and each cell is mapped onto one physical node in the network.

5.2.1 Kd-tree Mapping

We first describe the logical kd-tree data structure, which uses basic space partitioning. Suppose the data points in the given application are distributed in a k dimensional space, and the range of each dimension is known to all nodes in the system. Figure 5.1 shows an example in a 2-d space. A cell, C, in the kd-tree corresponds to a k-dimensional hyper-rectangle, which is defined by a vector that specifies its range along each dimension:

$$V_C = \{d_1 : [v^1_{min}, v^1_{max}), d_2 : [v^2_{min}, v^2_{max}), ..., d_k : [v^k_{min}, v^k_{max})\}.$$

Each cell has at most 2 children, and it uses its *discriminating dimension* to create the children. Since we use space partitioning, the division value used to create children is the middle point along the discriminating dimension: $(v^i_{min} + v^i_{max})/2$. The left child occupies the half space whose value along the discriminating dimension is less than the division value. Similarly, the right child is responsible for values that are greater than the discriminating value. We assume a default ordering of the discriminating dimensions. The root's discriminating dimension may be selected arbitrarily, and a child's discriminating dimension is the dimension next to its parent's in the ordering. The partitioning stops when certain criteria are met, e.g., the number of data points within a cell becomes less than or equal to a given bucket size, or the physical size of the leaf cell is smaller than a given size.

The mapping of a logical kd-tree onto the CDS's DHT-based overlay network is straightforward. To map a cell C in the logical tree onto the overlay network, we apply the system-wide hash function $\mathcal{H}$ to the cell's vector, V_C:

$$\mathcal{H}(V_C) \rightarrow N_{V_C}. \tag{5.2}$$

The physical node in the CDS network corresponding to this cell has an ID that is numerically closest to N_{V_C}. For simplicity, we refer to this node as N_{V_C}. From an endpoint's point of view, once the vector of a cell is determined, the DKDT node that is responsible for the cell can also be determined. We must point out that we can certainly map a cell onto an LBM, i.e., a matrix of nodes, as we did in the previous chapter, when the number of registrations is high within a cell. However, due to the different characteristics between similarity queries and a range queries, using a matrix does not necessarily reduce the cost of similarity queries. We will discuss this point in Section 5.3.3 after we explain the DKDT mechanisms.

It is important to note that in this simple mapping mechanism, the number of dimensions an application may have does not relate to the number of dimensions the underlying DHT may have. This separation makes it possible for applications with different number of dimensionality run on top of the same DHT. In particular, in the CDS, we use a DHT whose identifiers lie in a 1-d numerical space, such as Chord [64] and Pastry [58]. Our flexible design is in sharp contrast with the design of pSearch [65], which is built on top of a multidimensional DHT, CAN [55]. In pSearch, the dimensionality of the application needs to match the underlying CAN's dimensionality. If the application dimensionality changes, the DHT must be reconfigured and restarted. Different applications cannot coexist on the same network, if their dimensionalities are different.

5.2.2 Tree Construction with Compact Splitting

We now explain how the DKDT is constructed in the system, i.e., which cells are instantiated to form the distributed tree. The DKDT is built in a top-down fashion, starting with the root node which corresponds to the root cell in the logical tree and covers the whole space that all the data points lie in. We will present the data registration algorithm shortly, and for now we only need to know that each data point is registered with the leaf node that covers it in the DKDT.

Each node maintains a threshold of the total number of data points it can host, T_{reg}. If T_{reg} is crossed, it must split itself to create two children. With space partitioning, empty cells (and thus cells with only one useful child) are often created in the logical tree, since not every partitioning can separate the data. For example, in Figure 5.1, to separate data points 1 and 2, it requires three divisions of the left half of the space (once along the x dimension and twice along the y dimension). In DKDT, this means we may create a tree

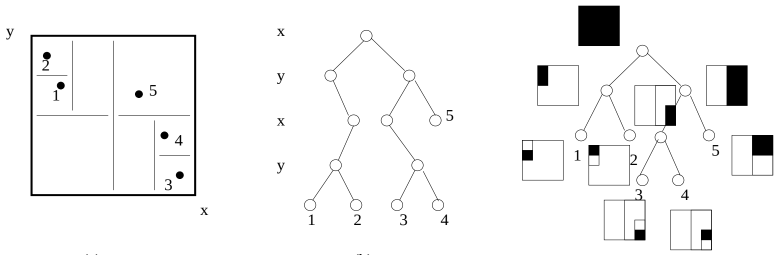


Figure 5.2: (a) 5 data points in a 2-d space. (b) Tree created without compact splitting. (c) The final DKDT. Each circle represents a physical node in the DHT. The black rectangles near a node are these nodes' corresponding cells.

that has a large height, which is not desirable because that will incur higher overhead to maintain the tree.

To avoid this problem, we design a *compact splitting* mechanism to ensure that after each split, two child nodes are created and both can inherit part of the data points from the parent node. We now describe the three steps in compact splitting by considering a node that has reached its threshold. Figure 5.2 shows a 2-d DKDT example.

1. **Locally determine the two child nodes' dimensions.** The node locally uses the basic space partitioning algorithm as described before to create two child cells using its discriminating dimension and value. The node then examines its data points, and if all the points belong to only one child cell, the node would continue to divide that cell recursively. When it obtains two cells that both contain some data points, it stops the splitting.

2. **Split data among two child nodes.** Once the two child cells are determined, the node determines two node IDs in the overlay network that correspond to these two cells by applying the hash function to the cell as defined in Eq. 5.2. It then sends each of its data point to one of the two nodes depending on which node covers the data point. As a result two child nodes are recruited to the DKDT, and the tree grows downwards.

3. **Compress parent node's dimension if necessary.** It is important to note that the union of the two children cells may be much smaller than the size of the parent node's cell due to compact splitting. After spawning the children, the parent node replaces its own cell with the union of the two children's cells. The benefit of using the union is that it reflects more accurately the data distribution within this node's space. For example, in Figure 5.2, the node that owns the left half of the space creates two small cells that divide data point 1 and 2. Its size is much larger than the cell at the top left corner that covers both 1 and 2 directly. Therefore it replaces its original cell with the top left cell, indicated by the black rectangle.

In addition to the splitting action, two child nodes may merge when both have small number of data points. They merge by shipping their data points to the parent, and the tree then shrinks upwards. This is important in a dynamic environment when the data set becomes smaller: the DKDT tree should become smaller correspondingly for management.

The compact splitting mechanism avoids creating one-child branches in the DKDT, and it is done in a distributed fashion in that decisions regarding the creation of children are made locally and no global re-adjustment is needed when splitting occurs. The DKDT created this way is similar to a BD-tree [49], which was proposed to compress a centralized kd-tree's size.

5.2.3 Distributed Tree Maintenance

In a centralized kd-tree, data insertion/registration and query involve traversing the tree from the root. However, when using the DKDT, sending all registrations and queries to the root first and then let the root node forward messages down the tree creates two problems: (1) the root node will become a bottleneck of the system when load is high; (2) the end points will potentially suffer a long delay, since each forwarding step involves an end-to-end message transfer over the network.

To efficiently use the DKDT, we extend the idea proposed in the previous chapter, where a lightweight protocol (PMP) is used to propagate and maintain the status of a distributed 1-d range search tree to support range queries. The idea here is to design a similar protocol that allows each node in the DKDT to collect and build a snapshot of the current tree shape, so that endpoints can register and query without having to traverse the tree. This is similar in spirit to building routing tables on the Internet via a routing protocol, such as OSPF [46], for packet delivering.

We first describe the DKDT's Tree Maintenance Protocol (TMP), which is similar to the PMP. Due to the multi-dimensional nature of the data, coupled with the compact splitting algorithm, the TMP is more complex than the PMP.

The tree maintenance protocol (TMP)

The purpose of the TMP is to allow endpoints to find out the status of the DKDT in a distributed fashion without having to traverse the tree from the root node. Each DKDT node builds and maintains a local database, called the Tree Information Base (TIB), which contains the current shape of the tree, in particular, it maintains the cell vector corresponding to each node. Similar to the PMP, there are two types of periodic messages to establish the TIB: the Tree Refreshing (TR) messages and the Tree Refreshing Reply (TRR) messages. Figure 5.3(a) shows an example.

First, each node in the DKDT periodically sends a tree refreshing (TR) message to its parent. A node knows its parent's dimensions, and thus the parent's ID, when the node was created by its parent via splitting. The TR message carries the part of its TIB that corresponds to the subtree rooted at this node. As a special case, the TR message from a leaf node would just contain its own dimension.

A parent node waits for the TR message from both children. If one or both of them do not arrive within a preset time, it will assume the child(ren) has left (or crashed). We will discuss how that is handled next. For now, assume it receives both TR messages, and the node will then update its TIB using the new information in the TR messages. It will in turn send another TR message up to its parent, which includes its updated subtree information.

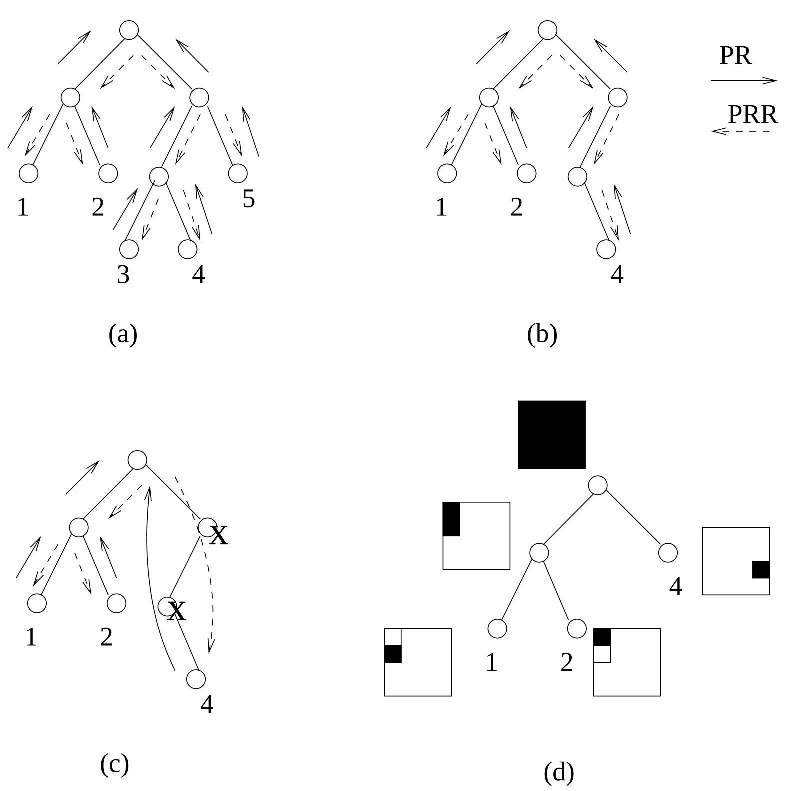


Figure 5.3: Example illustrating the tree maintenance protocol (TMP). (a) Normal message exchange. (b) Node 3 and 5 leave. (c) Branch coalescing. (d) DKDT after coalescing.

The TR messages will eventually propagate to the root. After one pass of the TR message, each node updates its subtree part of the TIB, and the root collects the full tree information.

Conversely, once the root node receives TR messages from its children, it will start sending one round of Tree Refreshing Reply (TRR) messages to its children. The goal of the TRR messages is to allow a node to update its information about parts of the DKDT other than its own subtree. Therefore, the TRR message from a parent to a child contains the full TIB on this node excluding the subtree part received from this child. For example, the root will send to its left child the part of the TIB corresponding to the right half of the tree. Upon receiving a TRR message, a node will first update its corresponding part of the TIB, and then sends its own TRR to its children.

After completing one round of TR/TRR message exchange, each DKDT node is able to establish/update its view of the current tree shape. As a simple optimization, in future rounds of TR/TRR message exchanges, instead of sending the complete information, a TR or TRR message only includes the parts of the tree whose shape has changed. For example, in a TR message, if the corresponding subtree did not change over the last time period, then the body of the message will be empty.

Handle node departure

Nodes in the DKDT may depart from the tree for several reasons. For example, a leaf node may leave the tree if its data points expire. Nodes may leave the system due to a crash. The consequence of a leaf node departure is different from an internal node departure. When a leaf node leaves, a branch containing only 1 child will be created, and the tree will become unnecessarily high, but the tree is still connected. On the other hand, if an internal node leaves, the DKDT will become disconnected. Depending on whether the departing node is a leaf or an internal node, we handle them differently as follows.

Handle leaf departure. When a leaf node leaves, the DKDT may end up having branches that have only one node. For example, Figure 5.3(b) shows the messages when nodes that host data point 3 and 5 leave, and the right half of the DKDT becomes a single branch. To reduce the height of the tree and thus minimize cost, we extend the basic TMP protocol to eliminate nodes that have only 1 child with the following rule:

After the TIB is established, a node will send TR message only to its lowest ancestor that has 2 children.

A node knows which of its ancestor meets this requirement by examining its TIB. With this rule, if a node (except the root) has only 1 child, e.g., the two middle nodes in the right branch in Figure 5.3(c), will not receive TR message in the next round and thus it will not send new TR message of its own. Subsequently, since a node only sends TRR messages to nodes from which it received a TR message before, these nodes will also stop receiving TRR messages. As a result, they are excluded from the DKDT, and the tree retains its compressed form.

We call this process of skipping nodes *branch coalescing*. By coalescing, paths are compressed and the resulting tree is consistent with the tree when it was first created using the compact splitting algorithm. Figure 5.3(d) shows the final tree shape after coalescing.

Handle internal node departure. When an internal node departs, or the parent child relationship is lost, a child node may continue to send TR messages, but it will not receive TRR messages. This node is effectively detached from the tree. When this occurs, the node must try to reconnect to the DKDT. It does so by sending a TR message to its default parent, which corresponds to the parent cell in the logical kd-tree without compact splitting. The child node can determine its default parent's cell size by local inference. In particular, suppose the child's discriminating dimension is d_i, since we assume a default ordering of the dimensions, its default parent's discriminating dimension is d_{i-1} (d_k if $i = 0$), and the parent's cell size is double the child's size along d_{i-1} dimension, and for all other dimensions, the size is the same as the child's.

The default parent node itself may not be in the DKDT, but when it receives TR messages, it will forward them up to its default parent as if it were in the tree. As such, TR messages will eventually reach a node that is in the DKDT, and the node that lost its original parent now rejoins the tree. As a side effect, the default parent nodes will become part of the DKDT due to the the reconnection process, and these nodes may have only one child. However, if this is the case, they will be eliminated by the the branch coalescing mechanism in future rounds of message exchanges.

5.2.4 Overhead of the TMP

We examine the cost of maintaining the DKDT. There are three factors that determine the overhead, namely the size of the DKDT (the number of nodes in the DKDT), the TMP message size and the TMP message frequency.

First, the size of the DKDT, or the number of nodes in the DKDT, determines the number of TMP messages needed in each refreshing round. The compact splitting and branch coalescing techniques ensure that no single branches will be created in the DKDT. The total number of nodes in the DKDT is bounded by its number of leaves. This is due to the following theorem.

Theorem 2. *Given a tree, if nodes in the tree either have 2 or 0 child, we call this type of tree a* **compact tree**. *The size (total number of nodes) of a compact tree is $2L - 1$, where L is the number of leaves.*

Proof. We prove by induction. As the base case, the smallest compact tree is a tree with a single node (one leaf, $L = 1$), and its size is 1 ($= 2L - 1$). Thus the theorem holds. The theorem also holds for the second smallest compact tree, which has 3 nodes and $L = 2$.

Now consider the size of an arbitrary compact tree that has L leaves, where $L > 1$. Since it is a compact tree, the root node must have 2 children. Both the left and right subtree must also be compact trees (otherwise, the full tree will not be a compact tree). Suppose the left and right subtree has l and r leaves respectively, and use the induction hypothesis, their sizes are $2l - 1$ and $2r - 1$. The full tree's size is:

$$T = 1 + T_{left} + T_{right} = 1 + (2l - 1) + (2r - 1) = 2(l + r) - 1$$

We know $l + r = L$, since a tree's leaves is composed of the leaves in its left subtree and right subtree. Thus we show that the tree's size is $2L - 1$. $\square$

In a DKDT, the compact splitting algorithm ensures that $L = O(\frac{N}{T_{reg}})$, where N is the total number of data points in the system. As such, the size of the DKDT is $O(\frac{N}{T_{reg}})$. The total number of TR (or TRR) messages needed in each message exchange round is equal to the number of nodes in the DKDT, since each node sends at most 1 TR message. As an example, suppose there are $N = 100,000$ data points in an application (e.g., 100,000 sensors), and each node in the CDS can take $T_{reg} = 1000$ registrations. In this case, the number of nodes needed in the DKDT is on the order of 200, and the total number of TR and TRR messages in each round is about 400. This is a very reasonable number for an application of this size. As a comparison, the size of a non-compact tree is unbounded.

We must note that the height of the DKDT is also important, since it determines the number of hops that a TR/TRR message must traverse. Because the tree is created using space partitioning, the tree may not be balanced, and the height may be larger than $O(\log L)$. Previous work [9] has shown that the height of a kd-tree created using empirical data based on space partitioning does not deviate much from this bound. As we will explain in the next section, since end points do not traverse the tree, the effect of the height on registrations and queries is not significant.

Second, we examine the size of the TR/TRR messages. The message size is no larger than the TIB on a node, which contains an entry for each node in the DKDT. However, the full TIB will be sent to a child node only when it is first recruited by its parent. Even in this case, since we only need to send each node's vector dimensions, not the data points that a node contains, the size of the initial message would still be reasonable.

More specifically, to represent each node, a k dimensional vector must be used. As such, the size of the initial message is $O(\frac{Nk}{T_{reg}})$, where k is the number of dimensions. In the above example, there are about 200 nodes in the tree, and suppose the application has $k = 6$ dimensions, and we use 8 bytes to represent each dimension (2 integers). The maximum message size is thus $200 \cdot 6 \cdot 8 = 10$KB.

After the TIB is established, in a future round of TR/TRR messages, since we only send the differences of the tree shape between now and the previous round, the message size could be very small. For example, if the tree shape did not change, then the size of TMP messages are negligible.

Third, the frequency of the TMP message exchange can be set relatively low, since the shape of the DKDT does not change often. There are primarily two reasons for this. (1) The change of the tree shape is a relatively rare event, since it happens only when a node needs to split itself. Since each network node can host a large number of data points (e.g., on the order of 10^5), this does not happen often in comparison to the data registration or query rate. (2) In the type of applications that the CDS is targeting, information such as geographical locations, content vectors, do not change very often, and once all the data points have been registered, the tree shape may change only if there is major shift of the data distribution, which again occurs on a slower time scale.

In summary, the TMP protocol is relatively lightweight, and the overhead it introduces while maintaining the TIB on DKDT nodes is low.

5.3 Endpoint Algorithms

With the TIB established on nodes in the DKDT, endpoints can carry out registrations and queries efficiently in a distributed fashion.

5.3.1 Registration

To register a data point, $p : \{d_1 = v_1, d_2 = v_2, ..., d_k = v_k\}$, the registering endpoint must first determine the lowest node in the DKDT that covers this data point. We say a node with vector

$$V_C = \{d_1 : [v^1_{min}, v^1_{max}), d_2 : [v^2_{min}, v^2_{max}), ..., d_k : [v^k_{min}, v^k_{max})\}$$

covers data point p, if the following holds:

$$v^i_{min} \leq v_i < v^i_{max}, \text{ for } i = 1 \text{ to } k.$$

With TIB built on each DKDT node, this is done by probing rather than traversing the tree. For any data point, there exists exactly one cell at each level in the logical kd-tree that covers it. The registering node first locally computes the path and then issues a probe message to a random node within this path.

The node being probed may or may not be in the DKDT. If the node being probed is not in the DKDT, e.g., it may be skipped due to the compact splitting, or has not been recruited by the DKDT yet, it will return NULL. When the probing endpoint gets such a response back, since the ordering of the cells in the path is known, the endpoint conducts a simple binary search along the path between the probed node and the root until it finds a node in the DKDT.

When a node in the DKDT receives a probe message, it will determine the lowest covering node for this data point by examining its local TIB. It then returns to the probing node the covering node P's dimension. If P is an internal node, it will also return P's two children, C_l and C_r.

Once the registering node receives the probing results, it runs a local algorithm to complete the registration. The covering node P may be a leaf node or an internal node. We describe the registration algorithm for theses two cases separately.

(1) If P is a leaf node, then the registering node sends the data point, p, to it. On node P, if after receiving the new data point, the threshold T_{reg} is crossed, it will conduct compact splitting as we discussed earlier in Section 5.2.2.

(2) If P is a non-leaf node, then this means that the new data point falls in a space that is not covered by any leaf node. The registration becomes more complex because we must maintain the DKDT's property. The registering node enumerates all three possible configurations between the new data point p, and the three nodes, P, C_l, and C_r, based on P's discriminating dimension and value. (See Figure 5.4 for an example in 2-d):

1. C_l and C_r belong to two different half planes, and p belongs to one of the half plane.
2. C_l and C_r belong to the same half plane, and p belongs to the other half plane;
3. C_l and C_r belong to the same half plane as p.

Since P is an internal node and neither of its two children covers p, the registering node must send its data point to a new leaf node in the system. In addition, a new internal node will be introduced to maintain the DKDT relationship. More specifically, there are three steps involved in registering a data point in this case.

- **Step 1: Locally determine the new leaf node and internal node's dimensions.**

  The registering node must first determine the dimension of the new leaf node. The rule to add a new non-leaf node is to make sure that the new node is the lowest common ancestor that covers either p and a child (case (1) and (3)) or the two children (case (2)). For example, for case (1) shown in Figure 5.4(1), where p and C_r are in the same half plane, the new cell, C_r', is the right half plane, which is the lowest common ancestor that covers p and C_r.

- **Step 2: Register with the new leaf node**

  The registering node then sends the data point p to the new leaf node for registration.

- **Step 3: Tree grafting**

  Finally, the registering node issues several "grafting" messages to link the new leaf node and internal node to the DKDT. For example, in Figure 5.4(1), it informs C_r

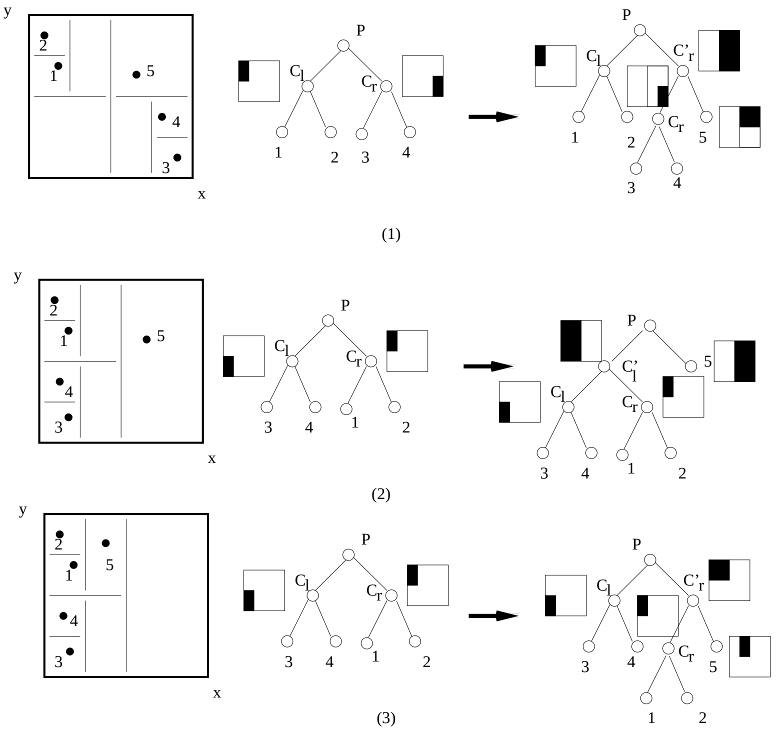


Figure 5.4: Example showing the three configurations when the covering node is a non-leaf. In all cases, the new data point is 5, and P is the current covering node and C_l and C_r are the children. The center figures show the DKDTs before 5's arrival, and the right figures show the tree after the registration.

that its new parent is C_r' and informs C_r' the new parent is P. Once the new nodes are added, they will start to send and subsequently receive TMP messages.

Registration Cost

From an endpoint's viewpoint, its registration cost comes from two types of operations. When using binary probing, the number of probing messages needed is $O(\log H)$, where H is the path length. This cost is small given a reasonable tree size. In addition, as we have done before, the probing result can be cached by the registering node for future registrations, so that the probing step may be omitted all together. Once the covering node is found, the actual registration costs one network message and several grafting messages if necessary. Hence, the cost of registering a data point is low, and our system is efficient for registration while not creating any bottlenecks.

5.3.2 Query

A node in the system may issue a nearest neighbor query, which also takes the form of a k-dimensional vector $q : \{d_1 = v_1, d_2 = v_2, ..., d_k = v_k\}$. The resolution of an NN query using a DKDT requires three steps: (1) The querying node must first determine which node

```
COMPUTE-DIST-TO-CELL(Cell V_C, Point q) {
    dist_sq = 0;
    foreach dimension i {
        if(q.v_i < V_C.v^i_min) {
            dist_sq+ = (q.v_i − V_C.v^i_min)^2;
        } else if (q.v_i ≥ V_C.v^i_max) {
            dist_sq+ = (q.v_i − V_C.v^i_max)^2;
        }
    }
    return SQRT(dist_sq);
}
```

Figure 5.5: The algorithm to calculate the distance between a query point and a cell vector.

in the DKDT covers the query vector; (2) It then sends to query to the covering node (or its subtree) to find the first candidate neighbor; and (3) To determine the final nearest neighbor, the querying node then sends the query to a list of nodes whose distance to the query is closer than the candidatate neighbor's distance. We now describe these steps in more detail.

- **Step 1: Determine the covering node**

  First, the querying node must determine the lowest node that covers the query point. Similar to the registration step, this is done by probing the tree, and we treat the query point as a data point. When a node receives a query probe, it examines its TIB and finds out the covering node's vector. The covering node's vector as well as the vectors of any leaf nodes in the covering node's subtree are returned the querying node. In the example shown in Figure 5.7, the node that contains data point 5 is returned, since its cell covers the query q. Once the querying node receives the probe return message, it will enqueue the list of nodes into a priority queue sorted in ascending order based on the distance from the query to each of the nodes. We call this queue the *Candidate Node List*, or CNL.

  We use Euclidean distance as the distance metric, and the distance between a query and a node is computed using the simple algorithm shown in Figure 5.5. The basic idea is to determine for each dimension the nearest distance between the data point and the hyper-rectangle. Along a given dimension, if the data point's value is outside the hyper-rectangle's range, the distance is the difference between the data point's value and the minimum or maximum value of the hyper-rectangle along that dimension. If the data point is within the range along one dimension, then the distance along this dimension is 0. Figure 5.6 is an example of the distance computation in 2-d.

- **Step 2: Determine the first candidate neighbor**

  The querying node dequeues the first node from the CNL, and sends the query to the corresponding node (node 5 in Figure 5.7). The node receiving the query does the following: (1) from its local database, it finds the closest data point p', whose distance to q is r; and (2) it checks its TIB, and determines a list of leaf nodes in the

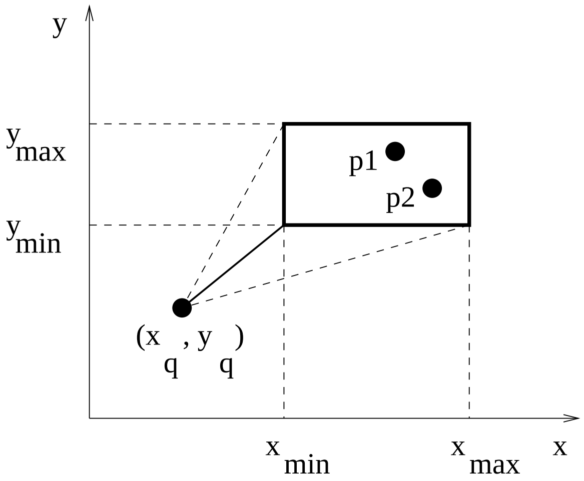


Figure 5.6: A 2d example of computing the distance between a query point and a cell.

tree whose distance to q is less than r (node 4 and 1 in Figure 5.7). These nodes may contain a closer data point. Finally, it returns p' along with the list to the querying node.

- **Step 3: Determine the final nearest neighbor**

  Once the querying node gets p' and the new list of nodes, it will enqueue nodes in this list into the CNL. Data point p' is known as the first candidate neighbor. The reason that this list is called the candidate list is that nodes on the list may contain data points that are closer than p'. The querying node then dequeues the first node from the queue, and sends the query to it. To a candidate node, the query q is known as a *candidate query*. Each node that receives the query returns the closest point on that node. The querying node updates its candidate neighbor if a closer data point is found. It stops issuing query when the current candidate nearest neighbor is closer than the distance of the next node to be dequeued. For example, in Figure 5.7, after data point 4 is found, the processes stops and node 1 will not be queried, since its distance is farther than the distance between q and 4. In this case, data point 4 is the nearest neighbor. What we described here is an iterative mechanism, where one query message is sent each time. As a simple optimization to reduce latency, the end point may choose to dequeue multiple nodes at once and send query to them simultaneously. This algorithm can be easily extended to find K nearest neighbors (KNN) by maintaining K candidate neighbors rather than 1.

Query Cost

The cost of resolving a query is determined by the number of nodes the querying node must visit. Through the use of centralized kd-trees, Friedman, Bentley and Finkel [26] showed that $O(\log N)$ query time is achievable in the expected case, where N is the number of data points. Since the distributed algorithm follows the centralized kd-tree algorithm,

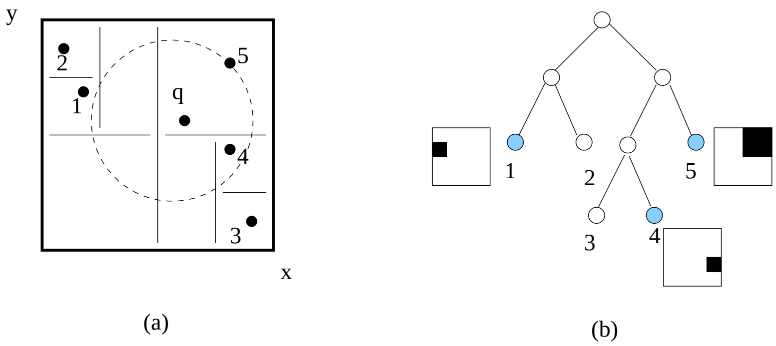


Figure 5.7: Example illustrating the query process. (a) 5 is the first candidate neighbor. 4 is the NN. (b) DKDT, and filled nodes' cells are enqueued.

the expected number of query messages needed to resolve an NN query is also $O(\log N)$. Further more, based on our analysis in Section 5.2.4, the number of leaf nodes in the DKDT $L = O(\frac{N}{T_{reg}})$, and hence the query cost is $O(\log N) = O(\log L)$. However, we must note that the constant factors hidden in the asymptotic bound contain 2^k, which will make the cost become much larger than $\log L$ when the dimensionality k increases. We will discuss how we handle this in the next section.

5.3.3 DKDT and LBM

As we alluded to earlier, when we map a kd-tree cell onto the overlay network, unlike what we did in the last chapter, here we only map it onto 1 physical node instead of an LBM. For range queries, since we know the size of a query, the most efficient way of resolving the query is to use the level in the RST that corresponds to the length of the range if possible. This is why handling queries with different ranges efficiently requires the aggregation of registrations at different levels of the RST using LBMs.

However, to resolve a similarity query, multiple partitions at a high level will not be helpful. We use our earlier example to illustrate why. With $N = 100,000$ and T_{reg}, if we use an LBM at the root level to host all the data points, the root level will have 100 partitions. To resolve any similarity query, all the 100 partitions must be visited. On the other hand, when we use a DKDT for this application, since the DKDT is formed in a top-down fashion, and leaves will be created only if a parent node reaches its threshold, the DKDT will have approximately 100 leaves. This equals the number of partitions above. For any nearest neighbor query, it is likely that that query will not be sent to all the leaves, since the tree structure will cut down the search space dramatically. What this means is that for similarity queries, the performance of using a tree will never be worse than using a matrix which provides no indexing information. Therefore, we do not keep data at internal levels of the DKDT. Of course, if we will also use the DKDT for multidimensional range queries, LBMs at high levels will certainly be useful for these queries.

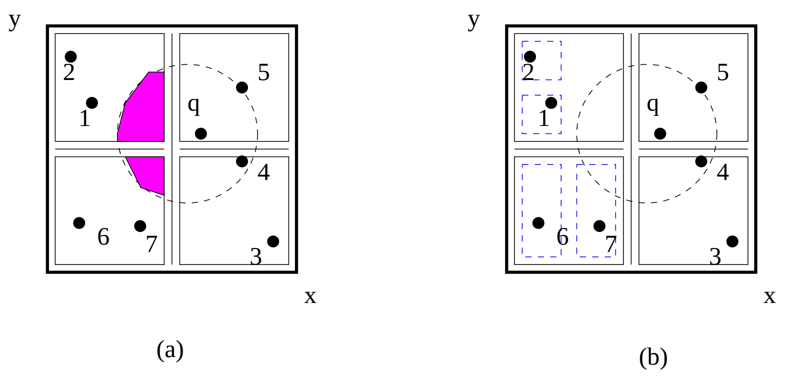


Figure 5.8: Virtual node shrinking example. (a) Without shrinking. (b) Shrinking by creating sub-trees. Dotted boxes denote sub-tree cells.

5.4 Virtual Node Shrinking

Mechanisms based on kd-trees work well for low dimensionality data, since the internal nodes can guide queries to cut down the search space quickly. However, all tree based indexing schemes suffer as dimensionality increases. This is known as the curse-of-dimensionality problem in the literature [73].

In DKDT, when a query determines whether to visit a node, it uses its distance to the node (the boundaries of the cell) to approximate its distance to data points within the node. This may be a bad approximation specially when the node's cell contains large empty space and the data points are clustered. As an example, in Figure 5.6, the distance between q and the cell is a poor estimation of the distance to data point p_1 or p_2. This problem is especially prominent for high dimensionality datasets: when $2^k >> N$, within each leaf node's cell, data points are distributed sparsely [73]. Consequentially, for any query, it is likely that every leaf node will appear in the query's candidate list and has to be visited. For example, in Figure 5.8, the four inner squares denote four leaf nodes in a DKDT, each of which hosts at most 2 data points. For query q, it is covered by the node who contains data point 5. The candidate list for q includes the other 3 leaf nodes, since the circle around it intersects with all of them. However, for the two leaf nodes on the left, their data points are far away from q and ideally should not be visited.

To improve query performance, i.e., reduce the number of nodes to be visited, we design an adaptive virtual shrinking mechanism that depending on the type of queries it receives and the data distribution within its cell, a node may "reduce" its cell size to reflect more accurately the type of data points it has. This way, future queries may be able to avoid visiting this node.

The virtual shrinking on a node occurs when both of the following criteria are met: (1) the node receives frequent candidate queries (exceeds a threshold); and (2) the candidate query's hyper-sphere does not intersect with any of the data points on this node. The first criteria means that the node is in the vicinity of many queries, and the second criteria means it the data points in this node are not close to the boundary and the distance estimation is poor.

If both criteria are met, the node makes a decision to give a more accurate representation of the data within it, and then propagate this information in the TMP messages. This way, in the second step of query resolution, the first node being queried will determine its candidate list by computing the distance from the query point to the finer representation of each leaf node. In Figure 5.8(a), the criteria are met on the two left nodes, but not on the lower right node, since data point 4 is within the circle.

There are many ways of shrinking a cell. For example, we can create a minimum bounding rectangle (MBR [11]) or further split the data within the cell to create an internal tree. In Figure 5.8(b), the two left nodes create virtual trees within themselves. In this example, q's candidate list will not include the top left node, since the distance from q to either of its two smaller sub-tree cells is larger than the first neighbor 5's distance.

The main issue with this scheme is that after virtual shrinking, this new representation must be propagated throughout the tree, so that it can be used by endpoints. Inevitably, adding information to the TMP messages increases the cost of the TMP. To ensure a low overhead, we leverage the VA-file mechanisms [73]. In particular, when a node decides virtual shrinking is necessary, it uses a small number of bits, b_i, (e.g., $b_i = 2 - 4$) along each dimension to divide its cell and create a grid. The VA-file representation of the data points in the cell is the list of sub-cells within the cell that contain any data point, so each sub-cell is represented using b_i bits along each dimension depending on its location within that dimension. This representation is small in comparison to the complete representation of the data points as shown in [73]. As an example, in Figure 5.9(a), a node contains 5 data points, and its cell is defined by vector $(x_{min}, x_{max}), (y_{min}, y_{max})$. The node decides to use 2 bits to divide each of the dimensions, and this results in 5 sub-cells that contain the 5 points, as shown in Figure 5.9(b). For example, to represent the top left sub-cell that contains data point 2, four bits are needed (00, 10), where 00 indicates that this sub-cell is located left-most along the x dimension, and 11 indicates that it is located top-most along the y dimension.

Once a VA-file representation is chosen, each node will send out its cell's VA-file representation in addition to its cell vector in future rounds of TMP message exchange. Subsequently, the TIB on each node will replace each node's entry with the VA-file representation. The registration algorithm is carried out as before. The query algorithm is changed in that the distance computation function is different from before. Instead of determining the distance based on the boundaries of the cell, the distance between the query point and the cell is determined by the distance between the query point and the closest sub-cell in the VA-file representation. This can be achieved since once a node's dimensions and the number of bits it uses to divide each dimension are known, each sub-cell's actual dimensions are is easily determined for the distance computation use. Figure 5.10 uses the same example shown in Figure 5.6, and shows the improvement of the distance estimation between the query point q and the data points p_1 and p_2 by using VA-file.

5.5 Evaluation

We incorporated the DKDT mechanisms in the CDS simulator presented in Section 3.4.1. In this section, we present evaluation results obtained from simulation. We use both synthetic data sets and the music dataset to drive the simulation. Each synthetic data set has 100,000 data points with a certain number of dimensions. The domain of each dimension

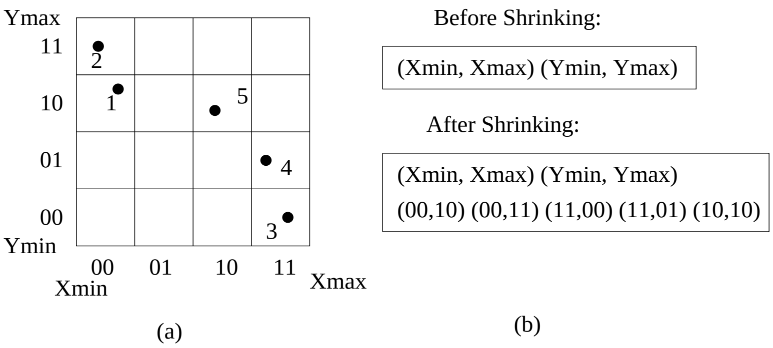


Figure 5.9: Virtual shrinking using VA-file representation. (a) Use 2 bits to divide a 2-d space with 5 data points. (b) The information transferred before and after shrinking.

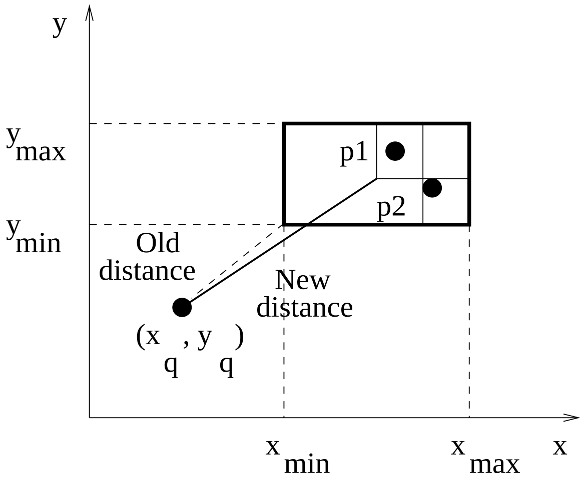


Figure 5.10: Distance computation with virtual shrinking.

is $[0, 20000)$. For the synthetic data sets, we use two distributions: (1) In the *uniform distribution*, a random position is generated in the k-d space for each data point. (2) In the *clustered distribution*, we first generate 100 cluster centers uniformly, then for each data center, we generate 1000 data points following a Normal distribution with $\sigma = 100$ around the center and along each dimension. The music data set consists of 5000 30-d feature vectors extracted from a set of assorted mp3 files as we described in Section 3.4.2.

For the query load, the synthetic data sets consist of 5000 uniformly distributed queries for both the uniform and clustered distribution. For the mp3 data set, we use the feature vectors also as queries, and here we find 2 nearest neighbors instead of 1, since the first nearest neighbor is the data point itself.

In our experiments, we set up an overlay network that has 20,000 nodes, and we set the T_{reg} on each node to be 100. The sender of a data point or query is chosen randomly among all nodes. To demonstrate the effectiveness of our system, we show the performance for queries, evaluate the DKDT maintenance cost, and study the importance of the virtual shrinking mechanism for high dimensionality datasets.

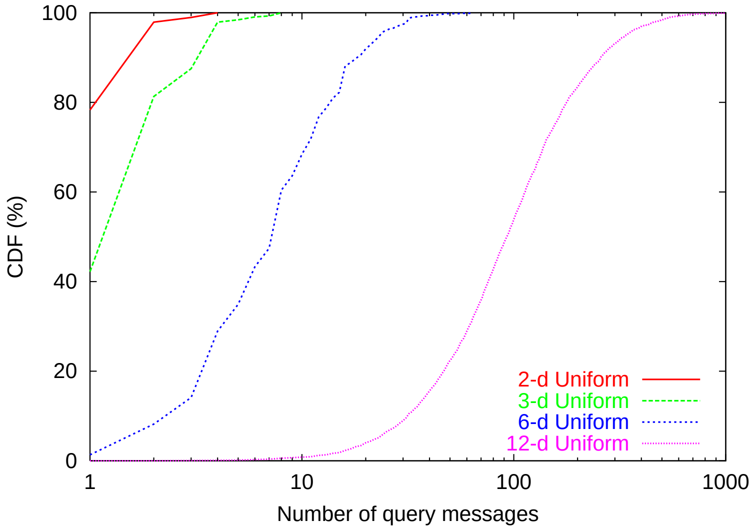


Figure 5.11: Cumulative distribution of query cost for uniform data sets.

5.5.1 Query Performance

We first examine the query performance in terms of the number of messages needed to resolve a nearest neighbor query. Here, the system only uses the basic DKDT mechanisms without virtual node shrinking. For each experiment, we first inject a registration load into the network and after all the data points are registered, we then inject a query load. We record the number of query messages needed to resolve each query. Figure 5.11 shows the distribution of the number of query messages for the uniform data sets with different dimensionalities. The DKDTs in different scenarios all have about 1500 leaf nodes. A naive linear algorithm to resolve a similarity query would have to visit all the leaf nodes.

As can be seen in the figure, for low dimensionality, e.g., $k = 2, 3$, the system is very efficient in that all the queries need to visit less than 10 nodes to find its nearest neighbor. As dimensionality increases, for $k \leq 6$, the performance is still very good: for example, over 90% of the queries need less than 20 messages. The performance starts to degrade for larger k. When $k = 12$, over 40% queries need to visit 100 or more leaves in the system. This shows the limitation of kd-trees. For $k = 6, 12$, we further compare the query performance using the clustered data set. Figure 5.12 shows that the performance is similar to the uniform data set.

The performance we observe here is consistent with centralized kd-tree based searching algorithm. We conclude that the DKDT mechanisms can effectively cut down the search space for nearest neighbor queries and thus are efficient to resolve endpoints' queries in a distributed fashion. The system works well when the dimensionality is reasonably small.

5.5.2 DKDT Maintenance Cost

In this section, we evaluate the overhead of the tree maintenance protocol. First, we note that an important advantage of the kd-tree based mechanism is that from a node's point

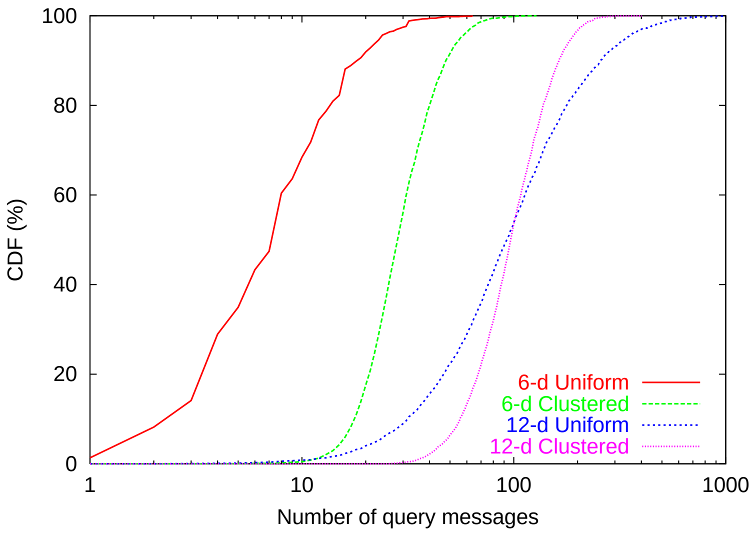


Figure 5.12: Query cost comparison for uniform and clustered data sets.

of view, in each round of message exchange, the number of maintenance messages sent and received is *constant* and independent of the data/query distribution or the dimensionality. In particular, a node receives at most 2 TR messages, 1 PRR message, and it sends 1 TR message and 2 TRR messages in each round. This is especially important as it avoids the typical problem encountered by tree based system where nodes higher up in the tree are often overloaded. As an example, in a design that is based on PR Quadtree [60], each node may have up to 2^k children, and a node will easily be inundated by the number of messages from its children for a larger k.

From the system's point of view, the total overhead of the DKDT maintenance protocol is determined by the size and height of the tree. First, we examine the number of hops a TR/TRR message initiated by the leaves or root must go through in each round of message exchange. The number of hops correspond to the number of levels between each leaf node and the root. Figure 5.13 compares two scenarios, with and without compact splitting when constructing the DKDT for the clustered data sets. For both $k = 6$ and 12, using compact splitting, the average number of hops is about 15 (the two curves essentially overlap), and it is primarily dependent on the number of data points and not the number of dimensions. In contrast, without compact splitting, the number of hops increases dramatically as dimensionality increases.

We then examine the size of the DKDT in these two scenarios. With compact splitting, for $k = 6$, there are 2786 internal nodes and 2787 leaf nodes in the DKDT, and the numbers are 3039 and 3040 for $k = 12$. Without compact splitting, the number of leaves are the same, but the tree sizes explode due to many internal nodes involved in the DKDT (5599 and 10525 internal nodes for $k = 6$ and 12 respectively). Table 5.1 provides a summary of the DKDT height and size comparison for these two scenarios.

We also experimented with the uniform data sets. As expected, the size and height of the tree with and without compact splitting are similar, since when the data are uniformly distributed, the DKDT is naturally balanced when it is created.

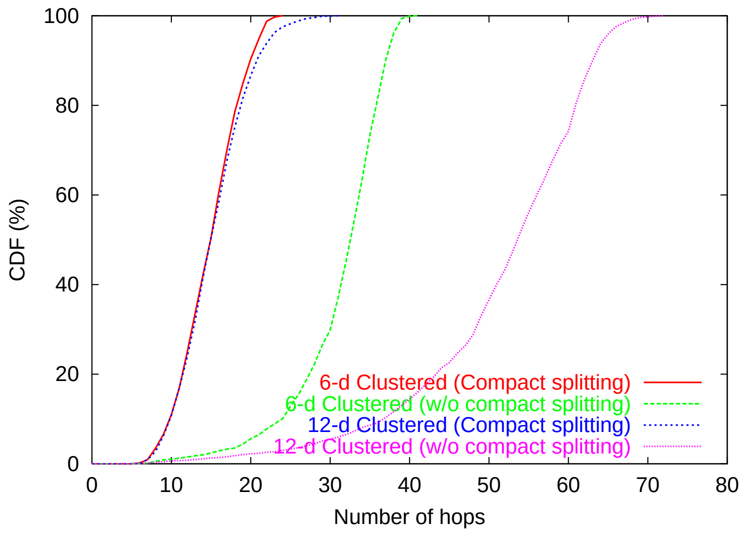


Figure 5.13: CDF of TMP message hops for clustered data sets.

Table 5.1: Summary of DKDT height and size for 6-d and 12-d clustered data sets with and without compact splitting.

	6-d	6-d w/o	12-d	12-d w/o
DKDT height				
Average	15.3	31.7	15.7	51.9
Min	6	6	6	6
Max	24	41	31	72
DKDT size				
Num. of internal nodes	2786	5599	3039	10525
Num. of leaf nodes:	2787	2787	3040	3040

In summary, the DKDT maintenance overhead is reasonably small and the compact splitting construction algorithm is vitally important to ensure a manageable DKDT, especially for realistic clustered data sets.

5.5.3 Effectiveness of Virtual Shrinking

We now examine the importance of the virtual shrinking technique in improving system performance for high dimensionality data sets. For the experiments shown here, we use both the synthetic datasets and the music data set.

Evaluation using Synthetic Dataset

First, we run experiments using the synthetic data sets with $k = 12$. Figure 5.14 shows the query performance improvement for random and clustered data sets. In this figure, "2 bits" means a node can use 2 bits to divide along each dimension for virtual shrinking. For both cases, the improvement is significant: the average number of query messages drops from 105 to 54 for the clustered data and from 125 to 64 for the uniform data set.

We further test the effectiveness of the mechanism using another synthetic clustered dataset, which has 500 uniformly chosen cluster centers and each cluster has 200 uniformly chosen points within a radius of 100. We call this clustered data set the Clustered (Uniform), and our previous clustered dataset, Clustered (Normal). Figure 5.15 compares the performance improvement of these two datasets. The improvement for the Clustered (Uniform) is very significant: the average number of query messages drops from 124 to 14. This is because, unlike in a Normal distribution where data can be fairly spread out, in this data set, the data points are contained within fixed ranges. As such, there are more "white spaces" near the boundaries of a cell, and it offers more opportunity for virtual shrinking.

These results validate our design: virtual shrinking makes a node's effective size smaller, and thus reduces the cost of queries. This technique is specially useful for highly clustered data, since it provides more opportunities for nodes to shrink itself and thus allows endpoints to significantly cut down a query's search space.

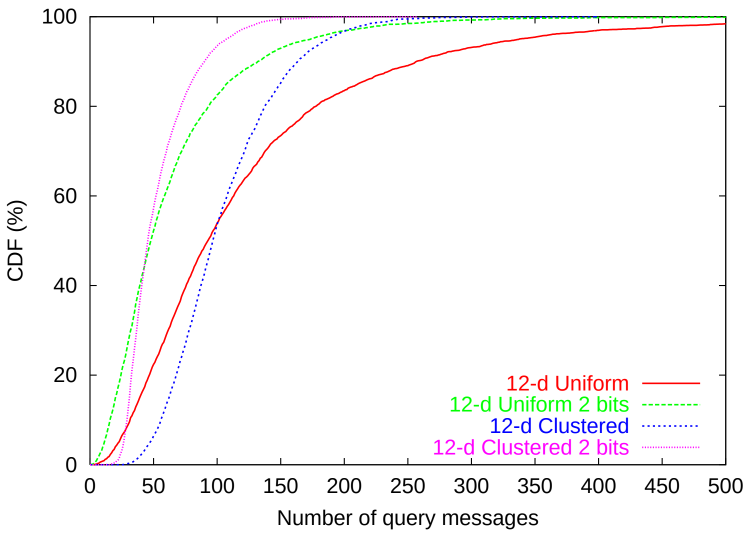


Figure 5.14: Effect of virtual shrinking on query performance.

Evaluation using Music Dataset

We then test the effectiveness of the virtual shrinking using the mp3 data set. This dataset has fewer data points (5000), but it has a large dimensionality, $k = 30$ ($5000 << 2^{30}$). Using the same threshold ($T_{reg} = 100$), the DKDT created has 119 leaf nodes.

Figure 5.16 shows that without virtual shrinking, as we expected, the system degrades to linear search for most of the queries: about 80% of the queries need to visit more than 80% of the leaf nodes to find the nearest neighbor. However, with virtual shrinking, the performance improves significantly as a node uses more bits to divide its space. In particular, when 4 bits are used, the average number of query messages needed per query reduces to less than 10. The effectiveness of virtual shrinking verifies that for high dimensionality data

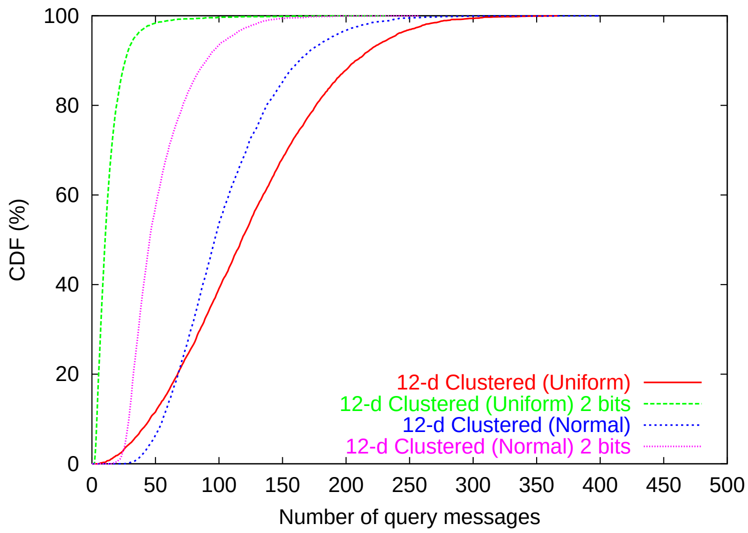


Figure 5.15: Comparison of the effect of virtual shrinking using 2 different clustered data sets.

sets, it is often the case that cells are sparsely populated, and there are many empty spaces in a cell.

The extra amount of data carried in the TMP due to the virtual shrinking is tolerable. In this experiment, for a 30-d space, for a node that has 100 data points and using 4 bits per dimension, the extra amount of data in a PR message that must be transferred is at most $100 \cdot 30 \cdot 4/8 = 1500$ bytes, if each data point occupies a different sub-cell.

For the above experiments, we also examine the query load observed on each node in the DKDT under different division granularities. After each experiment, we tally the number of candidate queries received on each node. Figure 5.17 shows the CDF of number of candidate queries received on nodes. Without virtual shrinking, about 50% of the nodes receive more than 80% of the total queries. Virtual shrinking significantly cuts down the number of candidate queries a node receives. As a result, with 4 bits, a node on average receives 7% of the total queries.

To summarize, virtual shrinking is useful in improving query performance for clustered data, and data with high dimensionality. As a side effect, virtual shrinking may also reduce query load on nodes. However, we note that the virtual shrinking mechanism is only an optimization useful for many realistic data distribution, and not a solution to the curse of dimensionality problem.

5.6 Related Work

Similarity queries have been studied heavily for the last several decades in the database community. Various multidimensional access methods have been proposed. Gaede et al. [27] provides a comprehensive survey of these work. In our work, we leverage these results and build our system, the DKDT, on top of a distributed kd-tree [12], and the DKDT shape resembles a BD-tree [49].

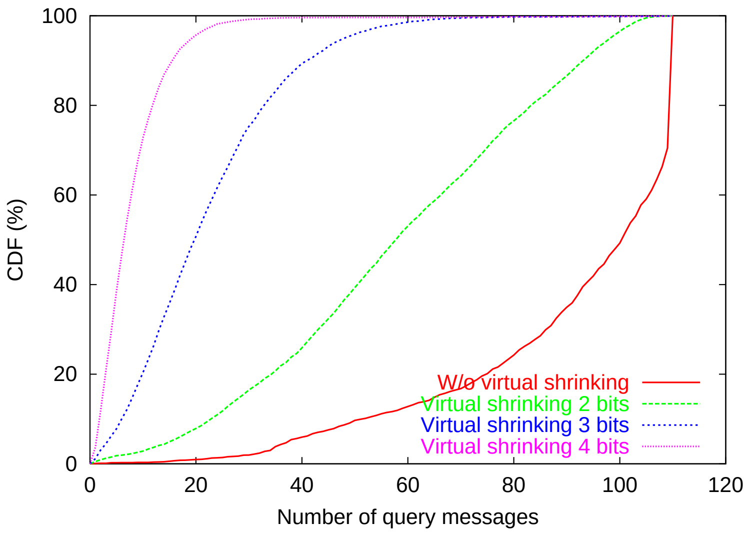


Figure 5.16: Virtual shrinking improves performance for the mp3 data set.

Supporting complex queries in a DHT-based system was proposed as an open question in [35]. Since then, many efforts have been made in this domain. In Chapter 4, a DHT-based system that supports range queries efficiently was proposed. The system is built on top of a distributed 1-d binary tree, known as Range Search Tree. [54] proposed a similar structure for range queries. In this work, we extend these ideas, and build our system around kd-tree to support similarity searches in multi-dimensional data sets.

Several systems are built on top of multi-dimensional DHTs such as CAN [55] to support complex queries. Most closely related to our work is the pSearch system [65], where CAN is used to implement an information retrieval system. One limitation of CAN-based system is the dimensionality of the application data points is tightly related to the dimensionality of the underlying CAN, which makes the system not usable for a different application with different dimensionality. pSearch works well for applications with very high dimensionality while relying on some application specific techniques. For example, nodes must communicate about their data and hopefully form clusters, and thus to guide similarity queries. In our system, the application dimensionality is independent of the underlying DHT, and one DHT-based network can support arbitrary number of applications with different dimensions. We use a tree structure to support similarity queries, and make no assumptions on the data distribution. Our system is not designed to handle very high dimensions (e.g., on the order of hundreds), and we propose optimization techniques to improve performance when dimensionality is high.

In a different context, Li et al [44] proposed a distributed mechanism to partition a multi-dimensional space using a data structure similar to kd-trees to support range queries in sensor networks. Our scheme is similar in that we also build distributed kd-tree, but we our goal here is to support similarity queries efficiently.

To address the inevitable dimensionality curse in similarity queries, locality sensitive hashing (LSH) has been proposed and it is efficient in supporting approximate nearest neighbor (ANN) queries [24]. In [33], locality sensitive hashing functions are used for range

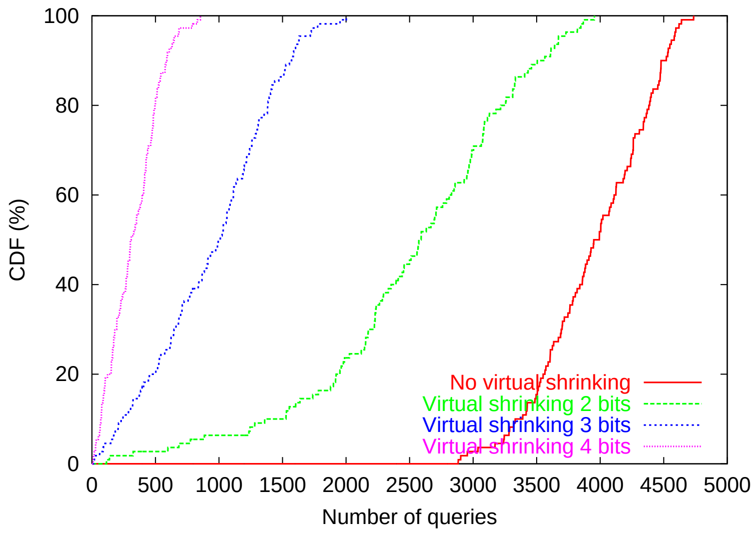


Figure 5.17: Number of candidate queries distribution for the music dataset.

queries in relational databases built on top of DHT. It seems LSH can also be used in a DHT-based system to address similarity queries. To make use of LSH, a set of parameters must be chosen properly depending on the type of data. This may be difficult in a distributed environment. The authors are exploring this technique as future work.

5.7 Chapter Summary

In this chapter, we presented a distributed kd-tree (DKDT) based mechanism to support similarity searches in the CDS efficiently. In particular, we map a space partitioned kd-tree onto the underlying DHT-based overlay network. In constructing and maintaining the DKDT, we use techniques such as compact splitting and branch coalescing to ensure the tree's size remain manageable as dimensionality increases. We use a lightweight protocol, the tree maintenance protocol (TMP) to propagate the current tree shape to all nodes in the DKDT, so that endpoints can discover the shape of the tree in a distributed fashion. As such, registrations and queries can be carried out efficiently without traversing the tree and creating bottlenecks at the root of the tree. In addition, we propose a novel virtual node shrinking mechanism to mitigate the dimensionality curse problem for datasets with high dimensionality. Extensive simulations show that our system works well for low dimensionality datasets and the virtual node shrinking mechanism can improve the system's performance for high dimensionality data, especially when the data are clustered, which is often the case in real applications.

Chapter 6

Prototype Implementation

To validate the feasibility of the CDS design presented in this thesis, we implemented a prototype of the CDS system, called Camel, which includes the basic Rendezvous Point based registration and query mechanisms with distributed load balancing. In this chapter, we describe the Camel implementation and our experience in deploying Camel on Planet Lab, an overlay network testbed on the Internet.

6.1 Implementation

In our implementation, we built Camel on top of a Chord implementation developed for the CFS project [18]. The version we worked with was downloaded from the Chord research group [52] in May 2003. As we described in Chapter 2, the CDS only requires a minimal programming interface from the underlying DHT layer, namely the `put(key, message)` function. Since this function is provided by all DHTs [3] including Chord, the CDS can be built on top of any DHT available. For example, we could also build Camel on top of other DHTs such as Pastry [58] the same way. It is worth mentioning that since we do not make any assumptions about the DHT other than the simple interface to support the CDS functionalities, we do not need to use more complex DHTs. For example, we do not need to use a multidimensional DHT such as CAN [55] to support complex queries. We now describe how we build Camel using the CFS software.

6.1.1 Modify DHash's Chord Layer

CFS is a distributed file system built on top of Chord, and from the software layering point of view, CFS is at the same level as the CDS. The CFS software is called DHash. DHash implements Chord's functionality such as forming and maintaining an overlay network, routing and forwarding messages within the network. A block-based file system is then layered on top of DHash. To make use of this implementation of Chord, we have to slightly modify the following two aspects of the DHash code.

First, since the goal of DHash is to build a distributed file system, its Chord API provides basic file system operations for blocks of file, e.g., `put_h(block)`, `put_s(block, pubkey)`, and `get(key)`. These functions can not be directly used by Camel. Based on these existing functions, we added the generic DHT method, `put(key, message)` method to the `dhashclient` class in `dhashclient.C`. This method basically allows upper layer applications such as Camel to send any message to any node in the DHT.

Second, once the Chord layer receives a message, instead of letting the DHash file system layer handle the message, Camel must intercept and process it. This is accomplished by modifying the `void dhash_impl::dispatch (user_args *sbp)` function in `server.C`. In particular, we added the following lines in this function to pass the received message to Camel:

```
cds_msg *message = new cds_msg;
s_dhash_fetch_arg *rcv_msg = sbp->template getarg<s_dhash_fetch_arg> ();
message->process_cds_msg(rcv_msg, ...)
```

As such, when the Chord layer receives a message that is destined for the upper layer, it will pass the message to Camel by calling the `process_cds_message()`, which we explain next.

6.1.2 Camel Software Structure

Camel lies in between the DHT layer and the application layer. It interacts with both: It may receive application requests such as registrations and queries through the Camel API (discussed next), and it sends and receives CDS messages to and from the DHT layer. As such, the core component of Camel consists of two functions: (1) send CDS message, and (2) process a received CDS message.

Send CDS message

To send a message, Camel simply calls the Chord API function `put(key, message)` that we added. It first creates an instantiation of the `dhashclient` class and then invokes its `put` method as follows:

```
void send_cds_msg(key, message)
     ...
     str control_socket = "/tmp/chord-sock";
     dhashclient *dhash = New dhashclient (control_socket);
     dhash->put(key, message, message_len, call_back_function));
     ...
}
```

Process CDS message

A CDS message is defined using the following structure:

```
typedef struct {
       int msg_type;        // The type of message
       int data_len;        // The length of the payload data to follow

       bigint src_node_ID;  // The sender's node ID
```

```
            bigint dst_node_ID;  // The receiver's node ID

            char *data[0];       // The payload of this message
        } cds_hdr;
```

The message types include registration and query messages, and messages pertaining to matrix management, e.g., matrix size probe, matrix expansion and shrinking etc. Unlike the simulator implementation, the prototype does not implement the various mechanisms for range and similarity queries, but these mechanisms can be added staightforwardly. When Camel receives a message, it processes it based on the mechanisms we described in the previous chapters. In our code, the processing function mainly consists of a `switch` statement:

```
void process_cds_message(msg){
        switch(msg->msg_type) {
                case CDS_MSG_INSERT:
                   // insert the content name to a local database
                   ...
                   break;
                case CDS_MSG_RETRIEVE:
                   // find a match from the local database
                   ...
                   break;
                ...
       }
}
```

Local Data Structures

The Camel layer on a node maintains two important local data structures. First, the node may be the head node of one or more matrices, therefore it must maintain the matrix size information for each of the matrices. Second, it must maintain a local content name database to store the registrations that it receives. This database is used to resolve queries that this node receives.

The Camel code is compiled into a library, *libcds.a*, and it is then linked to the rest of the modified DHash code to form a stand alone binary, called *cameld*. To use Camel, each node participating in the system must start *cameld*, which runs as a daemon. To join the CDS network, a new node must know at least one other node in the existing network. By contacting that node, the new node will be able to join the system using the underlying Chord protocol and become part of the CDS overlay network.

6.1.3 Camel Applications

Camel API

To facilitate the building of higher level applications, such as peer-to-peer object sharing systems, or distributed service discovery systems, Camel exports the following API functions (defined in a header file `cds_api.h`):

1. `int connect_to_cds(char *reg_filename, char *query_filename);`

   This function allows an application to connect to the CDS daemon for registration and query purposes. The two arguments specify output file names that will be used to record registration and query results. For example, they will record information such as whether a registration or query is successful and if a query succeeds, record the matched content names.

   Inside this function, the connection between the application and *cameld* is established by connecting to a Unix socket that the daemon is waiting on. If the connection is established successfully, "0" will be returned, otherwise, a non-zero value will be returned.

2. `void disconnect_from_cds(void);`

   This function allows the application to release the socket and other resources such as memory that has been allocated. It is called when the application is done using Camel.

3. `void register(char *content_name);`

   This function allows the application to register a content name with the system. The argument `*content_name` is a string that consists of a set of AV-pairs.

   Inside this function, it uses algorithms presented in previous chapters, and calls the `send_cds_message()` function to send messages to the proper nodes in the system.

4. `void search(char *query_str);`

   This function allows the application to issue a query, which is also a set of AV-pairs. Similarly, this function eventually calls the `send_cds_message()` function and send the query to proper nodes.

Note that this API is consistent with the API we proposed in Chapter 2.

Implementing CDS Applications

Given the Camel API, implementing an application becomes straightforward. The application must include the API's header file `cds_api.h` for compilation, and then link with *libcds.a* to create the final executable. As an example, Figure 6.1 is a complete list of a CDS application that first registers a song description and then issues a query. More complicated applications can be built similarly. For example, in the peer-to-peer music information retrieval system we described in Chapter 2.3, the application employs a sophisticated signal processing module to extract a feature vector for each song before it connects to Camel.

We now use the example application in Figure 6.1 to illustrate in more detail the sequence of function calls when the application issues a registration. The following steps are involved:

Step 1 Application $\longrightarrow$ Camel

The application calls Camel API `register()`.

Step 2 Within Camel, `register()` $\longrightarrow$ `send_cds_msg(key, message)`

The `register_content()` function first separates each AV-pairs. Suppose Camel does not know the AV-pair `artist=U2`'s corresponding matrix size, based on the algorithm presented in Chapter 3 it will try to find out its size by probing this pair's head node. It calls `send_cds_msg(key, message)`, where `key` equals the hash of {`artist=U2, 0, 0`}, which corresponds to the head node's ID, and the type of this message is CDS_MSG_MATRIX_PROBE. The `register_content()` function will then wait for the reply to come back.

Step 3 Camel $\longrightarrow$ Chord

`send_cds_msg(key, message)` then calls Chord API `put(key, message)`.

Step 4 Chord $\longrightarrow$ Destination node Chord

The Chord layer then forwards this message cross the network and eventually the message reaches the destination node whose ID is the successor of `key`. This node is the head node for {`artist=U2`}.

Step 5 Chord $\longrightarrow$ Camel

On the destination node, the Chord layer receives the message and passes it to the Camel layer by calling the `process_cds_message()` method.

Step 6 Within Camel.

The Camel layer then processes this message. In this case, since this node is a head node and the message is a probe message, it retrieves {`artist=U2`}'s matrix size, and then sends a CDS_MSG_MATRIX_PROBE_REPLY message back to the original node by calling `send_cds_msg(key, message)`. Here `key` corresponds to the original node's ID.

Step 7,8 Similar to **Step 4,5**, the Chord layer on the original node receives the message and passes it onto the Camel layer.

Step 9 Within Camel.

The Camel layer processes the probe reply message by extracting the matrix size of {`artist=U2`} from the message. Using the matrix size, the execution of the `register_content()` function is resumed. It will then choose a partition from the matrix and call `send_cds_msg(key, message)` method again. This time, the message is an actual registration message with a type CDS_MSG_REGISTER. The `key` corresponds to the hash of AV-pair, and the selected row and column index.

Step 10 Eventually when the corresponding node receives this message, its `process_cds_message()` function will be invoked, which stores the registration in its local database.

```
// ///////////////////////////////////////////////////////
// example.c
//   register a song's description, and then issue a search

#include <cds_api.h>

void main(){

    int connected;

    char song_desc[] = "artist=U2, year=1993, song=Zooropa";
    char query[] = "artist=U2";

    // output file names
    char register_fn[] = "registration.dat";
    char query_fn[] = "query.dat";

    // connect to the CDS daemon
    connected = connect_to_cds(register_fn, query_fn);

    // connected successfully
    if(connected) {
        // issue a registration
        register(song_desc);

        // issue a query
        search(query);

        // done using CDS
        disconnect_from_cds();
    }

    return;
}
// End example.c ///////////////////////////////////////////
```

Figure 6.1: Example code of a simple CDS application.

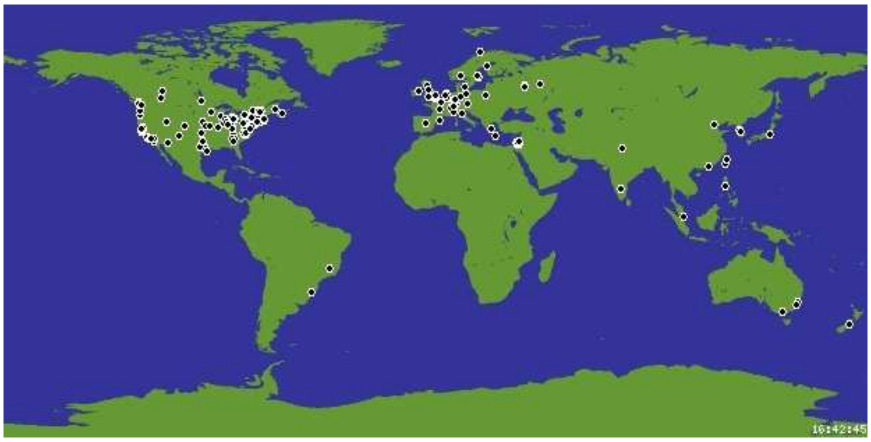


Figure 6.2: Planet Lab Testbed.

6.1.4 Implementation Discussion

Recently, the OpenHash [39] project proposes two different models for building distributed applications using DHTs. The first model is the so-called library model, where applications of a DHT are "bundled" together with the underlying DHT code. The advantage of this model is that applications can obtain local DHT states and thus can implement rich functionality. The disadvantage is that this model makes the deployment of DHT applications expensive, e.g., two different applications running on the same node must run two copies, possibly the same DHT. The second model is called the service model, where a DHT is run on a set of infrastructure hosts as a service, and can be shared by different applications.

The implementation we presented above uses the library model, since the CDS code is compiled together with the DHT code. However, we chose this model not because of technical necessity, since we do not need to obtain any information from the DHT. The CDS's minimal requirement from the DHT makes it also possible to build the CDS using the service model. In fact, our architecture fits the description of how OpenHash is used to support advanced applications perfectly. In our case, CDS is an "advanced application" of OpenHash. The CDS program needs to be co-located with the OpenHash program on each node that participates in the CDS network. The OpenHash program must pass the CDS messages the node receives to the CDS program, and OpenHash does not need to maintain local data stores for the CDS.

6.2 Internet Deployment and Evaluation

We deployed the Camel software on the Planet Lab testbed [51]. Planet Lab is a planetary scale overlay network testbed. As of August 2004, it consists of over 400 nodes on the Internet from about 180 sites around the world. These sites include universities, industrial and governmental organizations. Figure 6.2 is a snapshot of where the nodes are geographically located.

Nodes on Planet Lab are heterogeneous in that they may have different levels of network connectivity and CPU power, but they all run the same version of Linux. To test our software, we chose 100 nodes randomly from the US, Europe and Asia. We distributed two binaries to each of these nodes: the Camel daemon (*cameld*) and a simple application. We also installed on each node two data files that contain a set of content names and queries that the application on this node must issue. To run the system, we first start the *cameld* on all nodes and let them form a Chord-based overlay network. We then start the application program to issue registrations an queries according to their data files.

6.2.1 Evaluation Results

We conducted several experiments on Planet Lab to verify that the Camel software can work effectively on the Internet across heterogeneous hosts. Here we present some representative results that demonstrate the effectiveness of the distributed load balancing mechanism.

In the first experiment, we assign one node (a node located at CMU) a data file in which all the content names share one common AV-pair. This node registers these names one by one as fast as it can. As one can expect, the system needs to deploy LBMs with multiple partitions to host this registration load.

Figure 6.3 shows the registration rate observed on each partition of the matrix corresponding to this common pair as time proceeds. Each node sets up a registration threshold of $T_{reg} = 1reg/sec$. The matrix starts with 1 partition. As shown in the figure, the rate observed on this partition is about $3.7reg/sec$, much higher than the threshold. As a result, the matrix expands to include other partitions as time progresses. The expansion is done multiplicatively. For example, at time around $80sec$, the rate Partition 4 observes becomes higher than the threshold, and this causes another 4 partitions to be added to the matrix. After the matrix expands to 8 partitions, each partition observes similar registration rate, and their individual load remains under the threshold. This figure is consistent with our simulation results in Figure 3.14, where nodes are homogeneous.

In the second experiment, we have the CMU node issue a set of queries that share one AV-pair, and as a result, these queries must be sent to the matrix corresponding to this AV-pair. The query threshold on each Camel node is also set to be $T_q = 1q/s$. Figure 6.4 shows the query load distribution over time. As we can see from the figure, initially the queries go to the first replica, and the query rate this replica observes is about $1.2q/s$, which exceeds the threshold. The matrix then replicates once to include the second replica. After the second replica is in place, the query rates observed on these two replica are about the same, and both are below the threshold.

6.3 Chapter Summary

In this Chapter, we described Camel, our prototype implementation of the CDS system, and some evaluation results we obtained by deploying it on the the Planet Lab testbed. The deployment of the Camel system on the heterogeneous Internet, along with the experimental results confirmed our CDS design and demonstrated the feasibility of the system. The prototype implementation is joint work with Adam Kushner [43].

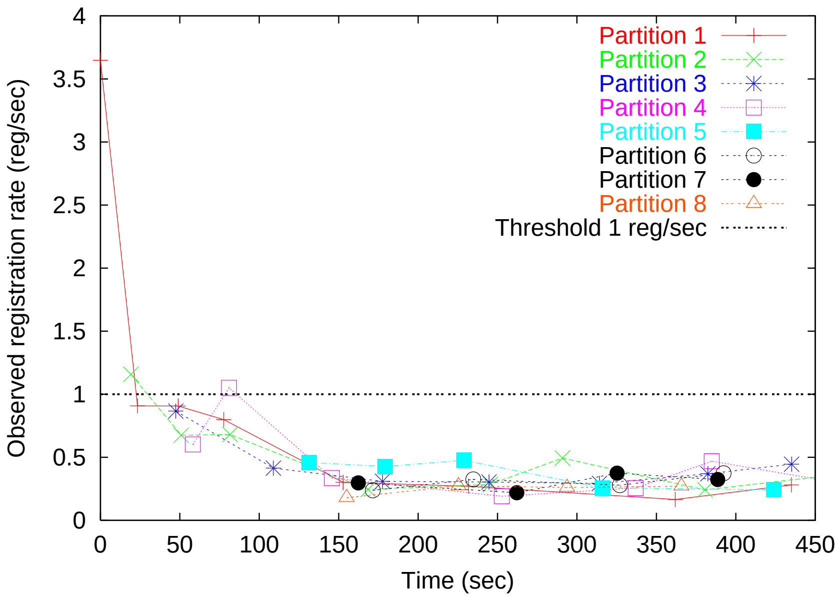


Figure 6.3: Registration load balancing on the PlanetLab.

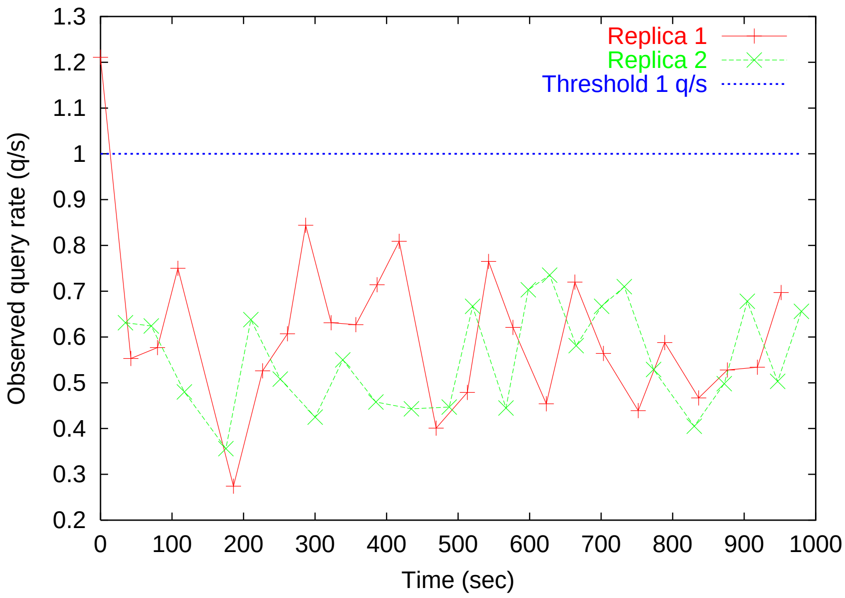


Figure 6.4: Query load balancing on the PlanetLab.

Chapter 7

Conclusions and Future Work

Due to the continued advances in both wired and wireless networking technology coupled with the decreasing cost of computer hardware, several new classes of applications running on the Internet have been emerging rapidly over the last few years. Among them are large scale monitoring services, where inexpensive sensors and cameras are used to monitor conditions such as weather and traffic, and peer-to-peer applications, where a large on-line community can be formed nearly instantaneously to contribute and share an enormous amount of resources.

In this thesis, we studied a central problem faced by these new classes of applications, namely the content discovery problem. The specific question we asked was how to build distributed systems that are both scalable with the number of nodes and the amount of available content, and searchable, in that, users can efficiently find the content that they are looking for. In this chapter, we conclude the thesis by summarizing our contributions and proposing several future research directions.

7.1 Contributions

In this thesis, we demonstrated that it is possible to meet both the scalability and searchability challenges faced by the wide-area content discovery problem. Towards this end, we designed, implemented and evaluated a distributed and scalable content discovery system that supports complex queries efficiently. In particular, we make the following original contributions.

- **Efficient registration and query mechanism that enables search.**

  We designed an efficient registration and query mechanism. A content name is registered with a small number of nodes in the system (the RPs) that equals the number of AV-pairs within the name. To resolve a query, regardless the number of AV-pairs the query may have, we only need to choose 1 AV-pair to find a RP node, and the query cost is 1. As such, resolving a query does not cause any network traffic between RP nodes. This simple yet powerful mechanism enables subset based search, in that a name can always be found using any combination of its AV-pairs as a search criteria. The efficiency for individual registrations and queries means that the CDS scales well as the number of registrations or queries increases.

- **Distributed load balancing**

  One of the fundamental problem faced by a CDS system is that due to the uneven distribution of registration and query load, some nodes may be overloaded while other nodes in the system may still be underutilized. The load imbalance will cause the system to degrade before its full capacity is reached. We developed a fully distributed load balancing mechanism that improves the system's throughput under skewed load by eliminating hot-spots. The load balancing is based on a novel data structure, the load balancing matrix (LBM), where columns and rows are used to share high registration and query load respectively. Each LBM dynamically adjusts its own size in a distributed fashion based on the local load it observes. An important property of the matrix organization of nodes is that it ensures that registrations and queries can still be carried out in a distributed fashion similar to the basic RP system. Our simulations based on realistic loads showed that the extra cost added to registrations and queries due to load balancing remains low.

- **Efficient support for range search**

  We developed an adaptive protocol to support range queries efficiently in the CDS. Our algorithm is based on a distributed data structure, the Range Search Tree (RST), for content registration and query resolution. Nodes at different levels of the RST represent different level of aggregation, and may be used to resolve queries with different range lengths efficiently. To ensure efficiency, we do not instantiate every node in the tree by default, instead, we only use a part of the tree, known as the *band* to accept registrations and resolve queries. The band changes its shape based on the load it observes: A node may be recruited to join the band if its inclusion to the band can lower the overall registration and query cost. The band adaptation is conducted in a fully distributed fashion, since only local information is needed. To ensure endpoints can discover the band efficiently, we designed a lightweight protocol, the Path Maintenance Protocol (PMP), to update the band information in the tree. As a result, registrations and queries are carried out efficiently without having to traverse the tree, and thus no bottlenecks are created.

- **Efficient support for similarity search**

  We developed a distributed kd-tree (DKDT) based mechanism to support similarity searches in the CDS efficiently. In particular, we map a space partitioned kd-tree onto the underlying DHT-based overlay network. In constructing and maintaining the DKDT, we use techniques such as compact splitting and branch coalescing to ensure the tree's size remains manageable as dimensionality increases. We use a lightweight protocol, the tree maintenance protocol (TMP), to propagate the current tree shape to all nodes in the DKDT, so that endpoints can discover the shape of the tree in a distributed fashion, and issue registrations and queries without traversing the tree. In addition, we proposed a novel virtual node shrinking mechanism to mitigate the dimensionality curse problem for datasets with high dimensionality.

In addition to the above system design contributions, we also made contributions to the architectural design of large scale distributed Internet applications by identifying *content* *discovery* as a fundamental building block. We presented a three-layered architecture for distributed applications such as wide area service discovery systems or peer-to-peer object sharing systems. The CDS operates between the underlying DHT layer and the high level application layer. It uses a DHT to form the CDS overlay network and deliver messages, and as such, the network inherits the scalability, robustness, and self-organizing properties from the DHT. The CDS only requires the DHT's minimal `put()` interface to implement all its mechanisms. This clean separation and minimal reliance ensure that the CDS can be deployed on top of any DHT without modifications to the DHT. In turn, the CDS provides a simple `register()` and `query()` interface to higher level applications. Thus, any application that requires the content discovery functionality can use the CDS as a component.

In summary, the CDS we presented in this thesis meets both the scalable and searchable challenges simultaneously. It is scalable with both the amount of registrations and queries, even under highly skewed load distribution. It is searchable by supporting subset-based matching, and providing efficient support for complex queries such as range and similarity queries. Our prototype implementation and the integration of CDS with a content based music information retrieval system demonstrate the effectiveness of the CDS design and its applicability to real world applications.

7.2 Future Work

In this section, we identify several research directions for future work.

CDS functionality inside the DHT layer

In our CDS system, the ultimate responsibility of the CDS is to *locate contents that match a client's query efficiently.* To meet this goal, we designed load balancing mechanisms and support for complex queries, that run on top of the DHT. This separation from the underlying DHT layer gives us great flexibility, in that the CDS may use any DHT available without any modification to the DHT. The downside is that we can not gain the potential benefits had these functionalities been implemented inside the DHT layer.

For example, with replication using the LBM, the system can host high query load while maintaining high throughput. However, from a query issuer's point of view, to reach any of the replicas, it still takes $O(\log N)$ hops, with N being the number of nodes in the system. This is determined by underlying DHT, and our CDS level replication does not reduce this latency. In comparison, recent work [53] shows that by aggressively replicating contents at the DHT level and assuming a certain analytical model of the load, the latency may be reduced to $O(1)$ for a popular query. As future research, it is important to understand the differences between these two approaches, and how they may complement each other.

Locality-aware DHTs

Node IDs in a DHT are typically assigned randomly. While this ensures that each node is in charge of roughly an equal segment in the key identifier space, it sacrifices locality. In particular, two nodes next to each other in the identifier space may be located on different continents physically. The implication for the CDS is that partitions and replicas of the same matrix are distributed randomly in the network. This will become a problem especially when the increase of load comes from local interests. For example, if nodes within a domain all issue the same query due to a local news event, it would make sense to replicate this content on nodes that are within the domain rather than elsewhere.

Recent work [76, 29] has proposed that DHTs can be built using IDs that encode locality information. It is worthwhile to investigate how a CDS can utilize the locality aware DHT to improve its performance.

Rich set of CDS applications

The CDS we presented is suitable for applications such as distributed monitoring, service discovery and peer-to-peer object sharing systems. There are many distributed applications that could benefit from the CDS. Examples of such applications include distributed databases and information retrieval systems. Due to the sophisticated operations required by these applications, new mechanisms may have to be designed and incorporated into the CDS.

We look at a large scale information retrieval system as an example. It appears that the CDS is a perfect fit to handle the enormous amount of information offered by all the peers, since it would be difficult if not impossible for any centralized system to digest all the contents. However, implementing full-fledged content-based information retrieval using the CDS poses a difficult problem. For example, a document may be represented using a vector with hundreds of dimensions. Our kd-tree based similarity search mechanism will not work well for this many dimensions. Some recent work [65] tried to address this problem, but this system is rather unsatisfactory in that it relies heavily on heuristics. As we mentioned earlier, locality sensitive hashing works well even for high dimensionality datasets. We believe that incorporating LSH [24] mechanism into the CDS is a promising approach to address information retrieval in high dimensionality.

Bibliography

[1] Google, other engines hit by worm variant. http://news.zdnet.con/2100-3513_22-5283750.html.

[2] The network simulator - ns-2. http://www.isi.edu/nsnam/ns.

[3] Project IRIS. http://iris.lcs.mit.edu.

[4] FIPS 180-1. Secure Hash Standard. Technical Report Technical Report Publication 180-1, Federal Information Processing Standard(FIPS), National Institute of Standards and Technology, Washington DC, April 1995.

[5] Karl Aberer. P-Grid: a Self-Organizing Access Structure for P2P Information Systems. In *Proceedings of the Sixth Intenational Conference on Cooperative Information Systems*, Trento, Italy, 2001.

[6] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The Design and Implementation of an Intentional Naming System. In *Proceedings of SOSP 1999*, Kiawah Island, SC, December 1999.

[7] Marcos K. Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar D. Chandra. Matching events in a content-based subscription system. In *Principles of Distributed Computing*, 1999.

[8] A. Andrzejak and Z. Xu. Scalable, Efficient Range Queries for Grid Information Services. In *Proceedings of P2P 2002*.

[9] S. Arya, D. Mount, and N. Netanyahu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. In *Proceedings of SODA*, 1994.

[10] M. Balazinska, H. Balakrishnan, and D. Karger. INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. In *Proceedings of Pervasive 2002*, Zurich, Switzerland, August 2002.

[11] N. Beckmann, H. Schneider, and B. Seeger. The r*-tree: an eficient and robust access method for points and rectangles. In *Proceedings of SIGMOD*, 1990.

[12] J.L. Bentley. Multidimensional binary search used for associative searching. *ACM Comm.*, 1975.

[13] Sergey Brin, Larry Page, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report Stanford Digital Libraries, Stanford University, 1998.

[14] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[15] M. Castro, P. Druschel, A-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralised publish-subscribe infrastructure. Submitted, September 2001.

[16] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making Gnutella-like P2P Systems Scalable. In *Proceedings of SIGCOMM 2003*.

[17] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz. An Architecture for a Secure Service Discovery Service. In *Proceedings of Mobicom 99*, Seattle, WA, August 1999.

[18] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area Cooperative Storage with CFS. In *Proceedings of SOSP 2001*, Banff, Canada, October 2001.

[19] S. Deering. Multicast routing in internetworks and extended lans. In *Proceedings of SIGCOMM 1988*, 1988.

[20] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. Liu, and L. Wei. The PIM Architecture for Wide-area Multicast Routing. *ACM Transactions on Networks*, April 1996.

[21] D. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[22] Elvin. http://elvin.dstc.edu.au/.

[23] K. B. Erickson, R. E. Ladner, and A. LaMarca. Optimizing static calendar queues. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 10(3):179–214, 2000.

[24] P. Indyk et al. Similarity search in high dimensions via hashing. In *Proceedings of the VLDB*, 1999.

[25] Freenet. http://freenet.sourceforge.net/.

[26] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, 1977.

[27] V. Gaede and O. Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30(2), June 1998.

[28] J. Gao, G. Tzanetakis, and P. Steenkiste. Content-Based Retrieval of Music in Scalable Peer-to-Peer Networks. In *Proceedings of ICME 2003*, Baltimore, MD, July 2003.

[29] Luis Garces-Erice, Keith W. Ross, Ernst W. Biersack, Pascal A. Felber, and Guillaume Urvoy-Keller. Topology-centric look-up service. In *Proceedings of COST264/ACM Fifth International Workshop on Networked Group Communications (NGC)*, 2003.

[30] Gnutella. http://gnutella.wego.com/.

[31] Google. http://www.google.com/.

[32] S. Gribble, E. Brewer, J. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of OSDI 2000.*

[33] A. Gupta, D. Agrawal, and A. El Abbadi. Approximate Range Selection Queries in Peer-to-Peer Systems. In *Proceedings of CIDR 2003.*

[34] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service Location Protocol. IETF, RFC 2165, November 1998.

[35] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica. Complex Queries in DHT-based Peer-to-Peer Networks. In *Proceedings of IPTPS'02.*

[36] N. Harvey, M. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A Scalable Overlay Network with Practical Locality Properties. In *Proceedings of USITS'03.*

[37] An-Cheng Huang and Peter Steenkiste. Network-sensitive service discovery. In *Proceedings of USITS '03.*

[38] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with pier. In *Proceedings of the 29th VLDB*, 2003.

[39] Brad Karp, Sylvia Ratnasamy, Sean Rhea, and Scott Shenker. Spurring Adoption of DHTs with OpenHash, a Public DHT service. In *Proceedings of IPTPS 2004.*

[40] Norio Katayama and Shinichi Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proceedings of the ACM SIGMOD 1997*, May 1997.

[41] Kazza. http://www.kazaa.com/.

[42] Y. Ke and R. Sukthankar. Pca-sift: A more distinctive representation for local image descriptors. In *Proceedings of Computer Vision and Pattern Recognition (CVPR)*, 2004.

[43] Adam Kushner. Project camel: Implementation and evaluation of a distributed content discovery system. Technical Report Undergraduate Honor Thesis, Department of ECE, Carnegie Mellon University, May 2004.

[44] X. Li, Y. Kim, R. Govindan, and W. Hong. Multi-dimensional Range Queries in Sensor Networks. In *Proceedings of SenSys'03.*

[45] P. Mockapetris. Domain Names - Concepts and Facilities. IETF, RFC 1034, November 1987.

[46] P. Mockapetris. OSPF version 2. IETF, RFC 2328, April 1998.

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

[48] T. S. Eugene Ng and Hui Zhang. Predicting internet network distance with coordinates-based approaches. In *Proceedings of INFOCOM'02*.

[49] Y. Ohsawa and M. Sakauchi. Bd-tree: a new n-dimensional data structure with efficient dynamic characteristics. In *Proceedings of the 9th world computer congress*, 1983.

[50] J. A. Orenstein. Multidimensional tries used for associative searching. *Information Processing Letters*, 14(4):150–157, June 1982.

[51] PlanetLab. http://www.planet-lab.org/.

[52] The Chord Project. http://www.pdos.lcs.mit.edu/chord/.

[53] V. Ramasubramanian and E. G. Sirer. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *Proceedings of NSDI 2004*.

[54] S. Ratnasamy, J. Hellerstein, and S. Shenker. Range Queries over DHTs. Technical Report IRB-TR-03-009, Intel Corp., June 2003.

[55] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-Addressable Network. In *Proceedings of SIGCOMM 2001*, pages 161–172, San Diego, CA, August 2001.

[56] P. Reynolds and A. Vahdat. Efficient Peer-to-Peer Keyword Searching. In *Proceedings of Middleware 2003*, Rio de Janeiro, Brazil, June 2003.

[57] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling Churn in a DHT. In *Proceedings of USENIX 2004*.

[58] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. In *Proceedings of Middleware 2001*, Heidelberg, Germany, November 2001.

[59] Antony Rowstron and Peter Druschel. Storage Management and Caching in PAST, a Large-scale, Persistent Peer-to-Peer Storage Utility. In *Proceedings of SOSP 2001*.

[60] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1989.

[61] Cristina Schmidt and Manish Parashar. Flexible information discovery in decentralized distributed systems. In *Proceedings of HPDC-12*, Seattle, WA, June 2003.

[62] SETI@home. http://setiathome.ssl.berkeley.edu/.

[63] Kunwadee Sripanidkulchai. The Popularity of Gnutella Queries and Its Implications on Scalability. http://www.cs.cmu.edu/~kunwadee/research/p2p/gnutella.html.

[64] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of SIGCOMM 2001*, pages 149–160, San Diego, CA, August 2001.

[65] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-Peer Information Retrieval Using Self-organizing Semantic Overlay Networks. In *Proceedings of SIGCOMM 2003*.

[66] Chunqiang Tang and Sandhya Dwarkadas. Hybrid global-local indexing for efficient peer-to-Peer Information Retrieval. In *Proceedings of NSDI 2004*.

[67] David Thaler and Chinya V. Ravishankar. Using Name-Based Mappings to Increase Hit Rates. *IEEE/ACM Transactions on Networking*, 6(1):1–14, 1998.

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

[69] George Tzanetakis, Jun Gao, and Peter Steenkiste. A scalable peer-to-peer system for music information retrieval. *Computer Music Journal*, 28(2):24–33, June 2004.

[70] M. Wahl, T. Howes, and S. Kille. Lightweight Directory Access Protocol (v3). IETF, RFC 2251, December 1997.

[71] D. Waitzman, C. Partridge, and S. E. Deering. Distance vector multicast routing protocol. IETF, RFC 1075, November 1988.

[72] L. Wang, V. Pai, and L. Peterson. The Effectiveness of Request Redirection on CDN Robustness. In *Proceedings of OSDI 2002*, Boston, MA, Dec. 2002.

[73] R. Weber, H.J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of VLDB'98*, pages 194–205, August 1998.

[74] H. Yu, D. Estrin, and R. Govindan. A Hierarchical Proxy Architecture for Internet-scale Event Services. In *Proceedings of WETICE 99*, Stanford, CA, June 1999.

[75] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB/CSD-01-1141, U. C. Berkeley, April 2001.

[76] Shuheng Zhou, Greg Ganger, and Peter Steenkiste. Balancing locality and randomness in dhts. Technical Report CMU-CS-03-203, Carnegie Mellon University, Nov. 2003.