

LSDS-IR



1st Workshop on Large-Scale Distributed
Systems for Information Retrieval

July 27, 2007

In conjunction with ACM SIGIR 2007, Amsterdam, The Netherlands

Organizers: Flavio Junqueira - *Yahoo! Research Barcelona*

Vassilis Plachouras - *Yahoo! Research Barcelona*

Fabrizio Silvestri - *ISTI-CNR*

Ivana Podnar Zarko - *FER University of Zagreb*

Preface

We, the organizers of the LSDS-IR workshop, are very enthusiastic with the program we have been able to put together. We have received a number of very good papers, that we are sure are going to promote very interesting and fruitful discussions during the workshop. Before moving to a discussion on the actual program, we would like to say a word on the reasons for organizing it.

It is not a new observation that the Web is growing. Researchers across the world have spent so far a significant amount of effort to invent techniques to improve the experience of users when they navigate on the Web. In particular, we refer to information retrieval and related areas targeting Web search. An important issue, however, is that work in these areas has mostly concentrated on algorithms and application-level strategies, and infrastructure is unquestionably an important component as well. At a very different front, researchers in computer systems have been paying attention to instances that go beyond a handful of devices, as the challenges are significantly different in such scenarios. As systems grow in size and distribution, it is often enough not trivial to enable solutions that are easily expandable and that perform well. We thus believe that bringing these two areas together is of extreme importance to the development of the Web. The main purpose of this workshop is hence to gather participants that have some interest in this exact intersection to discuss the future of systems for information retrieval.

The workshop comprises an invited talk by Ophir Frieder, two sessions, one panel, and a poster presentation that runs in parallel. The invited talk, “On Peer-to-Peer Search Applications”, discusses search accuracy in P2P file-sharing applications. The following session, “Elephants in the living room”, means to have papers that discuss new ideas, which potentially will change our view or give us new directions for large-scale distributed search. The second session, “Arrows in the quiver”, includes presentations of new mechanisms that may prove to be useful in the development of distributed search engines. We wrap up with a panel of distinguished researchers in the area, which will interactively discuss the new ideas, directions, and results. Finally, during the whole workshop, authors will exhibit posters. The main goal is to maximize interaction among the participants of the workshop.

We would like to thank the organization of SIGIR for providing all the resources necessary to enable this event; the members of our program committee for doing a great job in such a short period of time; Ophir Frieder for agreeing on giving an invited talk.

July 2006

Flavio Junqueira
Vassilis Plachouras
Fabrizio Silvestri
Ivana Podnar Zarko

LSDS-IR 2007 Organization

Workshop Organizers

Flavio Junqueira (Yahoo! Research Barcelona, Spain)
Vassilis Plachouras (Yahoo! Research Barcelona, Spain)
Fabrizio Silvestri (ISTI – CNR, Italy)
Ivana Podnar Zarko (FER University of Zagreb, Croatia)

Program Committee

Karl Aberer (Ecole Polytechnique Fédérale de Lausanne, Switzerland)
Ricardo Baeza-Yates (Yahoo! Research, Spain and Chile)
Fidel Cacheda (University of A Coruña, Spain)
Abdur Chowdhury (Illinois Institute of Technology, USA)
Norbert Fuhr (University of Duisburg-Essen, Germany)
Ronny Lempel (IBM Research Haifa, Israel)
Jie Lu (Carnegie Mellon University, USA)
Massimo Marchiori (University of Padova and UTILABS, Italy)
Iadh Ounis (University of Glasgow, UK)
Diego Puppin (ISTI – CNR, Italy)
Thomas Risse (L3S/University of Hannover, Germany)
Luo Si (Purdue University, USA)
Umberto Straccia (ISTI – CNR, Italy)
Christos Tryfonopoulos (Max Planck Institute, Germany)
Michalis Vazirgiannis (Athens University of Economics and Business, Greece)
Gerhard Weikum (Max Planck Institute, Germany)
Pavel Zezula (Masaryk University, Czech Republic)
Justin Zobel (RMIT University, Australia)

Table of Contents

On Peer-to-Peer Search Applications.....	1
<i>Ophir Frieder</i>	
A View of the Data on P2P File-sharing Systems.....	3
<i>Wai Gen Yee, Linh Thai Nguyen, Ophir Frieder</i>	
An Epidemic-based P2P Recommender System.....	11
<i>Jie Yang, Jun Wang, Maarten Clements, Johan A. Pouwelse, Arjen P. de Vries, Marcel Reinders</i>	
Design Alternatives for Large-Scale Web Search: Alexander was Great, Aeneas a Pioneer, and Anakin has the Force.....	16
<i>Matthias Bender, Sebastian Michel, Peter Triantafillou, Gerhard Weikum</i>	
Taming Hot-Spots in DHT Inverted Indexes.....	23
<i>Nuno Lopes, Carlos Baquero</i>	
Node Behavior Prediction for Large-Scale Approximate Information Filtering.....	30
<i>Christian Zimmer, Christos Tryfonopoulos, Klaus Berberich, Gerhard Weikum, Manolis Koubarakis</i>	
Evaluation of neighbourhood selection methods in decentralized recommendation systems.....	38
<i>Maarten Clements, Arjen P. de Vries, Johan A. Pouwelse, Jun Wang, Marcel J.T. Reinders</i>	

On Peer-to-Peer Search Applications

Ophir Frieder

Department of Computer Science
Georgetown University
Washington, DC 20057

(Invited Keynote Abstract)

ABSTRACT

Peer-to-Peer (P2P) applications are dominating the Internet. In a recent P2P Media Summit, a presentation from Cache Logic Research [1] stated that more than half of all Internet traffic is driven by P2P applications. Furthermore, by percentage of Internet traffic, only P2P generated traffic is on the increase. The World-Wide-Web generated traffic is on a steep decline, now responsible for roughly one-third of the traffic volume, and FTP (file transfer protocol) and e-mail generated traffic have leveled off, each at roughly 5% of the volume of traffic. P2P application popularity and dominance of bandwidth utilization necessitates research aimed at improving P2P functionality and efficiency. We focus on the search component of P2P file sharing applications.

Research aimed at improving search accuracy in P2P file sharing applications is limited as compared to related efforts for the World Wide Web, or even more so, for centralized conventional search engines. Furthermore, the unique characteristics of P2P file sharing applications further complicate the search process, resulting in relatively poor search accuracy. Several issues complicate the search process including:

- **Binary Files:** In conventional search applications, files are self-descriptive. That is, the content of the file, namely the text within, represents the file in the search process. In P2P file sharing applications, the file content is often a sound wave or image, and the file's associated descriptor is used to represent the file. File descriptors are typically short and poorly specified.
- **Lack of a Centralized Index:** Conventional search systems rely on a centralized index. With such an index, document identification and relevance ranking is established. In P2P applications, queries are broadcasted globally, potentially to only a subset of nodes, and peers with relevant documents respond. A global index does not exist. Thus, global information is

unavailable, complicating the identification and relevance ranking of documents.

- **Lack of Term Weights:** Lacking global information complicates the weighting of terms. Conventional search systems associate weights to terms according to term uniqueness. Retrieval is based on a combined weighting of terms present in the document. The lack of term weights degrades document relevance ranking. Furthermore, the dynamic nature of P2P file sharing applications limits the value of using historical, static term weights in the ranking process.
- **Conjunctive Queries:** In the majority of search applications, not all query terms must be present for a document to be retrieved. In the P2P file sharing environment, all query terms must appear in the descriptor for the associated file to be deemed relevant. Since descriptors are short and contain only relatively few words, poorly defined descriptors result in poor retrieval accuracy.

We address schemes to improve search accuracy in P2P file sharing environments. Initially, we analyze a peer-to-peer file sharing query log obtained by our query analysis tool [2]. Even our preliminary study [3] clearly demonstrates the differences that exist between Web and P2P applications in terms of query and content structures.

We continue by describing comprehensive solutions for efficiently improving search accuracy in peer-to-peer file-sharing systems. Our solutions leverage the traditional information retrieval focus of our IIT Information Retrieval Laboratory. Initially, we highlight several of our metadata management solutions [4]. By varying the descriptor replication scheme of the files selected for download, we improve the overall search accuracy by roughly 10%.

Our descriptor enrichment solutions [5, 6] are modeled after traditional information retrieval techniques but capitalize on the availability of descriptor hash keys to

improve accuracy. Descriptor enhancement techniques account for roughly a 20% improvement in search accuracy.

All of the above techniques individually improve accuracy and can be used conjunctively to further improve the search quality by approximately 25%. However, each technique also increases network traffic. Thus, we investigated an approach that uses various sampling techniques to reduce network traffic but still increase accuracy. Overall, integrating these sampling techniques with our accuracy enhancing approaches, we managed to improve search accuracy by nearly 20% while actually reducing network traffic [7].

We conclude the talk by briefly focusing on applying P2P applications to other environments, for example, to hand-held devices. We illustrate various scenarios that can benefit from such technology, but caution the developer on some potential pitfalls.

REFERENCES

- [1] A. Parker, *CacheLogic Research presentation at the First Annual P2P Media Summit LA*, October 2006. dcia.info/P2PMSLA/CacheLogic.ppt.
- [2] L. Nguyen, W. G. Yee, D. Jia, and O. Frieder, "A Tool for Information Retrieval Research in Peer-to-Peer File Sharing Systems," *IEEE Twenty-Third International Conference on Data Engineering (ICDE)*, Istanbul, Turkey, April 2007.
- [3] L. Nguyen, D. Jia, W. G. Yee, and O. Frieder, "Analysis of Query Logs in Gnutella Peer-to-Peer Network," *ACM Thirtieth Conference on Research and Development in Information Retrieval (SIGIR)*, Amsterdam, Netherlands, July 2007.
- [4] W. G. Yee and O. Frieder, "On Search in Peer-to-Peer File Sharing Systems," *ACM Twentieth Symposium on Applied Computing (SAC)*, Santa Fe, New Mexico, March 2005.
- [5] W-G Yee, D. Jia, and O. Frieder, "Finding Rare Data Objects in P2P File-Sharing Systems," *IEEE Fifth International Conference on Peer-to-Peer Computing (P2P)*, Konstanz, Germany, August 2005.
- [6] W. G. Yee, L. Nguyen, and O. Frieder, "Masked Queries for Search Accuracy in Peer-to-Peer File-sharing Systems," *IEEE/ACM Twenty-First International Parallel & Distributed Processing Symposium (IPDPS)*, Long Beach, California, March 2007.
- [7] D. Jia, W. G. Yee, L. Nguyen, and O. Frieder, "Distributed, Automatic File Descriptor Tuning in Peer-to-Peer File-Sharing Systems," *IEEE Seventh International Conference on Peer-to-Peer Computing (P2P)*, Galway, Ireland, September 2007.

BIOGRAPHY

Dr. Ophir Frieder is the Royden B. Davis Chair in Interdisciplinary Studies at Georgetown University. Since 1998, he has been the IITRI Chair Professor of Computer Science and the Director of the Information Retrieval Laboratory at the Illinois Institute of Technology, from which, he is currently on leave. He frequently consults for industry and government and for key intellectual property litigation. His research interests focus on scalable information retrieval systems spanning search and retrieval and communications issues. His systems are deployed in actual commercial and governmental production environments worldwide. He is a Fellow of the AAAS, ACM, and IEEE.

A View of the Data on P2P File-sharing Systems

Wai Gen Yee, Linh Thai Nguyen, Ophir Frieder
Information Retrieval Lab
Illinois Institute of Technology
10 W 31st Street
Chicago, IL 60616, USA
{waigen, linhnt, ophir}@ir.iit.edu

ABSTRACT

Peer-to-peer file-sharing is by some measures one of the leading Internet applications; millions of users are searching for and transferring millions of files daily. Due to its scale, it is important to fully understand how these systems work. We analyze the data shared on these systems, draw some conclusions on the nature of the application, and propose some research problems.

Categories and Subject Descriptors

H.3.5 [Information Storage and Retrieval]: Online Information Services – *Web-based services*.

General Terms

Measurement, Experimentation.

Keywords

Data crawler, Information retrieval, Peer-to-peer.

1. INTRODUCTION

The popularity and unique nature of P2P file-sharing systems make them a worthy direction for information retrieval research. With millions of daily users [3], the scale of this application makes important highly accurate, yet efficient search techniques. However, in order to address issues of accuracy and efficiency, it is necessary to first understand the nature of these systems, from how they are engineered to how they are used.

In this paper, we characterize the queries being issued in these systems and the data being shared. Previous measurement studies of P2P file-sharing systems focused more on network-level characteristics, such as the number of users, node uptime, number of queries, and number of shared files [9][10][17][19]. To the best of our knowledge, ours is the first work that examines P2P file-sharing systems from the perspective of data characterization (i.e., how shared data are described and searched).

2. GENERAL SPECIFICATION OF P2P FILE-SHARING SYSTEMS

P2P file-sharing systems are distinguished by the fact that there is no centralized data source. All participating nodes function as data sources and query routers. Furthermore, nodes can dynam-

cally join or leave the system. The function of these systems is to share the files of a dynamic group of users, each instance of which

is described by user-generated descriptors (e.g., filenames), and system-generated attributes, such as file size and hash key.

A query consists of a set of user-defined terms. It is generally routed via flooding until its time-to-live expires. At each node, the terms in the query are compared with each file's descriptor. If a file's descriptor contains all of the query's terms, then the file's descriptor and hash key, as well as the node's id are returned to the client [2].

The client groups results by hash key, and then ranks each group. In general, groups are ranked by their size (number of unique results in the group), although other ranking techniques are possible [4]. The user searches through this list of results, downloads the desired one, and then becomes a server for this file.

Another distinguishing characteristic of P2P file-sharing systems is that peers are autonomous. This implies that they can join or leave the system at any time and have local control of their shared file repositories. One consequence of this autonomy is that it becomes difficult to maintain any reliable statistics on shared data (e.g., document frequencies), which is common in centralized search engines. From a network perspective, autonomy makes the maintenance of structured or semi-structured routing topologies a difficult task. This makes it difficult to apply ranking functions that rely on global statistics (e.g., Okapi [5]), or network topologies (e.g., PageRank [6]).

P2P file-sharing systems are somewhat comparable to meta-search engines [7][8]. In both systems, user queries are routed to a set of independent data sources. Result sets from each data source, which possibly overlap, are collected, organized, and ranked locally for the user. The characteristics of P2P file-sharing systems that distinguish them from meta-search engines, however, are their dynamic nature, their scale, and the nature of their data (which is also the topic of this paper). Techniques developed for meta-search engines, therefore, are not generally applicable to P2P file-sharing systems.

3. DATA COLLECTION

We collected two types of data: *query data* and *shared data*. The former refers to queries issued by users and the latter refers to the data that are being shared by users. Our goal is to collect a representative set of data to yield a true picture of how P2P file-sharing systems are used.

Copyright is held by author/owners.

ACM SIGIR Workshop on Large Scale Distributed Systems for Information Retrieval '07, Amsterdam, The Netherlands.

To collect a representative data set, we used a Gnutella network crawling tool [1], which mimics a peer that can satisfy all user queries. By design, queries are routed to all “nearby” peers that can potentially answer them [2]. Consequently, all nearby queries are routed to our peer. Combined with the fact that Gnutella topologies are randomly created [2], this query set should be close to representative. Of course, effects such as lost messages due to transmission delays between physically distant peers may localize our queries.

Our query log was collected during two month-long spans – one starting on September, 2006 and another on April, 2007 – and includes query terms, desired file type, and timestamp. Queries were preprocessed by ignoring case difference and replacing punctuations with white space. We use a list of 130 terms manually built by us as our list of stopwords. Terms belonging to this list are removed from user queries and file descriptors. This list contains the standard set of stop words used in Information Retrieval (e.g., the English article “the” and the French article “le”). Foreign language stop words must be included due to the relative lack of locality of users and data sources in P2P file-sharing networks. Our stop word list also includes those that are particular to the file-sharing domain, such as filename extensions (e.g., “mp3”, “m4a”).

As an additional preprocessing step, we removed all system generated queries, such as the query “WhatIsNewXOXO”, which is sent by a super-peer to leaf nodes to inquire about their newly shared contents. We also removed all queries in some Asian languages, such as Chinese, Korean and Japanese, since we are unable to analyze those languages.

Shared data were collected in the Fall of 2006 and include the filenames, file sizes, file types, and hash keys of peers identified by their IP addresses. IP addresses were collected in two ways. First, we issued queries that are expected to be answered by a large number of peers (“sampling queries”), and extract the IP addresses from these responses. Sampling queries are based on query log term distributions, which we assume are the same as those of filenames. Second, we extract IP addresses from control messages defined by the Gnutella protocol needed to maintain the P2P overlay [2]. The use of control messages to identify IP addresses has been used in the previous measurement studies cited above. The use of sampling queries is particularly relevant to our work as we are mostly interested in the peers that share data. We believe that the combination of these two methods yields a large enough number of peer IP addresses to be representative.

For each collected IP address, we issue a “browse host” command to retrieve the set of files shared at this IP address. Browse host is a function built into the Gnutella specification that allows a client to view the directory of a peer identified by an IP address [2]. Note that the ability to browse hosts is active by default in all of the Gnutella clients we know of. However, browse host may be blocked by some users or by some firewalls, which may skew our results. We assume that hosts who function to spread spam are particularly likely to block this function, resulting in an underreporting of spam. We somewhat circumvent the browse-host-blocking actions with the use of sampling queries. Of course, using sampling queries does not guarantee that we retrieve all files shared by a peer. However, with enough sampling queries, we should be able to yield a reasonable view of a peer’s shared data.

Our datasets are published on our Web site (<http://ir.iit.edu/~waigen/proj/pirs/irwire/>). As they are collected from a public network, we encourage their use for research purposes. As well, the tools used to collect and analyze these logs are available on our Web site.

4. EXPERIMENTAL FINDINGS

4.1 Queries

We report an analysis of our query logs in this section. Note that many of our analyses can be found in a companion work [14], so to avoid redundancy, our analysis is of a more temporal nature – we examine how queries have changed over time.

Users can specify the type of files desired in P2P file-sharing systems. This avoids retrieving unwanted results and conserves network bandwidth. Two-thirds of queries do not specify a type as shown in Figure 1 (Indicated by “None” in the figure.). Of the queries that do specify type, over 75% are for audio files, approximately 19% are for video files, and the rest (less than 5%) are for the other types. Ostensibly, the queries that do not indicate a type also follow this distribution.

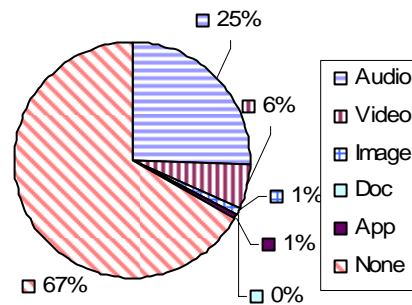


Figure 1. Query type distribution, September, 2006.

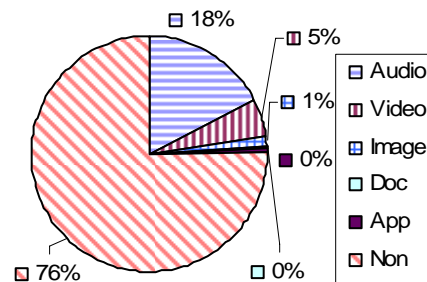


Figure 2. Query type distribution, April, 2007.

Today’s distribution of query types is similar as shown in Figure 2: most queries do not specify type, and most queries are for audio and video files. However, a greater proportion of queries do not specify type (76%). Of those that do, a smaller proportion is for audio files (71%), more are for video files (21%) and more are for the rest of the types (8%). This trend may suggest that the use

of P2P file-sharing systems is diversifying. Possible explanations for this may be either legal action by the recording industries or merely an evolution of their use.

The average query rate increased from 2006 to 2007 as shown in Figure 3. Over the two measured time periods, query rates increased by over 80% corroborating anecdotal evidence that P2P file-sharing is increasing in popularity.

Figure 3 also shows that query rate reaches a maximum in the evening, post-work hours. That queries are issued during non-work hours suggests that the use of P2P file-sharing systems is primarily recreational.

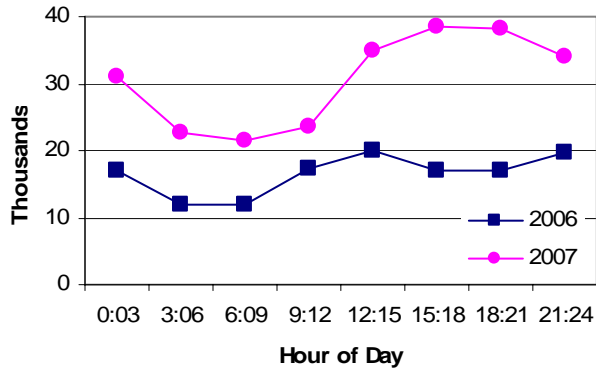


Figure 3. Average hourly query rate.

The average query length is shown in Figure 4, broken down by type. Over all queries, the average length is nearly three. Of the queries with known types, audio queries are the longest. One explanation for this phenomenon is that they are for known item searches, and often consist of a well-known artist and title. This is easily corroborated by examining the data, a sample of which we present below. Queries for other types (e.g., images) are less specific and therefore have shorter filenames. There is little difference between query length distributions from 2006 and 2007 data.

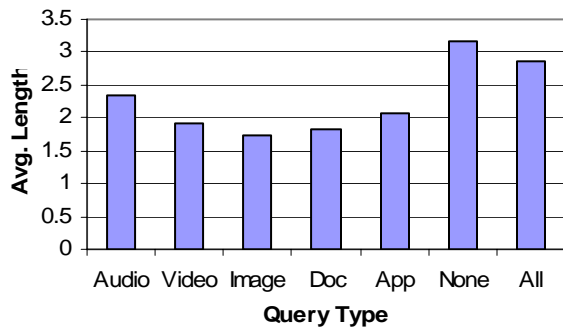


Figure 4. Average query length.

The distribution of queries to files is also highly skewed. Most queries are directed toward a few files. We divided our query set into 10 partitions and in the first partition placed the 10% most

common (unique) queries, in the next partition placed the next 10% most common queries, and so forth. Our results are shown in Figure 5. For audio files, for example, the top 10% most frequent queries constitute nearly 60% of all audio queries. The next 10% most popular queries constitute only about 10% of all audio queries. This trend is similar over all types and for both 2006 and 2007 data. Naturally, there is also a similar trend when performing the analysis with query terms instead of queries (not shown).

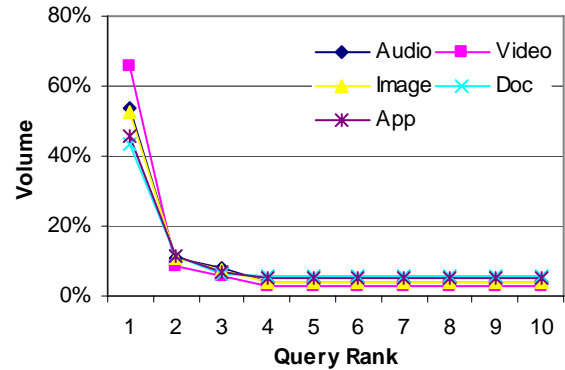


Figure 5. Distribution of query frequency.

An analysis of the actual content of the queries reveals that, although the particular files that are being searched for change over time, the “themes” of the desired content remain the same. Audio files are often love songs. Video and images are pornography-related. Documents are related to comics. Programs are related to operating systems and video games.

Table 1. Most popular queries and terms.

Top queries	Top terms
love	love
white nerdy	dj
know	remix
smack	los
young	like

a. 2006 audio files.

Top queries	Top terms
love	love
know	dj
ferhat gocer	remix
drank buy	los
drop lock pop	like

b. 2007 audio files.

4.2 Shared Data

Our shared content dataset consists of file information collected from 30,000 peers during September and October, 2006. We use the hash keys of files to uniquely identify replicas. We classified

Table 2. General statistics of the shared files.

Type	# Replicas	# Unique files	# Unique filenames	# Terms	# Unique terms	Avg. filename length
Audio	24 M	1.07 M	3 M	109 M	0.56 M	4.47
Video	1.12 M	0.1 M	0.3 M	7.53 M	0.11 M	6.75
Image	2.56 M	0.24 M	0.9 M	6.29 M	0.52 M	2.45
Document	1.26 M	0.13 M	0.27 M	2.42 M	0.18 M	1.92
Applica-tion	1.5 M	0.06 M	0.13 M	3.17 M	0.09 M	2.11
Unknown	3.59 M	0.35 M	0.62 M	8.53 M	0.38 M	2.37

files into types by their filename extensions (e.g., files with extension “.exe” are classified as application files, while files with extension “.pdf” are classified as document files). Files with unknown extensions are classified as “unknown”. In our results, we refer to each instance of a file as a “replica.”

4.3 Basic Filename Characteristics

General statistics of replicas for each type are reported in Table 2. Most shared files are of the type audio, which constitute over 79% of the files of known type. File sharing systems are still being used to trade music and this distribution of file types is consistent with the distribution of queries shown in Figure 1 and Figure 2.

The information in Table 2 also suggests that users do vary the way that they describe replicas of a given file. For example, although there are 1 million unique audio files, there are 3 million unique filenames – each file has 3 unique names, on average. However, that there are 24 million replicas suggests that 7/8 replicas have the same filename.

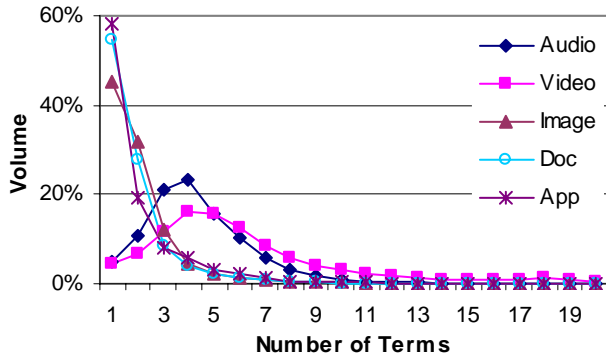


Figure 6. Filename length distribution.

The numbers of unique terms and average filename lengths vary over the different types of files as shown in Table 2; the distribution of filename lengths is shown in Figure 6. The types with the longest names are audio and video. Audio files often contain artist and title information extracted from well-known sources, such as freedb.org. Besides the filename length, the use of freedb.org also explains the lack of diversity in filenames for audio files (mentioned in the previous paragraph). Video files have long names because they are more descriptive of their contents. Fur-

thermore, many video files contain ads (i.e., spam) and to be included in more query results, contain more query terms. At the

Table 3. Replication degree of replicas.

Type	Max. # of replicas	Average replication factor
Audio	11,269	22.90
Video	1,793	10.68
Image	14,966	10.84
Document	6,157	10.00
Application	71,274	24.38
Unknown	64,531	10.23

other end of the spectrum are documents, images and programs. These file types have short filenames that are often unique, suggested by the high rate of unique terms compared with the number of unique files. Notice that, after removing the file extension, about 50% of files of type image, document, and application contain only one term (Figure 6). The generally short descriptions of shared data suggest that many queries will fail due to overspecification, necessitating specialized query processing techniques [15][16].

4.4 Replication of Data

The data in Table 3 shows how widely files are replicated, which can be extensive. The average replication factor (the number of replicas of a unique file) for audio files is 22.90. This suggests that the use of group size ranking, common in most available P2P file-sharing software, may be reasonable: a minimum requirement of group size ranking is that there is more than one replica of the desired file represented in a result set. The average replication factor for video, image and document is about 10, which is half of the replication factor of audio and application type. This is a consequence of the lower popularity of these types of data.

However, note that some peers are much more active in sharing files than others. As shown in Table 4, the average peer (who shares at least one audio) shares over 800 audio files. These data, however, are highly skewed. The standard deviation of the average number of replicas per peer is 1284.54, which makes it greater than the average number of replicas (819.13), so many

Table 4. Replica distribution over peers.

Type	# Peers	Max # replicas per peer	Avg #replicas per peer	Std dev # replicas per peer
Audio	29953	26336	819.13	1284.54
Video	27255	2405	40.91	81.3
Image	16951	18596	151.29	568.58
Document	13254	9414	95.27	307.44
Application	21461	27009	70.01	310.35
Unknown	25386	12900	141.54	413.22

peers share almost nothing. Furthermore, our data only includes data from those peers that share at least one file.

Also skewing our data are those peers that share an extraordinary number of files; one shared more than 26,000. The distribution of files over peers may be caused by large peers who may have misleading content (e.g., spam) to share. To share 26,000 files, each of size 3MB (the typical size of a song ripped from a CD) requires 75GB of disk space. Transmitting these files over the Internet would consume even more resources. It is therefore likely that the server of these files has some incentive (e.g., paid advertising) to share them.

Another indication of the spurious quality of the data available in P2P file-sharing systems is the replication factor of some of the files, which in some cases *exceeds* the number of peers examined. For example, there exists a file of type application that has over 71,000 replicas, even though we only examined the shared files of

Table 5. Analysis of filename variance.

Type	# Replicas	Avg. # terms	# Unique terms	Avg. overlap
Audio	11334	4.51	162	.027
	11172	7.06	16281	.00004
	11124	5.91	152	.039
Video	1796	20.90	462	.045
	1557	19.61	433	.045
	1490	20.29	466	.043
Image	14966	2	3	.667
	10059	2	2	1
	9539	2.003	31	.065
Doc	6157	2.000	26	.077
	5966	2	10	.2
	4246	2	9	.222
App	71274	4.981	13223	.0003
	61408	4.910	13345	.0003
	28521	4.936	10464	.0004
Unknown	64553	5.07	27894	.0001
	63491	4.81	21761	.0002
	19182	4.43	6245	.0007

Table 6. Most frequent audio terms, phrases and filenames.

Top filenames	Top terms	Top correlated terms	
track	ft	182	blink
cassie	love	pablo	petey
angel lips hinder	Feat	keys	alicia
back sexy justin timberlake	remix	shania	twain
fergie london bridges	lil	fighters	foo

30,000 peers. (We would expect that most users only share one replica of a given file, which would lead to a maximum replication factor of 30,000.)

By analyzing the variance in the way replicas are described, we can likely detect whether a file is of spurious quality. In Table 5, we present information on the average filename length and total number of unique terms used to describe the most highly replicated files of each type. There is clearly a significant amount of variance in the number of unique terms used to describe each file. We quantify this variance by computing the ratio of the average filename length to the total number of unique terms in the “Avg. overlap” column. The files with the high average overlap likely have easily discerned contents. For example, the image file with the average overlap of 1 has two terms in its filename - “lilmewire” and “gif” – making its identity clear. On the other hand, the application file with the 71,274 replicas, described by 13,223 unique terms is associated with the clearly unrelated terms, “autocad,” “café,” and “slam dunk.” This file is very likely spam, served by a malicious peer. Filename analysis in this way may be effective in detecting spam, clearly a large problem in P2P file-sharing systems.

4.5 Filename and Term Distribution

In Figure 7, we show the distribution of filenames for each file type. Filenames are ranked by their frequencies, and partitioned into 10 groups. The first group contains the 10% most frequent filenames, the second group contains the next 10% most frequent filenames, as so forth. For all file types, the first 10% of filenames account for from 60% to more than 80% of the filenames for the available replicas, while the next 10% of files are account for about 10% of the filenames for the available replicas. This highly skewed distribution makes us believe that searching for popular files is not difficult using current Gnutella query processing technique (particularly flooding and group size ranking). However, searches for rare files are most likely fail and thus require alternative techniques (e.g., [11][12]).

Similarly, the distribution of terms in filenames exhibits a high degree of skew (not shown). About 90% of term occurrences in filenames are covered by the 10% most frequent terms. Obviously, the skew in term distribution has an important effect in indexing the filenames. For example, more information should be allocated to highly frequent terms in order to improve search performance and reduce indexing cost.

In Table 6, we report the most frequent terms, phrases (correlated terms), and filenames for audio files. To identify the correlated terms, we considered the top 10,000 most frequent filenames for each type. Among these filenames, we extract the top 1000 most

frequent terms and compute the Pearson coefficient for each pair of them. The pairs of terms reported in Table 6 are ranked by their Pearson coefficient in descending order.

From the data in Table 6, it is obvious that the most replicated audio files are famous soundtracks that are downloaded (and shared) by many users, e.g., “sexy back – justin timberlake” and “fergie – london bridges”. The filename “track” is most frequent because many audio soundtracks that are yet annotated have the filenames “track 01”, “track 02”... as placeholders. We removed the numbers (e.g., “01”) in a preprocessing step.

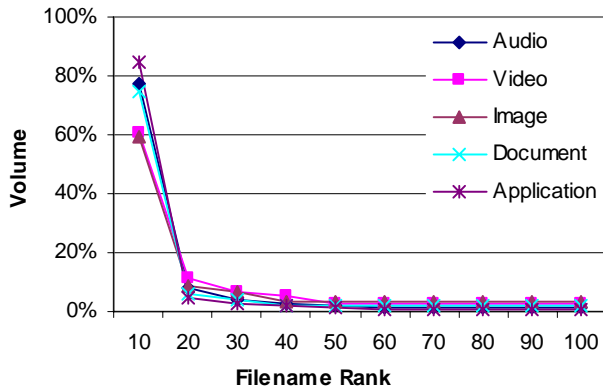


Figure 7. Distribution of filenames.

The most frequent terms used in video and image filenames (not shown) suggest that a large number of video and image files are pornography-related. This is consistent with our findings about queries in peer-to-peer file sharing systems, that queries for video and image file types are very likely related to pornography.

4.6 Content Distribution across Peer Population

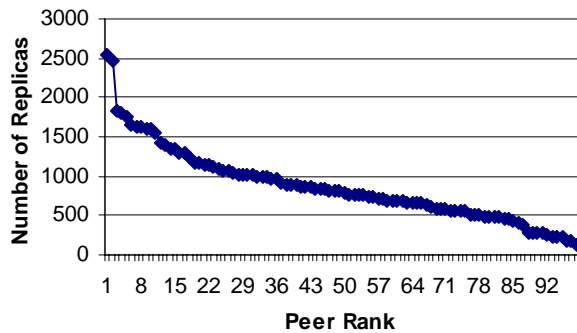


Figure 8. Distribution of replicas over 100 random peers.

We now turn our attention to the relationships among peers in terms of their similarity in shared content. We consider the set of files each peer is sharing and compare this set to those of other peers. To do this analysis, we selected 100 peers that share at least 100 files at random. We use 100 peers in this experiment to keep the problem tractable, and we use a threshold of sharing 100

files because, from our observations, peers who share fewer tend to have uninteresting similarity characteristics (e.g., they share files that no one else does). Collectively, the 100 peers share 85,036 replicas, distributed as shown in Figure 8.

To analyze peer file set similarity, we created a graph where each node is a peer, and an edge is drawn between two nodes if they share at least one file with the same hash key. The size of the node and the weight of the edge is proportional to the number of shared files and the number of files in common, respectively.

The resulting graph is not very informative, as it is too connected. There are 4,859 edges in the graph, out of a possible 4,950. The standard deviation in degree is 28.45. The distribution of node degree over the nodes is shown in Figure 9. Surprisingly, it is linear. Note that the degree of a node is only weakly related to the number of files it shares: their coefficient of correlation is 0.05.

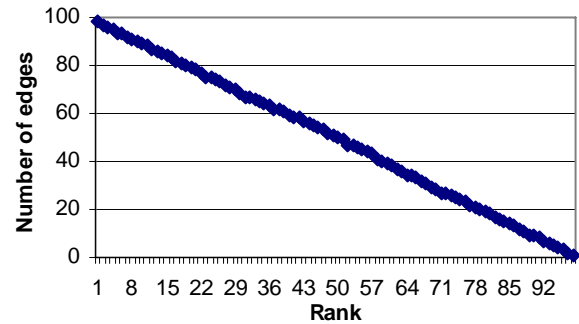


Figure 9. Distribution of node degree.

One possible reason for the linear degree distribution is that certain files are highly and uniformly replicated, resulting in many edges and uniformly distributed degrees. This reasoning is corroborated by the replication degree data we reported above. To circumvent this representation problem, we modify our graph to allow each node to maintain only its top five edges based on weight. The resultant graph (generated with the aiSee graph visualization software) is shown in Figure 10. Emerging is a graph that contains some nodes that are more centralized and highly connected than others. These are the nodes that share more popular files and therefore have more linkages to others. Note that the nodes toward the center of the graph have degrees greater than five because other nodes are connecting to them. The nodes at the borders generally have lower degrees. Furthermore, the border nodes are smaller (often a single point) because they share fewer files.

To reveal how the more “powerful” nodes connect to each other (nodes that share many files), we redrew the graph three times, restricting edges to those that represent at least 50, 100, and 200 files in common, respectively, as shown in Figure 11. By restricting the graph in this way, we can clearly see some clustering based on common interests. In Figure 11b, two clusters are connected by two nodes, (i.e., removing these nodes makes the two clusters disconnected). In Figure 11c, two clusters are connected by a single node.

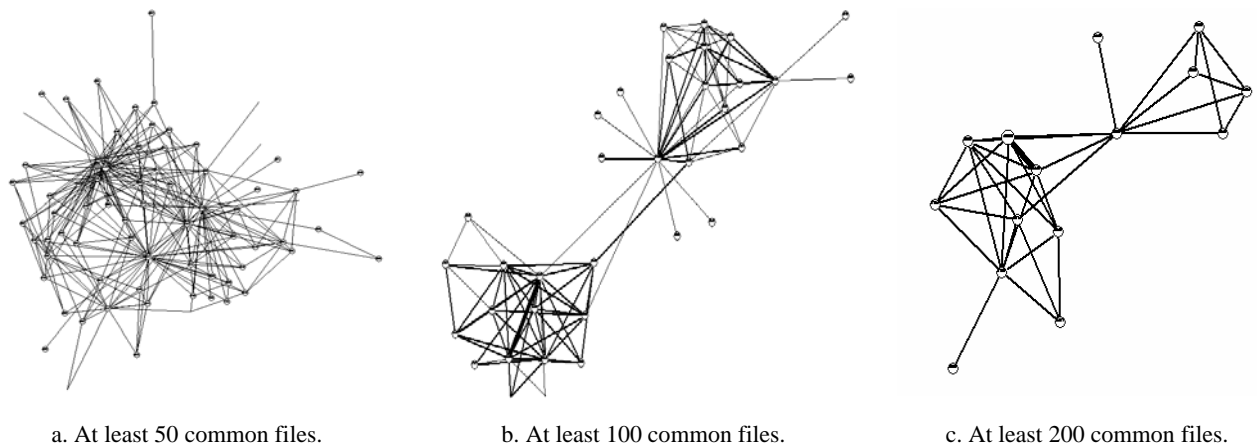


Figure 11. Node graph with various edge thresholds.

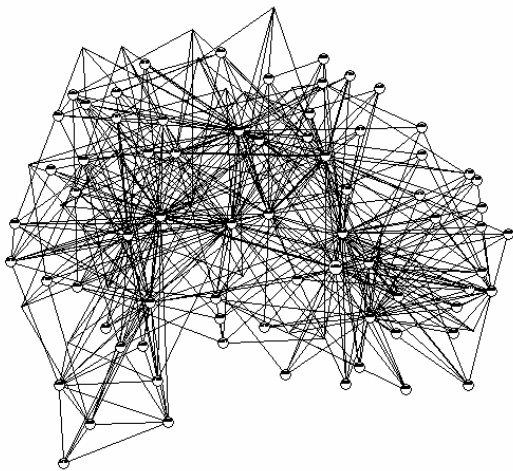


Figure 10. Node graph with top-5 edges.

The analysis reveals that there is a degree of interest-based clustering in P2P file-sharing systems. Most nodes share some degree of commonality, however, very few nodes share a high degree of interest with others.

This type of analysis is particularly important to research in the area of topology formation (e.g., [13][17]) in P2P networks. Such research relies on the fact that some peers are more similar to others in interests in order to create localized clusters of nodes that can mutually satisfy queries. The results above indicate that this is indeed the case.

5. CONCLUSION AND PROPOSED RESEARCH

Our study revealed important characteristics of queries and shared data in Gnutella, one of the today largest P2P file-sharing systems. It has been shown that the data in the Gnutella system is extensive and distinct in character from that of the Web. Also, the use of P2P file-sharing is increasing, as the query rate during

April, 2007 is almost doubled compare to that of September, 2006. This clearly indicates the significance of the P2P file-sharing application, and thus the importance of effective search techniques for them.

Based on our findings, we argue that the search technique used in unstructured P2P file-sharing systems like Gnutella (e.g., limited query flooding technique controlled by a time-to-live parameter), is not always efficient. As one of our findings indicates that there are many files with very low replication factors, a limited query flooding technique is likely fail to find them. Thus techniques to find rare files are required. One possible approach is estimating the replication factor of a possibly rare file, and if it is indeed rare, replicating it to other peers.

To search more efficiently, one possible approach is to cluster peers whose shared contents are similar via network links. As an initial step, our analysis shows that peers' contents exhibit a clustering characteristic. The remaining issues are how a peer discovers similar peers and how peer clusters can be maintained. Some techniques are proposed in [18], however, their next step is to demonstrate the effectiveness of their techniques with real world data (e.g., the data sets we collected).

Our finding about the skew of queries, query terms, filenames and terms in filenames can be used to improve system performance as well. For example, result caching is helpful for popular queries. Also, the difficulty of a query can be estimated based on the popularity of query terms, e.g., if a query contains one or more rare terms, it is very likely a difficult to satisfy query, and thus special technique might be required, for example, a higher time-to-live parameter. An analysis of query term distributions can also be useful in spam detection and security in general, which is an important issue in any unstructured, open system.

In summary, the understanding of the nature of a search system is vital in improving its performance. Knowledge about data characteristics can be used to guide the process of designing search techniques. Our study makes clear certain characteristics of the data that are being shared, what users are looking, and how they are searching for it in the Gnutella file sharing system.

6. REFERENCES

- [1] L. T. Nguyen, W. G. Yee, D. Jia, and O. Frieder. *A Tool for Information Retrieval Research in Peer-to-Peer File-Sharing Systems*. IEEE ICDE'07, Apr., 2007.
- [2] T. Klingberg and R. Manfredi. Gnutella Protocol 0.6, Web Document, 2002, rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html.
- [3] T. Mennecke, Interest in File-Sharing at All Time High, April 27, 2005, <http://www.slyck.com/news.php?story=763>.
- [4] W. G. Yee and O. Frieder, The Design of PIRS, a Peer-to-peer Information Retrieval System, In Proc. DBISP2P Workshop, 2004.
- [5] S. E. Robertson, S. Walker, S. Jones, M. M. Hancock-Beaulieu, M. Gatford 1995. Okapi at TREC-3. In Proc. of the Third Text REtrieval Conference (TREC-3). NIST Special Publication, pp. 500-225, 1995.
- [6] S. Brin, L. Page: The Anatomy of a Large-Scale Hypertextual Web Search Engine. *WWW7 / Computer Networks* 30(1-7): 107-117 (1998).
- [7] W. G. Yee, D. Jia, L. T. Nguyen, Search in Peer-to-Peer File-Sharing System: Like Metasearch Engines, But Not Really, In Proc. Workshop on Open Source Web Information Retrieval, Sept. 2005. ISBN:2-913923-19-4, p. 35-38.
- [8] W. Meng, C. Yu, and K.-L. Liu. Building efficient and effective metasearch engines. In *ACM Comp. Surveys*, 34(1):48-84, Mar. 2002.
- [9] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In Proc. *Multimed Comp. and Netw. (MMCN)*, 2002.
- [10] M. Ripeanu, I. Foster and A. Iamnitchi. Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System. In *IEEE Internet Computing* 6(1), 2002.
- [11] B. Loo, J. Hellerstein, R. Huebsch, S. Shenker and I. Stoica. Enhancing P2P File-sharing with an Internet-Scale Query Processor. In *Proc. VLDB Conf.* Aug., 2004.
- [12] W. G. Yee, D. Jia, and O. Frieder, Finding Rare Data Objects in P2P File-sharing Systems, In *Proc. of the Fifth IEEE Intl. Conf. on Peer-to-Peer Computing*, Sept. 2005.
- [13] V. Kalogeraki, D. Gunopulos, D. Zeinalipour-Yazti. A local search mechanism for peer-to-peer networks. In *Proc. CIKM*, 2002.
- [14] L. T. Nguyen, D. Jia, W. G. Yee, and O. Frieder. Analysis of Query Logs in Gnutella Peer-to-Peer Network. In *Proc. ACM SIGIR Conf.*, Amsterdam, July 2007.
- [15] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic Ranking of Database Results. In *Proc. VLDB*, 2004.
- [16] W. G. Yee, L. T. Nguyen, and O. Frieder. Masked Queries for Search Accuracy in Peer-to-Peer File-Sharing Systems. In *Proc. IEEE IPDPS*, 2007.
- [17] S.-T. Goh, P. Kalnis, S. Bakiras, K.-L. Tan, Real Datasets for File-Sharing Peer-to-Peer Systems. *DASFAA 2005*: 201-213.
- [18] M. Khambatti, K. Ryu, and P. Dasgupta, Efficient discovery of implicitly formed peer-to-peer communities, *Int'l. J. Parallel and Distr. Sys. and Networks*, 5(4), 2002, pp. 155-164.
- [19] D. G. Deschenes, S. D. Weber, and B. D. Davison, Crawling Gnutella: Lessons Learned, Lehigh Univ. Tech Report, LU-CSE-04-005, 2004.

An Epidemic-based P2P Recommender System

Jie Yang¹, Jun Wang¹, Maarten Clements¹, Johan A. Pouwelse¹, Arjen P. de Vries^{1,2},
Marcel Reinders¹

Faculty of Electrical Engineering, Mathematics and Computer Science¹,
Delft University of Technology, Delft, The Netherlands

CWI², Amsterdam, The Netherlands

{j.yang, jun.wang, m.clements, j.a.pouwelse, m.j.t.reinders}@tudelft.nl,
arjen@acm.org

ABSTRACT

Peer-to-peer (P2P) networks have developed into popular media to share and seek information. Given the large amount of information available, it is of great interest to design a distributed recommender system, to personalize the information seeking in these networks. This paper describes a distributed collaborative filtering framework. In this framework, we introduce an item ranking model for collaborative filtering, inspired by the Probability Ranking Principle (PRP) of information retrieval. However, the probability estimation for the item ranking requires a centralized database to store user preferences. Within a P2P network such a centralized database is not readily available. To overcome this problem, we developed a novel preference exchange algorithm called *BuddyCast*, based on the epidemic protocol. Under this overlay network, the distributed item ranking is realized by fully decomposing the computation loads of the model and preference data into the entire network. The formal derivation and analysis in this paper help us to outline and motivate possible future directions of research in P2P Recommender Systems.

The distributed recommendation framework described in this paper has been implemented in our Open Source P2P file sharing software Tribler (tribler.org).

Keywords

Peer-to-Peer Networks, Recommender Systems, Collaborative Filtering, Retrieval Models, Relevance Ranking

1. INTRODUCTION

The rapid progress in multimedia processing, communications, and storage technologies not only changes the availability of data, but also the way in which people interact with it. Particularly, peer-to-peer (P2P) networks have become a new and popular medium for people to exchange information, stored on their local devices. Examples of P2P file sharing systems include: Emule (EMule-Project.net), BitTorrent (BitTorrent.com), etc. These networks increase content availability dramatically, since the involvement of

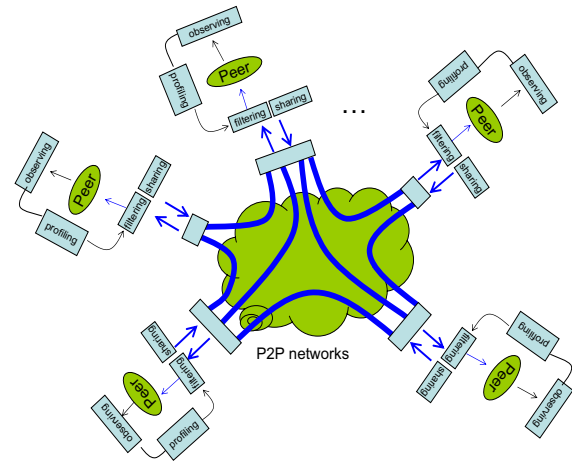


Figure 1: A schematic overview of a peer to peer file sharing system, that filters the view on the network based on the users observations.

third parties that manage the centrally stored information can be avoided.

A schematic overview of a P2P file sharing system is given in Fig. 1, showing that each peer performs two roles in the network: information seeker and sharer. The overwhelming information availability in most P2P networks causes the information seeking task to be difficult and time consuming. Therefore, a filter based on the peer's preference is needed to extract interesting content from the wealth of data in the network. Another problem in P2P networks lies in the fact that both users and content are distributed and dynamically changing. These characteristics make the design of content filters or search engines a challenging task.

To manage the poorly organized information and filter relevant content that fits the user's interest within a P2P network, techniques to create a semantic overlay need to be explored. In P2P research, some semantic structures [3, 20, 24] are proposed, that deploy the similarity between content descriptions or user demographics. In practice, the use of descriptions like meta data is problematic, since it is often unavailable or incomplete. Another way to describe the content is by making use of low-level features of the data, such as proposed in content-based multimedia analysis [6, 19]. However, these approaches often require that the data to be compared is from the same modality, which is an undesirable property when one wants to share *multimedia* files. Thus, there is a need for a more generic content similarity

measure that does not demand meta data and/or low level features.

Research on recommender systems has provided similarity measures that can be used to derive the relevance of an item to the preference of a user. Many current recommender systems are based on collaborative filtering, a filtering technique that derives similarities between users (user-based) or items (item-based) from a database of the users' rating or viewing profiles. Within the context of P2P networks there is, however, no centralized user profile database, so that current collaborative filtering approaches cannot be applied directly.

This paper presents our ongoing research on distributed collaborative filtering. We aim at providing an efficient and effective solution to recommend information items in a peer-to-peer environment. In this work, we first introduce an item ranking model for recommendation. To make the recommendation scalable in the P2P network, we then propose an epidemic (gossip based) approach [5, 8], using a user profile exchange algorithm called *BuddyCast*.

The remainder of the paper is organized as follows. We first summarize related work and then introduce our peer-to-peer recommendation system, providing our formal item ranking model of collaborative filtering and giving its distributed calculation. A conclusion is given to point out future directions.

2. RELATED WORK

2.1 Collaborative Filtering

Generically, collaborative filtering (CF) is any algorithm that filters information for a user, based on a collection of user profiles. The approaches to CF can be divided into two main categories: 1) memory-based methods. Examples are: item correlation-based methods [9], item clustering [4], user clustering [28], and unified methods [26]. 2) model-based methods. Examples are: decision trees [1], latent class models [7].

Recently, a few early attempts towards decentralized collaborative filtering have been introduced [2, 13, 14, 15, 23]. In [13], five architectures are proposed to find and store user rating data to make rating based recommendation, namely, a central server, random discovery similar to *Gnutella*, transitive traversal, Distributed Hash Tables (DHT), and secure Blackboard. These solutions aim to aggregate the rating data in order to make a recommendation and they hold independently of any semantic structure of the networks. This inevitably increases the amount of traffic within the network.

Different from these methods, we implicitly learn the users' interests from their interaction data. Most importantly, we take the epidemic algorithm as our basic overlay network to rank items, making any static structures unnecessary. In addition, in the overlay networks, we take user proximity into account when peers exchange user profiles, making the recommendation efficient and scalable.

2.2 Peer-to-Peer Networks

For a recent comprehensive survey on P2P networks we refer to [12]. Different index techniques for content located at different peers exist, such as: a local index (the owner of the data is only able to index the data, like in early *Gnutella*), the central index (a centralized server organizes indices to data residing at peers, like in *Napster*), and the distributed

index (other peers are also able to index the data residing at a peer, like in *Freenet*).

Jelasey and Van Steen [8] introduced newscast, an epidemic (or gossip) protocol that exploits randomness to disseminate information without keeping any static structures or requiring any sort of administration. Although this type of protocol successfully operates in dynamic networks, its lack of content-awareness restricts it to perform in an efficient way.

Another major indexing approach makes use of DHTs [12, 21]. In a DHT each location (index) is mapped to a unique key, and each peer maintains a certain range of the keys. In this way each peer generates a well-defined structure that can be used for routing queries that is scalable to some extent. However, an extension is necessary to perform a search based on arbitrary queries, rather than key lookups [12].

Recently, semantic indexing and routing techniques have been proposed to capture relationships between content [12, 22]. Distinct semantic groups of documents [3], or users [20, 24] are identified to create Semantic Overlay Networks (SONs). A document request is then handled by the overlay to which this document presumably belongs (based on either clusters of documents or peers). However, to match queries to documents, a content description (in the form of meta-data) is required. Identifying similarities between peers or non-textual content turns out to be difficult to establish in the absence of this information.

In this paper, meta-data and demographics are redundant, as user similarity is calculated by the concurrence of two users' preference profiles and a semantic overlay is created among users by randomly exchanging the *Buddycast* messages. We shall see that such an overlay network lies at the heart of the item ranking in P2P networks as it provides an effective way to collect the most valuable profiles for the purpose of ranking.

3. A P2P RECOMMENDER SYSTEM

In this section, we first describe an item ranking model for recommendation. We then realize the ranking in a distributed and dynamic manner by proposing an epidemic-based algorithm for exchanging user profiles. Notice that such proposed distributed recommendation algorithm has been implemented in our file sharing software [16]. Please refer to *Tribler.org* for detailed information and downloading.

3.1 Item Ranking

Our task for collaborative filtering is to find items that are relevant (useful) to a given user (his or her interest is implicitly indicated by a user profile). The well-known Probability Ranking Principle (PRP) of information retrieval [17] states that ranking documents in descending order by their probability of relevance produces "optimal" performance under reasonable assumptions [17]. Thus, it is natural to adopt the PRP for our recommendation task, treating the user profile as a query and rank items. For this, we need to introduce the concept of "relevancy" into collaborative filtering. By analogy with the relevance models in text retrieval [10, 18], the top- N recommendation items can be then generated by ranking items in order of their probability of relevance to the user profile or the underlying user interest.

To be able to rank items, this section formulates the estimation of the probability of relevance between an item

and a user (profile), using the following notation. Let u be a discrete random variable over the sample space of users $\Phi_U = \{1, \dots, M\}$, let i be a random variable over the sample space of items $\Phi_I = \{1, \dots, K\}$, and let R be a random variable over the relevance space Φ_R , where R is either ‘relevant’ r or ‘non-relevant’ \bar{r} .

In a probabilistic framework, we can generate a top- N item ranking list in order of their estimated log-odd of relevance: $\ln \frac{p(r|u,i)}{p(\bar{r}|u,i)}$. Using Bayes’ rule gives the following ranking formula:

$$o_u(i) = \ln \frac{p(r|u,i)}{p(\bar{r}|u,i)} = \ln \frac{p(i|u,r)}{p(i|u,\bar{r})} + \ln \frac{p(r|u)}{p(\bar{r}|u)} \quad (1)$$

where the last term can be discarded as it is independent on the target item. Notice that this is not only derivation. For other detailed derivations, please refer to our paper [25].

The relevancy between items and users can be explicitly obtained by asking users to rate items (content) that they know. However these explicit ratings are hard to gather in a real system. It is highly desirable to infer user profiles from implicit observations of user interactions with the system. In our system, for the sake of simplicity but without loss of generality, we only observe the positive evidence. By following the language modelling of information retrieval [10], we now assume equal priors for item i in the non-relevant case; Notice that these two negative terms in Eq. 1 can always be added to the model when the negative evidences are captured. Then, the non-relevance term can be removed and the ranking formula becomes:

$$o_u(i) \propto p(i|u,r) \quad (2)$$

where the probability $p(i|u,r)$ cannot be directly estimated because we need to predict “new” items, i.e., those that do not exist in the given user profile. To address this problem, we represent item explicitly by user’s judgment, such that they can be linked to the target user u . Formally, we introduce a list L_i for each item i , where $u \in L_i$ denotes that user u is in the list. This list enumerates the users who have expressed interest in the item i . Replacing i with L_i , we have:

$$o_u(i) \propto \sum_{\forall u': u' \in L_i} \log p(u'|u,r) \quad (3)$$

where the probability $p(u'|u,r)$ depicts the similarity between two users u and u' , which can be estimated using user profiles: counting the number of items that both users liked (denoted as $c(u', u)$), divided by the total number of items that user u liked (denoted as $c(u)$):

$$p(u'|u,r) = \frac{p(u', u|r)}{p(u|r)} := \frac{c(u', u)}{c(u)} \quad (4)$$

However due to the data sparsity, many co-occurrence counts of two users may be zero. To counter the sparsity and remove the zero probabilities, we propose to use the Bayes-smoothing technique [29] to further smooth the estimation. More formally, we have:

$$p(u'|u,r) := \frac{c(u', u) + \mu \cdot p(u'|r)}{c(u) + \mu} \quad (5)$$

where μ is the smoothing parameter and $p(u'|r) := \frac{c(u')}{\sum_u c(u')}$.

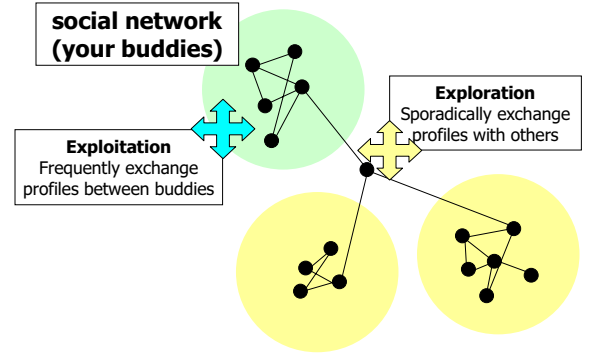


Figure 2: Exploitation v.s. Exploration.

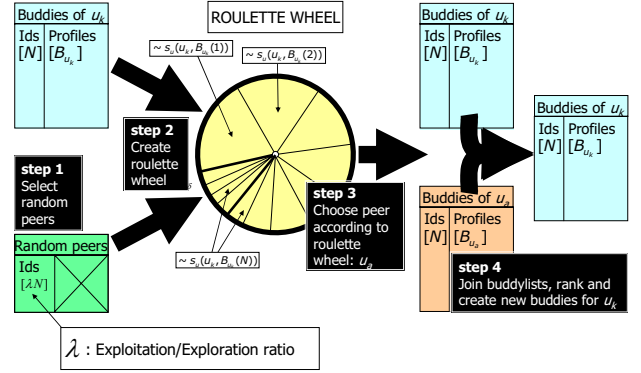


Figure 3: Peer Selection.

In summary, we now have our final ranking formula if we replace Eq. 5 into Eq. 3.

$$o_u(i) \propto \sum_{\forall u': u' \in L_i} \log \frac{c(u', u) + \mu \cdot \frac{c(u')}{\sum_u c(u')}}{c(u) + \mu} \quad (6)$$

3.2 Decentralized Ranking

It is easy to compute the ranking in a centralized server, but it is non-trivial in a P2P network because user profiles are distributed in the entire P2P network and for a particular peer (user), there is no centralized user profile database to be used to calculate the co-occurrence. Thus the ranking of items for a particular user (peer) has to be calculated locally in that peer.

Eq. 6 gives a formal ranking solution for collaborative filtering, indicating that, in order to make a recommendation - ranking the relevance of a target item towards the user, we mainly need to estimate the co-occurrence $c(u', u)$ of the target user u towards other users u' who have already expressed interest in the target item. Closely looking at Eq. 6, we find that, for the target user, those items that have been liked by the similar users will have high ranking scores when we rank the relevance to that target user. Thus an efficient and scalable way to calculate the ranking formula in Eq. 6 is to find the similar users to the target user and only rank items that they liked, rather than collecting *all* user profiles and ranking *all* items.

In this regard, we introduce the Buddycast algorithm, which is used to efficiently exchange the profiles of simi-

lar users. The Buddycast algorithm is based on an epidemic protocol [8] and works as follows (see Fig. 4). Each peer maintains a list of its top-N most similar peers along with their current preference lists. Similarities between preference lists are measured using the co-occurrence $c(u', u)$. Periodically, a peer connects to either (a) one of its buddies to exchange social networks and preference lists (exploitation), or (b) to a new peer, randomly chosen, to exchange this information (exploration). This is illustrated in Fig. 2. To maximize the exploration of the social network, every peer also maintains a list with the K most recently visited random peers, and avoids reconnecting to a peer already present in the list.

In contrast to other epidemic protocols such as Newscast [8], we use both exploitation and exploration branches, we limit the randomness of peer selection during the exploration, and we implicitly cluster peers into social groups (having common interest). To find a good balance between exploitation and exploration, the following procedure is therefore adopted (see Fig. 4 (b)). First, $r \cdot N$ random peers are chosen, where $r \geq 0$ is the exploitation-to-exploration ratio. Then, these random peers are joined with the N taste buddies in a single ranked list, with the random peers being assigned the lowest ranks. Then, one peer is randomly chosen from this ranked list according to a roulette wheel approach (probabilities proportional to the ranks), which gives taste buddies a higher probability of being selected than the random peers. Once a peer has selected some other peer, the buddy lists of the two peers are joined. The first peer then ranks the composite list according to the preference list similarities with its own preference list, and retains only the top-N best ranked peers. The peer selection step is illustrated in Fig. 3.

After collecting similar user profiles, each peer applies the ranking formula in Eq. 6 using the calculated co-occurrences to rank the items that are presented in these profiles.

4. DISCUSSIONS

Two views This paper has combine work in the field of information retrieval (e.g. the PRP of information retrieval) and that of P2P networks (e.g. an Epidemic-based overlay network). It takes a user oriented view of the problem as the user proximity lies at the center of our algorithm, both for ranking items and exchanging user profiles. Notice that there is an item oriented view of the recommender problem, which considers the item similarity instead [4, 25]. We refer to [27] for the discussion of this type of recommendation and its usage in P2P networks.

Practical Considerations The proposed P2P recommendation algorithm has been used in our P2P file sharing software Tribler since March 2006 and has been downloaded more than 100,000 copies. For such a real P2P environment, we have addressed some of the practical issues. Firstly, we establish a “strong” identifier for each peer since we do not have a central indexing server to index online users and their network addresses (IP addresses). Tribler has created a public key as a permanent identifier for each peer (client). Secondly, the network is very dynamic. Peers arrive and leave frequently. According to our observation, there are about 1/3 peers behind firewalls. To avoid including dead or unreachable peers in the Buddycast messages, each peer always maintains an online peer list, checking their connectability and keeping connected with those peers that have been con-

```

1 BuddyCast( $p_i$ ) {
2   do forever {
3      $e = \text{waitForEvent}()$ ;
4     if  $e$  is TIMEOUT {
5        $p_j = \text{selectPeer}(B_i)$ ;
6       send  $B_i$  to  $p_j$ ;
7       receive  $B_j$  from  $p_j$ ;
8        $B_i = \text{merg}(B_i; B_j)$ ;
9        $B_i = \text{updateSim}(B_i)$ ;
10       $B_i = \text{selectTopN}(B_i)$ ;
11    }
12    if  $e$  is message  $B_j$  from  $p_j$  {
13      send  $B_i$  to  $p_j$ ;
14       $B_i = \text{merg}(B_i; B_j)$ ;
15       $B_i = \text{updateSim}(B_i)$ ;
16       $B_i = \text{selectTopN}(B_i)$ ;
17    }
18  }
19 }
```

(a) Main Routine

```

1 selectPeer( $B_i$ ) {
2    $\{p_r\} = \text{randomPeers}(p_i, r \cdot N)$ ;
3    $\{w_r\} = \text{minWeight}(B_i)$ ;
4    $B_i = B_i + \{(p_r, w_r)\}$ ;
5    $B_i = \text{normalizeWeight}(B_i)$ ;
6   randomly select  $p$  according to  $w$ .
7 }
```

(b) Peer Selection

Figure 4: The Buddycast Algorithm. r is the exploration/exploitation ratio, B_i is the Buddy Cache.

tacted recently. Thirdly, we limit our storage of user profiles by removing dissimilar and old peers.

Our algorithm is scalable because each peer only keeps a small view of the whole network in which he or she is interested. Thus, there is less local storage required for each peer and therefore the computation is also reduced. In addition, the algorithm can cope with the dynamic nature of the network as our recommendation can update its results once more peers and their preferences are discovered.

Planned Evaluation We plan to conduct several experiments to evaluate the effectiveness of the proposed algorithm. We consider using a real world trace collected from an active BitTorrent community (FileList.Org). From December 2005 to May 2006, we have logged 90,000 worldwide-distributed peers along with their profiles. The trace also contains the connectability, the arrival and departure time of each peer. Currently, we are developing a simulator for the distributed recommendation algorithm. We are specifically interested in the convergence behavior of the profile exchange and the relationship between the recommendation performance and the exploitation-to-exploration ratio r . We will also compare our algorithm with other alternatives in order to find out the best strategies and policies.

5. CONCLUSIONS

This paper decentralized a probabilistic item ranking model of collaborative filtering on the basis of the epidemic algorithm. The underlying distributed item ranking framework calls for interesting future work. Since the formulation of our item ranking model of collaborative filtering is similar to other more general information retrieval models, like the language modeling of information retrieval, the BM25 model, etc. it is of particular interest to seek the possible usage of the current distributed ranking framework to decentralize these text retrieval models [11, 22].

6. REFERENCES

- [1] J. Breese, D. Heckerman, and C. Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In *Proceedings of the 14th Annual Conference on Uncertainty in Artificial Intelligence (UAI-98)*, pages 43–52, San Francisco, CA, 1998. Morgan Kaufmann.
- [2] J. Canny. Collaborative filtering with privacy via factor analysis. In *SIGIR '02: Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 238–245, New York, NY, 2002. ACM Press.
- [3] A. Crespo and H. Garcia-Molina. Semantic overlay networks for p2p systems. Technical report, Comp. Sci. Dept., Stanford University, 2003.
- [4] M. Deshpande and G. Karypis. Item-based top-n recommendation algorithms. *ACM Trans. Inf. Syst.*, 22(1):143–177, 2004.
- [5] P. T. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulie. From epidemics to distributed computing. *IEEE Computer*, 2004.
- [6] A. Hanjalic. *Content-Based Analysis of Digital Video*. Kluwer Academic Publishers, 2004.
- [7] T. Hofmann and J. Puzicha. Latent class models for collaborative filtering. In *Proc. of IJCAI*, 1999.
- [8] M. Jelasity and M. van Steen. Large-scale newscast computing on the Internet, Oct. 2002.
- [9] G. Karypis. Evaluation of item-based top-n recommendation algorithms. In *Proc. of the tenth international conference on Information and knowledge management*, 2001.
- [10] J. Lafferty and C. Zhai. Probabilistic relevance models based on document and query generation. *Language Modeling and Information Retrieval, Kluwer International Series on Information Retrieval*, V.13:1–10, 2003.
- [11] J. Lu and J. Callan. Content-based retrieval in hybrid peer-to-peer networks. In *CIKM '03: Proceedings of the twelfth international conference on Information and knowledge management*, pages 199–206, New York, NY, USA, 2003. ACM Press.
- [12] E. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Survey*, 2004.
- [13] B. N. Miller, J. A. Konstan, and J. Riedl. Pocketlens: Toward a personal recommender system. *ACM Trans. Inf. Syst.*, 22(3):437–476, 2004.
- [14] T. Oka, H. Morikawa, and T. Aoayama. Vineyard : A collaborative filtering service platform in distributed environment. In *Proc. of the IEEE/IPSJ Symposium on Applications and the Internet Workshops*, 2004.
- [15] H. Peng, X. Bo, Y. Fan, and S. Ruimin. A scalable p2p recommender system based on distributed collaborative filtering. *Expert systems with applications*, 2004.
- [16] J. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. Epema, M. Reinders, M. van Steen, and H. Sips. Tribler: A social-based based peer to peer system. In *5th Int'l Workshop on Peer-to-Peer Systems (IPTPS)*, Feb 2006.
- [17] S. E. Robertson. The probability ranking principle in IR. pages 281–286, 1997.
- [18] S. E. Robertson and K. SparckJones. Relevance weighting of search terms. *Journal of the American Society for Information Science*, 27(3):129–46, 1976.
- [19] A. W. Smeulders, M. Worring, S. Santini, A. Gupta, and R. Jain. Content-based image retrieval at the end of the early years. *IEEE PAMI*, December 2000.
- [20] K. Sripanidkulchai, B. Maggs, and H. Zhang. Efficient content location using interest-based locality in peer-to-peer systems. In *Proc. of Infocom*, 2003.
- [21] I. Stoica, D. K. R. Morris, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of SIG-COMM*, Aug. 2001.
- [22] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *Proc. of SIGCOMM '03*, 2003.
- [23] A. Tveit. peer-to-paper based recommendation for mobile commerce. In *Proc. of the First International Mobile Commerce Workshop*, pages 26–29, 2001.
- [24] S. Voulgaris, A.-M. Kermarrec, L. Massoulie, and M. van Steen. Exploiting semantic proximity in peer-to-peer content searching. In *Proc. of the 10th IEEE Int'l Workshop on Future Trends in Distributed Computing Systems*, 2004.
- [25] J. Wang, A. P. de Vries, and M. J. Reinders. A user-item relevance model for log-based collaborative filtering. In *Proc. of ECIR06, London, UK*, pages 37–48, Berlin, Germany, 2006. Springer Berlin / Heidelberg.
- [26] J. Wang, A. P. de Vries, and M. J. T. Reinders. Unifying user-based and item-based collaborative filtering approaches by similarity fusion. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 501–508, New York, NY, 2006. ACM Press.
- [27] J. Wang, J. Pouwelse, R. Lagendijk, and M. R. J. Reinders. Distributed collaborative filtering for peer-to-peer file sharing systems. In *Proc. of the 21st Annual ACM Symposium on Applied Computing*, 2006.
- [28] G.-R. Xue, C. Lin, Q. Yang, W. Xi, H.-J. Zeng, Y. Yu, and Z. Chen. Scalable collaborative filtering using cluster-based smoothing. In *SIGIR '05: Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 114–121, New York, NY, 2005. ACM Press.
- [29] C. Zhai and J. Lafferty. A study of smoothing methods for language models applied to ad hoc information retrieval. In *SIGIR '01: Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 334–342, New York, NY, 2001. ACM Press.

Design Alternatives for Large-Scale Web Search: Alexander was Great, Aeneas a Pioneer, and Anakin has the Force*

Matthias Bender[†], Sebastian Michel[†], Peter Triantafillou[‡], Gerhard Weikum[†]
[†] Max-Planck-Institut fuer Informatik [‡] RACTI and University of Patras
66123 Saarbruecken, Germany Rio, 26500, Greece
{mbender, smichel,
weikum}@mpi-inf.mpg.de peter@ceid.upatras.gr

ABSTRACT

Indexing the Web and meeting the throughput, response-time, and failure-resilience requirements of a search engine requires massive storage and computational resources and a careful system design for scalability. This is exemplified by the big data centers of the leading commercial search engines. Various proposals and debates have appeared in the literature as to whether Web indexes can be implemented in a fully distributed or even peer-to-peer manner without impeding scalability, and different partitioning strategies have been worked out. In this paper, we resume this ongoing discussion by analyzing the design space for distributed Web indexing, considering the influence of partitioning strategies as well as different storage technologies including Flash-RAM. We outline and discuss the pros and cons of three fundamental alternatives, and characterize their total costs for meeting all performance and availability requirements. We give arguments in favor of a system design based on term partitioning over a DHT-based peer-to-peer network with modern top-k query processing and a judiciously designed combination of disk and Flash-RAM storage, and we show that this design has intriguing properties and a very attractive cost/performance ratio.

1. INTRODUCTION

1.1 Motivation

Indexing the Web for fast keyword search is among the most challenging applications for scalable data management. Major search engines such as Google use gigantic computing power, with large data centers consisting of thousands of low-end computers (i.e., PC technology in customized

*This work is partially supported by the EU project AEO-LUS.

racks), very carefully designed techniques for scalability and availability (i.e., caching, redundancy, load balancing), and customized software such as GFS [4, 12]. Although detailed figures are not publicly known, it is estimated that Google indexes about 20 Bio. Web pages and needs to sustain a throughput of about 2,000 queries/s (i.e., 150-200 Mio. queries per day) with peak loads being up to 10,000 queries/s. In addition, there are stringent response time requirements: user-perceived latency must not exceed 1 s; with wide-area network transfers taking several hundred milliseconds, this requirement typically translates into a desired upper bound of 100 ms on the data-center side.

Google, Yahoo!, and MSN can obviously meet these requirements, but there is a high cost involved for operating such huge data centers. Moreover, it is not clear how much more this architectural approach can be scaled up at reasonable cost if both data and workload continue to grow in the coming years. Baeza-Yates et al. estimate that by 2010 a big search engine like Google will need one million computers [3]. And if such numbers are not perceived as a mega challenge (even literally), consider a futuristic search engine for a comprehensive Web Archive. The Internet Archive (archive.org) aims to preserve the history of the Web, but even its current 500 TB collection, containing more than 1,000 Bio. page versions, is very partial at best and allows only URL-based access in the temporal dimension. A reasonably complete Web archive with support for full-fledged time-travel querying is way beyond today's feasibilities for scalable data management.

Therefore, designing and investigating alternative architectures to the current ones is highly relevant for future scales. Distributed indexing and peer-to-peer (P2P) systems are intriguing approaches, but it is not clear if and how they can cope with the gigantic dimensions of the Web and the challenging workload of an Internet search engine. Li et al. [14] raised doubts that a wide-area distributed architecture like a P2P network would be feasible at all, but more recently various projects on P2P search (e.g., [6, 18, 7, 26, 16, 22]) and high-performance indexing (e.g., [20, 2]) have developed techniques towards scalable architectures that may possibly become practically viable. Baeza-Yates et al. [3] pose distributed indexing for Web scale as a grand challenge and outline various research routes that give hope.

1.2 Contribution

This paper intends to outline and analyze several architectures that may lead to scalable solutions, while in the mean time picking up the challenges put forth by [14, 3]. The main purpose of the paper is to shed more light into the overall design space and understand the pros and cons of different approaches. We consider three major dimensions for structuring the design space and in particular:

- the partitioning of the index:
 - partitioning by documents (i.e., Web pages, news or blog items, photos, etc.) versus
 - partitioning by content feature (i.e. text terms or Flickr-style tags)
- the way index partitions are distributed and handled during query processing:
 - by utilizing clusters of computers in data center environments versus
 - utilizing large-scale geographically distributed networks (such as DHTs), and
- the storage technology that is primarily used for holding index partitions on a computer:
 - RAM (the current choice by major search engines),
 - disks (a more database-style approach), or
 - Flash-RAM which is the technology used in digital cameras, MP3 players, USB sticks, and mobile phones with a strong trend towards becoming a viable alternative to disk storage [13, 21, 8].

We present three major alternatives as particularly interesting points in the design space. We have coined these approaches Alexander, Aeneas, and Anakin, for reasons that will become clear later¹. Each of them will be analyzed in terms of total costs needed to provide sufficient space, redundancy, throughput, and acceptable response time. Alexander largely corresponds to the current solution that large search engines employ with RAM orientation and document partitioning; Aeneas is a P2P-style distributed approach with disks for low-cost storage and feature partitioning; and Anakin is a more futuristic but most promising design with Flash-RAM and term partitioning. We also discuss the critical issues of load balancing and index maintenance for each of these architectures. All architectures can be deployed in a local-area high-speed network environment like a data center, and some are also suited for wide-area networks like P2P networks; however, the different properties of LAN vs. WAN affect the architectures' abilities to scale up and perform well.

The rest of the paper is organized as follows. Section 2 briefly reviews relevant characteristics of Web-scale data, access load, and storage technologies. Section 3 presents the Alexander design. Section 4 presents the Aeneas design. Section 5 presents Anakin. Section 6 discusses load balancing issues for the presented approaches. Section 7 discusses the issues of index construction and maintenance, before Section 8 concludes this work and points at future research directions.

¹The sequence Alexander, Aeneas, Anakin reflects history, and we are increasingly moving away from reality to fiction - mythology and modern subculture.

2. BRIEF REVIEW OF TECHNOLOGY AND WEB SCALE

We briefly review some Web properties and technology parameters that are relevant for our comparison of architectures. We assume that the Web has 20 Bio. pages, each with an average of 500 bytes indexable information (i.e., excluding embedded images etc.), which yields a total index size of 10 TB. Another way of looking at index size is in terms of the number and length of index lists in the corresponding inverted file. We assume that there is a total number of 10 Mio. distinct terms (keywords and their many variations incl. names, typos, numbers, etc.) and that each Web page contains 100 indexable terms on average. This creates 20 Bio. * 100 = $2 * 10^{12}$ = 2 Trio. postings to the index where a posting captures a (*document, term, score*)-triple. Obviously, many Web pages have way more than 100 terms, but a large fraction of the 20 Bio. pages is known merely through href anchors pointing to them and such anchors include only a handful of terms. So an average of 100 terms per document seems reasonable. The average length of an index list in the inverted file thus is $2 * 10^{12} / 10^7 = 2 * 10^5$ postings. Each posting has a raw length of 10-20 bytes (two ids and a numeric score), but inside an index list they are highly compressed. We assume an amortized length of 5 bytes per posting. This results in a net length of 10^6 bytes = 1 MB per index list. All 10 Mio. index lists together consume 10 TB (which is consistent with our first calculation).

We assume that a standard computer used in a cluster or a P2P network is a low-end PC but configured with ample memory, say 4 GB, and costs \$1,000. We consider a single 1 TB disk at a cost of \$500 as a standard unit for disk storage (we will later consider also small RAID's with coarse-grained striping for higher sequential bandwidth). The disk has a random access time of 5 ms for a 10 KB block, thus yielding up to 200 random IOs/s, and a sequential transfer rate of 20 MB/s, yielding up to 2,000 sequential IOs/s (for transfers of multiple, contiguous 10 KB blocks). Finally, for Flash-RAM we assume 16 GB units at a price of \$200. Flash-RAM has about the same sequential transfer rate of 20 MB/s offered by a disk, but it has much shorter random access time (it is some kind of RAM after all), namely, 0.1 ms random access time for a small 2 KB block. This results up to 10,000 random IOs/s for small blocks and up to 2,000 sequential IOs/s for large blocks.

For query processing we assume that typical queries include only a few keywords, and for simplicity our crude analysis of query processing costs here assumes single-term queries. So for each query we need to fetch the corresponding index list from disk if it does not reside in RAM or Flash-RAM and we need to process it. We assume that the processing has a fixed startup time of 1 ms and then needs 1 ms for every 1,000 postings, that is, 200 ms for a full index list with $2 * 10^5$ postings. Reading the full list from a single disk takes $1 \text{ MB} / 20 \text{ MB/s} = 50 \text{ ms}$. Reading from disk and processing can be pipelined; so the processing time determines response time and throughput on a single computer. The sustainable throughput with a single computer is therefore 5 queries/s. Obviously, partitioning lists across multiple computers in a cluster is an effective way of scaling-up.

In the following sections we use the above parameters for

crude cost-and-performance models. We will initially assume that (i) data and load are perfectly uniform, (ii) the same length for each index list and (iii) the same frequency of each term in the query workload. This is a big oversimplification, and we will revisit it in Section 6 on load balancing where we will assume heavily skewed data and load.

3. ALEXANDER: RAM-ORIENTED, DOCUMENT-PARTITIONED CLUSTERS

The first architecture we consider mimics more or less what major search engines such as Google do today. We use the name Alexander, after Alexander the Great, the great Greek strategist well known for a number of feats, including the brute-force cutting of the Gordian knot (not a bad method after all, whether real or myth). In technical terms, a good way of characterizing this architecture would be as "embarrassingly scalable"². The key means to achieve great scalability are

1. hash-partitioning the complete index by document ids and spreading the resulting partitions (referred to as "shards" in Google's architecture) across a few hundred computers so that each computer has a relatively small random fraction of a list's postings,
2. keeping all these partitions for all terms in RAM, and
3. replicating such a cluster a number of times for higher throughput and availability.

If we want to keep the entire index in RAM, we need 2,500 computers for one copy of the index (recall: each computer has 4 GB memory, and the total index is 10 TB). If we want to leave some leeway for workspace memory, we can assume that we need 3,000 computers. This is one cluster. Each computer, in principle, holds 1/3,000-th of every index list (we may deviate from this simple scheme by avoiding overly small partitions and using a smarter combinatorial or randomized assignment, but this detail does not really matter for the big picture).

Every query that arrives at such a cluster will now be executed by all 3,000 computers in parallel. According to our query processing model of Section 2, this takes $1 \text{ ms} + 200/3000 \text{ ms} \approx 1 \text{ ms}$ and occupies all computers simultaneously. This gives us a sustained throughput of 1,000 queries/s. To satisfy the peak throughput requirement of 10,000 queries/s, we need 10 such clusters, with a total of 30,000 computers. This also provides sufficient redundancy for fault-tolerance and very high availability. The response time requirement of 100 ms is easily met, assuming that the arriving query load is evenly spread across clusters which results in low utilization and thus only short queuing delays during normal load. During peak load, with 10,000 queries/s, queuing delays would no longer be negligible, but we do not further elaborate on this and rather assume that we stay well below this "maximum" load almost always (or otherwise increase the number of clusters from 10 to say 15). The total cost of the entire system, 10 clusters with 3,000 computers each, is $\$1,000 * 30,000 = \30 Mio . A similar

²In the eighties, the phrase "embarrassingly parallel" was used for algorithms that could be parallelized with perfect speedup in an almost trivial manner. Database scans are one example.

calculation is presented in [3] (with assumed costs of \$3,000 per computer which may include operational costs for the data center and its staff); our calculation is more detailed because we want to perform analogous analyses for other architectures, too.

A strong point of the Alexander architecture, probably its most compelling property, is the perfect load balance. Every computer that participates in executing a query performs the same share of work, by virtue of the hash partitioning on document ids. This property, in combination with the general simplicity of the architecture, allows perfect scalability. If the index size increased by a factor of 10, we would simply increase the number of computers per cluster from 3,000 to 30,000. If the index size stayed invariant but the load increased by a factor of 10, we would increase the number of clusters from 10 to 100. Of course, the cost of the overall system also increases linearly. On the downside, this embarrassing scalability works only for simple data like keyword indexing and simple queries like keyword queries. It is not clear how to make this approach work for sophisticated data like big tables with foreign-key relations or entity-relationship graphs and advanced operations like joins, proximity search, etc. (Google, Yahoo!, etc. would have efficient methods for phrase matching, proximity search, etc. as well, but certainly rely on the fact that advanced queries of this kind constitute only a tiny fraction of the overall workload.)

4. AENEAS: DISK-ORIENTED, TERM-PARTITIONED P2P NETWORK

In terms of scalability, the Alexander design cannot be beaten. But it has significant cost. Our alternative designs therefore aim to provide good scalability at much lower cost (i.e., with fewer computers). To this end, we switch from document partitioning to term partitioning, and we consider ways of replacing RAM by less expensive disk or Flash-RAM storage.

Term partitioning is intriguing because it leads to a data placement that allows serving a single query from only a few computers (one for each keyword in the query), as opposed to using thousands of computers (i.e., a whole cluster) on behalf of every query. So we intentionally reduce intra-query parallelism. This is a well-known recipe for increasing throughput or reducing cost/throughput [10, 17, 15, 24]. The same situation arises in RAID-style storage systems or in parallel database systems, and the standard technique is to allow only coarse-grained parallelism (e.g., by block or track striping as opposed to byte striping [9]). On the other hand, the challenge that we face is potential load imbalances. For now we will still assume uniform load which eliminates the danger of load imbalance by definition. Our design uses hash-partitioning for some degree of balancing, and we will see that the cost savings of term partitioning are so high that we can afford a safety margin of extra computers that would help countering any load imbalances. In addition, we will discuss some smarter algorithmic techniques later in Section 6 on load balancing.

Our first design in the above spirit is called Aeneas, after the smart hero in the Greek and Roman mythology. After the battle of Troy, Aeneas took an uncertain and risky course, but his descendants eventually founded Rome, the seed of another empire. The key algorithmic assets of our

Aeneas system architecture are its use of distributed hash tables (DHTs) (like Chord [25], CAN [23], or P-Grid [1]) and threshold algorithms for top- k queries [11]. DHTs are used to map (the index list of) a term to a DHT node. Hence, the index lists for the terms in the 10 Mio. vocabulary are spread across a certain number of computers. In contrast to standard hashing, DHTs that are based on consistent hashing can gracefully handle failing or otherwise departing computers and also newly joining ones. They also come with means for integrated replication. To save costs, each computer keeps the index lists that it is responsible for on disk rather than in memory. However, we can still utilize RAM for aggressive caching of popular lists.

With 1 TB disks and one disk per computer we would need only 10 computers to hold the entire index. But then our throughput would be too limited as one computer can sustain only 5 queries/s for single-term queries (recall that asynchronous disk I/O and list processing can be pipelined). But 200 computers of this kind could sustain 1,000 queries/s as each query is served by only one computer. Assuming that the typical query consists of 2 keywords and this involves two list-processing jobs on two different computers, we need 400 computers for the normal-load throughput. For peak load, fault-tolerance, and availability, we follow the Alexander design and replicate such a 400-computer cluster 10 times, arriving at 4,000 computers and a total cost of $(\$1,000 + \$500) * 4,000 = 6$ Mio. This is 5 times less than for the Alexander system.

We can do even better by employing a threshold algorithm for efficient top- k query processing. With these algorithms, postings in an index list are sorted in descending score (or "impact") order. The algorithm sequentially scans the index lists for the keywords in the query, aggregates score mass for result candidates, and incorporates a smart test for correct termination without having to read the entire lists. In our work on variants of these algorithms we have performed extensive experimentation with sizable collections (including the TREC Terabyte benchmark [5]). Typically, multikeyword queries need to scan only 10% or less of the index lists to compute the top-10 results (often the scan depth is even much smaller). Consequently, the work per query is only 1/10-th of the full-list processing, namely, $1 \text{ ms} + 200/10 \text{ ms} = 21 \text{ ms}$. This boosts the throughput per computer by a factor of approximately 10, and consequently we need only 40 computers per cluster to sustain 1,000 (2-term) queries/s. This smarter variant of the Aeneas design therefore needs a total of 400 computers for 10 clusters and costs \$ 600,000. This is 50 times less than for the Alexander system.

5. ANAKIN: FLASH-ORIENTED, TERM-PARTITIONED P2P NETWORK

Our third architecture builds on the Aeneas design, but replaces the disk storage by Flash-RAM. This opens up an opportunity for random access to index postings and we will see that it leads to a very attractive system design. We have named this architecture after the Star-Wars hero Anakin Skywalker (aka. Darth Vader, just in case someone does not know). We think this is appropriate as the main driver for Flash-RAM technology is the entertainment industry (cameras, MP3, portable video). Also, Anakin is a Jedi knight who can look a few seconds into the future, and our design speculates about Flash-RAM soon becoming

a major technology for server-oriented systems as well (see also [13, 21, 8]).

If we want to keep the entire index on Flash-RAM storage and each computer can be equipped with 64 GB at a cost of an additional \$800, we need $10 \text{ TB} / 64 \text{ GB} \approx 157$ computers. Flash-RAM allows us to make fast random accesses to postings, which can be utilized by the random-access-oriented variant of the threshold top- k algorithm. Our experiments with these algorithms indicate that random accesses lead to much better pruning of result candidates and much faster termination. Typically, we need to scan only 1% of the index lists to compute the final top-10 if we have the luxury of extensive random lookups. In order to be able to have both fast sequential scans and random lookups, we configure the computers to have both disk and Flash-RAM storage. This way, we can use the high sequential bandwidth offered by the disk without putting load on Flash-RAM, and utilize the Flash-RAM storage exclusively for random lookups.

Thus, the processing time per index list now becomes $1 \text{ ms} + 200/100 \text{ ms} = 3 \text{ ms}$, and we can process 333 single-term queries/s on one computer. So theoretically, to meet the throughput requirements we would need only 3 computers for a normal load of 1,000 single-term queries/s or, actually, 6 computers for 1,000 queries/s with 2 keywords per query. However, we still need 157 computers for storing the complete index, with each computer costing $\$1,000 + \$500 + \$800 = \$2,300$. This builds up to a total cost of $157 * \$2300 = \$361,100$. Once again, we configure the total system with a number of clusters of this kind. Note, though, that in this case we can do with a much smaller number of such clusters, since the replica clusters are needed solely for storage/availability/reliability and not for throughput as was the case for the previous architectures. Hence, assuming 5 clusters, leads to a total cost of $\$361,100 * 5 = \1.8 Mio. This is about 20 times cheaper than Alexander but still more expensive than Aeneas.

Furthermore, alternative Anakin configurations are possible, riding on the coexistence of disk, RAM, and flash-RAM memory technologies. Note that Anakin stores in the flash-RAM of each cluster the complete 10 TB index. Alternatively, we can opt for maintaining a single copy of the complete index in the Flash-RAM of the computers of all clusters. This still provides tolerance against failures since the disks in each computer can be used to store replicas of the complete index. Therefore, Anakin can now be configured with 10 clusters, each with 16 computers, yielding a total of 160 computers, each costing \$2,300. Thus, the total cost builds up to $160 * \$2,300 = \$368,000$. This represents an improvement of a factor of 2 compared to Aeneas.

Of course, this is a very crude calculation, with many underlying assumptions that may not hold in practice. But the potential savings compared to the Alexander architecture are so dramatic that even if many estimates are off by an order of magnitude, we would still arrive at much lower cost than Alexander. The most critical assumption that might render our design calculations questionable if not invalid is the issue of load balancing. We discuss this in the following section.

6. LOAD BALANCING

Two key assumptions made so far is that the query-keywords distribution and the index-list size distribution are

uniform. Obviously, this is not the case in the real world. We now study the three architectures under the prism of skewed keyword popularities and non-uniform index list sizes.

Alexander

Our first conclusion is that Alexander is immune to query-keyword skewness alone. This architecture delivers 1,000 queries/s per cluster, engaging all computers in a cluster and regardless of which keyword is involved, since each cluster stores the complete index and can thus serve any query. Things, however, change when index list size distributions are not uniform. Specifically, the query processing cost for queries involving terms with long index lists changes. Assuming that such long index lists are 10 times longer than the average size (which was set in Section 2 at 200 KB) the list-processing cost for such a long list in a single computer is now 2 s (compared to the previous 200 ms) and thus the list-processing cost in a cluster of 3,000 computers is now $2 \text{ s} / 3,000 \approx 0.67 \text{ ms}$. Therefore, the complete query processing cost now becomes 1.67 ms (1 ms startup cost plus 0.67 ms for list-processing at each computer). With this in mind, a cluster can now serve 598 queries/s. In order to match our peak load of 10,000 queries/s, Alexander now needs roughly 17 clusters, each with 3,000 computers, for a total of 51,000 computers and a total cost of \$51 Mio. Of course, this is an upper bound, since not all queries involve the terms with the longest lists. Correlations between term popularity and index-list sizes, thus, play a key role here. If the most popular terms have small index lists then the cost calculated in Section 3 for Alexander (i.e., \$30 Mio.) would be approximately correct here too. When the most popular terms have the longest index lists, the expected cost would be closer to the \$51 Mio figure.

Aeneas

Aeneas, on the other hand, seems at first sight more vulnerable to skewed query-term distributions. This is due to term partitioning, which stores a term's index list to (the disk of) a single computer capable of serving a mere 5 queries/s. However, recall that Aeneas utilizes clusters of 200 such computers to reach 1,000 queries/s and then employs 10 such clusters to match peak load requirements of 10,000 queries/s. The pivotal observation is that each cluster (of 200 computers) has enough disk space (200 TB) to store 20 copies of the complete index (10 TB). And thus, Aeneas with its 10 clusters can store 200 complete copies of the total index. The key to scalability within Aeneas lies in the intelligent exploitation of this large storage space. Retaining the fundamental properties of Aeneas (disk exploitation, top- k threshold algorithms, and DHT-based term partitioning) there is a large number of intra-cluster and inter-cluster configurations that exploit the available disk space and can meet the challenges of non-uniform term popularities and index list sizes. A detailed discussion of these lies outside the scope of this work. Here we consider only an indicative, extremely challenging scenario for Aeneas.

Let us assume that 1% of the terms (i.e., 10^5 terms) are the most "popular" terms and that these are mentioned (uniformly) in 99% of the queries. Further, assume that these popular terms have the longest index lists (which are in turn 10 times greater than the average index list, i.e., each being 10 MB). To store the index lists for all popular terms requires $10 \text{ MB} * 10^5 = 1 \text{ TB}$ of extra disk space (i.e.,

one extra disk of 1 TB per computer). This bumps up the computer cost in Aeneas to \$2,000.

Now, of the 10,000 queries/s (at peak load), 9,900 involve (uniformly) 9,900 terms (out of the 10^5 popular terms). The query processing cost is dominated by the list processing of the 10 MB index list, which includes 2 Mio. postings, the processing of which takes (1 ms per 1,000 postings and thus) a total of 2 s. Thus, at best each computer can yield 0.5 queries/s. Utilizing top- k threshold algorithms as before, we can cut down the list processing cost to 10%, yielding a query processing cost of about 200ms and improving the per-computer throughput to 5 queries/s. Thus, to reach our target of 10,000 queries/s we need to employ a total of 2,000 computers. To account for two-term queries we would thus need 4,000 computers at worse. These 4,000 computers already have great redundancy for the index lists of the popular and unpopular terms. The total cost for Aeneas then becomes $\$2,000 * 4,000 = \8 Mio , better by a factor of over 6, when compared against Alexander.

Anakin

Again, there are several alternative Anakin configurations that can deal effectively with nonuniform term-popularity and index list size distributions; for example in the spirit of [19]. Here we present a simple one that is indicative of Anakin's potential.

As before, in Anakin each computer has 64 GB of Flash-RAM, and 160 computers are needed to store one copy of the index lists of all terms in Flash-RAM. Note that the operating assumption is that the queries for the popular terms, involve terms uniformly from the set of popular terms. Thus, the 9,900 queries/s that arrive at peak load and involve popular terms, involve with high probability 9,900 distinct popular terms (from the 100,000 popular terms). Therefore, it is possible to forward each such query to a different computer, and in particular the one which happens to store the index list for the query's term in its Flash-RAM. Thus, each computer exploiting the cheap random accesses of Flash-RAM can cut down the disk access cost needing to scan only about 1% of the index list of a term (that is, only 20,000 of the 2,000,000 postings) requiring 1 ms for each 1,000 postings. Thus, the list processing cost is approximately 20 ms, for a total of 21 ms including startup costs. Hence, each computer can serve about 50 queries/s. To reach our peak load of 10,000 queries/s it is enough to employ a total of 200 computers, each costing \$2,300, for a total cost of \$460,000. To handle two-keyword queries, we employ 400 computers with the total cost becoming \$920,000. This represents an improvement of a factor of about 9 compared to Aeneas.

7. INDEX CONSTRUCTION AND MAINTENANCE

For the discussion on index construction and maintenance, we need to distinguish two cases: the document-based partitioning of index lists by Alexander, and the term-based partitioning of index lists by Aeneas and Anakin. The latter two architectures only differ with respect to their local data storage and degree of redundancy, which is a secondary issue as far as index building is concerned. Thus, we focus on the 10-cluster, 40-nodes-per-cluster Aeneas variant as a representative architecture for term-based partitioning.

Regarding the index construction we assume that all docu-

ment are initially conceptually present at some “super-node” (e.g., a subset of the computers in an Aeneas DHT); i.e., we consider the crawling process that downloads Web pages as an orthogonal issue not considered here. We further assume that this super-node has only downloaded the indexable content, but not yet performed the actual extraction of index postings (i.e., *(term, page, score)*-triples). In fact, we want to leverage our computational powers to speed up the indexing process far beyond the rate that the centralized super-node would be able to sustain.

Regarding index maintenance we assume for our calculations that within one year an additional 20 Bio. Web pages need to be indexed; i.e., roughly 50 Mio. documents per day — this includes both previously unseen/new pages as well as updates to existing pages. We further claim that instantaneous index updates are unnecessary; instead we claim that one update per day, e.g., during off-peak hours, meets the requirements of keeping the index up-to-date. (Notwithstanding this claim, certain highly popular and frequently changing pages like news portals could be re-indexed every few hours.)

As parts of both processes will potentially be limited by the available network bandwidth, we assume an intradecenter connectivity with a point-to-point bandwidth of 10 Gbit/s, which – for the sake of simplicity – is assumed to be exclusively utilizable at the full raw bandwidth.

Document-based partitioning

As the hash-partitioning of pages guarantees that (within one cluster) all index postings for one page reside on exactly one peer, it is straightforward for the super-node to assign the duties of indexing accordingly. Naively, in the 10-cluster setting illustrated in Section 3, we need to distribute $5 * 10^7$ pages to 10 nodes each (one target node for each page within each cluster), resulting in a network traffic volume of $10 * 5 * 10^7 * 500 \text{ bytes} = 250 * 10^9 \text{ bytes} = 250 \text{ GB}$. (recall that we assume an average of 500 bytes of indexable content per document). If the network can be fully utilized at its nominal raw bandwidth, this would keep the network busy for $250 \text{ GB} * 1 \text{ GB/s} = 250 \text{ seconds}$. Even if - more realistically - we can effectively utilize the network only at 10 percent of its raw bandwidth, this would take less than an hour. Each node in a 3,000-node cluster will approximately receive 17,000 pages per day, which is small enough so that local indexing time can be ignored. So daily batches of index maintenance incur only a moderate network and processing load on the Alexander data center.

Term-based partitioning

For our architectures based on term-based partitioning, the situation is more complicated, because conceptually each new page has an impact on those (up to) 40 nodes that maintain the index lists for its 100 indexable terms in each cluster. So the whole page would have to be shipped to up to 40 nodes. This would increase the network and processing load for index maintenance by a factor of 40, rendering the Aeneas design critical if not practically infeasible.

Here the key is a smart choice of where the indexing of a page is performed. Our proposed solution is a two-phase strategy that uses the existing DHT infrastructure with its hashing capabilities to partition the pages across nodes. In the first phase we hash-partition new pages by page IDs onto a sufficiently large number of nodes, and it is these

nodes that parse the pages and extract the index postings. The nodes can be chosen from the same network that handles the query workload; they merely need to provide a certain fraction of their memory and CPU capacities for low-priority background work. In the second phase, the nodes that have prepared the postings send them to the nodes that are responsible for the corresponding terms in the term-partitioned design, now using a hash function on term IDs. At the end of the second phase, all new index postings reside at the right target nodes, and need to be merged into the existing lists incurring the same local processing load that the Alexander design needs to sustain.

For an analysis, consider again that the super-node spreads the $5 * 10^7$ documents to 40 nodes of one Aeneas cluster, creating $5 * 10^7 * 500 \text{ bytes} = 25 \text{ GB}$ of network traffic, which consumes 25 seconds if the network could be fully utilized for these transfers in a single burst or 250 s at 10% network utilization or still less than an hour with load throttling that uses only 1% network utilization in the background. Each of these 40 nodes now performs the indexing of 500,000 pages, which is not totally negligible, but feasible. Subsequently, each of the 40 nodes can disseminate its $500,000 \text{ docs} * 100 \text{ terms} * 5 \text{ bytes/entry} = 250 \text{ MB}$ of new index content to the final targets in all 10 clusters. This second phase creates a total network traffic volume of 25 GB, and just like in the Alexander design, the entire dissemination can be done in less than an hour.

Thus, even if we have ignored certain additional aspects (like failures among the 400 nodes that compute the index postings), index maintenance seems practically feasible under realistic constraints in term-partitioned systems like Aeneas and Anakin.

8. CONCLUSION

This paper has aimed to shed new light into the ongoing debate about whether distributed and peer-to-peer indexing are practically viable for ultra-scalable Web search. It has identified various interesting design points, in terms of partitioning strategies and storage technologies, and discussed their strengths and weaknesses. In particular, we have offered simplified but hopefully insightful analyses of the cost/performance ratios for various designs. The Anakin design, based on term partitioning and a combination of disk and Flash-RAM storage, seems to be a particularly intriguing option.

Obviously, the paper’s analyses are not yet detailed enough for final conclusions. But we believe that they are indicative and encourage further research along these lines, in pursuing the quest for fully decentralized, scalable, and highly efficient Web indexing. The anticipated further growth of the Web and the load on search engines suggest that this is not only an academic exercise but will become a pressing issue in coming years.

9. REFERENCES

- [1] K. Aberer. P-grid: A self-organizing access structure for p2p information systems. In *CoopIS*, pages 179–194, 2001.
- [2] V. N. Anh and A. Moffat. Structured index organizations for high-throughput text querying. In *SPIRE*, pages 304–315, 2006.
- [3] R. Baeza-Yates, C. Castillo, F. Junqueira, V. Plachouras, and F. Silvestri. Challenges in

- distributed information retrieval (invited paper). In *ICDE*, 2007.
- [4] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.
- [5] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. IO-Top-k at TREC 2006: Terabyte Track. In E. M. Voorhees and L. P. Buckland, editors, *Proceedings of the 15th Text REtrieval Conference (TREC 2006)*, pages 551–555, Gaithersburg, Maryland, 2006. NIST.
- [6] M. Bender, S. Michel, J. X. Parreira, and T. Crecelius. P2p web search: Make it light, make it fly (demo). In *CIDR*, pages 164–168, 2007.
- [7] M. Bender, S. Michel, P. Triantafillou, G. Weikum, and C. Zimmer. P2P content search: Give the web back to the people. In *IPTPS*, Santa Barbara, US, 2006.
- [8] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high-performance flash disks. *SIGOPS Oper. Syst. Rev.*, 41(2):88–93, 2007.
- [9] P. M. Chen, E. L. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. Raid: High-performance, reliable secondary storage. *ACM Comput. Surv.*, 26(2):145–185, 1994.
- [10] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.
- [11] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.
- [12] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP*, pages 29–43, 2003.
- [13] J. Gray. Tape is dead, disk is tape, flash is disk, ram locality is king. In *CIDR*, 2007.
- [14] J. Li, B. T. Loo, J. M. Hellerstein, M. F. Kaashoek, D. R. Karger, and R. Morris. On the feasibility of peer-to-peer web indexing and search. In *IPTPS*, pages 207–215, 2003.
- [15] J. C. S. Lui, R. R. Muntz, and D. F. Towsley. Computing performance bounds of fork-join parallel programs under a multiprocessing environment. *IEEE Trans. Parallel Distrib. Syst.*, 9(3):295–311, 1998.
- [16] T. Luu, F. Klemm, I. Podnar, M. Rajman, and K. Aberer. Alvis peers: a scalable full-text peer-to-peer retrieval engine. In *P2PIR*, pages 41–48, New York, NY, USA, 2006. ACM Press.
- [17] M. Mehta and D. J. DeWitt. Data placement in shared-nothing parallel database systems. *VLDB J.*, 6(1):53–72, 1997.
- [18] S. Michel, M. Bender, N. Ntarmos, P. Triantafillou, G. Weikum, and C. Zimmer. Discovering and exploiting keyword and attribute-value co-occurrences to improve p2p routing indices. In *CIKM*, pages 172–181, 2006.
- [19] S. Michel, P. Triantafillou, and G. Weikum. Minerva_{infinity}: A scalable efficient peer-to-peer search engine. In *Middleware*, pages 60–81, 2005.
- [20] A. Moffat, W. Webber, and J. Zobel. Load balancing for term-distributed parallel retrieval. In *SIGIR*, pages 348–355, 2006.
- [21] S. Nath and A. Kansal. Flashdb: dynamic self-tuning database for nand flash. In *IPSN*, pages 410–419, 2007.
- [22] I. Podnar, M. Rajman, T. Luu, F. Klemm, and K. Aberer. Scalable Peer-to-Peer Web Retrieval with Highly Discriminative Keys. In *ICDE*, 2007.
- [23] S. Ratnasamy, P. Francis, M. Handley, R. M. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.
- [24] P. Scheuermann, G. Weikum, and P. Zabback. Data partitioning and load balancing in parallel disk systems. *VLDB J.*, 7(1):48–66, 1998.
- [25] I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.
- [26] T. Suel, C. Mathur, J. wen Wu, J. Zhang, A. Delis, M. Kharrazi, X. Long, and K. Shanmugasundaram. Odissea: A peer-to-peer architecture for scalable web search and information retrieval. In *WebDB*, pages 67–72, 2003.

Taming Hot-Spots in DHT Inverted Indexes

Nuno Lopes*
CCTC-Department of Informatics
University of Minho
Braga, Portugal
nuno.lopes@di.uminho.pt

Carlos Baquero
CCTC-Department of Informatics
University of Minho
Braga, Portugal
cbm@di.uminho.pt

ABSTRACT

DHT systems are structured overlay networks capable of using P2P resources as a scalable platform for very large data storage applications. However, their efficiency expects a level of uniformity in the association of data to index keys that is often not present in inverted indexes. Index data tends to follow non-uniform distributions, often power law distributions, creating intense local storage hotspots and network bottlenecks on specific hosts. Current techniques like caching cannot, alone, cope with this issue.

We propose a new distributed data structure based on a decentralized balanced tree to balance storage data and network load more uniformly across all hosts. The approach is stackable with standard DHTs and ensures that the DHT storage subsystem receives an uniform load by assigning fixed sized, or low variance, blocks.

1. INTRODUCTION

Distributed Hash Tables (DHTs) are structured overlay networks capable of efficiently storing and locating objects from a given key. Systems like Chord, Pastry and CAN [20, 18, 16] allow scalability in the number of hosts, requiring only logarithmic communication steps and routing state. A hash function is used to uniformly distribute keys to hosts so that key load is balanced.

This perfect distribution has two intrinsic assumptions: Keys are uniformly accessed, both in storage and retrieval; The size of the tuples $\langle key, object \rangle$ depict a low variance. However, these assumptions are often not possible. This is the case when building term-partitioned inverted indexes over DHTs [13, 17, 22], where words are mapped to the locations of the documents where they occur.

Hot spots created by data or query asymmetries will occur due to the power-law distribution of text keyword frequency [25]. When a single key is accessed very often (e.g. “Katrina”), a network bottleneck appears on the host storing that key. This situation known as “query flash crowd”

*Supported by a Ph.D. Scholarship from FCT - Foundation of Science and Technology, the Portuguese Research Agency.

can be minimized with caching schemes [19, 14]. On the other hand, storage hotspots occur when very large objects of skewed size are stored on individual DHT keys (e.g. the occurrence set for the word “the”). Although storage is often not a critical resource, due to the current trend on secondary storage capacity, storing such large objects creates an additional network bottleneck on the hosts mapping these keys. These network bottlenecks limit the scalability of term-partitioned indexes [1] and cannot be eliminated by caching, as caching is effective only when reading data and not when new data is being inserted into the system. Furthermore, solutions that dynamically redistribute keys across hosts [10, 15] are also unable to eliminate the storage hotspots because the storage unbalance is due to a single key containing a very large object.

In this paper we propose a solution for load balancing DHTs when storing (decomposable) objects with high size variance. We developed a new DEcentralized Balanced tree (DEB) tree algorithm capable of converting a very large object into multiple bounded size blocks suited for being stored and searched as objects over DHTs. We used the DEB algorithm to build a textual inverted index, allowing multiple keys retrieval. The system evaluation shows expressive improvements in the storage and network load distribution, for both index population operations and data retrieval.

Our paper is organized as follows: Section 2 shows an overview of the system interfaces, Section 3 describes the DEB tree algorithm and Section 4 the text indexing system. Section 5 shows our evaluation results, and Section 6 presents the related work and is followed by the Conclusions.

2. SYSTEM OVERVIEW

The system provides an inverted index interface to user applications and stores the index on a structured P2P overlay that is implemented by a DHT algorithm. The system architecture, depicted in Figure 1, is composed by the index layer that presents an inverted index interface to client applications, the tree management layer that implements our distributed balanced tree algorithm and the routing layer (the DHT algorithm) responsible for routing messages between hosts. The index layer receives requests from client applications and converts them into tree based operations to be executed by the tree layer. In turn, the tree layer uses the routing layer to locate the tree block that should process the operation. Tree blocks are stored on the P2P sub-system.

2.1 Index Model

Copyright is held by author/owners.
ACM SIGIR Workshop on Large Scale Distributed Systems for Information Retrieval '07, Amsterdam, The Netherlands.

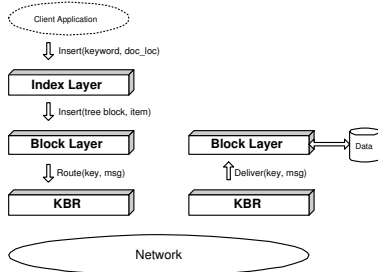
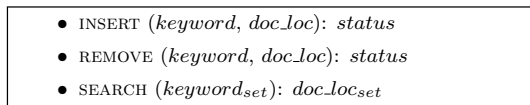
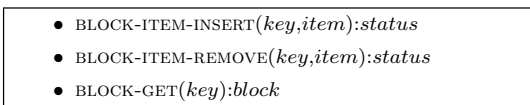


Figure 1: The system is built with a base DHT overlay network for managing hosts membership in a scalable way. Each host of the system contains three components: a key-based routing (KBR) layer, the block storage module and the client index interface.



(a) Index Layer Interface



(b) Block Layer Interface

Figure 2: Layered Interface.

A textual inverted index stores relations between text words (the vocabulary) and sets of document locations (the occurrences) [2] in the form:

$$keyword \mapsto \{document\ location\}_{SET}.$$

Since a single keyword can occur on multiple documents, we store a set of document locations for each keyword. Document locations are just single opaque objects capable of locating a document over the system. The pair $\langle host_address, docId_{local} \rangle$ is an example of a simple location scheme. Other location schemes could be used, like an URL link or the document content hash value, provided the retrieval of the document is possible from the location value [13, 21].

The index is made accessible to system peers through the interface on Figure 2(a). User applications, in any given node, contact the local index library through this interface. The index INSERT operation adds a new relation between a keyword and a document. Likewise, REMOVE cancels an association. The index search operation retrieves the list of documents associated with a keyword or a set of keywords.

We only considered the *and* Boolean operator for multiple keyword queries, although the remaining Boolean operators could also be implemented [2].

2.2 DHT Interface

The DEB Tree implementation uses a custom DHT interface that substitutes the typical GET/PUT interface. These operations, shown in Figure 2(b) allow a fine grain manipulation of the data object associated to a given key. Here the object has a Set structure and the operations allow the insertion and removal of individual items. Otherwise, if the usual GET/PUT operations were used, latency would double and consistency problems could arise due to lack of atomicity in GET,PUT sequences [4].

The actual custom DHT interface is slightly richer, in order to support other needed operations which are best performed on the node that hosts the block. Their implementation does not present additional difficulties once a Key Based Routing interface is available, ROUTE($key, message$), which is the case for all DHT implementations [6].

3. DEB TREE LAYER

We will now describe our DEB tree implementation. This tree algorithm was based on the B^+ -tree design [5] and shares the high-availability requirements present on B-link trees [9]. However, unlike the B-link tree algorithm which was designed for a cluster based architecture with global system view and centralized environment, our algorithm was designed for being deployed on wide-area systems requiring neither global knowledge nor centralized entities.

The DEB-tree algorithm supports a mapping interface for $\langle key, value \rangle$ tuples, just like the B^+ -tree, storing document locations (the key) and opaque payloads (the value). In this paper only document locations were used, hence the absence of a value payload on the block insert operation (see Figure 2(b)). Each DEB tree instance stores the document location set for a particular keyword. Therefore, each index keyword will have an unique DEB tree.

3.1 Tree Structure

The tree structure, just like in the B^+ -tree design, is composed by a root block and child blocks, the last level of blocks (further away from the root) are called leaf blocks. Leaf blocks store data items and are all at the same tree level (any leaf is accessed from the root block with the same number of block hops). Internal blocks serve exclusively for locating leaf blocks and do not contain any data, instead they contain child block keys.

All blocks contain a parent's field with the key to the upper level block. To improve availability, each block also stores the key to the next sibling block, following the *B-link* design.

The number of items in any block is bounded by the tree's degree t which defines the minimum $(t - 1)$ and maximum $(2t - 1)$ number of elements allowed inside a block [5]. For internal blocks the degree influences the number of child block keys it contains. For leaf blocks it influences the number of document locations the block stores.

In addition to the previous fields, each block contains the minimum and maximum limits, representing the interval of data, in terms of key ids, the block is responsible for. The root block has the whole key interval, covering all data. The sum of intervals at each tree level also covers the whole interval, and in a given level all intervals are disjoint.

3.2 Block Identification Scheme

Each tree block is identified by an unique key and stored

on the DHT using the hash value of its key. Since we store all tree blocks under the same name space, the DHT hash domain, we must ensure all blocks will have a unique identification.

Decentralized generation of (probabilistic) unique block ids is made by hashing a globally unique triplet that consists of $\langle keyword, level, minlimit \rangle$. Notice that each stored *keyword* (See index interface) gives rise to a distinct tree, and that all blocks in that tree will depict a distinct level number and minimum key limit on their local key range.

3.3 Tree Layer Algorithms

When client applications, in a node, request operations at the index interface the respective algorithms are executed and, possibly, tree management algorithms are triggered. These last algorithms are responsible for splitting or merging blocks.

Index interface operations are decomposed into one or more DHT block operations. These operations are issued from the node hosting the client application. On the contrary, tree management operations are triggered and issued in the node hosting the block that needs splitting or merging.

All these algorithms are made tolerant to concurrency issues on what concerns structural integrity of the tree and the stored data. To achieve this, some item insert operations may be delayed. Notice that inserts can even be made to timeout and be repeated, since the insert operation is idempotent.

An index SEARCH operation that covers data modified by a concurrent INSERT may, or may not, see the effect of the insert. However, once an insert completes on a client host, subsequent searches in that host or in any other host will see the insertion if the data is covered by the search.

This is achieved even in the presence of caching, since the algorithms only cache internal blocks, and these only contain references to other blocks. Stale information only enacts a performance penalty.

3.4 Data Resilience and Structural Repair

Basic resilience to host failures and churn should be provided by the underlying DHT algorithm. In particular it is desirable to use DHT solutions, like [14], that replicate more intensively blocks that are subject to more activity.

However, even in the event of a fault that is not masked by the DHT layer it is possible to recover the structural integrity of the tree by making use of the redundancy in the structure. The loss of an internal block does not remove relevant data from the tree. Faults at the leaves, however, lead to lost association between keys and locations. The lost data in this case can only be recovered by re-announcing at the clients.

4. INVERTED INDEX OPERATIONS

The index operations amount to inserting references and searching for keywords. These operations are available at the client's host and issue multiple block requests through the DHT to accomplish the initial index operation.

4.1 Document Insertion

For indexing a document into the system, peer clients use the INSERT $\langle keyword, doc.location \rangle$ function, which adds a document location to a keyword occurrence set. Since the

```

procedure insert (keyword, doc-location):
1: blk-key  $\leftarrow$  getRootBlockKey (word)
2: route (blk-key,  $\langle$  insert, doc-location  $\rangle$ )
3: answer  $\leftarrow$  wait for returning message
4: while answer  $\neq$  'ack':
5:     blk-key  $\leftarrow$  get forward block from answer
6:     route (blk-key,  $\langle$  insert, item  $\rangle$ )
7:     answer  $\leftarrow$  wait for returning message
8: end while

```

(a) Client side index insertion

```

procedure insert-block (block, item):
1: if leaf(block):
2:     insert(block, item)
3:     ret message  $\langle$ ack  $\rangle$ 
4: else:
5:     ret message  $\langle$ forward, successor(block, item)  $\rangle$ 
6: endif

```

(b) Block side index insertion

Figure 3: Simple pseudo-code for the index insertion procedure.

occurrence set is stored on a DEB tree instance, one tree per keyword, this is to say the document location will be inserted into the corresponding DEB tree. The client must call the INSERT function for every $\langle keyword, doc.location \rangle$ pair it wishes to index.

Tree insertion is made first by locating the block responsible for storing the item and then by inserting it on the block's data. If the tree only contains a single block, the root block, then the operation finishes after accessing this block. For bigger trees, the client starts at the root block and follows child block references until reaching the correct leaf block. The operation terminates after receiving the acknowledgment of the insertion from the leaf. Figure 3(a) shows simple pseudo-code for the insertion operation on the client index side. Removing an index occurrence works in the same way as for the insertion case.

Inserting a document location on a keyword occurrence set requires the insertion of an item on the keyword set tree. Inserting an item on a B-Tree with I items uses $O(\log I)$ block accesses, which corresponds to the tree height [5]. Each block access is made by the client host using the ROUTE function supplied by the DHT algorithm, which in turn uses $O(f(N))$ messages to locate the host for a key, having f as the lookup cost function on the DHT (typically the logarithmic function). The number of messages used to insert an index occurrence on a system will grow $O(\log I \cdot f(N))$ for N hosts. Since some DHT implementations only require $O(1)$ steps to locate a key [8, 14], this results in $O(\log I)$ complexity. Common DHT algorithms provide a logarithmic cost and therefore will use $O(\log I \cdot \log N)$ messages.

If a client's local cache is used for caching top level tree blocks, the client can access the target leaf block directly for insertion, reducing the complexity even further to $O(1 \cdot f(N))$ for inserting a single reference onto the index.

4.2 Multiple Keyword Search

Queries in this index system follow a multiple keyword intersection model, using the *and* Boolean Query operator for returning the set of document locations that are common to all the query keywords. To perform this intersection, the client would need to fetch all the occurrence sets and then perform a local intersection on the fetched data to determine the final result set.

This simple solution is clearly not optimal. Fetching a complete large occurrence set uses network bandwidth to retrieve data that may not be necessary to effectively answer the query. Remember that each keyword occurrence set is stored under a different DEB tree instance.

We opted for an incremental intersection evaluation that makes a parallel breadth-first traversal of all the trees simultaneously. The use of an incremental evaluation enables the use of two optimization techniques to considerably reduce the query network bandwidth: early-pruning and term re-ordering. The early-pruning heuristic was inspired by the adaptive set intersection algorithm suggested by Li et al. [11] to minimize data exchange when evaluating set intersections. The heuristic prunes tree sub-branches according to the rule that intersecting an empty set with any set will always be empty. By selecting the branches of large trees to visit according to items already found on smaller trees, and pruning the remaining branches, the heuristic reduces the number of visited blocks without affecting the operation's correction.

Term re-ordering is a database optimization that consists in accessing the intersection sets in order from the smallest to the largest so that the amount of exchanged data is reduced to the minimum. The re-ordering was implemented by accessing first the root blocks of smaller trees. The size of each keyword set was locally determined by the tree's height, which is available at the root blocks.

A single keyword query retrieval requires a tree vertical traversal until reaching the leaf level and then a linear leaf block traversal. Just like in the insertion case, a vertical traversal uses $O(\log I)$ block accesses and the leaf level grows with the number of items in the set $O(I)$, so that the total cost of the operation will be $O(I)$ on the number of items. Since each block access requires $O(f(N))$ host messages, the total number of messages on the system will be $O(f(N) \cdot I)$. Assuming an $O(1)$ hop DHT is used, a single keyword retrieval requires $O(I)$ messages.

The number of stored items I follows a power-law distribution on the number of documents. The value of I depends therefore on the popularity of the keyword. For a few very popular keywords (in storage frequency), I grows linearly with the number of documents. For the remaining keywords it tends to be constant towards the number of documents.

The previous complexity limit assumes that a multi-keyword query uses multiple independent single keyword retrievals. The search optimizations described previously reduce the total number of retrieved items significantly, bringing the query cost, in terms of number of retrieved items, closer to the number of common items instead of the total number of stored items for all keywords in the query.

5. SYSTEM EVALUATION

We will now evaluate the DEB tree index on a textual document collection. Document reference insertion and key-

word search will be tested. The evaluation focuses on the load balance properties of the solution when compared to the equivalent linear DHT mapping, considering both storage and network resources.

5.1 Setup

Our DEB tree implementation was deployed on a custom made discrete event simulator implementing a simplified SSF framework written in Python. The communication between hosts and DHT message routing, were simulated. Although our experiments ran over this network simulated environment, the algorithm implementation is made of actual code and could be placed on top of a real DHT system for deployment. In this article, we opted for the simulation model to test the algorithm under a controlled environment with a larger number of hosts.

5.2 Index Insertion

The simulation of the insertion procedure consisted in 1000 hosts concurrently inserting word references, in documents, into the distributed index. Each host was given 10 unique documents to insert. Each document was a newspaper article, with an average size of 3Kb. As expected, this dataset depicts a Zipf distribution, with a few high rank words present in most of the documents.

We evaluated the algorithm performance by varying the tree's block size value. We used a very large block size (shown as $+\infty$) to represent the case of a direct mapping of the index on the DHT [13, 7, 17]. This infinite block will never be full and consequently never split, creating exclusively single root block trees. The other sizes represent the block maximum size for each simulation.

We will now look at the effective load each host received. We assumed a perfect mapping between blocks and hosts. First, we calculated the hash value of the block's key. Then, we assigned a host to the block giving it's hash value. Each host received an equal size share of the hash domain. This allocation overcomes the absence of an actual DHT substrate, but the incurred simplification matches the behavior of an actual DHT with a reasonable number of hosts. It is asymptotically correct.

Figure 4 shows how the storage load distributes, with different block sizes. The infinite block size case shows the less uniform distribution. On the other side, the smaller block sizes show an almost perfect load distribution. However, very small sizes tend to over-split trees, creating excessive internal blocks and increasing the total load on the system. This is the case of the tree with block size 4. A block size of 32 shows an adequate trade-off, and is appropriate to ensure a mostly uniform storage load distribution from an highly skewed dataset.

It is also relevant to analyze how the network load is distributed during insertions. We can expect that root blocks of popular words will have a high demand. The algorithm uses block caches to control this demand.

Figure 5 shows that without caching, smaller blocks have a negative impact as they lead to larger trees and more split operations, increasing the overall message load. This situation would not be adequate.

The same figure shows the results of the same insertion procedure with client cache enabled on hosts. As expected, the variation between the minimum and maximum loaded hosts has decreased significantly for any block size. The

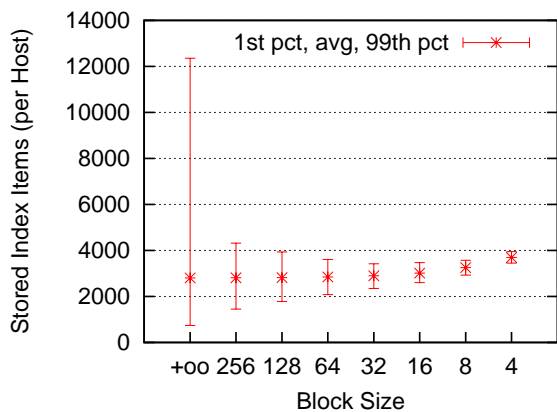


Figure 4: Storage load when inserting. Showing averages, 1st and 99th percentiles.

Block size	+oo	256	128	64
No cache	2799	3947	4270	4793
Cache	2799	3264	3503	3784
Block size	32	16	8	4
No cache	5487	6588	8312	11384
Cache	4183	4864	5972	7955

Figure 6: Average Insertion Messages Received per Host

simulation with the very large block size, containing only single root block trees, is identical to the previous simulation without cache because cache only acts on internal blocks. As block sizes get smaller, caching reduces the extra load caused by accessing the top level blocks on larger trees. As a consequence the network load is better distributed across hosts. A block size of 32 would also be an adequate choice, on what concerns data insertion. Table 6 summarizes the average number of insertion messages received per host for different block sizes, with and without cache, found in Figure 5.

5.3 Index Searching

The index searching procedure starts from a fully loaded index. A single host processes a list of multiword search queries. We generated 20000 queries with a multiple keyword distribution from the original text collection. Search keywords follow a Zipf distribution but the rank order was randomized, so that popular keywords are distinct in searches and insertions. The Zipf distribution for generating individual search terms does not consider the correlation between keywords found in real multi-term query traces. We plan as future work to run the same evaluation of the algorithm using a corpus with real query logs.

We measured the index search load by counting the number of index items replied to the caller. Index items correspond to document locations when accessing leaf blocks and correspond to children block limits when accessing internal blocks. We assumed that document locations (and block limits) have a constant or small variation size in bytes.

We will first show the impact of our query optimizations

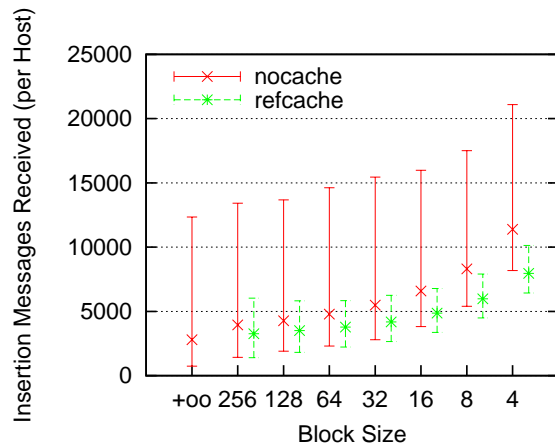


Figure 5: Message load when inserting. Showing averages, 1st and 99th percentiles.

on the system bandwidth. Figure 7 shows the cumulative distribution function (CDF) of the number of block requests (messages) received at hosts, according to the query optimizations used for a block size of 32 items. The worst result appears on the basic incremental method (label `inc`), which traverses all trees in a breadth-first order. It is followed by the early-pruning method (label `early`) which improves the basic incremental method by stopping the retrieval of further blocks that cannot contribute to the final result set. We improve further by adding a keyword term reordering (label `sort-loc`) that starts by accessing smaller trees first and leaving larger trees to the end. This term reordering works in conjunction with early-pruning to interrupt block retrieval as soon as the final result set can be computed.

The term reordering method was originally developed for local knowledge, so we also simulated a variation (label `sort-glob`) that supplied the client with the global index keyword frequency. This experiment allowed us to determine the maximum possible gain from using this heuristic, although it cannot be used in real systems.

We implemented the reference cache procedure over the local term re-ordering heuristic (label `sort-loc-cache`). When comparing it to the same query method without cache in Fig. 7 (label `sort-loc`), one observes that although cache reduced the overall load, it was only marginally. This performance can be explained by noticing that cache was operating only on internal blocks, having no effect on leaf accesses. Since leafs are not cached, the hosts storing leaf block data for popular keywords are overloaded with requests and hence the high number of messages received at some hosts. A query “flash crowd” on leaf blocks could be handled directly by the DHT layer [14].

Having established that the best usable heuristic is the one depicted by `sort-loc-cache`, we now proceed to analyze the impact of different block sizes. Figure 8 shows, in log scale, for various blocks sizes, the number of items that are sent by hosts in response to the generated query load.

Although smaller block sizes will lead to more messages, on what concerns the number of items sent (in a sense, the bandwidth) smaller blocks are better. Because the block size is smaller, each block request causes a smaller impact on the network load of each host (the quantity of data items

