ABSTRACT

Title of dissertation: SYSTEM SUPPORT FOR KEYWORD-BASED SEARCH IN STRUCTURED PEER-TO-PEER SYSTEMS

Vijay Gopalakrishnan, Doctor of Philosophy, 2006

Dissertation directed by: Dr. Bobby Bhattacharjee and Dr. Pete Keleher
Department of Computer Science

In this dissertation, we present protocols for building a distributed search infrastructure over structured Peer-to-Peer systems. Unlike existing search engines which consist of large server farms managed by a centralized authority, our approach makes use of a distributed set of end-hosts built out of commodity hardware. These end-hosts cooperatively construct and maintain the search infrastructure.

The main challenges with distributing such a system include node failures, churn, and data migration. Localities inherent in query patterns also cause load imbalances and hot spots that severely impair performance. Users of search systems want their results returned quickly, and in ranked order. Our main contribution is to show that a scalable, robust, and distributed search infrastructure can be built over existing Peer-to-Peer systems through the use of techniques that address these problems. We present a decentralized scheme for ranking search results without prohibitive network or storage overhead. We show that caching allows for efficient query evaluation and present a distributed data structure, called the View Tree, that enables efficient storage, and retrieval of cached

results. We also present a lightweight adaptive replication protocol, called `LAR` that can adapt to different kinds of query streams and is extremely effective at eliminating hotspots. Finally, we present techniques for storing indexes reliably. Our approach is to use an adaptive partitioning protocol to store large indexes and employ efficient redundancy techniques to handle failures. Through detailed analysis and experiments we show that our techniques are efficient and scalable, and that they make distributed search feasible.

SYSTEM SUPPORT FOR KEYWORD-BASED SEARCH IN STRUCTURED PEER-TO-PEER SYSTEMS

by

Vijay Gopalakrishnan

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2006

Advisory Committee:
Professor Bobby Bhattacharjee, Chair/Advisor
Professor Pete Keleher, Co-Advisor
Professor Alan Sussman
Professor Neil Spring
Professor Mark Shayman

DEDICATION

To my grandparents, parents, and sister. I owe my everything to you.

ACKNOWLEDGMENTS

This dissertation marks the end of six eventful years for me at the University of Maryland. During this period, I have interacted with a lot of people. This thesis would not been possible without the help of all these people. For this and much more, I am forever in their debt.

I am extremely grateful to my advisor, Bobby Bhattacharjee, for constantly encouraging me to pursue a Ph.D. I also thank him for giving me an opportunity to work on interesting projects, for his keen and active interest in my progress, for providing me with financial support throughout my stay here, and above all, for all the advice he has given me. He has been available for help and advice and there has never been an occasion when I've knocked on his door and he hasn't given me time. It has been a privilege to work with and learn from such an extraordinary individual.

I would also like to thank my co-advisor, Pete Keleher. Pete brought with him a certain calmness and clarity, without which this thesis would have been a distant dream. The discussions we have had have always been engaging and his thoughts insightful. I truly appreciate his help in improving the quality of my work. Thanks are due to Professor Alan Sussman, Professor Neil Spring and Professor Mark Shayman for agreeing to serve on my thesis committee and for sparing their invaluable time reviewing the manuscript.

My research has benefited tremendously from my interactions with my colleagues in the department. My thanks to Bujor Silaghi, Rob Sherwood, Ruggero Morselli, Se-

ungjoon Lee, Christopher Kommareddy, Ray Chen, Dave Levin and Christian Lumezanu: they were the ideal lab mates. I thank Arunchander Vasan, Arunesh Mishra, Indrajit Bhattacharya and Srinivasan Parthasarathy for their discussions, companionship and encouragement during the tough periods of my graduate study. I thank Suman Banerjee for his advice and encouragement during the early part of my stay. It is unfortunate that I did not get a chance to actually work with him.

I have had several house mates during my stay at Maryland. Their friendship, support and understanding has been crucial in the successful completion of my dissertation. My gratitude to Christopher Kommareddy, Sadagopan Srinivasan, Arunchander Vasan, Narendar Shankar, Harish Gadde, Deepak Agarwal, Vinay Shet, Ishan Banerjee, Sameer Sirdhonkar, Alap Karapurkar and Prithviraj Sen.

Last, but not least, I would like to express my earnest gratitude to my family for their love and support, without which none of my achievements would have been possible. Thanks to my late father who sparked my early interest in science and engineering, my mother for her love and countless sacrifices to raise and to give me the best possible education, and my sister for always encouraging me and seeing things from the lighter side. I would be remiss if I forget to thank my grandparents who were there for us during the darkest of times. Their constant encouragement has given me the strength to overcome any difficulties and to maintain high goals. I also thank my uncles, aunts and cousins for all their help during my stay here and for ensuring that I never felt away from home, despite the distance.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Advances in technology makes it possible to store and distribute large amounts of data and information to users. Users, however, need some way of *searching* through the data to identify information relevant to them. Search helps convert the large amounts of stored data into information for the users. Since early 1950s, researchers have studied the problem of automating the search process. Typical approaches allow users to specify a set of keywords that best describe the objects, and use these keywords to identify the set of relevant results. Over the years, support for keyword-based search has been incorporated into systems ranging from desktop file systems to online catalogs of libraries hosting large collections of documents. Since the early nineties, the world wide web has steadily grown in importance as a means of distributing and obtaining information, resulting in an explosive growth in the amount of data available to the users. This growth has only heightened the importance of efficient and effective search techniques that can handle this large scale of data.

The current approach used to build search systems involves a centralized approach. Web search, which is probably the most challenging in terms of scale, is provided by entities specializing in search. These entities run server farms that receive search queries from users, evaluate the queries, and return results. These servers represent a single centralized location to which all clients send requests. Internally, however, these search entities

make use of thousands of machines working in parallel, each responsible for a certain set of tasks. The entity providing the search service is centralized, in that it has complete authority and control over these machines. The entity can decide how to partition data across machines, provision for the different tasks depending on their complexity and requirements, control the assignment of tasks to machines to balance load, and can provision for the traffic generated by the various internal tasks.

In this dissertation, *we present protocols for building a keyword-based search infrastructure over a wide-area, distributed system*. Unlike existing search systems that are either centralized or comprise of large server farms, we consider systems consisting of communities of distributed end-hosts. These end-hosts connect with each other to form a virtual network, over which one can deploy various distributed applications. These end-hosts co-operate with each other, providing processing power, bandwidth, and storage necessary to run the different applications. An important distinction of these systems is that the construction and maintenance of these systems is completely decentralized; there is *no central authority* that exerts controls over these systems. We discuss the challenges involved with building such systems and present efficient techniques to address these challenges.

1.1 Decentralizing Search

There are many reasons that make studying the problem of distributed search interesting. The Peer-to-Peer (P2P) paradigm of computing has recently received much attention. P2P systems comprise entirely of end-host that connect with each other to form

a network over which different distributed applications can be run. Distributed applications including file systems [21, 62], file-sharing systems [31, 46], and archival storage systems [73] have been designed for these environments. An important requirement in all these systems is the ability to search for content. Technically, we could employ centralized search strategies by designating one node as being responsible for providing the search. However, it is not reasonable to employ centralized search systems in such inherently distributed environments. Such an approach would result in the peer getting overloaded. Further, there are the issues of scalability and fault-tolerance with using just one node.

Such a distributed search infrastructure could also be used to provide web searches. Entities that currently provide web search have complete control over the set and the order of results returned to each query. Hence, they have the ability to censor or tamper with the results that are returned to each query. Users have little or no control over these decisions. Distributed search infrastructures, by virtue of their collaborative nature, are resistant to such censorship. While it is true that malicious participants can tamper or corrupt data, we believe that there exist cryptographic techniques that can be employed to guard against such attacks.

Above all, our goal is to explore the limits of current decentralized techniques, and present a new architecture which we believe will enable a truly decentralized and distributed search infrastructure. The advances in hardware technology have made such a distributed application possible. Current day machines have more storage space and idle cycles for processing than is demanded by these applications. Further, the last-hop bandwidth has not only increased, but has also become cheaper, and thereby making high

bandwidth available and affordable to end-hosts. Systems such as Google have demonstrated the immense power of commodity cluster computing coordinated using centralized overview. It is fair to view our work as an attempt at an extension in which we dispense with the centralized control.

1.1.1 Constructing a Decentralized System

While there are compelling reasons for a distributed search system, there are non-trivial challenges involved in building such systems. Co-ordinating and maintaining these distributed, and possibly failure prone machines can be extremely challenging. Since the underlying system comprises of end-hosts, it is going to be extremely dynamic with frequent node arrivals and departures. However, the connectivity between the participants and the data stored on them needs to be maintained for the system to functional.

In order to answer queries, we need some way of associating keywords with objects that contain those keywords. Data structures such as *inverted indexes* are useful for this purpose[1]. Indexes are maintained for each keyword and contain a list of all the objects that contain that keyword. Additionally, each index could also maintain data that is useful for evaluating results or maintaining the index. Such data includes statistics about the index, its entries, etc. Figure 1.1 shows an example of inverted indexes. In this example, there are four documents labeled *Doc_1* through *Doc_4*, each containing a few keywords. As shown in the figure, an inverted index is maintained for each keyword, with each index containing a list of all documents having that particular keyword. For example, the index for *term_a* contains the set of documents that have *term_a* as a keyword, i.e., *Doc_1, Doc_3*

[1]We use the terms *inverted indexes* and *indexes* interchangeably.

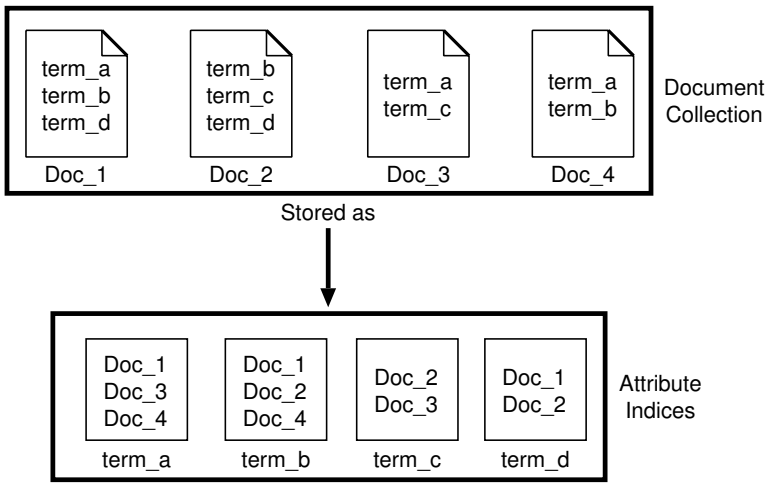


Figure 1.1: An example depicting the contents of inverted indexes.

and *Doc_4*. Typical search systems index and maintain large amounts of data to facilitate quick and accurate retrieval of results. For example, Google crawls billions of pages and maintains indexes on all these pages in order to answer search queries[2]. In a distributed system, all these indexes have to be distributed over the participating peers in such a way that it still facilitates efficient retrieval. Depending on how the indexes are constructed and stored there are two possible approaches: local indexing and global indexing.

Local Indexing As the name suggests, in this approach, each peer maintains indexes for content that is stored locally. This approach was pioneered by the file sharing systems such as Gnutella [31] and KaZaa [46]. In this approach, nodes connect to a random set of peers already participating in the network to form what is popularly known as an *unstructured* P2P network. In order to search for content, users submit queries consisting of keywords. These queries are then flooded to the peers' neighbors, who evaluate the

[2]Google no longer publishes the number of indexed pages on its home page; searching for the wild-card "$\star\star$" however returns 11.2B results.

query against their local indexes to see if they have entries that can be used to answer the query. If any valid results exist, they are returned to the peer that initiated the query. These neighbors, then flood the query to all their neighbors, who then repeat the process.

The primary advantage of this approach is its simplicity. Nodes only need to connect to a few random peers. There isn't much of state or data that is exchanged when a node joins the network. This makes the setup of the network very simple. Indexing the data is also very simple; the node has to read the local data and build indexes on that data. Finally, answering queries only requires looking at the local indexes and returning results, if any exist.

This approach, however, has severe shortcomings. Queries are answered by flooding; hence, it is possible that all participating nodes need to be contacted to get all the possible results. Since this is infeasible in a large system, the typical strategy is to limit the number of hops the query travels. This, of course, means that not all possible results can be returned. In fact, it is possible that certain queries do not return any results even though there exist valid results in the system. Further, users of a search system want the results to be returned quickly, and in ranked order, with the most relevant results being returned first. With flooding, however, results can only be returned when a node storing some results is located. Also, in order to rank the results, all possible results to the query must be known. Finally, flooding does not scale with the number of queries, consumes excessive network bandwidth, and there is lots of wasted work in terms of duplicates. For these reasons, we believe that such a design is unsuitable for a distributed search infrastructure.

1.2 Search using Global Indexing

Upon close observation, it is clear that the inefficiencies in systems such as Gnutella and KaZaa come from the fact that there is no organization in the way data is stored. Researchers have attempted to design alternate approaches for constructing similar systems. Until recently, most of the research was directed towards designing P2P systems that facilitate efficient content location. These include distributed hash table (DHT) based systems such as Chord [80], CAN [68], Pastry [72] and Tapestry [89], or hierarchical systems such as TerraDir [7] and P-Grid [1]. These systems impose a structure in the way nodes are organized and this structure determines where a particular object is stored. For this reason, these systems are known as *structured* P2P systems. Protocols, such as those based on DHTs, map the participating nodes and data to a large identifier space. The node whose identifier is closest to that of an object is responsible for storing that object. Given the identifier of an object, these systems can quickly and efficiently locate the node storing the identifier (typically in $O(log_2 N)$ steps, where N is the number of participating nodes).

Global indexing, originally proposed by Reynolds and Vahdat [70], makes use of this ability of structured systems to organize data over a large number of hosts. In global indexing, an single index is maintained globally for each keyword. For example, the entire index for `Maryland` is maintained at a single location rather than at all nodes having objects with the keyword. Each peer is now responsible for a subset of the total set of indexes, rather than just the index of local data. The peer responsible for an index is identified by mapping the keyword into the identifier space. For example, in a DHT based system such as Chord, the index for `Maryland` would be stored in the peer responsible

for the key $k = \text{Hash}(\texttt{Maryland})$.

This approach has quite a few desirable properties. Global indexes can be created and maintained incrementally, with peers inserting the entries of each object as and when these objects are introduced into the system. Distributing index entries in this manner allows them to be managed using the services provided by the P2P system for data objects. Among these services are transparent caching and replication, which are used to provide fault-tolerance and high availability. Answering queries involves just fetching the index; this is easily implemented using the look-up procedure provided by the underlying P2P infrastructure. Unlike local indexing, with this approach, the query is routed to the node storing the index and all possible results can be returned. Further, since all the results are stored at a single location, it is possible to return results incrementally, possibly using techniques such as ranking. Finally, it also allows for quick retrieval of results because the underlying system locates the node storing the index quickly.

1.2.1 Practical Challenges with Global Indexing

Structured P2P systems, in theory, present a great platform over which one can build a distributed search infrastructure. However, majority of the peers connect over shared links and share resources including disk space and processing power. P2P systems are also very dynamic; peers are expected to frequently arrive, depart and fail. This unique setting brings with it many challenges that system designers need to address. These challenges include:

- **Efficiently computating results:** Distributed indexes can be used to answer single-

keyword queries without resorting to flooding. In order to answer a query, the node holding the index for the keyword in the query needs to be identified. This is straightforward: the keyword needs to be mapped to an identifier and the underlying P2P infrastructure can be used to identify the node storing the identifier. Answering multi-keyword queries, however, is expensive. Queries, with keywords connected by disjunctions cannot be optimized much. The union of the two or more indexes has to be transferred to the initiator of the query. The evaluation of conjunctive queries with two or more keywords, on the other hand, requires the entire index for one or more keywords to be sent across the network. Unfortunately, indexes may be extremely large, as DHTs are intended for use with extremely large distributed data sets (data sets with up to 10^{14} objects have been discussed [42]). Further, keywords are often skewed according to Zipf distributions [91], potentially resulting in the size of at least some indexes growing linearly with the number of system objects, and hence becoming quite large. If this system is to serve billions of queries per day [3], then the overhead of these intersections will prove to be prohibitive.

- **Avoiding network hotspots:** Existing DHT-based structured systems [80, 89, 68, 72] randomize the mapping between data item names and locations in an attempt to balance the load on individual nodes. Under an assumption of uniform demand for all data items, the number of items retrieved from each node (i.e., the *destination load*) will be balanced. Further, the load incurred by peers due to routing queries

[3]This is a conservative estimate for Google. While an official count seems difficult to locate, an industry article in February 2003 states that Google was serving 250M queries per day within the US (http://searchenginewatch.com/reports/article.php/2156461).

(i.e., *routing load*) will be balanced as well. However, if the popularity of keywords is non-uniform, which is typically the case in practice, neither routing nor destination load will be balanced, and may be arbitrarily bad. The situation is even worse for hierarchical systems such as TerraDir [7], as the system topology is inherently non-uniform, resulting in uneven load across peers. If left unaddressed, this could result in a large number of queries getting dropped at the overloaded node. Load imbalances are also unfair on the peer hosting the popular index. It acts as a disincentive for the peer to participate in the system. Finally, a system that cannot answer popular queries is of not much use to users.

- **Storing large indexes:** With billions of documents, many individual indexes can be very large. Indexes for common words often contain hundreds of millions of entries (369M entries for `Apple`), and indexes for even specialized terms contain hundreds of thousands of entries (6.4M entries for `Terrapins`). Conservatively, each index entry requires 26 bytes (20 bytes for a document ID, and 6 bytes for a IPv4 address, port pair), leading to index sizes ranging from tens of megabytes for specialized terms to tens of gigabytes for popular terms. Since there is no control over which indexes gets mapped to which host, it is likely that the hosts holding the indexes for popular terms will be under-provisioned. Also, inverted indexes, especially of popular terms, can become to be too big to be stored at a single node. Finally, a node can run out of space because of the number of different indexes being stored in it. When a node runs out of space, indexes may not be stored in the system or might be incomplete. The consequence of not storing indexes is that queries with

keywords associated with these non-existent indexes cannot be answered.

- **Providing service despite churn:** P2P systems consist of end-hosts. Hence, such systems will have frequent peer arrivals and departures. This has two significant consequences: (a) the entries stored in indexes change with every arrival and departure, and (b) indexes need to be transferred or re-constructed with every arrival and departure. In particular, when a node departs, the indexes stored in this node have to transferred to some other existing node. If indexes are not transferred or re-constructed, it becomes impossible to answer queries containing the keywords associated with these indexes. Current approach to prevent loss of indexes is to statically replicate the index on a few nodes. However, in order to handle high rates of churn, a large number of replicas are required. Further, indexes can become very large and statically creating a large number of replicas for these indexes requires large amounts of disk space. Hence, techniques that guarantee similar levels of reliability, while requiring lesser space are needed.

- **Ranking results:** Current systems using inverted indexes support boolean queries, i.e., queries consisting of keywords connected by boolean connectives. However, boolean queries containing popular keywords return unordered and unmanageably many results. Even seemingly specific queries can return a very large number of mostly useless results, e.g., in March 2006, the query "University of Maryland" matched over 189M web pages indexed by Google. Ideally, query results would be *ranked*, and only relevant results returned to the user. Users are only interested in the most relevant results rather than an unordered set of all possible results. Rank-

ing also helps return fewer results, which in turn reduces the network bandwidth consumed and helps scale the system. While the advantages to prioritizing results are relatively obvious, techniques to enable them are not. Ranking results is essentially a global operation: *all* documents (matching a query) have to be compared with respect to each other. In a large, completely distributed system, the lack of a central location to aggregate global knowledge makes the problem of ranking search results challenging.

1.3 Our Contributions

Structured P2P systems, in combination with global indexing, presents an ideal platform over which one can build a distributed search infrastructure. Many of the issues associated with building a distributed search system, such as mapping indexes to nodes and the ability answer queries quickly are readily handled. However, as discussed in the previous section, a naive implementation has severe debilitating problems that still need to be addressed before the infrastructure can actually be deployed. These challenges arise because of (a) the stringent requirements of a useful search infrastructure, and (b) the environment in which deployments are expected to run in practice. In this dissertation, we assume the existence of a distributed search system consisting of global indexes, built over a structured peer-to-peer system. We then explore some of the practical challenges associated with this design and present efficient solutions to address these challenges. In particular, we make the following contributions:

- We describe a method of efficiently and consistently *ranking* search results in a

completely distributed manner, for an arbitrary set of documents. Using an approximation technique based on uniform random sampling, we extend a class of ranks based on the classic Vector Space Model (VSM) [76] originally due to Gerard Salton et al. We show how to generate the information needed for ranking, how to store this information in a distributed manner, and how to evaluate and rank the queries in a distributed manner. The most novel contribution of our work is to demonstrate that sampling and VSM compose well, i.e. sampling enables distributed ranking with very little network overhead. Our analysis shows that, under reasonable conditions, the cost of our sampling-based algorithm is small and *remains constant as the size of the system increases*.

- We present a distributed data structure called the *View Tree*. We show how the view tree can be used to cache results of queries and re-use these results to answer queries. In particular, view trees permit efficient evaluation of conjunctive queries. Non-conjunctive queries can be evaluated by converting the query into disjunctive normal form and evaluating the conjunctive components generated. Previous studies (e.g.,[78]) have indicated a Zipf-like distribution for queries in P2P systems such as Gnutella. Our approach attempts to efficiently exploit locality in Zipf-like query streams and object attributes to cache and re-use results between queries.

- The standard approach to handle hotspots is to replicate the objects proportional to their popularity. It is, however, hard to predict the popularity of objects *a priori*. Further, in a dynamic system, the popularity of objects changes over time, sometimes rather suddenly. To address this issue, we propose a lightweight adap-

tive replication protocol called LAR. LAR uses locally available load information to create replicas of popular objects and alleviate load. LAR is robust to query distribution and adapts to changes in popularity very quickly. LAR incurs much lower overhead compared to existing approaches, can balance load at a fine granularity, accommodates servers with differing capacities, and is relatively independent of the underlying P2P structure.

- We describe a partitioning scheme that adaptively distributes large indexes over multiple nodes. Our design of the partitioning scheme is motivated by three guidelines: (1) Intersections of sets of object identifiers (required for answering conjunctive queries) should be efficient. (2) The overhead of updating indexes in response to insertion of new objects and other changes should be small. (3) The partitioning scheme should not depend strongly on the underlying P2P object-location architecture. We also analyze the trade-offs between static replication and more advanced techniques like erasure codes for guaranteeing reliable storage. We show that under high failure rates, erasure codes offer the same levels of reliability while requiring lesser disk space.

The rest of this dissertation is structured as follows. We first present some background on Peer-to-Peer systems and on existing approaches for building distributed search over P2P systems in Chapter 2. We then describe our approach for distributed ranking of search results in Chapter 3. In Chapter 4, we describe the View Tree data structure and show how it can be used for caching search results. We present our Lightweight Adaptive Replication algorithm in Chapter 5. In Chapter 6, we discuss how indexes can be stored

reliably. We discuss our partitioning algorithm and present analysis on trade-offs of using replication versus erasure codes. We discuss some future directions and conclude in Chapter 7.

Chapter 2

Background and Related Work

Our work builds on existing work on Peer-to-Peer systems and on search and lookup in these systems. In the rest of the chapter, we document the different approaches to build Peer-to-Peer systems and distributed search infrastructures. We first discuss unstructured and structured systems in Section 2.1. Since we assume that our search system is built over a structured P2P system, we describe Chord [80] and TerraDir [7], two representative structured systems in Sections 2.1.1 and 2.1.2 respectively. In section 2.2, we describe existing designs to perform search over structured systems and point out the advantages and disadvantages of each design. There is significant amount of work related to the different components of a distributed search system, i.e., ranking, caching, load-balancing, reliable storage, etc. We discuss these related approaches in Sections 2.2.1, 2.2.2, 2.2.3 and 2.2.4.

2.1 Peer-to-Peer systems

Increasing processing power, memory, storage space, and network bandwidth in end-hosts has resulted in a perceptible shift in the design and use of wide-area distributed systems consisting primarily of end-hosts. Participating end-hosts cooperate with other to construct and maintain a virtual network over which different distributed applications can be deployed.

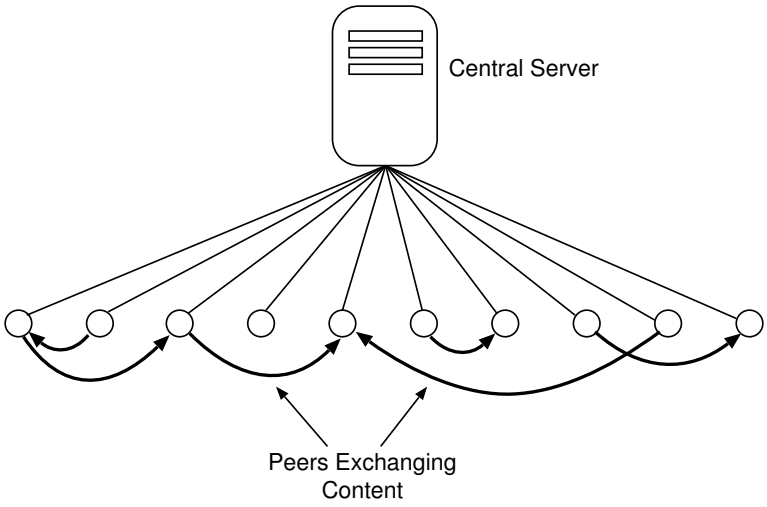


Figure 2.1: Architecture of the Napster file sharing system

Distributed file-sharing ranks as the most widely deployed application utilizing the Peer-to-Peer paradigm. Napster [63], which was the first P2P application to be widely deployed and used, adopted a hybrid approach for file and data sharing. The system consisted of a central server that was used for mapping available data to providers, and peers exchanging data directly with each other. An example of this architecture is shown in Figure 2.1. Users ran clients on their machines and shared data locally. These clients connected to central server that indexed the data shared by users participating in the system. While searching for content, users issued queries to the central server. The central server evaluated queries using the indexed data and identified results relevant to each query. Results returned included information about the available data and peers sharing them. Peers initiating queries, then contacted and fetched the data directly from the peer sharing it. The problem with this approach is obvious. The central server is responsible for answering queries of all peers. Hence it has to be provisioned to store all the informa-

tion, and handle the load generated by these connected clients. This central server is also a single point of failure. To overcome this limitation, purely decentralized file sharing applications were designed and deployed.

Gnutella [31] and KaZaa [46] were, by far, the most popular decentralized file sharing systems. In these protocols (typically classified as *unstructured* systems), peers connected to a random set of peers already connected in the system. Content shared by users was stored locally. Peers exporting content, maintained indexes on the content exported. In order to search for content, users submitted queries consisting of keywords. Queries were evaluated by flooding. Peers forwarded the query to each connected neighbor, who evaluated the query against the local indexes to identify relevant content. The neighbors then forwarded the query to each of their neighbors, who repeated the process. In order to reduce the overhead flooding, these systems limited the number of times these queries were forwarded.

Newer versions of these systems moved to a 2-level design in order to further reduce the overheads associated with flooding. In these designs, few peers were given the elevated status of *SuperPeer* or *UltraPeer*. Participating peers connected to one of these SuperPeers. SuperPeers maintained connections with a few other SuperPeers and maintained indexes for the content shared by all the peers connected to it. Queries are forwarded by each peer to the associated SuperPeer, which then flooded the query to other SuperPeers. Figure 2.2 shows an example Gnutella network with SuperPeers.

Despite this change in design, flooding remained the primary form of searching for content. Flooding, however, prevents the system from scaling. When there are a large number of nodes, flooding the query to all the nodes in the system becomes very expen-

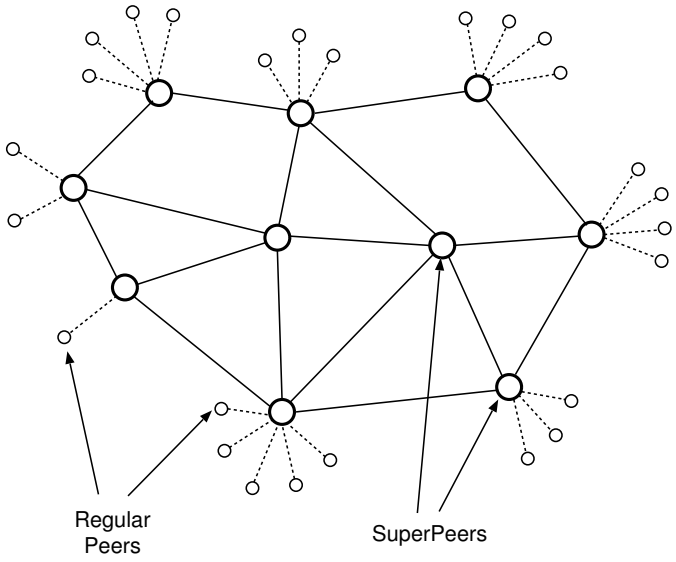


Figure 2.2: Architecture of the Gnutella file sharing system

sive. Since the query is only flooded to a few nodes in the system, these systems cannot guarantee that a query will return results, even though there may exist valid results to the query. A lot of effort has since been spent on trying to come up with designs that allow for efficient retrieval of content. All these efforts typically arrange the participating peers into a fixed topology and map content to specific peers so that they can be efficiently retrieved. Since there is an organization in the way nodes and data are arranged, these systems are classified as *structured* P2P systems. Typical structured systems use Distributed Hash Tables or a hierarchical design. While TerraDir [7] and P-Grid [1] are examples of hierarchical P2P systems, Chord [80], CAN [68], Pastry [72] and Tapestry [89] are prototypical hash-based peer-to-peer systems. There are also a number of other recent efforts in this area including Viceroy [57], Skipnet [41], Koorde [43], Kelips [38], etc. which provide similar or better latency bounds.

All of these systems use the same approach of mapping the object space into a virtual namespace where assignment of objects to hosts is more convenient because of the uniform spread of object mappings. In this dissertation, we use Chord an example DHT, and TerraDir as an example hierarchical system. We expand on Chord in Section 2.1.1 and TerraDir in Section 2.1.2. CAN defines a d-dimensional Cartesian coordinate space on a d-torus. Assuming an evenly partitioned space among the N servers, the path length between any two servers is $O(dN^{1/d})$, with each server maintaining information about d neighbors. Given parameter b, Pastry maintains routing tables of size approximately $O(2^b \log_{2^b} N)$ and performs routing in $O(\log_{2^b} N)$ steps, assuming accurate routing tables and no recent server failures. Tapestry is very similar in spirit to Pastry, both being inspired by the routing scheme introduced by Plaxton et al. [65].

2.1.1 P2P Systems Built on DHTs: Chord

We use Chord to represent the class of hash-based systems, but there is a great deal of other work on systems based on the same idea (e.g. Pastry, Tapestry, CAN, Viceroy, etc.). These systems differ from Chord in important ways, but few of these differences should affect the applicability of our work.

Chord is essentially a fast lookup service based on the notion of *consistent hashing* [44], implemented via SHA-1. The names of data items exported by the peers are hashed to item keys; peer ID's are hashed to peer keys. With large numbers of peers, and 160-bit keys, the peers will map relatively evenly around an identifier circle whose locations are specified by indexes $0 \ldots 2^{160} - 1$. Figure 2.1.1 shows an example Chord

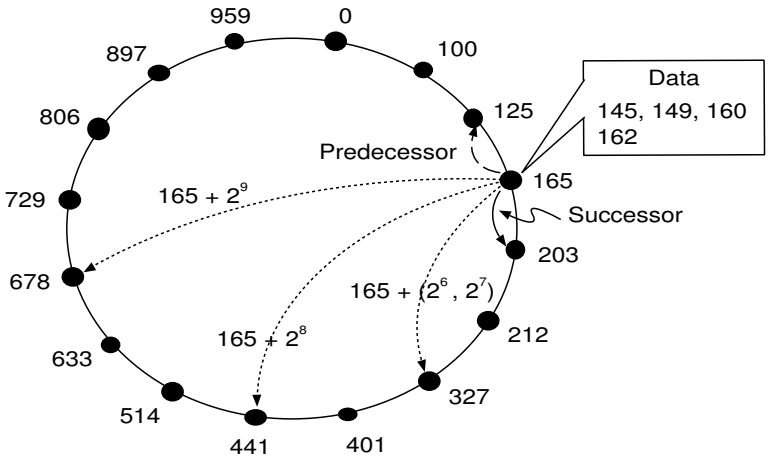


Figure 2.3: The figure shows an example Chord network. Chord is a DHT-based structured P2P system.

network. For the ease of exposition, we assume a network with identifiers ranging from 0-999. Key k's home is the first peer whose identifier is equal to or larger than k and is known as its *successor*. For example, the successor of object 145 is the node 165 and is hence stored at node 165. Each node maintains two pointers; a predecessor and a successor. The predecessor of a node n is the node p, such that p is the largest identifier smaller than n. Similarly, the successor is the node s whose identifier is greater than n. In Figure 2.1.1, 125 is the predecessor of 165 and 203 is the successor of 165. Note that the only bit of consistent state required by Chord is a peer's successor. If each peer reliably knows its successor, routing is guaranteed to complete successfully.

Using only successor pointers would, however, result in $O(n)$ query latency, so the successor pointers are enhanced with a routing table called the finger table. A finger table has at most m entries. The i^{th} entry at peer n contains the identity of the first peer that succeeds n by at least 2^{i-1} on the identifier circle. Thus, succeeding finger table

entries reach further and further around the circle and a peer can efficiently route a packet by forwarding each message to the finger table entry closest, but less than or equal to, the message destination's key. If finger table information is accurate, this approach will usually route packets successfully in at most $O(\log N)$ steps. Finger entries are shown in dashed lines in Figure 2.1.1.

Note that Chord peers both "export" and "serve" items. Exported items are hashed and inserted into the name space. Served items are those that are mapped to a peer by the hash mechanism. The hash-based approach helps load balance because even if all of the data items exported from one site are in high demand, they will generally be served by different machines. Similarly, routing load is distributed because paths to items exported by the same site are usually quite different. Just as importantly, the virtualization of the namespace provided by consistent hashing provides a clean, elegant abstraction of routing, with provable bounds for routing latency.

2.1.2 Hierarchical P2P Systems: TerraDir

The advantages of hash-based systems are clear, but there are also potential disadvantages [47]. First, hash-based virtualization destroys locality. By virtualizing keys, data items from a single site are not usually co-located, meaning that opportunities for enhancing browsing, prefetching, and efficient searching are lost. Second, hash-based virtualization discards useful application-specific information. The data used by many applications (file systems, auctions, resource discovery) is naturally described using hierarchies. A hierarchy exposes relationships between items near to each other in the

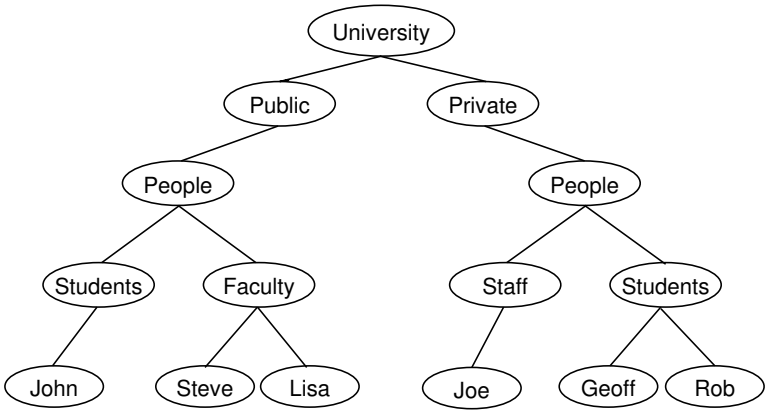


Figure 2.4: The figure shows an example TerraDir network. TerraDir is a hierarchical P2P system.

hierarchy; virtualization of the namespace discards this information.

We use TerraDir as a representative of hierarchical systems. TerraDir assumes that the input data is already organized hierarchically, and routes packets over the resulting tree structure. Unlike in hash-based schemes, an item's home in TerraDir is the server that exported it. Each item forms a node in the TerraDir namespace tree. A node in the tree structure is identified by its fully-qualified hierarchical name much like file names in UNIX-style file systems or hostnames in the DNS space. Each node has a set of neighbors that includes the parent and children of the node in the namespace. In this dissertation, we assume that the structure of the namespace is that of a tree with a special node as the root of the tree (node *University* in Figure 2.1.2). In general, TerraDir allows arbitrary graph-rooted topologies to be specified. Figure 2.1.2 shows an example TerraDir namespace. Node */university/public/people/students/john* represents the fully-qualified name for the item *john*.

Queries are answered by routing from the source to the greatest common prefix of the source and the destination, and then to the destination. For example, in Figure 2.1.2, assume that the node */university/private/people/staff/joe* initiates a query for */university/public/people/faculty*. This query is first routed "up" the tree to */university*, and then "down" to the destination. If the source is the ancestor (descendant) of the destination, the greatest common prefix is merely the source (destination). Note that this scheme generalizes to multiple items per server by ensuring that the local item "closest" to the destination is always picked when forwarding messages towards the destination. In practice, this basic scheme is enhanced by caching item-to-server mapping and through the use of digests containing item-to-server mapping.

2.2 Search over Peer-to-Peer Systems

The problem of searching over Peer-to-Peer systems started with searching for content over Gnutella and KaZaa file-sharing systems. In order to search for content, users submitted queries that were flooded in the system. Any result that matched the query terms was returned as valid results to users. The introduction of structured P2P systems, forced changes in the way content was searched. Structured P2P systems only supported the *lookup* operation. Given a particular identifier, they can efficiently and quickly identify the node storing the content. However, users were required to specify the exact name of the object they were interested in. There is no facility in structured systems to search for content based on attributes or keywords.

In order to provide a mechanism to search in structured P2P systems, Reynolds

and Vahdat [70] proposed the use of distributed inverted indexes. An inverted index is essentially a list of all documents that contain that keyword. Such indexes are maintained for all the keywords in the system. The node responsible for the keyword is identified by mapping the keyword to the identifier space; the node whose identifier is closet to the keyword's identifier is responsible for storing the index corresponding to the keyword. Given this setup, answering queries with single keywords is simple: map the keyword to its identifier and use the underlying lookup protocol to route the query to the node storing the index. In response all the entries in the index are returned as valid results.

Answering multi-keyword queries in this model however is tricky. In particular, in order to answer multi-keyword conjunctive queries, indexes of one or more keywords has to be transferred over the network and intersected. Indexes, however, can be very big and transferring them may consume a lot of network bandwidth. Repeating this process over and over again for each instance of the query makes it prohibitively expensive. To address this problem, Reynolds and Vahdat suggested the use of Bloom filters [9]. Consider intersecting the index for `foo` and `bar`. In their approach, Reynolds and Vahdat construct a Bloom filter for index `foo`, and transfer the filter to the node storing `bar`. The node storing `bar` checks for the existence of all the entries in the index with the filter and identifies all those entries that yield a positive result. These results are then returned as results to the query. Note that a similar strategy was suggested by Mullin [61] for computing semi-joins in distributed databases.

Since then, there have been a number of efforts to support search over P2P systems. Of these, *eSearch* [82] is probably the most similar in spirit to what is proposed in this dissertation and attempts to address some of the same problems. eSearch [82] uses

techniques from traditional information retrieval to extract keywords of each document. eSearch also maintains indexes of keywords. However, in eSearch, each index entry not only contains the document identifier but also all the other keywords associated with the document. This approach retains the benefits of the approach suggested by Reynolds and Vahdat, but can also answer multi-keyword queries efficiently. Given a multi-keyword query, a particular keyword is picked at random and the query is routed to the node storing the index of that keyword. While evaluating the query, the node scans all the entries in the index and identifies all documents that also contain the other keywords in the query. Since all the keywords of a document are stored locally, these documents can be identified without incurring any network cost. eSearch also utilizes a partitioning scheme that maps an index to a range rather than a point on the Chord DHT. The authors shows that this scheme can balance the amount of data stored at each node within a certain bound. eSearch, however, has its share of downsides. Storing the extra information consumes much more disk space compared to indexes with only document entries. eSearch also suffers greatly in the presence of failures. The cost associated with storing and maintaining indexes is very high because of the size of the indexes.

Gnawali [30], in his thesis, proposes storing indexes that are intersections of two keywords. The design completely does away with indexes for single keywords. Queries that contain single keywords, are answered by combining the results of all the indexes that contain the keyword. The design is motivated by the fact that a large fraction of the queries contain two keywords. Further, the size of an index for two keywords is expected to be is considerably smaller than the index for single keywords. This can reduce the challenges associated with storing large indexes. Our approach, on the other hand, is to

store the results for two keyword queries as they are generated rather than proactively creating all possible combinations. Further, we store both single keyword indexes, and caches of query results with two or more keywords.

PlanetP [20] is a publish-subscribe service for P2P communities, supporting ranking search. The content-based search scheme in PlanetP uses VSM [76] for ranking search results. Each node in PlanetP stores the document vector locally. PlanetP distinguishes local indexes and a global index to describe all peers and their shared information. The global index is stored in the form of digests of the content stored in all other nodes. This global information is spread using gossip. When evaluating queries, nodes first rank the *peers* using the digest and forward their queries to them. Results are evaluated locally in these peers using the cosine similarity metric and returned to the node that started the query. Since the system relies on gossip to propagate information needed to evaluate queries, the system does not scale to more than a few thousand peers.

Odissea [81], is a two-tier architecture designed to act as a distributed infrastructure for performing web-searches. The upper tier consists of external clients that either update the indexes stored in the system or pass queries to the system. The lower tier consists of the Odissea search system. Peers in the lower tier are arranged in a Chord-like ring and store global indexes that get mapped to them. Odissea uses distributed query execution algorithms based on ideas by Fagin and others [24] that asymptotically reduce communication. Hence only a small fraction of the inverted lists is usually transmitted.

Loo et al. [54] argue for a hybrid approach to design a search system. They contend that Gnutella and KaZaa are extremely efficient in answering queries for content that is popular. However, they perform poorly while answering queries for rare objects, some-

times not returning results even when instance of the object exist in the system. Global indexes, on the other hand, can return all possible results to all queries, irrespective of the popularity of the object. Back-of-the-envelope calculations, however, show that global index based designs are bandwidth-expensive. Loo et al. hence propose a hybrid solution consisting of regular clients connected to SuperPeers. SuperPeers, in turn, are organized in a DHT and make use of global indexes to locate rare documents and Gnutella-style flooding for popular documents.

Finally, Li et al. [52] question the feasibility of Web indexing and searching in Peer-to-Peer systems. They conclude that techniques, to reduce the amount of data transferred over the network are essential for the feasibility of search of P2P systems. Bulk of our work aims at reducing the amount of data transferred at the cost of relatively little disk space. Hence, we believe that our techniques can be used to make P2P indexing and searching practical.

2.2.1 Ranking Search Results

Providing ranked searches has been an important area of research. Prior work related to ranked searches can broadly be classified into two categories: traditional centralized approaches, and strategies over structured P2P networks.

Classic Information Retrieval Centralized information retrieval and automatically ordering documents by relevance has long been area of much research. We discuss the Vector Space Method [76] in Section 3.2. Salton and Buckley [74], Dumais [23] and Frakes et al. [26] discuss different term weighting models. Salton et. al. [75] discuss

an extension to the Boolean model using VSM. They show that the extended model has better result quality than the conventional boolean scheme. Latent Semantic Indexing (LSI) [22] is an extension to VSM that attempts to eliminate the issues of synonyms and polysemy. LSI employs singular value decomposition (SVD) to reduce the matrix generated by VSM. LSI then uses this reduced matrix to answer queries. LSI has empirically been shown to be very useful, especially with small collections. While techniques to compute SVD (e.g., using gossip [49]) in a distributed setting are known, it is still an open question as to how the data should be organized for effective retrieval. PageRank [64] has been deployed, with much success, to rank web pages. There has also been work on implementing PageRank in a distributed setting. Wang and DeWitt [86], present a P2P system that implements ranking of web pages in a distributed and scalable fashion, by employing a distributed implementation of PageRank. PageRank, however, uses the hypertext link information to compute the rank of a web page and cannot be applied on an arbitrary document set because of the lack of hyper-links.

Distributed Search over P2P systems The idea of using Vector Space Methods has been applied previously in the context of P2P similarity search. pSearch [83] uses VSM and LSI to generate document and query vectors, and maps these vectors to a high-dimension P2P system. The authors show that the query is mapped close to the documents relevant to the query and controlled flooding is employed to fetch relevant documents. The authors, however, do not compute the vectors from scratch. Instead, they use precomputed vectors and then aggregate information using "combining trees". While pSearch has the information to rank search results, the way data is organized makes it inefficient to return a globally ranked result set.

Bhattacharya et.al. [8] use similarity-preserving hashes (SPH) and the cosine similarity to compute similar documents over any DHT. Using Hamming distance as the metric, they show that SPH guarantees that the keys generated for similar documents are clustered. Given a query and a similarity threshold, they fetch all documents whose cosine similarity is greater than the threshold. Once again, in this system, the default organization of the data makes it inefficient to return globally ranked results. Further, the authors also assume that the document vectors are given to them.

PlanetP [20] is a content-based search scheme that uses VSM for ranking search results. Each node in PlanetP stores the document vector locally. Nodes also store digests of the content stored in all other nodes. PlanetP uses gossiping to spread this information. While evaluating queries, nodes rank the *peers* using the digest and forward their queries to them. Results are evaluated locally in these peers using the cosine similarity and returned to the node that started the query.

Fagin et al.'s Threshold Algorithm [25] computes the top-K data items in the following setting: given m sorted lists, each associating an item with an attribute value, the value of an item x is defined by applying a monotonic function to the attribute values that x has in each of the lists. The threshold algorithm makes a minimal number of accesses to the given lists. Their work applies also in a distributed systems where the lists are inverted indexes stored at different nodes. Cao and Wang [13] and Michel et al. [58] explore this technique and provide optimizations to reduce the amount of data transferred while aggregating results. Unlike our approach where we compute the weight assigned to each term in the document and query, these authors assume that a score is already available. We believe that the two techniques are complementary.

Recall that our ranking scheme uses random sampling to estimate global aggregates. An alternative to random sampling is to use gossip to aggregate the global information. Bawa et al. [5] show techniques to estimate the number of nodes and other aggregates in unstructured systems. Similar approaches could be applied to estimate total number of documents in the system. Note, however, that gossiping would be expensive when used to compute the number of documents with the individual terms.

2.2.2 Query Evaluation and Caching

We use result caching as a viable strategy to reduce the amount of data trasferred during query evaluation. Other researchers have also looked at this problem. The existing work can be broadly catagorized into work on reducing query evaluation costs in P2P systems, and work in the area of using cached results to answer queries.

Optimizing query evaluation in P2P systems Reynolds and Vahdat [70] investigate the use Bloom filters to efficiently compute approximate intersections for multi-attribute queries. While Bloom filters reduce the amount of data transferred over the network, they alone are not sufficient. With Bloom filters, all the entries of the index would have to be scanned, thereby incurring a high processing cost per query. We believe that result caching and the use of Bloom filters are orthogonal techniques, and could be used together. For example, Bloom filters can be used to compute the intersection of indices after the set of useful result caches have been identified.

In eSearch [82], Tang and Dwarakadas extract the keywords of each document using vector space models [76] and index the full text in each of the indexes corresponding

to the keywords. The advantage of this approach is that the network traffic to compute intersections of indexes is eliminated, albeit at the cost of more storage space at each node. Finally, note that eSearch is not affected by locality in the query stream, and requires only its aggregate indexes to answer queries.

Caching results and materialized views The idea of using cached results and materialized views to enable faster query processing has beens studied extensively in the database literature (e.g., Keller[48], Gupta[37], Zhuge[90]) and is also related to the problem of answering queries using views (e.g., Halevy[39], Ullman[85]). However, there is a significant and important difference between our work and these traditional systems. Traditional systems assume that the views are located at a central server and that they have complete knowledge of the existing views. In our work, the views are scattered over the P2P network and we attempt to locate the views that are useful for answering a query.

2.2.3 Load Balancing in Peer-to-Peer Systems

The problem of load balancing in Peer-to-Peer systems has gained much attention off-late. Rao et al. [67], Byers et al. [12], Godfrey et al. [32], Ruhl and Karger [45] and Tang et al. [82] propose different load balancing techniques. The primary focus of their work, however, is to balance the namespace distribution of each peer. These solutions typically make use of multiple "virtual servers" running at each physical node. These virtual servers function like regular nodes in the network, thereby increasing the number of participating nodes. This, in turn, reduces the imbalance in namespace assignments. While these techniques guarantee that each virtual server is responsible to similar-sized

regions, they do not guarantee balanced load in terms of the number of queries answered. On the other hand, we focus on the problem of load balancing due to answering queries.

All the hash-based systems perform well when there is uniformity in query distribution. However studies [3, 10] show that both spatial and temporal reference locality are present in requests submitted at web servers or proxies, and that such requests follow a Zipf-like distribution. Distributed caching protocols [44] have been motivated by the need to balance the load and relieve hot-spots on the World-Wide-Web. Similar Zipf-like patterns were found in traces collected from Gnutella [31], one of the most widely deployed peer-to-peer systems. Caching the results of popular Gnutella queries for a short period of time proves to be effective in this case [79]. The approach we suggest (in Chapter 5) makes use of path propagation, and is a generalization of this caching scheme. Stavrou et al. [4] propose the use of a Peer-to-Peer overlay to handle hotspots in the Internet. In contrast, we need to balance load and handle hotspots in the Peer-to-Peer network.

Recent work [56, 19] considers static replication in combination with a variant of Gnutella searching using k random walkers. The authors show that replicating objects proportionally to their popularity achieves optimal load balance, while replicating them proportionally to the square-root of their popularity minimizes the average search latency. Freenet [18] replicates objects both on insertion and retrieval on the path from the initiator to the target mainly for anonymity and availability purposes. It is not clear how a system like Freenet would react to query locality and hot-spots.

Bhattacharya et al. [8] propose an adaptive replication scheme to balance query load in DHTs. The scheme is based on randomized binary searches. Objects are replicated using different hash functions such that the replica tree is a complete binary tree. Peers

know the hash function needed to identify each replica in this complete binary tree. They start searching for the existence of a replica by picking one of the possible nodes in the tree randomly and use binary search to identify an existing replica. Since the initial position of search is randomized, it is guaranteed that replicas receive equal load. While the authors guarantee analytical bounds on the load imbalance present after employing this scheme, it is not clear how quickly hotspots are dissipated using this scheme.

A great deal of work addresses data replication in the context of distributed database systems. Adaptive replication algorithms change the replication scheme of an object to reflect the read-write patterns and are shown to eventually converge towards the optimal scheme [88]. Concurrency control mechanisms need to be specified for data replication to guarantee replica consistency. A recent analysis [77] of two popular peer-to-peer file sharing systems concludes that the most distinguishing feature of these systems is their heterogeneity. We believe that the adaptive nature of our replication model would make it a first-class candidate in exploiting system heterogeneity.

2.2.4 Data partitioning

The problem of data partitioning has been studied extensively, especially in the area of distributed databases. Based on the way the data is distributed, the partitioning schemes are classified under horizontal or vertical partitioning schemes.

In horizontal partitioning [15], different tuples of a relation are stored at different sites. All the attributes of the relation for that tuple are stored at the same location. All horizontal partitioning methods guarantee that the data can be reconstructed completely.

For efficiency, the data stored at two different locations is disjoint. Some of the common horizontal partitioning strategies include:

- *Round-robin partitioning* Round-robin partitioning works by assigning the tuples to a particular site in a round-robin fashion. The first tuple goes to the first partition, the second to the second and so on.

- *Hash Partitioning* In hash based partitioning, a hash of a key in the tuple is computed and the tuple is shipped to the location that is the result of the hash function.

- *Range Partitioning* In range partitioning, ranges are chosen on the values of a key. All entries with values within a range are stored at the same partition.

- *Predicate-based Partitioning* Here the relation is partitioned using set of predicates and the locations are allocated partitions using metrics like query response time, network traffic and network throughput.

An important point to note is that DHTs, in some sense, use a combination of range partitioning and hash partitioning. Each node is responsible for a range of identifiers and stores all data items mapping to that node. Items are assigned to nodes using hash-based partitioning.

There has also been very recent work on data over a range of nodes in P2P systems. Ganesan et al. [27], present an online algorithm to distribute range-partitioned data in a manner such that the ratio of the smallest partition to that of the largest partition is guaranteed to be bounded. The main aim of the work is to balance existing partitions and prevent skew in range partitioned data. The algorithm proceeds in two steps: first

they try to balance the size of neighboring partitions. If that is not enough, they use a re-adjustment step where the location with the smallest size moves its data to its neighbor and accepts half the data from the location with largest data size. The authors show that the protocol can provide good guarantees for amortized costs when there are a lot on insertions and deletions. Despite the fact that the main aim of the work is different, the idea could be applied to P2P systems to split the indexes. However, the protocol has two main issues when it comes to P2P systems: (a) The node join protocol in P2P systems must be modified, and (b) Information about the load of the neighbors and the most-loaded and least-loaded node is required.

Tang et.al.[82] present a decentralized approximation of the previous algorithm in the context of DHTs. As usual, they hash the keyword and map it into the P2P namespace. Then, they randomize the 20 least-significant bits, thereby mapping different entries of the same index to 2^{20} locations. However, this alone is not sufficient. One can easily see that the expected number of nodes in this range is less than one. This implies that, with high probability, all the entries will actually be stored by the same node. The authors address this problem by modifying the join protocol. When joining the system, nodes identify overloaded regions and join in these regions. This, however, means that the distribution of nodes is not uniformly random. Hence, it is not clear if the guarantees given by the look-up protocols are valid. Finally, for perfect load balance, this procedure requires that documents come into the system before nodes. Since this is not possible, nodes in eSearch periodically leave the system and re-join as a different node using the modified join procedure.

Weatherspoon and Kubiatowicz [87], and Rodrigues and Liskov [71] independently

studied the trade-offs between erasure codes (such as Reed-Solomon [69] codes) and static replication. Weatherspoon and Kubiatowicz compare erasure codes and replication in terms of the Mean Time To Failure (MTTF) and Disk space utilization and conclude that erasure codes are superior. Rodrigues and Liskov study the effect of failure rate on the two approaches. They also look into the overhead of maintaining each of them under different failure rates. They conclude that replication is useful when the failure rates are low, but when the environment is very dynamic, then erasure codes fare better than static replication.

Chapter 3

Ranking search results

The current approach to support search over structured systems, such as those based on distributed hash tables (e.g., Chord [80], Pastry [72], etc.), is to make use of inverted indexes of keywords. Recall that an inverted index of a keyword is essentially a list of all the objects that contain that keyword. Inverted indexes are stored using the same mechanisms employed to store other objects in the system. While there exist search mechanisms (e.g., [70, 82, 30, 53]) that use these inverted indexes to answer queries, these mechanisms typically support boolean queries (queries where keywords are connected by conjunctions and/or disjunctions). The downside to supporting boolean queries is that queries containing popular keywords return unmanageably many, unordered results. This issue stems from the fact that the boolean model does not have the information necessary to rank retrieved items.

Ideally, results should be *ranked*, and only relevant results returned to the user. Users are primarily interested in the most relevant results rather than an unordered set of all possible results. Further, returning fewer results reduces the network bandwidth consumed. This is an important property from the system's perspective because it reduces the load on the network and helps scale the system. While the advantages of ranking search results are well known, ranking results in a distributed manner is difficult. This is because ranking is global: *all* documents relevant to a query have to be compared with

respect to each other. In a large system, the lack of a central location to aggregate global knowledge makes the problem of ranking search results challenging.

3.1 Overview

In this chapter we describe our algorithm for distributed ranking of search results. Our approach, which applies to both structured and unstructured P2P networks, uses an approximation technique based on uniform random sampling to extend the Vector Space Model (VSM) [76], a classic information retrieval algorithm originally due to Gerard Salton et al. In VSM, documents and queries are mapped to vectors in a high-dimensional space (the dimension is usually the number of unique terms in the system). The intuition behind the approach is that vectors of similar items will lie close to each other in this space. Relevant documents are identified by computing the angle between the query vector and all document vectors; smaller the angle, higher the rank assigned to the document.

In order to apply VSM to a distributed setting, we need to address three important challenges:

- Compute the vector representation in a distributed manner
- Store these computed vectors in the underlying distributed system
- Evaluate the queries in a distributed manner

We address each of these challenges using simple and well-known techniques. We use random sampling to estimate term weights which form the basis for the vector representations of documents and queries. We use the existing distributed index infrastructure and

extend the indexes to also store the vector representation. Finally, we show queries can be evaluated in this distributed setting to return just the top-K results. While the techniques used in our approach are not new by themselves, the novelty of our contribution lies in the fact that we show that these different techniques compose well, i.e., sampling enables distributed ranking with very little network overhead. Our main contributions are as follows:

- **Algorithm:** We present a set of techniques for efficiently approximating centralized VSM document ranking in a distributed manner. We present detailed algorithms for computing the SMART [11] implementation of VSM[1]; however, our scheme is general, and can easily be modified to compute other VSM-based global ranks. We analyze the overhead of our approach, and quantify exactly how our system scales with increasing number of documents, system size, non-uniform document to node mapping (as may be the case in unstructured networks) and types of queries (rare versus popular terms).

- **Evaluation:** We evaluate our ranking technique in a simulator using real document sets from the TREC collection [84]. The results show that our approach is robust to sampling errors, initial document distribution, and query location. Our results show that the distributed ranking scheme, with relatively little overhead (50 samples per query), can approximate centralized ranking ($> 85\%$ accuracy) in large systems (5000 peers) with many documents (100,000 documents, 400,000 unique terms).

[1]SMART is probably the most popular implementation of this type of global ranking, and is regularly used to rank queries on IR collections [40].

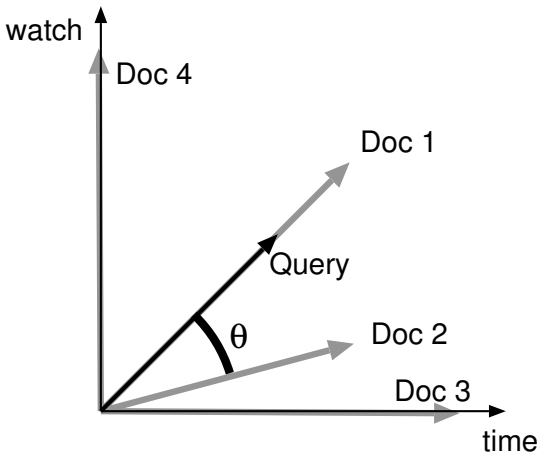
The rest of this chapter is organized as follows. We first present some background on ranking in classical information retrieval in Section 3.2. We then discuss our design for ranking results in Section 3.3 and analyze its properties. In Section 3.4, we present experimental results where we compare the performance of the distributed ranking scheme with a centralized scheme before concluding in Section 3.5.

3.2 Vector Space Model (VSM)

The Vector Space Model (VSM) is a classic information retrieval model due to Gerard Salton et al. [76]. VSM maps documents and queries to vectors in a T-dimensional term space, where T is the number of *unique* terms in the document collection. Each term i in the document d is assigned a weight $w_{i,d}$. The vector for a document d is defined as $\vec{d} = (w_{1,d}, w_{2,d}, \ldots, w_{T,d})$. A query is also represented as a vector $\vec{q} = (w_{1,q}, w_{2,q}, \ldots, w_{T,q})$, where q is treated as a document.

Vectors that are similar have a small angle between them. VSM uses this intuition to compute the set of relevant documents for a given query by looking at the angle between the vectors; relevant documents will differ from the query vector by a small angle while irrelevant documents will differ by a large angle.

Consider the example in Figure 3.2 which shows the vector representation of documents and query in a two-dimensional space. For simplicity, in this example we only consider the terms `time` and `watch`. In a real collection of documents, the number of unique terms would be much higher. `Doc 1`, `Doc 2` and the query contain both terms, therefore their vectors have positive components with respect to both axes. `Doc 3` only



Doc 1	"He checked the **time** on his **watch**."
Doc 2	"No **time**, no **time**, said the Mad Hatter while dipping his **watch** in his tea."
Doc 3	"**Time** flies like an arrow."
Doc 4	"Did you buy a new **watch**?"
Query	***"time, watch"***

Figure 3.1: How VSM works on a collection of four documents. For simplicity, in computing the vectors we only consider the two terms that appear in the query. The figure shows (only qualitatively) the five vectors and the angle θ, the cosine of which is the similarity measure between the second document and the query.

contains the term `time`; therefore its vector lies on the horizontal axis. Similarly, `Doc 4` lies on the `watch` axis. Since the word `time` appears twice in the second document, while `watch` appears only once, the vector for `Doc 2` is closer to the `time` axis. From Figure 3.2, it is clear the document most relevant to `Query` is `Doc 1`, followed by `Doc 2`.

Given two vectors X and Y, the angle θ between them can be computed using:

$$\cos\theta = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}}. \tag{3.1}$$

Equation 3.1, also known as the cosine similarity, has been used in traditional information retrieval to identify and rank relevant results. We will also use this measure of similarity for our ranking.

Cosine similarity has previously been used for similarity search in P2P systems by

Tang et.al. [83] and Bhattacharya et.al. [8]. Given a query vector, these systems use the cosine similarity measure to identify other similar documents. In theory, [83] and [8] can also use the cosine similarity to rank results; however, their default arrangement of data would make such ranking extremely inefficient.

3.2.1 Generating Vector Representation

The vector representation of a document is generated by computing the *weight* of each term in the document. The goal behind assigning weights is to identify terms that capture the semantics in the document and therefore help in discriminating between the documents. Effective term weighting techniques have been an area of much research [23, 74, 26], unfortunately with little consensus. However, most methods use three components in their weighting schemes:

1. The *term frequency* factor is a local weight component and is based on the observation that terms that occur frequently in a document are keywords that represent the document. Terms that occur frequently are assigned higher weight than less frequently occurring terms. Weighting schemes differ in how much importance they give to the actual frequency.

2. The *Inverse Document Frequency (IDF)* factor, which is a global factor, takes into account the importance of a term occurring infrequently in the document collection. In other words, it looks at how many other documents contain this word. In practice, words that occur in many documents are not useful for distinguishing between documents. For example, words like “the”, “because” etc. would get assigned high

term weights because they occur frequently in documents. Further, these words occur in almost all documents. Hence, these words are not useful in distinguishing documents.

3. The third component is a normalization factor. Terms that occur in longer documents get a higher weight, causing vectors of longer documents to have higher norms than shorter documents. Although cosine similarity is insensitive to the vector norms, other similarity measures are not. Normalizing the weights eliminates this advantage.

Table 3.2.1 lists a variety of different formulae proposed in the literature to compute local and global components. In our work, we use the weighting formula used in the SMART [11] system:

$$w_{t,d} = (\ln f_{t,d} + 1) \cdot \ln \left(\frac{D}{D_t} \right). \tag{3.2}$$

In this equation, $w_{t,d}$ is the weight of term t in document d, $f_{t,d}$ is the raw frequency of term t in document d, D is the total number of documents in the collection, and D_t is the number of documents in the collection that contain term t.

3.3 Distributed VSM Ranking

In this section, we present our distributed VSM ranking system for keyword-based queries. There are three main components needed for ranking: generating a vector representation for exported documents, storing the document vectors appropriately, and computing and ranking the query results. We first describe the assumptions we make and then discuss each of these components in detail.

Local Weight	$\chi\left(f_{t,d}\right)$	Salton et al.[74]
	$f_{t,d}$	Salton et al.[74]
	$\ln\left(f_{t,d}+1\right)$	Frakes et al.[26]
	$\ln f_{t,d}+1$	Dumais[23]
Global Weight	$\ln\left(\frac{\sum_{\forall d} f_{t,d}}{D_t}\right)$	Dumais[23]
	$\ln\left(\frac{D-D_t}{D_t}\right)$	Frakes et al.[26]
	$\ln\left(\frac{D}{D_t}\right)$	Salton et al.[74]
	$1+\sum_{j=1}^{D}\left(\frac{p_{t,j}\log p_{t,j}}{logD}\right)$ where, $p_{t,j}=f_{t,j}/\sum_{k=1}^{D} f_{t,k}$	Dumais[23]

Table 3.1: Formulae to compute local and global components of term weights. $f_{t,d}$ is the frequency of term t in document d. D is the total number of documents in the system. D_t is the number of documents with term t.

3.3.1 Assumptions

Existing P2P systems provide a substrate over which many distributed applications can be built. An important functionality provided by these systems is the *lookup* service: given an identifier (usually a bit string), these systems can quickly and efficiently locate the node (or the set of nodes) "responsible" for the identifier. Many existing systems use a distributed hash table (DHT) to arrange the participating peers in a strict topology. Each object is mapped to an identifier. Each peer is responsible for a range of identifiers and is responsible for all objects who's identifiers fall in its range. Such systems are classified as structured P2P systems and include popular implementations such as Chord [80] and Pastry [72]. There also exist other systems that do not mandate peers to connect in a strict topology. These systems are classified as unstructured systems. While typical implementations do not support lookup, there exist a few such as LMS [60] and Yappers [28] that provide the lookup functionality.

Our ranking algorithm is designed for existing P2P systems; both structured and unstructured. We assume that the underlying P2P system provides the *lookup* mechanisms necessary to route messages and identify nodes responsible for indexes. Our algorithm constructs an inverted index for each keyword and these indexes are distributed over participating nodes. The semantics of underlying systems dictate how the index is stored; with structured P2P systems indexes are stored at a single location, while an index may be partitioned over many locations in unstructured systems. Nodes may be controlled by autonomous entities, and can freely join or leave the system at any time. Each node is cooperative, in that, it stores the indexes that get mapped to it and processes any queries that are sent for the indexes mapped to it. Each node "exports" a set of documents when it joins the system. A set of keywords (by default, all words in the document) are associated with the document. The process of exporting a document consists of adding an entry for the document in the index associated with each keyword. When querying, users submit queries containing keywords and may specify that only the highest ranked K results be returned. The system then computes these K results in a distributed manner and returns the results to the user.

3.3.2 Generating Document Vectors

Recall Equation (3.2), which is used to compute the weight of each term t in a document. The equation has two components: a local component, $\ln f_{t,d} + 1$, which captures the relative importance of the term in the given document, and a global component, $\ln(D/D_t)$, which accounts for how infrequently the term is used across all documents.

The local component, which is the frequency of the term in the document, is obtained locally as in procedure COMPUTE-LOCAL-WEIGHT of Figure 3.2.

The global component is stated in terms of the number of documents D in the system, and the number of documents D_t that have the term t. We use random sampling to estimate these measures. In what follows, we describe our approach and analyze its properties. The pseudo-code for this step is in the procedure ESTIMATE-GLOBAL-WEIGHT in Figure 3.2.

Let N be the number of nodes in the system, and D and D_t be as above. Initially, we assume that a document is stored at exactly one node in the system. We will remove this assumption later.

We choose k nodes uniformly at random and independently. We then compute the total number $\tilde{D}$ of documents and $\tilde{D}_t$ of documents with term t at the sampled nodes. Choosing a random node can be done either with random walks, in unstructured systems [60], or routing to a random point in the namespace, in structured systems [50]. For simplicity, we accept that the same node may be sampled more than once. It is easy to see that:

$$E[\tilde{D}] = k\frac{D}{N} \quad \text{and} \quad E[\tilde{D}_t] = k\frac{D_t}{N},$$

where E indicates expectation of a random variable. The intuition is that, if we take enough samples, $\tilde{D}$ and $\tilde{D}_t$ are reasonably close to their expected value. If that is the case, then we can estimate D/D_t as:

$$\frac{D}{D_t} \approx \frac{\tilde{D}}{\tilde{D}_t}.$$

We now derive a sufficient condition for this approximation to hold. We introduce

```
EXPORT-DOCUMENT(n, d)
  for each t in d
    w_t ← COMPUTE-LOCAL-WEIGHT(d, t)
    w_t ← w_t · ESTIMATE-GLOBAL-WEIGHT(t, k)
  NORMALIZE-VECTOR(w⃗)
  for each t in d
    key ← generate-key(t)
    n' ← lookup(key)
    insert(n', key, d.id, w⃗)
  key ← generate-key(d.id)
  n' ← lookup(key)
  insert(n', key, d.id, d)

EVALUATE-QUERY(n, q, size)
  for each t in q
    w_t ← COMPUTE-LOCAL-WEIGHT(q, t)
    w_t ← w_t · ESTIMATE-GLOBAL-WEIGHT(t, k)
  NORMALIZE(w⃗)
  for each t in q
    key ← generate-key(t)
    n' ← lookup(key)
    R ← R ∪ GET-RESULTS(n', key, w⃗, size)
    R ← TOP-K(R, size)
  return R

COMPUTE-LOCAL-WEIGHT(d, t)
  if f_{t,d} = 0
    then return 0
    else return ln(f_{t,d}) + 1

ESTIMATE-GLOBAL-WEIGHT(t, k)
  d ← 0
  d_t ← 0
  for i ← 1 to k
    j ← GET-RANDOM-NODE()
    d ← d + GET-TOTAL-DOCS(j)
    d_t ← d_t + GET-DOC-COUNT(j, t)
  return ln(d / d_t)

NORMALIZE-VECTOR(w⃗)
  for i ← 1 to T
    w_i = w_i / sqrt(Σ_{i=1}^{T} w_i × w_i)

GET-RESULTS(n, key, q⃗, size)
  for each d⃗ in Index_key
    w_d ← Σ_{∀t} d_t q_t
    R ← R ∪ (d, w_d)
  R ← TOP-K(R, size)
  return R
```

Figure 3.2: Pseudo-code of our ranking algorithm. Procedure calls shown in italics are procedures provided by the underlying P2P system.

two new quantities: let M be the maximum number of documents at any node and M_t the maximum number of documents at any node with term t. We call the estimate $\tilde{D}$ (resp. $\tilde{D}_t$) "good", if it is within a factor of $(1 \pm \delta)$ of its expected value and we allow for the estimate to be "bad" with a small probability (ϵ).

Theorem 3.1 *Let D, N, k, M be as above. For any $0 < \delta \leq 1$ and $\epsilon > 0$, if*

$$k \geq \frac{3}{\delta^2} \frac{M}{D/N} \ln(2/\epsilon). \tag{3.3}$$

then the random variable $\tilde{D}$ (as defined above) is very close to its mean, except with probability at most ϵ. Specifically:

$$\Pr[(1-\delta)\frac{kD}{N} \leq \tilde{D} \leq (1+\delta)\frac{kD}{N}] > 1 - \epsilon. \tag{3.4}$$

Proof The proof is an application of the Chernoff bound. For $i = 1, \ldots, k$, let Y_i be the random variable representing the number of documents found during the i-th sample. Note that $\tilde{D} = \sum_{i=1}^{k} Y_i$. In order to apply the Chernoff bound, we need random variables in the interval $[0, 1]$. Let $X_i = Y_i/M$ and let $X = \sum_{i=1}^{k} X_i = \tilde{D}/M$. Define:

$$\mu = E[X] = \frac{kD}{MN}.$$

Since X_i are in $[0, 1]$ and are independent, we can use the Chernoff bound, which tells us that for any $0 < \delta \leq 1$.

$$\Pr[|X - E[X]| > \delta E[X]] \leq 2e^{-\frac{\mu\delta^2}{3}},$$

which can be rewritten as:

$$\Pr[(1-\delta)\frac{kD}{N} \leq \tilde{D} \leq (1+\delta)\frac{kD}{N}] > 1 - 2e^{-\frac{\mu\delta^2}{3}}.$$

We now impose the constraint that the probability above is at least $1 - \epsilon$:

$$2e^{-\frac{\mu\delta^2}{3}} \leq \epsilon,$$

from which we derive the bound on k:

$$k \geq \frac{3}{\delta^2}\frac{M}{D/N}\ln(2/\epsilon). \quad \blacksquare$$

Note that if we replace D, M, $\tilde{D}$ with D_t, M_t, $\tilde{D}_t$, the theorem also yields that, if:

$$k \geq \frac{3}{\delta^2}\frac{M_t}{D_t/N}\ln(2/\epsilon). \tag{3.5}$$

then the random variable $\tilde{D}_t$ is also a good estimate.

The following observations follow from Theorem 3.1:

- Theorem 3.1 tells us how many samples we need for $\tilde{D}/\tilde{D}_t$ to be a good estimate of the ratio D/D_t. Analogous to the classical problem of sampling a population, the number of samples needed does not depend on N directly, but only through the quantities D/N and D_t/N and, less importantly, on M and M_t.

  This means that as the system size grows, we do *not* need more samples as long as the number of exported documents (with term t) also increases. More samples are needed only if the system size grows without a corresponding increase in the number of documents. However, replication can be used to handle this case: our algorithm works without modification in the case where the same document may be stored at multiple (r) nodes, as long as r is the same for all of the documents. The analysis above still holds, as long as D and D_t are replaced by ed and rD_t respectively. Note that sampling performance actually *improves* with replication. Extending the algorithm to general replication is an open problem.

- The global component $\ln(D/D_t)$ that we are trying to estimate is relatively insensitive to estimation errors on D and D_t, because the algorithm uses the logarithm of the ratio.

- If we restrict our attention to the case where the number of documents D is much larger than the system size N and we focus on documents and queries consisting of popular terms ($D_t = \Omega(N)$), then our algorithm provides performance with ideal scaling behavior. Sampling a constant number of nodes gives us provably accurate results, *regardless of the system size*.

- In practice, documents and queries will contain rare (i.e., not popular) terms, for which $\ln(D/D_t)$ may be estimated incorrectly. However, we argue that such estimation error is both unimportant and inevitable. The estimation is relatively unimportant because if the query contains rare terms, then the entire set of results is relatively small, and ranking a small set is not as important. Finally, note that in centralized systems, ranking algorithms do not consider such rare terms (e.g. terms that appear only in one document) in ranking documents since they dominate the document weight. However, in a distributed setting, it is not possible to discern whether a term is truly rare since this requires global knowledge. In general, sampling is a poor approach for estimating rare properties; we describe other possible rank computation approaches that handle rare terms better in Section 3.3.6.

- The number of samples is proportional to the ratios $\frac{M}{D/N}$ and $\frac{M_t}{D_t/N}$ between the maximum and the average number of documents stored at a node (respectively, of all documents and of the documents with term t). This means that, as the distri-

bution of documents in the system becomes more imbalanced, more samples are needed to obtain accurate results.

Special case: uniform distribution

We next restrict our attention to the special case in which the underlying storage system randomly distributes documents to the nodes, uniformly and independently. Such distribution approximately models the behavior of a DHT.[2] In this special case, a stronger version of Theorem 3.1 holds.

Theorem 3.2 *Let D, N, k be as above and assume each document is independently and randomly stored at one node. For any $\epsilon > 0$, if*

$$k \geq \frac{3}{\delta^2}\frac{1}{D/N}\ln(2/\epsilon). \tag{3.6}$$

then the random variable $\tilde{D}$ (as defined above) is very close to its mean, except with probability at most ϵ. Specifically:

$$\Pr[(1-\delta)\frac{kD}{N} \leq \tilde{D} \leq (1+\delta)\frac{kD}{N}] > 1-\epsilon. \tag{3.7}$$

Proof For every document d, define the random variable X_d as indicator of the event that the document d is randomly stored at one of the k sampled nodes. For simplicity assume that the k sampled nodes are distinct, so $E[X_d] = k/N$. Defining $X = \sum_d X_d$ and applying the Chernoff bound to X yields the result. The details are analogous to the proof of Theorem 3.1. ∎

[2]More precisely, in a DHT like Chord [80] or Pastry [72], documents are not exactly uniformly distributed to nodes, but the probability that a document is assigned to a specific node may be as high as $\Theta((\log N)/N)$ and as low as $\Theta(1/N^2)$. For simplicity, we ignore this detail.

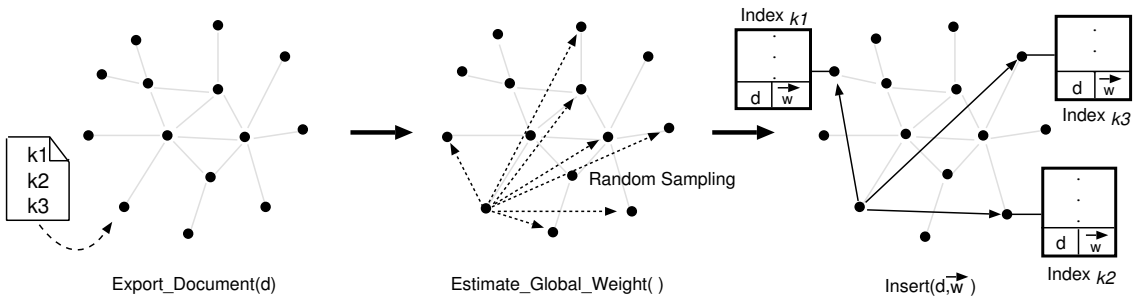


Figure 3.3: Various steps in exporting documents and their vector representation

Hence, for the uniform case, the number of samples does not need to be proportional to the maximum number M of documents at any node. Therefore, the cost of our sampling algorithm is significantly decreased.

3.3.3 Storing Document Vectors

Once the document vectors are computed, they need to be stored such that a query relevant to the document can quickly locate them. We store document vectors in distributed inverted indexes. As mentioned previously, an *inverted index* for a keyword t is a list of all the documents containing t. For each keyword t, our system stores the corresponding inverted index like any other object in the underlying P2P lookup system. This choice allows us to efficiently retrieve the vectors of all documents that share at least one term with the query.

Figure 3.3 shows the process of exporting a document. We first generate the corresponding vector by computing the term weights, which utilizes the procedure described in section 3.3.2. Next, using the API of the underlying storage system (*lookup()*), we identify the node storing the index associated with each term in the document and add

an entry to the index. Such entry includes a pointer to the document and the document vector.

The details of storing document vectors in inverted indexes depend on the underlying lookup protocol. In structured systems, given a keyword t, the index associated with t is stored at the node responsible for the key corresponding to t. The underlying protocol allows to efficiently locate such a node. Inverted indexes have previously been used for searching [82, 34, 70, 30, 53] in structured systems. We present the pseudo-code for exporting the document and storing the document vector in our system, over a structured network, in the procedure EXPORT-DOCUMENT in Figure 3.2.

In structured systems, inverted indexes enable us to group all documents related by a term at one location. Such a scheme is not directly applicable in unstructured networks. Instead we have to resort to approaches such as Yappers [28] or LMS [60]. As with structured systems, items are mapped to keys. However, due to the lack of structure in these systems, keys are not mapped to unique nodes in the network. Instead, depending on the facilities provided by the underlying network, each index can either be partitioned or replicated and stored at multiple nodes. We can use this facility to store document vectors in keywords indexes. The indexes can subsequently be located (and if necessary, reconstituted) using the underlying lookup mechanism.

3.3.4 Evaluating Query Results

In order to evaluate queries, they first need to be converted into their corresponding vector representations. We then compute the cosine similarity of each query with each

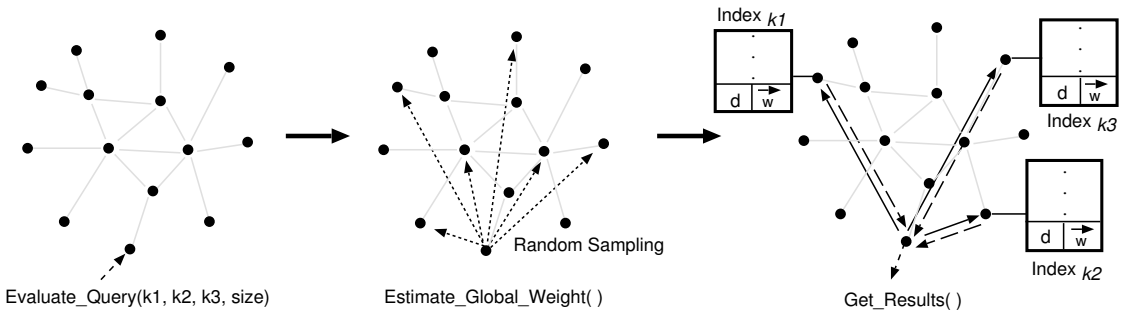


Figure 3.4: Process of computing query results

"relevant" document vector. Figure 3.4 shows these different steps involved in the query evaluation.

The first step in query evaluation is to generate the vector representation of each query. Query vectors are generated using the same techniques used to generate vector representation of documents. We first compute the local weight of the query terms, estimate the global weight by sampling, and then normalize the query vector. The next step is to locate the set of relevant documents. A key insight we use is that the set of results returned by VSM include only those documents that have one or more keywords in the query. This translates to the fact that in our system, we only need to evaluate the query at the index of each keyword in the query. Hence to evaluate the query, we forward the query to the node storing the index associated with each keyword in the query, using the lookup functionality provided by the underlying P2P system. These nodes run the VSM algorithm locally, and compute the cosine similarity between the query and each document in the index. This results in one ranked order of results relevant to the query. In order to obtain the final set of ranked results, we aggregate the results of computed at

each index, and compute the union of these results. The top-K results thus generated, sorted according to the decreasing order of cosine similarities, consists of the final set of results. We outline this entire process in the procedure EVALUATE-QUERY in Figure 3.2.

3.3.5 Practical Issues

Our approach for storing document vectors is susceptible to the practical problems found in systems using distributed inverted indexes. For example, the indexes for popular terms can be large; the system is susceptible to load imbalances, both in terms of query rates and the amount of data stored on nodes; node departures and failures can disrupt query evaluation, and so on. Solutions proposed in prior work for these problems [82, 34] directly apply in our setting. We do not consider these issues any further, because these problems are not unique to our approach and do not influence our algorithms or our evaluation. An implementation of our framework would need to employ the techniques, for example those proposed in eSearch [82] or by Gopalakrishnan et al. [34] to handle the indexes robustly and efficiently.

3.3.6 Possible Optimizations

The algorithms that we have proposed is generic and works over both structured and unstructured P2P networks. There, however, exist optimizations that can be applied on a case-by-case basis and can reduce overheads incurred in vector generation and storage.

Reducing sampling cost: Using random sampling to estimate the global component of the term weights is generic and works well in any setting: structured or otherwise.

However, the cost involved in sampling k nodes for each term is non-negligible. While we cannot eliminate the need for this estimate, we can use the information in the distributed indexes to reduce its cost.

Each inverted index contains a list of all documents that contain the keyword, and is hence an authoritative source for D_t. This eliminates the need to sample for D_t, thereby reducing the number of messages by a factor of k. We are, however, still left with estimating D, the total number of documents in the system. We discuss two approaches to estimating D. In the first, using gossip, nodes would periodically exchange estimates of system size.

The second approach is to continue using random sampling. Recall that $E[\tilde{D}]$ is dependent on N. Hence we need to estimate N in order to estimate D. In a recent result, King et al. [50] present an approach to estimate the number of nodes in a Chord-like system. Using this estimate on the number of nodes in the system, we can utilize random sampling to periodically estimate D. Techniques to estimate the number of nodes and other aggregates also exist for unstructured systems [5].

Both these approaches for reducing cost of sampling are promising, andwe plan to experiment with these techniques as part of our future work.

Reducing storage cost: Our approach to storing document vectors, as discussed thus far, assumes that each unique word in the document should be treated as a keyword, and be part of the document's descriptive vector. Hence a document entry is added to the indexes of all the unique keywords in the document. This is expensive both in the size of the vectors, and in the cost of propagating the vector to all of the corresponding indexes.

We employ a heuristic to reduce the number of vectors that are stored in the system.

We assume that there is a constant threshold w_{min}, that determines if the document entry is added to an index. The vector is not added to the index corresponding to the term t if the weight of t is below the threshold w_{min}. Note that the terms with weights below this threshold are still added to the vector. The intuition behind this approach is the following: a document will not appear in the top results of a query that has in common with the document only terms with low weight. Hence, discarding such entries in the index does not reduce the retrieval quality of the top results. This heuristic has previously been successfully used in eSearch [82].

Finally, we can further reduce the cost of storage at the price of increasing query cost. Instead of storing the full vector for a document entry in a keyword index, we can store just the normalized weight of that keyword. At query time, we can use the algorithm by Cao and Wang [13] or Michel et al. [58] to efficiently compute the top-K results. We, plan to experiment with them as part of future work.

3.4 Evaluation

In this section, we validate our distributed ranking system via simulation. We measure performance by comparing the quality of the query results returned by our algorithm with those of a centralized implementation of VSM. With these experiments, we demonstrate that our distributed system produces high quality results with little communication cost (of the order of 10 nodes visited per document insertion or query), for a reasonably large system (1000 nodes with 100K documents), and that such cost *is actually constant*, even as the system size increases. We also include results that show how to reduce the

Parameter	Values
# of runs for each experiment	50
# of Nodes	1000, 5000
# of Documents	100K, 1033
# of unique terms	418K
# of random samples	10, 20, 50
Mapping of doc. to nodes	Uniform, Zipf
Query term popularity	Q_{pop} - terms in $> 10K$ doc. Q_{5K} - terms in $\sim 5K$ doc. Q_{rare} - terms in < 200 doc.

Table 3.2: Experimental parameters and their values

storage required by the search protocol without impacting quality or increasing communication cost, and results that illustrate consistent ranks obtained by different users without prior coordination.

3.4.1 Experimental Setup

We use the TREC [84] Web-10G data-set for our documents. We used the first 100,000 documents in this dataset for our experiments. These 100K documents contain approximately 418K unique terms. Our default system size consists of 1000 nodes. We use two different distributions of documents over nodes: a uniform distribution to model the distribution of documents over a structured P2P system and a Zipf distribution to model a skewed distribution. Such a skewed distribution is possible in unstructured systems such as Yappers.

Since our large data set (100K documents) did not have queries associated with it, we generated queries of different lengths. Our default query set consists exclusively of terms that occur in approximately 5000 documents. We denote this query set as the Q_{5K}

query set in our experiments. The intuition behind picking these query terms is that they occur in a reasonable number of documents, and are hence popular. At the same time, they are useful enough to discriminate documents. We also use query sets that exclusively contain keywords that are either very popular (occur in more than 10K documents) or those that are very rare (occur in less than 200 documents). We denote these query sets as Q_{pop} and Q_{rare} respectively. In order to verify the performance against real queries, we use the smaller Medlars [11] medical dataset. This data set consists of 1033 documents and also has queries associated with it, which we use to evaluate our scheme with real queries. Each result presented (except for details from individual runs) is an average of 50 runs with different random seeds. We summarize the various parameters used in our experiments in Table 3.2.

The typical measures used to evaluate the quality of search results are *precision* (the fraction of valid results in the set of returned results) and *recall* (the ratio of valid results returned to the total number of valid results). However, since we did not have information about the set of valid results for the queries used in our experiments, we could not evaluate the performance of the system using these metrics. Instead, we used a modified set of metrics to evaluate the quality of distributed ranking. The metrics we used in our evaluation are:

1. **Coverage:** We define coverage as the number of top-K query results returned by the distributed scheme that are also present in the top-K results returned by a centralized VSM implementation for the same query. For example, if we're interested in the top 3 results, and the distributed scheme returns the documents (A, C, D)

while the centralized scheme returns (A, B, C), then the coverage for this query is 2.

2. **Fetch** We define fetch as the minimum number R' such that, when the user obtains the set of $\mathcal{R}'$ results as ranked by the distributed scheme, $\mathcal{R}'$ contains all the top-K results that a centralized implementation would return for the same query. In the previous example, if the fourth result returned by the distributed case had been B, then the fetch for $K = 3$ would be 4.

3. **Consistency** We define consistency as the similarity in the rank of results, for the same query, for different runs using different samples.

We do not explicitly present network overhead measures since the cost of the ranking (without counting the cost to access the indexes) is always equal to the number of nodes sampled. For the space optimization experiments, we note the original and reduced size of the indexes.

3.4.2 Coverage

In the first experiment, we measure the coverage of the distributed retrieval scheme. We show that by sampling only a few nodes even on a reasonably large system, our scheme produces results very close to a centralized implementation

In our base result, we use a 1000 node network. The documents are mapped uniformly to nodes. To compute the global weight of term t, we sample 10, 20 and 50 nodes in different runs of the experiment. The queries consist of keywords from the Q_{5K} query set, i.e. the keywords occur in approximately 5000 documents.

Table 3.3 shows the results of the experiment in which documents were mapped uniformly at random to nodes. As is clear from the table, distributed ranking scheme performs very similar to the centralized implementation. On a 1000 node network with documents distributed uniformly, the mean accuracy for the top-K results is close to 93% with 50 random samples. Even with 10 random samples, the results are only slightly worse at 85% accuracy.

With 5000 nodes, the retrieval quality is not as high as a network with 1000 nodes. With 20 random samples, the mean accuracy is 77% for top-K results. There is a 8% increase in mean accuracy when we increase the sampling level and visit 1% (50) of the nodes. This result is a direct consequence of Theorem 3.2. Here, the number of documents has remained the same, but the number of nodes has increased. Hence, higher number of nodes sampled leads to better estimates.

Table 3.3 also shows the retrieval quality for documents mapped to nodes using a Zipf distribution with parameter 0.80. With 1000 nodes and 50 samples, the retrieval quality is nearly equal to that of the uniformly distributed case. With 10 samples, however, the mean accuracy drops a few percentage points to between 82–83%. With 5000 nodes and 50 samples, we see similar trends. While the quality is not as good as it is with the uniformly distributed data, it does not differ by more than 2%. With 10 samples, the results worsen by about much as 7%. Hence, we believe our scheme can directly be applied over lookup protocols on unstructured networks without appreciable loss in quality.

Finally, Table 3.3 also shows coverage of the distributed scheme with the Medlar dataset. Since the dataset has only 1033 documents, we use a system size of 100 nodes

Network setup	Number of samples	Top-K results				
		10	20	30	40	50
1000 uniform	10	8.49 (1.08)	16.99 (1.20)	25.30 (1.55)	33.68 (2.07)	42.28 (2.01)
	20	8.90 (0.99)	17.81 (1.04)	26.44 (1.26)	35.23 (1.87)	44.30 (1.82)
	50	9.28 (0.82)	18.63 (0.82)	27.66 (1.04)	36.08 (1.45)	46.30 (1.46)
5000 uniform	10	6.78 (1.39)	13.58 (1.74)	20.43 (2.39)	27.35 (2.99)	34.59 (3.40)
	20	7.74 (1.29)	15.41 (1.46)	22.92 (1.96)	30.50 (2.47)	38.49 (2.58)
	50	8.52 (1.09)	16.96 (1.18)	25.20 (1.56)	33.59 (2.11)	42.34 (1.98)
1000 Zipf	10	8.27 (1.15)	16.52 (1.26)	24.66 (1.71)	32.82 (2.21)	41.20 (2.27)
	20	8.82 (0.99)	17.63 (1.06)	26.22 (1.35)	34.83 (1.93)	43.70 (1.88)
	50	9.26 (0.80)	18.54 (0.88)	27.52 (1.12)	36.71 (1.49)	46.12 (1.56)
5000 Zipf	10	6.09 (1.54)	12.29 (1.97)	18.58 (2.68)	25.01 (3.39)	31.67 (3.97)
	20	7.34 (1.31)	14.71 (1.62)	21.89 (2.10)	29.34 (2.64)	36.93 (2.90)
	50	8.41 (1.13)	16.73 (1.22)	24.92 (1.61)	33.22 (2.08)	41.71 (2.03)
Medlar	5	8.08 (1.26)	16.64 (1.90)	25.22 (2.47)	33.78 (2.85)	42.36 (3.03)

Table 3.3: Mean Coverage of the distributed ranking scheme. The standard deviation is given in brackets.

and sample 5 nodes. As shown in the table, the mean coverage is between 80–84%.

3.4.3 Fetch

Given the previous result, an obvious question to ask is how many results need to be fetched before all the top-K results from the centralized implementation are available (we called this measure Fetch). We ran an experiment with both 1000 and 5000 nodes with the documents uniformly distributed. We used the Q_{5K} query set for our evaluation. We plot the result in Figures 3.4.3 and 3.6. The x-axis is the top-K of results from the centralized implementation, while the y-axis represents the corresponding average fetch.

With a 1000 node network, we see that the fetch is quite small even if only ten nodes are sampled. For instance, sampling 10 nodes, we have to get a maximum of 13 results to match the top-10 results of the centralized case. With samples from 50 nodes,

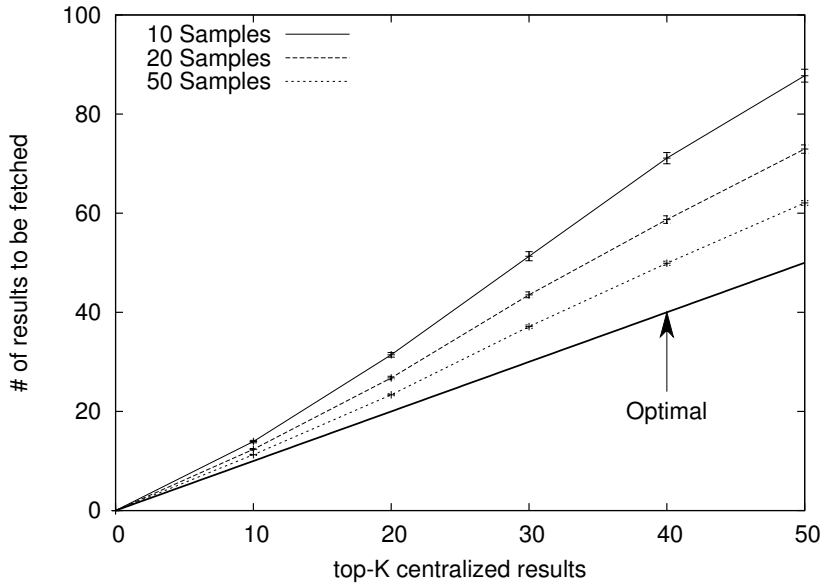


Figure 3.5: Mean fetch of the distributed ranking scheme with 1000 nodes. The error bars correspond to 95% confidence interval.

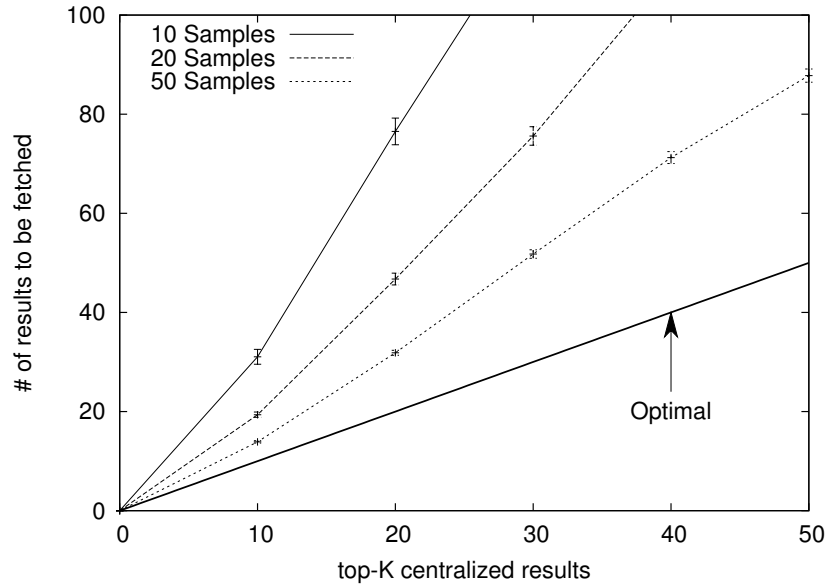


Figure 3.6: Mean fetch of the distributed ranking scheme with 5000 nodes. The error bars correspond to 95% confidence interval.

fetch is minimal even for less relevant documents: we only have to get 11 results to match the desired top-10 results and 63 to match the top-50 results from the centralized implementation.

As expected, with increasing network size, but for the same document set, the fetch increases. When we sample 1% of the 5000 nodes, we need 13 results to cover the top-10 and 88 to cover the top-50. With lower levels of sampling, however, we need to fetch a lot more results to cover the top-K. This behavior, again, is predicted by Theorem 3.2: when the number of nodes increases without a corresponding increase in the number of documents, the number of samples needed to guarantee a bound on the sampling error also increases.

Our experiments (which we merely summarize) yield similar results for Fetch when the document distribution is skewed. With a 1000 node network and 10 random samples, the fetch increases by 10% compared to the network where documents are mapped uniformly to nodes. In a 5000 node network, this increases by 35% compared to the uniform case. The results in both the network sizes with 50 random samples, however, are comparable to the uniform case.

3.4.4 Consistency

In our system, a query vector is generated using independent samples each time it is evaluated. This leads to different weights being assigned to the terms during different evaluations of the same query. This can increase the variance in ranking, and potentially lead to different results for different evaluations of the query. In this experiment, we show

that is not the case, and that the results are minimally affected by the different samples.

We use a network size of 1000 nodes with documents mapped uniformly to these nodes. We sample 20 random nodes while computing the query vector. We use Q_{5K} and Q_{rare} query sets for this experiment. We record the top-50 results for different runs and compare the results against each other and against the centralized implementation.

Figures 3.7 and 3.8 show the results obtained during five representative runs for three representative queries each. For each run, the figure includes a small box corresponding to a document ranked in the top-50 by centralized VSM if and only if this document was retrieved during this run. For example, in Figure 3.7, query 1, run 2 retrieved documents ranked $1 \ldots 25$, but did not return the document ranked 26 in its top 50 results. Also, note that the first 25 centrally ranked documents need not necessarily be ranked exactly in that order, but each of them were retrieved within the top-50.

There are two main observations to be drawn: first, the sampling does not adversely affect the consistency of the results, and different runs return essentially the same results. Further, note that these results show that the coverage of the top results is uniformly good, and the documents that are not retrieved are generally ranked towards the bottom of the top-50 by the centralized ranking. These observations also apply to the result with rare queries (Figure 3.8). In fact, a detailed analysis of our data shows that this trend holds in our other experiments as well.

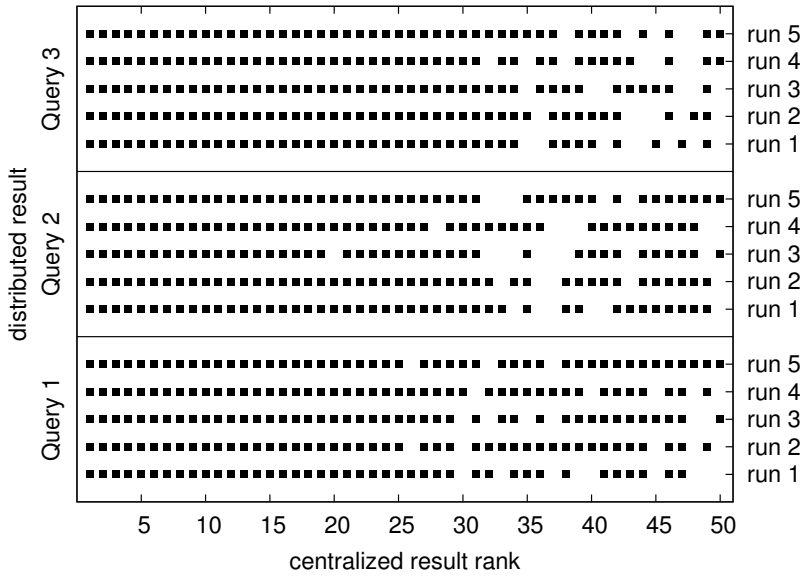


Figure 3.7: Consistency of top-50 results in distributed ranking for three different queries from Q_{5K} set

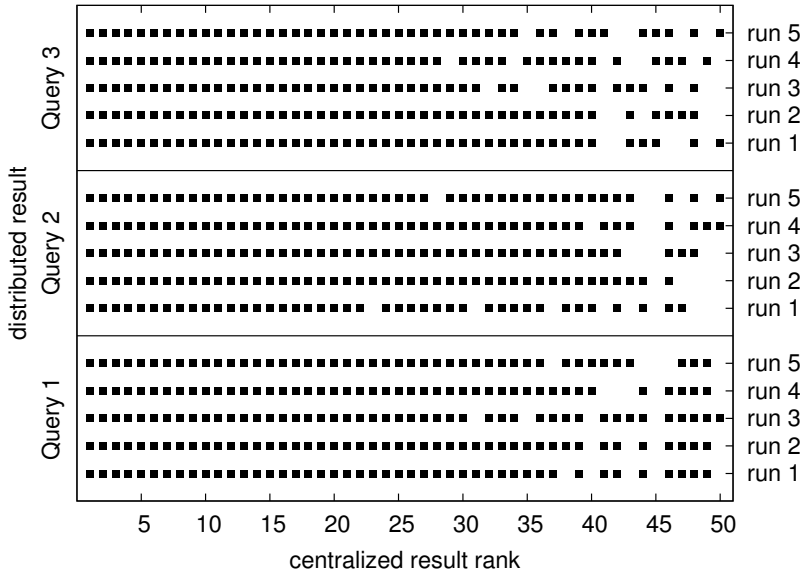


Figure 3.8: Consistency of top-50 results in distributed ranking for three different queries from Q_{rare} set

3.4.5 Scalability

In this experiment, we evaluate the scalability of our scheme with increasing system size. Theorem 3.2 states that the number of samples required is independent of the system size, under the condition that the size of the document set grows proportionally to the number of nodes. We demonstrate this fact by showing that coverage remains approximately constant as we increase the system size ten-fold (from 500 to 5000), while sampling the same number of nodes (20).

The number of documents in each experiment is 20 times the number of nodes in the system. For all the configurations, the terms used in queries occur in more than 10% of the total documents. For the 5000 node network, this corresponds to the Q_{pop} query set. In each case, we sample 20 random nodes to estimate the global weights.

Table 3.4 shows the mean and standard deviation in the coverage of our distribute scheme. As the table shows, the coverage of the distributed retrieval is very similar in most cases. In particular, there is very little difference in the results for 500-, 1000-, 2000- and 5000-node networks. This result confirms that our scheme depends almost entirely on the density of the number documents per node, and that it scales well as long as the density remains similar.

3.4.6 Reducing Storage Space

Recall our optimization (Section 3.3.6) to store document vectors only in the indexes of keywords whose weights are greater than a threshold w_{min}. In this experiment, we quantify the effect of this optimization. For this experiment, we used a network of

Network Setup	Top-K results				
	10	20	30	40	50
500 nodes	7.75 (1.26)	16.11 (1.65)	25.08 (1.85)	33.62 (2.06)	42.24 (2.57)
1000 nodes	7.99 (1.05)	16.33 (1.42)	24.58 (1.92)	32.98 (2.41)	41.59 (2.46)
2000 nodes	7.67 (1.31)	15.85 (1.85)	23.96 (2.05)	32.00 (2.61)	40.11 (3.04)
5000 nodes	6.95 (1.34)	15.21 (2.17)	22.99 (2.86)	30.66 (3.26)	38.85 (3.45)

Table 3.4: Mean (and std. dev.) coverage when the number of nodes and documents scale proportionally. All the results use 20 random samples.

1000 nodes with documents distributed uniformly at random over the nodes. We use all the three query sets and sample 20 nodes to estimate the weights. Note that we normalize the vectors; so the term weights range between 0.00 and 1.00. We present results for thresholds ranging from 0.00 to 0.30. We compare the results retrieved from the centralized implementation with $w_{min} = 0.00$.

The results of this experiment are tabulated in Table 3.5. Coverage of distributed ranking is not adversely affected when the threshold is set to 0.05 or 0.10. However, larger thresholds (say 0.20 and above) discard relevant entries, and consequently decrease rank quality appreciably.

In order to understand the reduction obtained by using the threshold, we recorded the total number of index entries in the system for each threshold. The total number of index entries in our system is 15.9M when the threshold is 0.0. Our experiments show a reduction of 55.5% entries when we use a threshold of 0.05. Increasing the threshold to 0.1 leads to an additional 30% reduction in index size. A threshold value of 0.1 seems to be a reasonable trade-off between search quality and decreased index size.

	Top-K results	Weight threshold (corresp. space reduction (%))				
		0.0 (0.0)	0.05 (55.5)	0.10 (85.0)	0.20 (97.2)	0.30 (99.3)
Q_{5K}	10	8.90	8.90	8.90	7.64	4.53
	20	17.81	17.84	17.81	14.58	6.68
	30	26.44	26.44	26.40	20.43	7.98
	40	35.23	35.30	35.18	26.03	8.67
	50	44.30	44.34	44.22	30.97	8.88
Q_{pop}	10	8.32	8.33	8.32	6.39	2.79
	20	17.45	17.49	17.44	12.80	5.19
	30	26.31	26.32	26.17	17.90	6.84
	40	34.87	34.94	34.64	22.10	8.39
	50	44.49	44.40	43.87	26.70	9.90
Q_{rare}	10	8.59	8.59	8.59	8.46	6.66
	20	17.22	17.22	17.22	15.66	8.84
	30	26.01	26.01	26.01	21.41	9.78
	40	35.31	35.31	36.19	25.66	9.94
	50	44.47	44.47	43.54	28.43	9.99

Table 3.5: The mean coverage of distributed ranking for different weight thresholds. The numbers in parenthesis show the reduction in the size of the indexes corresponding to the different thresholds.

3.4.7 Eliminating the Random Probes

The random probes used to create query (and document) vectors are used to establish accurate values of D and D_t. We ran an experiment to test the sensitivity of the coverage metric to the accuracy with which D is known. The results of this experiment are shown in Table 3.6. Assuming that D_t is known accurately, the system achieves nearly 75% coverage (up to the top 50 items) even when the assumed D value is 16 times larger than the real value.

Exact values of D_t can usually be retrieved directly from the indexes of the query or document terms, of which there are usually few. These results therefore suggest eliminat-

D-factor	Top-K results				
	10	20	30	40	50
0.50	8.74	17.68	25.68	34.26	42.89
0.75	9.58	19.21	28.32	37.68	47.16
1.00	10.00	20.00	30.00	40.00	50.00
1.25	9.63	19.26	28.63	38.21	47.89
1.50	9.32	18.63	28.16	37.00	46.53
2.00	9.05	17.95	27.00	35.58	44.63
4.00	8.37	16.47	25.05	33.00	41.37
8.00	7.68	15.37	23.58	31.05	39.05
16.00	7.26	14.74	22.42	29.21	37.21

Table 3.6: Mean coverage with accurate D_t and inaccurate D.

ing the random probes in favor of a new vector creation algorithm where D is maintained through a very low-priority background process (e.g., gossiping), and D_t's are retrieved directly from term indexes.

This technique relies on the fact that D_t can be accurately computed. However, when thresholds are used, low-weight document terms might not be inserted into system indexes because they fall below threshold, resulting in an inaccurate D_t. Hence, to use this technique, one has to trade-off network bandwidth for disk space. In practice, a hybrid scheme might best balance the different trade-offs. Another serious problem is that querying indexes directly might adversely affect load balance. However, load balance issues could be addressed by caching of indexes or prior query results [34].

3.5 Summary

In this chapter, we have presented a distributed algorithm for ranking search results. Our solution demonstrates that distributed ranking is feasible with little network overhead. Our approach is to adapt an existing technique from information retrieval, namely Vector

Space Model (VSM) [76] to Peer-to-Peer systems. We make use of random sampling to generate the vector representation of documents and queries used by VSM. Through formal analysis, we show that our sampling scheme is scalable for both uniform and arbitrary document distributions. We show that the number of samples needed depends only on the ratio number of documents to the number of nodes in the system. In order to store these vectors, we extend the existing distributed index infrastructure to store the vector representations in addition to storing the document entries. During evaluation, each query is evaluated locally at the index of each keyword in the query, and the top-K results from each of these local computations are aggregated to identify the eventual set of results. We validate our approach through detailed simulations. Our simulation results show that the accuracy of our distributed algorithm is comparable to that of a centralized system.

There are several areas worthy of further investigation. Performance could potentially be improved by mechanisms such as relevance feedback and caching. Our analysis could be extended to account for popularity-based replication. While our system provides one solution to distributed ranking, other approaches, e.g. using gossip (as described in Section 3.3.6), are also promising.

Chapter 4

Caching Search Results Using View Trees

In Chapter 3, we looked at the problem of reducing the amount of data transferred while answering queries. We showed that ranking is an effective strategy for this purpose and described an distributed algorithm to rank results over P2P systems. In this chapter, we address the problem of reducing the amount of data transferred while evaluating queries. In particular, we addresses the problem of reducing the amount of data exchanged while evaluating multi-keyword queries.

4.1 Introduction

Inverted indexes of keywords allow for evaluation of single-keyword queries without flooding the entire network. The straightforward use of distributed indexes to answer queries with two or more keywords , however, is potentially quite expensive. While not much can be done when evaluating queries of keywords connected using disjunctions (in the worst case all the results need to be returned), the evaluation of conjunctive queries typically requires the entire index for one or more keywords to be sent across the network. Unfortunately, indexes may be extremely large; DHTs are intended for use with extremely large distributed data sets (data sets with up to 10^{14} objects have been discussed [42]). Further, attributes are often skewed according to Zipf distributions [91]), potentially resulting in the size of at least some indexes growing linearly with the number

of system objects, and hence becoming quite large.

There are ways to minimize the amount of data transferred as part of query evaluation. The most trivial approach is to use a smart implementation that transfers the smaller index over the network. Reynolds and Vahdat [70] improve upon this trivial approach using techniques originally proposed to perform optimal semi-joins [61] in the area of distributed databases. Their method makes use of Bloom filters [9] to transfer digests of the data stored in the index. Since digest, and not entire indexes are transferred, the amount of data transferred is reduced. Note that the amount of data transferred could be reduced even further with the use of Compressed Bloom filters [59]. However, when Bloom filters are used, all the entries of the indexes would have to be scanned, leading to a high processing cost per query. Hence, the overhead, either in terms of bandwidth or processing (depending on whether Bloom filters are used), multiplied over billions of queries, is *prohibitive*.

We address the shortcomings of the above approaches by using *result caching*. Previous studies (e.g.,[78]) have indicated a Zipf-like distribution for queries in P2P systems such as Gnutella. Our approach is to exploit the locality in Zipf-like query streams to cache and re-use results between queries. We not only maintain single-keyword indexes but also conjunctive multi-keyword indexes, which we call *Result Caches*. Repeat multi-keyword conjunctive queries are answered by returning the results stored in these result caches. Queries that do not have cached results are evaluated, if possible, using the cached results of previously evaluated sub-queries. Non-conjunctive queries are evaluated by converting the query into its disjunctive normal form and evaluating the conjunctive components generated.

Our methods permit efficient evaluation of conjunctive queries using these result caches by using a distributed data structure called a *View Tree*. View trees can be used to efficiently cache, locate, and re-use stored results. Further, the view tree is a flexible data structure that preserves local autonomy. The decision of what to cache, and for how long, is made locally by each peer. Our results show that using the view tree can reduce search overhead (both network and processing costs) by more than an order of magnitude. To summarize, our main contributions are:

- We present a distributed data structure (*view tree*) for organizing cached query results (*views*) in a P2P system.
- We describe how to create and maintain multi-keyword indexes while maintaining the autonomy of peers.
- We present methods that use cached query results stored in a view tree to efficiently evaluate queries using a fraction of the network and processing costs of direct methods.
- We present results from detailed simulations that quantify the benefits of maintaining a view tree.

The rest of the chapter is organized as follows: Section 4.2 presents our search algorithm and our methods for creating and maintaining the view tree. Section 4.3 summarizes our results. We summarize this chapter in Section 4.4.

4.2 Searching Large Namespaces

We describe our result caching framework and the associated distributed data structure, that we call the *View Tree*, in this section. We show how to efficiently create and store result caches in the View Tree. We then show how we can use the View Tree to efficiently evaluate multi-keyword queries. We show how the View Tree can be used in conjunction with existing distributed lookup schemes such as Distributed Hash Tables (e.g., Chord [80], CAN [68], and Pastry [72]) and distributed directories (e.g., TerraDir [7]). We also discuss how we handle practical issues such as load-balancing, non-uniform index sizes and failure resilience. We start-off by describing our assumptions about the data and query model.

4.2.1 Data and Query Model

We assume that objects are uniquely identified by their names. Locating an object given its name is the basic operation supported by the P2P infrastructure (e.g., using a DHT). Each object has associated meta-data that we model as a set of keyword-value pairs. Attributes of an object could include keywords (for keyword search), the entire set of terms in the document (for full-text search), or other domain-specific attributes. Searching for objects by querying their indexed meta-data is the main operation that we explore.

For ease of exposition, we assume that the attributes are boolean, i.e., either an object has a named attribute or it does not. Thus, search for keyword a would return all objects that contained keyword a. Queries are thus boolean combinations of keywords

(e.g., $a \wedge (b \vee \neg c)$). Our methods use conjunctive queries of the form $a_1 \wedge a_2 \wedge \ldots \wedge a_k$ as the building blocks for supporting more general queries.

Note that the protocols in this chapter readily generalize to other types of attributes (including keyword search, arbitrary word search, and range queries) depending on how the individual inverted indexes (described next) are computed. Our primary focus is on the distribution, maintenance, and use of indexes and view caches, and not on the methods used for managing individual indexes. Specifically, we are interested in identifying and locating indexes that may be used to evaluate a given query, and not on the specifics of the data structures for the indexes themselves. Our methods support the identification and location of B-tree indexes, hash indexes or R-tree indexes equally well. Similarly, our methods could be generalized easily to more structured meta-data representations, such as XML.

Storing Inverted Indexes: We assume inverted indexes can be stored in the P2P network in a specially designated part of the namespace. For example, the indexes for keywords `manufacturer` and `price` are named `/idx/manufacturer` and `/idx/price`, respectively. (We use `/idx` as the reserved namespace for indexes; in practice, a more application-specific name is appropriate.) With this naming scheme, locating indexes is no different from locating other objects in the P2P system. Further, this scheme is applicable to both hash-based and tree-based methods for object lookup. For example, if the index in question is `/idx/price`, the index could be stored in the peer determined by the key $k = \text{Hash}(\texttt{/idx/price})$. In hierarchical systems(e.g., [7]), the index is stored in the node with fully qualified name `/idx/price`.

4.2.2 Caching Search Results

Inverted indexes, which essentially are lists of objects sharing common keywords, allow for single-keyword queries to be evaluated without flooding the network. Thus, a search for keyword a would efficiently and quickly return all objects that contain keyword a. Evaluating multi-keyword queries (e.g., $a \wedge b$), however, is very expensive and may require transferring large indexes over the network to sites with other inverted indexes. To avoid such large data transfers, our method uses cached query results or *result caches*. Our approach is motivated by the fact that previous studies (e.g., [78]) have observed a Zipf-like distribution for queries in P2P systems. This implies the existence of a high degree of locality in the query streams that can be exploited to reduce the amount of data transferred.

Formally, a result cache is the materialized result of a conjunctive query. For example, the result cache abc is the materialized result of the query $a \wedge b \wedge c$. Result Caches for non-conjunctive queries are obtained from the disjunctive normal form of the queries. The idea of using cached results and materialized views to enable faster query processing has been studied extensively in the database literature (e.g., [48], [37], [90]) and is also related to the problem of answering queries using views (e.g., [39], [85]). However, as we describe below, when caches are scattered over a P2P network instead of located at a centralized server, query evaluation using these caches poses a modified set of challenges: (a)*locating* a set of useful caches and (b)*selecting* a good subset for query evaluation.

The *location* problem consists of identifying result caches that are relevant to the query. At first glance, it may appear that the problem can be solved by assigning a canon-

ical name to each result cache (so that equivalent expressions of the same result cache are unified) and using the facilities of the P2P network for locating the caches. For example, if we render result caches in canonical form by listing the attribute in sorted order of their names, equivalent result caches $a \wedge c \wedge b$ and $b \wedge a \wedge c$ both map to $a \wedge b \wedge c$. Unfortunately, this idea solves only part of the problem: locating a result cache that is equivalent to a given result cache. This idea does not help us locate the result caches that are helpful in evaluating the above query in the absence of the result cache $a \wedge b \wedge c$. For example, the query may be answered by using result caches $a \wedge b$ and $b \wedge c$ or using $a \wedge b$ and $a \wedge c$, or using $a \wedge b$ and c, and so on. The number of result caches that are useful for answering this query is exponential in the size of the query (number of keywords) even without the repetition of equivalent result caches. This task highlights an important difference between this problem and the well-studied problem of answering queries using views: database methods assume that the set of views is well known and typically small, while in a P2P environment this set is unknown and large.

The *selection* problem, consists of determining a good query plan based on the available result caches (and statistics such as cardinalities of result caches). Prior work on this problem, has typically assumed the existence of a centralized repository containing the result cache meta-data. Unfortunately such an approach is not ideal in a P2P environment. The centralized approach equates to a particular peer storing and maintaining all indexes, as well as responding to all index lookups. Such a design has several problems. Not only is it unfair to this peer, it also creates a performance hotspot and a single point of failure. Further, this list would have to be kept reasonably consistent so that updates to the data can be reflected in the cached views.

4.2.3 The View Tree

Our solution to the problems of result cache location and selection is based on a distributed data structure we call the *View Tree*. The view tree maintains a distributed snapshot of the set of result caches materialized in the network, allowing efficient location of the result caches relevant to a query. We now describe the structure of this tree, the search algorithm used to locate result caches for a query, and the methods used to maintain the tree as both data and result caches in the network change.

Each node in a view tree represents a materialized conjunctive query. That is, each node corresponds to the cached results of a conjunctive query of the form $a_1 \wedge a_2 \wedge \ldots \wedge a_k$, where a_i are keywords. For brevity, we refer to result caches and their corresponding view-tree nodes as simply $a_1 a_2 \ldots a_k$, the conjunctions being implicit. Each view tree node is, in general, located at a different network host. Thus, traversing a tree link potentially incurs a network hop. Figure 4.1 depicts an example of such a view tree. The root of the tree is labeled with the special index prefix $/idx$. The first level of the tree represents all the inverted indexes (single-keyword result caches) in the network. The multi-keyword materialized results are mapped to nodes at greater depths in the tree. For example, the node bc corresponds to the result cache $b \wedge c$ and the node fcb corresponds to the result cache $f \wedge c \wedge b$.

In order to identify logically equivalent result caches (e.g., bac, cba, and bca), we refer to each result cache using a canonical name. The simplest choice for such a canonical name is the lexicographically sorted list of keywords (abc in our example). However, this scheme is likely to result in very unbalanced trees. Subtrees rooted at nodes labeled with

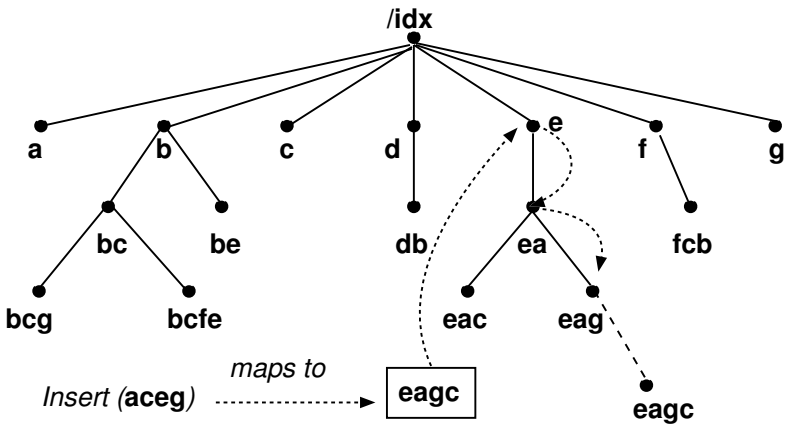


Figure 4.1: Example of a View Tree. Note that a node $a_1 a_2 \ldots a_k$ represents the result cache $a_1 \wedge a_2 \wedge \ldots \wedge a_k$. The picture depicts a result cache $a \wedge c \wedge e \wedge g$ being inserted into the view tree.

keywords that occur early in the sort order would likely to be much larger than those corresponding to keywords later in the sort order. This bias would create routing imbalances and hotspots. We avoid this problem by defining canonical names using a *permuting function* on the keywords of a result cache. The function is chosen so that it is equally likely to generate any one of the $l!$ permutations of a result cache with l keywords. Methods for generating such permutations are well-known [51]. All nodes use the same function; given the set of attributes, *all nodes* deterministically generate the same permutation.

View tree construction is illustrated in Figure 4.1. Suppose we wish to insert the result cache $aceg$ into the view tree. Further, suppose that the permuting function applied to this result cache results in $eagc$. The parent of this result cache in the view tree is the node corresponding to the longest prefix of $eagc$, i.e., eag. As illustrated by Figure 4.1, finding the parent of a new result cache when it is inserted into the view tree is simple:

we follow tree links corresponding to each keyword in the result cache, in order; the node that does not have a link to the required keyword is the parent of the node representing the new result cache.

Deleting Caches A result cache is deleted by locating its node n in the view tree. If n is a leaf of the view tree, it is simply removed from the tree. If n an interior node, then its parent $P(n)$ becomes the new parent of n's children. We also assume that the parent and children exchange heartbeat messages to handle ungraceful host departures. In the case of such failure, the child will attempts to rejoin the tree by following the insertion procedure. Thus, failures result in only temporary partitions from which it is easy to recover. The policy decisions of which results to cache, and for how long, are made autonomously by each host in the network using criteria such as query length, result size, and estimated utility.

4.2.4 Answering Queries using the View Tree

Given a view tree and a conjunctive query, finding the smallest set of result caches to evaluate the query is NP-hard even in the centralized case (by reduction from *Exact Set Cover* [29]). Thus, our approach is based on heuristics. We do a depth-first traversal of the view tree such that each visited node contains at least one of the query's attributes that has not yet been covered.

For example, Figure 4.2 depicts the actions of our method for the query $cbagekhilo$. The circled numbers in the figures denote the order in which computation occurs at view tree nodes. Intuitively, the algorithm first locates the best prefix match, which is $cbag$.

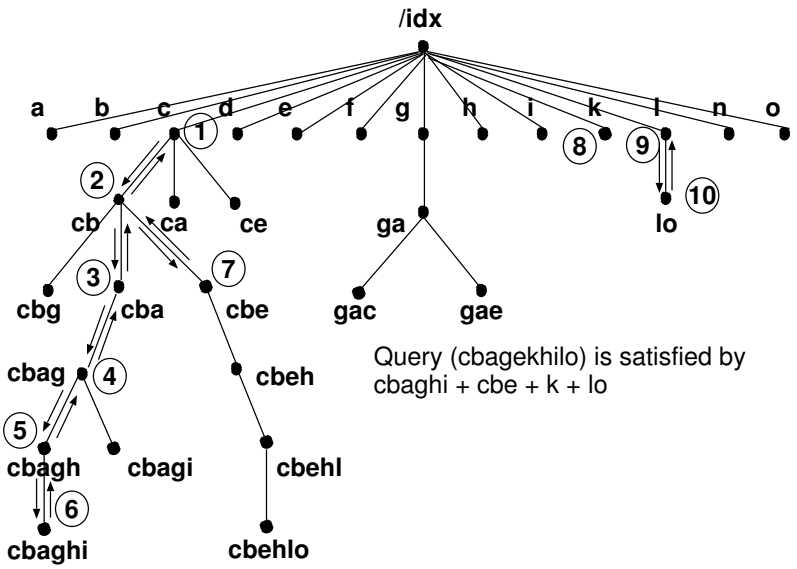


Figure 4.2: Example of a search using the View Tree. The search proceeds along the direction marked with arrows visiting the nodes in order they are numbered.

Even though the longer prefix $cbage$ is not materialized, the $cbagh$ child of $cbag$ is useful for this query, and thus this node is visited next. After node cbe, the query does not proceed to its child $cbeh$ because the node does not have any keyword that has not already been covered. The query can finally be answered using the intersection of the result caches $cbaghi, cbe, k$ and lo.

Each step of this search algorithm has the useful property of being guaranteed to make progress. If there is no result cache equivalent to the query, then each visited view tree node results in locating a result cache that contains at least one new query keyword (one that has not occurred in the result caches located so far). A given result cache can not only be used to satisfy a query equivalent to the one that created it, but also queries with keyword sets that are supersets of the initial query.

4.2.5 Generalized View Tree Search Algorithm

The search algorithm described so far essentially finds a cover for the set of keywords in the query using the sets of keywords occurring in the result caches. We may define an *optimal query plan* as one that uses result caches that contain as few tuples as possible, resulting in the smallest data exchange. Given the computational complexity of even the simpler set-cover problem, insisting on such an optimal query plan is not realistic. However, our preliminary experiments revealed that substantial benefits may be realized by avoiding plans that fare particularly badly by this metric. In the absence of global data statistics, we estimate the size of a result cache using the number of keywords in the result cache name. Since our queries are conjunctive, result caches with more keywords are expected to be smaller, and thus preferred in query plans.

A generalized version of our search algorithm is one that tries to find a query plan in which each result cache has at least t keywords, where t is parameter that may be set on a per-query basis at runtime. The essential difference from the earlier search method is the stopping condition: the earlier method stops when all keywords in a query are covered. The generalized search method stops when either all keywords in the query are covered with result caches of length (number of keywords) at least t or when none of the nodes explored so far has a child that represents a result cache of length l that can be used to replace a result cache of length $l' < t$ in the current plan. Note that the earlier search algorithm is a special case of the generalized algorithm with $t = 1$.

Figure 4.3 depicts the actions of the modified method on the query from the previous example. We use a threshold $t = 6$, implying that the method attempts to evaluate the

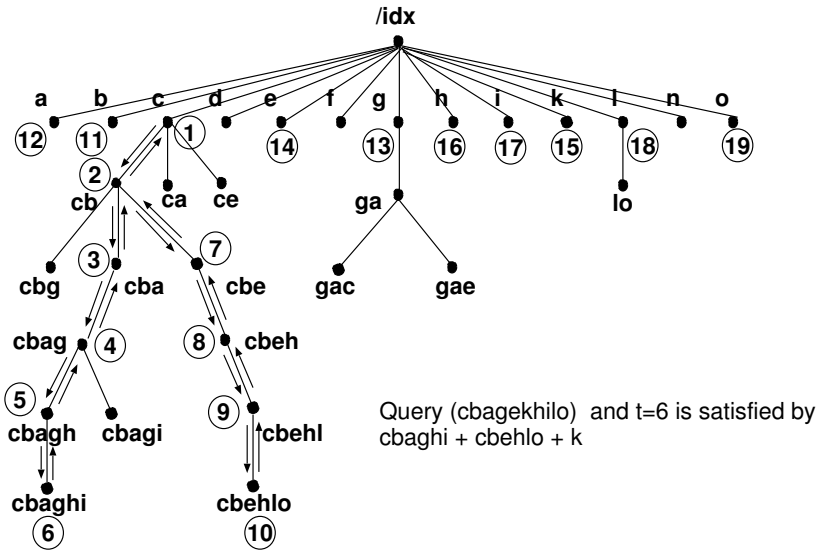


Figure 4.3: Example of the generalized search algorithm. The search still proceeds along the direction marked with arrows visiting the nodes in the order they are numbered.

query using result caches of at least six keywords each. The search proceeds from cbe to $cbeh$ (even though $cbeh$ is not in the canonical order of the query) because the latter covers more keywords. In this process, it also finds $cbehlo$. However, it now starts searching for a six-keyword result cache that contains k. It does so by visiting every inverted index not yet visited. Since there are no six-keyword indexes containing k, the search visits indexes for all 10 keywords before deciding on using $cbaghi, cbehlo$ and k to evaluate the query. This generalization presents a trade-off between the number of view tree nodes visited and the number of tuples transmitted.

4.2.6 Updating Result Caches

Since the view tree stores result caches of network data, there is an implicit consistency requirement that the contents of a result cache be identical to the result of evaluating the corresponding query directly on the network data. This implies that insertion of an object requires that result caches containing one or more of its keywords be updated. This update procedure need not be invoked immediately for most applications. For example, the presence of a document that does not appear in the indexes and result caches for several hours is not a serious problem for typical text-search applications. Further, not all indexes and result caches need be updated at once. Since shorter result caches (fewer keywords) are likely to be used by a greater number of queries, the update procedure may prefer to update them sooner than longer result caches. Therefore, when an object's insertion is to be reflected in the view tree, we update result caches in order of increasing length: first, all single-keyword result caches (i.e., the inverted indexes); next, all two-keyword result caches, followed by all three-keyword result caches; and so on.

The procedure for updating indexes and result caches in response to an object's deletion is completely analogous to the insertion procedure. Updates triggered by deletions can be postponed even longer than those triggered by insertions because the index entries for the nonexistent object can be flagged with a tombstone in a lazy manner when they are dereferenced by applications. Similarly, when a document is updated, we follow the deletion procedure for the dropped keywords and the insertion procedure for the added keywords.

4.2.7 Storing Large Caches

Since P2P systems consist of a number of regular hosts and there is no control over which index(es) gets mapped to which host, it is likely that the hosts holding the indexes for popular terms will be under-provisioned. Further, in a large system with many objects, popular indexes and result caches may grow to be too large to be stored at any single node. The simplest approach to solving this problem is to partition these indexes and caches, and store them at multiple nodes.

The problem of data partitioning in P2P systems has been studied previously by Ganesan et al. [27] and in eSearch [82]. Both the schemes are similar in that lightly loaded nodes transfer their contents and re-join the system by taking a share of the space held by a heavily loaded node. To get a good distribution of keywords to nodes, existing nodes in the system have to continually re-join and remap their ranges. This leads to extra costs for transferring tuples due to re-mapping.

In our scheme, we use a simple partitioning scheme that adaptively distributes large indexes over multiple nodes. The design of our partitioning scheme is motivated by three guidelines: (a) Intersections of sets of object identifiers (required for answering conjunctive queries) should be efficient, (b) The overhead of updating indexes in response to insertion of new objects and other changes should be small, and (c) The partitioning scheme should not depend strongly on the underlying P2P object-location architecture.

Please note that the partitioning algorithm is described in detail in Chapter 6. We describe the algorithm briefly for the sake of completeness. We assume that each participating node allocates some fixed amount of disk space, say S, for storing indexes.

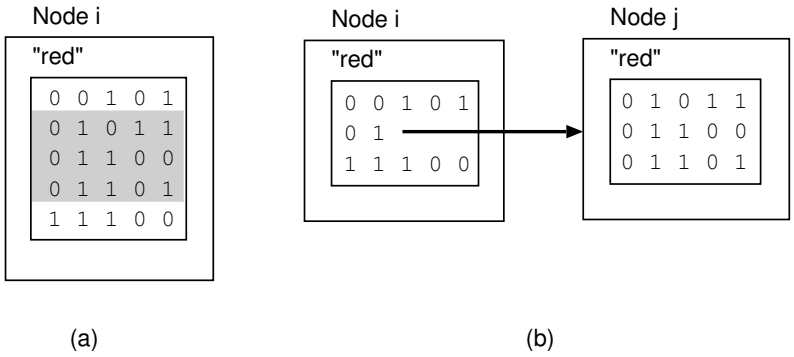


Figure 4.4: Index Splitting: In (a), node i has a single index for "red", containing the names (in binary) of five red objects. In (b), node i has split the index by sending the popular 2-bit prefix "0 1" to node j.

Partitioning is invoked when the storage exceeds this threshold. In this case, the largest index t is chosen for partitioning. The index is partitioned by dividing its object identifiers into equivalence classes based on common b-bit prefixes, where b is a system parameter. The largest such partition, $t.x$, is migrated to another peer. This peer is chosen randomly, by sampling, until one with sufficient space is found.

In Figure 4.4, we show an example of this algorithm. We set $b = 2$ for the example. In the figure, the prefix "01", is the largest partition and is moved to another peer j. Repeated application of the partitioning scheme outlined above results in the formation of a tree of partitions for each index, rooted at the node that is the original host for the index. We refer to this tree as the *partition tree*. For e.g. in 4.4, if node j needs to split, it will use bits 3 and 4 to partition the index further, creating a partition tree. Index lookups proceed by locating the partition tree's root using standard P2P services and proceeding down the tree as necessary. It is important to keep in mind that the partition tree is orthogonal to

the view tree and is over an individual result cache in the view tree.

4.2.8 Failure Recovery

If the index for a keyword were to become unavailable due to node or network failures, the performance of queries containing the keyword would suffer. In the worst case (e.g., a query that searches only on the keyword whose index is unavailable), it would be necessary to flood the network to generate complete results. Further, since a single large index can span multiple nodes due to partitioning, the probability that a large index is available depends on *all* of its host nodes being available. Thus, indexes (especially those that require flooding to recreate) must be redundantly stored. The level of redundancy depends on the probability of simultaneous node failures. Once the requisite number of static replicas are created, we can use standard techniques, such as those based on heartbeats [14], to maintain the proper number of replicas. Note that the locations of the current set of static replicas for a index, including all partitions, must be consistently maintained at all other replicas.

The performance of update operations depends on the method used to make indexes redundant. The most obvious (and often optimal) approach is to keep a literal copy of the index data at each replica. Note that these static replicas are only used for failure recovery and not for answering queries: we employ separate load-based replicas created using LAR for that purpose. It is also possible to use erasure coding, e.g. Reed-Solomon codes [69], to add redundancy into the system. In order to guarantee the same levels of resilience, Erasure codes require lesser disk space compared to replication. The reduced space,

however, comes at an increased update cost because the entire set of erasure codes must be re-published for each update. We compare the trade-offs between the two approaches, in detail, in Chapter 6.

4.2.9 Handling Load Imbalances

Locality in the query stream has a downside; nodes holding indexes of popular attributes will get overloaded and may not be able to respond to queries. Unfortunately, it is difficult to predict the popularity of keywords *a priori*. Further, the popularity of keywords changes over time. Hence, we use an adaptive replication protocol called LAR (refer Chapter 5) to prevent the overloading of peers hosting popular indexes.

Every peer that holds an inverted index specifies a local capacity and may request the creation of replicas when its load exceeds a threshold. The capacity of a node corresponds to the number of queries that can be routed or handled per second. Additionally, each node maintains a queue of queries that can be buffered for processing. Any arriving traffic that cannot be either processed or queued by a peer is dropped. Each peer defines load thresholds that are used to make decisions. Nodes attempt to create replicas when their load is above the threshold or when there is a significant difference in their individual loads. The load-based replication mechanism alleviates overloads by creating replicas of indexes hosted by overloaded peers onto lightly loaded peers.

A peer that receives a replica creation request is not obligated to accept it. If it rejects the request, the requesting peer waits until it finds another suitable candidate. When a result cache r is replicated, the peer storing the replica returns an acknowledgment

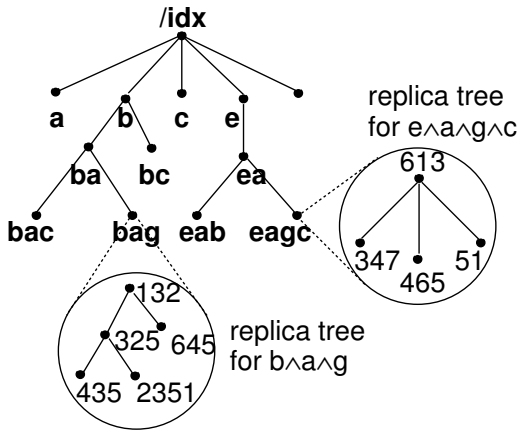


Figure 4.5: Example replica trees.

to the requesting peer. The requesting peer updates its list of replicas and propagates the updated information to r's parent $P(r)$ in the view tree (if one exists). Hosts with replicas may create further replicas, resulting in a *tree* of replicas for each index. Figure 4.5 shows the replica trees for views bag and $eagc$. The numbers in the replica tree represent the server IDs where the view is replicated. The parent of a peer in this tree is the peer from which a replica creation request was received. In the figure, the peers with IDs 613 and 132 created the first view replicas and hence are the parents of their respective replica trees.

When $P(r)$ receives a query that can be answered using r, $P(r)$ picks one of the replicas at random and forwards the query to the replica. Note that adaptive replication works orthogonal to the search system. As far as the view tree is concerned, all adaptively created replicas of r are equivalent. Modifications to r are propagated transparently to all of r's replicas. When a replicated node is deleted from the view tree, its children are not immediately grafted to its parent as described in Section 4.2.3; instead, they are left in place as long as there exists atleast one replica.

4.3 Results

This section presents simulation results from our experiments with the algorithms described in section 4.2. We implemented a packet-level simulator for evaluating our algorithms. We chose documents from the WT-10g data set of the Text REtrieval Conference (TREC [84]) data as our source data. We generated the mapping between data-items and keywords using the term-weighting schemes normally used in information retrieval methods like Vector Space models [76]. The term-weighting scheme computes the weight t_i of the term i in a document using equation 4.1, where f_i is the frequency of the word i in the document, n_i is the total number of documents that have the term i and N is the total number of documents in the collection.

$$t_i = (\ln(f_i) + 1) \cdot \ln\left(\frac{N}{n_i}\right) \tag{4.1}$$

The weights of terms in each document are then normalized and terms with weights greater than a threshold are considered as keywords for the document. For our data, we experimented with different values of threshold and settled on 0.12 as it gave a good distribution of keywords and index sizes. This resulted in a collection of 500k documents with 390k unique keywords. The number of keywords per document ranged from 1 to 55 with an average of 14.

For the queries, we chose a representative sample of web queries from the publicly available `search.com`. We also extracted web search queries from cache logs available from `ircache.net`. These web query sets do not provide an associated document set over which these queries would be valid. Hence, we could not directly use these queries

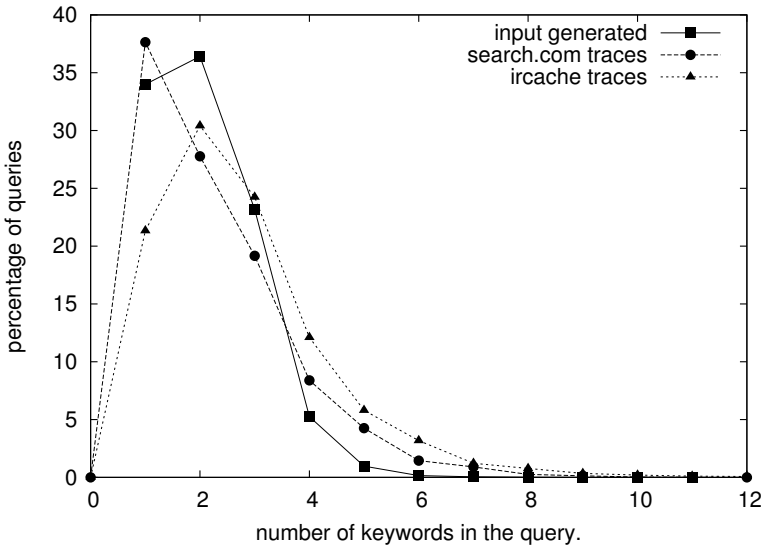


Figure 4.6: The distribution of keywords in our synthetically generated queries compared to actual query traces obtained from `search.com` and `ircache.net`.

for our experiments. We, therefore, generated queries with the same characteristics (distribution of keywords over queries, number of keywords per query, etc.) as the trace queries using keywords from the TREC-Web data set. Figure 4.6 shows the distribution of keywords in our queries. We generated query streams consisting of 1,000,000 queries. In order to represent locality in the query stream, we generated a *popular set* of 100,000 queries (10% of the total queries). Each query stream was generated by drawing either 5%, 10%, 20%, 50%, 90% or 99% of the queries, uniformly at random, from this popular set. The reminder of the queries, in each of the query streams, were generated by determining the size from the distribution and picking individual keywords uniformly at random from the entire set of keywords. By default, we used the query streams with 90% and 10% of queries from the popular set as the set with high- and low-locality respectively.

# of keywords	90% locality min	avg	max	10% locality min	avg	max
1	1	18	8514	1	18	8514
2	0	39	6540	0	6	6540
3	0	39	4776	0	6	4776
4	0	17	4480	0	2	3887
5	0	3	3614	0	1	3614

Table 4.1: Sizes (in no. of tuples) of n-keyword indexes.

Table 4.1 shows the sizes of resulting inverted indexes and cached results depending on the locality in the query stream. Note that our query stream generated a small number of indexes with more than five keywords, but their effect is negligible and they are not show here.

4.3.1 Experimental setup

The base system for our experiments consisted of 10,000 servers exporting 500,000 documents. By default, we ran each experiment with the 90% locality query stream. The query inter-arrival time was exponentially distributed with an average of 10 milliseconds. We assumed that hosts allocate some fixed amount of disk space to store the result caches. In most of our experiments, we *did not* take into account the space taken up by the keyword indexes. We did this both for correctness of the results and because we were interested primarily in understanding how various factors affected the performance of view trees. We allocated 750 tuples of space at each node by default, for the multi-keyword caches. This space was managed as a cache, with result caches being deleted from the view tree when marked for replacement by LRU. Unless otherwise noted, we set the search parameter $t = 1$ in the generalized search algorithm; this results in the use

of the regular view tree search algorithm. The rest of this section discusses the detailed experiments we performed and analyzes the results.

4.3.2 Effectiveness of Result Caching

Our first experiment quantifies the effectiveness of result caching. We consider six different input query distributions, each with varying degrees of locality. We start with a query stream with 5% of queries (50000) from the 100k working set and gradually increase locality till 99% of the queries are from the working set. We plot the result of the experiment in Figure 4.7. The x-axis of the represents the level of caching in the view tree. There are *no* caches when we have only inverted indexes. We increase the amount of caching in each step by progressively caching 2-keyword queries, 3-keyword queries and so on until we cache all queries. The y-axis presents the fraction of data exchanged for computing the results. We *normalize* the amount of data exchanged for each caching level with the case when there are no caches.

Figure 4.7 shows that result caching significantly reduces the amount of data transferred for query evaluation. The benefit is enormous for streams with high locality; when 90% of the queries are from the working set, caching 2-keyword queries reduces the number of tuples transferred by 85%, whereas caching all queries reduces the number of tuples transferred by over 90%. Somewhat surprisingly, queries with much lower locality(5% and 10%) also benefit from result caching. This is because the intersection of two random keywords usually results in a small index, which can be efficiently cached and reused. Finally, note that Figure 4.7 shows that a majority of the view tree's benefit coming from

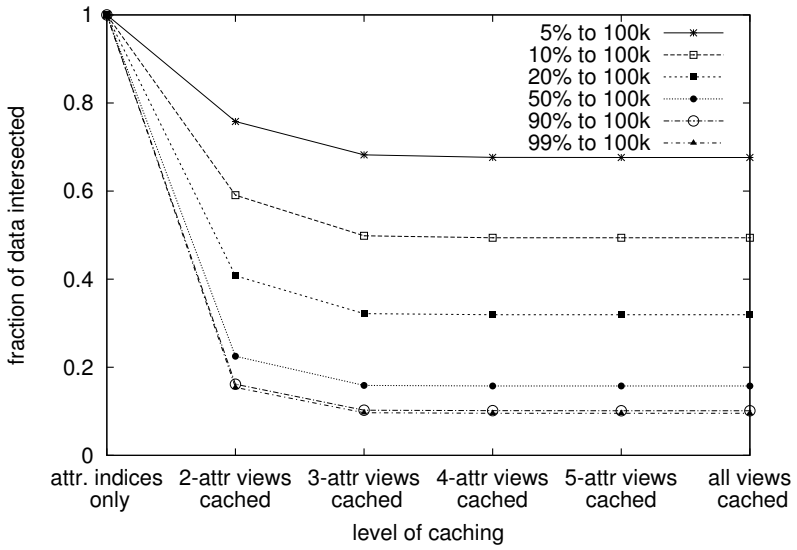


Figure 4.7: Caching benefit by level.

only the first two or three levels of the trees. This is because our query stream consists primarily of queries with few keywords (see Figure 4.6).

In Figure 4.8, we plot, over time, the number of hits to different k-keyword caches. The plot also shows the actual number of queries with different numbers of keywords over time. For each curve, we sum the hits and the queries over a 30-second period. There are several interesting points to note: first, result caching is effective, in that the number of (costly) accesses to the inverted indexes decrease rapidly as the multi-keyword caches are built. Second, after the two-keyword caches are built, almost all two keyword queries are satisfied using these caches. Ideally, the curve for hits on inverted indexes would converge with the line for single-keyword queries, and likewise for caches and queries with more keywords. They do not because of the multiplicity of the 3- and higher-keyword queries: there are so many distinct sets of keywords that it is not feasible to cache them all (or for

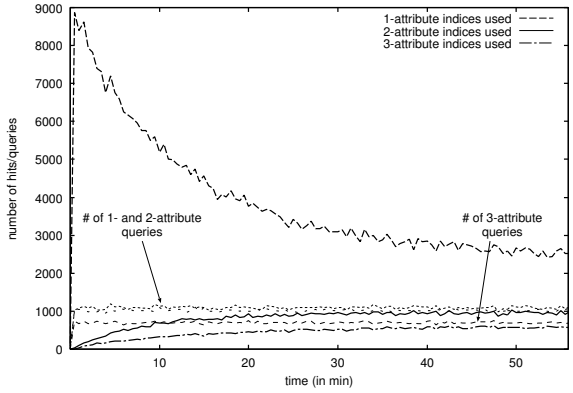


Figure 4.8: Index access and k-keyword queries over time.

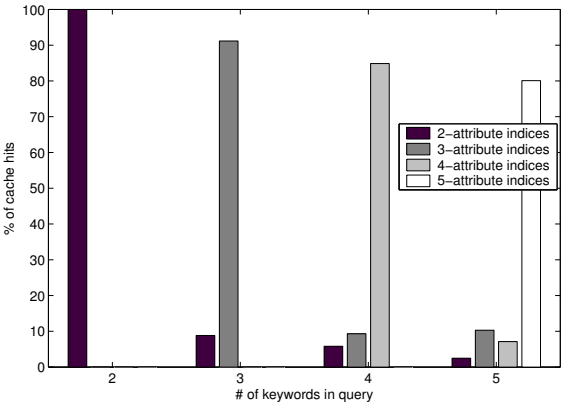


Figure 4.9: View Tree Hits

the intermediate result caches to cover them all), and satisfying these queries sometimes requires using inverted indexes.

In order to evaluate whether higher-level trees were more useful for queries with more keywords, we need to look specifically at the data for these queries. Figure 4.9 breaks down the data to show the number of each type of result cache used, for queries with between one and five keywords. The results show that the majority of result caches used in satisfying queries of more than two keywords have more than two keywords, i.e., deep view trees can be quite effective for queries with many keywords.

In fact, the majority of all matches in the view tree turn out to be exact matches. One explanation is that this is, to some extent, a function of a query stream in which the "working set" queries repeat. We tested this theory by running another set of experiments where the occurrence of individual keywords had locality, instead of specific multi-keyword queries. As expected, we found a much lower rate of exact matches, and a higher number of useful hits where the result cache is a subset of the query.

	10% locality		90% locality	
# of Updates	# of Extra Updates	Reduction in data transferred	# of Extra Updates	Reduction in data transferred
1k	344	95.50 M	654	165.99 M
10k	2064	95.53 M	5628	165.99 M
100k	24113	95.71 M	57989	166.00 M
1M	210k	97.33 M	534k	166.08 M

Table 4.2: Update overhead for 1 million queries. All the results are in number of tuples.

4.3.3 Cache Maintenance: Updates

Table 4.2 summarizes the cost and benefit of maintaining the view tree. In these experiments, we assume that we have infinite disk space and cache *all* results (which corresponds to the worst case for update overheads) and vary the number of updates. In each update, a keyword is added to or deleted from an object. We perform one update for every k queries where $k = \{1, 10, 100, 1000\}$. The number of actual updates performed depends on the number of caches in the system.

In Table 4.2, we present (1) the number of *extra* updates needed to maintain the view tree and (2) the reduction—due to the view tree—in the amount of data transferred to answer the queries. From the table, it is clear that even for unrealistically high update rates (one update per query), the cost of maintaining the view tree is essentially negligible.

The number of updates is higher in the query stream with higher locality. This is a counter-intuitive result because of the following reason: the stream with lower locality would create more number of replicas, and one would expect that this would result in more updates. The reason for the counter-intuitive result is a combination of factors. The updates typically tend to modify the popular keywords. The working set consists

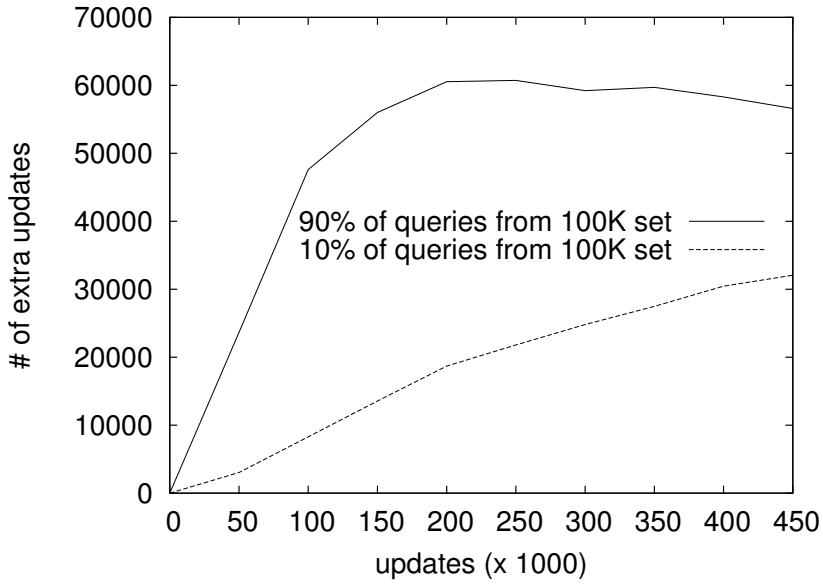


Figure 4.10: Effect of locality and time on the number of extra updates

of queries containing these popular keywords. Since the updates and queries are online, that the set of popular caches are created quickly for the case with the higher locality. This means that each update has to propogate to more caches much before the case with low-locality. We show evidence of this phenomenon in Figure 4.10.

Figure 4.10 compares the number of extra updates over time when there is an update for each query. We cluster the updates into bins of 50,000 updates. The Y-axis shows the number of extra updates for each bin. As is clear from the result, the number of extra updates grows much more quickly when there is locality. When there is low locality, the number of extra updates grows at a much slower rate.

Of course, as is evident from the table, this small increase in number of updates is quickly dwarfed by the savings due to the cache hits in the query phase.

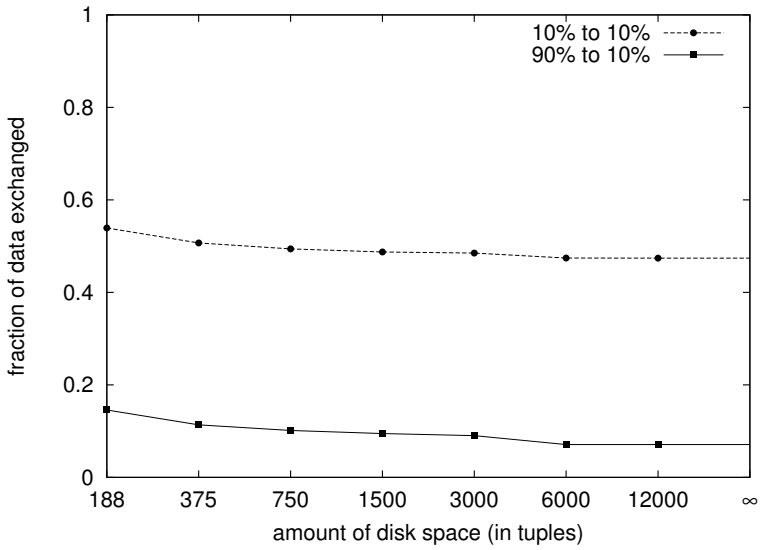


Figure 4.11: Effect of disk space: Even a small fraction of space allocated for the caches helps reduce the number of tuples intersected.

4.3.4 Effect of Disk Space

This experiment quantifies the effect of the amount of disk space on the efficacy of the view tree. Recall that in our simulations, the nominal amount of per-node disk space is 750 tuples. In Figure 4.11, we plot the performance of the view tree for disk space allocations ranging from 188 tuples (0.25×750) to 12000 tuples (16×750). The figure also contains a result with unbounded disk space (∞). We assume that keeping any information takes up space; caches with zero entries also take up space.

As expected, the number of tuples exchanged to resolve a query reduces as the amount of disk space increases (because there are more direct cache hits). The interesting aspect, however, is that disk space does not affect the performance of the view tree. For example, with 90% locality in the query stream, the number of tuples exchanged drops

by 83% with only 188 tuples. This surprising result can be explained by the fact that most of the multi-keyword caches are small (refer to Table 4.1) and hence can be easily stored. Also, for our experiments, at 12000 tuples, the performance is almost equivalent to having unbounded space.

4.3.5 Tree Organization and Search Procedure

Recall that the view tree search algorithm (Section 4.2.5) is based on heuristics, and is not guaranteed to find all useful indexes. Figure 4.12 compares the bandwidth reductions achieved by the view tree heuristic with the best possible reduction, as discovered via exhaustive search. We computed the results for exhaustive search using result caches that result in the smallest data exchange, just as with the unmodified view tree algorithm. The results show that both organizations perform similarly; the view tree approach differs from the exhaustive search by a maximum of 5% when we cache results of two-keyword queries. As we increase the level of caching, the difference in performance stays around 1%.

4.3.6 Handling Popular Indexes

Popular indexes can induce load imbalance across view tree nodes. We address this imbalance with two techniques: (1) adaptive replication via LAR, and (2) the use of permutation in defining canonical result cache names. Figure 4.13 plots the fraction of queries answered versus caching level both with and without adaptive replication. For the latter case, we plot data for the system with and without name permutation. In this exper-

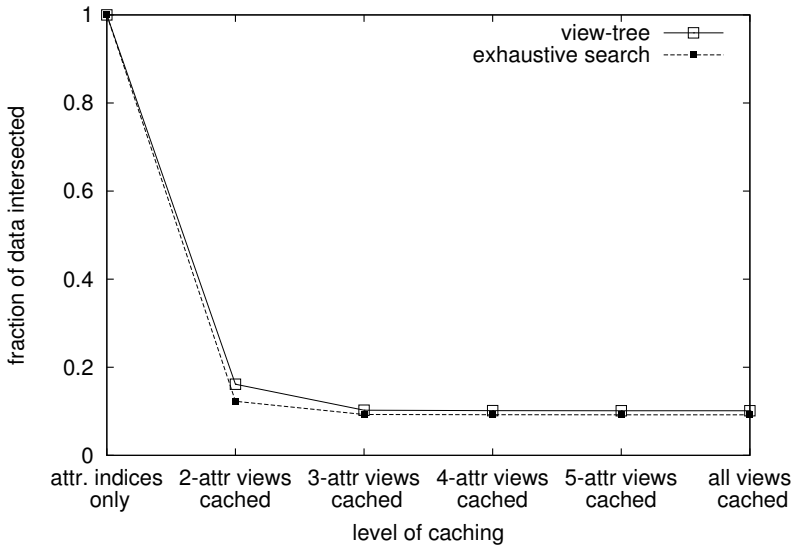


Figure 4.12: Effect of View Tree organization

iment, each peer can process 10 queries/second and can buffer a maximum of 32 queries. All queries received when the buffer is full are dropped. We carefully experimented with the system load to make sure that queries were not dropped because of a high query rate but rather because of the imbalance in the tree.

Figure 4.13 shows that adaptive replication is quite effective in distributing load (as indicated by the fraction of queries answered). Almost all queries (>99.99% in all cases) are answered irrespective of the level of caching.

Increasing the level of caching is also effective at reducing the number of dropped queries. With the non-permuted view tree, the number of queries answered increases from 91% to 96% with increasing level of caching, while almost all of the queries are answered with three-keyword caches and the regular view tree. The increase is because more caches imply a higher hit rate and more exact matches, and both of these effects

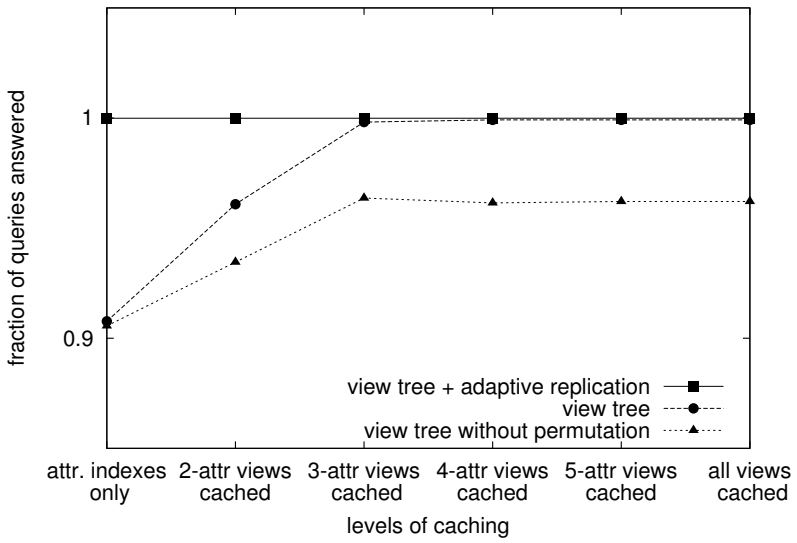


Figure 4.13: Number of queries answered with and without adaptive replication. Note that the y-axis ranges from 0.85 to 1.05

reduce bandwidth requirements. More caches also imply that the query load is distributed across a larger set of peers.

Figure 4.13 also shows the merit in creating the view tree by permuting the keywords identifying the index. The number of queries answered increases by an additional 5% with permutation, indicating that permuting the keywords facilitates the distribution of load over the nodes.

Adaptive replication comes with its own costs: the network overhead of creating replicas and the extra space taken up by these replicas. We quantify both costs in Table 4.3. The first row in Table 4.3 counts the number of replicas created per level of the view tree, while the second row tabulates the extra network traffic due to transferring these adaptive replicas. Note that in the worst cases (with inverted indexes and 2-keyword

	Inverted indexes	2-keyword caches	3-keyword caches	4-keyword caches	5-keyword caches	All caches
# of replicas	1281	904	411	327	339	339
data transferred	2.04 M	1.56 M	1.26 M	1.05 M	1.06 M	1.06 M

Table 4.3: Overhead of using the adaptive replication with view trees. The amount of data transferred is in tuples.

result caches), the cost of creating adaptive replicas (about 1300 create messages and 2 Megabytes of control messages over 1M queries) is negligible.

4.3.7 Handling Large Indexes

In section 4.2.7, we have argued that, in systems with many documents, indexes have to be partitioned onto multiple nodes. To understand the importance of partitioning we ran two sets of experiments.

The first experiment measured the importance of partitioning to maintain correctness of the indexes. To perform this experiment, we allowed each node to have 1500 tuples of disk space. This space was utilized to store both inverted indexes and cached results. The average space required to store just inverted indexes was 750 tuples. We allow an extra space of 750 tuples to store cached results. If a node runs out of space, it neither adds new entries in the inverted indexes hosted on the node, nor does it cache new results. Note that this is different from the experiments presented so far where we allowed the entire inverted index to be stored at a node, irrespective of its size. The amount of disk space for caches, however, is consistent with all our experiments. Our experiments showed that without partitioning, *only 28% of the queries were accurately answered*. When partition-

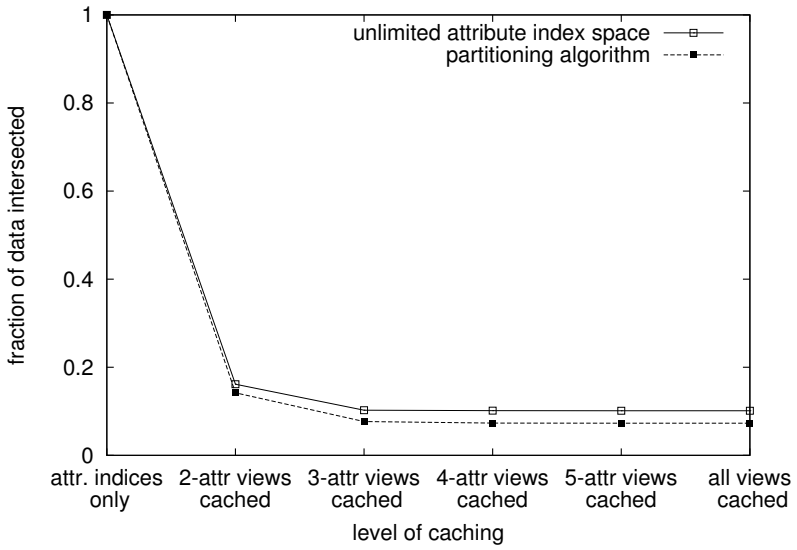


Figure 4.14: Performance of view tree with partitioning

ing was employed we were able to answer *all* the queries accurately for the same amount of disk space.

The second experiment measured the effect of partitioning on result caching. We compared the result in Figure 4.7 with 90% locality to the amount of data transferred for the same query stream with partitioning employed. In the case of partitioning, we also account for the data moved by the partitioning algorithm. Figure 4.14 presents the results of this comparison. It is clear from the figure that partitioning helps further reduce the amount of data exchanged.

4.4 Summary

We have described the design of a keyword search infrastructure that operates over distributed namespaces. Our design is independent of the specifics of how documents are

accessed in the underlying namespace, and can be used with all DHT-like P2P systems. Our main innovation is the view tree, which can be used to efficiently cache, locate, and reuse relevant search results. We have described how a view tree is constructed and updated and how multi-attribute queries can efficiently be resolved using a view tree. We have also described techniques for reconstructing the tree upon failures. We discuss and present algorithms for some of the practical considerations associated with implementing the view tree including load-balancing and failure resilience,

Our results show that using a view tree offers significant benefits over maintaining simple one-level inverted indexes. With our trace data, view trees reduce multi-keyword query overheads by over 90%, while consuming few resources in terms of network bandwidth and disk space. Our results show that a view tree permits extremely efficient updates (essentially zero overhead), and can produce significant benefits when servers fail in the network. Overall, our results show that view trees efficiently enable much more sophisticated document retrieval on P2P systems.

Chapter 5

Lightweight Adaptive Replication

Structured P2P systems provide both low latency and excellent load balance with query streams in which all data items are accessed with uniform probability. Under skewed access, which is often the norm, specific nodes in even structured P2P systems can be overloaded resulting in poor global performance. In this chapter, we describe and characterize a lightweight, adaptive, system-neutral replication protocol (`LAR`) that can quickly and efficiently redistribute load.

5.1 Introduction

Inverted indexes provide a simple method to allow search over structured P2P systems. Answering a query is merely an instance of data location (a lookup) with a fully qualified name. Unfortunately, this simple scheme is insufficient. In reality, there exists significant difference in the popularity of keywords with certain keywords requested much more frequently than others. This results in substantially higher query load on the nodes hosting the indexes corresponding to these popular keywords. Also, if the system consists of a number of end hosts connected over low-bandwidth links, then it is likely that the hosts holding the indexes for popular terms will be under-provisioned. If left unaddressed, this could result in nodes getting requests far beyond their capacity resulting in a large number of queries getting dropped at these overloaded nodes. Further, load

imbalances are also unfair on the peer hosting the popular index and acts as a disincentive for hosting indexes or participating in the system.

Existing DHT-based structured systems [80, 89, 68, 72] use cryptographic hashes to randomize the mapping between data item names and locations in an attempt to balance the load on individual nodes. Under an assumption of uniform demand for all data items, the number of items retrieved from each server (referred hereafter to as "destination load") will be balanced. Further, *routing load* incurred by peers in hash-based schemes will be balanced as well. However, if demand for individual data items is non-uniform, which is typically the case in practice (e.g., Gummadi et al. [36], Sripanidkulchai [78]), neither routing nor destination load will be balanced, and indeed may be arbitrarily bad. The situation is even worse for hierarchical systems such as TerraDir [7], as the system topology is inherently non-uniform, resulting in uneven routing load across servers.

The intuitive approach to address this problem is to statically create and store replicas of these indexes, and use all of these replicas to answer queries. The number of replicas created for each index would have to depend on its popularity; the higher the popularity, the more the number of replicas. This approach, however, is challenging to realize in practice because it requires knowledge of the popularity of indexes *a priori*. Even if this information were available, the scheme is ineffective because the popularity of keywords changes over time. Therefore, it is extremely hard to statically provision for such load imbalances.

So far, this problem has usually (e.g. PAST [73], CFS [21]) been addressed in an end-to-end manner by caching at the application level. Specifically, data is cached on all nodes on the path from the query destination back to the query source. Routing is not

affected, and "hot" items are quickly replicated throughout the network. However, the resulting protocol layering incurs the usual inefficiencies, and causes functionality to be duplicated in multiple applications. More importantly, our results show that while these schemes can adapt well to extremely skewed query distributions, they perform poorly under even moderate load because of their high overhead.

We describe a lightweight approach to adaptive replication that does not have the drawbacks of either static replication or application-level caching. Instead of creating replicas on all nodes on a source-destination path, we rely on server load measurements to precisely choose replication points. Our approach can potentially create replicas for an object on any node in the system, regardless of whether the original routing protocol would ever direct a query to the replica hosts. We augment the routing process with lightweight "hints" that effectively shortcut the original routing and direct queries towards new replicas (described in detail in Section 5.3). This protocol incurs much lower overhead, can balance load at fine granularities, accommodates servers with differing capacities, and is relatively independent of the underlying P2P structure.

The main contribution of this chapter is to show that a *minimalist approach to replication design is workable, and highly functional*. We derive a completely decentralized protocol that relies only on local information, is robust in the face of widely varying input and underlying system organization, adds very little overhead to the underlying system, and can allow individual server loads to be finely tuned. This latter point is important because of the potential usage scenarios for P2P systems. While P2P systems have been proposed as the solution to a diverse set of problems, many P2P system will be used to present services to end users. End users are often skeptical of services that consume

local resources in order to support anonymous outside users. User acceptance is often predicated on the extent to which end users feel they have fine-grained control over the intrusiveness of the service.

The rest of this chapter is structured as follows: Section 5.2 briefly describes the application level caching that is currently employed to address the issue of load imbalance. Section 5.3 gives an overview of our model and design goals. Section 5.4 describes the protocol in more detail. We present our experimental results in Section 5.5. Section 5.6 summarizes our findings and concludes the chapter.

5.2 Existing approaches to Adaptive Load Balancing

The problem of balancing the load of all participating nodes, in order to avoid "hotspots", is common to many scenarios. As expected, a lot of techniques have been proposed, both by the academia and the industry, to address the problem, . Existing load balancing solutions (and products) such as the Cisco Local/Global director [17] or even techniques used in content distribution networks such as Akamai [2], however, are simply not applicable in our context. This is because these systems require too much coordination and coupling between nodes and often require a centralized coordination point where global knowledge is available .

DHTs (such as Pastry or Chord) do not have any in-built mechanism to deal with non-uniform query distributions. Instead, they delegate the task to the individual applications running on top of them. For example, PAST [73] and CFS [21], which are distributed file sharing applications that run on top of Pastry and Chord respectively, implement their

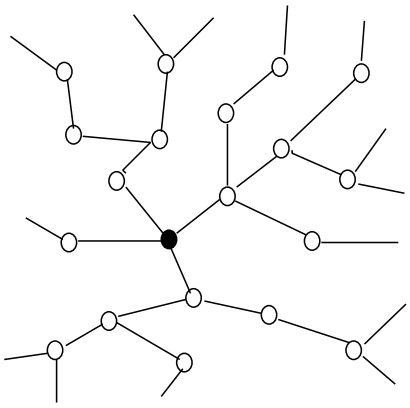


Figure 5.1: Snapshot of a base P2P network

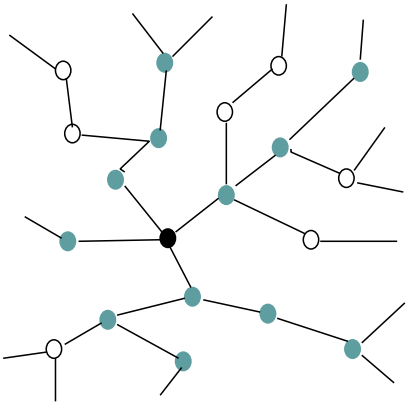


Figure 5.2: Snapshot of a P2P network with `app-cache`

own distributed replication scheme. Hotspots and dynamic streams are handled using caches which are used to store popular objects in the network, and lookups are considered resolved whenever cache hits occur along the path. CFS [21] for instance replicates data in k of its successive neighbors for data availability, and populates all the caches on the query path with the destination data after the lookup completes. We will refer to our generalization of the approach used in these applications as `app-cache` in the following.

`app-cache` deals with skewed loads through the use of caches. In a virgin state, this responding server will be the file's home. Copies of the requested file are then placed in the caches of all servers traversed as the query is routed from the source to whichever server finally replies with the file. However, subsequent queries for the file may hit cached copies because the neighborhood of the home becomes increasingly populated with cached copies.

As a result, the system responds quickly to sudden changes in item popularity. `app-cache` is very pro-active in that it distributes $k - 1$ cached copies of every single query target, where k is the average hop count.

Figure 5.1 depicts a simple network where the darkened dot represents the "home" server. An edge indicate that the server knows about the existence of the other end. Figure 5.2 shows how `app-cache` handles dynamic query streams by caching data (represented in shaded dots) in the path of the query. This leads to the neighbors caching copies of the data item and then its neighbors and so on.

5.3 Protocol goals and approach

The design of the replication protocol has been motivated by a couple of goals. Firstly, we wish to address overload conditions, which are common during flash crowds or if a server hosts a "hot" object. Therefore the goal of the replication protocol is to distribute load over replicas such that requests for hot objects or flash crowds can be handled. The key to achieving this is to employ Adaptive protocols. Adaptive protocols can cope efficiently with dynamic query streams, or even static streams that differ from expected input.

Second, we will attempt to balance load. Figure 5.3 shows a cumulative distribution function (CDF) of server loads with a uniform query distribution. The majority of servers are in an *acceptable* range, but a small subset of server have either very high or very low load (the two tails of the distribution). We concentrate on moving the relatively few servers in either tail into the "balanced" portion of the load curve.

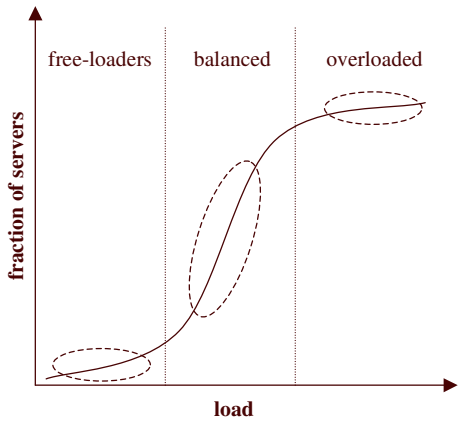


Figure 5.3: A CDF of load versus fraction of servers with a uniform query distribution. The majority of servers have acceptable load. We concentrate mainly on alleviating the relative overloading of those in the long high tail, and to a lesser extent on identifying "free-loaders", i.e. servers with little or no load.

Obviously, the sensitivity, overhead, and effectiveness of the algorithm will depend on exactly how *acceptable* is defined and what mechanisms are used to shed load. Our approach to achieving the above goals is to use load-based replication of data and routing hints and to augment the existing routing mechanism to use the replicas and routing hints.

Our third goal is to base all decisions on locally available information. Making local decisions is key to scaling the system, as P2P systems can be quite large. For example, the popular KaZaA file-sharing application routinely supports on the order of two million simultaneous users, exporting more than 300 million files [1]. Global decision-making implies distilling information from at least a significant subset of the system, and filtering decisions back to them. Further, there is the issue of consistency. There is a clear trade-

[1] As reported by the client.

off between the "freshness" of global information summaries and the amount of overhead needed to maintain a given level of freshness. Finally, systems using global information can be too unwieldy to handle dynamic situations, as both information-gathering and decision-making require communication among a large subset of servers.

The choice of local decision-making has its implications. For one, local decisions might be poor if locally available information is unrepresentative of the rest of the system. Also , local decision-making makes it difficult or impossible to maintain the consistency of global structures, such as replica sets for individual data items. A *replica set* is just a possibly incomplete enumeration of replicas of a given data item, which are the default unit of replication. Requiring that the "home" of a data item be reliably informed of all new and deleted replicas could be prohibitively costly in a large system. This difficulty led us to use soft state whenever possible. For example, instead of keeping summaries of all replicas at a data item's home, we allow some types of replicas to be created and deleted remotely without having any communication with other replicas or the home.

Finally, our protocol is intended to be independent of P2P structure. We intend our results to be applicable to DHTs [80, 89, 68, 72, 57, 41, 43, 38]; to hierarchical namespaces [7]; and also to unstructured P2P systems like Gnutella [31].

5.4 The LAR Protocol

In this section we describe the LAR protocol. We talk about the approaches we adopt to achieve the goals of the protocol. Then we discuss the various aspects of the protocol in detail. Finally, as an example, we describe how we adopt the protocol to

Chord (DHT-based system) and TerraDir (hierarchical system).

5.4.1 Protocol description

This section describes the policies that `LAR` uses for load re-distribution, replica and cache entry creation/deletion, and replica-augmented routing. There are three specific issues that must be addressed:

1. *Load measurement and replica creation*: The system must redistribute load relatively quickly in order to handle dynamic query streams. However, reacting too quickly could lead the system to thrash. We need to specify *when* new replicas are created, on what nodes, and which items a server replicates, and how replicas are discarded.

2. *Routing using cache hints and replicas*: Assume a server has knowledge of a set of replicas for a desired "next hop" in the routing process. The overlay routing algorithm must be augmented such that these replicas are visited instead of only the home node of an item. We need to specify which of the replicas to choose during routing, and (how) should the selection process attempt to incorporate knowledge of load at replica locations.

3. *Replica information dissemination and management*: New replicas are useless unless other servers know of their existence[2]. Information about new replicas must be disseminated, whether eagerly by a separate dissemination sub-protocol, or lazily

[2]This is not precisely true because routing can be short-circuited whenever a replica is encountered. However, this is a secondary effect.

by being appended to existing messages. Allowing remote sites to independently create and destroy replicas means that the number of system replicas of a given item is not bounded. The dissemination policy must determine the amount of replica pointer state that should be kept at each site, the way that new replica pointer information is merged with older information, and what state should be appended to outgoing messages (or pushed eagerly).

In the rest of this section, we specify, in detail, how LAR addresses each of these issues.

5.4.2 Load Measurement and Replica Creation

Local Load Measures: Our replication scheme is different from the existing schemes like [21] in that we introduce the construct of load in order to perform replication. We assume that each server has a locally configured resource capacity and queue length. We also assume that each server defines a high-load and low-load threshold. By default, the system sets high-load and low-load thresholds for each server based on fractions of servers' capacities. We also assume that we can keep track of the load due to each object (i.e., data and routing indexes) in the server.

For this work the capacity indicates the number of queries that can be routed or handled per second, and the queue length specifies the number of queries that can be buffered until additional capacity is available. Any arriving traffic that can not be either processed or queued by a server is dropped. The load metric in the protocol is abstract and in practice can be defined based on any of the factors like CPU load, network load,

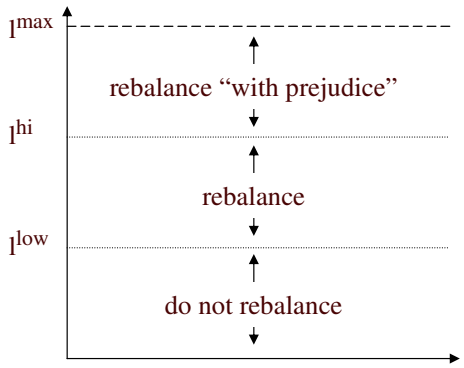


Figure 5.4: Local capacity thresholds: The server capacity is l^{max}. Load is sometimes re-balanced if greater than l^{low}, and always if greater than l^{hi}.

I/O load or even a combination of these factors. For example, Rabinovich et.al [66] make use of the ready queue, such as the output of the “uptime” command as a measure of computational load. Considering that most P2P clients like Kazaa and Gnutella allow the user to set the maximum upload and the download bandwidth, the application could keep track of the number of bits transferred or received to estimate the network load. Chawathe et.al. [16] suggest that the capacity should be a function of the server’s processing power, access bandwidth, disk speed etc.

As Figure 5.4 shows, high-load threshold indicates that a server is approaching capacity and should shed load to prevent itself from reaching its maximum capacity and drop requests. The intent of the low threshold is to attempt to bring the load of two servers closer to each other to achieve load balance. The difference in the load of two servers being greater than the low threshold indicates that one of the servers is much less loaded compared to the other and that it can be used to distribute the load. We will use the term “load balance” in the rest of this chapter to refer to this sense of distributing load.

Lastly, we assume that the fractions for thresholds are constant across the system in the simulations here, but the protocol will work unaltered with non- uniform fractions. Also note that it is relatively straightforward to incorporate load measures with multiple thresholds into the protocol, or indeed to use completely different load measures. In this chapter, we show that the simple two threshold scheme is both robust and efficient.

Replica creation detail: Load is redistributed according to a per-node capacity, l_i^{max}, and high- and low-load thresholds, l_i^{hi} and l_i^{low}, as in Figure 5.4. Each time a packet is routed through server S_i, S_i checks whether the current load, l_i, indicates that load redistribution is necessary. If necessary, load is redistributed to the source of the message, S_j. The source is chosen because it is in some sense "fair" (the source added to the local load), and because load information about the source can easily be added to all queries. Lastly, creating a replica at the source is often "cheap", since the source is likely transferring a popular object (which would be replicated).

If $l_i > l_i^{hi}$, S_i is overloaded. S_i attempts to create new replicas on S_j if l_i is greater than l_j by some fixed value. S_i then asks S_j to create replicas of the n most highly loaded items on S_i, such that the sum of the local loads due to these n items is greater than or equal to the difference in loads between the two servers.

If S_i's load is merely high, but not in an overload situation ($l_i^{low} \leq l_i \leq l_i^{hi}$), load is redistributed to S_j only if $l_i - l_j \geq l_i^{lo}$. The amount redistributed is calculated as above.

In both cases, further replication might be required to eliminate hotspots or load-imbalances completely. The load due to each replica is maintained by the node storing the replica and creates new replicas if the local load due to a replica is non-negligible.

Soft State replicas and Replica-augmented routing: We use two forms of soft

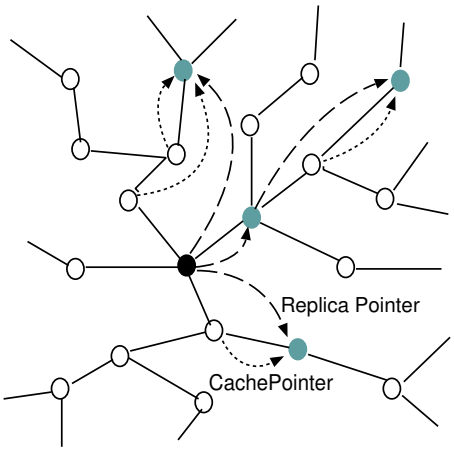


Figure 5.5: Snapshot of the P2P network with LAR

state: caches and replicas. Both operate on the granularity of a single data item. A cache entry consists of a data item label, the item's home, the home's physical address, and a set of known replica locations. Note that a cache entry does *not* contain the item's data: it is merely a routing hint that specifies where the data item can be found.

Cache entries are replaced using a least recently used (LRU) policy with an entry being touched whenever used in routing. Caches are populated by loading the path "so far" into the cache of each server encountered during the routing process. Both the source and destination cache the entire path. This form of path propagation not only brings remote items into the cache, it also brings in nearby items and a cross-section of items from different levels of the namespace tree. Our experience is that this mixture of close and far items performs significantly better than caching only the query endpoints.

Replicas differ in that (i) they contain the item data, and (ii) new replicas are advertised on the query path. When a replica is created, we install cache state on the path from the new replica to the node that created the replica. Figure 5.5 shows how LAR creates

replicas on the source of the query and adds pointers to the replica in caches along the path in the same network as Figure 5.1. Also note that a replica can further create replicas as shown in the figure. In this case, pointers are added in the path from the new replica to the original replica only. Also contrast the difference with `app-cache` protocol in Figure 5.2.

These cached entries effectively "short-cut" routing when encountered by queries. Our results show that adding the cache entries for routing significantly improves system performance and load balance regardless of input query distribution.

Replicas in our system are "soft" in the sense that they can be created and destroyed without any explicit coordination with other replicas or item homes. Hence, idle replicas consume no system resources except memory on the server that hosts them. Therefore, identifying and evicting redundant replicas is not urgent, and can be handled lazily via an LRU replacement scheme. Obviously, cache entries may point to stale replicas since there is no global coordination on when replicas are created or destroyed.

Since cache state includes information about multiple replicas, and during routing, we can choose one of these replicas uniformly at random. Obviously, cache entries can be used to distribute load among the servers that hold replicas of the data item being queried. However, they can also be used to find replicas of *next hop* nodes that are used to route queries (as in Figure 5.6). Thus, cache entries balance both data transfer load and routing load.

Replica-state Management and Dissemination: Once replicas are created, we need to disseminate information about new replica sets. Rather than introduce extra message traffic, we piggyback replica sets on existing messages containing cache entries.

Servers maintain only partial replica set information in order to bound the state required to store and transmit the information. A $2/32$ dissemination policy means that a maximum of two replicas locations are appended to cache insertion messages, while a maximum of 32 replica locations are stored, per data item, at a given server.

The merge policy determines how incoming replica locations are merged into the local store of replica locations, assuming both are fully populated. The locations to be retained are currently chosen randomly, as experiments with different preferences did not reveal any advantage.

The dissemination choice policy decides which of the locally known replica locations to append to outgoing messages. Random choice works well here as well, but we found a heuristic that slightly improves results. If a server has a local replica and has created others elsewhere, it prefers the replicas it has created elsewhere most recently. Otherwise, the choice is random. The intuition behind this heuristic is that if the existing load is high enough to cause the server to attempt shedding load, it is counter-productive to continuing advertising the server's own replicas. On the other hand, advertising newly created replicas helps to shed load.

Note that we have neglected consistency issues. However, it is highly unlikely that rapidly changing objects will be disseminated with this type of system and we have designed our protocol accordingly. We also do not address servers joining and leaving the system. These actions are handled by the underlying system P2P system and should not affect the applicability of the replication scheme.

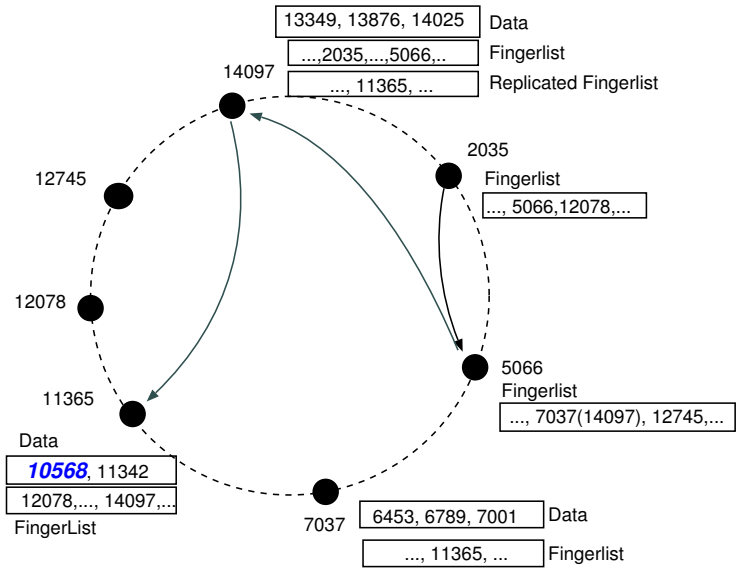


Figure 5.6: LAR routing and replication in Chord

LAR applied to Chord

When adapting LAR to Chord, the finger list is the default item of replication. We replicate the data item only if the load on the server due to the data item is more than that due to the finger list. However, when we replicate the data item, we also must replicate the finger list in order for the new replica to be seen by other servers.

When disseminating this information, we only know the query's source and the last sender. We therefore update the cache with this information rather than the full path.

Figure 5.6 shows an example using chord for a query of item ID 10568, initiated at server 2035. Recall that IDs increase in clockwise direction, and that an item is served by its first successor server. Each server is annotated with its data and fingerlist. In the example, item 10568 is served by server 11365. Server 7037's fingerlist is replicated at 14097, and this replication is known to server 5066.

Server 2035 chooses 5066 as the next hop because it has the highest ID about which 2035 knows, such that the ID is less than or equal to the item ID. Likewise, 5066 determines that 7037 is next, but randomly picks 7037's replica on 14097 instead. Finally, 14097 forwards to 11365, the final destination.

5.4.3 Summary

LAR takes a minimalist approach to replication. Servers periodically compare their load to local maximum and desired loads. High load causes a server to attempt creation of a new replica on one of the servers that caused the load (usually the sender of the last message). Since servers append load information to messages that they originate, "downstream" servers have recent information on which to base replication decisions. Information about new replicas is then spread on subsequent messages that contain requests for the same data item.

In implementation, the replication process only requires a single RPC between the loaded server and a message originator. Further, this RPC contains no data because the originator of a request has already requested it. Even this RPC can be optimized away if the loaded server is also the server that responds to the request. However, we retain it in order to allow the replication process to proceed asynchronously with respect to the lookup protocol.

5.5 Simulation Results

In this section, we present a comprehensive simulation-based evaluation of `LAR`, and compare its performance to `app-cache`. Our performance results are based on a heavily modified version of the simulator used in the Chord project, downloaded from *http://www.pdos.lcs.mit.edu/chord/*. The resulting simulator is discrete time and accommodates per-server thresholds and capacities.

5.5.1 Simulation Defaults

By default, simulations run with 1k servers, 32767 data items, and the server capacity l^{max} is set to 10 per second. We ran many experiments with higher capacities, but found no qualitative differences in the results. The load thresholds l^{hi} and l^{low} are set to 0.75 and 0.30 times l^{max}. The length of a server's queue is set to the number of locally homed items, in this case 32. For example, if an idle server with capacity $l^{max} = 10/second$ and queue length $q_{max} = 32$ receives 50 queries over one second, 8 will be dropped (10 will be processed, and 32 will be queued). The default load window size, which controls how quickly the system can adapt, is set to two seconds.Each network "hop" takes a single time unit, currently set to 25 milliseconds. The dissemination policy is set to $1/32$. By default, 500 queries are generated per second. The average query path is less than 5 hops, so these default values correspond to an average node load less than 25%.

In the simulations, query sources were selected uniformly at random, and the query inter-arrival had a Poisson distribution. The input query distributions ranged from uniform

to heavily skewed. We experimented with extremely heavy skew `90-1`, in which 90% of the input is directed to a single item (and the rest 10% uniformly distributed over all items). We also experimented with less skew in which 90% of the inputs were directed to 1% (327) or 10% (3276) items.

In terms of space at each peer, TerraDir peers each have 20 cache slots, and can accommodate replicas of 64 remote data items (twice as many as the number of items for which the server is home). Chord has no regular cache slots. Chord servers can accommodate replicas of five other servers, which gives them approximately the same amount of state as consumed by the combination of the TerraDir caching and replication schemes.

By default, each message transfer —whether it is a document, a query or a control message— contributes identically to loads and congestion. We chose this default to heavily favor `app-cache`, which creates and transfers many more document replicas. In Section 5.5.3, we show the effect of document transfers costing 2x, 4x, and 10x other control message transfers. In practice, this cost is likely to be 100 or 1000 times more, so even these values are biased positively towards `app-cache`. Lastly, we note that in the simulations, all messages, including control messages, are dropped when a server is beyond its capacity.

5.5.2 Effect of Query Distribution

In Figures 5.7, 5.8, 5.9 and 5.10, we show the effect of input distribution on `LAR`, `app-cache`, and plain Chord. In each figure, we plot the number of dropped messages

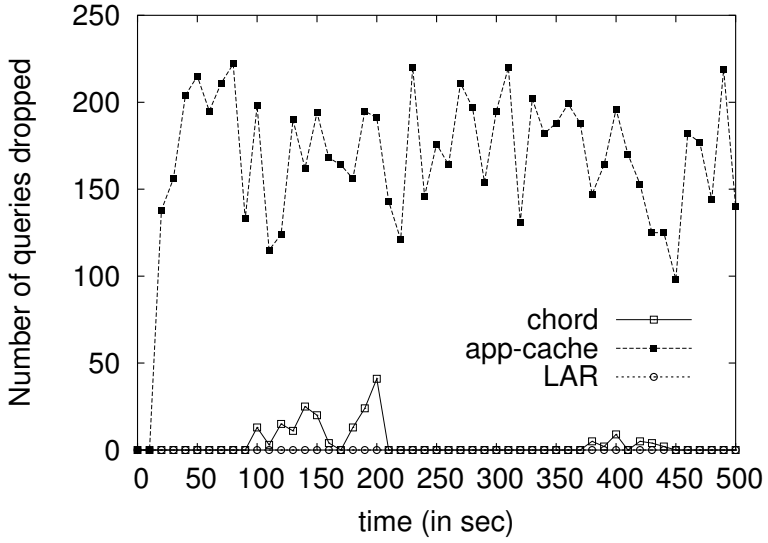


Figure 5.7: Number of drops with uniformly distributed queries

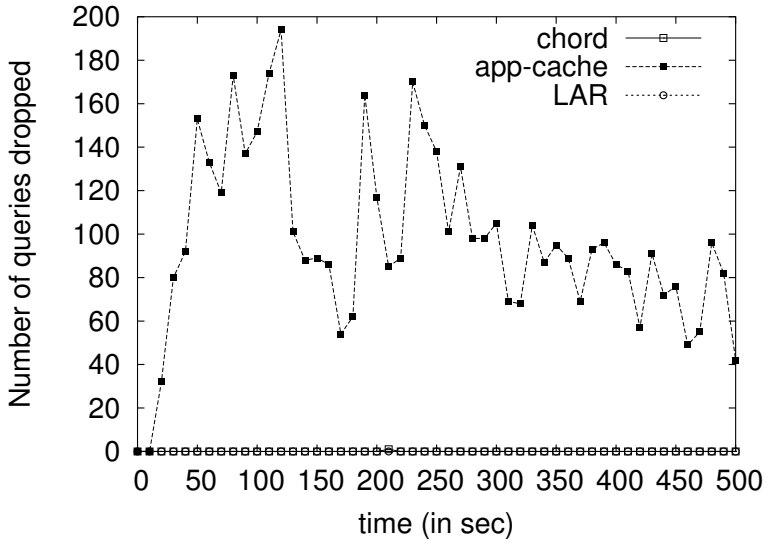


Figure 5.8: Number of drops with 90% of queries to 10% of items

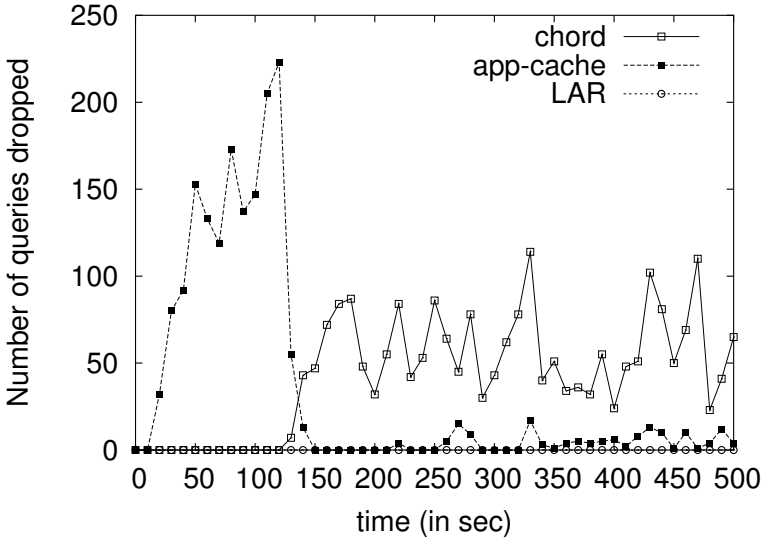


Figure 5.9: Number of drops with 90% of queries to 1% of items

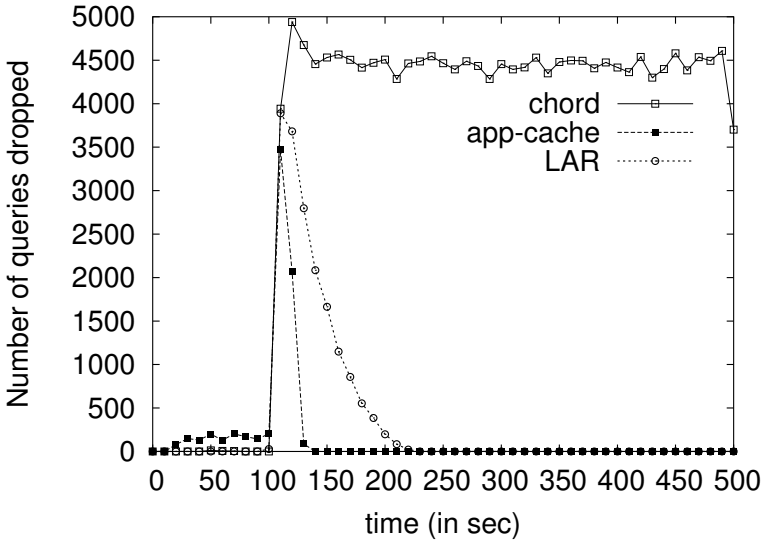


Figure 5.10: Number of drops with 90% of queries to 10% of items

over time (added over 10 seconds) for the three different schemes. For each experiment, we ran 10 trials with different random number seeds, and these results are from a single representative run. In all experiments, the first 100 seconds of input are uniform, and then the specific input distribution takes effect.

In these results, `app-cache` performs better when the query distribution is more skewed because the hot items quickly get replicated at essentially all nodes in the network. This is because there is no penalty for extra data transfer (since document transfer costs the same as control messages). Note that with even moderate average load (25%), `app-cache` drops messages with a uniform query distribution: this is because the blind replication scheme starts to thrash. As we show in later results, this causes severe problems with higher load, and when the cost of document transfer is increased.

We should note that plain Chord, while better than `app-cache` for inputs without significant skew, serves only about 10% of input queries for heavily skewed inputs (Figure 5.10). Note that the y-axis of this figure is significantly higher than the three other figures. Also notice that `app-cache` drops queries for the first 100 seconds of all the runs since the input stream is uniform. When the input shows heavy bias (starting at time=100seconds), `app-cache` *stops* dropping packets as the hot item is quickly replicated at all servers. At the onset of skew, `LAR` initially drops a number of queries, but the `LAR` adaptation reduces drops down to zero as the hot item it replicated and the routing state set up.

Protocol Overheads

In Table 5.1 we show the average overhead of `LAR` compared to `app-cache`. These results are averages of ten runs. First, note that plain Chord does not create replicas

Input dist.	Scheme	# q served (250K max)	# replicas creat.	# replicas evict.	# hints creat.	# hints evict.
Unif.	Chord	249.9K	-	-	-	-
	app-cache	242.4K	1.13M	1.09M	-	-
	LAR	249.9K	5K	0	10.8K	5.9K
90% → 10%	Chord	249.9K	-	-	-	-
	app-cache	245.7K	994K	962K	-	-
	LAR	249.9K	6.6K	0	12.5K	7.5K
90% → 1%	Chord	248.2K	-	-	-	-
	app-cache	248.1K	691K	660K	-	-
	LAR	249.9K	10.3K	0	17.3K	12.3K
90% → 1	Chord	72.1K	-	-	-	-
	app-cache	244.1K	328K	296K	-	-
	LAR	233.4K	2.6K	0	7.2K	2.4K

Table 5.1: Protocol Overhead of Chord, LAR, and app-cache.

or use routing hints, but loses significant numbers of queries when the input distribution is skewed. For skewed inputs, LAR and app-cache both serve over 93% of all queries. With 250K queries, app-cache is able to serve more queries for extremely skewed inputs (because the hot item is quickly cached everywhere in the system), but it should be clear from Figure 5.10 that LAR asymptotically approaches 100% service after the adaption takes effect. For inputs with less skew, LAR and even plain Chord outperforms app-cache.

The major difference in protocol performance is seen in the replica creation overhead: LAR creates anywhere between 1–3 orders of magnitude less replicas. For example, for uniform queries, app-cache creates over a *million* replicas and promptly deletes them! The perils of blind replication are clear, as it is relatively easy for app-cache to thrash even under moderate load. The lower number of replicas created by LAR directly translates to lower protocol overhead and lower bandwidth usage, since replica creation

Scheme	Cost of document transfer vs. control traffic			
	1x	2x	4x	10x
app-cache	4.0K	17.2K	45.5K	106K
LAR	0	0	2	25

Table 5.2: Transfer Cost: Number of queries dropped (250K queries max.)

involves transfer of the document itself. We should note that in LAR, replica creation, in the vast majority of cases, does *not* involve transferring the document since the source of the query becomes the new replica.

Lastly, the proactive state installed by LAR (the cached routing hints) are orders of magnitude smaller in size than the documents transferred by app-cache; in all cases, the number of hints placed by LAR is 1–2 orders of magnitude lower than the number of replicas created by app-cache, and is negligible compared to the number of queries served. Thus, LAR has extremely low overhead, and is stable over a wide range of input distributions.

5.5.3 Change in Transfer Costs

Table 5.2 shows the number of dropped queries when the cost of transferring a document increases. In these experiments, 90% of the queries went to 3276 items (10% of the input), and the entire experiment ran for 250K queries. Also, the server capacity l^{max} for these experiments is 20 per sec. The average load on any node in the system is approximately 25%. Since app-cache creates an order of magnitude more replicas (and hence transfers correspondingly more data), as document transfer costs increase, it drops close to 50% of the queries in the system. In contrast, LAR is essentially unaffected by

Scheme	Average load on server			
	10%	25%	33%	50%
app-cache	0	4K	15.1K	42.7K
LAR	0	0	56	5.4K

Table 5.3: Number of queries dropped (250K queries max) under different loads

these document transfer costs. As mentioned earlier, these experiments are still biased in favor of app-cache, and in practice, the document transfer cost is likely to be hundreds of times (or even thousands for large documents) more and app-cache performance will be even worse.

5.5.4 Change in Average System Load

In Table 5.3, we show how LAR and app-cache react when the average system load changes. For these experiments, 90% of the input queries went to 3276 items (10% of the original documents). The figure shows number of unanswered queries over time for average load values of 10%, 20%, 33%, and 50%. With a 90-10 input distribution, LAR is able to serve more than 97% of all queries even with 50% average load. The app-cache scheme is more sensitive to system load and loses about 20% of the input queries when the average system load is 50%.

5.5.5 Changes in data popularity.

Figure 5.11 shows how LAR reacts to changes in hotspots over 900 seconds. For these experiments, 90% of the queries went to a single item. There is a change in the hot item every 200 seconds (100K queries) and there are 4 such changes, with the first

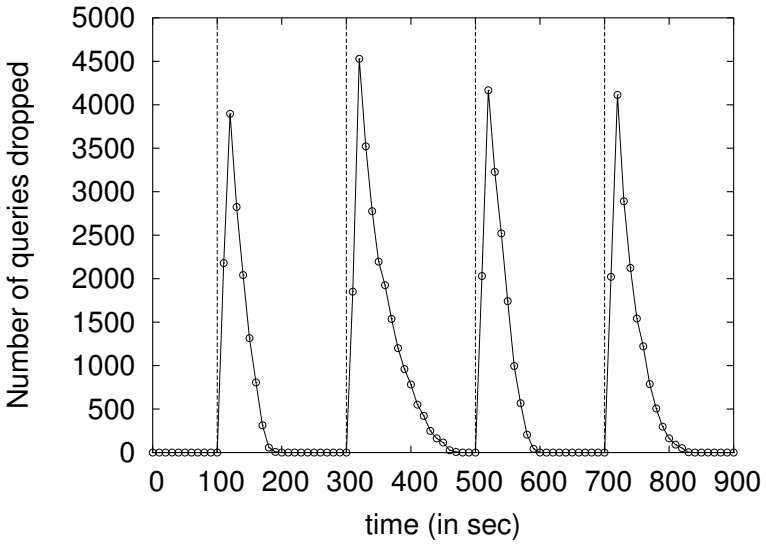


Figure 5.11: Adaptivity to changing hotspots: Number of queries dropped over 900 sec.

100 seconds having a uniform query distribution. The graph in Figure 5.11 is based on experiments over Chord; other experiments over TerraDir also yielded similar results [35].

The vertical lines show time points when the hot item changes. The drops are computed every 10 secs. As is shown in Table 5.1, this scenario is the worst possible case for LAR. From the plots, we see that LAR is able to adjust relatively quickly to these changes (on the order of 2 minutes), and in all cases the replication is adapted to the change in hot data item. Obviously, in practice, we do not expect such radical shifts in data access patterns to occur over such small intervals, but it is clear that LAR is robust against drastic changes in input distribution over very short timescales.

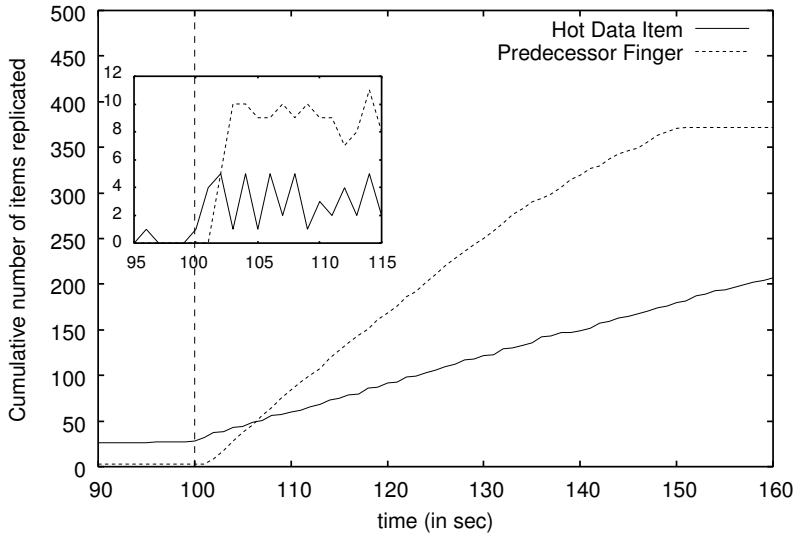


Figure 5.12: Cumulative number of replicas over time for the hot item and the predecessor's routing hints, when skewed query distribution begins.

5.5.6 Dynamics of Replication in Chord

Consider the skewed input case. Intuitively, it seems that the successor of the hot data item (which holds the data) would be the one dropping all the queries because of the deluge of queries. However, lookup requests in Chord are routed to the best locally known *predecessor* until the lookup request reaches the actual predecessor of the data item. The query is then resolved and sent to the successor of this node, which is also the successor of the data item. The implication of this is that the first bottleneck in case of skewed inputs is the predecessor of the data item. All the lookup requests need to go to the predecessor before they can by resolved to the successor. In Figure 5.12, we show the dynamics of replication of `LAR` over Chord for a skewed input (90% to 1). The plot shows the cumulative number of replicas created for the hot data item and the number of replicas

created for its predecessor between simulation time 90 and 160 secs. In the inset, we show the number of replicas created per second between the time period of 95 seconds and 115 seconds for both the hot item and its predecessor finger. Recall that the input distribution changes from uniform to skew at simulation time 100 seconds (noted by vertical line in the plot). In these plots, the replica creation numbers are computed from the simulation logs once every second, and the servers themselves recompute load once every two seconds. Both from the CDF and the inset plots, it is clear that the predecessor finger gets replicated at a quicker rate and more widely before the data item itself is replicated. The routing hints stop replicating at time 150 seconds, while the data is replicated until time 240 seconds (not shown in plot). This is a somewhat non-intuitive phenomenon, and is a direct result of the specifics of how Chord resolves its queries.

5.5.7 Scalability

Figure 5.13 shows the fraction of dropped queries with different system sizes. In these experiments, 90% of the queries went to 3276 items (10% of the input). Although the graph is plotted until 205 seconds, the experiments ran for 400 seconds, with no drops at any size after 205 seconds. Note that there are no uniformly distributed queries in the beginning. At each system size, the query stream is adjusted so that the average load on any server is approximately 25%. Since the number of data items are the same for all system sizes and the query rate increases with increase in system size (to maintain 25% load), but the individual server capacities remain the same, the skew in input is heavier for larger system sizes. Thus, larger system sizes drop correspondingly more queries while

the adaptation takes effect, but in all cases, LAR is able to control the skew and eventually create sufficient replicas and reduce drops to zero within about two minutes of simulation time.

5.5.8 Load Balancing

LAR has two primary goals: handling overloads and attempting to keep server loads under the low-water threshold, l^{low}, if possible. Figure 5.14 shows the average of the ten highest server loads for a run with adaptive replication and a skewed distribution on TerraDir. Each line represents a different l^{low} value; in all cases l^{hi} remains at 0.75.

In all but the two extreme cases, the averages quickly drop below the low-water threshold. They do not drop further because servers whose loads are less than l^{low} do not attempt to shed load. The slope of the $l^{low} = 0.15$ and $l^{low} = 0.00$ lines flattens considerably as they near 0.10, the mean load in the system. The overall load in the system varies from 10% with l^{low} at 0.75, to 5.5% with a l^{low} value of 0. The variation is due to increased queuing times when loads are highly loaded.

5.5.9 Parameter Sensitivity Study

Load Window Size - Figure 5.15 shows plots of message drops versus time for TerraDir with adaptive replication and different load window sizes. Smaller windows allow the system to react more quickly, adapting better to swiftly changing conditions at the cost of more replica creations and evictions.

The cost of this adaptivity is shown in Figure 5.16. One possible conclusion is that

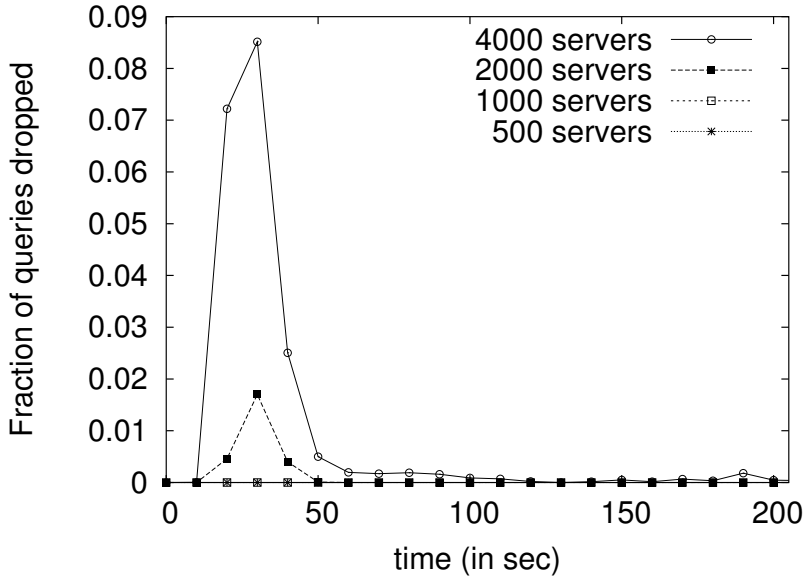


Figure 5.13: Scalability: fraction of queries dropped for various system sizes.

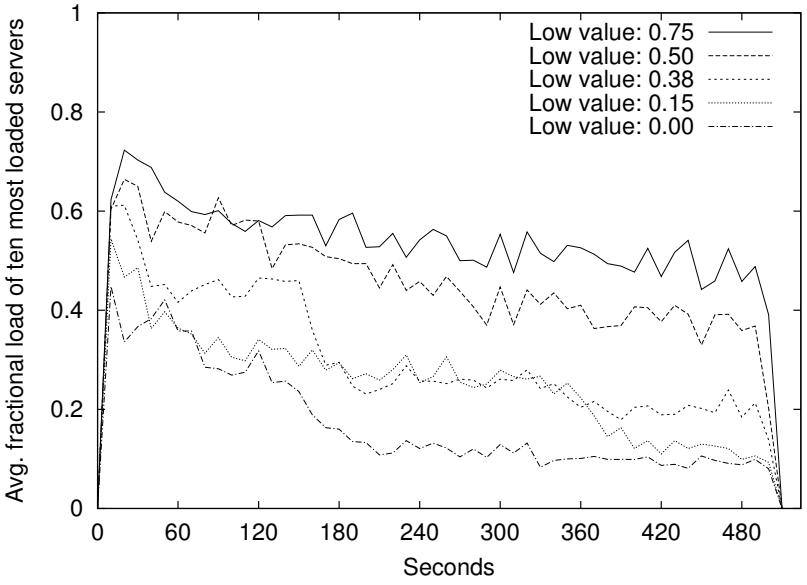


Figure 5.14: Average of 10 most loaded servers for TerraDir with skewed input and varying l^{low} values.

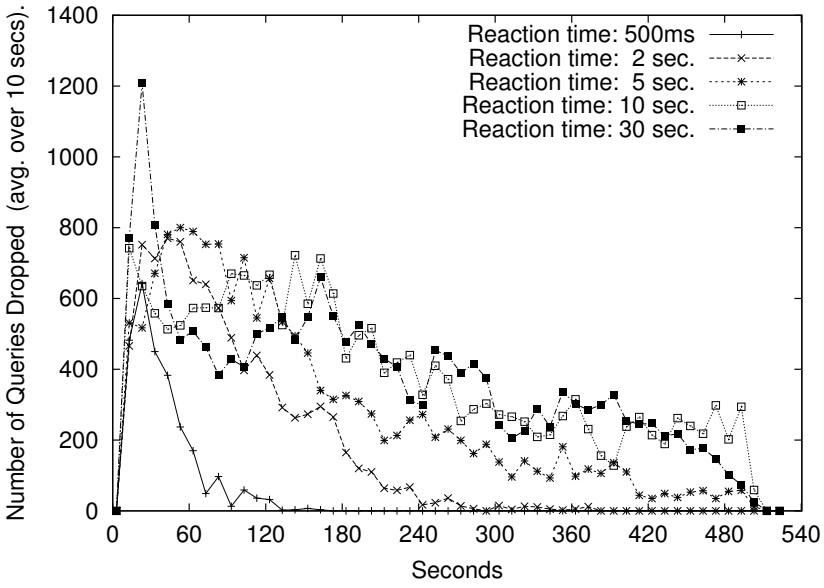


Figure 5.15: Effect of load window sizes: Number of queries dropped as a function of time.

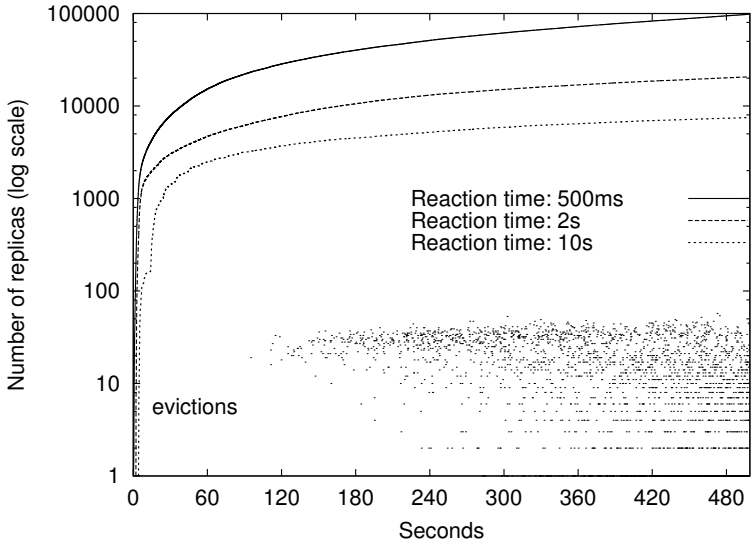


Figure 5.16: Cumulative replica creations versus time for a `90-1` distribution under TerraDir, for three different load window sizes. Dots indicate eviction events for a window size of 500 msecs. Larger windows sizes do not cause evictions. Note the log scale.

Policy	Drops	Replicas	Evictions.
0/1	162172	4803	0
1/1	19595	2634	0
1/8	19275	2582	0
2/8	16942	2493	0
2/32	17759	2462	0
16/32	17407	2493	0
32/32	17407	2503	0
64/128	17648	2447	0

Table 5.4: Drops and replica events versus dissemination policy.

the system is relatively insensitive to the size of the load window within a broad range (10-30 seconds). The use of smaller windows can dramatically improve drop rates, but only at the cost of increased protocol traffic.

Dissemination Constants - Table 5.4 shows the effect of the dissemination policy on drops and replica events for TerraDir and skewed input. Recall that `x/y` means that x replica locations are appended to outgoing messages about a given item and y are stored locally. Skewed input heavily overloads a single server and places a premium on quickly spreading knowledge of new replicas.

Nonetheless, the results show an almost complete lack of sensitivity to these parameters. Though we use `2/32` for the other experiments, the results show that keeping and propagating only a single other storage location is almost as effective.

5.5.10 Static versus Adaptive Replication

This section contrasts the performance of static versus adaptive replication for both Chord and TerraDir. A static distribution for Chord is easily created by replicating all servers evenly.

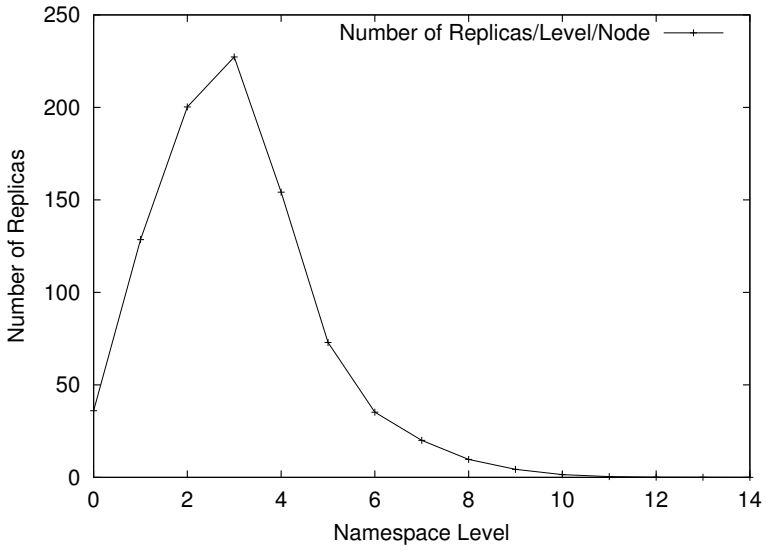


Figure 5.17: Average number of replicas created for items at a given level of the TerraDir tree. Level 0 is the root.

For TerraDir, calculating the proper static distribution of replicas is non-trivial. Without caches, and assuming that the tree is balanced and that the query distribution is uniform, the load on each item in the tree can easily be calculated analytically. However, caches change the load distribution considerably. Figure 5.17 shows the number of times each item is replicated in our adaptive scheme, a measure highly correlated with load. We use this result as the "static" replica distribution in the rest of this section.

The explanation for level three having the highest load is as follows. Assume that a cache is initially populated uniformly. Items from high in the tree (level 0 etc.) are quickly evicted, as cached elements slightly lower in the tree are closer to destinations and all fit in the cache. Items from low in the tree are also evicted. They are surely closer to some destinations, but are only touched by a small proportion of the queries. Hence, the caches become populated with items in a middle ground where the whole level fits

into the cache, yet each item on the level is used frequently.

Caches populated in this way cause a great deal of load on these middle levels, as cache entries only contain mappings from an item name to a server's address. They do not contain the data and so can not satisfy queries locally.

Given this distribution, Figures 5.18 and 5.19 show message losses versus time for all combinations of static or adaptive replication, and uniform or skewed input. The figures show that adaptivity is crucial in order to handle skewed distributions. Both Chord and TerraDir lose on the order of 90% of messages with static replication and skewed input, but stop losing messages within two and one half minutes when using adaptive replication.

Additionally, the graphs highlight the need of the hierarchical system to populate the system with replicas for items high in the graph. The adaptive experiments start without any replicas in the system, so the "unif. with adapt. rep" line in Figure 5.19 is showing the process of populating the system with the "best" static distribution discussed above. By contrast, Chord does not need any replication to handle uniform query distributions.

5.6 Summary

This chapter described LAR, a new soft-state replication scheme for peer-to-peer networks. LAR is a replication framework which can be used in conjunction with almost any distributed data access scheme. In this chapter, we have applied LAR to both a distributed hash-table algorithm (Chord) and a distributed directory (TerraDir).

Compared to previous work, LAR has an order of magnitude lower overhead, and at

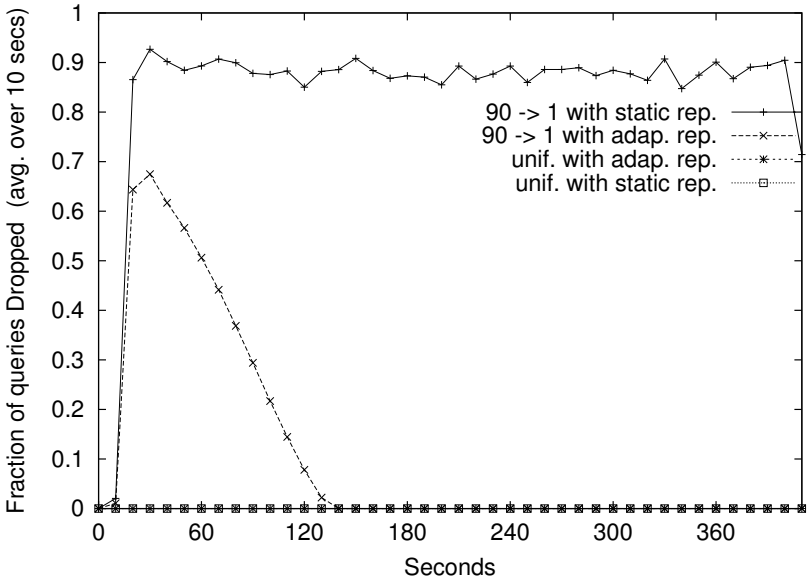


Figure 5.18: Message drops versus time with Chord for all combinations of uniform/skewed input and static/adaptive replication.

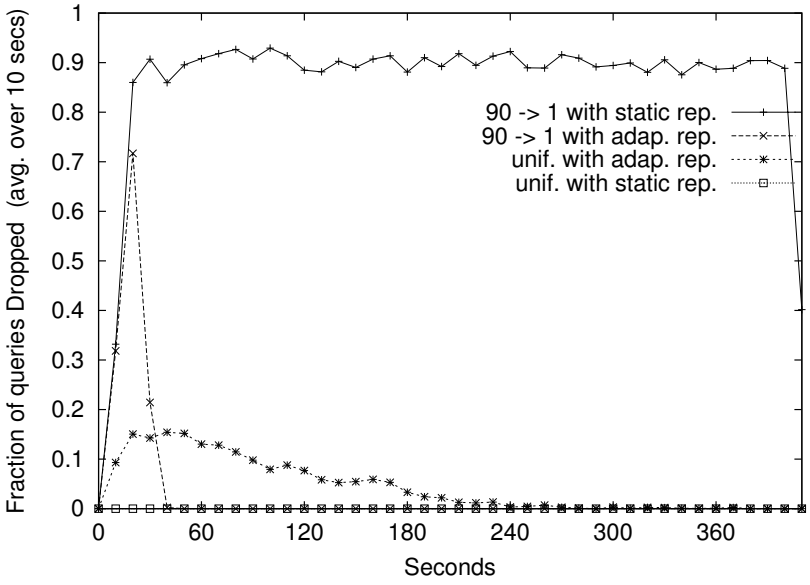


Figure 5.19: Message drops versus time with TerraDir for all combinations of uniform/skewed input and static/adaptive replication.

least comparable performance. More importantly, LAR is adaptive: it can efficiently track changes in the query stream and autonomously organize system resources to best meet current demands.

We have demonstrated the efficacy of LAR using a number of different experiments, all conducted over a detailed packet-level simulation framework. In our experiments, we show that LAR can adapt to several orders of magnitude changes in demand over a few minutes, and can be configured to balance the load of peer-servers within configurable bounds.

Chapter 6

Reliable Storage

In the chapters so far, we have looked at how to reduce the amount of data transferred over the network and how to distribute the load incurred by participating nodes. In this chapter, we examine an important, yet extremely challenging and perhaps the least-addressed of requirements: storing the indexes reliably.

6.1 Introduction

Inverted indexes provide a simple, yet powerful, approach to facilitate search over P2P systems. However, implementing a system in practice gets challenging. P2P systems have been designed for use with data sets running into billions, and with hundreds of thousands of keywords. Storing and maintaining indexes for these data sets in a distributed manner will require tens (or hundreds) of thousands of hosts. Fortunately, existing structured P2P systems are particularly efficient at organizing data over a large number of hosts. Further, structured systems have mechanisms to seamlessly scale just by adding new hosts as the corpus increases. Using inverted indexes, however, has two debilitating problems that arise from a combination of factors including the kind of data that is indexed and the properties of the environment in which these systems are deployed. The first problem deals with the storage of large indexes. The second problem deals with the dynamic nature of these systems and the repercussions thereof.

The storage problem is obvious: P2P systems consist of a number of regular hosts. Each of the participating host donates some disk space to the system to store the meta-data (including indexes) that are required for the system to function. The typical approach used to store indexes and other meta-data is to map these objects randomly to one of the participating nodes. It can easily be shown (using the standard [randomized?] balls-and-bins argument) that the load of nodes, in terms of the amount of data stored, will be balanced within a certain bound. This, of course, assumes that the amount of space available at each node and the size of each object is the same (similar?).

The reality, however, is very different. The distribution of words in documents is not uniform; in fact, the distribution is very skewed and usually follows what is known as the Zipf distribution (named after George Kingsley Zipf [91], a linguist who gave proof of this property). The implication of this observation is that indexes of popular keywords can grow very large, which, in turn, could result in the nodes responsible for the indexes to run out of space to store these popular indexes and/or other indexes that are also mapped to that node. Worse yet, it is possible that some of these indexes cannot be stored at any single node in the system. The problem is compounded further in many of these systems because there is no control over which index(es) gets mapped to which host. It is quite likely that a particular node is responsible for storing a large set of indexes, thereby running out of space to store index entries. It is also quite likely that the hosts holding the indexes for popular terms will be under-provisioned and cannot hold the entire index. Neither of these situations is favorable; we need complete indexes to answer queries and to identify relevant results while ranking.

The second problem stems from the dynamic nature of Peer-to-Peer systems. Since

these systems comprise primarily of end-hosts, there is a lot of churn with nodes arriving and departing frequently. In fact, studies by Bhagwan et al. on the Overnet P2P system [6] shows that, on average, less than 50% of the peers were present in the system for more than 5 hours. In such a dynamic systems, indexes may be lost when the node storing these indexes either fails or departs. Since the presence of inverted indexes are essential to answering queries, storing indexes reliably despite the dynamics in the system is key to the practical deployment of the system.

The standard approach used to address this problem has been to statically replicate on K successive neighbors. The actual value of K depends on the perceived failure rate in the system. This approach is particularly useful in DHT-based settings because when a node n departs, its neighbor takes over the address space that n was responsible for. This approach, while useful under low failure rates, requires a lot of replicas in order to guarantee resilience under high failure rates. However, space is a premium and creating a lot of replicas may, at times, be infeasible.

The problem of reliable storage is the least-addressed, while perhaps being the most important, among the problems common to P2P systems. The size of a given index potentially scales with the number of documents in the system. Such an index may be too large to fit on any single node in the system. In this chapter, we try to address the issue using horizontal partitioning of inverted indexes. We present a greedy online algorithm that identifies indexes that are very big and partitions them according to the prefixes of the entries. The largest set of entries with the same common prefix is then moved to a randomly chosen node with sufficient space. This algorithm is also useful in the context of View Trees, where we cache the results of queries. As the results in 6.5 show, we are

able to answer 45% more queries accurately with employing the algorithm.

To order to address the problem of reliable storage, we look at alternate approaches to provide reliability. In particular, we look at erasure codes, which are a kind of error-correcting codes, and analyze the trade-offs of employing erasure codes for resiliency. We derive analytical bounds on the amount of space needed for static replication and erasure codes, for guaranteeing the same levels of reliability in the presence of partitioning. We also compare the performance of replication and erasure coding in terms of the amount of space needed for various levels of failures and derive the cost updating indexes in each case. Based on our results, we suggest a hybrid scheme for providing reliable storage of indexes.

The reminder of this chapter is organized as follows: first, we discuss some background about partitioning in Section 6.2. We then present the requirements of a good partitioning algorithm and then describe our algorithm in section 6.3. In Section 6.4, we then look at the efficacy of using replication and erasure codes and derive analytical bounds that relate the failure rate, level of redundancy and the probability of answering queries. We then present some of the analytical and experimental results in section 6.5. Finally, we summarize this chapter in Section 6.6.

6.2 Data Partitioning: Background

The problem of partitioning data, in order to make optimal use of available disk space, is common not just P2P systems, but also file systems, databases, etc. Among the existing body of work, ideas proposed in the field of distributed databases is most sim-

ilar to and applicable in our setting. Depending on the approach used to partition and distribute the data, the partitioning schemes are classified either as horizontal or vertical partitioning schemes. In horizontal partitioning, a complete record is stored at one location, but different records belonging to the same database might be stored at different locations. In vertical partitioning, data is split based on the fields of the database. All the records of a particular field are stored together, and a single record is stored at multiple locations. An astute observer will realize that the indexing mechanism employed in this dissertation, where an index of a keyword is stored at at a single location, with the entire set of indexes distributed over the participating peers, is an example of vertical partitioning. In the rest of this chapter we will focus on horizontal partitioning since it is most relevant to our end goal.

6.2.1 Horizontal Partitioning

In horizontal partitioning [15], different records (or tuples) of an index are stored at different sites. All the fields of the index for that tuple, however, are stored at the same location. In general, there are two properties that horizontal partitioning techniques need to satisfy:

- Completeness and re-constructibility
- Disjointness

The first property is probably the more important property and guarantees correctness. It guarantees that when the index is reconstructed using the partitions, then the final result will be same as the original relation. For example, consider a index R split such that

$R = R_1, R_2, R_3, \ldots, R_k$. Then by the first property,

$$R = R_1 \cup R_2 \cup R_3 \cup \ldots \cup R_k \tag{6.1}$$

The second property states that split will not have any redundancy and that the sum of the number of tuples in all the partitions will be the same as the original index. To put it more rigorously, given an index R as above

$$R_i \bigcap R_j = \emptyset for\ any\ i \neq j \tag{6.2}$$

It is important to note that unlike the first property, this property is *not required* for correctness and only affects the efficiency of the space occupied.

The problem of data partitioning in P2P systems has been studied previously by Ganesan et al. [27] and in eSearch [82]. The former present an online algorithm to distribute range-partitioned data in a manner such that the ratio of the smallest partition to that of the largest partition is guaranteed to be bounded. The main aim of their technique is to balance existing partitions and prevent skew in range partitioned data. The algorithm proceeds in two steps: first they try to balance the size of neighboring partitions. If that is not enough, they use a re-adjustment step where the location with the smallest size moves its data to its neighbor and accepts half the data from the location with largest data size. The authors show that the protocol can provide good guarantees for amortized costs when there are a lot on insertions and deletions. Despite the fact that the main aim of the work is different, the idea could be applied to P2P systems to split the indexes. However, the protocol has two main issues when it comes to P2P systems: (a) The node join protocol in P2P systems must be modified, and (b) Global information about the load of the neighbors and the most-loaded and least-loaded node is required.

Tang et.al.[82] present a distributed approximation of the algorithm presented by Ganesan et al. Tang et al. build on the existing approach of hashing the keyword to map it into the P2P namespace. However, once they generate the identifier, they randomize the 20 least-significant bits thereby making different records of the same keyword to get mapped to 2^{20} locations. It is easy to see that this approach alone is not sufficient to distribute indexes. In the original Chord design, the expected number of nodes in a range of 2^{20} is less than one. Hence it is quite possible that indexes might not get partitioned at all. To overcome this, Tang et al. modify Chord's join protocol such that the distribution of nodes to identifiers is not uniformly at random. The downside to this approach is that it relies on the particulars of Chord to partition the data. Additionally, it requires modification to the underlying protocol.

6.3 Partitioning Algorithm

In this section we present our simple partitioning scheme to adaptively distribute large indexes over multiple nodes. Our algorithm is motivated by the following guidelines:

1. **Efficient Intersections:** The indexes in P2P systems are not only used to answer queries of single keywords, but also to answer queries having more than one keywords. In order to answer such queries, the indexes of each of the keywords have to be intersected. Intersections of sets of object identifiers (required for answering conjunctive queries) should still be efficient. The splitting algorithm should, ideally, not introduce extra overhead compared to the situation when the entire index

was available at a single location.

2. **Low overhead for updates:** Given the dynamic nature of P2P systems, there will constantly be updates to the system. Whenever nodes arrive or depart, there will be updates to the keywords that are exported. The overhead of updating indexes in response to insertion of new objects and other changes should be small.

3. **Non-intrusive with the underlying protocol:** Search functionality, although important, is being added on top of existing protocols. It is therefore important that the search protocols use the existing protocols to work well rather than expect to modify the protocols. Splitting of indexes should be no different and should not require changes to the lookup protocol to give guarantees.

4. **Independent of the underlying protocol:** The partitioning scheme should not depend strongly on the underlying P2P object-location architecture. It should work transparently with the protocol and must be able to run on most protocols using minimal changes.

6.3.1 Algorithm Description

We assume that each participating node allocates some amount of disk space for the indexes. When a node n joins the system, it exports some data items along with their keywords. The keywords of each data item are hashed to generate an identifier. The the node responsible for that identifier stores the index of that keyword. Hence, as part of the join process, this node is located and an entry is added to the index. When the node departs, it deletes these entries from the indexes. The invariant we try to maintain is that

the amount of data stored in each node is at most the amount of space allocated by the node for the searching. We also assume that in the temporary period between an addition and the partitioning, the invariant may be violated, but it holds after the partitioning is done.

The algorithm has two phases: the *partitioning phase* and the *merging phase*. The partitioning phase, which is probably the more important of the two phases, runs each time an entry is added to the index. Whenever the node adds an index entry, it checks to see if the amount of space allocated it filled up. In the event that it is running out of space, the node identifies the index with the largest set of entries. Recall that document identifiers in DHT are typically bit-strings that are $160-$bits long. In the case of hierarchical system, these entries are not bit-strings; hence $160-$bit hashes are generated for each of the entries. The algorithm then looks at $b-$bit prefixes and identifies the prefix set with the largest number of entries. It then randomly selects a node in the system with sufficient disk space and moves all the entries with this $b-$bit prefix to the chosen node. This partitioning step proceeds until the space invariant is satisfied.

Our partitioning scheme is summarized by the pseudo-code in Figure 6.1. When an object d appears in the network (perhaps as a result of a node joining the network), the remote procedure *addObjectToPartition* is invoked for d and the attribute indexes t_d for each of d's attributes. The procedure is invoked at the node known to be the root of the index t_d as identified by the mechanisms of the underlying P2P framework (for example, by looking up t_d in a DHT).

We assume that each participating node allocates some fixed amount of disk space, say S, for storing indexes. Partitioning is invoked when the storage exceeds this threshold

```
1: procedure addObjectToPartition(d, t_d)
2: t_d ← t_d ∪ {d}
3: while Σ_{t∈T}|t| > S do
4:     t ← largest local index
5:     t.x = most popular b-bit prefix of t
6:     m ← a peer with sufficient space
7:     m.T ← m.T ∪ t.x
8: end while
```

Figure 6.1: Index-splitting procedure. S is a per-node space limit, T is the set of all indexes at a node, and b is a system parameter that dictates how large a portion of the name space is to be migrated.

(line 4). In this case, the largest index (t, line 5) is chosen for partitioning. The index is partitioned by dividing its object identifiers into equivalence classes based on common b-bit prefixes, where b is a parameter. The largest such partition ($t.x$, line 6) is migrated to another peer. This peer is chosen by randomly, by sampling, until one with sufficient space is found. Also note that when partitions needs to split further, the algorithm pick the next $b-$bits to decide which part to partition. thereby building a $b - ary$ tree, similar to a trie on $b-$bits.

Figure 6.2 shows a tree created by splitting the index for `Red` held at the top-most node N. Since the disk space allocated on that node is full, the node splits the data in the index based on the prefix. In this example we assume that $b = 4$, i.e. we use $4 - bit$ prefixes. When the node wants to split for the first time, it identifies the prefix with the maximum number of entries. Let us assume that this was `0x1`$*$. It now moves all the entries whose first 4-bits have the value 1 to the random node. In the next iteration, when

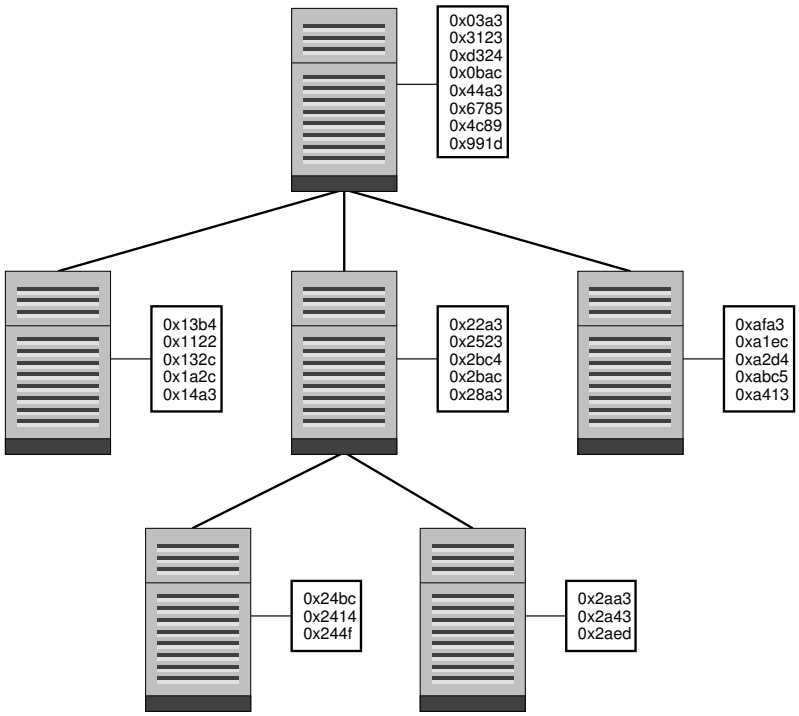


Figure 6.2: An example partition tree for the keyword `Red`.

it has to move, we assume that it moves all entries with prefix 2. Finally, it moves all entries with prefix a. Note that the node holding entries with prefix 2 splits further. This time, it picks the next 4 bits and identifies the prefix with the largest sent of entries. It then goes on to create indexes with prefixes `0x24`$*$ and `0x2a`$*$ two different nodes.

When a node tries to insert a document with attribute `Red`, it send a message to N. N now checks the prefix and decides if the message has to be forwarded to one of the children. If the prefix has not been split, then the entry is added locally. Otherwise, the message is forwarded to the node holding the partition with that prefix. For e.g., let us assume that a document with ID `0x2a31` is being added to the system. In this case, N will forward the message to the node responsible for the prefix 2. This node then

realizes that the prefix `0x2a*` has been split and again forwards the message to the node responsible for this partition.

The *merge phase*, as the name suggests, merges two previously partitioned indexes if possible. Nodes checks to see if the leaf partitions can be merged back with their parents. In the event that the parent has sufficient space to store the child, the node storing the child partition transfers the data back to the parent for merging. It is important to note that the merge phase is not mandatory; however, it is useful in preventing excessive fragmentation. Further, the merge phase is non-intrusive. It can run periodically and in the background.

Answering query proceeds in two phases: First the set of indexes needed to answer the query are identified. Once this is done, the set of partitions are identified and the tuples with appropriate prefixes are sent to the partitions. The results are then accumulated to get the final result.

6.4 Failure Recovery

If the index for an attribute were to become unavailable due to node or network failures, the performance of queries containing the attribute would suffer. In the worst case (e.g., a query that searches only on the attribute whose index is unavailable), it would be necessary to flood the network to generate complete results. Further, since a single large index can span multiple nodes due to partitioning, the probability that a large index is available depends on *all* of its host nodes being available. Thus, indexes (especially those that require flooding to recreate) must be redundantly stored. The level of redundancy de-

pends on the probability of simultaneous node failures. Once the indexes are sufficiently redundant, we can use standard techniques, such as those based on heartbeats [14], to maintain the level of redundancy. Note that the locations of the current set of redundant copies for a index, including all partitions, must be consistently maintained.

6.4.1 Resilience Through Replication

The most obvious (often optimal) and often used approach is to keep a literal copy of the index data at multiple locations. A common way of determining these locations in DHTs is to replicate the index at K-successive neighbors. This method works nicely under many circumstances and has very low update overheads. Whenever a node departs the system, its neighbor takes over the region it was responsible for. By pro-actively replicating in the neighbor, we get rid of the need to create a replica when the node departs. Updates to replicas are also simple; we just need to send a "diff" of the update to all the replicas. However, creating replicas becomes expensive when the failure rate increases. With static replication, a lot of replicas need to be created when the failure rate in the system increases. This places a lot of strain on the amount of available disk space which is already a premium resource. To show this more rigorously, we derive the relation between the failure rate, the number of replicas and the probability of answering a query in Theorem 6.1.

Theorem 6.1 *Consider a system with N nodes, where nodes fail independently with probability f, and where each index is partitioned into m partitions. If each of these partitions is statically replicated R times, then the probability of not answering a query*

is given by

$$1 - \left(1 - (f)^R\right)^m \tag{6.3}$$

Proof Since the mapping of partitions to nodes is uniformly at random and independent, the probability that a particular partition does not exist is f. When the partition is replicated R times, the query cannot be answered only when none of the replicas exist. This of course happens with probability f^R. Therefore, the probability that a replica of a particular partition exists in the system is $1 - f^R$. Extending this, the probability that all the partitions are available, is given by $\left(1 - f^R\right)^m$. When an index is partitioned into m blocks, the query cannot be answered when all the replicas of at least one of the blocks is lost. This happens with probability $1 - \left(1 - f^R\right)^m$. ∎

6.4.2 Resilience Through Erasure Codes

Since replication has high space requirements when there are high failure rates, we consider alternate redundancy methods that provide same levels of resiliency while requiring much lesser amounts of disk space. Fortunately, erasure codes, which are a kind of error-correcting codes, provide with this ability. Recall that error-correcting codes encode the data being transferred in a redundant manner such that it is possible to detect if the data has been corrupted. Erasure codes are extensions to error-correcting codes where it is also possible to recover the original data.

Erasure codes work by encoding the data, consisting of k blocks, into n redundant blocks. The key property of these codes is that *any* k blocks from this set of n blocks is sufficient to re-construct the original data. The amount of redundancy introduced n/k is

called the *rate* of encoding; higher the rate, greater the redundancy. Popular erasure code algorithms include Reed-Solomon codes [69] and Tornado codes [55]. Reed-Solomon codes make use of high-degree polynomial functions to encode and decode the data. They are commonly used in hard disks to store the data in a single hard drive redundantly. The advantage that Reed-Solomon codes have is that they require exactly same number of blocks as the original data to re-construct the original data. However, implementing Reed-Solomon codes is quite expensive in terms of computation. Tornado codes trade-off computational complexity by providing high probability guarantees. Briefly, Tornado codes work by XOR-ing different blocks to generate encoded blocks. These encoded blocks can be further XOR-ed to create newer blocks. This process is repeated until sufficient blocks are created. In order to re-create the original file, encoded blocks are XOR-ed with each other.

Erasure codes are interesting because they require much lesser disk space compared to replication in order to guarantee the same levels of resilience. We derive the relation between failures, rate of encoding and the probability of answering a query with erasure codes in Theorem 6.2. Erasure codes, however, are not an answer to all situations. Erasure codes are expensive to update; it is not possible to just send the single update to all locations. Instead, with Erasure codes, the entire set of encoded blocks need to be re-generated for each update.

Theorem 6.2 *Consider a system with N nodes, where nodes fail independently with probability f, and where each index is partitioned into m partitions. If each of these partitions consists of k blocks, and is encoded using erasure codes into n blocks, then the*

probability of not answering a query is given by

$$1 - \left[1 - e^{-n \cdot (1-f)}\right]^m \tag{6.4}$$

Proof Consider the case when we have erasure codes. If the index is split into m partitions and each partition is encoded into n fragments of which k are sufficient for regenerating the block. In this case,

$$\begin{aligned} Pr[\text{not answering a query}] &= Pr[\text{some block(s) is(are) not available}] \\ &= 1 - Pr[\text{all blocks are available}] \\ Pr[\text{a block is available}] &= Pr[\text{more than k fragments are available}] \end{aligned}$$

$$\text{let } X_i = \begin{cases} 0 & \text{if the } i^{th} \text{ fragment is lost} \\ 1 & \text{if the } i^{th} \text{ fragment is available} \end{cases}$$

Let $f = F/N$ be the probability of failure.

Let X be the random variable denoting the number of fragments that are available. Then

$$X = X_1 + X_2 + X_3 + \ldots + X_n$$

We want $P(X \geq k)$.

$$\begin{aligned} P(X \geq k) &= 1 - P(X < k) \\ P(X < k) &= P(X = 0) + P(X = 1) + \ldots + P(X = k-1) \\ &= f^n + \tbinom{n}{1} \cdot (f)^{n-1} \cdot (1-f) + \ldots + \tbinom{n}{k-1} \cdot (f)^{n-k+1} \cdot (1-f)^{k-1} \end{aligned}$$

Since X is the sum of independent indicator random variable, we can apply Chernoff bounds to this equation. By Chernoff bounds,

$$P(X < k) < \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{\mu} \text{ where } \mu \text{ is the mean and } k = \mu(1-\delta)$$

$$\therefore \mu = n \cdot (1 - f) \text{ and } \delta = 1 - \frac{k}{n \cdot (1 - f)}$$

Since $\delta \rightarrow 1$ for most of the useful values of k and n, we can approximate the above to

$$\begin{aligned} P(X < k) &= e^{-n \cdot (1-f)} \\ \Rightarrow P(X \geq k) &\geq 1 - e^{-n \cdot (1-f)} \\ \therefore Pr[\text{all blocks are available}] &\geq \left[1 - e^{-n \cdot (1-f)}\right]^m \\ \therefore Pr[\text{not answering a query}] &< 1 - \left[1 - e^{-n \cdot (1-f)}\right]^m \end{aligned}$$

∎

6.4.3 Discussion

Theorems 6.1 and 6.2 derive the probability of answering queries given a failure probability and the level of redundancy. In order to compare the space needed by both these techniques, we just need to invert the equation. If the nodes in the system fail with independent probability f, and we want to recreate an index (split into m parts) with probability $1 - p$, with simple replication, we would require $R = \frac{\ln(1-(1-p)^{1/m})}{\ln(f)}$ copies for each partition, which leads to $m \cdot \lceil R \rceil$ copies in total for the entire index. Instead, if we assume that using erasure codes, any k surviving nodes can recreate a partition, then for each partition, we only require $\frac{-\ln(1-(1-p)^{1/m})}{(1-f)}$ nodes for the same probability of recovery. For $m = 32$, $f = .1$, $k = 8$, and $p = 10^{-4}$, taking rounding into account, this analysis implies replication requires 3.2 times more space than erasure coding.

While coding reduces the amount of storage required, it increases the cost of processing updates, because when an index is updated, we have to reassemble the entire

index (potentially visiting a number of nodes), apply the update, and create new encoded blocks. This procedure can be made more efficient by encoding fixed ranges of the index into independently coded blocks such that only the affected ranges have to be reassembled upon updates. Finally, the cost of updates to encoded indexes can also be amortized by applying updates in batch. We explore the space-update overhead of encoded index storage in Section 6.5.7.

In practice, we envision a hybrid scheme in which we maintain one complete copy of the index (possibly partitioned among different nodes) and generate erasure-coded blocks to guarantee resiliency. Such a design (modulo the partitioning of the large indexes) has also been proposed by Rodrigues [71]. In this strategy, when a node holding a partition fails or departs, the partition is re-created using appropriate erasure-coded blocks. This strategy also helps us create updates efficiently. All updates are applied to this single copy of the index, and new sets of erasure coded blocks can generated periodically.

6.5 Results

In this section we will present some of the results obtained by simulating the split algorithm. We will first describe our simulation setup. We will then explain how the inputs to the simulation were generated. Finally, we quantify the cost and the benefit of the partitioning, replication, and view caching mechanisms. For index partitioning, we show that our scheme performs better than even a centralized best-case implementation of the remapping algorithm introduced in [27].

6.5.1 Experimental setup

Nominally, we ran each experiment with 500,000 queries; this number was sufficient in all experiments for the partitioning, replication, and caching behavior to stabilize. For each experiment, we use a *working set*, which is a set of unique queries to which some fraction of the overall queries are directed. We used a working set of size 50,000 to which 50%, 90%, or 99% of the queries were directed.

The base system for our experiments consisted of 10,000 servers, exporting 500,000 documents items, with approximately 15 attributes per data item. The query inter-arrival time was exponentially distributed with an average of 10 milliseconds. The upper (l_{hi}) and lower (l_{low}) thresholds , depicted in figure 5.4, for the load-based replication are set to 0.75 and 0.3 respectively, while the capacity of each server is assumed to be 10 queries/second. (We realize that this value is artificially low, but we have scaled it down in the simulator to reduce the running time of the experiments.)

Input data: The data for the experiments were generated in two steps. Firstly, we generated mappings between documents and keywords. In order to accomplish this, we used the TREC Web-10G dataset. Specifically, we extracted the KEYWORD meta-tag present in the files. The KEYWORD meta-tag is used in HTML to indicate the keywords associated with the document.

Then we needed to generate the queries based on these extracted keywords. We generated the queries based on the patterns observed in logs from search engines. We used real life traces from `search.com` and `ircache.net` to get various statistics such as the distribution of keywords per query, the distribution of individual keywords,

Index Type	% of queries answered	% of indexes affected
Attribute Indexes	47.43	17.06
Aggregate Indexes	47.38	17.27

Table 6.1: Accuracy of results without Index splitting.

etc. We then generated queries that followed these distributions.

6.5.2 Importance of Index Partitioning

We have argued that, in systems with many documents, indexes have to be partitioned onto multiple nodes. Recall that in our setting, we have 500,000 documents and 15 keywords per document. This leads to an average attribute index size of around 207 tuples, and we allow each node to store 1024 tuples. To motivate the necessity for splitting indexes, we ran an experiment (Table 6.1) in which nodes do not store indexes for which they do not have space. For the smaller attribute indexes, about half the queries cannot not be correctly answered (even though only about 17% of the indexes need to be split). Aggregate indexes, such as those used in eSearch [82], also show essentially the same performance; however, in this case, the allocated space is 10 times more (10240 tuples allowed at each node) since the indexes are, on average, 10 times larger.

6.5.3 Index Partitioning vs. eSearch Re-mapping

In Section 6.2, we noted that the eSearch remapping algorithm is difficult to deploy. In this section, we show the remapping scheme, even if deployed, will be less efficient than our index partitioning technique. We begin with a description of our implementation

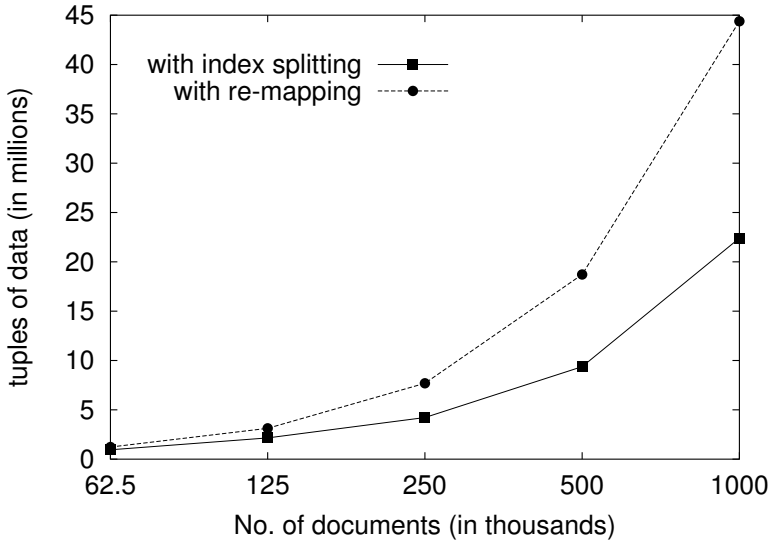


Figure 6.3: Amount of data transferred in eSearch with index partitioning vs. re-mapping.

of eSearch re-mapping.

We simulated the eSearch algorithm as described in [82]. The aggregate indexes in eSearch are large, and we partitioned these indexes among different nodes using both our index partitioning scheme and a centralized version of the index remapping scheme described in [27]. In our implementation, when a node n runs out of space, we locate the node n' with the least disk utilization, and re-map n' such that it splits the range occupied by the n. This provides an upper bound on any distributed implementation of the remapping scheme, since we always pick the best candidate node for re-mapping without accounting for the search overhead.

Figure 6.3 plots the amount of data transferred for building aggregate indexes for eSearch with the node re-mapping and index splitting algorithms. In these experiments, as the number of documents in the system doubles (from 62.5K – 1 million documents), we

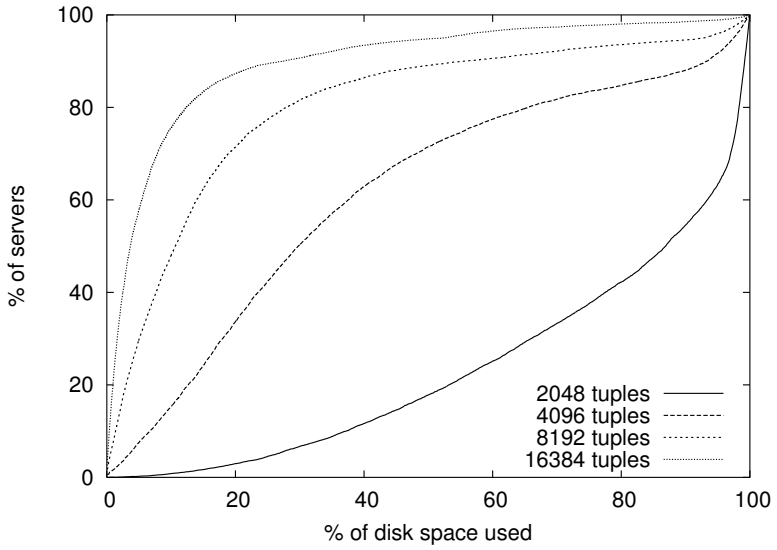


Figure 6.4: Space usage at nodes for different disk space allocations.

correspondingly increase the number of servers (from 1K servers for 62.5K documents to 16K servers for 1M documents).

The amount of data transferred using the re-mapping is at least *twice* that of our partitioning algorithm. This additional cost can be attributed to the fact that when a node re-maps itself, it has to move the data it is already storing to its successor, before re-attaching itself into the network.

6.5.4 Sensitivity to Disk space allocation

It is important to understand the sensitivity of the index partitioning algorithm to the amount of allocated space. In Figure 6.4, we plot the fraction of disk space used versus the fraction of nodes, as the space allocated for indexes is changed. (These experiments ran with view caching enabled, and all intersections were stored if space permitted). When the

amount of space available is small (e.g. 2048 tuples), the splitting algorithm is aggressive and distributes the data over a large set of nodes. As more disk space becomes available, the partitioning algorithm needs to run less often and indexes are split only when required. While smaller amount of allocated space means that the available space is used more evenly at each node, this also implies that the partitioning algorithm has to re-balance the available space more frequently, leading to higher overhead. For example, in this experiment the 2K tuple allocation transferred about 10 times more tuples than the 16K allocation.

6.5.5 Index Maintenance

Once an index has been created, the primary cost is in maintaining the index up to date. Indexes have to be updated when the data exported by the system changes (tuples are added or deleted), and when nodes join or leave. We first consider the case in which only the data changes (but no new nodes join the system, or existing nodes fail).

In Figure 6.5, we plot the number of tuples transferred over time as 1000 nodes join the system and export 64K documents (in first 500 seconds). For the next 1000 seconds, indexes are updated. Keywords are added and deleted from documents, with twice as many additions as deletions — thus, the number of tuples stored in the system slowly increases. As expected, the aggregate indexes have to store (and update) about 10 times more tuples, and this is shown in the plot.

Obviously, maintaining result caches (refer Chapter 4) and aggregate indexes (as used in eSearch) result in higher update overheads compared to when only the minimal

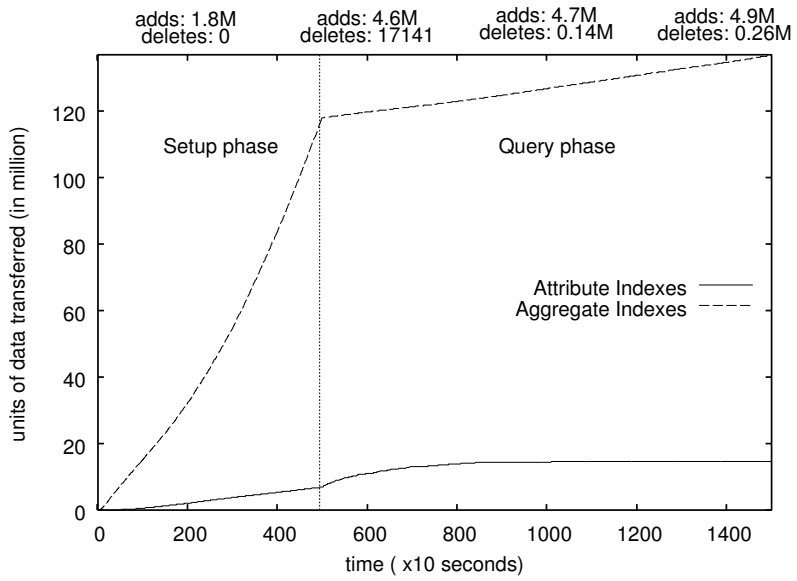


Figure 6.5: Update overhead without failures

attribute indexes are stored. In our experiments, result caches increase update costs by about a factor of 3, while updates to aggregate indexes are about 10 times more expensive.

6.5.6 Node Arrivals and Departures

In Table 6.2, we show the number of tuples transferred in maintaining indexes as nodes join and leave the system which nominally consisted of 1000 nodes, each with an allocation of 10240 tuples. In each experiment, the system generates 100 queries per second, and in total 30.25 million tuples are transferred as results over the simulation lifetime (5000 seconds). We consider two different values for node lifetimes. In the first case, the mean lifetime for a node in the system is 30 minutes, and over the course of the simulation, 1027 new nodes joined the system, while 989 existing nodes failed. When attribute indexes are used, the overhead due to churn is between 2–13%, depending on

	Lifetime (in min)	No. of Replicas 0	1	2	3
Attribute Indexes	30	0.64	1.8	2.94	4.17
	60	0.17	0.69	1.32	1.93
Aggregate Indexes	30	4.33	18.63	33.51	47.96
	60	1.41	8.79	16.03	23.29

Table 6.2: Amount of data transferred (in millions of tuples) for index maintenance

the number of replicas that have to be updated. The overhead decreases by about a factor of 5 when the node lifetimes is increase to 1 hour (50 joins and 52 leaves over 5000 seconds). The amount of data that has to be transferred using aggregate indexes (using index partitioning) is about one order of magnitude higher in all cases. This is because the indexes are about 10 times larger, and incur correspondingly higher overhead. Note that if remapping were used instead, the amount of data transferred again increase by about a factor of two.

6.5.7 Comparison of Replication and Erasure codes

Recall that an alternate strategy for storing static replicas uses erasure codes. Here, we analyze the trade-off for using such coding. For this experiment, we fix the number of partitions of the index to 32 and assume that the size of each partition is 100 times the size of an update (i.e. each partition holds 100 tuples, and an update changes a single tuple). We assume the number of erasure coded fragments required to re-create the partition is fixed at 8.

Table 6.3 compares the amount of storage required when erasure codes are used vs. when the data is simply copied at each replica, when nodes in the system fail with

Failure prob.	Index Size (EC = 1)	Update Cost (EC = 1) Online	Batch
0.01	1.85	0.018	0.46
0.1	3.2	0.032	0.8
0.2	4	0.04	1.0
0.5	5.85	0.058	1.46
0.9	7.62	0.076	1.91

Table 6.3: Normalized size and update cost for replicated indexes for different node failure probabilities. The data is normalized such the size/cost for using Erasure Coding is unit.

different (independent) failure probabilities. Specifically, in each row of the table, we choose an independent probability with which each node in the system fails, and create sufficient replicas such that the index can be recovered, using either replication or erasure coding, with probability $(1 - 10^{-4})$. The "Index Size" column shows the space required for replication as a factor of the space required for coded indexes. As is clear, replication uses 2–10 times more space depending on the failure probability. However, the trade-off is in the amount of data transferred during an update. When individual updates are immediately applied (Online column in Table 6.3), pure replication saves between 93–99% of the data transfers required by encoded storage. However, as we pointed out earlier, if updates can be batched, then the disparity can be significantly reduced ("Batch" column in Table). Here, we batch 25 updates, and the cost for updating coded indexes is reduced to at most twice that of replication, for even very low failure rates. Interestingly, for high enough failure rates, encoding can even be *more* efficient for batch updates than replication. Obviously, the particular storage scheme used by any system will depend on the expected operating regime of the deployment: our results indicate that encoded

storage is preferable when (1) update rates are very low, or (2) failure rates are high, or (3) if updates can be applied in batch.

6.6 Summary

In this chapter, we presented an algorithm for horizontal partitioning of indexes for use in P2P systems. The algorithm partitions index entries based on their prefixes. We show that such partitioning is feasible and helps both in increasing the quality of the results and in reducing network traffic, especially compared to other existing partitioning techniques.

We also analyze the effectiveness of replication and erasure codes in achieving resilience to failures. We show that under high failure rates, erasure codes are preferable to replication because they utilize lesser disk space while providing the same reliability guarantees. However, erasure codes are expensive when it comes to updating the indexes; the entire set of blocks need to be re-coded. In this chapter we showed results that indicate that the update overhead of erasure codes can be reduced by batching updates. However, further experiments are required to analyze the practical costs involved with implementing these strategies.

Chapter 7

Conclusion and Future Work

In this dissertation, we presented the challenges in building a decentralized search infrastructure. The main challenges with distributing such a system include node failures, churn, and data migration. Localities inherent in query patterns also cause load imbalances and hot spots that severely impair performance. From the users perspective, search systems should return results quickly, and in ranked order.

While there exist proposals for distributed search systems, we showed that these existing approaches do not meet all the requirements or have several debilitating problems in practice. Existing approaches only support a boolean query model and return all possible results, in no particular order. An ideal search system, however, should return only the most relevant results, rather than a large, unordered set of results. The result computation process of existing systems is extremely expensive. The usual procedure for computing multi-keyword (conjunctive) queries is to intersect the entries in these indexes. This requires transferring one or more indexes to the peer computing the intersection. Depending on the popularity of the indexes, these indexes can be very large and transferring them can get very expensive. If this system is to serve millions of queries per day, then the network (and processing) overhead of these intersections will prove to be prohibitive. Finally, there the storage problem. These systems consist of a number of regular hosts, and there is no control over which index(es) gets mapped to which host. Hence, it is highly

likely that the hosts holding the indexes for popular terms will be under-provisioned.

In this dissertation, we showed that a scalable, robust, and distributed search infrastructure *can be built* over existing Peer-to-Peer systems through the use of techniques that address these problems. Our main contribution is to present protocols necessary for building a wide-area, decentralized search system. In particular, we have designed protocols for the following:

- **Ranking** (Chapter 3) We presented a distributed algorithm for ranking search results. Our approach was to adapt extend a centralized information retrieval algorithm, namely Vector Space Model (VSM) [76] to Peer-to-Peer systems. The main challenge with applying VSM in a distributed manner is to compute the vectors in a distributed manner, distribute these generated vectors, and evaluate the query in a distributed manner. We showed that the global values of document and query vectors can be estimated accurately using random sampling. In order to store these vectors, we extend the existing distributed index infrastructure to store the vector representations in addition to storing the document entries. During evaluation, each query is evaluated locally at the index of each keyword in the query, and the top-k results from each of these local computations are aggregated to identify the eventual set of top-k results. Our solution demonstrates that distributed ranking is feasible with little network overhead.

- **Caching** (Chapter 4) The computation of multi-keyword conjunctive queries requires intersecting indexes. Depending on the popularity of the index, these can be very large. Repeating this process for every query makes this process prohibitively

expensive. We showed that caching is a viable strategy to address this problem. The challenge with caching results is to store them efficiently, identify stored caches, and determine the best set of caches for answering queries. To that end, we have designed a novel distributed data structure called the *View Tree* for this purpose. View trees can be used to efficiently cache and locate results from prior queries. View trees can also be used to answers queries using cached results of sub-queries. We showed that our technique reduced search overhead (both network and processing costs) by more than an order of magnitude.

- **Load Balancing** (Chapter 5) The side-effects of the imbalance in the popularity of keywords is that the load on peers storing indexes will be imbalanced. The standard approach to handle load imbalance is to replicate data based on their popularity. It is extremely hard, however, to predict the popularity of objects *a priori*. Further, the popularity of objects changes over time. To this extent, we have developed a new soft-state replication scheme for peer-to-peer networks called `LAR`. `LAR` is a replication framework which can be used in conjunction with almost any distributed data access scheme. `LAR` is adaptive: it can efficiently track changes in the query stream and autonomously organize system resources to best meet current demands. `LAR` can be configured to balance the load of peers by setting local bounds, and adapts to several orders of magnitude changes in demand over a few minutes. Compared to previous work, `LAR` has an order of magnitude lower overhead, and at least comparable performance.

- **Reliable Storage** (Chapter 6) In this chapter, we presented an adaptive algorithm

for horizontal partitioning of indexes for use in P2P systems. The algorithm partitions index entries based on their prefixes. We showed that such partitioning is feasible and helps both in increasing the quality of the results and in reducing network traffic, especially compared to other existing partitioning techniques. We also analyzed the effectiveness of replication and erasure codes in achieving resilience to failures. We showed that under high failure rates, erasure codes are preferable to replication because they utilize much lesser disk space while guaranteeing the same levels of reliability.

7.1 Future Work

As part of this dissertation, we have shown that the limitations in existing designs for a search infrastructure over P2P systems render them impractical. Additionally, we have proposed innovative techniques that effectively address some of the issues associated with these existing designs. While we believe that our techniques conceptually remove the most critical barriers from the realization of a wide-area distributed search infrastructure, there still remain many open questions and avenues for future work which we discuss in the rest of the section.

7.1.1 Two-level Search Infrastructure

In its current design, the structured P2P system forms the basic layer over which the different modules for supporting search are built. These different modules can be broadly classified into two groups: (a) techniques for reliability, and (b) techniques for

efficiency. In the current design, these two different techniques are both applied to the search meta-data (i.e, inverted indexes and result caches). However, there is tremendous heterogeneity in this meta-data, both in terms of purpose and characteristics. Inverted indexes are mandatory to support search and need to be stored reliably. However, they can get very large and as we have already shown, they alone are not sufficient for a practical search system. Result caches, on the other hand, are required for efficiency; they are useful in eliminating network traffic and can be highly replicated because they require very little disk space. This heterogeneity in the data forces reliability and efficiency mechanisms to be in constant tension with each other because they are fighting for the same set of shared resources (i.e., disk space and bandwidth).

As part of our future work, we propose a two-level architecture consisting of an index store layer and a caching layer. This architecture is designed to explicitly decouple reliability and efficiency. This is a key property since it acknowledges the heterogeneity in meta-data and allows us to use well-understood, slower timescale mechanisms for storing and updating large data (the indexes) while still enabling quick access to data that must be accessed quickly (search results).

Index Store Layer: The index store layer is designed to store large indexes *reliably*, and provides reasonably efficient access to index data. The index store layer accounts for available resources at individual hosts and partitions index data accordingly; it also accounts for expected failure rates and encodes/replicates index data such that indexes can be reconstructed when nodes fail.

The index store layer uses techniques described in Chapter 6 to partition large indexes among participating nodes. Since failures and node churn is frequent, the index

store keeps redundant copies of the index. We discussed some of the trade-offs of the various approaches to redundancy in Chapter 6. In practice, we envision a hybrid scheme in which one copy of the index is stored (partitioned among different nodes) without encoding and replicas are stored using erasure-coded blocks. When a node holding a partition fails or departs, the partition is re-created using appropriate erasure-coded blocks. The final operation required of the index store is to compute (ideally once) the relevant results of a query. We demonstrated one technique that uses Vector Space Models [76] in Chapter 3. Since information about all possible query results are present within the index store layer, there is sufficient information here to rank order query results and extract only the top few results.

While the index store allows complete access to indexes, the protocols are not optimized for frequent accesses by individual queries. The protocols in the index store layer are designed to manipulate large data objects and must be reliable: all of the replication here is to ensure that failures do not cause loss of difficult to recreate data. Access to the index data need not be instantaneous as long as the caching layer has an adequate hit rate.

Caching layer: In Chapter 4, we asserted that result caching is *necessary* for a large scale search infrastructure. For almost all queries, users are interested in only the top few *relevant* results, and efficiently fetching the top-k results is an optimization built into many information retrieval tools. The ability to store and retrieve a small set of highly ranked results also further reduces the overhead of individual queries.

The caching layer serves that exact purpose by providing *efficient* access to popular indexes, and to the results of prior queries. All queries are initially directed to the caching layer. The query terms are used to locate appropriate caches, and if the query

can be satisfied within the caching layer, results are immediately returned. Upon a miss, the caching layer transparently locates the appropriate index in the index store layer and requests the requisite number of top results for the query. The index store layer computes the relevant results and forwards the results to the caching layer, which then returns the results to the query. The challenge with caching results lies in storing the caches such that they can be efficiently located and re-used throughout the system. We plan to make use of View Trees (refer to Chapter 4) for this purpose. Finally, cached results are dynamically replicated using LAR (Chapter 5) to account for skews in the query distribution, i.e., results for popular queries are cached at more nodes. Since these caches are small, they can be replicated aggressively. Thus, the caching layer, is designed for access to relatively small items (top k results for different queries). These items are accessed frequently, and are replicated based on their access frequency. The caching protocols must provide fast access to applicable results. A miss is not fatal, however, since the result can be slowly recomputed using the data stored in the indexing layer.

7.1.2 Index Storage

There is much work still needed in understanding how to store large objects in decentralized systems, including DHTs. While data partitioning is a prerequisite, the amount of data that must be moved when a node joins/leave the system may render the entire system unusable. The overall impact of the replication versus erasure coding choice needs further study. While erasure codes have advantages in bandwidth and storage overhead, they require contacting many more nodes than does replication. Further, a deter-

ministic scheme is needed to publish erasure coded blocks in such a way that they can be easily retrieved, but the best method to do this is not yet obvious. Batching can reduce the overhead of using erasure codes. However, it also increases the chance of losing updates due to failures. Finally, the effect of the update period on overhead and reliability needs to be analyzed.

7.1.3 Query Distribution, Caching and Ranking

The characteristics of web-search data sets and query streams need to be tightly characterized. Highly dynamic systems and variable access patterns can play havoc with complicated caches. If the most relevant documents for a given query change, the cache must be updated or the quality of the results degrade. Clearly this issue is highly dependent on the characteristics of the data sets and query streams used, but traces of search engines like Google [33] are prohibitively large. We would ideally like to create a set of relatively small traces that are provably representative of Google-like systems. Failing this, there is still a great deal of research that can be done studying properties of systems like the one proposed in this dissertation, such as analyzing how quickly cached results degrade with increasingly dynamic systems.

7.1.4 Security and System Model

An unstated and somewhat naive assumption in this dissertation has been that of an entirely cooperative (and non-malicious) set of peers, and we have designed the system to withstand random node failures only. While to the best of our knowledge our architecture

does not enable any *new* security holes, there are any number of attacks that can render the system essentially useless. A number of these attacks require formulation of global policy, e.g. how to handle nodes that publish junk data to fill up all available space? Other attacks include nodes that selectively deny service to other nodes, or respond with spurious results. We believe the encoded storage techniques can help the resilience of the system since they allow data to be reconstructed even if a large number of nodes are malicious/attacked; however, a systematic study of their resilience to different attacks is open. Spurious data supplied by malicious nodes is not an issue if all data is self-certifying, but in a wide-area network, it is not clear how third-party signatures can be checked without a trusted CA or a PKI. If a decentralized system is to be widely used for search, all of these issues must eventually be addressed explicitly.

BIBLIOGRAPHY

[1] K. Aberer, P. Cudr-Maurou, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-Grid: A Self-organizing Structured P2P System. *SIGMOD Record*, 32(2), September 2003.

[2] Akamai home page. `http://www.akamai.com/`.

[3] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the WWW. In *Proceedings of PDIS'96: The IEEE Conference on Parallel and Distributed Information Systems*, Miami Beach, FL, December 1996.

[4] D. R. Angelos Stavrou and S. Sahu. A lightweight, robust P2P system to handle flash crowds. In *Proceedings of 10th IEEE International Conference on Network Protocols (ICNP 2002)*, Paris, France, November 2002.

[5] M. Bawa, A. Gionis, H. Garcia-Molina, and R. Motwani. Estimating aggregates on a peer-to-peer network. Technical report, Computer Science Dept., Stanford University, 2003.

[6] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Feb. 2003.

[7] B. Bhattacharjee, P. Keleher, and B. Silaghi. The design of TerraDir. Technical Report CS-TR-4299, University of Maryland, College Park, MD, October 2001.

[8] I. Bhattacharya, S. R. Kashyap, and S. Parthasarathy. Similarity searching in peer-to-peer databases. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pages 329–338, 2005.

[9] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[10] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of the INFOCOM '99 conference*, March 1999.

[11] C. Buckley. Implementation of the SMART information retrieval system. Technical report, Dept. of Computer Science, Cornell University, Ithaca, NY, USA, 1985.

[12] J. Byers, J. Considine, and M. Mitzenmacher. Simple load balancing for distributed hash tables. In *The 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, February 2003.

[13] P. Cao and Z. Wang. Efficient top-k query calculation in distributed networks. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 206–215, New York, NY, USA, 2004. ACM Press.

[14] M. Castro, M. Costa, and A. Rowstron. Performance and dependability of structured peer-to-peer overlays. Technical Report MSR-TR-2003-94, Microsoft Research, Cambridge, UK, December 2003.

[15] S. Ceri, M. Negri, and G. Pelagatti. Horizontal data partitioning in database design. In *Proceedings of the 1982 ACM SIGMOD international conference on Management of data*, pages 128–136. ACM Press, 1982.

[16] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making gnutella-like P2P systems scalable. In *Proceedings of the ACM SIGCOMM '03*, August 2003.

[17] Cisco localdirector. `http://www.cisco.com/warp/public/751/lodir/`.

[18] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system, 2000.

[19] E. Cohen and S. Shenker. Replication strategies in unstructured peer-to-peer networks. In *The ACM SIGCOMM'02*, August 2002.

[20] F. M. Cuenca-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen. PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities. In *Twelfth IEEE International Symposium on High Performance Distributed Computing (HPDC-12)*. IEEE Press, June 2003.

[21] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the* 18th *ACM Symposium on Operating Systems Principles*, Chateau Lake Louise, Banff, Canada, October 2001.

[22] S. Deerwester, S. Dumais, T. Landauer, G. Furnas, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[23] S. T. Dumais. Improving the retrieval of information from external sources. *Behavior Research Methods, Instruments, and Computers*, 23(2):229–236, 1991.

[24] R. Fagin. Combining fuzzy information: an overview. *SIGMOD Record*, 31(2):109–118, June 2002.

[25] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences (JCSS)*, 66(4):614–656, 2003.

[26] W. B. Frakes and R. Baeza-Yates. *Information retrieval: data structures and algorithms*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[27] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *Proceedings of the* 30th *VLDB Conference*, Toronto, Canada, August 2004.

[28] P. Ganesan, Q. Sun, and H. Garcia-Molina. Yappers: A peer-to-peer lookup service over arbitrary topology. In *22nd Annual Joint Conf. of the IEEE Computer and Communications Societies (INFOCOM)*, San Francisco, USA, March 2003.

[29] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman Company, Nov. 1990.

[30] O. D. Gnawali. A keyword set search system for peer-to-peer networks. Master's thesis, Massachusetts Institute of Technology, June 2002.

[31] Gnutella2 specification. `http://www.gnutella2.com`.

[32] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in dynamic structured p2p systems. In *23nd Annual Joint Conference of the IEEE Computer and Communications Societies(INFOCOM 2004)*, March 2004.

[33] Google home page. `http://www.google.com`.

[34] V. Gopalakrishnan, B. Bhattacharjee, S. Chawathe, and P. Keleher. Efficient peer-to-peer namespace searches. Technical Report CS-TR-4568, University of Maryland, College Park, MD, February 2004.

[35] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, and P. Keleher. Adaptive replication in peer-to-peer systems. Technical Report CS-TR-4515, University of Maryland, College Park, MD, July 2003.

[36] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 314–329. ACM Press, 2003.

[37] A. Gupta, H. Jagadish, and I. Mumick. Data integration using self-maintainable views. Technical memorandum, AT&T Bell Laboratories, Nov. 1994.

[38] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, March 2003.

[39] A. Y. Halevy. Theory of answering queries using views. *SIGMOD Record*, 29(4), Dec. 2000.

[40] D. Harman, editor. *The Second Text Retrieval Conference*, Gaithersburg, MD, 1994. National Institute of Standards and Technology.

[41] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A scalable overlay network with practical locality properties. In *Fourth USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Seattle, WA, March 2003.

[42] J. Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of the* 9^{th} *International Conference on Architectural Support for Programming Laguages and Operating Systems*. ACM, November 2000.

[43] M. F. Kaashoek and D. R. Karger. Koorde: A simple degree-optimal distributed hash table. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, March 2003.

[44] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[45] D. R. Karger and M. Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 36–43, New York, NY, USA, 2004. ACM Press.

[46] Kazaa home page. `http://www.kazaa.com`.

[47] P. Keleher, S. Bhattacharjee, and B. Silaghi. Are virtualized overlay networks too much of a good thing? In *The 1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*, 2002.

[48] A. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. In *Proceedings of the IEEE International Conference on Parallel and Distributed Information Systems*, pages 229–238, Austin, Texas, Sept. 1994.

[49] D. Kempe and F. McSherry. A decentralized algorithm for spectral analysis. In *STOC '04: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 561–568, New York, NY, USA, 2004. ACM Press.

[50] V. King and J. Saia. Choosing a random peer. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 125–130, New York, NY, USA, 2004.

[51] D. E. Knuth. *The Art of Computer Programming, volume 1: Fundamental Algorithms*. Addison-Wesley, Reading, MA, 1973.

[52] J. Li, B. T. Loo, J. Hellerstein, F. Kaashoek, D. Karger, and R. Morris. On the feasibility of peer-to-peer web indexing and search. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, Feb 2003.

[53] B. T. Loo, J. M. Hellerstein, R. Huebsch, S. Shenker, and I. Stoica. Enhancing P2P file-sharing with an internet-scale query processor. In *Thirtieth International Conference on Very Large Data Bases (VLDB '04)*, pages 432–443, Toronto, Canada, August 2004.

[54] B. T. Loo, R. Huebsch, I. Stoica, and J. M. Hellerstein. The Case for a Hybrid P2P Search Infrastructure. In *3rd International Workshop on Peer-to-Peer Systems (IPTPS)*, San Diego, CA, Feb 2004.

[55] M. Luby, M. Mitzenmacher, A. Shokrollahi, and D. Spielman. Efficient erasure correcting codes. *IEEE Transactions on Information Theory*, 47:569–584, February 2001.

[56] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th ACM International Conference on Supercomputing*, New York, USA, June 2002.

[57] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *21st ACM Symposium on Principles of Distributed Computing (PODC '02)*, Monterey, CA, August 2002.

[58] S. Michel, P. Triantafillou, and G. Weikum. KLEE: A framework for distributed top-k query algorithms. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB '05)*, pages 637–648, Trondheim, Norway, August 2005.

[59] M. Mitzenmacher. Compressed Bloom filters. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 144–150. ACM Press, 2001.

[60] R. Morselli, B. Bhattacharjee, A. Srinivasan, and M. A. Marsh. Efficient lookup on unstructured topologies. In *PODC '05: Proceedings of the twenty-fourth annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 77–86, New York, NY, USA, 2005.

[61] J. Mullin. Optimal semijoins for distributed database systems. *IEEE Transactions on Software Engineering*, 16(5):558–560, 1990.

[62] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the* 5th *USENIX Symposium on Operating Systems Design and Implementation*, Boston, Massachusetts, December 2002.

[63] Napster home page. `http://www.napster.com`.

[64] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation algorithm: bringing order to the web. Technical report, Dept. of Computer Science, Stanford University, 1999.

[65] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[66] M. Rabinovich, I. Rabinovich, R. Rajaraman, and A. Aggarwal. A dynamic object replication and migration protocol for an internet hosting service. In *International Conference on Distributed Computing Systems*, pages 101–113, 1999.

[67] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in structured p2p systems. In *The 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, February 2003.

[68] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM '01*, San Diego, California, August 2001.

[69] I. Reed and G. Solomon. Polynomial codes over certain finite fields. *J. Soc. Indust. Appl. Math*, 8:300–304, 1960.

[70] P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In *Proceedings of IFIP/ACM Middleware*, 2003.

[71] R. Rodrigues and B. Liskov. High availability in DHTs: Erasure coding vs. replication. In *Proceedings of the 4th International Workshop on Peer-to-Peer systems (IPTPS)*, February 2005.

[72] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of IFIP/ACM Middleware*, Heidelberg, Germany, November 2001.

[73] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the* 18th *ACM Symposium on Operating Systems Principles*, Chateau Lake Louise, Banff, Canada, October 2001.

[74] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing and Management*, 24(5):513–523, 1988.

[75] G. Salton, E. A. Fox, and H. Wu. Extended boolean information retrieval. *Commun. ACM*, 26(11):1022–1036, 1983.

[76] G. Salton, A. Wong, and C. Yang. A vector space model for information retrieval. *Journal of the American Society for Information Retrieval*, 18(11):613–620, 1975.

[77] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN '02)*, San Jose, CA, USA, January 2002.

[78] K. Sripanidkulchai. The popularity of gnutella queries and its implications on scalability. Available from `http://www.cs.cmu.edu/~kunwadee`, February 2001.

[79] K. Sripanidkulchai. The popularity of Gnutella queries and its implications on scalability, February 2001.

[80] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01*, San Diego, California, August 2001.

[81] T. Suel, C. Mathur, J. Wu, J. Zhang, A. Delis, M. Kharrazi, X. Long, and K. Shanmugasundaram. Odissea: A peer-to-peer architecture for scalable web search and information retrieval. In *6th International Workshop on the Web and Databases (WebDB)*, June 2003.

[82] C. Tang and S. Dwarakadas. Hybrid global-local indexing for efficient peer-to-peer information retrieval. In *Proceedings of USENIX NSDI '04 Conference*, San Fransisco, CA, March 2004.

[83] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *Proceedings of ACM SIGCOMM '03*, pages 175–186, Karlsruhe, Germany, 2003. ACM Press.

[84] Trec: Text REtrieval Conference. `http://trec.nist.gov/`.

[85] J. D. Ullman. Information integration using logical views. In *Proceedings of the International Conference on Database Theory*, 1997.

[86] Y. Wang and D. J. DeWitt. Computing PageRank in a distributed internet search engine system. In *Thirtieth International Conference on Very Large Data Bases (VLDB '04)*, pages 420–431, Toronto, Canada, August 2004.

[87] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proceedings of the 1st International Workshop on Peer-to-Peer systems (IPTPS)*, March 2002.

[88] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems*, 22(2):255–314, 1997.

[89] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for wide-area location and routing. Technical Report UCB//CSD-01-1141, U.C.Berkeley, Berkeley, CA, April 2001.

[90] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Jose, California, May 1995.

[91] G. K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley, Cambridge, Massachusetts, 1949.