KEYWORD-BASED SEARCH IN PEER-TO-PEER NETWORKS

Yingguang Li

NATIONAL UNIVERSITY OF SINGAPORE
2008

KEYWORD-BASED SEARCH IN PEER-TO-PEER NETWORKS

Yingguang Li

(M.Sc. NATIONAL UNIVERSITY OF SINGAPORE)

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
DEPARTMENT OF COMPUTER SCIENCE
SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE
2008

Acknowledgment

I would like to express my sincere and deep gratitude to my advisor, Professor Kian-Lee Tan for his guidance during my research and study at the National University of Singapore (NUS). His patience, understanding and encouragement have helped me greatly throughout my five years of Ph.D. study. When I brought many naive ideas to him, he explained to me why they are too simple or impractical, but he also discussed with me possible extensions from them; when I changed a few research problems in the early stage, he gave me time to broaden my knowledge; when I was frustrated by some rejections on my paper submissions, he encouraged me and helped me to get the papers accepted eventually. Moreover, I appreciate the countless hours he spent to update my writings and improve my presentations.

I would also like to thank the oversea co-authors: Professor H. V. Jagadish from the University of Michigan, Professor M. Tamer Özsu from the University of Waterloo and Associate Professor Lidan Shou from Zhejiang University. Professor Jagadish's insight on SPRITE improves the technical content and literary style of the paper. His theorization on SPRITE has inspired me in the early stage of my Ph.D. study. Professor Özsu spent a lot of time to discuss with me on the XCube work when he was visiting NUS. Associate Professor Shou discussed with me about the idea on CYBER. After he went back to China, we continued the discussion until the work was accepted for publication.

I am very thankful to the members of my thesis evaluation committee: Dr.

Chee Yong Chan and Dr. Panagiotis Kalnis. The advice and comment from them on my term paper and thesis proposal helped me to refine my work and explore new research problems in the early stage of my Ph.D. study.

I am so happy that I have been a member of the database group, a big family full of joy and research spirit. I would like to thank Professor Beng Chin Ooi. He taught and inspired me many things when I worked with him as a research assistant. I also thank Dr. Chee Yong Chan for his kind support in the later stage of my study. I want to thank Dr. Panagiotis Kalnis for the discussions with him when I was looking for research problems in the P2P realm. Thank Dr. Anthony Tung for sharing with us his understanding on research. I would like to thank Dr. Stéphane Bressan and Dr. Mong Li Lee who showed me the research path.

I would like to thank my friends in NUS also, for their encouragement, discussions, team work, and company, especially before conference deadlines. They are: Xuan Zhou, Yanfeng Shu, Wee Hyong Tok, Wenqiang Wang, Chenyi Xia, Bin Cui, Qi He, Zhuo Chen, Wei Ni, Shili Xiang, Changqing Li, Yuan Ni, Ting Chen, Jing Hu, Enhua Jiao, Wei Zhang, Han Zhang, Wei Zheng, Chong Sun, Weiwei Cheng, Gabriel Ghinita, Ding Chen, Xianjun Wang, Jianneng Cao, Bin Liu, Chang Sheng, Xiaoyan Yang, Zhifeng Bao, Liang Xu, Huayu Wu, Yueguo Chen, Bei Yu, Sai Wu, Quang Hieu Vu, Mihai Lupu, Zhenjie Zhang, Yu Cao, Su Chen, Dongxiang Zhang, Bingtian Dai, Ji Wu, Wei Wu, Yongluan Zhou, Xuyang Song, Linhao Xu and many others. They have made my study in the big family very enjoyable.

I would like to thank my parents for their consistent love, encouragement and understanding. I also want to thank my wife, Jun, for her love, her support during my Ph.D. study and the happiness she brings to me.

Finally, I want to thank NUS for providing me the scholarship so that I can concentrate studying.

Contents

Summary

Information sharing is one of the most useful applications of Internet. Peer-to-peer (P2P) platform attracts many researchers' attention because of the increasing number of users and the advantages of P2P systems over traditional centralized systems, such as scalability and administration-free. While P2P platforms provide many advantages, we are facing many new research challenges as well. In this dissertation, we focus on issues related to keyword-based search in P2P networks, because keyword-based search is the most feasible and easiest searching interface in a decentralized system where users are not expected to have apriori knowledge about the remote data.

We first propose **SPRITE** (**S**elective **PR**ogressive **I**ndex **T**uning by **E**xamples), to build effective index on the shared data in a structured P2P network. In a P2P network, building complete inverted index for documents is infeasible due to the high maintenance cost. SPRITE builds partial index based on the query history so that only the representative terms of a document are chosen and indexed. With the compact, yet accurate index, SPRITE is able to achieve good search performance close to a centralized system with complete index.

We then propose **CYBER** (a **C**ommunit**Y**-**B**ased s**E**a**R**ch engine) to further improve the search effectiveness by incorporating social network and feedback techniques. In CYBER, users with similar interests (a community) are linked together with their profiles implicitly. Within such a community, a document identified as relevant by a user is likely ranked higher to a query issued by another user. Our experimental results show that CYBER outperforms the traditional feedback techniques because it accumulates positive feedback.

Besides searching plain text data, we also investigate how to share and query XML data, which is also a kind of text data, yet with more complex structure. We propose **XCube** to process **X**Path (and tag-based) queries in a hyper**Cube** overlay network. The XCube system extracts the tag names from an XML document, and then indexes them together as one entry. Given an XPath query, the tag names in the query are extracted in the same way first. A group of peers containing the supersets of the query tags are searched. The structural constraints and predicates are examined in the related indexing peers and owner peers respectively. We compare XCUBE with the scheme that indexes individual tags and show that XCUBE is more efficient.

We believe that our research has identified and solved some significant problems in keyword-based searching systems in P2P networks. Our comprehensive experimental results and the comparison with the representative existing methods prove that the proposed schemes improve the searching effectiveness and efficiency tremendously. Such improvements make keyword-based search in P2P networks more feasible and attractive to end users.

List of Figures

List of Tables

Chapter 1

Introduction

Along with the inventions of intranet and internet, the amount of information increases dramatically. On one hand, more information is generated and shared; on the other hand, more users are searching useful information from the internet/intranet. The last decade has witnessed the limitations of the traditional client-server (C/S) computing architecture on searching data. In a C/S structured network, a server can cause a single point of failure easily, which makes the whole network vulnerable. Data sharing *in situ* is infeasible in many applications, such as search engines, which mainly rely on crawlers to collect data. Moreover, the C/S architecture apparently limits the scalability of the network.

Personal computers are becoming more powerful with faster processor, larger RAM and storage, yet more affordable in terms of price. The network bandwidth for normal users is increased significantly nowadays. Such hardware improvement makes Peer-to-peer (P2P) network architecture practical. A P2P network incorporates a number of computing nodes with some shared resources, such as storage and bandwidth, to provide some network services. Among these resources, bandwidth is usually the bottleneck because data indexing, monitoring, searching and routing require message transmission. A key characteristic of a P2P network is that every peer plays the role of both server and client. One arbitrary peer (or even several peers) going offline will not stop the entire network service. Similarly, it is important for all peers to have similar work load so that some offline peers will not affect the network service seriously. The number of peers/users in a P2P network can increase freely as every peer consumes resources as well as provides resources.

It is worth noting that we are not advocating that P2P networks will dominate and replace C/S networks completely. On the contrary, we believe that they are mutually complementary and suitable for different applications. A C/S network tends to minimize the resources consumed, while a P2P network manages to fully utilize the resources in the network. Therefore, we do not compare the two types of networks on the network cost in this thesis.

Currently, the existing P2P network systems provide several kinds of services, such as data sharing, storage sharing, audio and video media streaming. In this thesis, we focus on the service of data sharing. Query processing has been addressed for various types of queries, such as range query and K-nearest neighbor (KNN) query [7, 47, 71, 75], skyline query [87, 22, 45, 19] and queries in publish/subscribe systems [17, 6, 3, 74]. In a P2P network, different softwares and applications are being used by peers with various operating systems. Hence, many different types of data are generated and shared. In order to share data among peers, an easy way is to convert or annotate them to text format, which is acceptable for all operating systems. Keyword-based queries can be easily interpreted by all peers. Keyword search has been extensively studied on pure text data [10, 40, 16, 15], XML data [5, 32, 93, 36], and relational data [35, 4, 34] in centralized systems. However, many research challenges on keyword search in a P2P environment are not addressed. Moreover, processing complicated queries, such as SQL queries in traditional database management systems, in a P2P network is non-trivial. Such queries usually require users to have better knowledge on the data sources they are querying on, which is hardly true in a P2P network. While keyword search has also been investigated in the relational context [98], this thesis focuses on keyword-based search for textual (document and xml) data in P2P networks.

1.1 Keyword-based Search in P2P Networks

Supporting keyword-based search (also known as text retrieval) in a large scale distributed environment (e.g., P2P networks) is a challenging task. Traditional document retrieval techniques need statistical information of the entire corpus (global knowledge) to calculate similarities and rank the result list, such as the document frequency of a term and the corpus size (total number of documents). Hence, such techniques cannot be directly applied to a distributed environment where global knowledge is unavailable.

In the literature, there are mainly four approaches to support keyword-based search in P2P systems. The most straightforward approach, typically adopted in unstructured systems, such as Gnutella [30], is to flood a query within a certain radius of the neighborhood of the querying peer. However, such an approach is not only bandwidth inefficient but may have low recall (the ratio of discovered relevant answers over all relevant answers) as peers containing relevant documents may be beyond the search scope and unknown within the local neighborhood searched. To reduce the communication overhead, an alternative approach is to employ *routing indexes* [21] that provide more directed search as only peers with matching query terms are searched. However, this method also operates within a certain radius in an unstructured environment, and has the same limitation of low recall.

A third approach employs a structured overlay network. Every document is indexed in the structured network on the terms it contains [83, 49]. In other words, each peer maintains an inverted list for the terms assigned to it by the overlay network. To process a query, all peers responsible for the query keywords are visited, and the relevant index entries are returned to the querying peer. The querying peer can then compute the similarities between the query and the documents containing those keywords to generate the ranked list. This approach is relatively query-efficient, and is expected to have higher recall than the other approaches.

The fourth approach is to index the documents on some combinations of certain terms in a structured P2P network [29, 85, 38]. Each term combination is indexed in an indexing peer similar to the term indexing scheme in the third approach. When processing a query, peers responsible for the related term combinations are contacted. This approach attempts to reduce the number of participating peers for a query from the third approach.

The former two approaches employing unstructured P2P networks have some key drawbacks. Broadcasting a query in a P2P network is expensive, even with TTL to control the search radius. Information discovered is always from "nearby" peers, which limits the scalability of the network. Many relevant documents of better quality (with larger similarities) may be missed out as they are beyond the search radius, thus the recall is seriously affected. Such approaches based on unstructured networks are only suitable for some applications. Therefore, we mainly investigate the mechanisms employing structured P2P networks. Structured P2P networks guarantee that existing answers can be found with routing cost of logarithmic bound.

The latter two approaches reduce the number of contacted peers to a query significantly by leveraging on the efficiency of structured P2P networks. However, the construction of a distributed index may involve a large number of peers because the number index entries is usually proportional to the number of terms in a document. The cost to build such indices is high, and maintaining the indices will cost even more messages. We tackle this challenge in Chapter 4.

1.2 Motivations

As text data can be processed on all types platforms, the demand on keyword-based search in P2P networks is increasing rapidly. For example, many big organizations are employing P2P systems to store, backup and share documents. Employees search text data, such as emails and documents by issuing keyword queries. Even in conventional file sharing, eg. software distribution, P2P users start to issue keyword queries to search softwares with certain functionality and hardware requirements. We have seen the limitations of unstructured networks on data sharing in the previous section. We now investigate several key issues of keyword-based search in structured networks. These issues motivate the work in this thesis.

1.2.1 Building Compact Yet Effective Index

The number of shared documents in a P2P network is usually proportional to the number of users. Each shared document contains a large number of terms (keywords) also. In a P2P network, a peer is allowed to join and leave the network freely without notifying other peers. When a peer, P_i (indexing peer), indexes a term for a document shared by another peer, P_o (owner peer), either P_i needs to ping P_o periodically to check its availability and thus maintains its index up to date; or P_o needs to ping P_i periodically to ensure the indexing peer is alive (otherwise, P_o will re-index the document on that particular term). If all terms in each document are indexed in a P2P network, then peers will be busy with pinging the indexing peers or the owner peers. Such maintenance overhead is significantly huge when more peers join the network and share more documents. Therefore, building complete index seriously degrades the scalability of a P2P network.

Although such pinging messages are small in terms of size, the total number of such messages is huge in a P2P network. Assume there are 10000 peers in a P2P network; on average, every peer shares 10 documents; each document contains 1000 distinct terms; and an owner peer pings an indexing peer every 1 hour. An owner peer has to check the availability of 10000 indexing peers periodically (equivalent to broadcast), which means the peer has to handle about 3 pinging messages every second. From the point of view of an individual peer, such frequent pinging messages will surely degrade its performance. From the point of view of the entire P2P network, the significant overhead on the maintenance over-consumes the network bandwidth. Moreover, the complete distributed indices cause the sizes of many index entries on popular terms to be large. When such an indexing peer reacts to some queries, the size of the replied message to the querying peers is large too. Hence, there is a need to investigate ways to reduce maintenance overhead without sacrificing the answer quality.

Besides the maintenance overhead, Li. et al. also extensively discuss the impracticality of building complete index in a P2P network with storage constraints in [46]. Without compression, each peer has to contribute several gigabytes of storage on average to store complete index entries, which is a significant overhead as a program requirement in a personal computer.

1.2.2 Improving search quality

In centralized information retrieval systems, techniques based on user feedback have been effective in improving the query precision and recall. These methods typically re-formulate and re-evaluate a query based on the feedback provided by the user who issues the query. After a ranked list is returned to a user, the user selects some results as relevant answers. According to the relevant answers, some terms are injected into the query or their weights in the query are increased. The new query, which reflects the user interest more accurately, is sent back for evaluation again. However, it is non-trivial to deploy these feedback-based techniques directly in a P2P network. In a relatively dynamic (unstructured or structured) P2P network, submitting a query multiple times means increasing the cost for routing the query proportionally. Additionally, because of the dynamism of the system as peers join and leave the network, the user may have to wait for a longer response time or some answers may be missed. Therefore, more intelligent novel methods are required to improve searching effectiveness in P2P networks without sacrificing the efficiency.

Moreover, we observe that many users share some common interests. Such users construct a community and tend to issue similar/overlapping queries. The existing research work has demonstrated that a single ranked list cannot satisfy users from different communities issuing the same query [99, 44, 20, 88]. Ideally, a unique ranked list should be generated for each community. If a query can be re-formulated and re-evaluated based on the past queries from the same user community, then we can achieve similar search quality as employing the feedback techniques. However, how to incorporate community-based relevance feedback in a P2P network has not yet to be clearly defined. Since a user can have multiple interests at a time, it is not clear how the query of his current interest can be associated with the correct community. Therefore, a community-based relevance feedback technique is desired to improve search accuracy in P2P networks.

1.2.3 Handling structural constraints

We have seen keyword-based search on plain text data in the previous two sections. In many applications, searching for data of richer format is strongly demanded. On one hand, a lot of information has been described and represented with richer format; on the other hand, many data are generated by some programs or applications, rather than by the users manually. XML - a text-based, self-descriptive, tagged language for encoding hierarchical data structures - can be readily understood by users and machines, and as such, has been widely used as a standard to represent and exchange data. Comparing with the pure text data, which is document-centric, XML data are more data-centric. The text content in every element can be queried possibly. Therefore, we cannot summarize an XML document with a small number of terms only.

Designing a peer-based XML data management system requires addressing two tightly integrated issues: *search capability* and *query expressiveness*. The first issue is influenced by the overlay structure of the P2P network. In order to find all available answers, a structured network is employed to avoid broadcasting the entire network. And the second issue deals with the query types that can be supported. Structural constraints are always embedded in most XML queries, such as XPath [91] and XQuery [92]. For the sake of simplicity, we discuss the XPath query processing solely in this thesis, but it is easy to extend our work to support XQuery as well. Only the elements in certain "paths" in some XML documents are potential answers to an XPath query. XPath queries mainly contain two types of conditions to examine: structural constraints and predicates on attribute/element. Hence, XML documents can be indexed on the structure of the document, the attribute values or both of them. Due to the data-centric constraint, building distributed index on every attribute value and element content is infeasible because of the high index maintenance cost. This is because every attribute or element could be queried, such as author names and book titles. The data-centric characteristic of XML documents renders that summarizing the content is ineffective in reducing the number of index entries. On the contrary, structural information is easier to summarize since the number of tags is usually much smaller than the number of keywords/numbers in the content. Hence, indexing the structure of a document is both feasible to deploy and selective for many queries.

Since an XPath query cannot be completely handled with the indices on structure, content or both of them, we have to locate the owner peers of the potentially relevant documents first, and then process the query in every owner peer. In a P2P network, the size of a document shared by a normal user is usually very small and a peer shares a small number of documents, thus processing XPath queries locally can be easily handled by many existing softwares[1]. Instead, locating the relevant owner peers efficiently for an XPath query is the core operation. It is exactly this challenge that we tackle in this work. Many existing works are proposed to index all the distinct tags in XML documents [25, 2]. The query issuing peer process a query by consolidating all path/fragment metadata collected from the related indexing peers. This approach incurs two problems. One problem is popular tags can overload some indexing peers easily; the other one is the the querying peer cannot locate the relevant data sources until the last message (on a tag) is replied. Therefore, a novel mechanism is needed to balance the load and improve the efficiency.

[1]In the case that a large number of XML documents or an XML document of large size are shared by a peer, we assume the peer is as capable as a server. Thus, query processing is also efficient in such peers

1.3 Contributions

The major contributions of this thesis are three-fold:

- In Chapter 4, we propose **SPRITE** (**S**elective **PR**ogressive **I**ndex **T**uning by **E**xamples) to bring down the cost of index construction and maintenance in a DHT network. In SPRITE, a small number of representative terms are selected and indexed for a document. This is extremely important in a P2P system, not only for index construction and update, but also because periodic checking on distributed indexes is required. Moreover, SPRITE refines the selected index terms by learning from past queries progressively, so that the search effectiveness can recover very soon when the query patterns change. Our extensive simulation study shows that SPRITE can achieve performance similar to a centralized system in terms of precision and recall, and considerably outperforms a static index term selection approach.

- In Chapter 5, we propose **CYBER**, a **C**ommunit**Y**-**B**ased s**E**a**R**ch engine, for information retrieval utilizing community-based feedback information in a DHT network. In CYBER, each user is associated with a set of user profiles that capture his/her interests. As such, a group of users sharing similar interests will have similar profiles and form a (virtual) community. Likewise, a document is associated with a set of profiles - one for each indexed term. A document profile is updated by users who query on the term and consider the document as a relevant answer. Thus, the profile acts as a consolidation of users feedback from the same community, and reflects their interests. In this way, as one user finds a document to be relevant, another user in the same community issuing a similar query will benefit from the feedback provided by the earlier user. Hence, the search quality in terms of both precision and recall is improved. We conduct a comprehensive experimental study and the results show the effectiveness of our scheme.

- In Chapter 6, we propose **XCube**, a tag-based scheme that manages **X**ML data in a hyper**Cube** overlay network to support XPath (and tag-based) queries. In XCube, each node in a d-dimensional hypercube is identified by a d-bit vector. A peer manages a smaller hypercube with dimension $d' < d$. An XML document is compactly represented as a *structure summary* and a *content summary.* The *structure summary* comprises a d-bit vector derived from the distinct tag names in the document and a synopsis capturing the structure of the document. The *content summary* consists of a bit map that summarizes the document content. The metadata of a document, i.e., owner IP, document identifier, structure summary and content summary, is indexed at its *anchor peer* (the peer that manages the node with matching bit vector). In addition, the structure summary is further indexed at all peers that manages nodes whose bit vectors are *covered by* the document's bit vector. An XPath query is processed in four phases. In phase 1, the query is routed to its anchor peer according to the bit vector of the query. In phase 2, the query is evaluated against all the synopses stored in its anchor peer and forwarded to the anchor peers of the matching synopses. In phase 3, the anchor peer of each related synopsis examines the query on the related bit maps and forwards the query to the related owner peers. Finally in phase 4, the owner peers evaluate the query on the XML documents and return answers to the querying peer. We also present a scheme that dynamically partitions the hypercube to balance the load across peers. We further exploit the *partition history* to remove redundant messages.

The work in this thesis have resulted in a number of publications and manuscript: [49], [50] and [48].

1.4 Thesis Organization

Hereby, we outline the organization of this thesis. The rest of the thesis contains 6 chapters. In Chapter 2, we first introduce the background knowledge on P2P networks and some related techniques on keyword search in traditional information retrieval systems. A survey on the related work is provided in Chapter 3, where we mainly focus on the existing works on keyword search and XML query processing in P2P network.

Chapter 4 proposes our solution, SPRITE, to build practical partial index. SPRITE selects and indexes representative terms in a structured network, and refines them according to the queries. We conduct experiments to show that SPRITE is nearly as effective as the centralized system, and considerably outperforms the static scheme.

In Chapter 5, we propose CYBER, which leverages on community-based feedback to improve search quality. Our comprehensive experimental results show that CYBER outperforms the scheme based on individual feedback techniques.

We then present the design and evaluation of XCube, a system to process XML queries in a P2P network in Chapter 6. In XCube, an XML document is indexed on all of its tags as a whole entry, and XPath queries are routed according to its tags as well. Our extensive experimental results show that XCube is more efficient than the scheme that indexes individual tags.

Finally, Chapter 7 concludes this thesis and discusses some directions for future work.

Chapter 2

Background

In this chapter, we introduce some fundamental overlay structures of P2P networks, which are employed in our proposed schemes or some closely related works. In addition, we also briefly review some background knowledge on keyword search over text data and XPath queries over XML data.

2.1 Peer-to-Peer Networks

Peer-to-Peer (P2P) systems are becoming the key paradigm in information sharing and retrieval today. In a P2P network, a number of computing peers construct a logical network, where the peers cooperate loosely to share resources and services. In this work, we mainly focus on keyword-based search, which requires a certain percentage of hard disk space, CPU and bandwidth sharing. Among these resources, bandwidth is the bottleneck because data indexing, monitoring, searching and downloading all require message transmission.

In a P2P network, messages are routed by following the overlay network and the indexing scheme (broadcast in case of no indices), so the routing efficiency highly depends on the structure of the overlay network. According to the structure that peers are organized in the network, we can classify P2P networks into unstructured P2P networks and structured P2P networks. Note that usually the index of a datum, instead of the datum itself, is stored in a remote peer, which is named as the indexing peer in this thesis. We focus on the search procedure among the indexing peers, cause the downloading procedure is done in a client-server architecture in all P2P networks. We now introduce the two categories of P2P networks with some representative overlay structures.

2.1.1 Unstructured P2P Networks

In an unstructured P2P network, peers join the network randomly. Each peer maintains several links pointing to a few neighbors. The neighbors are randomly selected and may be optimized according to additional information provided by users or obtained from other peers, such as the historical query results.

The straightforward searching strategy is flooding. Without any index built beforehand, a query is broadcast to all of the neighbors within a radius, which is usually controlled by a counter, Time To Live (TTL[1]). The receiving peers then decide whether to continue forwarding the message according to the TTL. Peers containing relevant answers will reply the querying peer. Gnutella [30] is a well known decentralized P2P application. The search scheme is a kind of Breadth First Search (BFS). It is fast in terms of response time, but costly in terms of routing hops. Usually, most of the peers in the searching scope do not contain any answer, so the overhead is very large. Moreover, the searching scope is always limited to a certain group of peers, thus only local optimal answers are found usually, instead of global optimal answers.

Many refined strategies are proposed on top of the basic BFS scheme. In [95], a

[1]The TTL is usually implemented as the number of hops to forward the message in a P2P network

small TTL is initialized when issuing a query. If the query results are insufficient, the TTL is increased and the search radius is enlarged. A query on popular data items may not be broadcast to too many peers, but there may be many duplicated messages for sending queries multiple times. In the k-walker strategy [55], a peer sends a query to a subset of neighbors rather than broadcast the query. If there are more replicas of a file in the network, then the query will have a higher chance to find relevant answers in a few hops. However, this strategy does not have any guarantee on the search results.

Routing Index [21] and Q-Routing [51] make use of historical metadata to guide the routing. Routing Index records the past query results from every neighbor on each topic. A query is only forwarded to the peers that may contain sufficient answers. Q-Routing maintains the routing cost, in terms of time, to retrieve each data item. A query is sent to the neighbor that can reach the answer peer in the shortest time.

In summary, the naive strategies are upgraded by maintaining more detailed and complex neighbor information. However, because the neighbors are loosely indexed and the restricted search scope, the mentioned systems cannot guarantee a query can find some answers or the query can find all existing (online) answers. The retrieval techniques that require certain global knowledge cannot be applied in this kind of networks either. Therefore, these strategies are only suitable for applications, in which the users only demand some answers without requirement on global ranking.

2.1.2 Structured P2P Networks

In structured P2P systems, the network structure is predefined. Both the scheme that a peer joins the network and the manner that data is indexed follow the network structure. Structured P2P networks are attracting the interests from many researchers for its bounded routing performance and guarantee on finding existing data. The advantages come with the price of the acceptable overhead on network construction & maintenance and index insertion & maintenance. More specifically, a message can be routed to its destination peer in $\log N$ hops on average, where N is the total number of peers in the network. An arbitrary peer needs to maintain links to $\log N$ remote peers on average. The precondition of the bounded routing cost and maintenance overhead is that peers are uniformly distributed in the predefined space. The uniform distribution is implemented with a consistent hash function usually[2]. Thus, such structured networks are generally called Distributed Hash Table (DHT) networks.

Many DHT networks have been proposed: Chord [81], CAN [63], Pastry [66], Symphony [56], HyperCup [69] and BATON [37]. Here, we illustrate DHT networks with two representative examples, Chord [81] and HyperCuP [69], because they are employed in our proposed schemes: Chord is employed in SPRITE (Chapter 4) and CYBER (Chapter 5); and HyperCuP is altered and employed in XCube (Chapter 6). However, as our proposed schemes mainly exploit the common *lookup* interface of the DHT networks [23], they can be easily substituted with another DHT network.

Chord

Chord [81] is one of the most well known DHT overlay. Chord defines a universal space as a ring with 2^m identifiers. A peer obtains its identifier (ID) by hashing sustainable object, such as its IP address. A peer is responsible for the segment whose Chord ID locates between its ID and its clockwise predecessor's ID. Data

[2]The consistent hash functions in data encryption, such as SHA-1 and MD5 are employed.

items are hashed using the same hash function (SHA-1 is used in Chord), thus the length of a Chord ID is 160 bits (m=160). Every peer manages the indices of the data items whose hash values fall in its responsible segment. If the ID of a new peer is hashed to the segment managed by an existing peer, the segment is split and each peer is assigned with a new, smaller segment.

In Chord, every peer needs to maintain two sets of links pointing to some remote peers: a small number of successor links[3] to ensure the ring is always close and m finger links to achieve efficient routing performance. A peer periodically checks the availability of its successors. When all of the successors fail in a short period, the ring is not closed, which rarely happens. The finger table is built up in a manner analogous to binary-search-tree (BST). The 2^m identifiers are halved recursively with respect to the Chord ID of the peer who is building the finger table. The peer maintains a finger pointing to the peer who is responsible for the splitting point.

Chord can route point queries very efficiently with the successor links and finger links. Given a point query, Chord first obtains its hash value based on the same hash function that is used to generate Chord IDs. The query and its hash value are encapsulated in a routing message. When a peer receives the message, it lookups the peer (from its finger table) that is the nearest to the destination point, and then forwards the message to the peer. Such forwarding process halts until the message is sent to the destination peer. The routing is performed in a binary search manner because of the BST-like finger table.

It has been proven experimentally and theoretically in [81] that routing a message to an arbitrary peer costs $\log N$ hops on average, where N is the total number of peers in the network. Because many fingers point to the same peer, the average number of effective fingers is $\log N$ too. The routing performance degrades slightly

[3]We consider the predecessor link as a special successor link counter-clockwise.

when a small fraction of pointers in the finger table and successor list are out of date. It is worth noting the key assumption to achieve the average $\log N$ routing hops is the uniformity of peer distribution. In the worst case, the routing hops from one peer to another is m rather than $\log N$ ($m > \log N$). This assumption is shared by the other DHT networks as well.

HyperCuP

In HyperCuP [69], peers are organized in a hypercube graph. In a d-dimensional hypercube, there are 2^d nodes (vertices)[4]. Each hypercube node can be represented as a bit vector. Every hypercube node has one adjacent neighbor node in an arbitrary dimension by altering the corresponding bit in the vector.

HyperCuP is originally designed to perform broadcast efficiently, so all dimensions follow a certain order. An existing hypercube with d dimensions is "unfolded" when a new peer joins the network, i.e. a new dimension is created, if all of the 2^d nodes are assigned to the existing peers. The new nodes (except for the one that is assigned to the newly joined peer) are assigned to the corresponding existing peers. In this manner, dimensions are sorted according to the order the hypercube is "unfolded". In order to broadcast a message, peers forward the message in the dimensions that is subsequent to the dimension in which the message is received. Therefore, each peer receives a broadcast message exactly once. Moreover, the longest distance in the broadcast process is d (Each forwarding is equivalent to altering 1 bit in the bit vector, so after d bits are altered, the message reaches the destination peer.). The search algorithm is basically a broadcast controlled with a time-to-live token.

However, it is easy to see that the above structure can be changed to route a

[4]For the sake of simplicity, we only study hypercubes with base 2 (2 nodes in each dimension). In [70], an extension on hypercubes with a base greater than 2 is presented

message within $\log N$ hops, where N is the number of peers. We can predefine the dimensionality of the hypercube (similar to the length of Chord ID). An data item can be hashed with a consistent hash function, such as SHA-1. The hash value can be represented as a binary number, which can be mapped to a bit vector. When a new peer joins the network, the hypercube nodes of an existing peer are halved in a dimension randomly, and each of them are responsible for a number of hypercube nodes, which construct a sub-hypercube. When routing a message, a peer forwards the message to a neighbor peer, who is responsible for a node with more similar bit vector (with more matching bits).

2.2 Keyword Search

In this section, we introduce some traditional keyword search strategies, which include the model to calculate the similarity between a query and a document, the method to calculate the weight of a term/keyword, and the mechanism to improve the quality of search results. These strategies are related to our proposed solutions or other existing methods.

2.2.1 Vector Space Model and $TF{\cdot}IDF$

The Vector Space Model (VSM) has been well studied. In VSM, every document is mapped to a point in a vector space based on the weights of the terms it contains. Analogously, a query is mapped to the vector space based on the keywords appearing in it. By calculating the similarity between the two points, we can obtain the similarity between the query and the document. Usually, the cosine similarity function is employed as the distance function. Finally, all documents are sorted according to the similarities in descending order to generate the ranked list.

In traditional IR techniques, every term in a document is assigned a certain weight based on some statistics. One of the most popular formulas is $TF \cdot IDF$. The weight of term k in document i is:

$$w_{ik} = tf_{ik} \times idf_k.$$

Here, tf_{ik} is the frequency of term k in document i and idf_k is the invert document frequency of term k in the entire document repository. The intuitive meaning of this formula is that a term is important to a document in the repository if (i) it occurs frequently in the document, and (ii) it appears infrequently in the repository.

More specifically, tf_{ik} is the normalized term frequency, by either the document length or the maximum term frequency in the document. While idf_k is more complicated:

$$idf_k = \log \frac{N}{n_k}.$$

Here, N is the total number of documents in the repository; and n_k is the number of documents containing term k, which is called *document frequency* of term k.

Given the term weights, we can now calculate the cosine similarity between a query and a document:

$$sim(Q, D_i) = \frac{\sum_{j=1}^{n} w_{Q,j} \times w_{i,j}}{\sqrt{\sum_j w_{Q,j}^2 \cdot \sum_i w_{i,j}^2}}$$

where $w_{Q,j}$ is the weight of the jth term in query Q, and $w_{i,j}$ is the weight of the jth term in document D_i.

To facilitate keyword search in a P2P network, two issues are closely related in VSM. One issue is an inverted index is built to improve the searching performance in a centralized system. When calculating the dissimilarity between a query and a document, only terms appearing in the query are checked. In order to avoid checking irrelevant terms, a distributed inverted index should be built in a P2P network. The other issue is how to calculate the weight of a term in a document, as both N and n_k are global information that are not readily available in a P2P network. Besides the two issues, some observations that motivate our solution are further discussed in Chapter 4.

2.2.2 Relevance Feedback

Relevance feedback is a general technique to improve search quality. We present the technique with keyword based search as the sample application. In the $TF{\cdot}IDF$ scheme, terms are weighted solely based on repository. As a retrieval system involves user interactions with the system, relevance feedback is proposed to refine the queries, more specifically, to tune the term weights in queries. After a user issues a query, an initial ranked list is returned first. The user then selects some results as relevant answers. The terms weights in the query is refined according to the user selections:

$$Q' = Q + \alpha{\cdot}\textstyle\sum_{D_i \in R} D_i \text{ - } \beta{\cdot}\textstyle\sum_{D_j \notin R} D_j$$

Here, Q is the original term vector of the query and Q' is the refined term vector. α and β are some tunable parameters. R is the set of relevant answers in the returned list. Intuitively, the terms appearing in the relevant documents should be assigned with larger weights, while the terms appearing in the irrelevant documents should be assigned with smaller weights. In this way, relevant documents and irrelevant documents are better separated regarding to the query. Usually, only relevant documents in the returned list are used to refine the query.

In order to obtain more accurate results, users are expected to participate the feedback process. Apparently, the overhead is the longer response time. We will see in Chapter 5, how CYBER avoids such explicit user involvement in a P2P network while improving the search quality.

2.3 XPath Queries

The *eXtensible Markup Language* XML [24] has been widely used to represent and exchange data. XML is self-describing (user-readable), text-native (machine readable) and extensible. In a P2P network, users have little knowledge on remote data, use different platforms and softwares and need to describe data in their own ways. Because of the P2P user demands and the XML's properties, XML is becoming an ideal data format naturally in P2P networks. Here, we introduce XPath queries [91], the fundamental query language for XML data.

An XPath query mainly contains two types of constraints: structural constraints and attributive constraints. The structural constraints examine if the structure of an XML document matches the structure specified in a query. Such constraints concentrate on the element relationships and existences. The element relationships include: parent-child relationship, ancestor-descendant relationship and sibling relationship. The attributive constraints examine if the values of some attributes or the content of some elements satisfy some conditions. Consider the sample XPath query Q_{xp} below.

Q_{xp}: //author[conference="VLDB"][@year=2008]/name

It is looking for the names of authors who publish some papers in VLDB'08. The structural constraint for Q_{xp} is //author[conference][@year]/name; while the attributive constraints include (conference="VLDB") and (@year=2008). We will present how XCube in Chapter 6, processes XPath queries in a P2P network.

Chapter 3

Related Work

In this Chapter, we discuss the existing work on keyword search in P2P systems. We first review the schemes on supporting document retrieval in P2P networks. Then, we present how personalized search and relevance feedback techniques are exploited to improve the search performance in the existing work. Finally, we discuss the mechanisms of processing XML queries in P2P networks.

3.1 Document Retrieval in P2P Networks

In structured P2P networks [81, 63], including the "loosely structured" networks[1] [9], search on file names can be easily handled. Moreover, the *lookup* function guarantees that a term can be found in $\log N$ hops, where N is the number of peers in the network. A file name can be treated as an integrated entry or a set of terms, and hashed if necessary, and then indexed in the network. However, indexing file content involves more challenging issues. Many of these have been addressed in [46], in which two major concerns are discussed: storage constraints and communication

[1] In a DHT network, peers build their routing tables strictly following a predefined manner; while in a "loosely structured" network, the routing tables are built based on some probabilities.

constraints. Both of these are caused by the large number of terms in a document to be indexed.

To the best of our knowledge, the most similar work to our SPRITE is eSearch [83]. In eSearch, a document is indexed on the top k terms and the complete inverted list of the document is replicated and stored in k indexing peers. In the description of top term selection, the authors assume that some global statistics can be obtained. However, global statistics are expensive to obtain and tend to be inaccurate in a P2P network, where peers frequently join and leave the network, and documents are shared and unshared frequently as well. In SPRITE, we do not make this assumption. Term expansion is employed in eSearch. This is orthogonal to the basic scheme, and not discussed further in this thesis, though term expansion could also be used with SPRITE.

In [53], Lu and Callan proposed a scheme to process content-based retrieval in hybrid P2P networks. In the hybrid network, a superpeer is responsible for summarizing the contents among its normal peers. The summaries are defined as "resource descriptions". Queries are routed according to the "resource descriptions": a query is forwarded to the peers containing the relevant resources with some probability above a threshold. KSS [29] divides predefined queries into a set of combinations. Each element in the set is hashed and indexed in a structured DHT. The query term space can be very large and the combination is too complex to forecast. Besides addressing some challenges of keyword search in P2P systems, Li et al. [46] proposed to combine some techniques (e.g., caching and query compression) to reduce communication cost. In [64], bloom filter is employed to compress the message size. Works based on latent semantic indexing (LSI), such as *pSearch* [85, 84], predefines the term spaces. A global knowledge is assumed to compress documents with LSI into fewer dimensions. The indexes are rotated several times and a set of important indexes are placed into an overlay of CAN [63] each time. A query is preprocessed similarly and answered as a KNN search in the CAN space.

Podnar et al. present an indexing/retrieval model with highly discriminative keys stored in a distributed global index [60]. Their experiments show reduced total traffic compared with distributed single-term strategies, and the retrieval performance is also good. The authors also refine the work by introducing a querying-driven indexing scheme later in [80].Chen et al. propose a scheme based on Bloom Filter to reduce the message size when processing queries in [18]. The peers that index the query terms are visited sequentially. In a query message, only the metadata of the documents that potentially contain all keywords are encapsulated. However, the supported queries are *AND*-based and *OR*-based only. The cost for similarity-based queries are still very high.

Papapetrou et al. propose a technique to eliminate replicated documents shared in a P2P network in [59]. In [59], Global Document Occurrence is employed to reduce the importance of the replicated documents, so that a final ranked list contains few replicated answers.

Traditional distributed information retrieval has been extensively studied [14, 77, 76, 58]. In a traditional distributed environment, servers are organized statically, so the methods are not applicable in dynamic P2P networks and beyond our scope.

3.2 Social Networks and Personalized Search

In recent years, many studies on social search techniques have been carried out. As stated by Watts et al., social networks have the surprising property of being "searchable" [39]. A social search engine is a certain type of search engine that determines the relevance of the search results by taking into consideration the interactions of users. The main techniques involved in social search engine include *recommendation*, *relevance feedback*, and *personalization* etc. In many existing works, these techniques are often combined to achieve better performance.

An original work on social recommendation is Ringo [72], a social music recommendation system which employs the social filtering technique to offer music among users of similar tastes. Social filtering in a centralized manner has been well understood and the similar idea has been in use by popular Web sites such as Amazon and eBay, but it cannot be directly applied in a P2P environment as its computation requires global knowledge.

Shen et al. studied the method to infer a user's interest from the user's search context, and proposed a framework for implicit user modeling[73]. Unfortunately, this framework is implemented on client-side search agent, and therefore cannot be used for P2P environment.

In [57] Mislove et al. proposed a Web search framework enhanced by social networks, and study the mechanisms for content publishing and location in social networks. By using cached results from a connected group of individuals during their search, the framework led to considerable improvement in search effectiveness. Beydoun et al. presented a "semantic annotation approach" to support search in a social network [11]. In a P2P search environment, we can also adopt a personalization scheme to suggest results (or change the ranking of the results) based on previous user feedback. However, a P2P model that utilizes such a scheme among a group of socialized users has never been reported.

Löser et al. proposed a semantic social routing mechanism, called INGA, based on an unstructured overlay network [52]. INGA treats each peer in the network as a person in a social network. Each peer in the network maintains local "topical knowledge" and determines the relevance of a remote peer to a query using a *personal semantic shortcut index.* Routing of queries can be based on a shortcut selection function being able to identify and group peers with similar interests. This work differs from our proposed CYBER model in two ways. First, it exploits social connections explicitly as routing index, but CYBER treats the socially similar users implicitly as profile vectors. Since INGA has to rely heavily on the generated social shortcuts to route queries, a TTL limits the length of the "social path" that a query can follow. In contrast, CYBER does not have that limitation. Second, INGA is proposed for an unstructured overlay, while CYBER works for a structured one.

3.3 XML Query Processing in P2P Networks

There are mainly two broad categories of mechanisms to search XML data in P2P networks. The first category is based on unstructured overlay networks. The key idea here is to cluster peers with similar XML documents close to one another (based on some similarity measurement in content or structure). Like the routing index [21], once a query is routed to a peer containing (potentially) relevant documents, it can be expected that the cluster around this peer will also hold relevant answers, and hence broadcasting the query within the cluster and the clusters close to it will provide better search performance.

In [42], multi-level bloom filters are used to calculate the structural similarity between XML documents. Peers in the network are organized in a hierarchical manner according to structural similarity. Queries are forwarded to superpeers in upper levels until the most similar bloom filters are found. The superpeers then forward the queries to the related peers downwards. However, this method requires a larger number of powerful/capable superpeers in the hierarchical network. This requirement is much more difficult to comply with than the superpeer network presented in [96] and thus limits its scalability. The failure of a peer in the hierarchical structure can disconnect its entire subtree, so queries cannot be sent upwards and data in the subtree cannot be found. XPeer [68] clusters peers according to a *schema-similarity*. Superpeers are employed to route the queries. A query is sent to superpeers, who search for the related peers. However, only a framework is proposed without technical details. In [26], each peer maintains two inverted indices: a local index on its own XML documents and a peer index on others' XML documents. Queries are routed based on the peer index in a similar fashion as in Routing Index [21]. The peer index cannot be too large because of the high storage and update cost. This method is not scalable and may miss many answers. The advantage of mechanisms in this category is that network construction and maintenance costs are low. However, the disadvantages are more dominating: existing answers are not guaranteed to be found, and routing cost is high because of broadcast.

The second category indexes XML documents in structured overlay networks. This category can be further classified into three approaches according to the indices. One extreme approach, the content-based approach, indexes all terms in XML documents. A CAN-like [63] mechanism is proposed in [86] to process XML queries in the granularity of XML elements. A Cartesian space is predefined with all possible paths. Every path corresponds to a dimension. Every XML element is mapped to a CAN *region*. An important assumption is that all users must adopt the same schema, which is very hard to achieve in a P2P network. The other limitation is the number of XML elements can be very large. Monitoring the indices is too expensive in a P2P network.

Another extreme approach is the structural approach that indexes complex structures, more specifically, paths in XML documents. In XP2P [12], simple path queries are supported in a DHT network. An XML document is partitioned into multiple path-based fragments and each fragment is indexed according to its path. A query is routed heuristically by shrinking the query path (one tag each time from the leaf node) until all matching paths are found. It is very expensive to find all matching documents in this way. Another drawback of XP2P is that users have to know exactly how others fragment their XML documents and issue queries based on the fragmentations. In [79], all distinct paths rooted at different levels for every XML element are indexed in a P-Grid overlay network [1]. An XPath query is partitioned into sub-paths delimited with "//". All peers responsible for the sub-paths are visited to fetch the metadata of related XML documents. The final results are generated in the querying peer. The drawback of this method is the number of paths to index for a document can be very large, which increases exponentially with the document structure size (i.e., the number of distinct tags).

The third approach, the tag-based approach, indexes the tags in XML documents. The tag-based approach incurs low index maintenance cost and users can issue queries without knowing the remote peer schemas. In [25], XML documents are indexed using an inverted file approach. However, instead of indexing terms, only tags are indexed. We refer to this scheme as the *inverted-file tag-based* (IFT) scheme. Each tag of a document is hashed and assigned to an indexing peer in a DHT network. Along with a tag name, the document URI, paths reachable to the tag, and data summary (such as histograms for numerical data) are attached to the index entry. When processing a query, the set of related peers are searched in sequence. Initially, the related peer set is large after visiting the first peer. In the subsequent hops, potential document structures are reconstructed by combining document URIs and paths of the checked tags. Many peers are pruned from the related peer set if their documents lack some paths in the query or they do not satisfy some constraints according to the data summary. After all tags in the query are checked, the query is sent to all the remaining relevant peers for final processing. A key drawback of this method is that some element/attribute names are very popular in the system and frequently queried. In such cases, peers responsible for these "hot" names will be easily overloaded. Unbalanced load can cause network instability as overloaded peers have strong incentives to leave or rejoin the network and thus affect data availability. The failure of a peer responsible for some popular tags has a serious impact on the whole system as some frequently accessed information would be missing. The system performance is also degraded because the overloaded peers can delay forwarding many messages. Though a splitting technique is proposed to balance the load by distributing the load of popular tags to some closely related tags, it is not sufficiently autonomous and heavy administrative work is assigned to expert users. This may not be achievable in many real applications, and hence limits the applicability of the scheme. Another IFT-based approach, KaDoP [2], also indexes individual tags in a DHT network. Besides the tag indices, documents with similar contents are linked together so that the queries can be easily extended. However, the extension comes at the expense of user effort: the links between documents have to be identified manually. This requirement limits the scalability of the scheme.

Wu et al. propose a just-in-time technique to index data in a structured P2P network [89]. Based on the cost to route queries, popular values are indexed as data points while unpopular values are indexed as data ranges. This technique can be applied to index XML data in a P2P network where a common schema is employed by all users. However, there are many different schemas in a P2P network usually.

In [38], the hypercube structure is used to support keyword search in document retrieval. A hypercube node can be identified by a d-bit vector. Similarly, a document can also be mapped into a d-bit vector derived from its terms. In this way, the document can be indexed at the node with the matching bit vector. A keyword search is performed by first mapping the query keywords to a d-bit query vector Q in the same manner as a document is mapped. Clearly, to locate the documents that match the query exactly, we only need to search the node with identifier Q. In addition, to find documents that contain the query keywords, we can search every node whose identifier V_N *covers* V_Q (V_N covers V_Q if for every bit in V_Q that is set to 1 the corresponding bit in V_N is also set to 1). While this scheme is simple, it has two limitations: (a) the number of terms in a document is large and hence the dimension has to be very large for effective performance; (b) containment search is inefficient as the number of nodes to be searched is large; moreover, many of these may not contain relevant documents.

3.4 Load Balancing in Structured P2P Networks

Load balancing is one of the key criteria for a P2P network. Unbalanced load violates the spirit of P2P network and can incur many problems of a client-server structured network. In the DHT networks, storage load is uniformly distributed across peers by hashing data items. However, some peers still receive $O(\log N)$ times as much load as the average peers [13]. Overloaded peers tend to leave or rejoin the network to avoid the heavy load, which renders the entire network unstable and increase the maintenance overhead. Such overloaded peers are the hot spots, and thus the vulnerable points of the network. We review load balancing techniques in this section.

In [82], Godfrey et al. employ the concept of virtual nodes and extend their static scheme presented in [62] to balance the load dynamically. Every peer is responsible for a number of virtual nodes. Some heavily-loaded peers transfer some virtual nodes (including its load) to some lightly-loaded peers. The lightly-loaded peers actively search for heavily-loaded peers; or the other way around. A "many-to-many" scheme is also proposed to balance the load from the entire network point of view, where a number of directory peers are employed to gather the information of lightly-loaded peers and heavily-loaded peers. The overhead is the maintenance cost of the network structure for all of the virtual nodes (instead of one peer).

In [13], Byers et al. propose to balance the load with the heuristic paradigm, "power of two choices". When indexing an item, it is hashed d times ($d > 2$) and d peers are contacted initially. Among the potential indexing peers, the peer with the lowest load is chosen to index the item. When processing a search request, the search key is hashed d times and d related peers are contacted, which introduces a large overhead (d-1 times more routing messages). In order to avoid this problem, a *redirection pointer* linking to the actual indexing peer is maintained by every related peer when indexing the item.

In [27], Ganesan et at. employ two fundamental operations: NBRADJUST and REORDER to balance the load. While NBRADJUST adjusts load with directly linked peers only, REORDER transfers load from overloaded peers to underloaded peers globally. By combining the two operations, the load is evenly distributed among all peers. In order to obtain the peer with the heaviest load, all peers must be sorted according to their load. Therefore, a separate skip graph is built on the load, which is the major overhead of this scheme.

In [8], Aspnes et al. decouple a structured network into two layers: routing layer and bucket layer. The routing layer is responsible for search and network maintenance by following the original routing protocol. The bucket layer stores data/index entires with buckets. A number of similar keys are placed in a bucket. Each peer is responsible for 2 or 3 buckets, so that keys in heavily-loaded buckets can be shed to lightly-loaded buckets. When all buckets are heavily-loaded, the peer and its neighbor peers reconstruct buckets with all keys they are responsible fore.

Replication-based load balancing algorithms are proposed in [94] and [65]. In [94], the requested data is replicated in some peers in the access path. Roussopoulos and Baker focus on balancing the load of downloading data in [65]. Peers announce their maximum capacity that they can provide. A peer forwards a request to a peer with probability proportional to its maximum capacity.

In summary, all of the above algorithms are designed for general structured P2P networks. Therefore, we can easily adopt them in our work since SPRITE, CYBER and XCube are all based on some DHT networks. We will present how to apply a proper load balancing algorithm in the corresponding chapters.

Chapter 4

SPRITE: ***S***elective ***PR***ogressive ***I***ndex ***T***uning by ***E***xamples

4.1 Introduction

The goal of the thesis is to design efficient and effective schemes to retrieve data through keyword search. As presented in the introduction and literature review, existing techniques are limited. In particular, for DHT-based techniques, the index construction and maintenance overhead are not acceptable. In this chapter, we propose SPRITE (*S*elective *PR*ogressive *I*ndex *T*uning by *E*xamples) to bring down the cost of index construction and maintenance. Our proposed solution is motivated by three observations. First, a document will most often be queried using a small number of terms that characterize it. It may suffice to index a document on only these characteristic terms, and drop all others. In fact, it has been argued in [83] that if a query term p is not among the top frequent terms of a document, then adding p to the query is unlikely to materially affect the ranking of this document. Second, a term that is not used in a query has no effect on the ranking of the documents. If we can know which terms will be used in queries that seek a particular document, then we should index only those terms for the document: all other terms merely increase the index size without providing any additional accuracy. Third, users with similar interests are likely to retrieve a similar collection of documents with a similar set of queries that share some common keywords. Such a *query locality* phenomenon is not uncommon in search engine queries - analysis of Excite search engine trace [90] and Altavista search engine trace [78] showed that queries submitted to these search engines not only have significant locality, many are repeatedly issued by either the same or other users, and that multiple-word queries are common.

We note that the first and second observations suggest that it may suffice to index only a small well-chosen set of representative terms in each document. The second and third observations also hint that the query keywords may potentially contribute to the set of representative terms. Furthermore, the third observation suggests that it may be possible to learn from past queries - since similar queries share certain common keywords, past queries may be used to refine the selected representative terms.

As we shall see, our proposed algorithm SPRITE ensures that a small set of representative terms are well-chosen. SPRITE also progressively learns from (past) queries to refine the set of chosen indexing terms. In this way, new terms may be injected into the system, while "obsolete" terms (as a result of changing access patterns) may be removed/replaced.

The rest of this chapter is organized as follows: In Section 4.2, an overview of the SPRITE architecture is described. Section 4.3 discusses how queries are processed in SPRITE. We also discuss how to integrate the text retrieval task with the overlay network routing protocols, using Chord[81] as a specific example. Whereas we have used Chord in all our examples and in our implementation, there is nothing in our central idea that depends on Chord, and the reader should be able to see how to make the necessary adaptation to a different overlay network.

In Section 4.4, we present the scheme to select and refine indexing terms. We have implemented the proposed strategy, and compare its retrieval effectiveness (in terms of both precision and recall) against a static scheme (without learning) and an ideal centralized system. Our experimental results, presented in Section 4.5, show that SPRITE is nearly as effective as the centralized system, and outperforms the static scheme. Finally, we make some concluding remarks in Section 4.6.

4.2 Overview of SPRITE

The SPRITE system comprises a large number of computers (peers) that are organized into a structured overlay network, such as Chord, that is capable of supporting simple indexing through a distributed hash table. Each peer plays two roles: owner peer and indexing peer. An *owner peer* owns and shares certain documents. It is responsible for maintaining each shared document it owns, locally indexing it, and selecting the *global index terms* (A global index term is a document term to be injected into SPRITE to facilitate query searching.) for it. An *indexing peer* is responsible for managing meta-data for terms assigned to it. This meta-data is primarily an inverted index of the (global index) terms managed by the peer. The information maintained in the inverted list include the documents containing the term and their respective owner peers. In addition, each indexing peer also maintains a history of past queries (rather, the keywords corresponding to the queries). To reduce the storage, each indexing peer maintains only the most recently issued queries.

There are two main services supported by SPRITE. First, a peer can share a new document with other users. In this case, the document owner has to select and publish corresponding global index terms into the SPRITE system. Second, a peer can submit a query to retrieve relevant documents through keyword search. While the query processing service is straightforward, the document sharing service is challenging. As noted above, it is too expensive to publish all the terms (even after stemming and stop-words elimination) in a document. Moreover, based on the observations in the introduction, we believe it would suffice to index only a small well-chosen set of representative terms. Thus, SPRITE publishes only a small subset of representative terms that are subsequently refined based on past queries.

More formally, let the set of documents in the network be D and the set of queries be Q, over all time. Suppose document d_i is determined to be relevant to queries q_{i1}, q_{i2}, ... q_{ik}. Let the union of the keywords in the queries be K_i. In the ideal case (with perfect knowledge into the future), document d_i is only indexed on the keywords in K_i. SPRITE attempts to do exactly this with limited knowledge: it learns a set of keywords, K'_i, which approximates K_i. Terms in $K'_i - K_i$ are indexed unnecessarily; terms in $K_i - K'_i$ may cause document d_i to be misjudged as irrelevant to some query q_{ij}. Choosing K'_i wisely is at the heart of SPRITE.

Towards this end, for each document, SPRITE begins with an initial guess at the important terms. This guess can be based on user input or through automatic selection of high frequency terms in the document, or a combination of such techniques. For ease of presentation, in this thesis, we shall simply pick the most frequent terms as the initial global index terms. Next, with these terms, SPRITE examines past queries that have queried these terms. (Recall that the queries are stored at indexing peers. As such, they can be obtained from the indexing peers.)

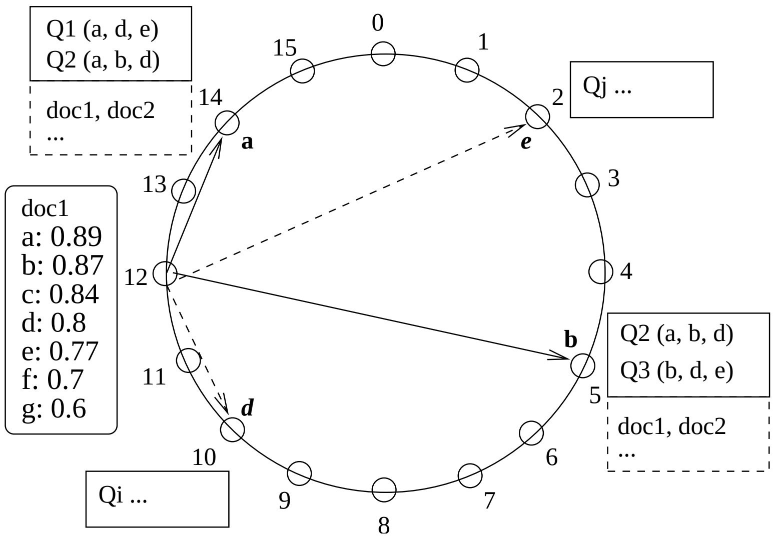


Figure 4.1: Indexing terms in a Chord Ring.

Based on these queries, SPRITE identifies a new set of terms to be indexed, augmenting and replacing the initial set of index terms. This process of examining past queries, and refining the indexing terms, is repeated periodically.

When an owner peer of a document D wants to update the indexed terms of D, it polls the indexing peers with an *index update message* that contains all the global index terms of D. It is possible that a past query contains multiple global index terms of D and thus is cached by multiple indexing peers. Apparently, it involves much redundancy if such a query is sent to the owner peer by all related indexing peers. In SPRITE, every cached query is hashed also, which can be precomputed offline in fact. The *closest* term to the query can be identified among all global index terms by comparing the hash values. Only the indexing peer responsible for the closest term sends the query back. In this way, we avoid sending the same query multiple times. Note that the number of global index terms is much smaller than the number of past queries cached in the indexing peers. Therefore, the redundancy above can be removed effectively.

Figure 4.1 illustrates an example with peer 12 as the owner of document *doc*1.

Suppose two terms a and b are selected as the most important terms in $doc1$ to be indexed initially. These terms are published to the appropriate indexing peers, say peer 14 and peer 5 for a and b respectively. Now suppose Peer 14 receives two queries, $Q1$ and $Q2$, on term a, and peer 5 has two queries, $Q3$ and $Q4$, on term b. In the next learning period, peer 12 sends messages to peer 14 and peer 5 for past queries on terms a and b respectively. Upon receiving the four queries, peer 12 calculates the similarity between the queries and document $doc1$ and then chooses another set of terms to be published further. In this example, terms d and e are chosen and added into the index. It is worth noting that even though term c has a higher rank (more frequent) than d and e for $doc1$, yet it is not indexed because it has not been used in any query for $doc1$ thus far. One may worry that c may have been specified as a search term in many queries, none of which returned $doc1$, and regarding which peer 12 is thus completely unaware. However, we note that peer 12 can be unaware of such queries only if they do not involve any of $\{a, b, d, e\}$. $doc1$ will not be relevant to any such query with high probability, since it only specifies one of multiple frequent terms in $doc1$.

Next, let's look at the information retrieval service. A query is processed in searching and retrieval phases. The searching phase is more complicated and important since it decides the quality of the answers. Given a query, all indexing peers responsible for the query terms are visited, and the related indices are obtained by the querying peer. Besides the term frequency, the document length and the counted document frequency are also returned along with each index entry. The term frequency and document length can be combined as a normalized term frequency. Next, at the querying peer, index entries for the same document are consolidated and used to calculate the similarity between the document and the query. Finally a ranked list is constructed and a desired number of documents are returned to users as answers. The retrieval phase is simply a downloading action to the relevant documents, so we do not discuss it further in this thesis.

Note that we do not have the precise document frequency of a term (i.e., the number of documents containing the term). Instead we use as surrogate the *indexed document frequency*, which is the number of documents for which this term has been chosen as a global index term. The difference between these two frequencies is the set of documents in which the term occurs but has not been chosen for the global index. The indexed document frequency for each term is easily available at its indexing peer. Semantically, one can see that indexed document frequency serves the same purpose as, and can even be argued to be more appropriate than, regular document frequency. This intuition is borne out by the retrieval quality results we present in Section 4.5.

4.3 Query Processing

Consider a query peer that issues a keyword search, say comprising n terms. The query peer first hashes on each keyword to determine the indexing peer responsible, and retrieves the corresponding inverted list entries. Using these, it can determine the similarity between the query and potentially relevant documents.

In traditional IR techniques, every term in a document is assigned a certain weight based on some statistics. One of the most popular formulas is $TF{\cdot}IDF$. The weight of term k in document i is:

$$w_{ik} = tf_{ik} \times \log \tfrac{N}{n_k}.$$

Here, tf_{ik} is the frequency of term k in document i normalized by the document length, N is the total number of documents in the entire corpus and n_k is the document frequency, or number of documents containing term k.

In a structured P2P network, tf_{ik} is available as part of the metadata in the inverted list. The number of documents containing term k, n'_k, can be counted by the querying peer once the list is retrieved. However, this *indexed document frequency* is smaller than n_k in the case of SPRITE because the term may appear in some documents but is not selected as a global index term because it is lowly ranked among other terms in these documents. N, unfortunately, cannot be accurately determined in a P2P context: peers join and leave the network and documents may be shared and unshared at will. However, N is usually much larger than n_k, except for the terms in the stop word list, which are filtered away anyway. As long as N is the same for all the peers in calculating term weights, only the absolute IDF values will be affected and so does the similarity. Thus, it will not affect the relative positions of documents in the final ranked list. Therefore, we can simply use a sufficiently large N.

Given the individual term weights, we use the similarity formula proposed in [43] (the second method):

$$sim(Q, D_i) = \frac{\sum_{j=1}^{n} w_{Q,j} \times w_{i,j}}{\sqrt{number\ of\ terms\ in\ D_i}}$$

where $w_{Q,j}$ is the weight of the jth term in query Q, and $w_{i,j}$ is the weight of the jth term in document D_i. Note that the number of terms in D_i is available in the metadata of the inverted list retrieved. This formula simplifies the normalization (compared to the original similarity formula) and reduces the computation cost. Its performance is shown to be almost the same as the original formula in [43].

A document D_i containing a specified query term t_j may not have chosen t_j to be a global index term. In this case, w_{ij} is erroneously assumed to be zero rather than positive, and the value of $sim(Q, D_i)$ computed is decreased. In the next section, we will show how to choose index terms such that if the true value of w_{ij} is large, then t_j is chosen as a global index term for document D_i with high probability. If the true value of w_{ij} is small, then approximating it to zero introduces only a small error in the score computation and may make no difference to whether D_i is included in the ranked list for Q.

Before leaving this section, we mention an alternative approach to compute the similarity between a query and a document. Instead of the querying peer performing the computation, we can push the task to the indexing peers. This approach is adopted in [83]. Here, for each term of a document indexed, all the terms of the entire document are also stored as meta-data. In this way, the indexing peers can determine the similarity between a keyword and the documents containing the term to produce the ranked list. However, the indexing peers have to return their locally produced ranked lists to the querying peer eventually and the querying peer needs to merge the ranked lists into one. Many similarity calculations and ranked list sorting are performed repeatedly and redundantly. Therefore, we choose to assign the entire task to the querying peers.

4.4 Index Construction and Tuning

When an owner peer shares a document D, it indexes some representative terms in the system. This involves two stages. First, some initial terms in D are chosen and injected into the system. Next, the second stage is performed periodically to tune the index progressively. Essentially, at each run, more terms are selected from D based on the historical queries. The index terms for D are then refined by inserting new terms and removing noisy terms. To control the number of terms to be maintained, we limit the maximum number of terms to be indexed to a small value (say, 30). We will present the two steps below. Before that, we describe the metadata maintained at each peer.

4.4.1 Metadata in SPRITE

Recall that in SPRITE, each peer plays two roles: owner peer and indexing peer. Every indexing peer maintains two types of information: (a) A number of terms and the corresponding inverted lists, i.e. the documents that contain those terms. For each indexed term, the indexing peer also needs to store the owner peer's IP address, the owner document ID, the term frequency in the document and the document length. These metadata are used in query processing. (b) A set of queries, σ. Each query essentially comprises a set of keywords. Note that a query is only maintained at peers whose indexing terms contain at least one query term. These queries are used in the learning process.

At every owner peer, for each term in a document, two values are stored: (1) $qScore$, the similarity between the document and the most similar historical query (maintained at an indexing peer) containing this term (to be discussed shortly); and (2) QF (query frequency), the number of historical queries containing this term.

4.4.2 Initial term selection

When an owner peer first shares a document, we need to select an initial representative set of global index terms. The initial important terms of a document can be selected systematically or input by users. As a first cut, we adopt the following approach. First, we summarize the terms in a document and filter them with a stop-word-list to remove frequent but meaningless terms, such as "the" and "is". Second, we apply the stemming algorithm to unify terms by removing the suffix, such as "ed" and "ing". These two methods are well studied in the text retrieval community. The top F most frequent terms are then chosen as the initial terms. Note that at this point, only local information is available, so initial term selection

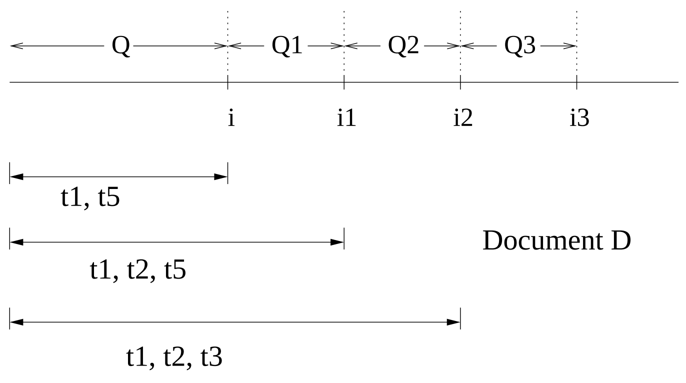


Figure 4.2: The learning phase in SPRITE.

solely relies on term frequency in the owner peer.

4.4.3 Tuning indexing terms

The learning stage is invoked periodically. We shall first present the basic idea with a naive implementation, and then discuss an efficient scheme. We use Figure 4.2 to illustrate the learning stage. In the figure, we show several iterations of learning: at iteration i, an owner peer bases its learning on the historical query set Q; at iteration i_1, the peer learns from a larger set of queries, $Q \cup Q1$; and so on. At iteration i, it is able to identify two terms, say t_1 and t_5. At iteration i_1, it identifies two new terms t_2 and t_6. However, suppose that we are limited to indexing only 3 terms, and it turns out that t_6 is the lowest ranked among the 4 terms, thus t_6 is removed. In iteration i_2, a new term t_3 replaces an obsolete term t_5.

The query set used for learning is determined by the current set of indexed terms. Essentially, for each indexing term, the indexing peer is polled to retrieve the query metadata of that term. The query set is then the union of all queries over all the indexing terms. The crux of the learning scheme lies in selecting the useful terms of a document from the query set. Now, from the query set, we can gather two important pieces of information. First, we can determine how similar is the document to the past queries. We define the query score, $qScore$, as follows:

$$qScore(Q, D) = \frac{|Q \cap D|}{|Q|}$$

Intuitively, if a query is very similar to a document, then it indicates that the terms in the query can represent the key meaning of the document. In other words, the document is likely to be relevant to that query. Careful readers may question why we have not used the conventional formula to measure the similarity between a query and a document. If the conventional formula is employed, the role of a query and document are interchanged: the document is treated as a query and the queries are treated as the document corpus. This is because we are now selecting similar queries for a document. In the conventional formula, the more documents a term occurs in, the less important the term is, which is not true in our scenario. When choosing descriptive queries, a term occurring in many queries as well as in the document indicates that the term is more descriptive of the document. Therefore, $qScore$ can represent the similarity between a query and a document better than the conventional formula.

Second, for each term t in the query set ϑ, we can determine how frequently it appears in ϑ. This is denoted as $QF(t, \vartheta)$, the query frequency of t in ϑ. This essentially tells us how common the query term is. Intuitively, if a term occurs frequently in many queries, it may be potentially useful to index it.

Now, given a set of queries ϑ, the similarity of term j in query i to document D ($t_{ij} \in D$) is defined in the following formula:

$$Score(t_{ij}, D) = qScore(Q_i, D) \cdot \log QF(t_{ij}, \vartheta).$$

The formula indicates that a term is representative to a document if (1) the query containing it is similar to the document; and (2) the term in the document is frequent among the queries. Intuitively, it is insufficient to consider (1) alone since

Algorithm 1: The basic learning algorithm.

```
1  $Q$ is the past query set;
2  $Q'$ is the current query set;
3  $RL$ is a rank list, which is empty initially;
4  for each shared document, $D_k$ do
5      for each $t \in D_k$ do
6          let $qf = QF(t, Q)$;
7          let $qf' = QF(t, Q')$;
8          for each $Q_i \in Q \cup Q'$ do
9              if $t \in Q_i$ then
10                 let $s = qScore(Q_i, D_k) \cdot \log(qf + qf')$;
11                 if $t$ is not in $RL$ then
12                     Insert ($s$, $t$) into $RL$;
13                 else
14                     if the existing similarity is smaller than $s$ then Replace
                       the existing similarity with $s$;
15     Choose top $T$ highest ranked terms for $D_k$;
```

it does not factor in the frequency of the occurrences of the terms in a query (and hence fails to consider similar queries). It is insufficient to consider (2) alone because a document is relevant to a query if there are more matching terms from the query. Thus, a combination of the two is necessary. In combining the two, we have used a logarithm of the QF to give higher weight to the contribution of *qScore*. The reason for reducing the effect of QF is because the qualities of the queries are different. Expert users usually have good domain knowledge and issue high quality queries. Such queries are very useful in differentiating the requested documents from others. On the other hand, poor queries always include terms that are too general to distinguish the requested documents.

Algorithm 1 presents a straightforward implementation of this scheme. The algorithm checks every term in the shared documents against all the queries.

Based on the ranking by (this combined) *Score*, we pick the high scoring terms to be indexed. Now, a straightforward optimization is for the owner peer to store the query sets whenever they are retrieved, so that each iteration only needs to pull back the incremental query set. Even so, this algorithm is expensive in terms of both storage cost and computation cost. The owner peer has to keep all the past queries and check all of them in each iteration of learning. We propose an algorithm that can compute $Score$ for all terms based on only the incremental query set between iterations (without having to recompute from the entire historical query set). See Algorithm 2.

Algorithm 2 gives an algorithmic description of the scheme. Let the query set between the current iteration and the last iteration be Q'. Here, the owner peer only needs to store some statistics for the past queries (up to the last iteration, but excluding queries in Q') along with the documents instead of the entire set of queries. For each term in a shared document, only its query frequency and the largest query score in the history are maintained. Then every new query in Q' is processed. If the term occurs in the query, we calculate the query score for this term and count its query frequency in Q'. If the query score is larger than the one saved for past queries, we update it for this term. The query frequency of this term is the sum of the one for past queries and the one in Q'. A new similarity between the term and the document is calculated with the two parameters. We then insert the term with its new similarity into a list sorted by similarity. If the term exists in the list, then we simply update its similarity value. After all the new queries are processed for a term, the largest query score of the term is stored in the statistics and the query frequency of the term is increased also. Given two sets, S_1 and S_2, it is obvious that $max(S_1 \cup S_2) = max(max(S_1), max(S_2))$. So, the query score used is the largest for a term. QF is simply a count function and is thus cumulative. With the same two factors, the multiplication is the same, so the results of Algorithm 2 is equivalent to the naive scheme described earlier (that

Algorithm 2: The optimized learning algorithm.

```
1  $Q'$ is current query set;
2  $RL$ is a rank list, which is empty initially;
3  for each $t$ in the document $D_k$ do
4      Let $qf$ be the query frequency of $t$ stored for the past queries;
5      Let $qf' = QF(t, Q')$;
6      for each $Q_i \in Q \cup Q'$ do
7          if $t \in Q_i$ then
8              Let $qs$ be the largest query score associated with $t$ in the past queries;
9              $qs' = qScore(Q_i, D_k)$;
10             if $qs < qs'$ then $qs = qs'$;
11             Let $s = qs \cdot \log(qf + qf')$;
12             if $t$ is not in $RL$ then
13                 Insert ($s$, $t$) into $RL$;
14             else
15                 if the existing similarity is smaller than $s$ then
16                 Replace the existing similarity with $s$;
17 Choose top $T$ ranked terms for this document;
```

reprocesses all the queries in each learning iteration). Clearly, since Algorithm 2 exploits incremental computation (i.e., only need to compute for queries that arrive between the last iteration and the current iteration), it is very efficient.

The score of a term to a document calculated using Algorithm 2 is the same as the one using Algorithm 1. Let us look at QF first. QF is counted on the union of the past query set and the current query set in Algorithm 1. In Algorithm 2, QF on the past query set is stored as metadata and QF on the current query set is counted. They are summed to get the new QF, which is obviously the same as the one in Algorithm 1. Algorithm 1 calculates the largest query score in the union of the past query set and the current query set. In Algorithm 2, the largest query score, $Score(t, D)$, of the past query set is stored. The largest query score of the current query set can be calculated. The larger score between the two is the largest score in the union of the two sets. With the same two factors, the multiplication is the same, so the two algorithms basically select the same terms for a document.

A learning example with Algorithm 2 is discussed in figure 4.3. A document,

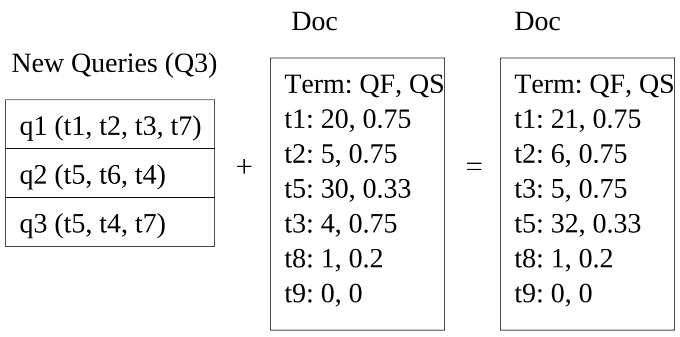


Figure 4.3: The learning example in SPRITE.

Doc, is limited to be indexed with three terms. At time i, $t1$, $t2$ and $t5$ are indexed (shown in the left *Doc*). Their similarities to the documents for the past queries are: 0.75*log 20=0.975, 0.75*log 5=0.524 and 0.33*log 30=0.492 respectively. Three queries are pulled back in the learning process: {Q1, Q2, Q3}. Then the query frequency and the largest query score are updated accordingly (shown in the right *Doc*). We recalculate the similarities and obtain a new ranked list. The new score of $t3$ is 0.75*log 5=0.524 and the new score of $t5$ is 0.33*log 32=0.501. Thus, $t3$ is indexed and $t5$ is removed from the distributed index for *Doc*.

4.5 Experimental Study

In this section, we evaluate the performance of SPRITE. As reference, we use a centralized text retrieval system and the basic eSearch system [83]. The centralized system acts as an ideal distributed system with perfect global knowledge, including the exact document frequency and total number of documents in the corpus. (We used a classic $TF \cdot IDF$ scheme in the centralized system). Hence, it is expected to be superior. By comparing against it, we will be able to see how close SPRITE is to an optimal solution. The basic eSearch system indexes a fixed number of most frequent terms in a document. It is the best distributed search system currently known. The comparison against eSearch demonstrates the gain that can be derived from adaptivity/learning.

We preprocessed the documents in the standard way: removing the terms in the stop-word-list, and then stemming is applied to the remaining terms. The default stop-word-list in Lucene [54] is used for this purpose. We used the two standard metrics for text search: precision and recall. If the top K documents are returned for a query, K' of them are relevant to the query and there are R relevant documents in the entire corpus, then the precision is defined as K'/K and the recall as K'/R. All precision and recall results presented later are in terms of the ratio of a specific system over the centralized system.

We implemented Chord as designed in [81]. All terms are hashed using MD5 hash function. Our study is based on simulation, and all experiments are conducted on a dual-Pentium4 3.0GH CPU PC with 1GB RAM.

4.5.1 Data set and query set

To evaluate SPRITE, we need queries to be "similar" (share some keywords and relevant documents) for SPRITE to learn from. Unfortunately, benchmarks are usually created to exercise a maximum of functionality with as few queries as possible. Hence, there is little similarity between queries. To deal with this, we implemented a query generator to generate queries from a real dataset and its corresponding queries. We used the TREC9 dataset and its queries [33] as the base dataset. This dataset contains 348565 documents and 63 queries and their corresponding relevant documents (identified by experts). Our generator is designed based on two reasonable properties: (a) queries with similar relevant documents as answers ought to share some common keywords; and (b) the term distribution and result distribution should follow those of the original query set. The first property ensures that the system can build an effective index with the training queries and the testing queries can benefit from the learning process. The second property ensures fairness: popular terms in the original query set should occur frequently in the generated query set. If an original query has many answers (the documents), then the new queries derived from it should have many answers as well. In the centralized system (the benchmark), the relevant document distribution in the ranked list of a new query should be similar to that of its original query. The query generator comprises the following two phases.

Phase 1: Term Selection. In phase 1, for each query in the original dataset, we generate k new queries. (In our study, we set k to 9.) We shall define some terminology first and then use them to illustrate how to generate new queries from an existing one.

- Q: The original query with a set of terms: $\{q_1, q_2, ...q_n\}$.
- Q': A new query generated from Q. It contains another set of terms: $\{q'_1, q'_2, ...q'_m\}$.
- $|Q|$: Number of terms in Q.
- E: Number of examined answers.[1]
- R: The set of documents identified as relevant to Q among the top E answers by experts: $\{D_1, D_2, ...D_r\}$.
- D_i: A relevant document to Q, $D_i \in R$. The document contains terms: $\{t_{i1}, t_{i2}, ...t_{ik}\}$.
- R': The set of documents defined as relevant to Q' among the top E answers.

[1]Some relevant documents will never be returned to users because their ranks are very low and users are usually interested in a small number of highly ranked results only. Thus, they will not affect the precision or recall and are not considered when defining relevant documents for the new queries.

A new query Q' is composed of two sets: $Q' = Q'_1 \cup Q'_2$. The terms in Q'_1 are from the Q: $Q'_1 \subset Q$. Each term in Q'_2 is randomly selected from the term space, which contains all terms appearing in all documents. Thus, while Q'_1 inherits some terms from Q, Q'_2 targets different aspects of the documents to introduce some noisy terms to model a more realistic scenario. For the new query Q', we need to identify a set of documents R' as its relevant results (see phase 2).

We define a tunable parameter to control the overlaps between the original queries and new queries.

$$O = \frac{|Q'_1|}{|Q|}$$

The threshold, O (overlap), determines the percentage of terms in the original query that is retained in the new queries. Tuning this factor will change the overlap between the original query and new queries. The actual terms in Q'_1 are randomly picked from Q.

In order to select terms of type Q'_2, we pick terms from the entire corpus that are "equally" important as the terms that have been dropped from Q. The importance depends on the distribution of the term: the number of term occurrence and the number of documents containing the term. We define a simple metric to measure the distribution of a term in a corpus.

$$Distribution(t_i) = Freq(t_i) \times Num(t_i)$$

Here, $Freq(t_i)$ is the total term frequency of term t_i in all documents and $Num(t_i)$ is the number of documents containing term t_i. The two factors are used to measure the importance of the term. The reason we do not use the conventional term weight formula $TF \cdot IDF$ is that it can only represent the weight of a term in a document. $Distribution(t_i)$ focuses more on the distribution of a term in the corpus. Given a term in $Q - Q'_1$, we find the top S similar terms and choose one of them randomly to replace the old term. Here, the difference between two terms t_i and t_j is measured by $|Distribution(t_i) - Distribution(t_j)|$ (the smaller the value is, the more similar they are). In our study, S is set to 5. All terms in Q'_2 are selected randomly from the replaced terms in $Q - Q'_1$.

Phase 2: Identifying Relevant Documents. In phase 2, the relevant documents of the generated queries are defined based on the relevant documents of the original queries. We now define some documents as relevant answers to the new queries. A new query ought to share some relevant documents with the original query and have some new relevant documents for itself. With the centralized system, we can calculate the ranked list, RL for the original query Q, and RL' for a new query Q', over all the documents. The top E documents in the ranked lists are considered when defining relevant documents for Q'. Some relevant documents will never be returned to users because their ranks are very low and users are usually interested in a small number of results only. Thus, they will not affect the precision or recall and are not considered when defining relevant documents for the new queries. For each such document in RL' and relevant to Q, we define it as relevant to Q' and mark the relevant document in RL with the most similar rank. Then, for each unmarked relevant document in RL, the document in RL' with the same rank is defined as relevant to Q'. An example is shown in Figure 4.4. Here, RL is the ranked list to an original query Q, and RL_1, RL_2 and RL_3 are the new ranked lists for new queries Q_1, Q_2 and Q_3 derived from Q. Circles are the original relevant documents to Q and crosses are newly defined relevant documents. The left most document has the highest rank. In this example, $E = 14$ (In the experiments, $E = 1000$). For Q_1, 3 original relevant documents (marked with circles) are in its top E ranked list. Three documents in RL with the most similar ranks are marked (indicated by the dashed lines). For the remaining two relevant documents, two

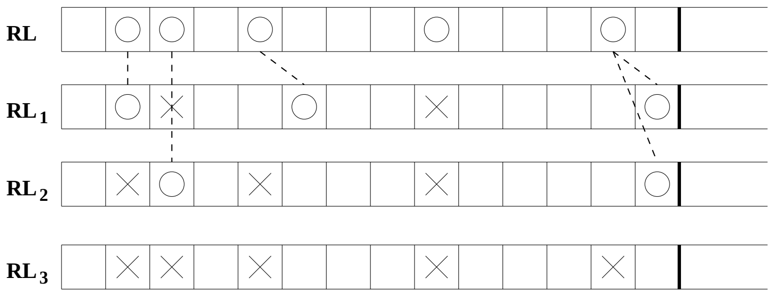


Figure 4.4: Defining relevant documents.

documents in RL_1 with the same ranks are defined as relevant to Q_1 (marked with crosses). In this way, the distribution of the new relevant documents is similar to the distribution of the original relevant documents.

4.5.2 Experimental setup

We started with 63 queries from the TREC9 dataset, so we eventually have 630 queries with the overlap ratio $O = 70\%$ after the query generation. We split these queries into 2 equal groups: a training set and a testing set. The queries are randomly assigned to the groups. For each query in the training set, the keywords are inserted into SPRITE. Next, we insert the metadata of the documents into the system as follows. For each document to be inserted, 5 most frequent terms are initially indexed. Following the 5 initial terms, 3 iterations of learning are executed by the owner peer of a document. In each iteration, 5 new terms are indexed. So the total number of terms indexed equal to 20. Once all the documents have been indexed, we run the queries in the testing set. For each query, we retrieve top 20 answers and determine its precision and recall. For eSearch, we set the number of indexed terms as 20. In the above description, the parameters used (e.g., 5 initial terms), are the default settings. Unless otherwise stated, we assume the default settings.

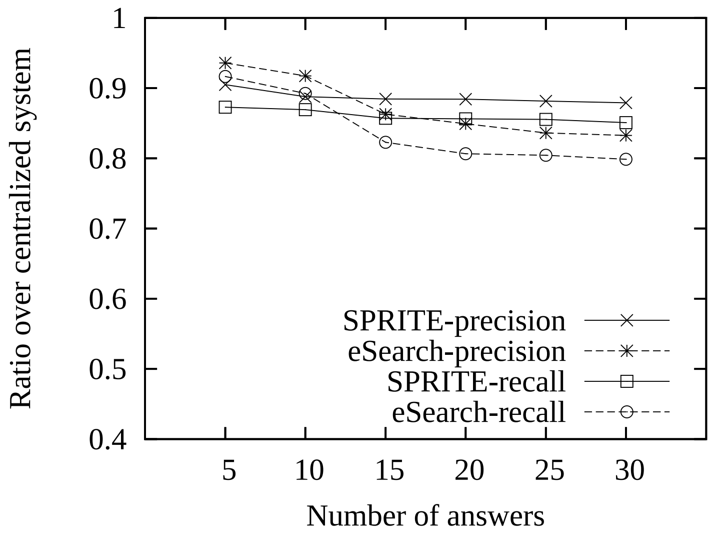


Figure 4.5: Varying number of answers.

4.5.3 Experimental results

First, we compare the precision and recall between SPRITE and eSearch when the number of answers varies. As shown in Figure 4.5, the eSearch system outperforms SPRITE when the number of answers is small (5-10); but SPRITE gives better performance when the number of answers is larger (15-30). Both eSearch and SPRITE are not as good as the centralized system, which is the price for indexing 20 terms only. Some relevant documents are missed due to some unindexed terms. We also observe that SPRITE's precision of 89% and recall of 87% are relatively constant with respect to the centralized scheme. The eSearch system degrades much faster when the number of answers is larger. The terms indexed in SPRITE are more representative for the documents because it is able to learn from past queries. Therefore, SPRITE can perform constantly well when the number of answers increases; the most frequent terms indexed in eSearch can only benefit a small fraction of documents in the collection.

Next, we vary the number of terms indexed. Figure 4.6 shows the results of

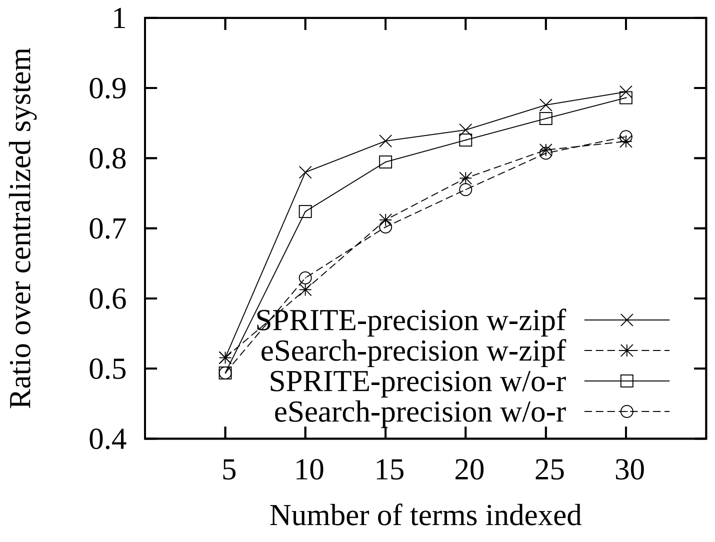


Figure 4.6: Varying number of index terms.

two sets of queries: "w/o-r" (without repeats), where every query appears exactly once and "w-zipf" (with Zipfian distribution, whose slope is set to 0.5), where the frequency of a query is roughly inversely proportional to the popularity of the query. The "w/o-r" query set is an extreme case that is biased against SPRITE. Most queries are repeated as we mentioned previously and the phenomenon is shown in [90] and [78]. SRPITE can obtain the least knowledge from the past queries in this case. Note that when 5 terms are initially indexed, no learning process is involved, so the two systems have the same performance. First, we observe that SPRITE outperforms eSearch with the same number of terms indexed. In fact, the gain over eSearch is larger with fewer terms indexed (except when the number of terms is 5). Second, SPRITE can achieve similar performance as eSearch with fewer terms. For example, the performance of SPRITE with only 20 terms indexed is nearly the same as that of eSearch with 30 terms indexed. This is very important and useful in a P2P system since indexing fewer terms means lower cost for inserting the global index terms initially as well as for maintaining the index subsequently. This also suggests that many frequent terms indexed by eSearch

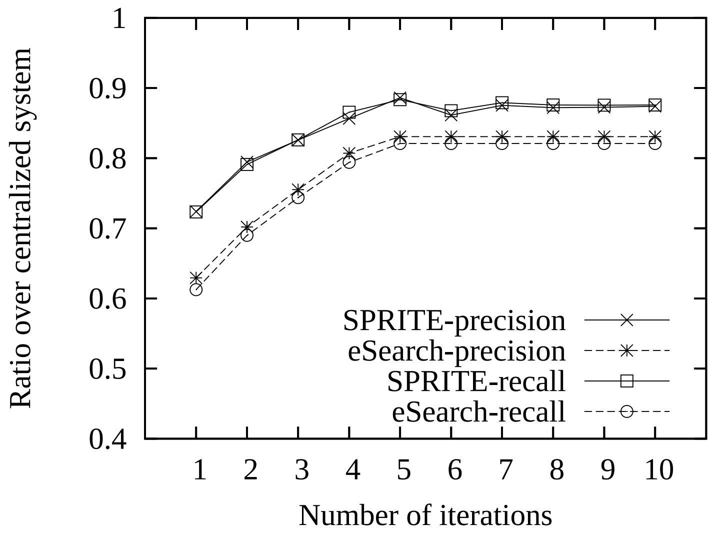


Figure 4.7: Change on query pattern.

do not contribute to answering queries. Instead, SPRITE successfully removed these redundant terms. Lastly, under the circumstance that either queries do not even repeat ("w/o-r") or queries are issued in a very skewed distribution ("w-zipf"), SPRITE always outperforms eSearch. SPRITE can sufficiently learn the key meanings of a document from similar queries or identical queries. We observe similar trend for recalls and do not present the results due to the space limitation.

Finally, we study SPRITE's robustness to changes in query access patterns, e.g., users may be interested in one collection of documents in a period and then in another collection later. Figure 4.7 depicts the precision and recall when query pattern changes. The query set is evenly partitioned into two groups such that all new queries and their corresponding original query are in the same group. In the first 5 learning iterations, queries in one group are processed and evaluated. In the next 5 iterations, the other group of queries are processed and evaluated. Thus, in the first 5 iterations, none of the queries in the second group is known to the system. In this set of experiments, we set the maximum number of terms to index to 30, after which the number of indexed terms remains unchanged. Instead, we apply term replacement (as described in Algorithm 2) only. This is also the reason the performance of eSearch remain unchanged after iteration 6. SPRITE always outperforms eSearch as usual when the number of indexed terms increases in the first 5 iterations. From the 6th iteration, new queries are issued in the system. As can be seen from the results, SPRITE adapts to the changes very quickly. The precision and recall decrease a little bit at the beginning of the new queries arrival, but are still better than those of eSearch. After just one iteration, SPRITE recovers and gives good performance in a stable status. The reasons are twofold: The first 5 iterations mainly polish the indices of related documents based on the first group of queries. They have very little effect on the later queries and their relevant documents. When the new queries are issued (in the 6th iteration), the terms indexed (based on the first group of queries) are unable to provide adequate relevant documents. However, SPRITE's learning capability ensures that the indices are carefully tuned to meet the new set of queries in the following iterations.

4.6 Summary

In this chapter, we have presented a novel scheme to build compact yet effective index on text data in a P2P network. Building complete index on text documents is impractical in a P2P network because of the extremely high maintenance overhead. SPRITE reduces the overhead cost by indexing a small number of terms in a DHT network. In SPRITE, the index is tuned progressively based on past queries, so that only a small number of representative terms of a document are selected and indexed. The meaning of a document is mainly characterized by the indexed terms. Therefore, queries can be processed based on the partial index effectively.

SPRITE offers the following advantages over the IFT scheme. First, only a small number of selected terms in a document are indexed based on past queries. The major meaning of the document is represented by these terms. This is extremely important in a P2P system, not only for index construction and update, but also because periodic checking on distributed indexes is required. Second, SPRITE uses progressive learning to refine the set of selected index terms. Even when users change their interests, SPRITE can adapt quickly to tune the index. Our extensive simulation study showed that SPRITE can achieve performance similar to a centralized system in terms of precision and recall, and considerably outperforms a static index term selection approach.

Chapter 5

CYBER: a ***C***ommunit***Y***-***B***ased s***E***a***R***ch engine

5.1 Introduction

In the previous chapter, we have presented SPRITE as an effective solution to reduce the index construction and maintenance overhead. In this chapter, we investigate how to further improve the accuracy of information retrieval in DHT-based schemes by leveraging on community-based feedback.

In centralized systems, techniques based on relevance feedback have been effective in improving the query precision and recall. We can classify feedback-based techniques in a P2P network into three groups based on the granularity of the community that a user belongs to.

On one extreme, we have the *single-user* community-based approaches. This is essentially a straightforward adaptation of centralized methods, and is accomplished in two steps. Every individual user is returned with a preliminary ranked list, from which the user makes some selection. The query is refined by increasing the weights of some existing query terms or introducing new query terms. The refined query is then routed again to construct a new ranked list. This process is repeated until the user is satisfied with the answers. While the quality of the ranked list improves in each iteration of user feedback, the routing cost is also higher. In addition, this approach fails to exploit the feedback of other users who share the same interests.

On the other extreme, we have the *global* community-based approaches where the entire user base is treated as a single global community. Here, the weights of terms in a document is adjusted as follows: whenever a document is selected as relevant to a particular query, its terms appearing in the query are assigned larger weights. In this manner, a document will eventually be characterized by user queries. However, this approach implicitly assumes that all users in the global community share the same interest. In practice, end-users come from different communities - while users within a community are expected to share similar interests, users from different communities have little overlap in their interests (even if the query terms are the same, they may be looking for different data). As a result, by treating all users alike, a document that is relevant to a community may have a negative impact on the query results of users in another community.

The third approach, which we advocate in this chapter, is a *pure* community-based approach, where each community corresponds to a group of users with similar interests. The objective is to leverage on community information so that the documents can be ranked according to the community that the user belongs to. In other words, the feedback of users in the same community are unified such that they only have impact on queries that are issued by other users from the same community. Thus, users from different communities sharing the same query keywords should retrieve different ranked results. This approach overcomes the problems in

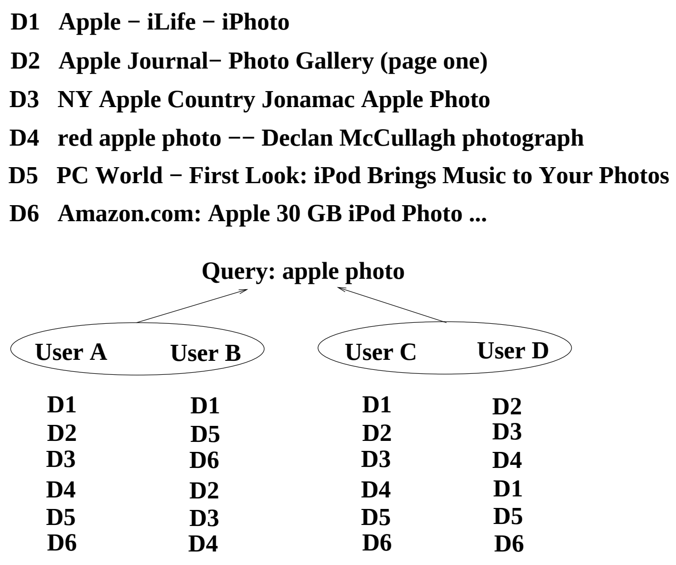

D1 Apple − iLife − iPhoto
D2 Apple Journal− Photo Gallery (page one)
D3 NY Apple Country Jonamac Apple Photo
D4 red apple photo −− Declan McCullagh photograph
D5 PC World − First Look: iPod Brings Music to Your Photos
D6 Amazon.com: Apple 30 GB iPod Photo ...
Query: apple photo
User A
User B
User C
User D
D1 D2 D3 D4 D5 D6
D1 D5 D6 D2 D3 D4
D1 D2 D3 D4 D5 D6
D2 D3 D4 D1 D5 D6

Figure 5.1: A search example with query “apple photo” and 6 documents in the ranked list.

the first two approaches. On one hand, a user is not required to refine his query manually/explicitly. On the other hand, the feedback are more accurate since they are from users in the same community.

Figure 5.1 illustrates a search example. There are four users, A and B belong to the community of Apple fans who are interested in the latest products from Apple, and C and D are nutritionists who are looking for information on apples. At first, users A and C issue the same query and obtain the default ranked list shown in the figure (these 6 documents are obtained using Google search engine). From the figure, it is clear that the ranking is not satisfactory - for user A, D_5 is ranked lower than D_2, D_3 and D_4; for user B, the highest ranked answer has nothing to do with apples. In the ideal case, when user B queries after user A, had user A's preferences been noted, the result should be a more accurate rank list where documents related to Apple are ranked higher than documents related to apples (e.g., in the figure, D_5 is ranked higher than D_2, D_3 and D_4). Likewise, for user D to issue his query after user C, (s)he should retrieve a list that ranks apples higher than Apple products. Thus, a community-aware system is desirable for improving search effectiveness.

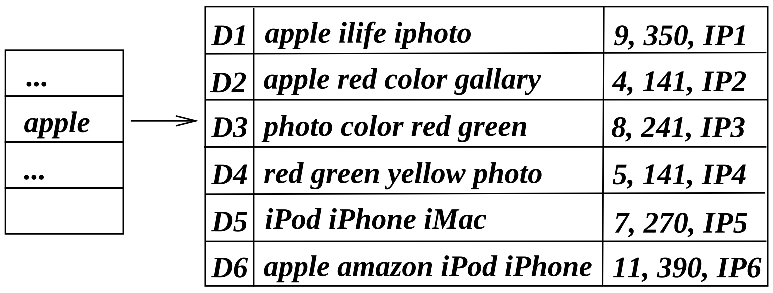
Our solution is motivated by the following three observations. First, users with common interests tend to query on similar objects. On the other hand, users querying on similar objects share some common interests. In reality, an earlier query issuer always recommend "good" articles, books or movies to his friends in a particular community. Second, in the real world, a user base usually consists of many communities (e.g., Apple products and fruit lovers), and each community may further consist of smaller groups (e.g. in the Apple community, a group may be interested in Apple iPod and another in Apple iMac). It is very hard to fully exploit any recommendations because of the overhead to build and maintain the relationships amongst all users/peers with similar interests. A more practical approach is to somehow consolidate community feedback so that peers can locate them easily. This will also reduce the initialization and maintenance cost on the community. Third, recommendations should be made available only to related peers. For those who are not in the same community, these recommendations should be ignored.

With the three observations in mind, we propose *CYBER*, a *C*ommunit*Y* *B*ased s*E*a*R*ch engine, for information retrieval utilizing community feedback information in a DHT network. Like existing DHT-based document retrieval systems [49], CYBER builds a DHT-based index on (selected) terms of a document. However, CYBER distinguishes itself from these systems as follows. First, for each term indexed at a peer, the peer also maintains a number of *document profiles* of the term. Each such profile reflects a community's interests on the document w.r.t. the term. For example, for a community that finds the document relevant, the profile facilitates higher ranking of the document when another user of the community queries with the term. On the other hand, for a community that finds the same document to be irrelevant, its profile will result in this document being weighted lowly for a user of the community. To some extent, the indexing peer acts as a "meeting place" where users of a community can "annotate" terms of documents to "post" their "recommendations" (by updating the document profile for that community).

Second, unlike existing feedback-based mechanism, users' answers are obtained based on the aggregated feedback from the community: the weight of every queried terms is adjusted according to the similarity between the document profile and user profile, and then the new weights are used to calculate the ranked list. Essentially, if the user profile matches a document profile, the term weight for this document would be adjusted to be higher; and vise versa. More importantly, no iterative feedback is required.

Third, in CYBER, users offer feedback after viewing some selected objects for his query. The feedback can be as simple as relevance judgment, i.e. relevant/irrelevant or as complex as some scores/ranks of relevant objects. However, these feedback are not used to refine the same user's query. Instead this feedback and the user's profile of interests are sent to the peers indexing the queried terms to refine the profile of the selected documents. We conducted an extensive performance study, and our results showed that CYBER is an effective P2P system for document retrieval.

The rest of this chapter is organized as follows: We present the proposed CYBER system in Section 5.2, and report results of an experimental study in Section 5.4. Finally, we conclude this chapter in Section 5.5.



D1	*apple ilife iphoto*	*9, 350, IP1*
D2	*apple red color gallary*	*4, 141, IP2*
D3	*photo color red green*	*8, 241, IP3*
D4	*red green yellow photo*	*5, 141, IP4*
D5	*iPod iPhone iMac*	*7, 270, IP5*
D6	*apple amazon iPod iPhone*	*11, 390, IP6*

Figure 5.2: Index entry example.

5.2 CYBER

We now present how CYBER improves the search effectiveness with community feedbacks. When a document is shared, a number of terms are chosen to index the document in order to reduce the index maintenance overhead. For each indexing term, a profile is constructed, which contains a set of representative terms. Initially, all document profiles are identical for all indexing terms of a document. However, they will be updated by different user feedbacks and each of them then reflects the interest of a community of users on a particular term. Figure 5.2 shows an example on an index entry stored in an indexing peer. The indexing peer may be responsible for several terms. There are 6 documents (D_1 - D_6) containing the term, *apple*. Let us consider D_1 in the first row of term *apple*'s table. The indexing peer stores the document profile, {apple, ilife, iphoto}; the term frequency, 9; the document length, 350; and the IP address of the owner peer. From the profiles, we can see that documents D_1, D_5 and D_6 are related to Apple products; and documents D_2, D_3 and D_4 are about the fruit.

A user has several profiles to represent his/her multiple interests. Similarly, each user profile also contains a term vector. Figure 5.3 illustrates the profiles of two users, A and B. User A has two profiles and user B has one only. When a user issues a query, his profiles are also attached. For each queried term, its original

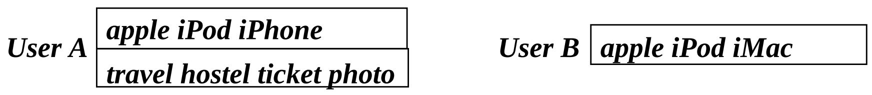


Figure 5.3: An example of user profiles.

weight in a particular document (measured with $TF \cdot IDF$) is adjusted with the similarity between the document profile and the user profile. The score of the document in the ranked list is affected by the weights of all queried terms. After the user selects a few relevant documents by clicking the related links, the document profile of each queried term w.r.t. the relevant documents is refined according to the user profile. User profiles are enhanced with relevant (downloaded) documents as well.

In order to facilitate query refinement with community feedback in CYBER, *owner peers* and *indexing peers* need to perform some additional tasks. After some documents are shared in an *owner peer*, a profile is initialized for the user. When routing a query, the profile is attached as well. Besides storing the metadata of a term, an *indexing peer* is also responsible for maintaining a document profile for the term on each related document. When processing queries, the user profiles (in *owner peers*) and the document profiles (in *indexing peers*) that are involved will be updated automatically.

5.2.1 Profile initialization

Initializing a document profile is straightforward. Suppose that a user is willing to share a document, D_i. We use the same method for choosing the indexing terms to initialize the profile. The K' most frequent terms are extracted to construct the document profile. Alternatively, the owner can initialize the document profile or update the top K' terms manually. When a document is indexed, its profile is attached to every indexing term and stored in the corresponding indexing peer.

When an owner peer periodically checks if the indexing peer is alive, the owner peer will backup the document profile when it is changed significantly. We will present profile updating shortly. In case the indexing peer is offline, rebuilding the profile (based on feedback) is avoided. However, the frequency is much lower than checking the availability of the indexing peer. This is because an offline indexing peer can lead to missing answers for some queries, while a stale profile will only cause relatively inaccurate results.

It is a bit more complicated to initialize a user's profiles as the user can have multiple interests. All of the shared documents are clustered first. In each cluster, the frequency of every term is accumulated. The M terms with the highest frequencies in a cluster are chosen to construct a user profile. The reason the user profiles are not combined into one is that profile matching calculation is normalized by the user profile. A large user profile will always cause a small matching value. Therefore, using one of the multiple user profiles is more accurate. Since the user profiles are stored at the local machine, the owner peer does not need to backup these profiles. Figure 5.3 illustrates the profiles of two users, A and B. User A has two profiles and user B has one only.

5.2.2 Profile-based query processing

When a user issues a query, his profiles are sent to the involved indexing peers with the query together. For each user profile, the indexing peer calculates its similarity to every document profile associated with the queried term. The similarity between two profiles is defined as: $vsim(V_u, V_D) = |V_u \cap V_D|/|V_u|$, where V_u is the uth profile of a user and V_D is a document profile. $|V|$ is the size of the profile V. Recall the document profile example in Figure 5.2 and the user profile example in Figure 5.10. For user A, the profile similarity $vsim(V_{A_1}, V_{D_5}) = 2/3$, since there are 3 terms in the first profile of A and 2 of them appears in the profile of document D_5.

The similarity between the profiles has an effect on the weight of the querying term in the related document. Instead of weighting the term simply with $TF{\cdot}IDF$, the term weight is adjusted by $vsim$. The new weight of a term is: $W_{i,j,u} = w_{i,j} \times (1 + vsim(V_u, V_D))$. The factor, $w_{i,j}$ is the original term weight derived from $TF{\cdot}IDF$. The factor $(1 + vsim(V_u, V_D)$ increases the weight of the term by taking the similarity between the two profiles into account. In our running example, the original weight of *apple* in document D_5 is $w_{1,5} = 7 \times \log 300/6 = 11.89$, given that *apple* is the first term in the query and the total number of documents is defined as 300. After the augmentation with profile similarity, the new weight $W_{1,5,1} = 11.89 \times (1{+}2/3) = 19.82$. In document D_3, the original weight of *apple*, $w_{1,3} = 13.59$ and the new weight, $W_{1,3,1} = 18.12$. For query term *apple*, we can see that its weight in D_5 is increased tremendously and larger than that in D_3 with the factor of profile similarity considered. Hence D_5 will be ranked in front of D_3 with larger probability. Note that we cannot guarantee every relevant document will be ranked before irrelevant documents. However, the probability that relevant documents are moved forward in the ranked list is increased, which is the ultimate goal of a search engine.

Because a user may have several profiles, an indexing peer can interpret the query term in multiple ways for every document indexed on the queried term. Therefore, given a query term, multiple weights are returned to the querying peer for the same document. After all the metadata of the query terms are returned, the querying peer calculates the similarities between the query and involved documents for each user profile. Then the maximal value is associated to the document as its final similarity to the query. The similarity between a document, D_i and a query, Q is:

$$SIM(Q,D_i)=Max(\forall u\in U,\frac{\sum_{j\in Q} w_{Q,j}\times W_{i,j,u}}{\sqrt{number\ of\ terms\ in\ D_i}})$$

Every query term is still weighted using the $TF{\cdot}IDF$. Each term in the document is weighted with the profile matching considered. There are U user profiles, and the one with the maximum similarity is selected. Referring to our running example, for the query “apple photo”, the second profile of user A is dominated by the first one.

5.2.3 Document profile updating

The document profile consisting the most frequent terms may not capture the meaning of the document accurately. Therefore, CYBER refines the document profile according to user profiles progressively. After a user issues a query, a ranked list is returned to the querying peer. The user clicks a few documents from the result list to view them or download them from their owner peers. At the same time, a message on the user selection is sent to all participating indexing peers. The user selection includes both the relevant document identifiers and the user profile identifier that is used in increasing the document score. Previously, when the query is submitted to the indexing peers, the user profile is attached. The user profile is not abandoned within a certain time. If a user selection message is sent back, then it indicates that some documents are determined as relevant answers due to the indexed query term. Otherwise, the profile is discarded after a certain period of time. Note that a query usually contains a few terms only and the querying peer can directly contact the involved indexing peers, so the routing overhead is very small.

With the user profile, the original query and the user selection, the involved indexing peers can now perform the update on the document profile for those queried terms. First, the indexing peer checks the similarity between the profiles.

The profiles of relevant documents will only be updated according to the user profile if the similarity is above a certain threshold. For each term in the involved user profile, the indexing peer checks if it is in the document profile. If not, the term is injected into the document profile. For the indexing term, its term frequency (initially calculated in its owner peer and sent to the indexing peer) in the relevant document is increased by a certain percentage (In our implementation, the default incremental rate is set to 0.5).

In such a manner, the document profile is eventually constructed by the keywords from users who consider the document as a relevant answer. By then, the document profile will rarely be changed as nearly all important descriptive terms appear in the document profile. Such users share the common interest on this document, and thus their virtual relationship in the community is established with the assistance of the document profile in the indexing peer.

Assume D_5 is selected by the user in our running example in Figure 5.2 and the profile similarity threshold is 0.2. A message is sent to the indexing peer responsible for *apple*. Since the profile similarity between the profile of D_5 and the user profile is 2/3 (which is great than 0.2), the document profile should be updated by the user profile. The keyword *apple* is inserted and the new profile of D_5 is (*apple*, *iPod*, *iPhone*, *iMac*). As both users A and B are *Apple* fans, their profiles are expected to be similar. When user B's query is sent to the indexing peer responsible for *apple*, because his profile similarity to D_5 is 1 (see B's profile in Figure 5.10), document D_5 will (very likely) be ranked higher.

5.2.4 User profile updating

The profile of a user should be able to evolve with both the change in user's interests and the trend of documents shared in the network. After a user downloads some relevant documents for a issued query, a number of the most representative terms are extracted from them and compared with the user profile. If the overlap between them is below a certain threshold, then a new profile is initialized for the user, which indicates that the user is interested in a new topic most likely. Otherwise, the most similar profile is updated according to the new set of terms. Essentially, the downloaded documents and the original cluster of documents are merged and a number of representative terms are extracted to construct the profile. In this case, the new vocabulary in the interested topic is captured and stored in the user profile. In this manner, changes in either the user interest or the document trend are reflected in the user profile. Depending on $iMac$'s frequency in user A's collection, including both shared and downloaded documents, it may be inserted into the first profile of user A.

5.3 Dynamic Tuning of CYBER Indexes

In CYBER, a subset of terms are selected for indexing. However, these terms are picked statically. To ensure the set of terms are always relevant and representative, it is necessary to remove terms that are not used in queries and to add terms that are frequently used (but not previously indexed). In this section, we shall present two approaches to extend CYBER to facilitate dynamic index tuning: CYBER+ and CYBER++, which are based on SPRITE.

5.3.1 CYBER+

Our first approach to tuning the indexes of CYBER is essentially a straightforward adaptation of SPRITE. Besides re-evaluating queries with community-based relevance feedback, CYBER+ refines the index in the same manner as SPRITE. In

CYBER+, refining index and re-evaluating queries are loosely coupled. An owner peer periodically pulls back related past queries, which are stored in indexing peers. Then the score of each overlapping term between the shared documents and past queries is calculated. A certain number of terms with the highest scores are indexed after each learning iteration. When a new term is selected, the owner peer indexes it in the network, with its initial document profile attached. For each indexed term that is dominated by some new terms, the document profile of the term with respect to the document is discarded by the indexing peer. For each remaining term that is not dominated by any new term, its document profile is not affected. The document profile of such a term will then be updated by user profiles in the future query process as described in Section 5.2. While CYBER builds index on the most frequent terms, CYBER+ attempts to index terms that are more likely to be queried.

Consider there are only two queries in the past: Q_1 (t_0, t_1, t_2, t_3) and Q_2 (t_0, t_4, t_5), where t_i is a term. A document, D, that has indexed t_0 needs to refine its index. Its owner peer first pulls back Q_1 and Q_2. Suppose D also contains t_1 and t_4. Obviously, the overlap ratio between D and Q_2 is larger. Therefore, t_4 is selected as the new indexing term. In the future, if a query containing t_0 and t_4 is issued in the network, then D will be returned as an answer very likely.

5.3.2 CYBER++

We now present a more tightly coupling scheme, CYBER++. Apparently, using past queries alone to select indexing terms is not adequate. Relevance feedback can also be used to refine the index. If a document is identified as a relevant answer to a query, the overlapping terms between the query and the document should be selected and indexed.

After a user issues a query, a ranked list is returned to the user. The user clicks a few answers and download the documents from their owner peers. When an owner peer receives a download request, it stores the query with respect to the document.

When an owner peer needs to refine the index for its shared documents, it first pulls back past queries from the related indexing peers. The owner peer then checks the relevant queries it stores locally. To calculate the score of a term, relevance feedback is considered, if available. If D is relevant to Q_j, then the score of a term to a document is augmented as:

$$Score(t_{ij}, D) = \left(\frac{|Q_j \cap D|}{|Q_j|} + 1\right) \cdot \log QF(t_{ij}, \vartheta).$$

From the formula, we can see that the first factor is increased by 1[1], for terms in relevant documents, while the second factor is not changed. The terms with top scores are selected and indexed after each learning iteration. Therefore, if the document is relevant to a query, the overlapping term is very likely to be selected and indexed. In this manner, we expect fewer irrelevant terms are indexed, and thus some noise are removed to the document. Query processing is the same as described in CYBER.

Let us reconsider the running example in the previous section. Suppose a user identified that D is relevant to Q_1. The relevance feedback is kept in the owner peer of D. When D wants to refine its index, its owner peer checks both the overlap ratio between D and past queries and the relevance relationship. In CYBER++, terms in Q_1 is more important than terms in Q_2, so t_1 is chosen as the new indexing term. In this manner, D_1 will be ranked higher in future queries containing t_0 and t_1.

[1]The increment is large, considering the range of the original value is (0, 1].

5.4 Experimental Evaluation

In this section, we evaluate the performance of CYBER. We compare CYBER against two other DHT-based systems: (a) a search engine that is based on a complete index (CI); (b) a search engine based on partial index with "single-user" feedback technique (PIF). Recall that CYBER is based on partial index and employs community feedback. The full index search engine (CI) is an ideal (but impractical) system where all global information is known and its performance is expected to be as good as a centralized system. By comparing with the CI system, we can study the benefits of feedback-based systems. For both PIF and CYBER, the partial index is built in the same manner so the difference between the two is the way feedback mechanisms are employed. The PIF scheme processes a query in two steps: a preliminary ranked list is returned first; the user clicks some relevant documents and then the weights of query terms are refined and a new ranked list is returned. CYBER is clearly more efficient than PIF as the user receives a final ranked list after submitting the query without any iterative feedback process. Moreover, CYBER follows the Chord routing protocol. Thus, the routing cost is bounded by the number of terms in a query and the number of peers in the network logarithmically.

We use two standard metrics to evaluate the three systems: precision and recall. Given a query, let the number of returned documents be K and K' of them are identified as relevant answers, then the precision $P = K'/K$. If there are A relevant documents in the repository, then the recall of this query $R = K'/A$. All results are presented in terms of the improvement ratio of a specific system (either CYBER or PIF) over the CI system. For example, the precision result of CYBER is defined as: $\frac{P_{CYBER}-P_{CI}}{P_{CI}}$, where P_{CYBER} is the precision of CYBER and P_{CI} is the precision of the CI system. A value larger than 0 means the scheme has better precision/recall than CI.

We implemented the basic Chord protocol as designed in [81]. All terms are hashed using MD5 hash functions. Our study is simulation based and all experiments are conducted on a dual-Pentium4 3.0GH CPU PC with 512MB RAM.

5.4.1 Data set and query set

In a scalable P2P network, both the number of users and the number of queries issued by them are large. In order to evaluate CYBER, we need users to share many documents and issue many queries. The users belong to various communities. In each community, the users issue queries on similar topics. Unfortunately, existing benchmarks are usually created to exercise a maximum of functionality with as few queries as possible. User profiles are not considered either. Therefore, we extend our query generator presented in Chapter 4 to generate queries from the TREC9 dataset and queries [33] as a first step. The TREC9 dataset contains 348565 documents and 63 queries. All relevant documents of every query are identified by experts.

Our original generator mainly derives a number of queries from each existing query and defines a set of documents as relevant answers to every new query. Every new query shares some terms with the original query. Because the queries are similar, they share some documents as relevant answers also. The importance of a new query term (measured as a combination of overall frequency and distribution amongst documents) is similar to that of the original term it replaces. When defining new relevant documents, we endeavor to choose documents that have similar ranks to the relevant documents of the original query. For each original query, we generate 5 queries (including the original one), so there are $63 \times 5 = 315$ new queries. The number of relevant documents of a query is approximately the same as the number for its original query.

Modeling Community

With the larger set of queries and their relevant documents, we now define communities. Even if two users issue two identical queries, they may expect two different sets of documents if their interests/profiles are different. With this in mind, we first define an expected number of communities, C, for every query, which means there are C different interpretations for the query and thus C different sets of answers are expected. Every query is processed with Lucene [54] to generate a ranked list. From the top 200 documents, the first group, R_0, containing all documents that are defined as relevant to the query in the original dataset is extracted first. Let its size be $|R_0|$. The remaining documents are then clustered into $2C$ groups. If the size of a group is above a threshold (we set it to 5 in the experiments), then at most $|R_0|$ documents in this group are defined as relevant answers to the query issued from a community. Finally, C clusters of documents are randomly selected to model community effects. In our experiments, we generate at most 5 communities for each query. Again, we attempt to define new relevant relationships between queries and documents following the relationships in the original dataset. All remaining documents are considered as noise for the query by all communities. The new relevance relationship is defined as: a document D_i is relevant to a new query, Q_j, with respect to a user in community C_k.

We now introduce how user profiles in every querying peer are modeled. A user can have several interests and thus belong to multiple communities. N peers are selected as query issuing peers, where N is smaller than the total number of queries. Let the total number of effective queries be N_Q. On average, each peer will issue N_Q/N queries. For each query the peer will issue, a subset of relevant

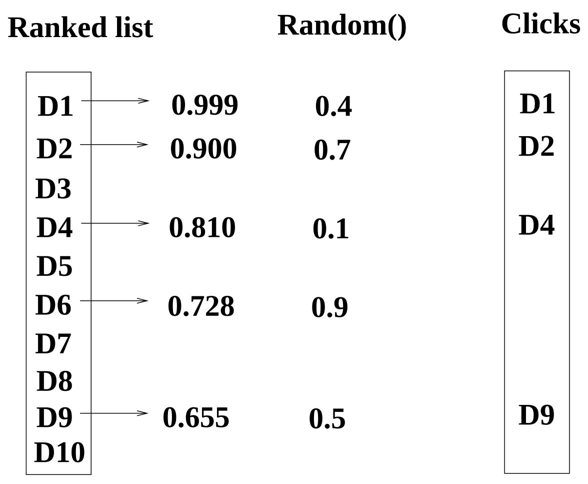


Figure 5.4: User clicks simulation

documents are chosen as the publications of this peer, so that a profile is generated for the user accordingly. For the TREC9 dataset, the total number of effective new queries should be $63 \times 5 \times 5 = 1575$. Because some cluster distributions are very skew (fewer than 5 effective clusters are generated from a query because one cluster size is extremely large), we generated 1356 new queries. The new queries are distributed among 1000 peers in the network, so a user issues 1.356 queries on average. For every query a peer issues, 10% of all relevant documents are randomly selected as the user's publications.

User click simulation

We also need to simulate user clicks when a ranked list is returned to a user. The key idea is that the higher a relevant document is ranked, the more likely it will be clicked. We set a click decay ratio to 0.9 and the probability of clicking the first relevant document is 0.999. So the probability that the user clicks on the ith relevant document is $0.999 * 0.9^{i-1}$. With a very low probability, an irrelevant document is clicked, which simulates mis-clickings on user behaviors. The mis-clicking probability is set to 0.001. Figure 5.4 illustrates how user clicks are simulated. The

Table 5.1: Experiment Settings.

Parameters	*Default values*	*Ranges*
number of answers	20	[10, 30]
number of indexed terms	30	[10, 40]
document profile size	20	[5, 30]
user profile size	10	[5, 30]
number of user clicks	2	[1, 4]

top 10 results are displayed to the user and 5 documents are relevant to the query. Their probabilities of being selected are pointed from the documents. A number is generated in the range of [0, 1) for every relevant document. Since the probability of document D_6 is smaller than the random number, it is not clicked. The other 4 documents are clicked finally. In this way, we can control the user cooperation level.

5.4.2 Experiment setup

We started with 63 queries in the TREC9 dataset and generated 1356 queries from them. While the queries are randomly issued from peers in the network, we measure the precision and recall for each of them. By default, the top 30 terms are indexed in a document; the top 20 terms are used to initialize the document profile; every user profile contains 10 terms; only 2 relevant documents are clicked; and the top 20 documents in the ranked list are evaluated. The default settings and the ranges of these parameters are listed in Table 5.1. We vary the parameters to evaluate the performance of CYBER and PIF.

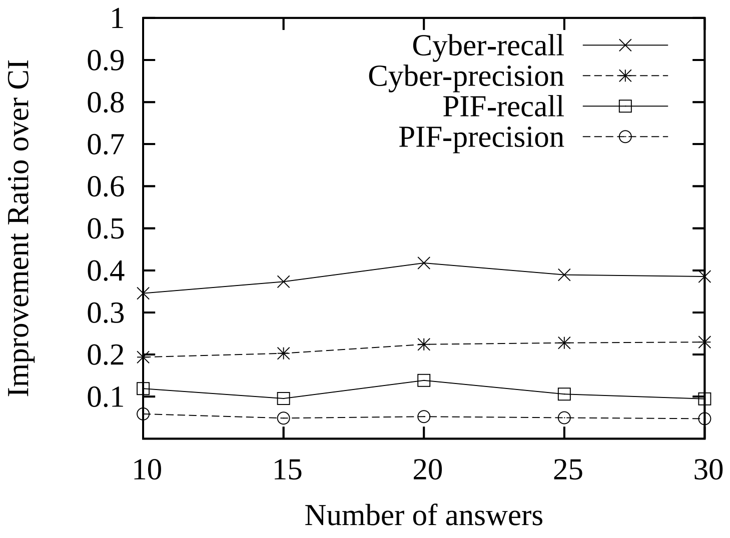


Figure 5.5: Varying number of answers.

5.4.3 Experimental results

As shown in the figures, both CYBER and PIF have higher recall and precision compared to CI. Though they only index a small number of terms (while CI indexes all terms), user feedback can contribute significantly improved performance. Furthermore, the small number of selected terms are important terms that are frequently used in queries.

First, we compare CYBER and the PIF scheme by varying the number of answers. The results are shown in Figure 5.5. CYBER outperforms the PIF scheme in terms of both precision and recall. This is a pleasant surprise as, for a particular query, the community feedback is less accurate than the single user feedback. Our investigation shows that CYBER's effectiveness arises because it accumulates positive feedback for every query. The user profiles keep updating the document profile so that the document profile evolves and becomes an accurate annotation to the document. On the contrary, in the PIF scheme, a query only benefits from the feedback once.

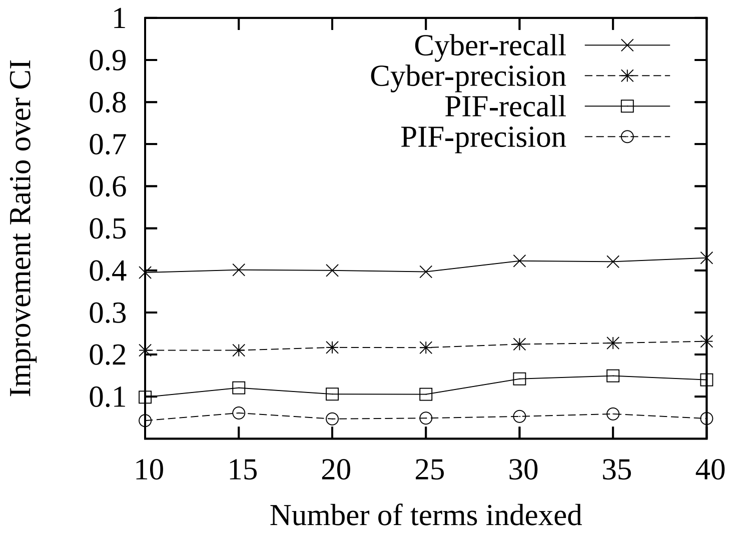


Figure 5.6: Varying number of index terms.

Next, we compare the effects of varying the number of indexed terms, since we built partial index for both CYBER and PIF. We can see clearly that CYBER outperforms the PIF scheme again in Figure 5.6. When the number of indexed terms is greater than 30, the performance of PIF does not change much any more, while CYBER's performance is relatively consistent. The reason is that the terms indexed in a later stage do not occur in many relevant documents. Therefore, only a small number of queries that overlap with such documents can benefit from feedbacks in PIF. On the contrary, in CYBER, such terms cause some relevant document profiles to be enriched. Thus, the weights of terms in related queries are increased by the profile similarity.

We then compare CYBER and the PIF scheme by varying the number of relevant documents clicked by a user. Figure 5.7 shows that CYBER is superior to PIF. When more documents are clicked, CYBER becomes less accurate, but PIF's accuracy increases. This is because document profiles are less accurate when lower ranked relevant documents are selected. One user clicks more documents means a document is clicked by more users in a community. The profile of such a document

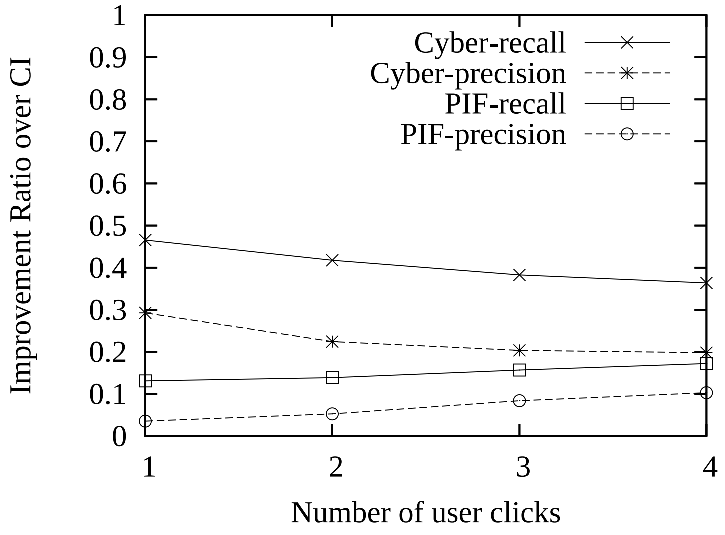


Figure 5.7: Varying number of clicked documents.

is updated according to more terms in the user profiles, so the chance of irrelevant terms are used is increased. However, a user usually stop clicking additional relevant documents once he found the information. Therefore, a small number of clicks is more realistic and preferred by CYBER.

Figure 5.8 shows the the robustness of CYBER to changes in query patterns, eg., a user may be interested in a completely new topic, on which his existing profiles do not help. To simulate that a new query pattern occurs, we randomly choose a number of terms from the document corpus to construct a user profile. Such user profiles have little overlap with the document profiles of the queried terms. Therefore the queries are "new" to the initial user profiles and the user interests change. In the figure, CYBER$'$ denotes the search results on new topics. We can see the performance of CYBER$'$ degrades a bit from CYBER, but it still clearly outperforms PIF. Initially, the existing profiles do not help on the queries. As a new profile is built up for the new topic, the latter queries are better evaluated and thus the answers become more accurate and the average precision and recall are improved.

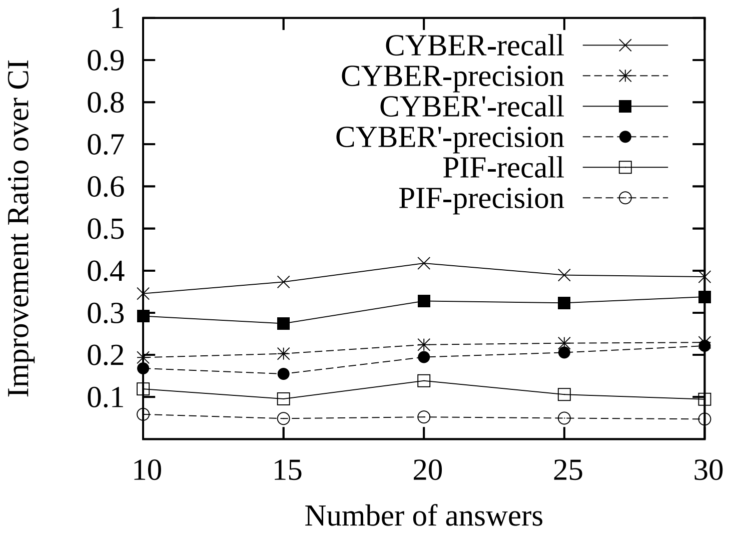


Figure 5.8: Changes in query pattern.

We then evaluate CYBER by varying the number of terms in the document profile in Figure 5.9. When more terms are inserted into the document profile, the precision and recall increase, until the document profile size reaches 20. This indicates that the document profiles are refined by user profiles in feedback information effectively, which shows the robustness of CYBER. Larger document profile increases the opportunity that a user profile overlaps with it. However, after the most common vocabularies of users are covered by the document profile, increasing the profile size does not help any more.

In Figure 5.10, we show CYBER's performance over various sizes of user profiles. When the user profile size is larger, the search result of CYBER is slightly more accurate. When the user profile contains 20 or more terms, CYBER performs consistently well. We believe this is due to two reasons. One is that only a small number of terms are commonly accepted by various users to describe their interest on a particular topic. The additional terms do not capture user interest accurately for many queries, instead, they introduce noise to user interest. The other is that there is no repeated queries, thus updates on document profiles will not be fully

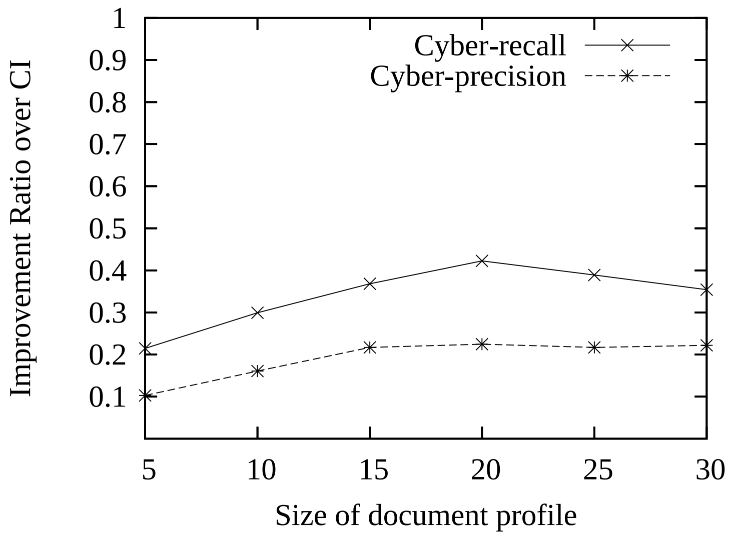


Figure 5.9: Varying size of document profiles.

utilized by user profiles.

Experimental results of CYBER+ and CYBER++

Having shown the effectiveness of CYBER, we now study the effectiveness of the two extended versions, CYBER+ and CYBER++.

Figure 5.11 shows the effectiveness on CYBER, CYBER+ and CYBER++. CYBER+ and CYBER give very similar performance in terms of both precision and recall. In the similarity calculation of the two schemes, query terms and profile terms are involved. The two kinds of terms construct a larger term set, which can capture the major meaning of a document. The majority of terms in the two sets of a document in CYBER and CYBER+ overlap. CYBER++ outperforms the other two significantly. This is because terms are selected and indexed more accurately. This shows that it is necessary to consider both past queries as well as community feedback for improved performance.

In Figure 5.12, we compare the three systems by varying the number of indexing terms. For CYBER+ and CYBER++, each learning iteration introduces 5 more

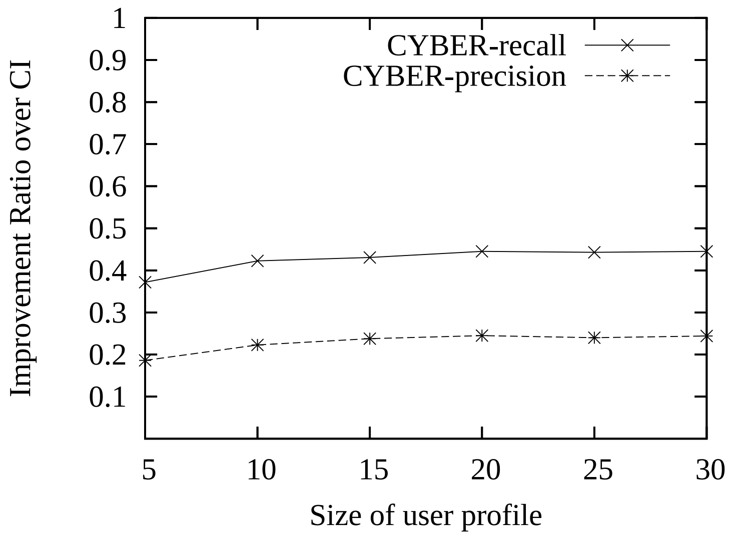


Figure 5.10: Varying size of user profiles.

terms to be indexed. We can see that CYBER and CYBER+ still perform similarly, while CYBER++ shows its advantages on refining its indices with relevance feedback. With more learning iterations, CYBER++ learns more indexing terms. Such terms are more precise because they appear in the queries, to which the document is identified as a relevant answer. More precise indexing terms indicate less noise terms are indexed, therefore, CYBER++ outperforms the other two schemes.

In summary, we see that CYBER outperforms the PIF scheme in terms of both precision and recall. Moreover, CYBER is robust in updating profiles and adaptive to the change of query patterns. The overhead for CYBER is small because document profiles tend to stabilize after some feedback information is consolidated. Therefore, we consider such cost can be amortized.

5.5 Summary

In this chapter, we have presented the design and evaluation of CYBER, a keyword search system utilizing community feedback information deployed on top of a DHT

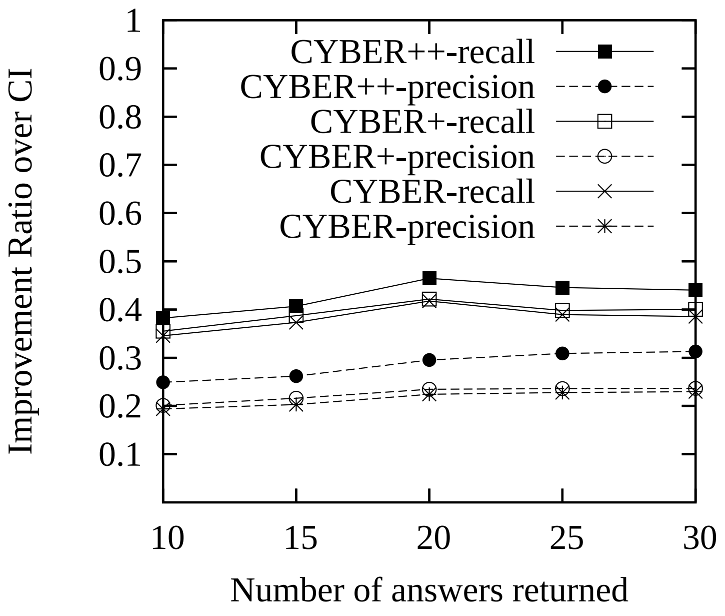


Figure 5.11: Comparison on varying the number of answers.

network. The traditional feedback techniques have been proved to be effective in improving search quality in centralized systems. However, applying the recursive procedure of traditional feedback techniques increases the response time dramatically in a P2P network. In CYBER, feedbacks from previous users are utilized to organize users as virtual communities. The feedbacks also help to annotate the relevant documents. A new query is evaluated once only based on past feedbacks from users with similar interests. In this manner, more accurate results are returned to the current user.

CYBER offers the following advantages over the PIF scheme. First, CYBER does not require explicit user efforts and reduces the response time. Second, the precision and recall of a search is improved significantly by CYBER with community feedback. Third, CYBER is robust in initializing and managing both user profiles and document profiles and adaptive to the change of query patterns. Our extensive simulation study confirms that CYBER outperforms the single-user feed-

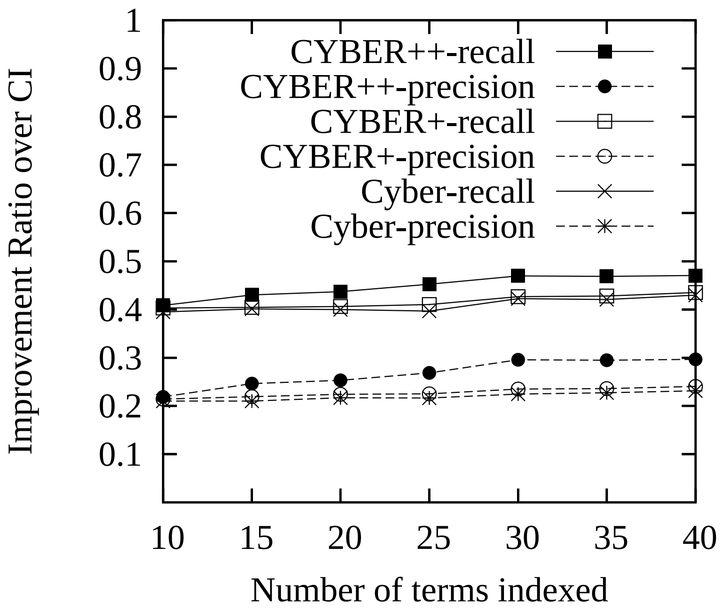


Figure 5.12: Comparison on varying the number of indexing terms.

back scheme in terms of both precision and recall.

Chapter 6

XCube: Processing *X*Path Queries in a Hyper*Cube* Overlay Network

6.1 Introduction

We have so far focused on keyword search over unstructured data like text documents. In this chapter, we look at the problem of querying XML documents in P2P networks. We focus on XPath and tag-based queries.

We propose *XCube*, a tag-based scheme that manages ***X***ML data in a hyper***Cube*** overlay network to support XPath (and tag-based) queries. In a d-dimensional hypercube, each node is represented as a d-bit vector, and a link is built between two adjacent nodes whose bit vectors differ from each other in one bit only. This property supports efficient routing for superset/subset search, which is the core of the XML document indexing and query routing strategies used in XCube. Each peer in XCube is responsible for managing a subset of nodes in the

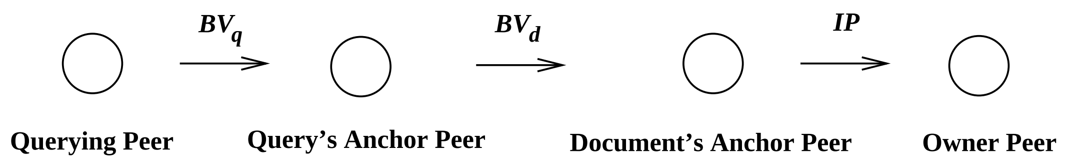


Figure 6.1: The querying flow in XCube.

hypercube, i.e., a smaller hypercube with dimension $d' < d$.

An XML document is compactly represented as a triple: a d-bit vector derived from the distinct tag names in the document, a synopsis of the document, and a bit map of the content summary. In this work, we adopt the strong DataGuide in [31] as the synopsis of an XML document. To facilitate speedy search, the metadata of a document is associated with an *anchor peer* (the peer that manages the node with matching bit vector). The metadata includes the IP address of the owner peer, the document identifier and the triple values. We also refer to the (bit vector and synopsis)-pair as the *structure summary* and the bit map as the *content summary*.

In addition, the synopsis alone is further indexed at all peers that manage hypercube nodes whose bit vectors are *covered by* the document bit vector. We refer to these peers as the *indexing peers*. We exclude the IP addresses, the document identifiers and the bit map from the indexing peers because (a) storing them will incur significant maintenance overhead (periodically ping messages) to ensure that the IP addresses are up-to-date, (b) values are much more dynamic than schemas, and (c) they can be obtained indirectly from the structure summaries, if needed.

When a user issues an XPath query, the query tags are first extracted. The query is then processed in four phases as shown in Figure 6.1. In phase 1, the bit vector, BV_q, derived from the query tags is used to locate the anchor peer of the query (which contains a superset of the synopses of all potentially matching answers). In phase 2, the query is compared against all the synopses at its anchor peer and then forwarded to the anchor peer of every document with matching synopsis according to its bit vector, BV_d. At the end of this phase, a coarse answer set is generated, i.e., for each relevant document, its synopsis satisfies the structural constraint. In phase 3, the predicates in the query are examined based on the bit maps stored in the anchor peer of every relevant document. The answer set is refined since many documents that can fulfill the structural requirements but not the predicates in the query are pruned away. In phase 4, the query is forwarded to every owner peer in the answer set based on its IP address, IP. The owner peers evaluate the query against the related XML documents and return the answers to the querying peer.

Since the number of peers is much smaller than the number of hypercube nodes, we also present a scheme to dynamically partition the hypercube to balance the loads across the peers. Furthermore, we exploit the *partition history* to remove redundant messages during routing. We have conducted an extensive performance evaluation of XCube. Our results show that XCube is efficient.

6.2 Preliminaries

In this section, we shall first give an overview of the hypercube structure. Following that, we shall look at how XML documents can be represented, and present a naive tag-based strategy for managing XML documents over a hypercube network.

6.2.1 The Hypercube Structure

In a d-dimensional hypercube, there are 2^d nodes. Each node is represented by a d-bit vector. Two nodes are directly connected when their bit vectors differ by exactly one bit. Thus, every node has exactly d neighbors. Figure 6.2 shows an

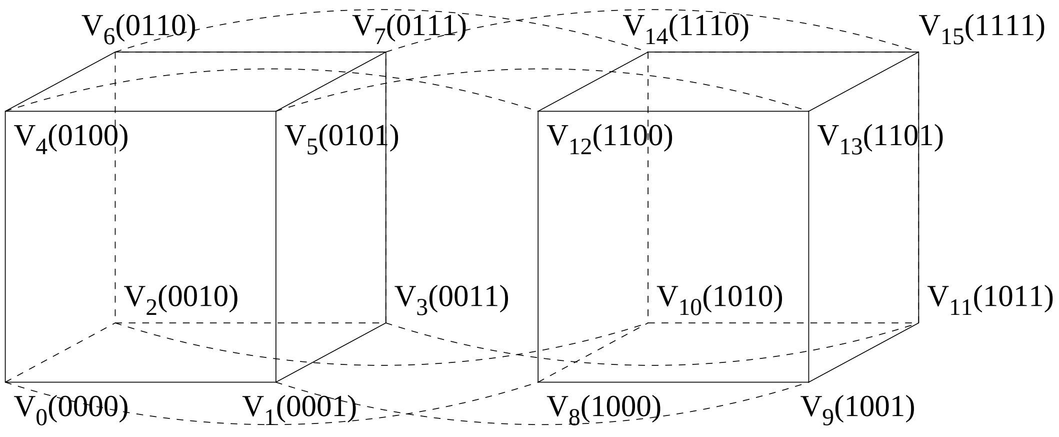


Figure 6.2: A 4-dimensional cube.

example of a complete 4-dimensional hypercube. As shown in the figure, every node is represented by a bit vector of length 4; node V_5 (0101) has four neighbors - V_1 (0001), V_4 (0100), V_7 (0111), and V_{13} (1101).

As a network structure, a hypercube has two very nice properties: (1) For a network with 2^d nodes, the network diameter is only d. This means that the searching/routing cost can be bounded logarithmically w.r.t. the network size. (2) Dissemination of data to nodes within a sub-cube of dimension d' ($d' \leq d$) can be done efficiently using broadcasting. The longest path for the broadcast in the sub-cube is d'. Moreover, the d' neighbors ensure effective parallelism of a broadcast.

In this thesis, we shall use the following definitions to describe the features of a hypercube.

Definition 1 *Given a bit vector, v, its **weight** is defined as the number of 1 bits in v, represented as $|v|$.*

In Figure 6.2, the weight of V_4 is 1; and that of V_0, V_9 and V_{15} are 0, 2, and 4 respectively.

Definition 2 *The **distance** between two nodes in a hypercube is the weight of the XOR (denoted by $\otimes$) of their bit vectors: Dist(V_1, V_2) = $|v_1 \otimes v_2|$*

In Figure 6.2, the distance between V_7 and V_8 is $|0111 \otimes 1000| = |1111| = 4$, and that between V_3 and V_{11} is 1. Intuitively, the distance between two nodes is the length of the shortest path between them. The routing algorithm is based on the *distance* definition.

Definition 3 *Let the bit vector of node V_i be v_i, and the bit vector of node V_j be v_j. We say that v_i **covers** v_j if for any bit in v_j that is 1, the corresponding bit in v_i is 1 also. For convenience, we will also say that node V_i **covers** V_j.*

In Figure 6.2, the bit vector of V_7 (0111) *covers* the bit vector of V_5 (0101). Likewise, we have node V_{13} covering V_{12}. Note that by definition, a bit vector always *covers* itself.

Definition 4 *Given a node, V, in a hypercube, we define its **SubCube**$^+$ as the cube constructed with all nodes whose bit vectors cover the bit vector of V, and its **SubCube**$^-$ as the cube constructed with all nodes whose bit vectors are covered by the bit vector of V.*

If $|V| = m$, then there are 2^{d-m} nodes in its *SubCube*$^+$ and 2^m nodes in its *SubCube*$^-$. In Figure 6.2, the *SubCube*$^+$ of node V_9 includes (V_9, V_{11}, V_{13}, V_{15}), and the *SubCube*$^-$ of node V_4 includes (V_4, V_0).

6.2.2 XML Documents and Representations

Given an XML document, we obtain its *structural summary* which comprises a (V_D, S_D)-pair where V_D is a d-bit vector and S_D is a synopsis. V_D is obtained as follows. Every tag in the XML tree[1] is hashed into the range [0, d-1]. The ith bit is set

[1] We will use the terms document structure, synopsis, and XML tree interchangeably in the chapter.

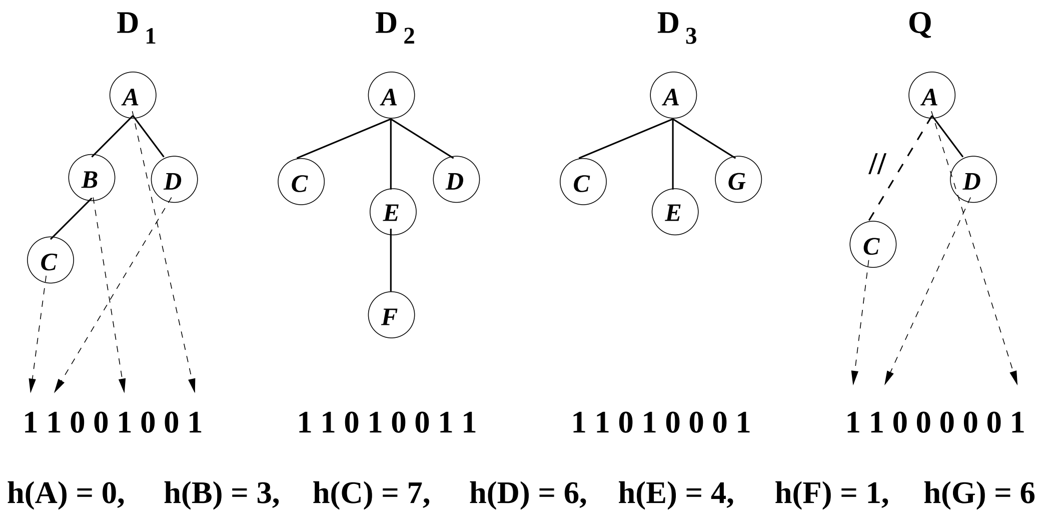


Figure 6.3: Bit vector calculation (d=8).

to 1 if there exists a tag name whose hash value is i; otherwise, the bit is 0. A query can also be treated as an XML tree and hashed in the same manner to obtain a bit vector. In Figure 6.3, we describe the calculation of document bit vector and query bit vector with dimension=8. The document and query trees and their corresponding bit vectors are shown at the top. The hash values of all tags (from A to G) are shown at the bottom. The calculation of the first tree, D_1, is depicted in detail. Tags A, B, C and D are hashed to 0, 3, 7 and 6 respectively, so the bit vector of D_1 is (11001001). The rightmost tree in Figure 6.3 represents an XPath query, Q: $/A[D]//C$, where only the structural constraints are considered. The dashed line in the query tree reflects that the user is unsure about the relationship between A and C, which is represented as "//" in the query. The bit vector of Q is (11000001).

The synopsis of a document, S_D, is essentially a tree representation, as adopted in many existing works [31, 41, 100, 61]. Users can easily apply one of the existing tools to generate a synopsis for an XML document. In this chapter, we adopt the strong DataGuide in [31] as the synopsis of an XML document. Figure 6.4 shows

```
<SigmodRecord>
    <issue>
        <volume/>
        <number/>
        <articles>
            <article>
                <title/>
                <initPage/>
                <endPage/>
                <authors>
                    <author/>
                </authors>
            </article>
        </articles>
    </issue>
</SigmodRecord>
```

Figure 6.4: The synopsis of SigmodRecord.xml

the synopsis of SigmodRecord.xml, which is stored as an XML document also. As can be seen, S_D is very compact, and its size is very minimal compared to the original document.

Summarizing contents is orthogonal to summarizing structures. Various algorithms can be applied to summarize content values in XML documents. We opt to employ bit map since it is compact (high compression ratio), easy to update (value changing may affect a few bits only), and the co-occurrence relationship among XML elements are taken into account, which is frequently checked in XPath queries. In an XML document, there are usually a number of subtrees of the same structure. Each subtree is used to describe the properties of an object. In this chapter, we only demonstrate how to summarize numerical values in an XML document of regular structure for simplicity. For example, SigmodRecord.xml contains a number of "issue" elements and each of them can be identified by two numbers: "volume" and "number" as shown in its synopsis in Figure 6.4. We summarize the values in such subtrees of the same structure in one bit map. Each type of element constructs one dimension of the bit map. We denote the dimensions of a bit map as a set of tag names. SigmodRecord.xml can be summarized with two bit maps of dimensions $\{volume, number\}$ and $\{initPage, endPage\}$. Let the number of leaves of the subtree be s and each dimension is evenly divided into p partitions. The volume (the number of distinguishable objects) of the bit map is p^s. On each dimension, a value is normalized into the range [0, 1], α, and it falls into the ith range, where $i = \lfloor \alpha \cdot p \rfloor$. Each subtree can be mapped to one bit in the s-dimensional bit map according to the values of its leave nodes, and then the bit is set to true. Due to the space limitation, let us consider a small version of SigmodRecord.xml that contains 3 "issue" elements with ("volume", "number") pair values (1, 1), (1, 3) and (2, 2). Each dimension is divided into 3 partitions and the value bound is 3. The matrix presentation of its bit map ($\{volume, number\}$) is shown below:

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

We illustrate how to map the element (1, 3) to a bit in the bit map. First, 1 is mapped to the first row ($1 / 3 \times 3$) and 3 is mapped to the third column ($3 / 3 \times 3$). So The corresponding bit is set to 1.

6.2.3 A Naive Tag-based Scheme over Hypercube Overlay

To appreciate the proposed XCube scheme, we first present a naive scheme (denoted NAIVE-XCube) for managing XML documents over a hypercube network. Given a set of N peers, and a d-dimensional hypercube with 2^d nodes($N << 2^d$), each peer manages a hypercube with dimension $d' \leq d$. Referring to Figure 6.2, if $N = 1$, then the entire hypercube is managed by a single peer. On the other hand, with $N = 5$, the hypercube may be partitioned to 5 “smaller” hypercubes - nodes $V_0 - V_7$, $V_{12} - V_{15}$, $V_8 - V_9$, V_{10}, V_{11} - each managed by one of the peers. We shall defer the discussion on how we can partition the hypercube to balance the load across the peers to Section 6.4. It suffices for now to note that each partition is essentially a hypercube and each peer has $\log N$ neighbors on average (d neighbors at most).

Under NAIVE-XCube, a document D's structure summary (i.e., d-bit vector and synopsis) and content summary are first extracted. D has a matching hypercube node with the same bit vector. We refer to this node as the *anchor node* of the bit vector (and of the document), and the peer that manages the anchor node the *anchor peer*. D is then indexed at its anchor peer. The anchor peer stores the corresponding metadata. In this work, the metadata are the owner peer's IP address, the document identifier, the structure summary and the content summary. Note that the owner peer has to ping the anchor peer periodically to ensure that it is online; otherwise, the document has to be (reinserted and) indexed in another peer (that is newly assigned to manage the anchor node).

When an XPath query, Q, is issued, its bit vector V_Q is derived (recall that we can treat an XPath query as a document). All the documents whose bit vectors *cover* V_Q are potential answers to Q. Thus, answering a query is equivalent to finding all bit vectors that cover the query bit vector. In fact, the nodes whose bit vectors cover the query bit vector are exactly the set of nodes that need to be searched. Recall that this set corresponds to the $SubCube^+$ of V_Q in the hypercube. Thus, all that is needed is to broadcast V_Q to all peers that manage nodes of the $SubCube^+$. As each peer receives the query, it compares the query against the synopses of all documents indexed by it. For each indexed document, if its synopsis can answer the query structurally and the bit map indicates that the document satisfies the predicates, the corresponding document identifier and the query are forwarded to the owner peer based on the IP address stored in the anchor peer. Finally, the owner peer processes the query and returns results (if any) to the querying peer directly.

As would be expected, this is an expensive operation since a large number of peers must be visited but not all of them contain relevant documents. The structural constraints and content predicates are checked in a very late stage, while the bit vector of the query alone is not very selective. Referring to our running example in Figure 6.3, D_1, D_2 and D_3 are indexed in the $SubCube^+$ of Q (11000001). While D_1 and D_2 are relevant documents, D_3 is a false positive. For a broadcast in a $SubCube^+$, the total number of nodes involved is $2^{d-|V_Q|}$ and therefore a large portion of peers (responsible for such nodes) would be visited. Similarly, broadcasting within a $SubCube^-$ incurs high routing cost also. The proposed XCube algorithm avoids such high cost on broadcasting.

Before leaving this section, we shall briefly discuss how a query is routed in the hypercube overlay. To route the query with bit vector V_Q from the query peer towards its destination peer (i.e., anchor peer of V_Q), a greedy mechanism can be adopted: each hop should bring the query at its current location/peer (with a bit vector V_C) closer to the destination peer through a neighbor node/peer (containing a bit vector V_N) such that distance(V_C, V_Q) $>$ distance(V_N, V_Q). In other words, each hop should take the query closer to its destination peer. Once the query reaches the destination peer, it can be broadcast to all peers that manage the nodes of the $SubCube^+$.

6.3 The XCUBE System

We note that NAIVE-XCube is essentially a two-phase strategy: In phase 1, which is tag-based, it finds all potentially matching documents. In fact, it does not miss any relevant documents. The answer set, however, can be very large. In phase 2, the structure summary and content summary are used to prune away irrelevant documents that provide relevant bit vectors but have different structures from the query or lack certain contents in the query predicates.

The proposed XCube system adopts a 4-phase strategy. XCube aims to overcome the poor performance of NAIVE-XCube that arises because the entire $SubCube^{+}$ needs to be searched during a retrieval process. The design of XCube is based on the following observations:

- If we had replicated the metadata from all nodes in a $SubCube^{+}$ at its anchor node, then we would have all information at the anchor node. Hence, the search would be efficient. This, however, implies that the overhead to insert a document becomes larger. Moreover, the maintenance cost may also be high. The challenge is to be able to control these overheads. Our solution is to selectively replicate part of the metadata, but not all. In particular, we do not need to maintain owner peer information at all indexing peers.

- Given that we choose to replicate the metadata, this can be easily done during a document insertion (i.e., when a document is shared or injected into the network). Essentially, a document D is relevant to all queries that its bit vector covers, which means that the metadata of D should be replicated at all nodes in its $SubCube^{-}$. In other words, a query whose bit vector matches a node in the $SubCube^{-}$ would find D potentially relevant. As such, our solution is to replicate certain metadata in a document's $SubCube^{-}$ when it is inserted.

By replicating some metadata in the corresponding $SubCubes^{-}$, queries can be examined on some constraints in an early stage, so that fewer peers are visited. We are now ready to present the XCube system.

6.3.1 Document Indexing

To share an XML document, the owner peer first extracts the structure summary, i.e., bit vector and synopsis, and the content summary, i.e., the bit maps. The metadata (IP address of owner, document ID, structure summary and content summary) is then routed to the peer responsible for the bit vector, the anchor peer of the document. The anchor peer determines the $SubCube^{-}$ of the inserted document. Recall that nodes in the $SubCube^{-}$ correspond to the potential query bit vectors covered by the bit vector of the inserted document. We shall refer to the set of peers that manage the nodes in a $SubCube^{-}$ as the *indexing peers*. The anchor peer then broadcasts the *existence* of the metadata to its indexing peers. By *existence*, we mean that the information maintained by each indexing peer is much less than the anchor peer - while the anchor peer maintains the complete metadata of the document, the indexing peers store the structure summary only.

As noted, by replicating the structure summary, searching can be performed very efficiently. We shall discuss how searching is performed shortly. Now, let us examine why this scheme is reasonable in terms of maintenance and insertion overhead. We argue that the maintenance overhead of this scheme is no worse than that of NAIVE-XCube. First, the owner peer only needs to keep in touch with the anchor peer. Since the indexing peers do not store any information about the owner peer, they need not be checked periodically. To determine the owner peer from the indexing peer, all that is needed is an “indirect addressing” - to find the owner of a synopsis, locate the anchor peer of the corresponding document using the accompanying bit vector, and from the anchor peer, the IP address of the owner can be obtained. Second, if the anchor peer goes offline, an existing peer takes over its partition and indices. The owner peer does not need to broadcast the existing synopses again. Only the new anchor peer is contacted. Third, when a new peer joins the network, it learns the existing structure summaries from the peer accepting it. Hence, we do not need to periodically broadcast the existence of every synopsis because of the dynamism of the network.

The insertion overhead cannot be avoided, but XCube attempts to minimize redundant messages in two ways. First, as we shall discuss in Section 6.4, we exploit a partition history to remove redundant messages from being disseminated when broadcasting a structure summary to the indexing peers. Second, for multiple documents that share the same synopsis, the information need not be re-distributed. In other words, an anchor peer only needs to replicate each unique structure summary. In addition, we note that the insertion overhead is incurred only once when a document of new synopsis is shared. The broadcast overhead is essentially amortized over the life time of the document, so the amortized cost is very low. Moreover, in the entire P2P system, documents are shared at different time, so the insertion overhead can hardly cause network congestion. There is no extra network maintenance cost for these indexing peers. The broadcast is efficient and the summary can reach the essential peers in a few hops as the diameter of any *SubCube* is bounded by d.

In terms of storage, assume the average size of a summary is 1K bytes. Storing 1000 such summaries only consumes 1M bytes, which is very manageable with modern hardwares.

An example on indexing the 3 XML documents in Figure 6.3 is shown in Figure

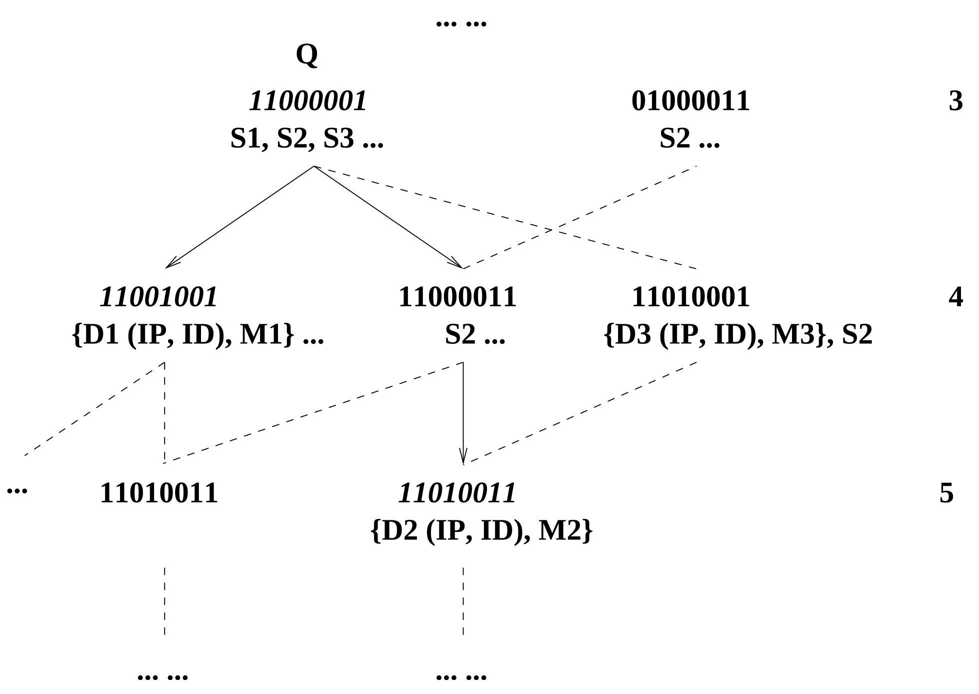


Figure 6.5: Document indexing and query routing.

6.5. Only a partial set of nodes at 3 different levels (here, a level is determined by the weight of every bit vector) in a 8-dimensional hypercube are depicted due to space limitation. D_i represents the index entry of an XML document, which contains the structure summary, bit maps, owner's IP address and the document identifier. S_i represents the structure summary only and M_i represents the bit maps of document i. The dashed lines and the solid arrows are the links amongst the nodes. When a new document is indexed, for instance, D_2, it is sent to the anchor peer (11010011) first. The anchor peer then broadcasts the existence of D_2 towards upper levels, level 4 and level 3 in the figure (higher levels denote nodes in the $SubCube^-$). Every peer receiving the message stores the summary of D_2 (i.e. S_2) locally. At the top-left peer (11000001), 3 summaries are stored, S_1, S_2 and S_3.

6.3.2 Querying Documents

Given an XPath query, XCube processes it in four phases. In phase 1, the querying peer obtains its bit vector by hashing all of the tags in it. The query and its bit vector are encapsulated into a message. The message is then routed to the anchor peer responsible for the hypercube node matching the query bit vector. Since this phase is essentially based on the tags, the query's anchor peer stores the synopses of all potentially matching documents. Thus, the anchor peer stores a superset of the answer synopses, containing both answers and false positives. This phase takes $\log N$ hops to route the query message to its corresponding anchor peer (It has been proven that a point query can be accomplished in $\log N$ hops on average in [69]).

In phase 2, the query's anchor peer examines all synopses stored locally against the query structure. Here, the structural constraints, such as parent-child relationships(/), ancestor-descendant relationships(//) and wildcard(*), are fully checked. A subset of the synopses are obtained as the refined answer set, whose anchor peers should be visited to process the query's predicates, such as equality and range containment. In case that an anchor peer is offline, an arbitrary 1-bit in the query's bit vector is set to 0. The query is routed to the peer responsible for the new bit vector, which also has the complete structure summaries that can answer the query. In this way, all and only the documents that can answer the query structurally are searched.

In phase 3, the query is forwarded to the anchor peers of structurally matching documents. This is accomplished by routing the message according to the bit vector of every related document. In the anchor peer of a potentially matching document, the predicates in the query is evaluated based on the bit maps stored locally. Many XML documents share a common synopsis, but their contents may be very different. By evaluating the predicates in the query, a lot of documents that do not contain contents requested in the query are eliminated. In this manner, the answer set is further refined.

Finally in phase 4, each document's anchor peer forwards the query to the related owner peer according to its IP address. In an owner peer, the query can be processed against the related XML document with many existing tools/softwares, and then the final results are returned to the querying peer. We will not discuss this phase further because the XML query processing techniques are very mature.

In an XPath query, the ancestor-descendant relationship "//" and the wildcard "*" are always used when a user is unsure about some structural details of a document. In XCube, such uncertain structural constraints are not encoded to guide query routing in phase 1, but they are checked in phase 2. Hence, the query expressiveness is not limited and no false positive is included after phase 2. Moreover, the efficiency is not affected by this type of expensive structural constraint, which is a strong advantage over the path-based mechanisms.

The algorithm to route a query is presented in Algorithm 3. Initially, the query is routed to the anchor peer responsible for its bit vector (line 3). The anchor peer examines the structure summaries stored in it and determines the answer set (line 4). For each answer (a document summary) in the set, the anchor peer can obtain its bit vector and judge which direction the query should be forwarded to. For the answers in the same direction, the anchor peer sends one message including their summaries and the query to the neighbor peer in that direction (lines 13-17). The neighbor peers continue to check their stored summaries and forward the query in groups. When a peer storing the metadata of related document receives the query, it will first examine the query against the bit maps of the document. If the bit maps can answer the predicates in the query, the peer then forwards the query to the

Algorithm 3: The query routing algorithm.

```
1  Let $Q$ be the query;
2  Let $P_q$ be the query originating peer;
3  Peer $P_a = P_q$.route($Q$);
4  Initialize $D$ as the set of bit vectors indexed in $P_a$ whose synopses can
   answer $Q$ structurally;
5  Let Peer $P_i$ is $P_a$'s neighbor in the $i$th dimension;
6  Let $D_i$ be the answer set to forward to $P_i$;
7  for each index entry with complete metadata do
8      if the bit vector of the entry appears in $D$ then
9          if the corresponding bit maps can answer $Q$ on the predicates then
10             Forward $Q$ to the related owner peer;
11             Remove the bit vector from $D$;
12 for each bit vector $v_j$ in $D$ do
13     for each dimension $i$ in the partition history (from the oldest to the
       newest) do
14         if the $i$th bit in $v_j$ is 1 then
15             put $v_j$ in $D_i$;
16             break;
17 for each dimension $i$ do
18     Forward the message containing $D_i$ to $P_i$;
19     $P_i$ repeats the procedure until $D_i$ is empty;
```

corresponding owner peer based on its IP address (lines 7-12). Note that the peer does not need to check the structural constraint again, because it has been done in the anchor peer of the query. This mechanism combines routing paths to a set of messages, which reduces the total number of hops without affecting parallelism. Answers are produced incrementally. Users can receive some answers very fast, which is valuable in a distributed environment. Users can react fast to judge the query/answer quality.

In Figure 6.5, query Q is sent to its anchor peer (the left peer at level 3). The anchor peer has three structure summaries, S_1, S_2 and S_3. Though all of their bit vectors can cover the bit vector of Q, we can easily see that only D_1 and D_2 can answer the query from Figure 6.3. If the anchor peer does not have the summaries of the related documents, it can only decide based on the bit vectors: forward Q to all of the related peers. With the guidance of the summaries (more specifically, the bit vectors of the relevant synopses), the anchor peer can now avoid forwarding the query to the anchor peer of D_3 (11010001). For a relevant document, say D_2, the anchor peer examines the corresponding bit vector of its synopsis, and forwards Q to the peer responsible for the node with the bit vector (the right peer at level 5). This peer is essentially the anchor peer of D_2, which would have stored the complete metadata of D_2. The anchor peer of D_2 then evaluate Q using M_2. If M_2 can answer Q, then Q is sent to the owner peer of D_2 based on the IP address in the complete metadata. All owner peers of the relevant documents receive Q in a similar manner.

6.4 Load Balancing Issues

For XCube to be efficient, there are two challenges on load balancing to be addressed: (a) how to partition the hypercube such that the system load is balanced across all peers and redundant messages are removed? and (b) how to store the synopsis copies evenly among the indexing peers?

6.4.1 Load-Balanced Partitioning of the Hypercube

In practice, the number of peers in a P2P network (N) is much smaller than the number of nodes in a d-dimensional hypercube (2^d). Hence, we need to partition the hypercube into N subcubes, and assign them to the physical peers. Partitioning the hypercube introduces two issues. First, load imbalance arises because some partitions may manage more synopses than others. We can view the data space

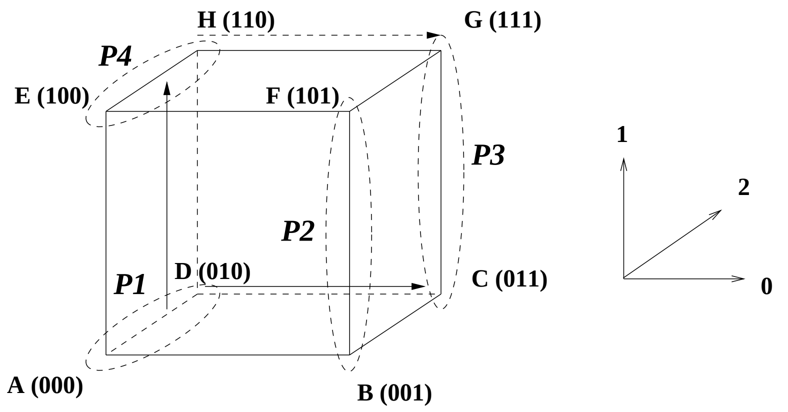


Figure 6.6: A dynamically partitioned 3-d cube.

as all points in the d-dimensional space (a.k.a. each node is a data point) in XCube. Moreover, each XML document corresponds to a point in the d-dimensional space based on its bit vector. Unlike the inverted-file tag-based scheme, popular terms/tags are distributed among all structures containing the hot tags. Hence, as long as documents do not share some terms/tags, they would very likely be mapped to different bit vectors and hence scattered across different hypercube nodes. However, it is still not uncommon to find that some topics are very popular, which leads to a lot of similar structures. This means that these documents would be clustered into the same regions in the hypercube space.

Second, the same message may be transmitted multiple times through a peer when documents are inserted as a result of broadcasting the synopses to the peers that manage nodes in the $SubCube^{-}$. As an example, consider the 3-dimensional hypercube in Figure 6.6. Here, we assume that there are four peers, P_1, P_2, P_3 and P_4, which share four documents whose bit vectors are located at nodes A, B, C and H respectively. Suppose, initially, there is only P_1 in the system, which is assigned all the 8 hypercube nodes to manage. When P_2 joins, the hypercube is partitioned in dimension 0 (so that P_1 manages nodes A, D, E and H, while P_2 manages B, C, F and G). Next P_4 joins P_1, and P_3 joins P_2. The two subcubes are partitioned in dimensions 1 and 2 respectively. Assume a message, whose bit vector is (010), is sent to P_1, and it is supposed to be broadcast in the *SubCube*$^+$ of D(010). P_1 forwards the message to P_3 (D$\rightarrow$C) and P_4 (D$\rightarrow$H). P_4 further forwards the message to P_3 (H$\rightarrow$G). We have a redundant message from P_4 to P_3, i.e., the forwarding message represented by the dashed arrow from P_4(H) to P_3(G) should not have been transmitted.

In HyperCuP [69], the dimensions are preordered and the hypercube is partitioned by cutting the edges in the predefined order. Because of the preordered partitioning, redundant messages can be avoided completely. However, this method is not adaptive to unbalanced load.

There have been several load balancing techniques introduced in the literature (e.g., [28]). The key is for a peer to adjust its load with its neighbor peers in a certain dimension *dynamically*. However, as we have seen in our example above, a dynamic and random partitioning will lead to redundant messages being transmitted.

In XCube, we opted to build on existing load balancing techniques and address the redundant message problem within such a load-balanced mechanism. Our load-balanced partitioning scheme works as follows. A peer joins the network based on the data item (in our case, the bit vector of an XML document) it is going to share. The peer accepting the newly joined peer splits the load in its original sub-hypercube (together with the data points of the new peer) and assigns one part to the new peer. In this way, the dense area where many documents are located can be finer partitioned. While this method leads to load balancing, it is unable to remove the redundant messages from different neighbors. We introduce *partition history* to overcome the problem. For simplicity in discussion, we shall present our solution in the context of broadcasting in *SubCubes*$^+$. Our solution is also applicable if messages are broadcast in *SubCubes*$^-$ and in a general partitioning scheme.

To handle the redundant messages, each peer stores a *partition history.* This is simply an array of integers representing the sequence of splits that eventually lead to the sub-hypercube maintained by a peer. This history is obtained as follows. When a peer joins the network, it learns the partition history from the peer accepting it. The sub-hypercube managed by the accepting peer is split and the history is appended with the new dimension, in which the two peers point to each other. We note that each dimension is split at most once, and hence the partition history is at most of size d. Thus, the partition history introduces very marginal additional storage cost. Let us illustrate this using Figure 6.6. With partition histories, P_1 and P_2 partitioned the cube along dimension 0 first, so their partition histories are the same, [0], at first. Then P_3 and P_2 partition P_2's sub-hypercube along dimension 2, so their partition histories are the same: [0, 2]. Similarly, the partition history of P_4 and P_1 are the same as well: [0, 1].

The partition history is used to guide a peer in broadcasting a message to avoid redundant messages. We shall describe how this is achieved. The starting point is at the anchor peer that is supposed to broadcast the message. Let the broadcast space be *Sub-Cube*. Each peer along the broadcast space that receives the message (and the broadcast space *Sub-Cube*) performs the operations in Algorithm 4. Every peer receiving an incoming space, *Sub-Cube*, to broadcast a message, splits it along each dimension in its partition history and forwards the new *Sub-Cube* to its neighbor in the corresponding dimension. Note that given a hypercube, it is partitionable in an arbitrary dimension at most once. The two *Sub-Cubes* cannot be further split along that dimension. Hence, every node in the *Sub-Cube* is visited exactly once. Coming back to our running example in Figure 6.6, P_1 broadcasts a message to P_3 and P_4. P_4 stops forwarding the message because the *Sub-Cube* received by P_4

Algorithm 4: The broadcast algorithm.

```
1 Let $Sub\text{-}Cube$ be the hypercube in which the message should be broadcast;
2 Let $P$ be the peer that receives $Sub\text{-}Cube$;
3 Let $H_P$ be the partition history of $P$ (from the earliest to the latest);
4 for each dimension, $d$ in $H_P$ do
5     if the $Sub\text{-}Cube$ is not split along $d$ then
6         Partition the $Sub\text{-}Cube$ along $d$ into $Sub\text{-}Cube_1$ and $Sub\text{-}Cube_2$;
7         $Sub\text{-}Cube = Sub\text{-}Cube_1$;
8         Forward the $Sub\text{-}Cube_2$ to the neighbor peer in dimension $d$;
```

contains one node only, i.e. H(110). Hence, the redundant message from P_4 to P_3 is avoided.

6.4.2 Balancing Storage Load

When indexing a synopsis, all nodes in the $SubCube^-$ determined by the bit vector of the synopsis should be contacted. If a node corresponds to a bit vector with a small number of 1-bits (we refer to such bit vectors as *light* weight vectors), it is very possible that this node is involved in more $SubCubes^-$. Recall the example in Figure 6.3, the node corresponding with the bit vector (10000000) is contained in the $SubCubes^-$ determined by bit vectors (11001001), (11010011) and (11010001) of the three synopses. A peer responsible for such a node will store many synopses and be contacted by many peers. While this may be a concern, it is manageable. The reason is because storage cost is not a major concern for modern hardwares. Moreover, each synopsis is only indexed once in the network, and used many times. Thus, the (amortized) indexing overhead is small compared to the query routing cost. In addition, it is worth noting that the number of synopses indexed in a peer does not affect the performance of routing queries, since the number of answers determines the number of messages.

In this thesis, we offer a solution to handle the unbalanced storage load. Our solution is based on the following observations. On average, the hypercube node corresponding to a bit vector of weight b has to store $\frac{1}{C_b^d}$ fraction of the total synopses, where d is the dimensionality of the hypercube. For instance, when the weight is 1, the node is responsible for $\frac{1}{d}$ fraction of the total synopses. Therefore, if more peers partition the hypercube near the *all-zero* bit vector (000...0), the load will be more balanced.

After a peer joins the network, it takes over an additional subcube as a "virtual peer" if the peer is connecting to the network with a broadband or more advanced connection. The peer randomly selects a bit vector from the hypercube nodes it is currently responsible for. The chosen bit vector is then altered by randomly turning some 1 bits to 0, so that the new bit vector is nearer to the *all-zero* bit vector. According to the new bit vector, the peer joins the network again as a "virtual peer" (without giving up the current subcube). In this manner, nodes corresponding to bit vectors of lighter weights are taken over by more peers, and thus the load is more balanced. In the experiment, we restrict that each peer can take over at most two subcubes.

6.5 Experimental Study

In this section, we report an experimental study to evaluate the performance of XCube. We compare XCube against the following schemes which have the same query expressiveness (recall that path-based schemes cannot efficiently support ancestoral-descendant relationships and wildcards):

- **NAIVE-XCube**. Recall that NAIVE-XCube (see Section 6.2) broadcasts queries to indexing peers (those managing nodes in the $SubCube^+$ of the query bit vector). By comparing XCube and NAIVE-XCube, we can study the performance gain in XCube as a result of replicating the structure summaries.

- **PC-XCube**. PC-XCube builds on and extends XCube to incorporate parent-child (PC) relationships in generating bit-vectors. All other protocols (e.g., routing, querying, searching) remain the same as that of XCube. Basically, every PC relationship is treated as a tag, which can be represented as P/C, where P is the parent node name and C is the child node name in the synopsis. The bit vector is derived by hashing all tag names and P/C patterns. For queries, their tags and PC relationships are also encoded in the same manner. However, ancestor-descendant ("//") relationships are omitted. Note that this poses no problem in terms of finding the final answers as such relationships are checked in the query's anchor node (through the comparison between the query and the synopses). Given that PC-XCube encodes more information, it is expected to lead to fewer false positives in phase 1, a shorter computation time at the anchor peer, and fewer peers to be searched for matching synopsis in phase 2. However, as more bits in the bit vector has to be set, it also means that peers with matching synopses may be further away (in terms of hops). Comparing XCube and PC-XCube allows us to evaluate the benefit of complex encoding schemes (in our case, the additional PC relationships).

- **IFT**. IFT is the inverted-file tag-based scheme. By comparing XCube and IFT, we show the advantages of indexing XML documents on the complete synopsis over separate tags in it.

6.5.1 Data and Query Generation

The XML research community has relied on many real data and benchmark data to test and evaluate their works in centralized systems, such as INEX[2] (real), XMark[3] (benchmark) and XBench [97] (benchmark). However, these data are not suitable for a large scale P2P network where the XML documents are diverse, their structures are more heterogeneous and many similar structures are used to describe data on the same topics. Therefore, we generated syntactic data (essentially the structure summary of XML documents) to evaluate the performance of XCube.

To generate heterogeneous structures, we project a number of (document and query) trees from a real, large industrial DTD, NITF[4], and distort their tag names afterwards. In this way, we can model users describing the data with a subset of tags even if they agree on several common DTDs. Such scenarios are supported by DTDs mainly with the two options, '*' and '?'. Therefore, we simulate such cases with smaller XML trees. First, we define the number of clusters existing in the network to be 50 to simulate many different user interests. The structure summaries in each cluster describe documents on the same topic and thus have similar structures and tags. Each synopsis is essentially represented as a tree (IDREF is omitted for simplicity, but XCube can also index synopses with IDREF or route queries with such constraints). The cluster sizes follow a Gaussian distribution. Second, a topic tree is projected from the original XML tree for each cluster. Each tag name in the topic tree of a cluster is concatenated with the cluster ID, so that there are many tags of different names in the entire corpus. Finally, a number of instance trees are projected from the topic tree. In a new tree, all relationships between two directly connected nodes are set to parent-child relationships. The instance

[2]http://inex.is.informatik.uni-duisburg.de
[3]http://monetdb.cwi.nl/xml
[4]http://www.nitf.org

Table 6.1: Experiment Settings.

Parameters	*Default values*	*Ranges*
virtual peer ratio	0.4	[0.1, 0.5]
query size	6	[3, 10]
network size	2000	[1000, 10000]
synopsis size	40	[10, 100]
bit map dimension	4	[2, 6]
bit map partitions per dimension	20	[5, 30]

trees have different number of tags. The default range is 40-45 tags in our study. Each instance tree is treated as a synopsis and 100 XML documents are generated from it with different contents. The values of a type of element follow Gaussian distribution. Query trees are projected from instance trees in a similar manner with fewer tags. The predicates in a query is generated based on some existing elements in a document. To show the effectiveness of the bit map, one document is summarized with one bit map[5] and only equality queries are considered.

6.5.2 Experiment Settings

We implemented four tag-based schemes: XCube, NAIVE-XCube, PC-XCube, and IFT. We have conducted many studies, and present representative results on the effects of some parameters: query size, network size and synopsis size. The default settings and the range of these parameters are listed in Table 6.1. For fair comparison, we adapt the load balance technique [28] to the IFT-base scheme as well. All terms are hashed using MD5 hash function. The length of the hash value is 120 bits, so we set the dimension of the hypercube to 120. Because the processing on structural constraints and predicate constraints are orthogonal, we present their costs separately. The results before subsection 6.5.4 are for finding relevant syn-

[5]If a document can be summarized with multiple bit maps, the bit maps can be built with more bits, and thus they are more accurate.

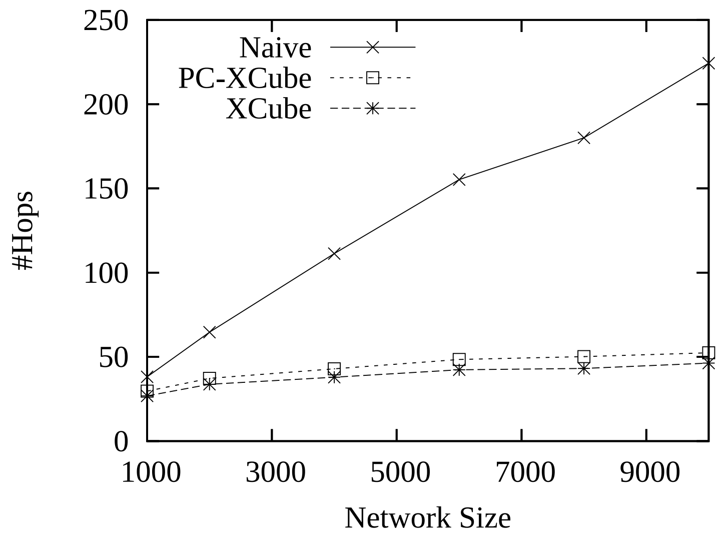


Figure 6.7: Comparison among XCube, NAIVE-XCube and PC-XCube.

opses to a query (on structural constraints only); and subsection 6.5.4 presents the cost on routing a query to the related owner peers according to the predicates. The routing cost on predicates is the same for all the implemented tag-based schemes. Our study is based on simulation, and all experiments are conducted on an Intel Xeon 3.0GH CPU Server with 18GB RAM (2GB in use).

6.5.3 Comparing XCube, NAIVE-XCube and PC-XCube

Since XCube, NAIVE-XCube and PC-XCube are essentially variants of the tag-based approach, we first compare their performance.

Figure 6.7 shows the performance of XCube, NAIVE-XCube and PC-XCube in networks of various sizes. As expected, NAIVE-XCube does not scale with the network size. Both PC-XCube and XCube are highly scalable. However, contrary to expectation, PC-XCube costs more hops than XCube. Our investigations conclude that capturing additional structural information does not guarantee all false positive results are filtered out. Therefore, the anchor peer of a query still must

Table 6.2: Local process at anchor peers.

	PC-XCube			XCube		
query size	#CS	#FS	Time(ms)	#CS	#FS	Time(ms)
4	579.00	7.86	7.55	253.56	8.29	7.58
5	251.83	4.16	4.08	101.53	4.56	4.32
6	99.91	2.64	2.66	46.14	2.95	2.89
7	39.41	1.92	2.01	22.91	2.16	2.21
8	17.02	1.57	1.74	13.34	1.68	1.81
9	8.36	1.33	1.53	8.99	1.4	1.57

check all answer candidates against the query and then forward the query to all related indexing peers. The cost, in terms of number of hops, to find the anchor peer for a query are about the same for the two schemes, both are bounded by $\log N$ (phase 1). The key factor for the different cost is the number of indexing peers to visit (phase 2). In XCube, the number of 1-bits in a bit vector is bounded by the number of tags, while in PC-XCube, the bound is doubled[6]. The bit vectors are sparsely distributed in the hypercube space in PC-XCube, so XML documents with similar synopses are indexed in peers with longer distance. On the other hand, in XCube, similar synopses are indexed in closer peers and hence fewer number of hops are needed.

We note that XCube needs to process an XPath query against some synopses stored at the anchor peer of the query. As the number of synopses might be large, the computational overhead may be significant. This study evaluates the processing time for anchor peers to choose relevant synopses. Every synopsis is stored as an XML document in the hard disk of an anchor peer. The anchor peer selects all synopses that can answer the query structurally in two steps. In the first step, for each synopsis whose bit vector *covers* the query bit vector (termed as *coarse synopsis*), the anchor peer checks if it contains all the tag names in the query. This

[6]The number of edges in a tree is $(T-1)$, where T is the number of nodes in the tree. Therefore, the number of 1-bits in the bit vector is bounded by $2N$.

is to prune away false positives that arise because of collisions in the mapping, i.e, different tags are mapped to the same position in the bit vector. In the second step, Xalan[7] is employed to evaluate the query on structural constraints against every synopsis chosen in step one (termed as *fine synopsis*). The total processing time of the two steps is measured. Table 6.2 shows the results, with abbreviations: #CS for the number of coarse synopses, #FS for the number of fine synopses and Time(ms) for the process time in millisecond. PC-XCube checks more coarse synopses than XCube because more 1-bits in bit vectors introduce more false positives. However, it can be done much more efficiently (simply containment checking) comparing with structure checking, which is a tree matching operation. The number of fine synopses PC-XCube needs to check is fewer than that of XCube because the parent-child relationships prune away some synopses. Therefore, PC-XCube is slightly faster than XCube for the entire processing time. Although tree matching is expensive, the process turns out to be very efficient due to two reasons. First, the number of nodes in a synopsis is much smaller than the number of nodes in a document. Second, the number of candidates to check is small also. The shorter a query is, the more potential relevant synopses to check. As shown, the average processing time in an anchor peer is negligible. Moreover, an anchor peer does not need to wait till all synopses are checked. Once a synopsis is judged as relevant, a message is sent to its indexing peer.

To summarize, XCube outperforms NAIVE-XCube and PC-XCube. In particular, the fact that it is superior over PC-XCube makes it attractive given that it is simpler to implement. For the rest of this section, we shall focus on XCube only.

[7]http://xml.apache.org/xalan-j/

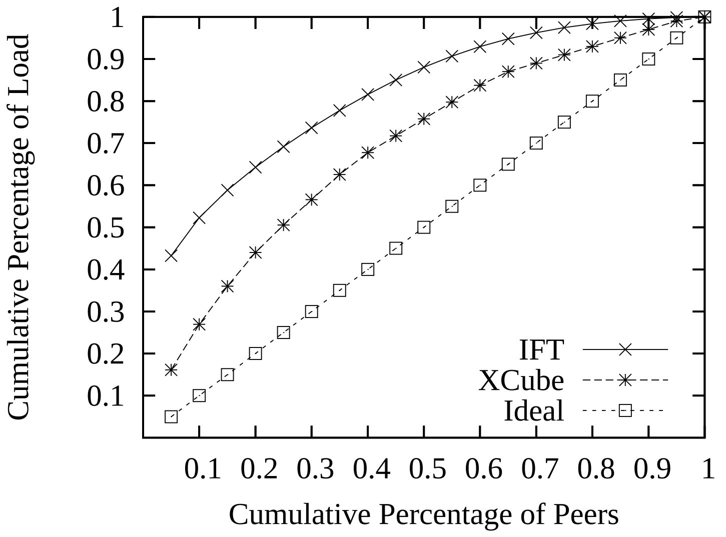


Figure 6.8: Overhead load distribution comparison.

6.5.4 Comparing XCube and IFT

In this section, we shall evaluate the performance of XCube and IFT.

Load Distribution

First, we study the load distribution on the overhead across the peers in the system. The overhead includes the index maintenance overhead (owner peers need to ping anchor peers periodically) and the query processing overhead (anchor peers need to process the query against the bit maps and forward the query to related owner peers). Therefore, we can easily see that the overhead is proportional to the number of complete metadata a peer stores. Figure 6.8 shows the load for 2000 peers, with 40 distinct tag names in each document. Peers are sorted according to the number of indices they store in descending order. The x-axis shows the percentage of peers and the y-axis shows the percentage of indices they maintain in the system. The perfectly balanced load is presented in the dashed straight line (Ideal). Clearly, indices are more balanced in XCube than in IFT. The reason is that the load

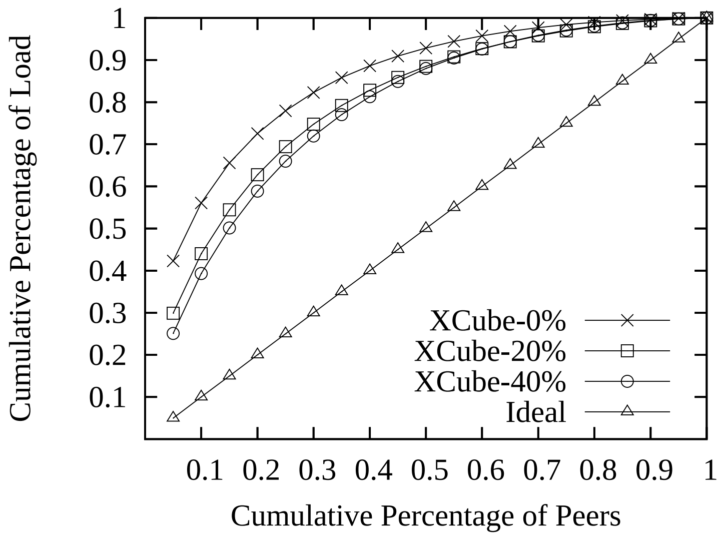


Figure 6.9: Storage load distribution with various number of virtual peers.

of popular tag names are shared by many different related synopses in XCube. The peers with more metadata entries are typically more likely to be contacted by some query anchor peers and thus receive more queries. We assume the query distribution follows the data distribution. However, it is straightforward to tune the data distribution by introducing weight to documents according to their popularity if necessary.

Figure 6.9 presents the cumulative percentage of synopses stored by cumulative percentage of peers to show the effectiveness of virtual peers. Peers are sorted according to the number of synopses they store in descending order. XCube-x% represents that x% peers are chosen to join the network again as virtual peers. Ideally, every peer should store a similar number of synopses (the "Ideal" line). Without any virtual peers, 5% of peers store more than 40% of synopses; while some virtual peers are involved, it is reduced to 25%. In [67], the study has shown that at least 70% peers have broadband or even faster connections, while at most 30% peers make the connections through dial-up modems in Napster and Gnutella.

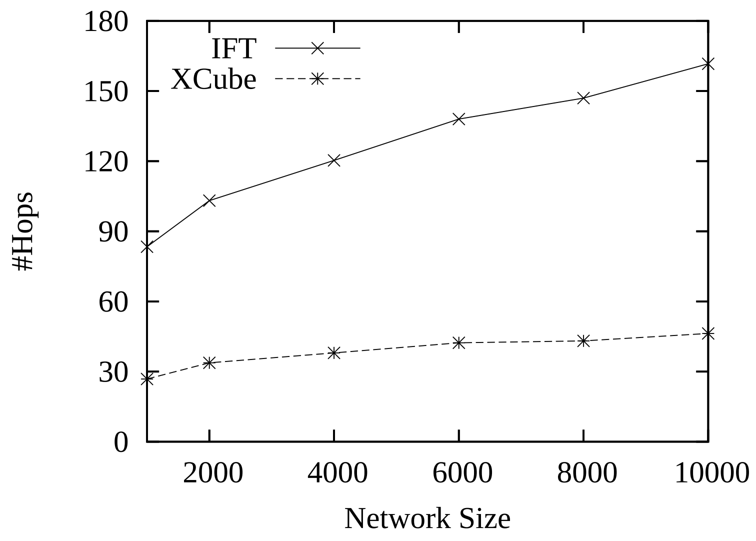


Figure 6.10: Comparison on various network sizes.

Therefore, we believe that employing 40% of peers as virtual peers are very feasible. When more capable peers are in charge of additional subcubes as virtual peers, the load is more balanced. This is because the nodes corresponding to many synopses are assigned to more peers.

Varying Network Size

In another experiment, we compare XCube and IFT by varying the network sizes. The results are shown in Figure 6.10. In XCube, the major cost is consumed by the anchor peer to forward the query to the indexing peers that store relevant synopses. Once a query is routed to its anchor peer (in $\log N$ hops), the distances between the anchor peer and the indexing peers are not affected by the network size significantly. The routing cost is closely related to the distance between the query's bit vector and the relevant synopses' bit vectors. On the other hand, in IFT, the total cost is the sum of costs of all individual point (tag) queries, and the cost of each point query (in terms of the number of hops) is closely related to the

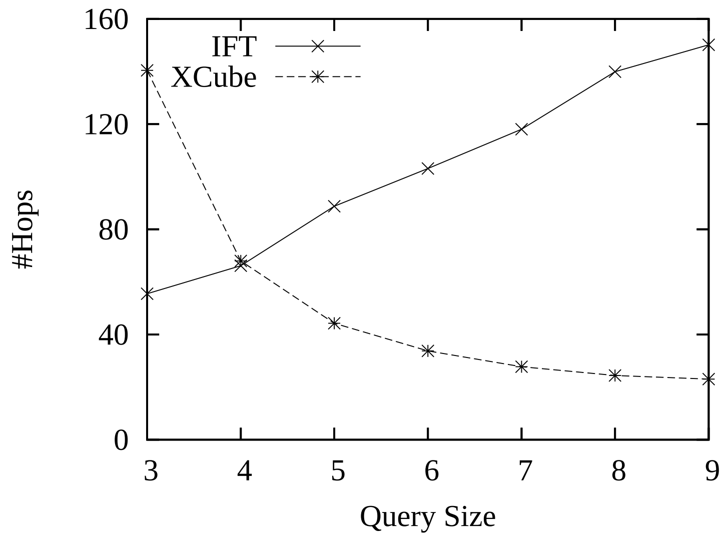


Figure 6.11: Comparison on various query sizes.

network size, N. Hence, XCube is less sensitive to the network size than IFT.

Varying Query Size

Next, we present the performance of the two schemes by varying the query size (the number of tags in a query). The results are shown in Figure 6.11. XCube outperforms IFT when the query size is larger than 4. The two schemes give two contrasting trends: XCube performs better when the query size is larger; while IFT is superior for shorter queries. The reasons are straightforward. In XCube, a query of large size means more 1-bits in the query bit vector, so the anchor peer of the query is nearer to the anchor peers of the related documents. In IFT, a shorter query means fewer peers to visit. As XML queries tend to be more specific and contain more tag information (as can be seen in the benchmark queries in traditional benchmark datasets), this result shows that XCube is a promising and more effective scheme over the IFT scheme.

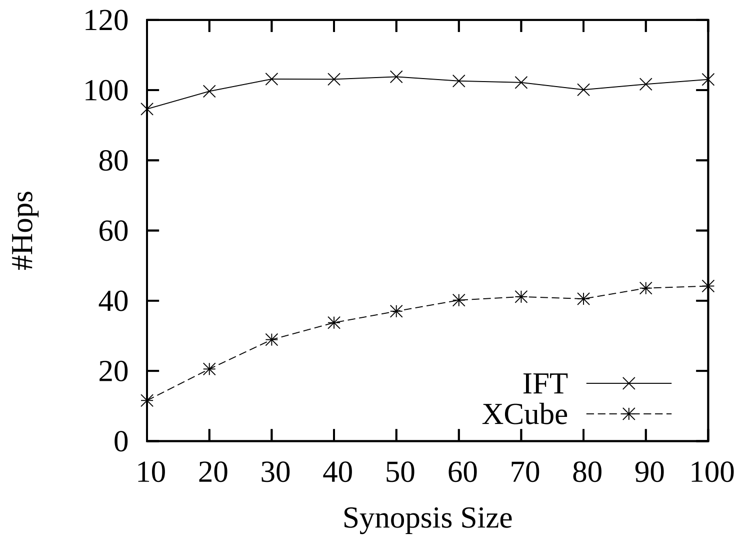


Figure 6.12: Comparison on various synopsis sizes.

Varying Synopsis Size

In Figure 6.12, we show the results for the performance of XCube and IFT on various synopsis sizes. The x-axis is the synopsis size, the y-axis is the number of hops to route a query on average. The cost of IFT does not change much when the synopsis size increases. This shows that the performance of IFT is not affected by synopsis sizes, but mainly by the query size and network size. XCube incurs more hops when the synopsis size increases because the distance between the anchor peer of the query and the anchor peers of related documents are larger. In practice, the synopsis size is usually much smaller than 100.

Efficiency comparison

We compare the efficiency of XCube and IFT in Figure 6.13. Here, we examine the time (in terms of the number of hops) that answers are progressively returned. As shown in the figure, XCube incrementally returns answers to a query as they are produced, while IFT can only start to generate output when the metadata of the

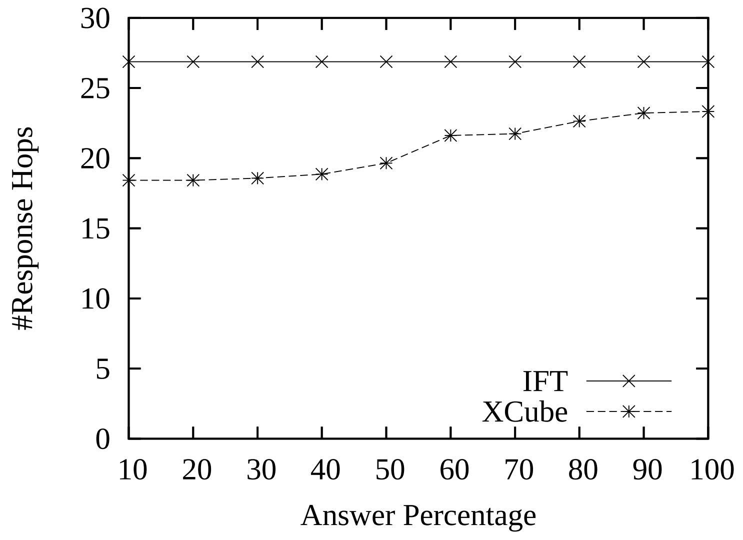


Figure 6.13: Efficiency comparison.

last tag is returned. From the figure, we can see that it takes around 18 hops for XCube to return the first answer and around 23 hops to return the last answer; on the other hand, IFT returns answers after the metadata of all tags are retrieved at about 27 hops. Clearly, XCube is more efficient than IFT, because XCube starts to find answers once the query is routed to its anchor indexing peer.

Effectiveness of bit maps

Finally, we evaluate the effectiveness of bit maps on pruning away irrelevant documents in Figure 6.14. The size of a bit map is determined by two factors: the dimension of the bit map and the number of partitions in each dimension. BitMap-p indicates that each dimension is divided into p partitions. The x-axis is the bit map dimension; and the y-axis is percentage of positive answers. We can see clearly from the figure that the accuracy of a bit map increases when the dimension or the number of partitions increases. In the best case where the bit map represents data of 6 dimensions and each dimension is partitioned into 10 segments, the size of the

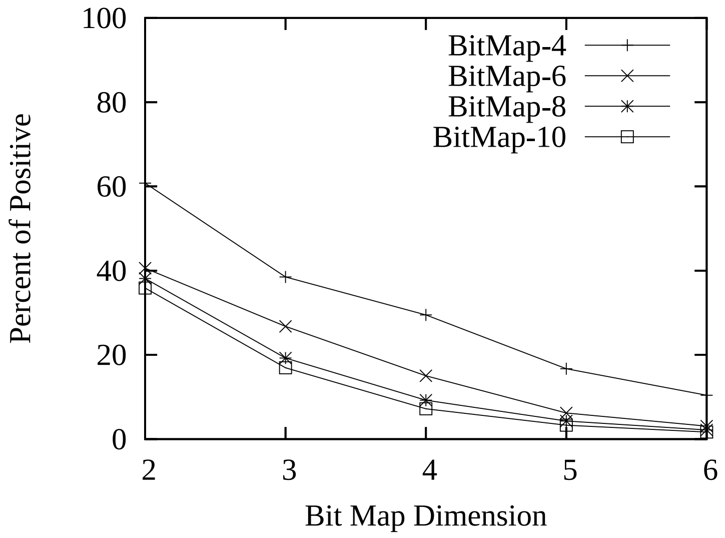


Figure 6.14: Effectiveness of bit maps.

bit map is $10^6/8 \approx$122K bytes. For a shared XML document, the owner peer only needs to index its bit map in one anchor peer (a transmission of 122K bytes data can even be finished in a few seconds for a dial-up peer). We believe that such bit maps are commonly acceptable in terms of both storage cost and bandwidth cost.

In summary, XCube offers more balanced load, requires fewer messages to complete routing a query and fewer messages to obtain the first and last answers, and is clearly superior over IFT.

6.6 Summary

In this chapter, we have presented the design and evaluation of XCube, a system to process XPath queries over a P2P network. In XCube, an XML document is indexed based on its structure summary and content summary in a hypercube overlay network. The two types of summaries and the complete metadata are indexed in the anchor peer of the document. Then the structure summary alone is further indexed in the $SubCube^{-}$. The IP address of the owner peer is excluded in the $SubCube^{-}$. An XPath query is processed in 4 phases. In phase 1, the bit vector of the query is derived to locate the anchor peer of the query. In phase 2, the query is evaluated on the structural constraints and then forwarded to the anchor peers of the related documents. In phase 3, the predicates are examined based on the content summary in every anchor peer of involved document. In phase 4, the query is forwarded to the related owner peer according to its IP address stored in the document's anchor peer. The owner peers then evaluate the query against the related XML documents and return the answer to the querying peer.

XCube offers the following advantages over traditional methods. First, the load in XCube is more balanced, which is extremely important for a P2P network. Balanced load ensures that peers share load fairly and the network is relatively stable. Load balancing is accomplished automatically with few user efforts. Second, users do not need to know the precise information about remote XML schemas. They can issue queries over XML elements or structures according to their demands including queries involving ancestor-descendant relationships and wildcards. Third, answers to a query are returned incrementally. This is also an important feature for P2P applications as users can refine the query or issue a new query earlier in case of poor quality queries. Our comprehensive experimental study shows that XCube is adaptive to varying query sizes and scalable to large P2P networks and outperforms several other methods (NAIVE-XCube, PC-XCube and IFT).

Chapter 7

Conclusion

In this chapter, we summarize the contributions of the thesis and discuss some future works.

7.1 Summary of Contributions

This thesis focuses on keyword-based search in P2P networks. We propose SPRITE to build partial distributed index for text data first. Then, CYBER is proposed to exploit relevance feedback in a community basis. Last, keyword-based search on structure information of XML data is supported in XCube.

Many documents are shared in P2P networks. Building a complete index on every document in a DHT network is impractical, because the construction and maintenance of such index is extremely expensive. SPRITE builds a partial index on a small number of representative terms. Progressively, the index is refined by learning from query history, so that the documents can still be found even if the user interests change. We conducted a comprehensive simulation study to show that SPRITE performs nearly as good as a centralized system with complete index in terms of precision and recall. SPRITE also outperforms a static partial indexing scheme by a wide margin.

Thereafter, we propose CYBER to enhance the search quality by involving community-based relevance feedback. Different from the traditional relevance feedback techniques, CYBER frees users from selecting a set of relevant answers and waiting for the re-evaluation. Every group of users with a similar interest construct a community. Users of the same community are discovered by matching user profiles and document profiles when routing queries in a DHT network. Given a query, CYBER leverages on the community based feedback to refine the queries on-the-fly. The user profiles and document profiles are updated by the system automatically so that query patterns are always reflected in the profiles. Our extensive experimental study showed that CYBER outperforms the traditional single-user relevance feedback technique, because user feedbacks are accumulated in a community.

Besides processing simple queries on pure text data, we also investigate keyword-based queries on data of richer format, XML data. In XCube, the structure summary and content summary are indexed for a shared XML document in various peers. Instead of indexing every individual tag name, XCube indexes the synopsis of an XML document as one entry. Indexing content summaries can prune away a large portion of documents that fulfill the structural constraints but not predicate constraints. XPath queries are routed to indexing peers responsible for related synopses. XCube offers the following advantages over traditional methods. First, the load in XCube is balanced as popular tag names are distributed to various synopses containing them. Balanced load ensures that peers share load fairly and the network is relatively stable. Second, users do not need to know the precise information about remote XML schemas. They can issue queries over XML elements or structures according to their demands including queries involving ancestor-descendant relationships and wildcards. Third, answers to a query are returned incrementally.

This is also an important feature for P2P applications as users can refine the query or issue a new query earlier in case of poor quality queries. Our comprehensive experimental study shows that XCube is adaptive to varying query sizes and scalable to large P2P networks and outperforms several other methods (NAIVE-XCube, PC-XCube and IFT).

7.2 Future Work

In this section, we suggest the following major possible research directions as future work.

7.2.1 Searching pure text data

Term positions

We have seen that SPRITE successfully reduces index size and CYBER improves search quality with community-based feedback. There are still some aspects, in which search quality can be further improved. Techniques in the literature have been focusing on simple formulas to calculate term weights. The relationships among query terms are not considered. Their positions in documents are not fully utilized. Intuitively, terms appearing in the same sentence is more important than terms appearing in the same paragraph/document. Such information has been used to calculate term weights in centralized systems. However, keeping such information in a P2P network can easily increase the size of an inverted list dramatically. Methods to optimize building such complex index, such as combining some terms, in a distributed environment are desired in future research.

Searching queries

Answer-based applications become popular recently, such as "Yahoo! Answers"[1] and "Baidu Zhidao"[2]. Such applications empowers users to participate a huge community more deeply. Indexing queries in a P2P network is straightforward, but maintaining the answers (documents) is non-trivial. Replication algorithms should be introduced to incorporate searching and replicating large chunk of text data. Moreover, the answers should be searchable to the users as well. The answers should be ranked first, which involves user interactions. The top answers can then be labeled by some queries, summarized and indexed. New techniques are required to accomplish these tasks.

7.2.2 Searching richer text data

XML is commonly accepted as data exchange format for its text nature. In XCube, most irrelevant documents are pruned away by checking structural constraints and predicates when routing a query. However, pure keyword search on XML documents is useful because of its simplicity. A compact fragment of the relevant document, instead of the entire document, should be returned as a result. The main challenge is how to reduce index maintenance cost. Summarizing schemes are not applicable in this case because a large number of keywords can be queried in a data-centric document. A possible solution is to cluster similar XML documents/fragments first, and then build index on every cluster. The index can also be built in just-in-time manner to further reduce the index entries.

[1]answers.yahoo.com

[2]http://zhidao.baidu.com

7.2.3 Browsing

When Web users look for some useful information, the two major actions are searching and browsing. Keyword-based search usually leads a user to some relevant data sources. The user usually can find additional information by browsing from the sources. In a P2P network, browsing is still weakly supported either within a peer or across peers. How can we browse related data stored in various peers from a peer that is discovered by a normal keyword query? The key challenge is how to link related data/peers effectively. More specifically, there are two problems: (i) grouping similar documents in a P2P network and (ii) identify some important phrases as anchor text, like the hyperlinks in Web. New algorithms are required to accomplish these tasks.

Bibliography

[1] K. Aberer. P-Grid: A self-organizing access structure for P2P information systems. In *Proceedings of the 6th CoopIS Conference*, pages 179–194, 2001.

[2] S. Abiteboul, I. Manolescu, and N. Preda. Constructing and querying a peer-to-peer warehouse of XML resources. In *Semantic Web and Databases Workshop*, pages 219–225, 2004.

[3] I. Aekaterinidis and P. Triantafillou. Pastrystrings: A comprehensive content-based publish/subscribe dht network. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, page 23, 2006.

[4] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE '02: Proceedings of the 18th International Conference on Data Engineering*, page 5, Washington, DC, USA, 2002. IEEE Computer Society.

[5] S. Amer-Yahia, N. Koudas, A. Marian, D. Srivastava, and D. Toman. Structure and content scoring for xml. In *Proceedings of the 31st VLDB Conference*, pages 361–372, Trondheim, Norway, 2005.

[6] E. Anceaume, M. Gradinariu, A. K. Datta, G. Simon, and A. Virgillito. A semantic overlay for self- peer-to-peer publish/subscribe. In *Proceedings of*

the 26th IEEE International Conference on Distributed Computing Systems, page 22, 2006.

[7] A. Andrzejak and Z. Xu. Scalable, efficient range queries for grid information services. In *Proceedings of the 2nd IEEE International Conference on Peer-to-Peer Computing*, 2002.

[8] J. Aspnes, J. Kirsch, and A. Krishnamurthy. Load balancing and locality in range-queriable data structures. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 115–124, 2004.

[9] J. Aspnes and G. Shah. Skip graphs. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2003.

[10] R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval.* ACM Press / Addison-Wesley, 1999.

[11] G. Beydoun, R. Kultchitsky, and G. Manasseh. Evolving semantic web with social navigation. *Expert Syst. Appl.*, 32(2):265–276, 2007.

[12] A. Bonifati, U. Matrangolo, A. Cuzzocrea, and M. Jain. XPath lookup queries in p2p networks. In *WIDM'04: Proceedings of the 6th annual ACM international workshop on Web information and data management*, pages 48–55, New York, NY, USA, 2004. ACM Press.

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

[14] J. Callan. Distributed information retrieval. In *Advances in information retrieval*, pages 127–150, 2000.

[15] J. Callan and M. Connell. Query-based sampling of text databases. *ACM Transactions on Information Systems*, 19(2):97–130, 2001.

[16] D. Carmel, D. Cohen, R. Fagin, E. Farchi, M. Herscovici, Y. S. Maarek, and A. Soffer. Static index pruning for information retrieval systems. In *Proceedings of the 24th annual International ACM SIGIR Conference*, pages 43–50, 2001.

[17] R. Chand and P. Felber. Semantic peer-to-peer overlays for publish/subscribe networks. In *Euro-Par*, pages 1194–1204, 2005.

[18] H. Chen, H. Jin, J. Wang, L. Chen, Y. Liu, and L. M. Ni. Efficient multi-keyword search over p2p web. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 989–998, 2008.

[19] L. Chen, B. Cui, H. Lu, L. Xu, and Q. Xu. iSky: Efficient and progressive skyline computing in a structured p2p network. *Proceedings of the 28th International Conference on Distributed Computing Systems*, pages 160–167, 2008.

[20] P.-A. Chirita, C. S. Firan, and W. Nejdl. Summarizing local context to personalize global web search. In *CIKM '06: Proceedings of the 15th ACM international conference on Information and knowledge management*, pages 287–296, 2006.

[21] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *Proceedings of the 22nd ICDCS Conference*, July, 2002.

[22] B. Cui, H. Lu, Q. Xu, L. Chen, Y. Dai, and Y. Zhou. Parallel distributed processing of constrained skyline queries by filtering. In *ICDE*, 2008.

[23] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common api for structured peer-to-peer overlays. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 33–44, 2003.

[24] Extensible Markup Language (XML). www.w3.org/xml/.

[25] L. Galanis, Y. Wang, S. Jeffery, and D. DeWitt. Locating data sources in large distributed systems. In *Proceedings of VLDB'03*, pages 874–885, Berlin, Germany, 2003.

[26] L. Galanis, Y. Wang, S. R. Jeffery, and D. J. Dewitt. Processing queries in a large peer-to-peer system. In *Proceedings of the 16th CAiSE Conference*, pages 273–288, 2003.

[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 VLDB'04*, pages 444–455, 2004.

[28] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *Proceedings of VLDB'04*, pages 444–455, 2004.

[29] O. D. Gnawali. A keyword-set search system for peer-to-peer networks.pdf. In *Master thesis. Massachusetts Institute of Technology*, 2002.

[30] Gnutella Development Home Page. http://gnutella.wego.com/.

[31] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of VLDB'97*, pages 436–445, 1997.

[32] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: ranked keyword search over xml documents. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 16–27, 2003.

[33] W. Hersh, C. Buckley, T. Leone, and D. Hickam. Ohsumed: An interactive retrieval evaluation and new large test collection for research. In *SIGIR*, 1994.

[34] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient ir-style keyword search over relational databases. In *Proceedings of the 29th international conference on Very large data bases*, pages 850–861. VLDB Endowment, 2003.

[35] V. Hristidis and Y. Papakonstantinou. Discover: keyword search in relational databases. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 670–681. VLDB Endowment, 2002.

[36] V. Hristidis and Y. Papakonstantinou. Keyword proximity search in xml trees. *IEEE Trans. on Knowl. and Data Eng.*, 18(4):525–539, 2006.

[37] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. Baton: a balanced tree structure for peer-to-peer networks. In *VLDB'05: Proceedings of the 31st international conference on Very large data bases*, pages 661–672, 2005.

[38] Y. J. Joung, C. T. Fang, and L. W. Yang. Keyword search in DHT-based peer-to-peer networks. In *Proceedings of ICDCS'05*, pages 339–348, 2005.

[39] D. J.Watts, P. S. Dodds, and M. J. Newman. Identity and search in social networks. *Science*, 296, 2002.

[40] G. Karypis and E.-H. Han. Concept indexing: A fast dimensionality reduction algorithm with applications to document retrieval and categorization. Technical report tr-00-0016, University of Minnesota, 2000.

[41] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *Proceedings of ACM SIGMOD'02*, pages 133–144, 2002.

[42] G. Koloniari and E. Pitoura. Content-based routing of path queries in peer-to-peer systems. In *Proceedings of the EDBT Conference*, 2004.

[43] D. L. Lee, H. Chuang, and K. Seamons. Document ranking and the vector-space model. *IEEE Software*, 14(2):67–76, 1997.

[44] U. Lee, Z. Liu, and J. Cho. Automatic identification of user goals in web search. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 391–400, 2005.

[45] H. Li, Q. Tan, and W.-C. Lee. Efficient progressive processing of skyline queries in peer-to-peer systems. In *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems*, page 26, 2006.

[46] J. Li, B. T. Loo, J. Hellerstein, F. Kaashoek, D. R. 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)*, 2003.

[47] X. Li, Y. J. Kim, R. Govindan, and W. Hong. Multi-dimensional range queries in sensor networks. In *Proceedings of Communications of the ACM*, 2003.

[48] Y. Li, M. T. Özsu, and K.-L. Tan. XCube: Processing XPath queries in a hypercube overlay network. *Peer-to-Peer Networking and Applications*, 2008.

[49] Y. G. Li, H. V. Jagadish, and K.-L. Tan. Sprite: A learning-based text retrieval system in dht networks. In *ICDE*, pages 1106 – 1115, 2007.

[50] Y. G. Li, L. D. Shou, and K.-L. Tan. Cyber: Community-based search engine. In *proceedings of the 8th International Conference on Peer-to-Peer Computing (P2P)*, Aachen, Germany, September 8-11, 2008.

[51] C. Y. Liau, S. Bressan, and A. N. Hidayanto. Adaptive peer-to-peer routing with proximity. In *Proceedings of The 14th International Conference on Database and Expert Systems Applications (DEXA)*, 2003.

[52] A. Löser, S. Staab, and C. Tempich. Semantic social overlay networks. *IEEE JSAC*, 25(1):5–14, 1 2007.

[53] J. Lu and J. Callan. Content-based retrieval in hybrid peer-to-peer networks. In *Proceedings of the 12th International Conference on Information and Knowledge Management*, pages 199–206. ACM Press, 2003.

[54] Lucene Home Page. http://lucene.apache.org/.

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

[56] G. Manku, M. Bawa, and P. Raghavan. Symphony: Distributed hashing in a small world. In *Proceedings of USITS*, 2003.

[57] A. Mislove, K. P. Gummadi, and P. Druschel. Exploiting social networks for internet search. In *HotNets*, 2006.

[58] P. Ogilvie and J. Callan. The effectiveness of query expansion for distributed information retrieval. In *Proceedings of the 10th International Conference on Information and Knowledge Management*, pages 183–190, 2001.

[59] O. Papapetrou, S. Michel, M. Bender, and G. Weikum. On the usage of global document occurrences in peer-to-peer information systems. In *CoopIS/DOA/ODBASE (1)*, 2005.

[60] I. Podnar, M. Rajman, T. Luu, F. Klemm, and K. Aberer. Scalable peer-to-peer web retrieval with highly discriminative keys. In *ICDE*, pages 1096–1105, 2007.

[61] N. Polyzotis and M. Garofalakis. XSKETCH synopses for XML data graphs. *ACM Trans. Database Syst.*, 31(3):1014–1063, 2006.

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

[63] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, 2001.

[64] P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In *Proceedings of the International Middleware Conference*, June, 2003.

[65] M. Roussopoulos and M. Baker. Practical load balancing for content requests in peer-to-peer networks. *Distributed Computing*, 18(6):421–434, June 2006.

[66] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001.

[67] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. ofMultimedia Computing and Networking*, 2002.

[68] C. Sartiani, P. Manghi, G. Ghelli, and G. Conforti. XPeer: A self-organizing XML P2P database system. In *Proceedings of the First EDBT Workshop on P2P and Databases*, 2004.

[69] M. Schlosser, M. Sintek, S. Decker, and W. Nejdl. HyperCuP - hypercubes, ontologies and efficient search on P2P networks. In *Workshop on Agents and P2P Computing*, pages 112–124, 2002.

[70] M. Schlosser, M. Sintek, S. Decker, and W. Nejdl. Shaping up peer-to-peer networks. Technical report, Stanford University, 2002.

[71] C. Schmidt and M. Parashar. Flexible information discovery in decentralized distributed systems. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, 2003.

[72] U. Shardanand and P. Maes. Social information filtering: Algorithms for automating "word of mouth". In *CHI*, 1995.

[73] X. Shen, B. Tan, and C. Zhai. Implicit user modeling for personalized search. In *CIKM*, pages 824–831, 2005.

[74] Z. Shen and S. Tirthapura. Approximate covering detection among content-based subscriptions using space filling curves. In *Proceedings of the 27th International Conference on Distributed Computing Systems*, page 2, 2007.

[75] Y. Shu, B. C. Ooi, K.-L. Tan, and A. Zhou. Supporting multi-dimensional range queries in peer-to-peer systems. In *Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing*, pages 173–180, Washington, DC, USA, 2005.

[76] L. Si and J. Callan. The effect of database size distribution on resource selection algorithms, 2003.

[77] L. Si and J. Callan. Relevant document distribution estimation method for resource selection, 2003.

[78] C. Silverstein, M. Henzinger, H. Marais, and M. Moricz. Analysis of a very large alta vista query log. In *Digital System Research Center, Technical Report 1998-014*, Oct, 1998.

[79] G. Skobeltsyn, M. Hauswirth, and K. Aberer. Efficient processing of XPath queries with structured overlay networks. In *OTM Conferences*, pages 1243–1260, 2005.

[80] G. Skobeltsyn, T. Luu, K. Aberer, M. Rajman, and I. P. Zarko. Query-driven indexing for peer-to-peer text retrieval. In *WWW*, pages 1185–1186, 2007.

[81] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, 2001.

[82] S. Surana, B. Godfrey, K. Lakshminarayanan, R. Karp, and I. Stoica. Load balancing in dynamic structured peer-to-peer systems. *Perform. Eval.*, 63(3):217–240, 2006.

[83] C. Tang and S. Dwarkadas. Hybrid global-local indexing for efficient peer-to-peer information retrieval. In *NSDI*, 2004.

[84] C. Tang, S. Dwarkadas, and Z. Xu. On scaling latent semantic indexing for large peer-to-peer systems. In *Proceedings of the 27th annual International ACM SIGIR Conference*, Sheffield, UK, 2004.

[85] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *SIGCOMM*, 2003.

[86] Q. Wang and M. T. Özsu. A data locating mechanism for distributed XML data over P2P networks. In *Technical report CS-2004-45, University of Waterloo*, 2004.

[87] S. Wang, B. C. Ooi, A. K. H. Tung, and L. Xu. Efficient skyline query processing on peer-to-peer networks. In *ICDE*, pages 1126–1135, 2007.

[88] X. Wang and C. Zhai. Learn from web search logs to organize search results. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 87–94, 2007.

[89] S. Wu, J. Li, B. C. Ooi, and K.-L. Tan. Just-in-time query retrieval over partially indexed data on structured p2p overlays. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 279–290, New York, NY, USA, 2008. ACM.

[90] Y. Xie and D. O'Hallaron. Locality in search engine queries and its implications for caching. In *Proceedings of IEEE Infocom 2002*, July, 2002.

[91] XML Path Language (XPath). www.w3.org/tr/xpath/.

[92] XQuery 1.0: An XML Query Language. www.w3.org/tr/xquery/.

[93] Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest lcas in xml databases. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 527–538, 2005.

[94] H. Yamamoto, D. Maruta, and Y. Oie. Replication methods for load balancing on distributed storages in p2p networks. In *SAINT*, pages 264–271, 2005.

[95] B. Yang and H. Garcia-Molina. Improving search in peer-to-peer networks. In *Proceedings of the 22nd ICDCS Conference*, 2002.

[96] B. Yang and H. Garcia-Molina. Designing a super-peer network. In *Proceedings of the 18th International Conference on Data Engineering*, 2003.

[97] B. B. Yao, M. T. Özsu, and N. Khandelwal. XBench benchmark and performance testing of XML DBMSs. In *Proceedings of ICDE'04*, page 621, 2004.

[98] B. Yu, G. Li, K. Sollins, and A. K. H. Tung. Effective keyword-based selection of relational databases. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 139–150, 2007.

[99] H.-J. Zeng, Q.-C. He, Z. Chen, W.-Y. Ma, and J. Ma. Learning to cluster web search results. In *SIGIR '04: Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 210–217, 2004.

[100] N. Zhang, M. T. Özsu, A. Aboulnaga, and I. F. Ilyas. XSEED: Accurate and fast cardinality estimation for XPath queries. In *Proceedings of ICDE'06*, page 61, 2006.