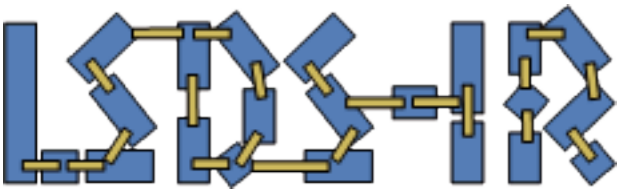
Claudio Lucchese Gleb Skobeltsyn Wai Gen Yee (Eds.)



LSDS-IR'09

7^{th} Workshop on Large-Scale Distributed Systems for Information Retrieval

Workshop co-located with ACM SIGIR 2009

Boston, MA, USA, July 23, 2009

Proceedings

Editors' addresses:

Claudio Lucchese
Istituto di Scienza e Tecnologie dell'Informazione
Consiglio Nazionale delle Ricerche
Area della Ricerca di Pisa
Via G. Moruzzi, 1
56124 Pisa (PI)
Italy
claudio.lucchese@isti.cnr.it

Gleb Skobeltsyn
Google Inc.

Brandschenkestrasse 110
8002 Zurich
Switzerland
glebs@google.com

Wai Gen Yee
Department of Computer Science
Illinois Institute of Technology

10 W. 31^{st} Street
Chicago, IL 60616
USA
yee@iit.edu

Preface

The Web is continuously growing. Currently, it contains more than 20 billion pages (some sources suggest more than 100 billion), compared with fewer than 1 billion in 1998. Traditionally, Web-scale search engines employ large and highly replicated systems, operating on computer clusters in one or few data centers. Coping with the increasing number of user requests and indexable pages requires adding more resources. However, data centers cannot grow indefinitely. Scalability problems in Information Retrieval (IR) have to be addressed in the near future, and new distributed applications are likely to drive the way in which people use the Web. Distributed IR is the point in which these two directions converge.

The 7^{th} Large-Scale Distributed Systems Workshop (LSDS-IR'09), co-located with the 2009 ACM SIGIR Conference, provides a space for researchers to discuss these problems and to define new research directions in the area. It brings together researchers from the domains of IR and Databases, working on distributed and peer-to-peer (P2P) information systems to foster closer collaboration that could have a large impact on the future of distributed and P2P IR.

The LSDS-IR'09 Workshop continues the efforts from previous workshops held in conjunction with leading conferences:

- CIKM'08: Workshop on Large-Scale Distributed Systems for Information Retrieval - LSDS-IR'08
- SIGIR'07: Workshop on Large-Scale Distributed Systems for Information Retrieval - LSDS-IR'07
- CIKM'06: Workshop on Information Retrieval in Peer-to-Peer Networks - P2PIR'06
- CIKM'05: Workshop on Information Retrieval in Peer-to-Peer Networks - P2PIR'05
- SIGIR'05: Workshop on Heterogeneous and Distributed Information Retrieval - HDIR'05
- SIGIR'04: Workshop on Information Retrieval in Peer-to-Peer Networks - P2PIR'04

This year's program features two keynotes, six full and three short papers covering a wide range of topics related to Large Scale Distributed Systems.

The first keynote speaker, Leonidas Kontothanassis (Google Inc.), discusses access patterns and trends in video information systems pertaining to YouTube. The second keynote speaker, Dennis Fetterly (Microsoft Research), talks about experiences with using the DryadLINQ system, a programming environment for writing high performance parallel applications on PC clusters, for Information Retrieval experiments.

As in the past years, the workshop features several papers on P2P-IR. Bockthing et al. present an approach for collection selection based on indexing of popular term combinations. Esuli suggests using permutation prefixes for similarity search. Ke et al. study the clustering paradox for decentralized search. Finally, Dazzi et al. discuss the clustering of users in a P2P network for sharing browsing interests.

Another cluster of papers deals with efficiency of large scale infrastructures for information retrieval. Nguyen proposes a new static index pruning approach. Capannini et al. propose a sorting algorithm that uses CUDA. McCreadie et al. discuss how suitable the MapReduce paradigm is for efficient indexing. Lin addresses the load-balancing issue with MapReduce. Finally, on the search quality side, Yee et al. study the benefits of using user comments for improving search in social Web sites.

We thank the authors for their submissions and the program committee for their hard work.

July, 2009 Claudio Lucchese, Gleb Skobeltsyn, Wai Gen Yee

Workshop Chairs

Claudio Lucchese, National Research Council - ISTI, Italy
Gleb Skobeltsyn, EPFL & Google, Switzerland
Wai Gen Yee, Illinois Institute of Technology, Chicago, USA

Steering Committee

Flavio Junqueira, Yahoo! Research Barcelona, Spain
Fabrizio Silvestri, ISTI-CNR, Italy
Ivana Podnar Zarko, University of Zagreb, Croatia

Program Committee

Karl Aberer, EPFL, Switzerland
Ricardo Baeza-Yates, Yahoo! Research Barcelona, Spain
Gregory Buehrer, Microsoft Live Labs, USA
Roi Blanco, University of A Coruna, Spain
Fabrizio Falchi, ISTI-CNR, Italy
Ophir Frieder, Illinois Institute of Technology, Chicago, USA
Flavio Junqueira, Yahoo! Research Barcelona, Spain
Claudio Lucchese, ISTI-CNR, Italy
Sebastian Michel, EPFL, Switzerland
Wolfgang Nejdl, University of Hannover, Germany
Kjetil Norvag, Norwegian University of Science and Technology, Norway
Salvatore Orlando, University of Venice, Italy
Josiane Xavier Parreira, Max-Planck-Institut Informatik, Germany
Raffaele Perego, ISTI-CNR, Italy
Diego Puppin, Google, USA
Martin Rajman, EPFL, Switzerland
Fabrizio Silvestri, ISTI-CNR, Italy
Gleb Skobeltsyn, EPFL & Google, Switzerland
Torsten Suel, Polytechnic University, USA
Christos Tryfonopoulos, Max-Planck-Institut Informatik, Germany
Wai Gen Yee, Illinois Institute of Technology, USA
Ivana Podnar Žarko, University of Zagreb, Croatia
Pavel Zezula, Masaryk University of Brno, Czech Republic

Contents

The YouTube Video Delivery System

Leonidas Kontothanassis
Google Inc.

This talk will cover the YouTube Video Delivery System. It will discuss access patterns and trends for both video uploads and downloads. It will describe the storage and delivery mechanisms for popular and unpopular content and the impact YouTube has on the network storage infrastructure for Google. We will also discuss the networking impact for ISPs around the world.

Leonidas Kontothanassis joined Google in 2006 and immediately started working on networking and video delivery issues and have been ever since. He currently acts as the manager of the teams working in these areas. Previously he has worked in such areas as computer architecture, parallel programming, and content delivery with multiple companies in the Kendall/MIT area include DEC/HP/Intel Labs and Akamai. He received a PhD in computer architecture in 1996 and has served as committee member or organizer for academic conferences and research funding organizations like NSF.

DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language

Dennis Fetterly
Microsoft Research

The goal of DryadLINQ is to make distributed computing on large compute clusters simple. DryadLINQ combines two important pieces of technology: the Dryad distributed execution engine and the .NET Language INtegrated Query (LINQ). Dryad provides reliable, distributed computing on thousands of applications in a SQL-like query language, relying on the entire -NET library and using Visual Studio. DryadLINQ is a simple, powerful, and elegant programming environment for writing large-scale data parallel applications running on large PC clusters. This talk will also describe the experience using DryadLINQ for a series of information retrieval experiments.

Dennis Fetterly is a Research Software Development Engineer in Microsoft Research's Silicon Valley lab, which he joined in May, 2003. His research interests include a wide variety of topics including web crawling, the evolution and similarity of pages on the web, identifying spam web pages, and large scale distributed systems. He is currently working on DryadLINQ, TidyFS, and a project evaluating policies for corpus selection. Interesting past projects include the MSRBot web crawler, Dryad, the Your Desktop and Your Keychain project, which utilizes flash memory devices to enable users to carry their desktop PC state with them from machine to machine, and PageTurner, a large scale study of the evolution of web-pages.

Collection Selection with Highly Discriminative Keys

Sander Bockting
Avanade Netherlands B.V.
Versterkerstraat 6
1322 AP, Almere, Netherlands
sander.bockting@avanade.com

Djoerd Hiemstra
University of Twente
P.O. Box 217
7500 AE, Enschede, Netherlands
d.hiemstra@utwente.nl

ABSTRACT

The centralized web search paradigm introduces several problems, such as large data traffic requirements for crawling, index freshness problems and problems to index everything. In this study, we look at collection selection using highly discriminative keys and query-driven indexing as part of a distributed web search system. The approach is evaluated on different splits of the TREC WT10g corpus. Experimental results show that the approach outperforms a Dirichlet smoothing language modeling approach for collection selection, if we assume that web servers index their local content.

1. INTRODUCTION

The web search approach of major search engines, like Google, Yahoo! and Bing, amounts to crawling, indexing and searching. We call this approach *centralized search*, because all operations are controlled by the search engines themselves, be it from a relatively limited number of locations on large clusters of thousands of machines. The centralized web search paradigm poses several problems.

The amount of web data is estimated to grow exponentially [34]. The changing and growing data requires frequent visits by crawlers, just to keep the index fresh. Crawling should be done often, but generates a huge amount of traffic, making it impossible to do frequent crawls of all pages. With an estimated four weeks update interval, updates are performed relatively slow [25, 35]. Also, it is impossible to index everything, as the search engine accessible *visible* web is only a fraction of the total number of web pages [16].

Callan [5] identified the distributed information retrieval problem set, consisting of resource description, resource selection and result merging. We believe that distributing the search efforts may be an approach to solve the problems described above. This research focuses on resource description and resource selection [10, 22]. Resource description, i.e. indexing of the peers, is distributed; resource selection, or *collection selection*, is centralized in our research.

LSDS-IR Workshop. July 2009. Boston, USA.

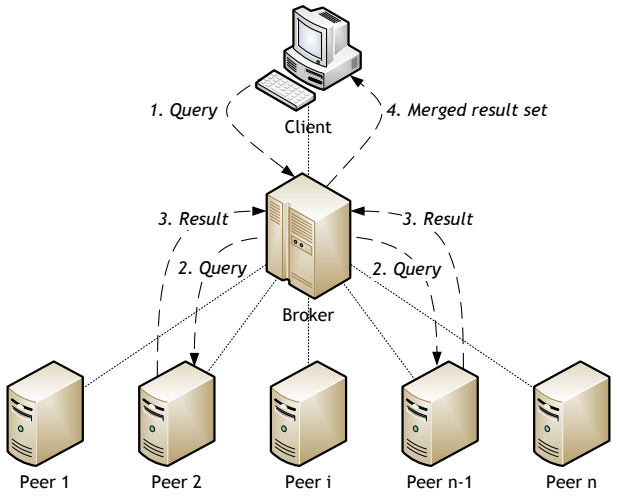


Figure 1: Peers are accessible via a broker to answer queries from clients

Figure 1 shows three types of entities: peers, brokers and clients. We assume that peers are collaborative. Every peer runs a search engine enabled web server. The search engine indexes the local website(s), but it may also index other websites. In this scenario, there can be many millions of peers. When a user has an information need, he can pose a query at the client. The client sends the query to the broker. In a response, the broker tries to identify the most promising peers to answer the query. This has to be a small amount of peers, e.g. five to ten peers, so not a lot of traffic is generated. The query is routed to those peers and the results are returned to the broker. The broker merges the results and sends the merged list to the client.

Peers and brokers cooperate to enable brokers to identify most promising peers. Therefore, every peer sends a small part of its index to the broker. This part cannot be too small, to still allow for proper judging about the peers' ability to satisfactory answer queries. The index part cannot be too large due to index data traffic scalability.

Techniques have been proposed to manage the index size. Podnar *et al* [20] used the concept of highly discriminative keys (HDKs) for document retrieval in distributed information retrieval. An HDK is a set of terms that is highly discriminative, i.e., that only match a few documents in the collection. Because the terms are pre-coordinated (they are combined at index time, not at search time) and because

only a few document match all terms in a pre-coordinated set, the HDK approach is able to very efficiently retrieve the top documents for a query. Although searching can be efficient, the HDK indexing process, described in more detail in Section 3.1, has the negative side-effect that a huge amount of highly discriminative keys are generated. To reduce the number of keys, Skobeltsyn [32] proposed a query-driven indexing strategy that uses caching techniques to adapt to the changing querying behavior of users. The combination of HDK with query-driven indexing allows for completely distributed document retrieval that in theory grows to web scale proportions [31].

This paper contributes to the field of distributed information retrieval by the applying HDKs and query-driven indexing to select collections, instead of documents. Such an approach would in theory scale the distributed search scenario described above to millions of peers: The broker lists for every HDK a small number of peers to send the query to, and the peers retrieve the documents; possibly many. Unlike a traditional inverted file index that typically consists of huge posting lists and a, in comparison, tiny dictionary [36], our HDK index consists of a huge dictionary and, in comparison, tiny posting lists. The system is fitted into the previously sketched scenario, which allows for control at the broker. This control can for example be used to prevent misuse or to allow for domain-specific search.

This paper is organized as follows: the next section discusses earlier collection selection methods, Section 3 introduces our collection selection system and Section 4 describes the evaluation. The paper concludes with results and conclusions.

2. EARLIER WORK ON COLLECTION SELECTION

Collection selection systems have been developed to select collections containing documents that are relevant to a user's query. The generalized Glossary-Of-Servers Server (gGlOSS) is such a system [13]. It uses a vector space model representing index items (document collections) and user queries as weight vectors in a high dimensional Euclidean space to calculate the distance (or similarity) between document collections and queries [24].

Another well-known approach is CVV, which exploits the variation in cue validity to select collections [38]. The *cue validity* $CV_{i,j}$ of query term t_j for collection c_i measures the extent that t_j discriminates c_i from the other collections, by comparing the ratio of documents in c_i containing t_j to the ratios of documents in other collections containing t_j. The larger the variation in cue validities for collections with respect to a term, the better the term is for selecting collections.

This section will describe two collection selection methods in more detail: inference networks and language modeling.

2.1 Inference networks

CORI [7] is a collection ranking algorithm for the INQUERY retrieval system [6], and uses an inference network to rank collections. A simple document inference network has leafs d representing the document collections. The terms that occur in those collections are represented by representation nodes r. Flowing along the arcs between the leaves and nodes are probabilities based on document collection statistics. Opposed to TF.IDF, the probabilities are calculated using document frequencies *df* and inverse collection frequencies *icf* (DF.ICF). The inverse collection frequency is the number of collections that contain the term. An inference network with these properties is called a collection retrieval inference network (CORI net).

Given query q with terms r_k, the network is used to obtain a ranked list of collections by calculating the belief $p(r_k|c_i)$ in collection c_i due to the observation of r_k. The collection ranking score of c_i for query q is the sum of all beliefs $p(r_k|c_i)$, where $r \in q$. The belief is calculated using Formula 1. In this formula, b and l are constants, cw_i is the number of words in c_i, $\overline{cw}$ is the mean number of words in the collections and $|C|$ is the number of collections. *df* and *cf* respectively are the number of documents and collections that contain r_k. Finally, d_t and d_b respectively are the minimum term frequency component and minimum belief component when r_k occurs in c_i.

To improve retrieval, the component L is used to scale the document frequency in the calculation of T [6, 23]. L is made sensitive to the number of documents that contain term r_k (using b) and is made large using l. L should be large, because *df* is generally large. Proper tuning of these two parameters is required for every data set, but deemed impossible as the correct settings are highly sensitive to data set variations [12]; the value of l should be varied largely even when keeping the data set constant while varying the query type.

Further research showed that it is not justified to use standard CORI as a collection selection benchmark. D'Souza *et al.* showed that HIGHSIM outperformed CORI in 15 of 21 cases [11]. Si and Callan [28] found limitations with different collection sizes. Large collections are not often ranked high by CORI, although they often are the most promising collections.

$$
\begin{aligned}
L &= l \cdot ((1-b) + b \cdot cw_i/\overline{cw}) \\
T &= d_t + (1-d_t) \cdot \frac{\log(df)}{\log(df+L)} \\
I &= \frac{\log\left(\frac{|C|+0.5}{cf}\right)}{\log|C| + 1.0} \\
p(r_k|c_i) &= d_b + (1-d_b) \cdot T \cdot I
\end{aligned} \tag{1}
$$

2.2 Language modeling

A language model is a statistical model to assign a probability to a sequence of words (e.g. a query) being generated by a particular document or document collection. Language models can be used for collection selection in distributed search, by creating a language model for each collection [29, 37]. They have also been used for collection selection for other tasks, for instance for blog search [2, 26].

INDRI is an improved version of the INQUERY retrieval system [33], as it is capable of dealing with larger collections, allows for more complex queries due to new query constructs and uses formal probabilistic document representations that use language models. The combined model of inference networks and language modeling is capable of achieving more favorable retrieval results than INQUERY [18]. Due to these differences, term representation beliefs are calculated in an-

other way than with CORI (as described in Section 2.1):

$$P(r|D) = \frac{tf_{r,D} + \alpha_r}{|D| + \alpha_r + \beta_r}$$

The belief is calculated for representation concept r of document node D (in document collection C). Examples of representation concepts are a term in the body or title of a document. D and r are nodes in the belief network. The term frequency of representation node r in D is denoted by $tf_{r,D}$. α_r and β_r are smoothing parameters. Smoothing is used to make the maximum likelihood estimations of a language model more accurate, which could have been less accurate due to data sparseness, because not every term occurs in a document [39]. Smoothing ensures that terms that are unseen in the document, are not assigned zero probability. The default smoothing model for INDRI is Dirichlet smoothing, which affects the smoothing parameter choices [17]. In setting the smoothing parameters, it was assumed that the likeliness of observing representation concept r is equal to the probability of observing it in collection C, given by $P(r|C)$. This means that $\alpha_r/(\alpha_r + \beta_r) = P(r|C)$. The following parameter values were chosen:

$$\alpha_r = \mu P(r|C)$$
$$\beta_r = \mu(1 - P(r|C))$$

This results in the following term representation belief, where the free parameter μ has a default value of 2500:

$$P(r|D) = \frac{tf_{r,D} + \mu P(r|C)}{|D| + \mu}$$

2.3 Discussion

The language modeling approach of INDRI has a better formal probabilistic document representation than CORI and INDRI is an improved version of INQUERY (which is the foundation of CORI). We will use the language model on document collections as implemented by INDRI as our baseline collection selection system. Si *et al.* [29] showed that a language modeling approach for collection selection outperforms CORI. Furthermore, CORI outperforms algorithms like CVV and GGLOSS in several studies [9, 21].

3. SOPHOS

Sophos is a collection selection prototype that uses HDKs to index collections. The keys are used to assign scores to the collections. Using a scoring function, collection scores can be calculated to rank collections for a particular query. A general overview is depicted in Figure 2. This section describes how the broker index is created, explains index size reduction using a query-driven indexing approach, identifies query result parameters, and concludes with a collection ranking formula (ScoreFunction in Figure 2).

3.1 Highly discriminative keys

Building the index is done in two phases. First, every peer builds an index of its document collection and sends that index to the broker. Second, the broker constructs a broker index from all peer indices.

3.1.1 Peer indexing

Peer indexing starts with the generation of term set statistics. First, single term statistics are generated by counting

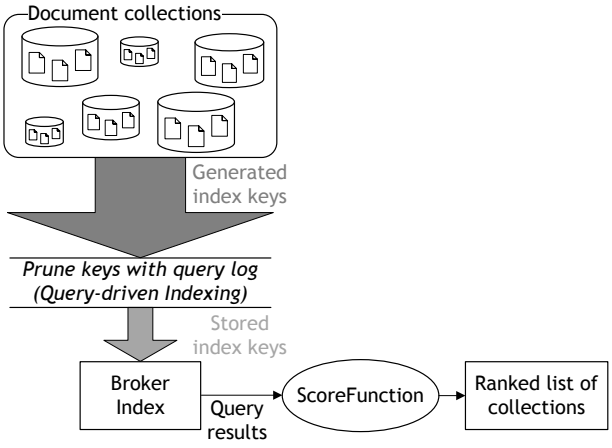


Figure 2: General overview of the indexing and collection selection system Sophos

term frequencies of every word in the collection, without looking at document boundaries in the collection. A term set is called *frequent* when it occurs more times than a term set frequency maximum $tf_{\max}$. Every infrequent single term is added to the peer index together with its frequency count. The frequent keys are stored for further analysis.

The next step consists of obtaining double term set statistics. For every frequent term in the collection, frequency statistics are created for term combinations that consist of the frequent term and a term that appears after that term within a window size ws. The result is a list of double terms with corresponding frequency counts. Once again, the term set frequency maximum defines which term sets are frequent and will be used for further analysis, and which term sets are infrequent and will be stored in the peer index. This procedure can be repeated as long as the generated term sets do not contain more than ws terms, or when a predefined maximum number of terms in a term set, $h_{\max}$, has been reached.

Summarizing, the peer index consists of tuples with term sets and corresponding frequency counts.

3.1.2 Broker indexing

The infrequent term sets from the peer indices are sent to the broker. The broker index contains term sets with associated sets of collection identifier counters. A *collection identifier counter* is a tuple of a collection identifier and a term set frequency counter. A collection identifier is a short representation of a collection where the term set occurs.

When a term set is inserted into the broker index, it is called a highly discriminative key (HDK). The broker index will contain a maximum number of collections per HDK, denoted by the collection maximum cm. As soon as the maximum number of collections is reached for a particular HDK, the cm collections with the largest term set frequencies will be stored in the broker index.

3.2 Query-driven indexing

A list of popular keys can be created by extracting all unique queries from a query log. Every key that is infrequent at a peer, and which is present in the unique query list, will be sent to the broker; the other keys are filtered out to reduce the broker index size and to reduce traffic.

c	sum of query term set occurrence within one collections (grouped by sets with h terms)
h	#terms in an HDK
$h_{\max}$	maximum #terms that HDK can consist of
q	#terms in query
n	#distinct query terms found in a collection
$tf_{\max}$	maximum frequency of a term set in a collection

Table 1: Parameter definitions for query result handling

This index pruning strategy was used before by Shokouhi *et al.* [27]. It is not the most desirable strategy for query-driven indexing, because it is unable to deal with unseen query terms. However, it will give a good idea about the possible index size reduction and the loss of retrieval performance.

3.3 Identifying query result parameters

Once the broker index has been built, the system is ready to be queried. The broker index contains HDKs with h terms, where h varies from 1 to $h_{\max}$. In Sophos, $h_{\max}$ is set to 3. Every key has an associated posting list, which contains tuples of collection identifiers and counters. A counter indicates the number of occurrences of a certain key within the corresponding collection. The counter value cannot exceed the term set frequency maximum, $tf_{\max}$, as a key would otherwise have been locally frequent and new HDKs would have been generated when the maximum number of terms, $h_{\max}$, was not yet reached.

When a user poses a query with q terms, e.g. abcde with $q = 5$, the query is first decomposed in query term combinations with length $h_{\max}$ (i.e. abc, abd, ..., bde, cde). This results in $\binom{q}{h}$ combinations. Each combination is looked up in the broker index. Note that this introduces additional load on the broker, but these lookups do not require network access to the peers. The results of each of those smaller queries are merged by summing the number of occurrences per collection. The sum of all counters, c, has a potential maximum of $\binom{q}{h} \cdot tf_{\max}$. It may happen that this procedure results in little or no collections where an HDK occurs. The procedure is then repeated for smaller term sets; first term sets of two terms will be looked up in the index. When even this gives too few results, single terms will be looked up in the index. In the case that one collection contains two different combinations, e.g. both abc and bce occur in collection X, the number of occurrences are summed (this is c that was just introduced). However, it also interesting to note that 4 out of 5 query terms can be found in collection X. The number of query terms that can be found using HDKs of a particular length is indicated by n. The different parameters are displayed in Table 1.

3.4 ScoreFunction: ranking query results

ScoreFunction, given in Formula 2, is a ranking formula that uses the query result parameters to enforce a collection ranking conforming to our desired ranking properties. It calculates a score s for a collection for a given query

$$s = \log_{10}\left(\left[h - 1 + \frac{n-1}{q} + \frac{\sqrt{\frac{c}{(h_{\max}+1-h)\cdot\binom{n}{h}\cdot tf_{\max}}\cdot\alpha^{(q-n)}}}{q}\right]/h_{\max}\right) \quad (2)$$

It consists of three components; one component per desired ranking property. The properties, and corresponding components, are listed below in order of importance.

1. Collections that contain longer HDKs should be ranked higher. Component 1: $h - 1$.
2. A collection should be ranked higher if it contains more distinct query terms. Component 2: $(n-1)/q$.
3. More occurrences of query term sets within a collection should result in a higher collection ranking. Component 3: $\frac{\sqrt{\frac{c}{(h_{\max}+1-h)\cdot\binom{n}{h}\cdot tf_{\max}}\cdot\alpha^{(q-n)}}}{q}$.

   The general intuition behind this component is that a collection is more important when it contains more query terms. This is controlled by a damping factor α. The term set counts have less impact when they get closer to $tf_{\max}$, because longer keys would be generated for a term set in a collection when its frequency exceeds $tf_{\max}$. We refer to earlier work for a more detailed explanation about ScoreFunction [4].

An important detail to mention about querying and ranking is that collection X can be found after looking for HDKs with length h. When the same collection X is found after looking for HDKs with length $h - 1$, those results are discarded as the collection was already found using larger HDKs. Counts c are only compared with other counts that are retrieved after looking for HDKs of the same length. The motivation for this behavior is that smaller HDKs tend to have higher counts.

Each of the three components has a share in the collection score. The component share of a less desired property never exceeds that smallest possible share of a more desired property's component value.

4. EXPERIMENT SETUP

This section describes the corpus, query set and query logs that were used in the evaluation process, and continues to describe how the collection selection effectiveness of Sophos was measured.

4.1 Data collections

4.1.1 WT10G corpus splits

The Web Track 10GB corpus (WT10g) was developed for the Text REtrieval Conference[1] and consists of 10GB of web pages (1,692,096 documents on 11,680 servers). Compared to regular TREC corpora, WT10g should be more suited for distributed information retrieval experiments, due to the existence of hyperlinks, differences in topics, variation in quality and presence of duplicates [3, 8].

[1] http://trec.nist.gov/

Four splits of the WT10G document corpus were made to look at the effect of document corpora on collection selection. Every split is a set of collections; every collection is a set of documents. The numbers 100 and 11512 indicate the amount of collections in the corpus split.

IP Split: Documents are put in collections based on the IP addresses of the site where a document was residing. This results in 11,512 collections.

IP Merge 100: A cluster is created by grouping up to 116 collections, which results in 100 collections. Grouping is done in order of IP address. This split simulates the scenario of indexing the search engines that index many servers.

Random 100 and Random 11512: Two random splits with 100 collections and with 11,512 collections were created. Documents were randomly assigned to a collection. The number of 11,512 collections was chosen to be able to compare a random split with the IP Split.

The number of documents in random splits is relatively constant, but varies in IP based collections from less than 10 up to more than 1000 documents. This is typical for the size of sites on the Internet; the number of documents per server follows a Zipf distribution on the Internet [1]. The IP based splits show signs of conformance to a Zipf distribution [4]. This means that the IP based splits can be compared to the Internet in terms of distribution of the number of documents and the sizes of the collections.

The *merit* of a collection is the number of relevant documents in a collection for a particular query. The IP based corpus splits have a larger deviation in merit among the collections. This contrasts with random splits, which by approximation have equal merit for each collection [4].

4.1.2 WT10g retrieval tasks

Fifty WT10g informational 'ad-hoc' queries were used for evaluation (query numbers 501–550). The queries have a query number, title, description and a narrative description of the result that is considered relevant. The title is a small set of words which was used as the query text. The narrative descriptions were used by humans to assign relevance judgments to documents. The relevance judgments can be used to count the number of relevant documents in the collections, which in turn can be used to measure collection selection effectiveness.

There are three types of relevance judgments: not relevant (0), relevant (1) and authoritative (2). There can be multiple authoritative documents in the document corpus for a query, but for some queries there are no authoritative documents. All authoritative judgments are converted to 1, so documents are either relevant or not relevant. This allows for evaluation of the collected merit.

4.1.3 Query logs

AOL published a query log with 21,011,340 queries [19]. The log has been anonymized and consists of several data fields: the actual query issued and the query submission date and time, and an anonymous user ID number. The release of the anonymized data set was controversial at the time because it was proven possible to link an ID to a real person. To respect the anonymity of the users, we used a stripped version of the query log that only contains the actual queries issued in random order (and none of the other metadata).

We also used two query logs that were published by Excite[2] that were stripped in the same way. One query log from 1997 contains 1,025,907 queries and another query log from 1999 contains 1,777,251 queries.

Finally, a fourth query log with 3,512,320 unique queries was created by removing all queries from the AOL query log that were issued only once. This query log will be referred to as AOL2. The other logs are called AOL, Excite97 and Excite99.

4.2 Method

To evaluate the performance of our collection selection system, we adopted the precision and recall metrics for collection selection as defined by Gravano *et al.* [13]. We start by obtaining the *retrieval system ranking* (S) for a query, which contains up to 1,000 collections. We also create the *best ranking* (B) which is the best possible ranking for a particular query; collections are ranked by their amount of merit with respect to a query.

Given query q and collection c_i, the merit within a collection can be expressed using $merit(q, c_i)$. The merit of the i^{th} ranked collection in rankings S and B is given by $S_i = merit(q, c_{s_i})$ and $B_i = merit(q, c_{b_i})$.

The obtained recall after selecting n collections can be calculated by dividing the merit selected by the best possible retrieval system:

$$R_n = \frac{\sum_{i=1}^{n} S_i}{\sum_{i=1}^{n} B_i} \tag{3}$$

Precision P_n is the fraction of top n collections that have non-zero merit:

$$P_n = \frac{|\{\text{sc} \in \text{Top}_n(S) | \text{merit}(q, \text{sc}) > 0\}|}{|\text{Top}_n(S)|} \tag{4}$$

The precision and recall obtained by Sophos is compared to the collection selection results from a baseline of Language Modeling with Dirichlet Smoothing (LMDS) as implemented by INDRI. The baseline system has one parameter μ that is set to 2500. Krovetz word stemming is applied to the collections.

Section 3 introduced Sophos and its parameters. There are three parameters for peers: the maximum key length $h_{\max}$, the maximum key frequency before calling a key frequent ($tf_{\max}$) and the window size ws. Based on numbers used by Luu *et al.* [15], we use the following settings: $tf_{\max} = \{250, 500, 750\}$ and $ws = \{6, 12, 18\}$. The average query length on the Internet is 2.3 [14, 30]. We use this observation to set $h_{\max}$ to 3. Setting it smaller would require many intersections of term sets (with associated collections) to answer queries. Setting it larger would result in many term sets that are rarely queried for. For the broker, the collection maximum cm is tested for values 5, 10, 20 and 50.

To evaluate Sophos, the collections are processed by removing stop words – drastically reducing the number of invaluable keys and speeding up term set generation – and applying Porter word stemming.

Finally, we will look at the precision and recall with query-driven indexing using four different query logs. By pruning the keys from the peer indices that do not occur in a query

[2] `http://www.excite.com/`

log. At the same time, we will look at the number of term sets (keys) in the broker index to get an idea about its size.

5. RESULTS

5.1 Index size

Figure 3 shows the number of collection identifier counters within the broker index for different indexing settings of indexing the IP Split. Spread over single, double and triple term set collection identifier counters, the number of counters are a good indication for the actual broker index size. The figure shows that the number of counters decreases when the term set frequency maximum is increased.

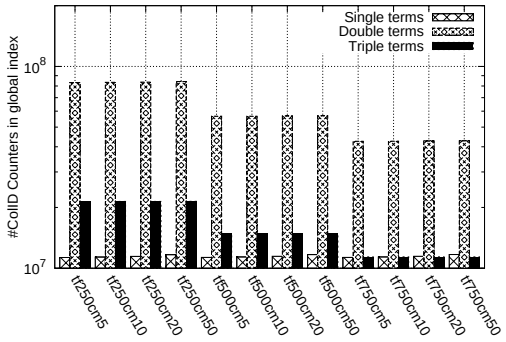


Figure 3: Number of collection ID counters with IP Split.

A more substantial reduction of collection identifier counters – of roughly 70% – can be achieved by using query-driven indexing, as shown in Figure 4. The figure shows the number of collection identifier counters after indexing the Random 11,512 corpus split with or without query-driven indexing. Figure 5 depicts the obtainable index size reduction by using different query logs. The Excite query logs contain significantly less query term sets than the AOL query log. The figure shows that more keys are pruned from the peer indices, resulting in a smaller broker index. The figure shows that using Excite query logs instead of the standard AOL query log can result in roughly 75% less collection identifier counters.

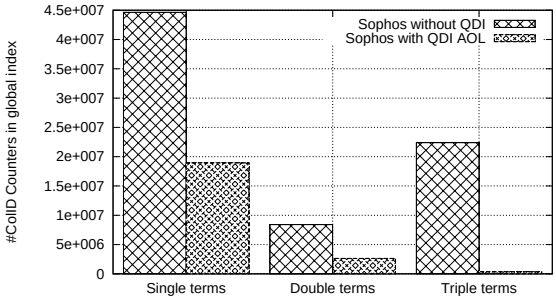


Figure 4: Number of collection identifier counters stored per QDI strategy for Random 11512 corpus split

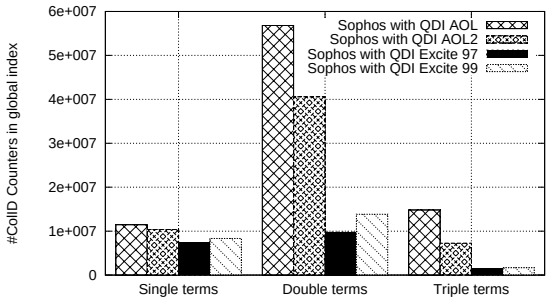


Figure 5: Number of collection identifier counters stored per QDI strategy for IP Split

5.2 Collection selection performance

Table 2 shows the average precision and recall over 50 queries that were calculated with Formula 3 and Formula 4. The numbers are calculated for four different corpus splits with which the baseline (LMDS) and Sophos were tested. Sophos was used with the following settings: query-driven indexing with the AOL query log, $tf_{\max} = 250$, $cm = 20$ and $ws = 6$. Due to memory constraints, we were unable to run Sophos with a window size larger than 6. The table shows that the baseline outperforms Sophos on the Random 11,512 corpus split, but Sophos outperforms the baseline on the IP split.

This is displayed in more detail in Figure 6, which shows the average recall of Sophos and the baseline after selecting n collections. Sophos was tested using different query logs, $tf_{\max} = 250$ and $cm = 50$. Interestingly, pruning with the smallest Excite query logs results in the best recall figures, possibly because the queries were logged at the same time as when the corpus was crawled.

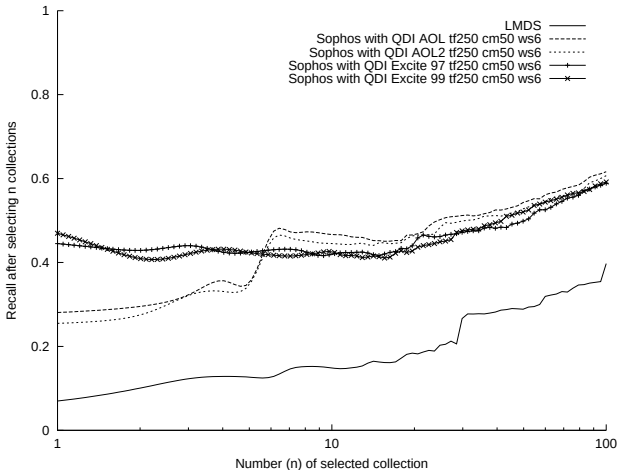


Figure 6: Recall of different collections selection methods on IP Split

6. CONCLUSIONS

We introduced the collection selection system Sophos that uses highly discriminative keys in peer indices to construct a broker index. The broker index contains keys that are good discriminators to select collections (or peers). To limit

Corpus split	Collection selection method	P@1	P@10	P@20	P@50	R@1	R@10	R@20	R@50
Random 100	LMDS	0.290	0.251	**0.249**	**0.217**	0.314	0.435	**0.526**	**0.683**
	Sophos QDI AOL tf250 cm20	**0.330**	**0.254**	0.237	0.208	**0.379**	**0.436**	0.493	0.644
Random 11512	LMDS	**0.140**	**0.096**	**0.084**	**0.069**	**0.233**	**0.196**	**0.186**	**0.198**
	Sophos QDI AOL tf250 cm20	0.040	0.036	0.037	0.026	0.067	0.073	0.082	0.073
IP Merge 100	LMDS	0.280	0.202	0.188	0.154	**0.489**	**0.489**	0.567	0.755
	Sophos QDI AOL tf250 cm20	**0.300**	**0.254**	**0.214**	**0.160**	0.211	0.485	**0.626**	**0.825**
IP Split	LMDS	**0.170**	0.110	0.083	0.056	0.070	0.149	0.183	0.289
	Sophos QDI AOL tf250 cm20	**0.170**	**0.147**	**0.121**	**0.091**	**0.280**	**0.466**	**0.466**	**0.548**

Table 2: Average precision and recall over 50 queries after selecting n collections, high scores shown in bold

the number of keys transferred to the broker, and to reduce the broker index size, we employed query-driven indexing to only store the keys that are queried for by users. Similar studies were performed for document retrieval [15], but to the best of our knowledge, we are the first to apply this approach for collection selection.

Precision and recall was measured using 50 queries on the WT10g TREC test corpus and compared to a state-of-the-art baseline that uses language modeling with Dirichlet smoothing (LMDS). The results showed that our system outperformed the baseline with the IP split as test corpus. This is promising, because the IP based splits most closely resemble the information structure on the Internet. LMDS was better capable of selecting information in random based splits, because it stores all available information about the collections. In random based splits, relevant documents (and their corresponding terms) are mixed over many collections, making it hard for our approach to select highly discriminative keys that can discriminate collections with relevant documents.

Query-driven indexing is required to keep the broker index size manageable; a 70% index size reduction can be obtained by pruning keys using the AOL query log, another 75% reduction is possible by using a smaller query log. Our results on the IP split showed that pruning using the smaller Excite query logs resulted in higher precision and recall than with AOL query logs. The use of any query log resulted in higher precision and recall than the baseline results. This motivates further research using more advanced query-driven indexing strategies, such as described by Skobeltsyn [32], to further reduce the index size while improving the performance. It would also be interesting to make $tf_{\max}$ depending on the collection size.

Acknowledgements

We are grateful to the Yahoo Research Faculty Grant programme and to the Netherlands Organisation for Scientific Research (NWO, Grant 639.022.809) for supporting this work. We would like to thank Berkant Barla Cambazoglu and the anonymous reviewers for their valuable comments that improved this paper.

7. REFERENCES

[1] L. A. Adamic and B. A. Huberman. Zipf's law and the internet. *Glottometrics*, 3:143–150, 2002.

[2] J. Arguello, J. L. Elsas, J. Callan, and J. G. Carbonell. Document representation and query expansion models for blog recommendation. In *Proc. of the 2nd Intl. Conf. on Weblogs and Social Media (ICWSM)*, 2008.

[3] P. Bailey, N. Craswell, and D. Hawking. Engineering a multi-purpose test collection for web retrieval experiments. *Inf. Process. Manage.*, 39(6):853–871, 2003.

[4] S. Bockting. Collection selection for distributed web search. Master's thesis, University of Twente, Feb. 2009.

[5] J. Callan. Distributed information retrieval. *Advances in Information Retrieval*, pages 127–150, 2000.

[6] J. Callan, W. B. Croft, and S. M. Harding. The inquery retrieval system. In *Proc. of the 3rd International Conference on Database and Expert Systems Applications*, pages 78–83, Valencia, Spain, 1992. Springer-Verlag.

[7] J. Callan, Z. Lu, and W. B. Croft. Searching distributed collections with inference networks. In *SIGIR '95: Proc. of the 18th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 21–28, New York, NY, USA, 1995. ACM.

[8] N. Craswell, P. Bailey, and D. Hawking. Is it fair to evaluate web systems using trec ad hoc methods. In *Workshop on Web Evaluation*. SIGIR'99, 1999.

[9] N. Craswell, P. Bailey, and D. Hawking. Server selection on the world wide web. In *DL '00: Proc. of the 5th ACM conference on Digital libraries*, pages 37–46, New York, NY, USA, 2000. ACM.

[10] D. D'Souza, J. Thom, and J. Zobel. A comparison of techniques for selecting text collections. In *ADC '00: Proceedings of the Australasian Database Conference*, page 28, Washington, DC, USA, 2000. IEEE Computer Society.

[11] D. D'Souza, J. A. Thom, and J. Zobel. Collection selection for managed distributed document databases. *Information Processing & Management*, 40(3):527–546, May 2004.

[12] D. D'Souza, J. Zobel, and J. A. Thom. Is cori effective for collection selection? an exploration of parameters, queries, and data. In *Proc. of the 9th Australasian Document Computing Symposium*, pages 41–46, Dec. 2004.

[13] L. Gravano and H. Garcia-Molina. Generalizing GlOSS to vector-space databases and broker hierarchies. In *International Conference on Very Large Databases, VLDB*, pages 78–89, 1995.

[14] T. Lau and E. Horvitz. Patterns of search: analyzing and modeling web query refinement. In *UM '99: Proc. of the 7th international conference on User modeling*, pages 119–128, Secaucus, NJ, USA, 1999.

Springer-Verlag New York, Inc.

[15] T. Luu, F. Klemm, M. Rajman, and K. Aberer. Using highly discriminative keys for indexing in a peer-to-peer full-text retrieval system. Technical report, TR: 2005041, EPFL Lausanne, 2005.

[16] P. Lyman and H. R. Varian. How much information, 2003. `http://www.sims.berkeley.edu/how-much-info-2003`, retrieved on April 23, 2008.

[17] D. Metzler. Indri retrieval model overview, July 2005. `http://ciir.cs.umass.edu/~metzler/indriretmodel.html`, retrieved on January 20, 2008.

[18] D. Metzler and W. B. Croft. Combining the language model and inference network approaches to retrieval. *Information Processing and Management*, 40(5):735–750, 2004.

[19] G. Pass, A. Chowdhury, and C. Torgeson. A picture of search. In *InfoScale '06: Proc. of the 1st international conference on Scalable information systems*, page 1, New York, NY, USA, 2006. ACM.

[20] I. Podnar Zarko, M. Rajman, T. Luu, F. Klemm, and K. Aberer. Scalable peer-to-peer web retrieval with highly discriminative keys. *IEEE 23rd International Conference on Data Engineering (ICDE)*, 2007.

[21] A. L. Powell and J. C. French. Comparing the performance of collection selection algorithms. *ACM Trans. Inf. Syst.*, 21(4):412–456, 2003.

[22] D. Puppin, F. Silvestri, and D. Laforenza. Query-driven document partitioning and collection selection. In *InfoScale '06: Proc. of the 1st international conference on Scalable information systems*, page 34, New York, NY, USA, 2006. ACM.

[23] S. E. Robertson, S. Walker, S. Jones, M. M. Hancock-beaulieu, and M. Gatford. Okapi at trec-3. In *TREC-3 Proc.*, pages 109–126, 1995.

[24] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[25] N. Sato, M. Udagawa, M. Uehara, Y. Sakai, and H. Mori. Query based site selection for distributed search engines. *Distributed Computing Systems Workshops, 2003. Proc.. 23rd International Conference on*, pages 556–561, 2003.

[26] J. Seo and W. B. Croft. Umass at trec 2007 blog distillation task. In *Proc. of the 2008 Text REtrieval Conference*. NIST, 2007.

[27] M. Shokouhi, J. Zobel, S. Tahaghoghi, and F. Scholer. Using query logs to establish vocabularies in distributed information retrieval. *Information Processing and Management*, 43(1):169–180, 2007.

[28] L. Si and J. Callan. Relevant document distribution estimation method for resource selection. In *SIGIR '03: Proc. of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pages 298–305, New York, NY, USA, 2003. ACM.

[29] L. Si, R. Jin, J. Callan, and P. Ogilvie. A language modeling framework for resource selection and results merging. *Proc. of the 11th international conference on Information and knowledge management*, pages 391–397, 2002.

[30] C. Silverstein, H. Marais, M. Henzinger, and M. Moricz. Analysis of a very large web search engine query log. *SIGIR Forum*, 33(1):6–12, 1999.

[31] G. Skobeltsyn. *Query-driven indexing in large-scale distributed systems*. PhD thesis, EPFL, Lausanne, 2009.

[32] G. Skobeltsyn, T. Luu, I. Podnar Zarko, M. Rajman, and K. Aberer. Query-driven indexing for scalable peer-to-peer text retrieval. *Future Generation Computer Systems*, 25(1):89–99, June 2009.

[33] T. Strohman, D. Metzler, H. Turtle, and W. B. Croft. Indri: A language model-based search engine for complex queries. In *Proc. of the International Conference on Intelligence Analysis*, 2004.

[34] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. *Proc. of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 175–186, 2003.

[35] Y. Wang and D. J. DeWitt. Computing pagerank in a distributed internet search system. In Proc. of the International Conference on Very Large Databases (VLDB), Aug. 2004.

[36] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.

[37] J. Xu and W. B. Croft. Cluster-based language models for distributed retrieval. *Proc. of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 254–261, 1999.

[38] B. Yuwono and D. L. Lee. Server ranking for distributed text retrieval systems on the internet. In *Proc. of the 5th International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 41–50. World Scientific Press, 1997.

[39] C. Zhai and J. Lafferty. A study of smoothing methods for language models applied to ad hoc information retrieval. In *SIGIR '01: Proc. of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 334–342, New York, NY, USA, 2001. ACM.

PP-Index: Using Permutation Prefixes for Efficient and Scalable Approximate Similarity Search

Andrea Esuli
Istituto di Scienza e Tecnologie dell'Informazione- Consiglio Nazionale delle Ricerche
via Giuseppe Moruzzi, 1- 56124, Pisa - ITALY
andrea.esuli@isti.cnr.it

ABSTRACT

We present the Permutation Prefix Index (PP-Index), an index data structure that allows to perform efficient approximate similarity search.

The PP-Index belongs to the family of the permutation-based indexes, which are based on representing any indexed object with "its view of the surrounding world", i.e., a list of the elements of a set of reference objects sorted by their distance order with respect to the indexed object.

In its basic formulation, the PP-Index is strongly biased toward efficiency, treating effectiveness as a secondary aspect. We show how the effectiveness can easily reach optimal levels just by adopting two "boosting" strategies: multiple index search and multiple query search. Such strategies have nice parallelization properties that allow to distribute the search process in order to keep high efficiency levels.

We study both the efficiency and the effectiveness properties of the PP-Index. We report experiments on collections of sizes up to one hundred million images, represented in a very high-dimensional similarity space based on the combination of five MPEG-7 visual descriptors.

Categories and Subject Descriptors

E.1 [**Data**]: Data Structures; H.3.3 [**Information Systems**]: Information Storage and Retrieval—*Search process*

General Terms

Algorithms,Experimentation,Performance

Keywords

approximate similarity search, metric space, scalability

1. INTRODUCTION

The similarity search model [12] is a search model in which, given a query q and a collection of objects D, all belonging to a domain $\mathcal{O}$, the objects in D have to be sorted by their similarity to the query, according to a given *distance function* $d : \mathcal{O} \times \mathcal{O} \rightarrow \mathbb{R}^+$ (i.e., the closer two objects are, the most similar they are considered). Typically only the k-top ranked objects are returned (*k-NN query*), or those within a maximum distance value r (*range query*).

One of the main research topics on similarity search is the study of the scalability of similarity search methods when applied to high-dimensional similarity spaces.

The well known "curse of dimensionality" [7] is one of the hardest obstacles that researchers have to deal with when working on this topic. Along the years, such obstacle has been attacked by many proposals, using many different approaches. The earliest and most direct approach consisted in trying to improve the data structures used to perform *exact* similarity search. Research moved then toward the exploration of *approximate* similarity search methods, mainly proposing variants of exact methods in which some of the constraints that guarantee the exactness of the results are relaxed, trading effectiveness for efficiency.

Approximate methods [17] that are not derived from exact methods have been also proposed. On this field, the recent research on *permutation-based indexes* (PBI) [1, 6] has shown a promising direction toward scalable data structures for similarity search.

In this work we present the Permutation Prefix Index (PP-Index), an approximate similarity search structure belonging to the family of the permutation-based indexes. We describe the PP-Index data structures and algorithms, and test it on data sets containing up to 100 million objects, distributed on a very high-dimensional similarity space. Experiments show that the PP-Index is a very efficient and scalable data structure both at index time and at search time, and it also allows to obtain very good effectiveness values. The PP-Index has also nice parallelization properties that allow to distribute both the index and the search process in order to further improve efficiency.

2. RELATED WORKS

The PP-Index belongs to the family of the permutation-based indexes, a recent family of data structure for approximate similarity search, which has been independently introduced by Amato and Savino [1] and Chavez et al. [6].

The PP-Index has however a key difference with respect to previously presented PBIs: as we detail in this section, such PBIs use permutations in order to estimate the real distance order of the indexed objects with respect to a query. The PP-Index instead uses the permutation prefixes in order to quickly retrieve a reasonably-sized set of candidate objects,

LSDS-IR Workshop. July 2009. Boston, USA.

which are likely to be at close distance to the query object, then leaving to the original distance function the selection of the best elements among the candidates.

For a more detailed review of the most relevant methods for similarity search in metric spaces we point the reader to the book of Zezula et al. [18]. The recent work of Patella and Ciaccia [8] more specifically analyzes and classifies the characteristics of many approximate search methods.

Chávez et al. [6] present an approximate similarity search method based on the intuition of "*predicting the closeness between elements according to how they order their distances towards a distinguished set of anchor objects*".

A set of *reference objects* $R = \{r_0, \ldots, r_{|R|-1}\} \subset \mathcal{O}$ is defined by randomly selecting $|R|$ objects from D. Every object $o_i \in D$ is then represented by Π_{o_i}, consisting of the list of identifiers of reference objects, sorted by their distance with respect to the object o_i. More formally, Π_{o_i} is a *permutation* of $\langle 0, \ldots, |R|-1 \rangle$ so that, for $0 < i < |R|$ it holds either (i) $d(o_i, r_{\Pi_{o_x}(i-1)}) < d(o_i, r_{\Pi_{o_x}(i)})$, or (ii) $d(o_i, r_{\Pi_{o_x}(i-1)}) = d(o_i, r_{\Pi_{o_x}(i)})$ and $\Pi_{o_x}(i-1) < \Pi_{o_x}(i)$, where $\Pi_{o_x}(x)$ returns the i-th value of Π_{o_x}.

All the permutations for the index objects are stored in main memory. Given a query q, all the indexed permutations are sorted by their similarity with Π_q, using a similarity measure defined on permutations. The real distance d between the query and the objects in the data set is then computed by selecting the objects from the data set following the order of similarity of their permutations, until the requested number of objects is retrieved. An example of similarity measure on permutations is the *Spearman Footrule Distance* [9]:

$$SFD(o_x, o_y) = \Sigma_{r \in R} |P(\Pi_{o_x}, r) - P(\Pi_{o_y}, r)| \qquad (1)$$

where $P(\Pi_{o_x}, r)$ returns the position of the reference object r in the permutation assigned to Π_{o_x}.

Chávez et al. do not discuss the applicability of their method to very large data sets, i.e., when the permutations cannot be all kept in main memory.

Amato and Savino [1], independently of [6], propose an approximate similarity search method based on the intuition of representing the objects in the search space with "*their view of the surrounding world*".

For each object $o_i \in D$, they compute the permutation Π_{o_i} in the same manner as [6]. All the permutations are used to build a set of inverted lists, one for each reference object. The inverted list for a reference object r_i stores the position of such reference object in each of the indexed permutations. The inverted lists are used to rank the indexed objects by their SFD value (equation 1) with respect to a query object q, similarly to [6]. In fact, if full-length permutations are used to represent the indexed objects and the query, the search process is perfectly equivalent to the one of [6]. In [1], the authors propose two optimizations that improve the efficiency of the search process, not affecting the accuracy of the produced ranking. Both optimizations are based on the intuition that the information about the order of the closest reference objects is more relevant than the information about distant ones.

One optimization consists in inserting into the inverted lists only the information related to $\Pi^{k_i}_{o_i}$, i.e., the part of Π_{o_i} including only the first k_i elements of the permutation, thus reducing by a factor $\frac{|R|}{k_i}$ the size of the index. For example, given $|R| = 500$ a value of $k_i = 100$ reduces by five times the number of disk accesses with respect to using full-length permutations, with a negligible loss in accuracy.

Similarly, a value k_s is adopted for the query, in order to select only the first k_s elements of Π_q. Given $|R| = 500$ a value of $k_s = 50$ reduces by ten times the number of disk accesses, also slightly improving the accuracy.

3. THE PP-Index

The PP-Index represents each indexed object with a *very short* permutation prefix.

The PP-Index data structures consists of a *prefix tree* kept in main memory, indexing the permutation prefixes, and a *data storage* kept on disk, from which objects are retrieved by sequential disk accesses.

This configuration of data structures is interestingly similar to the one used by Bawa et al. [4], however, it is relevant to note that our work and [4] are based on completely different approaches to the problem. The latter proposes the LSH-Forest, an improvement to the LSH-Index [11] that is based on using hash keys of variable lenght. These are used to identify a set of candidate objects with hash keys that have a prefix match with the hash key of to the query. Thus the LSH-Forest, like the other LSH-based methods, is based only on probabilistic considerations, while the PP-Index, like the other PBIs, relies on geometrical considerations.

More generally, a key difference between the PBI model and the LSH model is that the hash functions of the LSH model are solely derived from the similarity measures in use, independently of the way the indexed objects are distributed in the similarity space, while in the SPI model the reference objects provide information about this aspect.

The PP-Index is designed to allow very efficient indexing by performing bulk processing of all the objects indexed. Such bulk processing model is based on the intuitive assumption that the data, in the very large collections the PP-Index is designed for, have a relatively static nature. However, it is easy to provide the PP-Index with update capabilities (see Section 3.6).

3.1 Data structures

Given a collection of objects D to be indexed, and the similarity measure d, a PP-Index is built by specifying a set of *reference objects* R, and a *permutation prefix length* l.

Any object $o_i \in D$ is represented by a *permutation prefix* w_{o_i} consisting of the first l elements of the permutation Π_{o_i}, i.e., $w_{o_i} = \Pi^l_{o_i}$. Any object $o_i \in D$ is also associated with a *data block*. A data block b_{o_i} contains (i) the information required to univocally identify the object o_i and (ii) the essential data used by the function d in order to compute the similarity between the object o_i and any other object in $\mathcal{O}$.

The prefix tree of the PP-Index is built on all the permutation prefixes generated for the indexed objects. The leaf at the end of a path relative to a permutation prefix w keeps the information required to retrieve the data blocks relative to the objects represented by w from the data storage.

The data storage consists of a file in which all the data blocks are sequentially stored. The order of objects (represented by data blocks) in the data storage is the same as the one produced by performing an ordered visit of the prefix tree. This is a key property of the PP-Index, which allows to use the prefix tree to efficiently access the data storage.

Specifically, the leaf of the prefix tree relative to permutation prefix w contains two counter values h^{start}_w and h^{end}_w, and two pointer values p^{start}_w and p^{end}_w. The counter values

```
BuildIndex(D, d, R, l)
 1  prefixTree ← EmptyPrefixTree()
 2  dataStorage ← EmptyDataStorage()
 3  for i ← 0 to |D − 1|
 4  do o_i ← GetObject(D, i)
 5     dataBlock_{o_i} ← GetDataBlock(o_i)
 6     p_{o_i} ← Append(dataBlock_{o_i}, dataStorage)
 7     w_{o_i} ← ComputePrefix(o_i, R, d, l)
 8     h_{o_i} ← i
 9     Insert(w_{o_i}, h_{o_i}, p_{o_i}, prefixTree)
10  L ← ListPointersByOrderedVisit(prefixTree)
11  P ← CreateInvertedList(L)
12  ReorderStorage(dataStorage, P)
13  CorrectLeafValues(prefixTree, dataStorage)
14  index ← NewIndex(d, R, l, prefixTree, dataStorage)
15  return index
```

Figure 1: The BuildIndex function.

indicate the ordinal position in the sequence of data blocks in the data storage of the first and the last data blocks relative to the permutation prefix w. The pointer values indicate instead the byte offset in the data storage where the data blocks are effectively serialized. In case the data blocks have a fixed size s, the p values can be omitted and computed when necessary as $p_w = h_w \cdot s$.

The data storage implementation must allow, given any two pointers p' and p'', to sequentially retrieve all the data blocks included between them.

3.2 Building the index

Figure 1 shows a pseudo-code description of the indexing function for the PP-Index.

The indexing algorithm first initializes an empty prefix tree in main memory, and an empty file on disk, to be used as the data storage.

The algorithm takes in input an object $o_i \in D$, for i from 0 to $|D-1|$, and appends its data block at the current end position p_{end} of the data storage file. Then the algorithm computes, for the object o_i, the permutation prefix w_{o_i} (ComputePrefix), and inserts w_{o_i} into the prefix tree. The values $h_{o_i} = i$ and $p_{o_i} = p_{end}$ are stored in the leaf of the prefix tree corresponding to permutation prefix w_{o_i}. When more that one value has to be stored in a leaf, a list is created.

Figure 2 shows an example list of permutation prefixes generated for a set of objects and the data structures resulting from the above discussed first phase of data indexing.

The successive phase (ReorderStorage) consists in reordering the data blocks in the data storage to satisfy the order constrains described in the previous section. An ordered visit of the prefix tree is made in order to produce a list L of the h_{o_i} values stored in the leaves. Thus, the h_{o_i} values in the list L are sorted in alphabetical order of the permutation prefixes their relative objects are associated with.

Data blocks in the data storage are reordered following the order of appearance of h_{o_i} values in list L. For example, given a list for $L = \langle 0, 4, 8, 6, 1, 3, 5, 9, 2, 7 \rangle$, the data block relative to object o_7, identified in the list by the value $h_{o_7} = 7$, has to be moved to the last position in the data storage, since h_{o_7} appears in the last position of the list L (see the values in the leaves of the prefix tree in Figure 2).

To efficiently perform the reordering, the list L is *inverted* into a list P. The i-th position of the list P indicates the new position in which the i-th element of the data storage has to be moved. For the above example,

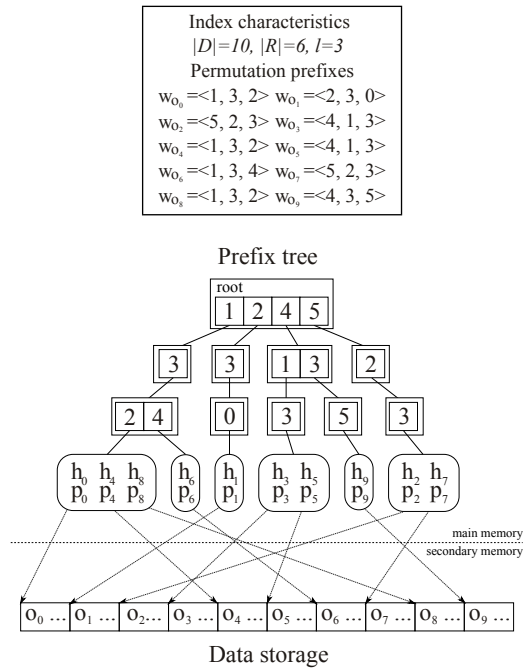


Figure 2: Sample data and partially-built index data structure after the first indexing phase.

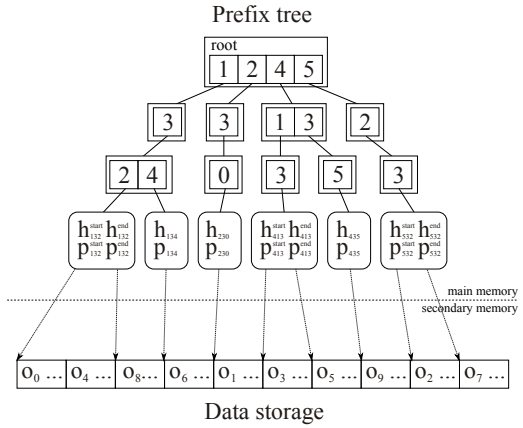


Figure 3: The final index data structure.

$P = \langle 0, 4, 8, 5, 1, 6, 3, 9, 2, 7 \rangle$.

Once P is generated the data storage is reordered accordingly, using an m-way merge sorting method [13]. The advantages of using this method are that it involves only sequential disk accesses, and that it has a small (and configurable) main memory space occupation.

In order to obtain the final index structure, the values in the leaves of the prefix tree have to be updated accordingly to the new data storage (CorrectLeafValues). This is obtained by performing an ordered visit to the prefix tree, the same performed when building the list L, synchronized with a sequential scan of the reordered data storage. The number of elements in the list of a leaf determines the two h^{start} and h^{end} values that replace such list, and also the number of data blocks to be sequentially read from the data storage, in ordered to determine the p^{start} and p^{end} values.

Figure 3 shows the final index data structure.

3.3 Search function

The search function is designed to use the index in order to

```
SEARCH(q, k, z, index)
 1  (p^start, p^end) ← FINDCANDIDATES(q, index.prefixTree,
 2  index.R, index.d, index.l, z)
 3  resultsHeap ← EMPTYHEAP()
 4  cursor ← p^start
 5  while cursor ≤ p^end
 6  do dataBlock ← READ(cursor, index.dataStorage)
 7     ADVANCECURSOR(cursor)
 8     distance ← index.d(q, dataBlock.data)
 9     if resultsHeap.size < k
10        then INSERT(resultsHeap, distance, dataBlock.id)
11        else if distance < resultsHeap.top.distance
12                then REPLACETOP(resultsHeap, distance,
13                     dataBlock.id)
14  SORT(resultsHeap)
15  return resultsHeap

FINDCANDIDATES(q, prefixTree, R, d, l, z)
 1  w_q ← COMPUTEPREFIX(q, R, d, l)
 2  for i ← l to 1
 3  do w_q^i ← SUBPREFIX(w_q, i)
 4     node ← SEARCHPATH(w_q^i, prefixTree)
 5     if node ≠ NIL
 6        then minLeaf ← GETMIN(node, prefixTree)
 7             maxLeaf ← GETMAX(node, prefixTree)
 8             if (maxLeaf.h^end − minLeaf.h^start + 1) ≥ z
 9                ∨i = 1
10                then return (minLeaf.p^start,
11                     maxLeaf.p^end)
12  return (0, 0)
```

Figure 4: The Search function.

efficiently answer to k nearest neighbor queries. The search strategy consists in searching a subtree of the prefix tree that identifies a specified number of candidate objects all represented by permutation prefixes having a prefix match with the permutation prefix representing the query.

A k-NN query is composed of the query object q, the k value, and the z value, indicating the minimum number of *candidate* objects among which the k nearest neighbors have to be selected.

The FINDCANDIDATES function determines the smallest subtree of the prefix tree having a prefix match with the permutation prefix w_q, i.e., the permutation prefix relative to the query q, and retrieving at least z objects. The function returns two pointers p^{start} and p^{end} to the positions in the data storage of the data blocks of the first and the last candidate objects.

The distance of each candidate object with the query is computed, using the distance function d. A heap is used to keep track of the k objects closest to the query.

3.4 Prefix tree optimizations

In order to reduce the main memory occupation of the prefix tree it is possible to simplify its structure without affecting the quality of results. These are search-specific optimizations, and a non-optimized version of the prefix tree should be saved for other operations (e.g., update and merge).

A first optimization consists in pruning any path reaching a leaf which is composed of only-child, given that this kind of path does not add relevant information to distinguish between different existing groups in the index. Another optimization consists in compressing any path of the prefix tree composed entirely of only-children into a single label [15], thus saving the memory space required to keep the chain of nodes composing the path.

A PP-Index-specific optimization, applicable when the z value is hardcoded into the search function, consists in reducing to a single leaf the subtrees of the prefix tree that points to less than z objects, given that none of such subtrees will be ever selected by the search function.

3.5 Improving the search effectiveness

The "basic" search function described is Section 3.3 is strongly biased toward efficiency, treating effectiveness as a secondary aspect. The PP-Index allows to easily tune effectiveness/efficiency trade-off, and effectiveness can easily reach optimal levels just by adopting the two following "boosting" strategies:

Multiple index: t indexes are built, based on different $R_1 \dots R_t$ sets of reference objects. This is based on the intuition that different reference object sets produce many differently *shaped* partitions of the similarity space, resulting in a more complete coverage of the area around queries.

A search process the using multiple index strategy can be parallelized by distributing the indexes over multiple machines, or just on different processes/CPUs on the same machine, maintaing almost the same performance of the basic search function, with a negligible overhead for merging the t k-NN results, as far as there are enough hardware resources to support the number of indexes involved in the process.

Multiple query: at search time, p additional permutation prefixes from the query permutation prefix w_q are generated, by swapping the position of some of its elements. The geometric rationale is that a permutation prefix w' differing from another permutation prefix w'' for the swap of two adjacent/near elements identifies an area $V_{w'}$ of the similarity space adjacent/near to $V_{w''}$ allowing to extend the search process to areas of the search space that are likely to contain relevant objects.

The heuristic we adopt in our experiments for swapping permutation prefix elements consists in sorting all the reference objects pairs appearing in the permutation prefix by their difference of distance with respect to the query object. Then the swapped permutation prefixes are generated by first selecting for swap those pairs of identifiers that have the smallest distance difference.

The multiple query strategy can be parallelized by distributing the queries over different processes/CPUs on the machine handling the index structure.

3.6 Update and merge, distributed indexing

The PP-Index data structures allows to very efficiently merge indexes built using the same parameters into a single index. The merge functionality supports three operations:

Supporting update operations: it is easy to add update capabilities to an index by maintaining a few additional data structures. Deleted objects are managed using a vector of their identifiers. Newly inserted or modified objects are stored in an all-in-memory secondary PP-Index used in conjunction with the main index structure. A periodic merge procedure is used when the secondary index reaches a given memory occupation limit.

Indexing very large collection: the main memory occupation of a prefix tree reaches its maximum during the indexing process, when it has to be entirely kept in memory, while during search, thanks to the optimization methods described in Section 3.4, its size can be reduced by orders of magnitude. This issue is solved building a number of smaller partial indexes and then merging them into the final index.

Distributing the indexing process: the indexing process

of smaller indexes can be distributed of different machines, given that the information contained in any smaller index is completely independent of the one contained in the others. Also the merge process can be distributed, if it is performed in a number of steps that involve the creation of intermediate indexes.

The merge process consists in merging the prefix trees of the source indexes into a single prefix tree, i.e., by enumerating, in alphabetical order, all the permutation prefixes contained in the source indexes.

Such enumeration can be easily produced by performing a parallel ordered visit of all the prefix trees being merged. If the prefix trees of the source indexes are saved to the storage using a depth first visit, the merge process requires only a single read of the serialized prefix trees. Obviously, the new prefix tree is directly serialized on disk during the permutation prefix enumeration process.

In the case of an update process, the identifiers of the deleted objects are used to skip deleted objects during the merge process.

The data storages are merged during the permutation prefix enumeration.

4. EXPERIMENTS

4.1 The CoPhIR data set

The CoPhIR[1] [5] data set has been recently developed within the SAPIR project, and it is currently the largest multimedia metadata collection available for research purposes. It consists of a crawl of 106 millions images from the Flickrphoto sharing website.

The information relative to five MPEG-7 visual descriptors [16] have been extracted from each image, resulting in more than 240 gigabytes of XML description data.

We have randomly selected 100 images from the collection as queries and we have run experiments using the first million (1M), ten millions (10M), and 100 millions (100M) images from the data set.

We have run experiments on a linear combination of the five distance functions for the five descriptors, using the weights proposed in [3].

Descriptor	Type	Dimensions	Weight
Scalable Color	L_1	64	2
Color Structure	L_1	64	3
Color Layout	sum of L_2	80	2
Edge Histogram	L_1	62	4
Homogeneous Texture	L_1	12	0.5

Table 1: Details on the five MPEG-7 visual descriptors used in CoPhIR.

4.2 Configurations and evaluation measures

We have explored the effect of using different sized R sets, by running the experiments using three R set sizes consisting of 100, 200, 500, and 1,000 reference objects. We have adopted a random selection policy of objects from D for the generation of the various R sets, following [6], which reports the random selection policy as a good performer.

In all the experiments we have used a fixed value of $l = 6$.

[1] `http://cophir.isti.cnr.it/`

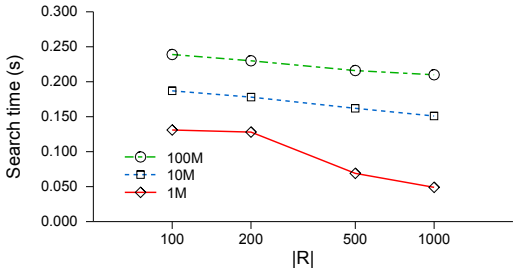


Figure 5: Search time w.r.t. to the size of R, for $z = 1,000$ and $k = 100$ (single index, single query).

We have tested a basic configuration based on the use of a single index and the search function described in Section 3.3, i.e., an efficiency-aimed configuration.

We have then tested the use of multiple indexes , on configurations using 2, 4 and 8 indexes, and also the multiple query search strategy by using a total of 2, 4, and 8 multiple queries. We have also tested the combination of the two strategies.

We have applied the index optimization strategies described in Section 3.4 to all the generated indexes.

On the held-out data we have tested various z values, in this paper we report the results obtained for $z = 1,000$, which has produced a good trade-off in effectiveness/efficiency.

The experiments have been run on a desktop machine running Windows XP Professional, equipped with a Intel Pentium Core 2 Quad 2.4 GHz CPU, a single 1 TB Seagate Barracuda 7,200 rpm SATA disk (with 32 MB cache), and 4 GB RAM. The PP-Index has been implemented in c#. All the experiments have been run in a single-threaded application, with a completely sequential execution of the multiple index/query searches.

We have evaluated the effectiveness of the PP-Index by adopting a *ranking*-based measure and a *distance*-based measure [17], *recall* and *relative distance error*, defined as:

$$Recall(k) = \frac{|D_q^k \cap P_q^k|}{k} \quad (2)$$

$$RDE(k) = \frac{1}{k}\sum_{i=1}^{k}\frac{d(q, P_q^k(i))}{d(q, D_q^k(i))} - 1 \quad (3)$$

where D_q is the list of the elements of D sorted by their distance with respect to q, D_q^k is the list of the k closest elements, P_q^k is the list returned by the algorithm, and $L_q^k(i)$ returns the i-th element of the list L.

4.3 Results

$\|D\|$	indexing time (sec)	prefix tree size		data storage	l'
		full	comp.		
1M	419	7.7 MB	91 kB	349 MB	2.1
10M	4385	53.8 MB	848 kB	3.4 GB	2.7
100M	45664	354.5 MB	6.5 MB	34 GB	3.5

Table 2: Indexing times (with $|R| = 100$), resulting index sizes, and average prefix tree depth l' (after prefix tree compression with $z = 1,000$).

Table 2 reports the indexing times for the various data set sizes ($|R| = 100$), showing the almost perfect linear proportion between indexing time and data set size. With respect to the indexing times we note that: (i) the twelve hours time, required to build the 100M index for the $|R| = 100$, is

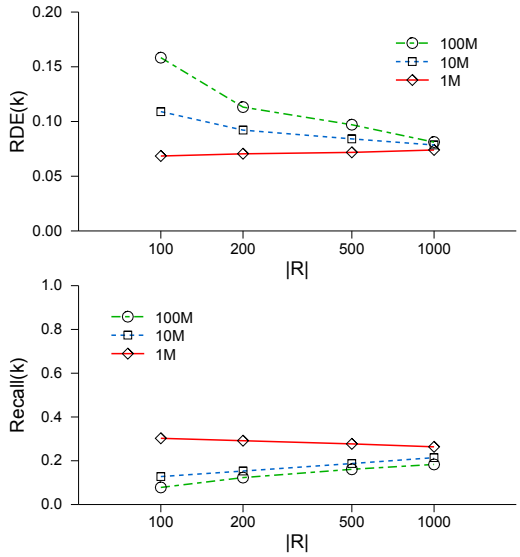


Figure 6: Effectiveness with respect of the size of R set, for $k = 100$ and $z = 1,000$ (single index, single query).

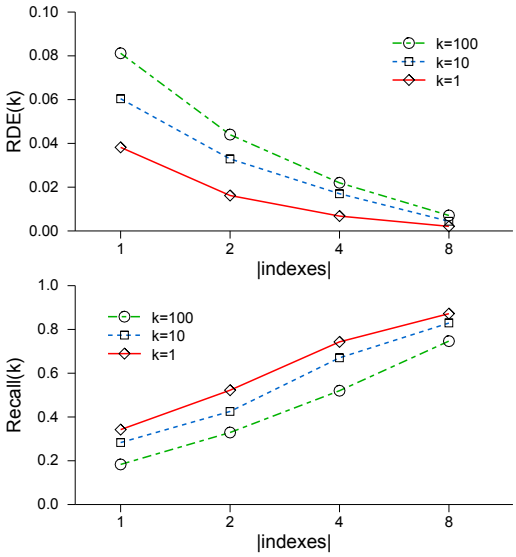


Figure 7: Multiple index search strategy on the 100M index, using $|R| = 1,000$ and $z = 1,000$.

$\|R\|$	$\|D\|$		
	1M	10M	100M
100	4,075	5,817	7,941
200	3,320	5,571	7,302
500	1,803	5,065	6,853
1,000	1,091	4,748	6,644

Table 3: Average z' value, for $z = 1,000$.

in line with the fourteen hours we have measured to build a text search index on the descriptions and the comments associated with the indexed images; (ii) this time refers to a completely sequential indexing process, not leveraging on the parallelization possibilities described in Section 3.6; (iii) we have not explored the possibility of using a similarity search data structure in order to answer l-NN query on the R set necessary to build the permutation prefix.

The table also shows the resulting memory occupation of the prefix tree before and after the application of the compression strategies described in Section 3.4. The values shows how such strategies allows to reduce by orders of magnitude the main memory requirement of the PP-Index (at least by a factor fifty in our case) without affecting the quality of the results.

As expected, the disk occupation is perfectly linear with respect to the data set size, given that the disk data storage contains only a sequential serialization of data blocks (375 bytes each one).

The last column of Table 2 reports the average depth of the leaves of the prefix tree, after the compression. The l' values show that the l value is not crucial in the definition of a PP-Index, given that the only requirement is to choose a l value large enough in order to perform a sufficient differentiation of the indexed objects.

The graph of Figure 5 plots the search time with respect to the size of R and the data set size, for $k = 100$ (single index, single query). For the worst case, with $|R| = 100$, we have measured an average 0.239 seconds search time on the 100M index, with an average of less than eight thousands candidates retrieved from the data storage (see Table 3). The search time decreases in a direct proportion the decrease of the z' value, which follows from the more detailed partitioning of objects into the permutation prefix space, determined by the increase of $|R|$. Even though the z' value increases as D gets larger, the increase of z' is largely sub-proportional to the growth of D: when D grows by a factor 100, z' increases at worst by a factor 6.6.

A possible issue with the z' value is that it is not so close the z value, and it has potentially no limits. However, during the experiments the z' value has never been a critical factor with respect to efficiency: we have not observed any extreme case in which $z' >> z$, and the z' value has never required more than a single read operation from disk to retrieve all the candidate objects (e.g., retrieving 10,000 data blocks from the data storage involves reading only 3.7 MB from disk). We leave to future works an investigation of the relations of the z value with the other parameters.

Figure 6 shows the effectiveness of the PP-Index with respect to the size of the R and the data set size, using a single-index/single-query configuration, for $k = 100$.

Effectiveness values improve with the increase of $|R|$ for the 10M and 100M data sets, while the 1M data set shows the inverse tendency. This confirms the intuition that larger data sets requires a richer permutation prefix space (generated by a larger set R) to better distribute their elements, until a limit is reached and objects became too sparse in the permutation prefix space and the effectiveness worsen.

The maximum-efficiency (0.210 seconds answer time) configuration of PP-Index has obtained a 18.3% recall and 8.1% RDE on the 100M data set, for $k = 100$.

Figures 7 and 8 show respectively the effects on effectiveness of the multiple index and multiple query strategies, for three k values.

With respect to the multiple index strategy we have measured a great improvement on both measures reaching a 74% recall (four times better than the single-index case) and a 0.7% RDE (eleven times better) for the eight index case.

For the above mentioned eight index configuration we have measured an average 1.72 seconds search time, for a completely sequential search process. The four index configuration allows to reach a 52% recall (67% for $k = 10$) and just a 2.2% RDE with a sub-second answer time.

It is relevant to note that, given the small memory occu-

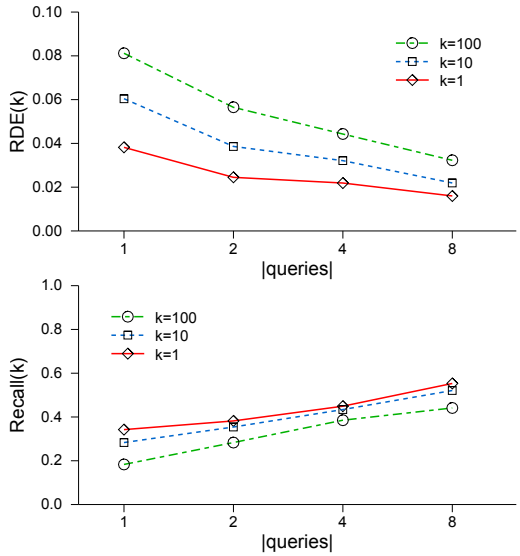


Figure 8: Multiple query search strategy on the 100M index, using $|R| = 1,000$ and $z = 1,000$.

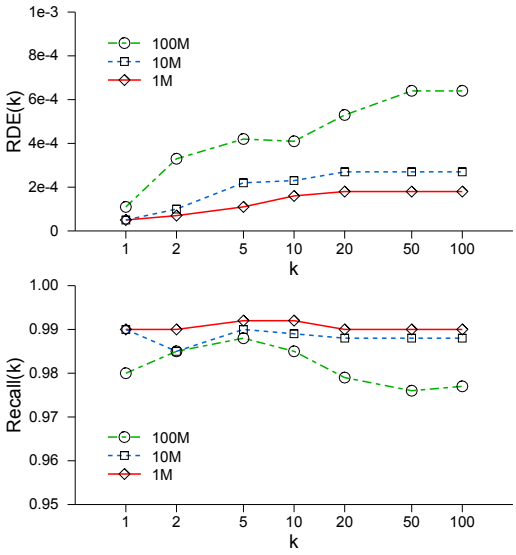


Figure 9: Combined multiple query and multiple index strategies, using eight queries and eight indexes, using $|R| = 1,000$ and $z = 1,000$.

pation of the compressed prefix tree, we have been able to simultaneously load eight 100M indexes into the memory, thus practically performing search on an 800 million objects index, though with replicated data, on a single computer.

The multiple query strategy also shows relevant improvements, though of minor entity with respect to the multiple index strategy. This is in part motivated by the fact that many of the queries, generated by permuting the elements of the original query permutation prefix, actually resulted in retrieving the same candidates of other queries[2]. On the 100M index, for $|R| = 1,000$, on the average, 1.92 distinct queries to be effective in retrieving candidates for the two queries configuration, 3.18 queries for the four queries configuration, and 5.25 queries for the eight queries configuration.

Figure 9 shows the effectiveness of the combined multiple query and multiple index search strategies, using eight queries and eight indexes, for $|R| = 1,000$. We have measured an average search time of 12.45 seconds, for a fully sequential search process. This setup produces almost exact results, with a recall $> 97\%$ and a RDE $< 0.01\%$.

We have measured, on the average, a total of $370,000$ data blocks retrieved from the data storage among the average 44.5 queries being effectively used to access the data storages for each original query. Although this z' value is relatively high, it just represents the 0.3% of the whole collection. This is a very low value considering, for example, that Lv et al. [14], proposing a multiple query strategy for the LSH-Index, have measured a percentage of distance computations with respect to the data set size, in order to obtain a 96% recall, of 4.4% on a 1.3 million objects data set and of 6.3% on a 2.6 million objects data set.

4.4 Comparison experiments

It is a hard task to run comparative experiments on novel and very large data sets, such as CoPhIR due to many reasons: (i) lack of previous results on the same data set; (ii) lack of a publicly available implementation for many of the methods involved in the comparison; (iii) when an implementation is available, it is typically not designed to scale to very large data sets, but just a proof of concept; (iv) moreover, the implementation is usually designed to take in input only a specific data type/format, which makes harder to port the application to different data types.

For this reasons we have currently limited our comparison to replicating two experiments that, among the others, are most closely related to our work: Batko et al. [2], which have run experiments using an early release of the CoPhIR data set, and Amato and Savino [1], whose proposal is the most closely related to our own.

Batko et al. [2] have run experiments on the CoPhIR data set, with data sets size of 10 and 50 millions images[3]. They reports an 80% recall level for 100-NN queries on both collection. For the 50M they test both a 80-CPU infrastructure with 250 GB RAM, keeping all the index data in memory, and a 32-CPU infrastructure with 32 disks storing the index data, obtaining a 0.44 seconds answer time for the memory-based solution and 1.4 seconds for the disk-based one.

The PP-Index could achieve a better performance than [2] by distributing the search process, yet using much less resources than [2]. We have simulated the distribution of the eight indexes on eight distinct computers, each one using two processes executing four queries each, measuring the query answer time as the time of the slowest of the 16 processes plus the time to merge the 16 100-NN intermediate results. We have measured an average 1.02 second answer time to obtain $> 95\%$ recall on the 50M data set.

Amato and Savino [1] test their method on the Corel data set[4]. The data set consists of $50,000$ 32-dimensions color HSV histograms extracted from the images. The distance function used to compare the histograms is L_1.

Replicating [1], we have selected 50 random objects as queries, and indexed the rest of the collection. Given the small size of the data set, we have set $|R| = 50$. The time required for generating the PP-Index is 4.9 second, with a

[2]Such candidates are read only once from the data storage.

[3]We suppose they use the same linear combination of visual descriptors of our experiments, given that two authors are also the authors of [3], from which we take our weights.

[4]`http://kdd.ics.uci.edu/databases/CorelFeatures/CorelFeatures.html`

disk occupation of 13 MB and a memory occupation of 450 kB. In [1] the index structure generated by setting $k_i = 100$ is estimated to require 20 MB. This value does not include the HSV histograms, which are required to reorder the retrieved objects by the true similarity.

The maximum recall level obtained in [1] for $k = 50$ is about 54%, requiring to read 2.4 MB of data from disk (600 blocks of 4 kB size). The PP-Index, in a single-index/four-query configuration ($z = 500$), obtains a 89.6% recall, requiring to read just 900 kB of data from disk, in four sequential reads. The single-index/single-query configuration obtains a 66% recall.

5. CONCLUSIONS

We have introduced the PP-Index, an approximate similarity search data structure based on the use of short permutation prefixes. We have described the PP-Index data structures and algorithms, including a number of optimization methods and search strategies aimed at improving the scalability of the index, its efficiency, and its effectiveness.

The PP-Index has been designed to take advantage of the relatively static nature one could expect from very large collections. However, as we have described, it is easy to support fast update operations.

We have evaluated the PP-Index on a very large and high-dimensional data set. Results show that it is both efficient and effective in performing similarity search, and it scales well to very large data sets.

We have shown how a limited-resources configuration obtains good effectiveness results in less than a second, and how almost exact results are produced in a relatively short amount of time. The parallel processing capabilities of the PP-Index allow to distribute the search process in order to further improve its efficiency.

The comparison with experimental results published for two closely related method, which are among the top-performers on the task, shows that the PP-Index outperforms the compared methods, both in efficiency and effectiveness. Only one [2] of the works we compare with uses a data set of a size comparable to our largest one. We plan to extend the comparison with some of the competing methods, by porting them on the larger data set sizes.

The PP-Index has been already used to build a performing similarity search system[5] [10].

There are many aspect of our proposal that are worth to be further investigated. For example, the R set is a crucial element of the PP-Index. We plan to study element selection policies alternative to the random policy, e.g., selecting centroids of clusters of D, or the most frequent queries from a query log.

6. REFERENCES

[1] G. Amato and P. Savino. Approximate similarity search in metric spaces using inverted files. In *INFOSCALE '08: Proceeding of the 3rd International ICST Conference on Scalable Information Systems*, pages 1–10, Vico Equense, Italy, 2008.

[2] M. Batko, F. Falchi, C. Lucchese, D. Novak, R. Perego, F. Rabitti, J. Sedmidubský, and P. Zezula. Crawling, indexing, and similarity searching images on the web. In *Proceedings of SEDB '08, the 16th Italian Symposium on Advanced Database Systems*, pages 382–389, Mondello, Italy, 2008.

[5] `http://mipai.esuli.it/`

[3] M. Batko, P. Kohoutkova, and P. Zezula. Combining metric features in large collections. *SISAP '08, 1st International Workshop on Similarity Search and Applications*, pages 79–86, 2008.

[4] M. Bawa, T. Condie, and P. Ganesan. Lsh forest: self-tuning indexes for similarity search. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 651–660, Chiba, Japan, 2005.

[5] P. Bolettieri, A. Esuli, F. Falchi, C. Lucchese, R. Perego, T. Piccioli, and F. Rabitti. CoPhIR: a test collection for content-based image retrieval. *CoRR*, abs/0905.4627, 2009.

[6] E. Chávez, K. Figueroa, and G. Navarro. Effective proximity retrieval by ordering permutations. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 30(9):1647–1658, 2008.

[7] E. Chavez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Proximity searching in metrix spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.

[8] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 426–435, Athens, Greece, 1997.

[9] P. Diaconis. Group representation in probability and statistics. *IMS Lecture Series*, 11, 1988.

[10] A. Esuli. MiPai: using the PP-Index to build an efficient and scalable similarity search system. In *SISAP '09, Proceedings of the 2nd International Workshop on Similarity Search and Applications*, Prague, CZ, 2009.

[11] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC '98: Proceedings of the 30th ACM symposium on Theory of computing*, pages 604–613, Dallas, USA, 1998.

[12] H. V. Jagadish, A. O. Mendelzon, and T. Milo. Similarity-based queries. In *PODS '95: Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 36–45, San Jose, US, 1995.

[13] D. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching, chapter 5.4: External Sorting, pages 248–379. second edition, 1998.

[14] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 950–961, Vienna, Austria, 2007.

[15] D. R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[16] MPEG-7. Mpeg-7. multimedia content description interfaces, part3: Visual, 2002. ISO/IEC 15938-3:2002.

[17] M. Patella and P. Ciaccia. The many facets of approximate similarity search. *SISAP '08, 1st International Workshop on Similarity Search and Applications*, pages 10–21, 2008.

[18] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach (Advances in Database Systems)*. 2005.

Static Index Pruning for Information Retrieval Systems: A Posting-Based Approach

Linh Thai Nguyen
Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616 USA
+1-312-567-5330
nguylin@iit.edu

ABSTRACT

Static index pruning methods have been proposed to reduce size of the inverted index of information retrieval systems. The goal is to increase efficiency (in terms of query response time) while preserving effectiveness (in terms of ranking quality). Current state-of-the-art approaches include the term-centric pruning approach and the document-centric pruning approach. While the term-centric pruning considers each inverted list independently and removes less important postings from each inverted list, the document-centric approach considers each document independently and removes less important terms from each document. In other words, the term-centric approach does not consider the relative importance of a posting in comparison with others in the same document, and the document-centric approach does not consider the relative importance of a posting in comparison with others in the same inverted list. The consequence is less important postings are not pruned in some situations, and important postings are pruned in some other situations. We propose a posting-based pruning approach, which is a generalization of both the term-centric and document-centric approaches. This approach ranks all postings and keeps only a subset of top ranked ones. The rank of a posting depends on several factors, such as its rank in its inverted list, its rank in its document, its weighting score, the term weight and the document weight. The effectiveness of our approach is verified by experiments using TREC queries and TREC datasets.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval – *Index pruning, Search process.*

General Terms

Algorithms, Performance, Experimentation.

Keywords

Static Index Pruning, Document-centric Index Pruning, Term-centric Index Pruning, Posting-centric Index Pruning.

LSDS-IR Workshop. July 2009. Boston, USA.

1. INTRODUCTION

Text information retrieval systems are based on an inverted index to efficiently process queries. The most important part of an inverted index is its inverted file, a file that contains posting list for each term in the text collection [1]. In general, a posting list of a term contains its posting entries (or index pointers), each in the form of <*docID*, *freq*>, where *docID* is the ID of a document that contains the term, and *freq* is its frequency in the document. For a multi-keyword query, all posting lists of query terms are retrieved from the inverted file, and document scores are accumulated for each document in the union of the posting lists, based on a specified weighting scheme. A list of documents in descending order of rank scores is presented to the user.

For a large text corpus, the inverted file is too large to fit into memory of the search server. Thus, query processing involves a lot of disk access, which increases query response time. For a text information retrieval system that has to process thousands of queries per second, it is critical to improve query processing performance.

Beside the parallel query processing approach that uses a cluster of servers to process queries, the index compression approach is widely used. The lossless compression approach uses data compression techniques to compress index data, thereby reducing the volume of data transferred from disk. The compressed index data is then decompressed in memory, and queries are processed based on the original index information. Common data compression technique used in information retrieval systems is variable length data coding [2]. In contrast, lossy compression approach opts for keeping only important information in the index, discarding other less important information [4][5][6][8] [11]. Thus ranking quality of queries processed based on a lossy compressed index (i.e. a pruned index) might be affected.

In practice, a lossless compression technique can be applied on a lossy pruned index to further reduce index size. In addition, both types of compressed/pruned index can be used by an information retrieval system: a lossy pruned index is used to answer a large portion of user queries, and a lossless compressed index is used only if result quality is significantly hurt [4].

In this work, we concentrate on lossy index compression. Current state-of-the-art approaches include term-centric pruning [5] and document-centric pruning [6]. While term-centric pruning method considers each inverted list independently and removes less important postings from each inverted list, document-centric

pruning considers each document independently and removes less important terms from each document. In other words, the term-centric method does not consider the relative importance of a posting in comparison with others in the same document, and document-centric method does not consider the relative importance of a posting in comparison with others in the same inverted list. The consequence is less important postings are not pruned in some situations, and important postings are pruned in some other situations.

We propose a posting-based pruning approach, which is a generalization of both the term-centric and document-centric approaches. Our approach ranks all postings and keeps only a subset of the top ranked ones, removing the others. We consider a couple of factors when ranking a posting, such as its rank in its posting list, its rank in its document, its weighting score, the normalized weight of the term, and the normalized weight of the document. Our experiments based on TREC queries and TREC datasets [22] show that posting-based pruning method outperforms both the term-centric and document-centric methods.

2. RELATED WORK

Lossless index compression techniques are well studied, for example, see Witten et al. [2][3]. Those techniques are mainly based on the fact that the frequencies of terms in documents, which are stored in the inverted index, follow a skewed distribution. In that case, variable length coding technique can be used to encode index information, consuming only a few bits for most of term frequency values. In general, this helps to reduce index size by about one-tenth. However, for large scale information retrieval systems, the compressed index is still too big to fit into memory. In addition, using a compressed index reduces time to access index data from disk, but does not reduce time to process the posting lists. Thus, using lossless index compression alone cannot significantly improve efficiency.

Lossy index compression techniques opt for discarding postings that are not informative. By removing a large number of postings from the inverted index, lossy index compression techniques not only significantly reduce index size, but also significantly reduce length of posting lists. Therefore, lossy index compression techniques can reduce both time to access index data from disk and time to process posting lists. However, as some index information is lost, lossy index compression techniques may lead to a drop in query ranking quality.

In [5], Carmel et al. introduced a term-centric approach to static index pruning. Entries in each posting list are sorted in descending order of a weighting scores. Only entries whose weighting scores are greater than a threshold value are kept in the pruned index. The threshold value can be the same for all terms (uniform pruning), or it can be different for each term (term-based pruning).

Buttcher and Clarke introduce a document-centric pruning technique [6]. Instead of posting list pruning, they propose document pruning. For each document, they keep only a small number of representative, highly-ranked terms in the pruned index. Terms in each document are ranked based on their contribution to the Kullback-Leibler divergence [16] between the document and the text collection. The intuition behind this is that those document-representative terms are powerful enough to distinguish the document from others. They also show experimentally that if the document is ranked high for a given query, it is very likely that query terms are among its representative terms. Thus indexing sets of representative terms is a good method to preserve ranking quality while reducing index size.

Other index pruning techniques (some are for distributed, peer-to-peer information retrieval systems) belong to either the term-centric approach or document-centric approach. Blanco et al. [8], and Shokouhi et al. [10], try to find terms whose posting lists can be completely removed. De Moura et al. [11] propose to index a set of representative sentences for each document. Lu and Callan [7] propose a number of methods to identify a representative term set for each document. Podna et al. [12] and Skobeltsyn et al. [13] propose to index term combinations to reduce the negative effect of posting list pruning to ranking quality. Blanco and Barreiro [9] improve the precision of term-centric pruning by considering a number of designs overlooked by the original work.

Looking at other aspect of static index pruning, Skobeltsyn et al. [14] point out that the use of results caching fundamentally affects the performance of a pruned index, due to the change in query pattern introduced by results caching. They then propose to combine results caching and index pruning to reduce the query workload of back-end servers.

3. TERM-CENTRIC PRUNING VERSUS DOCUMENT-CENTRIC PRUNING

3.1 Term-Centric Index Pruning

Term-centric pruning fits very well with the inverted index structure of information retrieval systems. As queries are processed based on inverted lists, it is natural to truncate inverted lists in order to reduce index size. Based on the inverted index structure, the "idealized, term-based" pruning technique proposed by Carmel et al. is well-formed and mathematically provable. This clearly shows that a pruned index, even though not containing all information, still can guarantee the ranking quality to some extent [5].

There are several properties that are specific to term-centric pruning. It preserves the collection vocabulary. For every term, there are always some entries in its inverted list in the pruned index. (The works of Blanco et al. [8] and Shokouhi et al. [10] are exceptions, as their work reduces the vocabulary size.) In contrast, term-centric pruning does not necessarily preserve the set of documents. As posting entries are removed, it is possible that some documents will be totally removed from the pruned index.

The fact that term-centric index pruning preserves the set of terms demonstrates its support for the possibility of all terms appearing in user queries. Due to this support, in order to guarantee the quality of top-K results for any queries, term-centric pruning must not prune any of the top-K entries in any posting list. Obviously, pruning any of these makes the pruned index unable to guarantee the top-K results of the query containing only that single term. In addition, term-centric pruning assumes (implicitly) that every term in the vocabulary is equally important. In contrast, for documents, term-centric pruning assumes that some are more important than others. This is inferred from the fact that term-centric pruning might totally removed some documents from the pruned index.

3.2 Document-Centric Data Pruning

Document-centric pruning does not make any assumption about the index structure and how queries are processed. Precisely, document-centric pruning should be considered as a "data" pruning technique instead of an index pruning technique, as what it actually does is to prune the documents, not an index structure.

In contrast to term-centric pruning, document-centric pruning preserves the set of documents, not the set of terms. While any document in the collection is represented by a subset of its terms (i.e., its set of representative terms), there is no guarantee that every term will be indexed. It is likely that there are terms that are always ranked low in any document and are removed by document-centric pruning.

The first assumption implied by document-centric pruning is that every document can be ranked first by some queries (one such query might be the query that contains all its representative terms). Due to this assumption, document-centric pruning opts for including every document in the pruned index. The second implied assumption is that terms are not equally important, and some terms can be totally removed from the pruned index.

4. POSTING-BASED INDEX PRUNING

As pointed out above, term-centric pruning prunes index elements, which are posting lists; while document-centric pruning prunes data elements, which are documents. Both approaches assume that all elements are equally important, and thus the pruned index should keep some information about every element, either they are posting lists or documents.

We first find that the decision to keep some amount of information for each posting list or document to be reasonable. Without any information about the user queries, we must assume any term can be used by users. Thus no term can be totally removed. Similarly, without any information about what users will search for, we also have to assume any document can be an answer (to some queries). Therefore, no document can be totally removed.

However, given the requirement of significantly reducing index size, it is not affordable to keep information for all posting lists and all documents. We believe that the pruned index should contain neither all terms, nor all documents, but only the most important postings, given the desired pruning level.

We suspect that non-informative terms are common in any large text collection. Non-informative terms are those terms that do not help to discriminate documents. One example of non-informative terms is a term that appears in every document, such as the term "abstract" in a collection of scientific papers. Those terms are expected to have similar weighting scores to every document. Therefore, eliminating those terms will not hurt ranking quality. Unfortunately, term-centric pruning tends not to prune any entries from the posting lists of those terms. The reason is, as entry scores are almost similar, all scores are likely to be greater than the threshold value computed by the term-based method proposed in [5].

We also suspect that there are many "rarely asked for" documents in any large text collection. A document is called "rarely asked for" if it does not appear in the top-ranked results of any real world query. In practice, users normally look at only the top 20 results, so any document that does not appear in the top-20 results of a large number of queries can be removed. Puppin et al. [17] observed that, for a collection of 5,939,061 documents and a set of 190,000 unique queries, around 52% of the documents were not returned among the first 100 top-ranked results of any query.

We propose a posting-based index pruning method. We choose neither posting lists, nor documents as our working elements. Instead, we choose postings, i.e. tuples of the form <term ID, document ID, term frequency>. Postings are contained in both index elements (as posting entries in posting lists) and data elements (as terms in documents). Also, choosing posting entries as working elements, we open the possibility of removing any document and any term's posting list from the pruned index. With this flexibility, our method is able to remove non-informative terms as well as "rarely asked for" documents.

4.1 Term Weighting

When building a pruned index, terms should not be treated equally. Non-informative terms appear in a large number of documents, results in long posting lists. However, non-informative terms do not help much in ranking documents. Thus, the pruned index should significantly prune the posting lists of non-informative terms and reserve places for other informative terms. Our posting-based pruning method assigns a weight to each term as its informativeness value. Blanco and Barreiro [8] have studied a number of term-weighting schemes for the purpose of posting list pruning. Their finding is that residual inverse document frequency (RIDF) is a good quantity to measure the informativeness of terms, among other schemes such as the classic inverse document frequency and the term discriminative value. We adopt RIDF to calculate term informativeness values. As specified in [8], the RIDF value of a term is

$$RIDF = -\log\left(\frac{df}{N}\right) + \log\left(1 - e^{-\frac{tf}{N}}\right) \quad (1)$$

where *df* is the term's document frequency and *N* is the number of documents in the collection. As pointed out by Blanco, RIDF values can be computed efficiently.

4.2 Document Weighting

Documents in a large text collection are not equally important and therefore, in the pruned index, more terms should be kept for important documents. As for term weighting, we also assign a weight to each document in the collection to reflect how important it is. For a Web collection, the PageRank [18] or HITS [19] algorithm can be used to compute document important values. However, PageRank and HITS are not applicable for non-Web documents, as there is no link structure among documents. We adopt the approach of Buttcher and Clarke [6] for this purpose. For each document, we assign its Kullback-Leibler distance to the collection as its important value. Thus, "out standing" documents (i.e., documents which are very different from the collection) will be assigned high important values, while documents which are similar to others will be assigned low important values. Our important value for a document *d* is defined as

$$KLD(d \,\|\, C) = \sum_{t \in d} \frac{tf(t)}{|d|} \log\left(\frac{tf(t)}{|d|} \times \frac{|C|}{TF(t)}\right) \quad (2)$$

where $tf(t)$ is the term frequency of term *t*, $TF(t)$ is the collection term frequency of term *t*, $|d|$ is the length of document *d* (i.e., the

number of terms in d), and $|C|$ is the length of the collection (i.e., the sum of document lengths).

4.3 Posting Ranking Function

Our static pruning method evaluates postings and assigns each a usefulness value. We then build the pruned index based on these values. Given a desired level of pruning, posting entries are selected based on their usefulness values and added into the pruned index until the pruned index size reaches its limit.

According to term-centric pruning, we should assign a high usefulness value to a posting entry which appears at the top of its posting list. According to document-centric pruning, we should not assign a high usefulness value to a posting whose term does not belong to the set of top-ranked terms of the document. In addition, as discussed above, we should assign low values to non-informative terms and "rarely asked for" documents, and vice versa. Also, we obviously want to assign high usefulness value to posting entries with high scores.

To rank postings, for each posting $<t, d>$, we compute the following quantities:

- $S(t, d)$ = the score term t contributes to the rank score of document d.
- $RIDF(t)$ = the informativeness value of term t.
- $KLD(d \parallel C)$ = the important value of document d.
- $Rank(t)$ = the rank of term t relatively with other terms in document d.
- $Rank(d)$ = the rank of document d relatively with other documents in the posting list of term t.

Among the quantities above, $S(t, d)$ is computed using a weighting scheme, such as the classic TFIDF weighting scheme, or the state-of-the-art BM25 weighting scheme; $RIDF(t)$ is calculated as specified in (1); $KLD(d \parallel C)$ is calculated according to (2); $Rank(d)$ is the position of document d in the posting list of term t, where posting entries are sorted in descending order of its scores $S(t,d_i)$; and $Rank(t)$ is the position of term t in document d, where terms are sorted in descending order of its "feedback" score [6], defined below:

$$Score_{DCP}(t) = \left(\frac{tf}{|d|}\right)\log\left(\frac{tf}{|d|}\times\frac{|C|}{TF}\right) \qquad (3)$$

In this work, we use the BM25 weighting scheme [20] (given below) to calculate $S(t, d)$ due to its widely use in other research works.

$$S_{BM25}(t,d) = \log\left(\frac{N-df+0.5}{df+0.5}\right)\times\frac{tf(k_1+1)}{tf+k_1\left((1-b)+b\frac{|d|}{avgdl}\right)} \qquad (4)$$

where $avgdl$ is the average length of documents, k_1 is set to its "standard" value of 1.2 and b is set to its "standard" value of 0.75.

In combination, our posting entry ranking function takes as parameters all the above quantities and returns a usefulness value. Note that we apply normalization and transformation to parameter values. First, RIDF(t) values are normalized so that they sum up to one. Similar normalization is applied to KLD($d \parallel C$) values. Normalization step is necessary, as the range of RIDF(t) and KLD($d \parallel C$) are different. Second, we use a sigmoid function to transform the term rank values and document rank values. This transformation is necessary, too. Using the term rank values makes it appears that the 10th term is ten time less important than the top ranked term, which does not seem right. Therefore, we use a non-linear function specified below (5) as a transform function. The parameter x_0 is used to shift the "transition" point, where the sigmoid function switches from high value state to low value state, and the parameter a is used to control the slope of the transition period of the function.

$$sigmoid(x) = 1 - \frac{1}{1+e^{(-x+x0)/a}} \qquad (5)$$

The sigmoid function above returns a value between zero and one. Rank value close to zero (i.e., top ranked element) will be transformed to a value close to one, while other rank values will be transformed depend on two parameters x_0 and a. In Figure 1, we show the shape of the sigmoid function for several combinations of parameters.

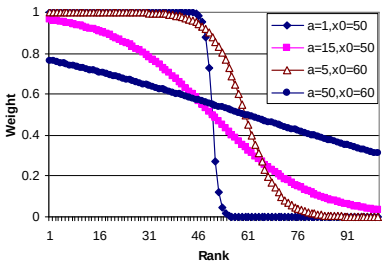


Figure 1. Sigmoid functions.

Our posting entry ranking function is given below:

$$f(\langle t,d\rangle) = S_{BM25}(t,d)\cdot\{\alpha\cdot RIDF(t)\cdot sigmoid(Rank(d)) + (1-\alpha)\cdot KLD(d\parallel C)\cdot sigmoid(Rank(t))\} \qquad (6)$$

where α is a parameter taking value between zero and one.

4.4 Posting-Based Pruning versus Document-Centric Pruning and Term-Centric Pruning

We show that our posting entry ranking function generalizes document-centric index pruning method and term-centric pruning method.

If we set the value of α to zero, replace the document weighting function $KLD(d\|C)$ with the unity function $u(d) = 1$, and set the parameter a of the sigmoid function to 1 so that the sigmoid function in (5) becomes a threshold function at x_0, then for each document d, our ranking function $f()$ assigns a non-zero value to the posting entry $<t, d>$ if t is ranked in the top-x_0 among all unique terms in d, otherwise $f()$ returns a zero value. In this case, our posting-based pruning technique is equivalent to the "constant" document-centric pruning technique proposed by Buttcher and Clarke in [6], which select a fixed number of top ranked terms from each document. Obviously, the "relative" document-centric pruning technique proposed in [6] can be easily obtained from our posting entry ranking function by adjusting x_0 for each document according to its length.

Similarly, if we set the value of α to one, replace the term weighting function $RIDF(t)$ with the unity function $u(t) = 1$, set the parameter a of the sigmoid function to 1, and set the parameter x_0 to the rank of the i-th posting entry in the posting list of term t

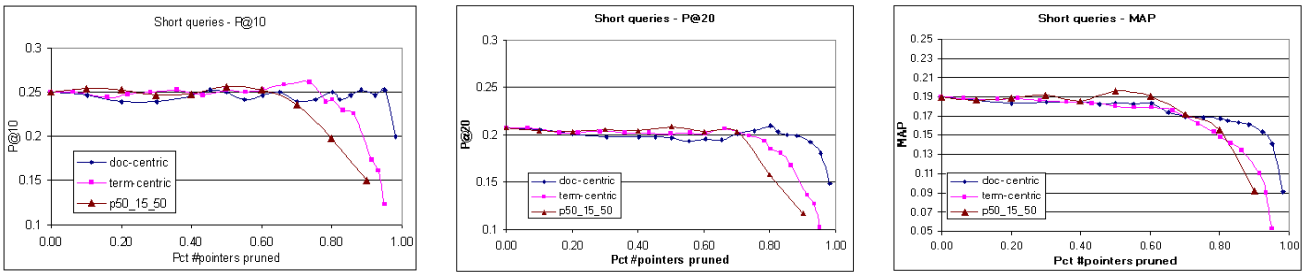


Figure 2: Querying effectiveness for short TREC queries.

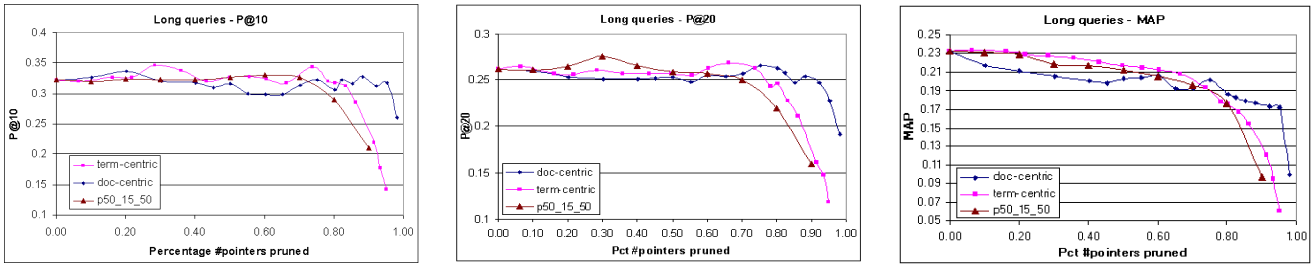


Figure 3: Querying effectiveness for long TREC queries.

such that $S(t, d_i) > \tau(t)$ and $S(t, d_{i+1}) \leq \tau(t)$, where $\tau(t)$ is the cut-off threshold proposed by Carmel et al. in [5], then our posting entry ranking function assigns a non-zero value to any posting entry selected by term-centric pruning technique, and a zero value to any non-selected posting entry.

Our posting-based pruning technique is different from term-centric and document-centric pruning techniques in several aspects. By using term weighting function and document weighting function other than the unity function, we allow the pruned index to include more information for informative terms and outstanding documents. By using a sigmoid function instead of a threshold function to transform the term and document ranks, we open the possibility that a posting entry which is ranked low in a posting list could be selected, if it is ranked high in the document, and vice versa.

5. EXPERIMENTAL SETTINGS

We use the WT10G collection as our data set. This collection contains 1,692,096 Web documents crawled from the Web. We use the Terrier platform [21] to index and rank queries, and we develop our posting-based index pruning technique based on Terrier.

We use the BM25 weighting scheme for both calculating term-document scores and query ranking. Potter stemming algorithm [23] and a standard list of stop-words are used for preprocessing documents. After stemming and stop-word removal, the vocabulary contains 3,161,488 unique terms. The un-pruned index contains 280,632,807 posting entries.

We use TREC [22] topics 451–500 as our query set. Precision is measured for each pruned index using the set of relevance judgment provided by TREC for topics 451–500. From TREC topics 451–500, we build two sets of queries: one set of long queries, wherein each query includes the title and the description fields from the TREC topic, and one set of short queries, wherein each query includes only the title field from the TREC topic.

We also implement term-centric index pruning and document-centric index pruning exactly as specified in their original works [5][6]. The only difference is that in [5], Carmel et al. used the SMART term weighting scheme for both index pruning and query ranking. We instead use BM25 term weighting scheme for query ranking, but still use SMART term weighting scheme for index pruning.

We conduct experiments to compare the effectiveness of our proposed posting-based pruning technique with the term-based, "score shifting" pruning technique proposed in [5], and the "relative" document-centric pruning technique proposed in [6]. In the next section, we report precision at 10, precision at 20, and average precision at each pruning level for each technique.

6. EXPERIMENTAL RESULTS

In Figure 2, we show the effectiveness of pruned indices for the set of short TREC queries; and in Figure 3, we show the effectiveness of pruned indices for the set of long TREC queries. Posting-centric pruned index is marked as "pXX_YY_ZZ", where XX is the value of parameter α (percentage), YY is the value of parameter a, and ZZ is the value of parameter x_0. Figure 2 and Figure 3 show experiment results for a posting-centric pruned index with α = 50%, a = 15, and x_0 = 50. With the pruning level less than 70%, posting-centric pruning has similar performance as compare with document-centric pruning and term-centric pruning. However, for pruning level of 70% or more, posting-centric pruning is outperformed by document-centric pruning.

We turn our parameters for posting-centric index pruning technique. Table 1, Table 2, and Table 3 show experiment results for short queries of document-centric pruning technique and posting-centric pruning techniques with various combinations of

Table 1. Precision at 10 for short TREC queries of document-centric pruning technique and posting-centric pruning techniques of various parameter combinations.

Pct #posting entry-pruned	Document-centric	p50_15_50	p20_15_50	p50_35_200	p50_200_50	p80_200_1000	p50_300_1000	p80_300_1000	p80_400_1000
0	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
10	0.2458	0.2542	0.2542	0.2542	0.2542	0.2542	0.2542	**0.2563**	0.2521
20	0.2396	**0.2521**	0.2521	0.2542	0.2375	0.2333	0.2333	0.2354	0.2354
30	0.2396	**0.2458**	0.2458	0.2458	0.2292	0.2271	0.2292	0.2313	0.2292
40	0.2458	**0.2479**	0.2479	0.2271	0.2250	0.2271	0.2271	0.2250	0.2333
50	0.2500	**0.2563**	0.2542	0.2375	0.2292	0.2250	0.2312	0.2312	0.2208
60	0.2458	**0.2521**	0.2500	0.2250	0.2208	0.2104	0.2208	0.2167	0.2146
70	**0.2396**	0.2354	0.2354	0.2271	0.2396	0.2229	0.2354	0.2187	0.2187
80	**0.2500**	0.1979	0.2167	0.2021	0.2104	0.2063	0.2333	0.1979	0.2021
90	**0.2458**	0.15	0.1625	0.1625	0.1854	0.1958	0.1875	0.2021	0.2000

Table 2. Precision at 20 for short TREC queries of document-centric pruning technique and posting-centric pruning techniques of various parameter combinations.

Pct #posting entry-pruned	Document-centric	p50_15_50	p20_15_50	p50_35_200	p50_200_50	p80_200_1000	p50_300_1000	p80_300_1000	p80_400_1000
0	0.2073	0.2073	0.2073	0.2073	0.2073	0.2073	0.2073	0.2073	0.2073
10	**0.2052**	0.2042	0.2042	0.2052	0.2052	0.2052	0.2052	**0.2052**	**0.2052**
20	0.201	**0.2031**	0.2031	0.2021	0.1990	0.1938	0.1948	0.1948	0.1958
30	0.1979	**0.2052**	0.2063	0.1990	0.1854	0.1844	0.1854	0.1854	0.1844
40	0.1979	**0.2042**	0.2042	0.1917	0.1750	0.1740	0.1740	0.1750	0.1781
50	0.1969	**0.2083**	0.2073	0.1875	0.1771	0.1740	0.1781	0.1792	0.1719
60	0.1958	**0.2031**	0.2010	0.1802	0.1813	0.1813	0.1844	0.1813	0.1833
70	0.2010	**0.2031**	0.1990	0.1719	0.1802	0.1740	0.1802	0.1750	0.1750
80	**0.2094**	0.1583	0.1729	0.1573	0.1552	0.1635	0.1729	0.1635	0.1646
90	**0.1927**	0.1177	0.1229	0.1208	0.1354	0.1469	0.1448	0.1531	0.1500

Table 3. MAP for short TREC queries of document-centric pruning technique and posting-centric pruning techniques of various parameter combinations.

Pct #posting entry-pruned	Document-centric	p50_15_50	p20_15_50	p50_35_200	p50_200_50	p80_200_1000	p50_300_1000	p80_300_1000	p80_400_1000
0	0.1892	0.1892	0.1892	0.1892	0.1892	0.1892	0.1892	0.1892	0.1892
10	0.1864	0.1868	0.1871	0.1869	0.1879	0.1875	0.1878	0.1873	**0.1880**
20	0.1838	**0.1891**	0.1892	0.1889	0.1835	0.1818	0.1824	0.1835	0.1826
30	0.1846	**0.1914**	0.1913	0.1901	0.1816	0.1806	0.1807	0.1821	0.1825
40	0.1847	**0.1855**	0.1854	0.1880	0.1805	0.1822	0.1820	0.1827	0.1845
50	0.1837	**0.1958**	0.1958	0.1815	0.1834	0.1809	0.1839	0.1840	0.1809
60	0.1835	**0.1911**	0.1915	0.1804	0.1755	0.1743	0.1785	0.1747	0.1747
70	0.1693	**0.172**	0.1739	0.1698	0.1702	0.1619	0.1657	0.1692	0.1627
80	**0.1679**	0.1557	0.1571	0.1476	0.1373	0.1494	0.1575	0.1486	0.1440
90	**0.1533**	0.0919	0.1136	0.1238	0.1338	0.1404	0.1314	0.1378	0.1421

parameter values. Experiments with long queries have similar trend and therefore are omitted.

In Table 1, Table 2, and Table 3, the values in bold are the highest performance for a specific pruning level. The first observation is that posting-centric pruning is better at low and moderate pruning level, while document-centric pruning is better at higher pruning level. The second observation is that, even though posting-centric pruning is better than document-centric pruning at low and moderate pruning level, the differences are small. In contrast, at higher pruning level, the differences between the performance of posting-centric pruning and document-centric pruning are larger. In addition, none of the posting-centric pruning techniques outperforms document-centric pruning technique at high pruning level.

In all previous experiments of posting-centric pruning technique, parameters of the posting entry ranking function (6) are fixed for all posting entries. We consider the possibility of adaptively setting parameters for each posting entry, by adapting the slope of

Table 4. Performance at 90% pruning level of different posting-centric pruning techniques.

Pruning method	Short TREC queries			Long TREC queries		
	P@10	P@20	MAP	P@10	P@20	MAP
Document-centric	0.2458	0.1927	0.1533	0.3120	0.2470	**0.1731**
AP1: no term weighting, no document weighting, $\alpha = 0.5$, using two different sigmoid functions with parameters vary for each posting entry	0.2375	0.1688	0.1456	0.3020	0.2100	0.1561
AP2: similar to AP1, except that term-document scores are not used	0.1277	0.1106	0.0831	0.1660	0.1250	0.0903
AP3: similar to AP1, except that term weighting is used	**0.2604**	**0.1969**	**0.1592**	**0.3300**	**0.2500**	0.1714
AP4: similar to AP1, except that both term weighting and document weighting are used	0.2271	0.1573	0.1354	0.2740	0.1900	0.1479

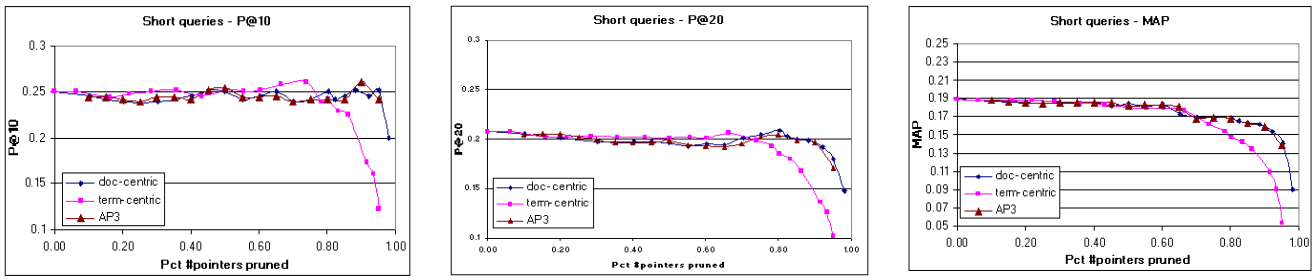


Figure 4: Querying effectiveness for short TREC queries.

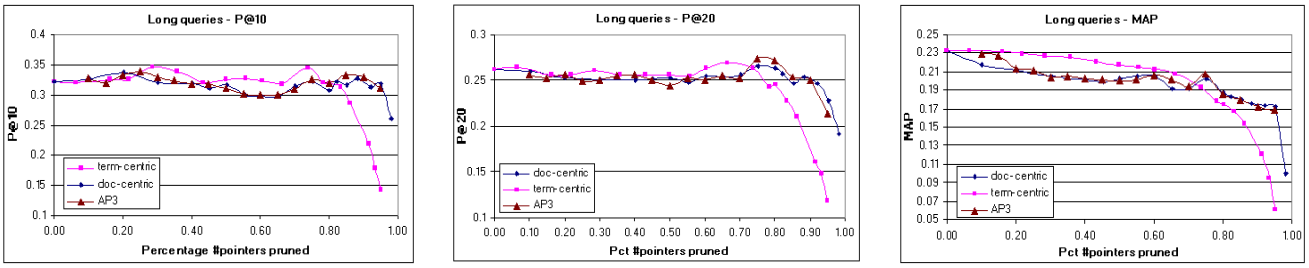


Figure 5: Querying effectiveness for long TREC queries.

the sigmoid function as follows. For a posting entry <t, d>, we use two sigmoid functions, one for the rank of term t in the document d, the other for the rank of document d in the posting list of term t. We call the former document-oriented sigmoid function, and the later term-oriented sigmoid function. For the term-oriented sigmoid function, its x_0 parameter is set according to the pruning level (i.e., if the pruning level is 90%, x_0 is equal to 10% of the length of the term posting list). Similarly, for the document-oriented sigmoid function, if the pruning level is 90%, its x_0 parameter is set to 10% of the number of unique terms in the document. For both sigmoid functions, the parameter a, which control the slope of the function, is set so that the function returns a value close to 1 for input value which is less than $0.9 \times x_0$ and returns a value close to 0 for input value which is greater than $1.1 \times x_0$. We also explore the performance of several alternatives, which may or may not use term-weighting and/or document-weighting (refer to the ranking function (6) in Section 4.3).

In Table 4, we report the performance of different alternatives at the approximately 90% pruning level. Document-centric pruning performance is included for reference. For each column, the best value is in bold. From the results reported in Table 4, we can see that:

(i) Term-document scores should be used in the posting entry ranking function (6), which is revealed by the significant differences between the performance of AP1 and AP2.

(ii) Term-weighting is useful, which is confirmed by the differences between the performance of AP1 and AP3.

(iii) Document weighting seems to be harmful, which hurt the performance of AP4, compare to the performance of AP3 and AP1.

Among all alternatives, AP3 is the best, which is only slightly outperformed by document-centric pruning for long queries according to MAP. We therefore consider it as the best among our alternatives. Below, we report its performance in comparison with document-centric pruning and term-centric pruning at various level of pruning in Figure 4 and Figure 5.

By adapting the sigmoid functions to posting entries, the performance of our posting-centric pruning technique is much better, as good as the performance of document-centric pruning

technique (posting-centric pruning is better than document-centric pruning at some pruning level, while the inverse is true at other pruning levels).

7. CONCLUSIONS AND FUTURE WORK

We evaluate document-centric and term-centric static index pruning based on the WT10G corpus and TREC query sets. Based on our experimental results, term-centric index pruning is better than document-centric index pruning at low and moderate pruning level (i.e., less than 70% pruning, according to our results), while document-centric index pruning is better at higher pruning level.

We propose posting-centric index pruning technique, which ranks each posting entry (i.e., a term-document pair) based on a set of features such as the rank of the term in the document and the rank of the document in the inverted list of the term. We show that posting-centric index pruning generalizes both document-centric and term-centric pruning, and therefore, the solution space of term-centric pruning covers the solution spaces of both document-centric and term-centric index pruning. This implies that, by exploring this larger solution space, better solution can be found.

We explore the solution space of posting-centric pruning by studying a family of posting entry ranking functions. We discover that term weighting based on RIDF is useful, while document weighting based on KL-divergence is harmful. We also notice that parameters of the sigmoid function, which we use to transform the rank of a term/document to its score, should be adapted to each posting entry. Fixing these parameters makes posting-centric pruning less effective than document-centric pruning.

Other term weighting and document weighting methods are possible. We are evaluating a method of weighting terms and documents based on user queries and the PageRank algorithm applying on the graph of terms and documents. Our goal is to discover important terms and documents by analyzing the relationship among terms and documents given the context of user queries. Once the important terms and the important documents are discovered, their information is kept in the pruned index, while information about others, less important terms and documents, can be partially removed or totally discarded.

8. REFERENCES

[1] D. Grossman and O. Frieder, "Information Retrieval: Algorithms and Heuristics," Springer, 2nd ed, 2004.

[2] I. H. Witten, A. Moffat, and T. C. Bell, "Managing Gigabytes," Morgan Kaufmann, 2nd ed, 1999.

[3] J. Zobel and A. Moffat, "Inverted Files for Text Search Engines," in ACM Computing Surveys, 38(2), 2006.

[4] A. Ntoulas and J. Cho, "Pruning Policies for Two-Tiered Inverted Index with Correctness Guarantee," in Proc. ACM SIGIR, 2007.

[5] D. Carmel et al., "Static Index Pruning for Information Retrieval Systems," in Proc. ACM SIGIR, 2001.

[6] S. Buttcher and C. L. A. Clarke, "A Document-Centric Approach to Static Index Pruning in Text Retrieval Systems," in Proc. ACM CIKM, 2006.

[7] J. Lu and J. Callan, "Pruning Long Documents for Distributed Information Retrieval," in Proc. ACM CIKM, 2002.

[8] R. Blanco and A. Barreiro, "Static Pruning of Terms in Inverted Files," in Proc. ECIR, 2007.

[9] R. Blanco and A. Barreiro, "Boosting Static Pruning of Inverted Files," in Proc. ACM SIGIR, 2007.

[10] M. Shokouhi et al., "Using Query Logs to Establish Vocabularies in Distributed Information Retrieval," In Int'l Journal on Information Processing and Management, 2007.

[11] E. S. de Moura et al, "Improving Web Search Efficiency via a Locality Based Static Pruning Method," in Proc. ACM WWW, 2005.

[12] I. Podna, M. Rajman, T. Luu, F. Klemn, and K. Aberer, "Scalable Peer-to-peer Web Retrieval with Highly Discriminative Keys," in Proc. IEEE ICDE, 2007.

[13] G. Skobeltsyn, T. Luu, I. P. Zarko, M. Rajman, and K. Aberer, "Web Text Retrieval with a P2P Query Driven Index," in Proc. ACM SIGIR, 2007.

[14] G. Skobeltsyn, F. Junqueira, V. Plachouras, and R. Baeza-Yates, "ResIn: a Combination of Results Caching and Index Pruning for High-performance Web Search Engines," in Proc. ACM SIGIR, 2008.

[15] C. Tang, and S. Dwarkadas, "Hybrid Global-local Indexing for Efficient Peer-to-peer Information Retrieval," in Proc. NSDI, 2004.

[16] S. Kullback, "The Kullback-Leibler Distance," The American Statistician, 41:340-341, 1987.

[17] D. Puppin, F. Silvestri, and D. Laforenza, "Query-Driven Document Partitioning and Collection Selection," in Proc. INFOSCALE, 2006.

[18] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Technical Report, Stanford University.

[19] J. Kleinberg, "Authoritative Sources in a Hyperlinked Environment," in Journal of the ACM, 46(5), 1999.

[20] S. E. Robertson. S. Walker, and M. Hancock-Beaulieu, "Okapi at TREC-7," in Proc. of the Seventh Text Retrieval Conference, 1998.

[21] TERabyte RetrIEveR, http://ir.dcs.gla.ac.uk/terrier/

[22] TREC (WT10G, TREC-9)

[23] M. F. Porter, "An Algorithm for Suffix Stripping," in Program, 14(3), 1980.

Sorting using BItonic netwoRk wIth CUDA

Ranieri Baraglia
ISTI - CNR
Via G. Moruzzi, 1
56124 Pisa, Italy
r.baraglia@isti.cnr.it

Gabriele Capannini
ISTI - CNR
Via G. Moruzzi, 1
56124 Pisa, Italy
g.capannini@isti.cnr.it

Franco Maria Nardini
ISTI - CNR
Via G. Moruzzi, 1
56124 Pisa, Italy
f.nardini@isti.cnr.it

Fabrizio Silvestri
ISTI - CNR
Via G. Moruzzi, 1
56124 Pisa, Italy
f.silvestri@isti.cnr.it

ABSTRACT

Novel "manycore" architectures, such as graphics processors, are high-parallel and high-performance shared-memory architectures [7] born to solve specific problems such as the graphical ones. Those architectures can be exploited to solve a wider range of problems by designing the related algorithm for such architectures. We present a fast sorting algorithm implementing an efficient bitonic sorting network. This algorithm is highly suitable for information retrieval applications. Sorting is a fundamental and universal problem in computer science. Even if sort has been extensively addressed by many research works, it still remains an interesting challenge to make it faster by exploiting novel technologies. In this light, this paper shows how to use graphics processors as coprocessors to speed up sorting while allowing CPU to perform other tasks. Our new algorithm exploits a memory-efficient data access pattern maintaining the minimum number of accesses to the memory out of the chip. We introduce an efficient instruction dispatch mechanism to improve the overall sorting performance. We also present a cache-based computational model for graphics processors. Experimental results highlight remarkable improvements over prior CPU-based sorting methods, and a significant improvement over previous GPU-based sorting algorithms.

1. INTRODUCTION

Every day people use Web Search Engines as a tool for accessing information, sites, and services on the Web. Information retrieval has to face those issues due to the growing amount of information on the web, as well as the number of new users. Creating a Web Search Engine which scales even to today's Web contents presents many challenges. A fast crawling technology is needed to gather web documents and keep them up to date. Storage space must be used efficiently to store indexes, and documents. The indexing system must process hundreds of gigabytes of data efficiently. Queries must be handled quickly, at a rate of hundreds to thousands per second. All these services run on clusters of homogeneous PCs. PCs in these clusters depends upon price, CPU speed, memory and disk size, heat output, and physical size [3]. Nowadays these characteristics can be find also in other commodity hardware originally designed for specific graphics computations. Many special processor architectures have been proposed to exploit data parallelism for data intensive computations and graphics processors (GPUs) are one of those. For example, the scientific community uses GPUs for general purpose computation. The result obtained, in term of computational latency, outperform the time charge requested on classical processors. Unfortunately, such programs must rely on APIs to access the hardware, for example OpenGL or DirectX. These APIs are simultaneously overspecified, forcing programmer to manipulate data that is not directly relevant, and drivers. These APIs make critical policy decisions, such as deciding where data resides in memory and when they are copied.

In last years, due to the growing trend of media market, the request of rendering algorithms is rapidly evolving. For those companies producing hardware, it means to design every time new hardware both able to run novel algorithms and able to provide higher rate of computations per second. Such processors require significant design effort and are thus difficult to change as applications and algorithms evolve. The request for flexibility in media processing motivates the use of programmable processors, and the existing need for non-graphical APIs pushed the same companies into creating new abstractions designed to last.

Finally, according to what Moore's law foresee, the number of transistor density doubles every two years. Furthermore, mainly due to power-related issues, new generation processors (such as traditional CPUs) tend to incorporate an ever-increasing number of processors (also called cores) on the same chip [9]. The result is that nowadays the market proposes low-cost commodity hardware that is able to execute heavy loads of computations. In order to enable developers to leverage the power of such architectures, they usually make available ad-hoc programming tools for.

For the reasons listed so far, these architectures are ideal

LSDS-IR Workshop. July 2009. Boston, USA.

candidates for the efficient implementation of those component of a Large-Scale Search Engine that are eager of computational power.

This paper focuses on using a GPU as a co-processor for sorting. Sorting is a core problem in computer science that has been extensively researched over the last five decades and still remains a bottleneck in many applications involving large volumes of data. One could argue why efficient sorting is related with LSDS-IR. First of all, sorting is a basic application for indexing. We will show in Section 3 how many indexing algorithms are basically a sorting operation over integer sequences. Large scale indexing, thus, required scalable sorting. Second, the technique we are introducing here is of crucial importance for Distributed Systems for IR since it is designed to run on GPUs that are considered by many as a basic building block for future generation data-centers [4]. Our bitonic sorting network can be seen as a viable alternative for sorting large quantities of data on graphics processors. In the last years general purpose processors have been specialized adopting mechanisms to make more flexible their work. Such facilities (i.e. more levels of caches, out-of-order execution paradigm, and branch prediction techniques) leads to make the theoretical performance of CPUs closer to the real achievable one. From the other side specialized processors, like GPUs, expose lower flexibility at design phase, but are able to reach higher computational power providing more computational cores with respect to other the class of processors. We map a bitonic sorting network on GPU exploiting the its high bandwidth memory interface. Our novel data partitioning improves GPU cache efficiency and minimizes data transfers between on-chip and off-chip memories.

This paper is organized as follows. Section 2 discusses related works, while Sections 3 introduces some relevant characteristics about the applicability of GPU-based sorting in Web Search Engines. Section 4 presents some issues arising from the stream programming model and the single-instruction multiple-data (SIMD) architecture. The central part is devoted to expose our solution, and the computational model used to formalize Graphics Processing Units. Section 6 presents some results obtained in a preliminary evaluation phase. Section 7 discuss hot to evolve and improve this research activity.

2. RELATED WORK

Since most sorting algorithms are memory bandwidth bound, there is no surprise that there is currently a big interest in sorting on the high bandwidth GPUs.

Purcell *et al.* [24] presented an implementation of bitonic merge sort on GPUs based on an implementation by Kapasi *et al.* [17]. Author used that approach to sort photons into a spatial data structure providing an efficient search mechanism for GPU-based photon mapping. Comparator stages were entirely realized in the fragment units[1], including arithmetic, logical and texture operations. Authors reported their implementation to be compute-bound rather than bandwidth-bound, and they achieve a throughput far below the theoretical optimum of the target architecture.

Kipfer *et al.* [19, 20] showed an improved version of the bitonic sort as well as an odd-even merge sort. They presented an improved bitonic sort routine that achieves a performance gain by minimizing both the number of instructions to be executed in the fragment program and the number of texture operations.

Zachmann *et al.* [14] presented a novel approach for parallel sorting on stream processing architectures based on an adaptive bitonic sorting [6]. They presented an implementation based on modern programmable graphics hardware showing that they approach is competitive with common sequential sorting algorithms not only from a theoretical viewpoint, but also from a practical one. Good results are achieved by using efficient linear stream memory accesses and by combining the optimal time approach with algorithms.

Govindaraju *et al.* [13] implemented sorting as the main computational component for histogram approximation. This solution is based on the periodic balanced sorting network method by Dowd *et al.* [10]. In order to achieve high computational performance on the GPUs, they used a sorting network based algorithm and each stage is computed using rasterization. They later presented a hybrid bitonic-radix sort that is able to sort vast quantities of data [12], called GPUTeraSort. This algorithm was proposed to sort record contained in databases using a GPU. This approach uses the data and task parallelism on the GPU to perform memory-intensive and compute-intensive tasks while the CPU is used to perform I/O and resource management.

Sengupta *et al.* [25] presented a radix-sort and a Quicksort implementation based on segmented *scan* primitives. Authors presented new approaches of implementing several classic applications using this primitives and shows that this primitives are an excellent match for a broad set of problems on parallel hardware.

Recently, Sintorn *et al.* [28] presented a sorting algorithm that combines bucket sort with merge sort. In addition, authors show this new GPU sorting method sorts on *nlog(n)* time.

Cederman *et al.* [8] showed that their GPU-Quicksort is a viable sorting alternative. The algorithm recursively partition the sequence to be sorted with respect to a pivot. This is done in parallel by each GPU-thread until the entire sequence has been sorted. In each partition iteration a new pivot value is picked and as a result two new subsequences are created that can be sorted independently by each thread block can be assigned one of them. Finally, experiments pointed out that GPU-Quicksort can outperform other GPU-based sorting algorithms.

3. APPLICATIONS TO INDEXING

A modern search engine must scale even with the growing of today's Web contents. Large-scale and distributed applications in Information Retrieval such as crawling, indexing, and query processing can exploit the computational power of new GPU architectures to keep up with this exponential grow.

We consider here one of the core-component of a large-scale search engine: *the indexer*. In the indexing phase, each crawled document is converted into a set of word occurrences called *hits*. For each word the hits record: frequency, position in document, and some other information. Indexing, then, can be considered as a "sort" operation on a set of records representing term occurrences [2]. Records repre-

[1]In addition to computational functionality, fragment units also provide an efficient memory interface to server-side data, i.e. texture maps and frame buffer objects.

sent distinct occurrences of each term in each distinct document. Sorting efficiently these records using a good balance of memory and disk usage, is a very challenging operation.

In the last years it has been shown that sort-based approaches [29], or single-pass algorithms [21], are efficient in several scenarios, where indexing of a large amount of data is performed with limited resources.

A sort-based approach first makes a pass through the collection assembling all term-docID pairs. Then it sorts the pairs with the term as the dominant key and docID as the secondary key. Finally, it organizes the docIDs for each term into a postings list (it also computes statistics like term and document frequency). For small collections, all this can be done in memory.

When memory is not sufficient, we need to use an external sorting algorithm [22]. The main requirement of such algorithm is that it minimizes the number of random disk seeks during sorting. One solution is the Blocked Sort-Based Indexing algorithm (BSBI). BSBI segments the collection into parts of equal size, then it sorts the termID-docID pairs of each part in memory, finally stores intermediate sorted results on disk. When all the segments are sorted, it merges all intermediate results into the final index.

A more scalable alternative is Single-Pass In-Memory Indexing (SPIMI). SPIMI uses terms instead of termIDs, writes each block's dictionary to disk, and then starts a new dictionary for the next block. SPIMI can index collections of any size as long as there is enough disk space available. The algorithm parses documents and turns them into a stream of term-docID pairs, called tokens. Tokens are then processed one by one. For each token, SPIMI adds a posting directly to its postings list. Instead of first collecting all termID-docID pairs and then sorting them (as BSBI does), each postings list is dynamic. This means that its size is adjusted as it grows. This has two advantages: it is faster because there is no sorting required, and it saves memory because it keeps track of the term a postings list belongs to, so the termIDs of postings need not be stored.

When memory finished, SPIMI writes the index of the block (which consists of the dictionary and the postings lists) to disk. Before doing this, SPIMI sorts the terms to facilitate the final merging step: if each block's postings lists were written in unsorted order, merging blocks could not be accomplished by a simple linear scan through each block. The last step of SPIMI is then to merge the blocks into the final inverted index.

SPIMI, which time complexity is lower because no sorting of tokens is required, is usually preferred with respect to BSBI that presents an higher time complexity.

In both the methods presented for indexing, sorting is involved: BSBI sorts the termID-docID pairs of all parts in memory, SPIMI sorts the terms to facilitate the final merging step [22].

In order to efficiently evaluate these two approaches on a heterogeneous cluster we have to compare "standard" SPIMI performances with the performances of a BSBI-based sorter implemented by us. Moreover, to fully analyze the indexing phase, we need a GPU-based string sorter able to outperform CPUs as well as our sorter for integers does. In this way we have the possibility to compare both solutions, on all architectures, then to choose the best combination. Having all possible implementations available, a *flexible* execution of indexing running on various hardware can be imagined. This option is even more important if the allocation of the task is scheduled dynamically, as it can be done depending of the workload of the single resources.

The-state-of-art in string sort lacks of solution for GPU architectures: nowadays we are not aware of solutions for parallel SIMD processors. In the literature, this problem is efficiently solved by using different approaches. The most interesting and suitable for us seems to be Burstsort [27]. It is a technique that combines the burst trie [15] to distribute string-items into small buckets whose contents are then sorted with standard (string) sorting algorithms. Successively, Sinha *et al.* [26] introduced improvements that reduce by a significant margin the memory requirements of Burstsort. This aspect is even more relevant for GPU architectures having small-sized on-chip memories.

4. SORTING WITH GPUS

This section gives a brief overview of GPUs highlighting features that make them useful for sorting. GPUs are designed to execute geometric transformations that generate a data stream of display-pixels. A data stream is processed by a program running on multiple SIMD processors, which are capable for data-intensive computations. The output, then, is written back to the memory.

4.1 SIMD Architecture

SIMD machines are classified as processor-array machines: a SIMD machine basically consists of an array of computational units connected together by a simple network topology [23]. This processor array is connected to a control processor, which is responsible for fetching and interpreting instructions. The control processor issues arithmetic and data processing instructions to the processor array, and handles any control flow or serial computation that cannot be parallelized. Processing elements can be individually disabled for conditional execution: this option give more flexibility during the design of an algorithm.

Although SIMD machines are very effective for certain classes of problems, the architecture is specifically tailored for data computation-intensive work, and it results to be quite "inflexible" on some classes of problems[2].

4.2 Stream Programming Model

A stream program [18] organizes data as *streams* and expresses all computation as *kernels*. A stream is a sequence of similar data elements, that are defined by a regular access pattern. A kernel typically loops through all the input stream elements, performing a sequence of operations on each element, and appending results to an output stream. These operations exhibits an high instruction level parallelism. Moreover, these operations cannot access to arbitrary memory locations but they keep all the intermediate values locally, into kernels. Since each element of the input stream can be processed simultaneously, kernels also expose large amounts of data-level parallelism. Furthermore, stream memory transfers can be executed concurrently with kernels, thereby exposing task-level parallelism in the stream program. Some other important characteristics common to all stream-processing applications are: (i) elements are read once from memory, (ii) elements are not visited twice, and

[2] For example, these architectures cannot efficiently run the control-flow dominated code.

(iii) the application requires high level of arithmetic operations per memory reference, i.e. computationally intensive.

4.3 Nvidia's CUDA

CUDA [1], acronym of Compute Unified Device Architecture, is defined as an architecture built around a scalable array of multithreaded streaming multiprocessors (MPs). Each MP is defined as a unit comprising one instruction unit, a collection of eight single precision pipelines also called cores, a functional units for special arithmetical operations and a 16 KB local store also called shared memory. In practice, this means that each MP is a *SIMD* processor, whose cores form an arithmetic pipeline that executes scalar instructions. All MPs create, manage, and execute concurrent threads in hardware with zero scheduling overhead, and implements a barrier for threads synchronization. Nevertheless, only threads concurrently running on the same MP can be synchronized.

CUDA model also introduces the entity *warp* as a group of 32 parallel scalar threads, and reports that each warp executes one common instruction at a time. This is another way of saying that warp is a stream of vector instructions: scalar threads are then vector elements. But, unlike others SIMD instruction set, such as Intel's SSE, a particular value of the vector length is not specified. A *thread block* is defined as a collection of warps that run on the same MP and share a partition of local store. The number of warps in a thread block is configurable.

4.3.1 CUDA SDK

The SDK provided by Nvidia for its GPUs consist of a large, collection of code examples, compilers and run-time libraries. Clearly the CUDA model is "restricted" to Nvidia products, mainly for efficiency reasons, and it is conform to the stream programming model. Threads and thread blocks can be created only by invoking a parallel kernel, not from within a parallel kernel. Task parallelism can be expressed at the thread-block level, but block-wide barriers are not well suited for supporting task parallelism among threads in a block. To enable CUDA programs to run on any number of processors, communication between different blocks of threads, is not allowed, so they must execute independently. Since CUDA requires that thread blocks are independent and allows blocks to be executed in any order. Combining results generated by multiple blocks must in general be done by launching a second kernel. However, multiple thread blocks can coordinate their work using atomic operations on the external (off-chip) memory.

Recursive function calls are not allowed in CUDA kernels. Recursion is, in fact, unattractive in a massively parallel kernel. Providing stack space for all the active threads it would require substantial amounts of memory. To support an heterogeneous system architecture combining a CPU and a GPU, each with its own memory system, CUDA programs must copy data and results between host memory and device memory. The overhead of CPU/GPU interaction and data transfers is minimized by using DMA block-transfer engines and fast interconnects.

4.4 Cache-oblivious algorithms

The cost of communication can be larger up to an order of magnitude than the cost of the pure computation on such architectures. Our idea is to model the proposed solution as cache-oblivious algorithms. The model underlying this type of algorithms is not directly applicable on GPU's parallel architecture, which is equipped with local memory instead of cache. Adopting local memory approach, the programmer has to bear the effort of synchronizing, sizing, and scheduling the computation of data and its movement through the off-chip memory and the in-chip one. On the other hand in cache-based architectures this aspect is automatically managed by the underlying support. A local memory approach permits to move data located in different addresses composing a specific access pattern. This capability is impossible to realize with caches, where the hardware hides this operation by automatically replacing missed cache lines.

Frigo *et al.* [11] presents cache-oblivious algorithms that use both asymptotically optimal amounts of work, and asymptotically optimal number of transfers among multiple levels of cache. An algorithm is cache oblivious if no program variables dependent on hardware configuration parameters, such as cache size and cache-line length need to be tuned to minimize the number of cache misses. Authors introduce the "Z, L" ideal-cache model to study the cache complexity of algorithms. This model describes a computer with a two-level memory hierarchy consisting of an ideal data cache of Z words of constant size, and an arbitrarily large main memory. The cache is partitioned into cache lines, each consisting of L consecutive words that are always moved together between cache and main memory.

The processor can only reference words that reside in the cache. If the referenced word belongs to a line already in cache, a cache hit occurs, and the word is delivered to the processor. Otherwise, a cache-miss occurs, and the line is fetched into the cache. If the cache is full, a cache line must be evicted. An algorithm with an input of size n is measured in the ideal-cache model in terms of its work complexity $W(n)$ and its cache complexity $Q(n, Z, L)$, that is the number of cache misses it incurs as a function of the size Z and line length L of the ideal cache.

The metrics used to measure cache-oblivious algorithms need to be reconsidered in order to be used with GPUs that are parallel architectures. To do that $W()$ has to be defined taking care of the level of parallelism exposed by GPUs. Evaluating $Q()$, we must translate the concept of Z and L that refer to cache characteristics. More precisely, a GPUs is provided of p MPs each one with a local memory. We can abstract such architectural organization by considering each local memory as one cache-line, and the union of all local memories as the entire cache, taking no care of which processor is using data. In this point of view, if the shared memory of each MP is 16 KB, we obtain $L = 16$ KB and $Z = 16 \cdot p$ KB.

5. BITONIC SORT

To design our sorting algorithm in the stream programming model, we started from the popular Bitonic Sort (BS) network and we extend it to adapt to our specific architecture. Specifically, BS is one of the fastest sorting networks [5]. A sorting network is a special kind of sorting algorithm, where the sequence of comparisons do not depend on the order with which the data is presented, see Figure 1. This makes sorting networks suitable for implementation on GPUs. In particular, the regularity of the schema used to compare the elements to sort, makes this kind of sorting network particularly suitable for partitioning the elements in

the stream programming fashion, as GPUs require. Finally, we compared theoretical results with the ones resulting from the tests, in order to say if the adapted ideal-cache model is useful to abstract GPUs.

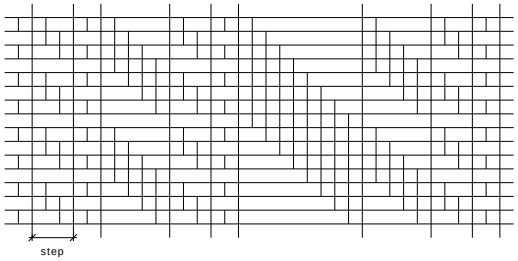


Figure 1: Bitonic sorting networks for 16 elements. Each step is completed when all comparisons involved are computed. In the figure each comparison is represented with a vertical line that link two elements, which are represented with horizontal lines.

The main issue to address is to define an efficient schema to map all comparisons involved in the BS on the elements composing the streams invoked.

The first constraint is that the elements composing each stream must be "distinct". This means that each item in the input has to be included in exactly one element of the stream. From this point of view, each stream of elements defines a partition of the input array to be sorted. This characteristic is necessary due to the runtime support, because it does not permit any type of data-exchange, or synchronization, between two elements (Figure 2).

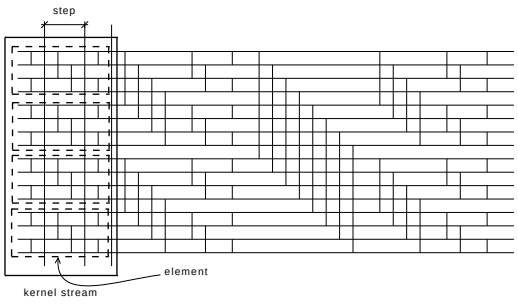


Figure 2: Example of a kernel stream comprising more sorting network steps. The subset of items composing each element must perform comparison only inside itself.

The second aspect to optimize is the partitioning of the steps composing the sorting network. Since a BS is composed by several steps (Figure 1), we have to map the execution of all steps into a sequence of independent runs, each of them corresponding to the invocation of a kernel. Since each kernel invocation implies a communication phase, such mapping should be done in order to reduce the number of these invocations, thus the communication overhead. Specifically, this overhead is generated whenever the SIMD processor begins the execution of a new stream element. In that case, the processor needs to flush the results contained in the proper shared memory, then to fetch the new data from the off-chip memory. In the ideal-cache computational model, it corresponds to a cache-miss event, which wastes the performance of the algorithm.

Resuming, performing several network steps in the same kernel has the double effect to reduce the number of cache-misses, i.e. improving $Q()$ metric, and to augment the level of arithmetic operations per memory reference. The unique constraint is that the computation of an element has to be independent from the one of another element in the same stream.

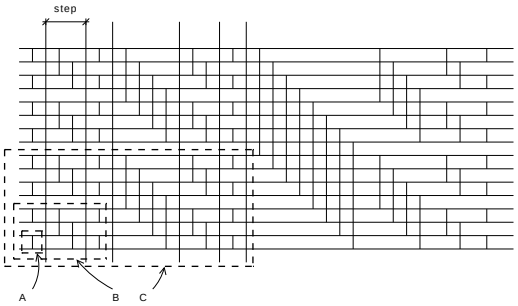


Figure 3: Increasing the number of steps covered by the partition, the number of items included doubles. A, B and C are partitions respectively for local memory of 2, 4 and 8 locations.

Let us introduce our solution. First of all, we need to establish the number of consecutive steps to be executed by one kernel. We must consider that for each step assigned to a kernel, in order to maintain the all stream elements independent, the number of memory location needed by the relative partition doubles, see Figure 3. So, the number of steps a kernel can cover is bounded by the number of items that it is possible to include into the stream element. Furthermore, the number of items is bounded by the size of the shared memory available for each SIMD processor.

Algorithm 1 Bitonic Sort algorithm.

```
a ← array to sort
for s = 1 to log2 |a| do
  for c = s − 1 down to 0 do
    for r = 0 to |a| − 1 do
      if r/2^c ≡ r/2^s (mod 2) ∧ a[r] > a[r ⊕ 2^c] then
        a[r] ⇆ a[r ⊕ 2^c]
      end if
    end for
  end for
end for
```

More precisely, to know how many steps can be included in one partition, we have to count how many distinct values c assumes, see Algorithm 1. Due to the fixed size of memory locations, i.e. 16 KB, we can specifies partition of SH = 4 K items, for items of 32 bits. Moreover such partition is able to cover "at least" $sh = log(SH) = 12$ steps. From this evaluation it is also possible to estimate the size of the kernel stream: if a partition representing an element of the stream contains SH items, and the array a to sort contains $N = 2^n$ items, then the stream contains $b = N/SH = 2^{n-sh}$ elements.

A important consideration must be done for the first kernel invoked by the algorithm: until the variable s in the Algorithm 1 is not greater than sh the computation of the several first steps can be done with this first kernel. This because the values assumed by c remain in a range of sh distinct values. More precisely the first kernel computes the first $\frac{sh\cdot(sh+1)}{2}$ steps (Figure 3).

This access pattern schema can be traduced in the function $\ell_c(i)$ that for the current kernel stream, given the current value of c, is able to define the subset of a to be assigned to the i-th stream element. In other words, ℓ describes a method to enumerate the set of indexes in a that the i-th element of the kernel stream has to perform. Formally, it is defined as:

$$\ell_c : i \in [0, b-1] \rightarrow \Pi \subseteq \pi_{sh}(a)$$

where $\pi_{sh}(a)$ is a partition of a, namely a set of nonempty subsets of a such that every element in a is in exactly one of these subsets, and each subset contains exactly 2^{sh} elements of a.

Let us assume $n = 32$ and the size of the shared memory can contains 4 K items, so we have $|a| = 2^{32}$ and $b = 2^{n-sh} = 2^{32-12} = 2^{20}$ elements for each stream. Basically, we need 32 bits to point an element of a, and $\log(b) = 20$ bits to identify the i-th partition among the b existing. The i value is used to build a bit-mask that is equal for each address produced by $\ell_c(i)$. Such mask sets $\log(b)$ bits of the 32 bits composing an index for a. The missing sh bits are generated by using a variable x to enumerate all values in the range $\mathbb{X} = [0, 2^{sh}-1]$ and by inserting each of them in c-th position of the i mask. This composition leads to a set of addresses of n bits whose relative items compose the b-th partition. Formally, we obtain:

$$\ell_c(i) = \{x \in \mathbb{X} \ : \ i_{[31\ldots c]} \circ x \circ i_{[c+1\ldots 0]}\}$$

where $i_{[31\ldots c]}$ notation identifies the leftmost bits, namely from the 31th bit down to the c-th one, and "$\circ$" is the concatenation operator.

The rule to compose the elements in $\ell_c(i)$ is easy, but in some case it leads to some exception. When $c < sh$, then x is divided in two parts, that are x_L and x_R, and they are inserted in c-th position and in the c'-th position of i respectively. In particular, the statement $c < sh$ occurs whenever the middle loop in the Algorithm 1 ends and c start a new loop getting the new value of s, denoted with c'. Specifically, it happens that, depending on the current value of c, the algorithm needs to make two insertions: to add x_R at position c, and to add x_L at position c'. Formally, when $c < sh$, we have to define a second function $\ell_{c,c'}()$ as in the following:

$$\ell_{c,c'}(i) = \{x \in \mathbb{X} \ : \ i_{[31\ldots\bar{c}]} \circ x_L \circ i_{[c'+sh-c\ldots c]} \circ x_R\}$$

where $x_L = x_{[sh-1\ldots c]}$, $x_R = x_{[c-1\ldots 0]}$, and $c' = s + 1$.

5.1 Evaluation

For our solution we obtain that $W(n,p)$, where p indicates the number of MPs, and n the size of a, is equal to the computational cost of the sequential BS divided p, specifically $W(n,p) = O\big(\frac{n}{p} \cdot \log^2 n\big)$. To know $Q(n,Z,L)$ we must estimate the number of cache-misses. Assuming $|a| = n$, we obtain that the sorting network is made of $\sigma = \frac{1}{2}\big(\log^2(n) + \log(n)\big)$ steps. Furthermore, let us assume $L = SH$, $Z = SH \cdot p$, and each kernel covers sh steps, except the first kernel that covers $\sigma' = \frac{1}{2}\big(sh^2 + sh\big)$. Then the number of cache-misses is $\lceil(\sigma - \sigma')/sh\rceil$.

The last consideration regards $W(n,p)$ measure, that should be estimated considering that each MP is a SIMD processor. In fact, each MP reaches its maximum performance whenever the data-flow permits to the control unit to issue the same instruction for all cores. In this way such instruction is executed in parallel on different data in a SIMD fashion.

In order to compare our bitonic sort with the quick sort proposed by Cederman *et al.*, we tried to extend the analysis of ideal-cache model metrics to their solution. Unfortunately their solution does not permit to be accurately measured like ours. In particular, it is possible to estimate the number of transfers among multiple levels of cache, but quick sort uses off-chip memory also to implement prefix-sum for each stream element ran. In particular quick sort splits input array in two parts with respect to a pivot by invoking a procedure on GPU, and recursively repeats this operation until each part can be entirely contained in the shared memory. In the optimistic case, assuming $|a| = n$ and a shared memory equal to $L = SH$, this operation is invoked $\log(n) - \log(L)$ times, that is also the number of cache-misses for quick sort, namely $Q()$. This value is sensibly lower to the $Q()$ measured for bitonic sort, but the evaluation of the number of such cache-misses should be proportionally augmented due to the prefix-sum procedure. Regarding $W()$, the authors report a computational complexity equal to the one obtained for sequential case, i.e. $O\big(n \cdot \log(n)\big)$, without referring the parallelism of the underlying hardware. However, optimistic evaluation of such parallel, version should be, also in this case, lower than $W()$ computed for bitonic sort.

6. RESULTS

The experiments have been conducted on an Ubuntu Linux Desktop with an Nvidia 8800GT, namely a GPU provided with 14 SIMD processors and CUDA SDK 2.1. To generate the arrays for these preliminary tests we used uniform, gaussian and zipfian distributions. Specifically, for each distribution was generated 20 different arrays. Figure 4 shows: the means, the standard deviation, the maximum and the minimum for the times elapsed for all runs of each distribution. The tests involved CPU-based quick sort provided with C standard library, our solution and the one proposed by Cederman *et al.* [8]. In that paper, the GPU-based quick sort resulted the most performing algorithm in literature, so our preliminary tests take into consideration only such GPU-based algorithm.

Figure 4 shows that GPU-based solutions are able to outperform CPU's performance for the sorting problem. The same figure also shows that our solution is not competitive with respect to the one of Cederman until the number on items to sort reaches 8 MB. One more consideration is that GPU-based quick sort is not able to successfully conclude the tests for arrays greater than 16 MB. Further analysis pointed out that bitonic algorithm spends the main part of the time elapsed for the data-transfer phase of some specific kernel instance. Since element streams were always of the same size, we deduced that the number of transfers is not the only aspect to take into consideration to minimize the communication overhead, as metrics of ideal-cache models suggests.

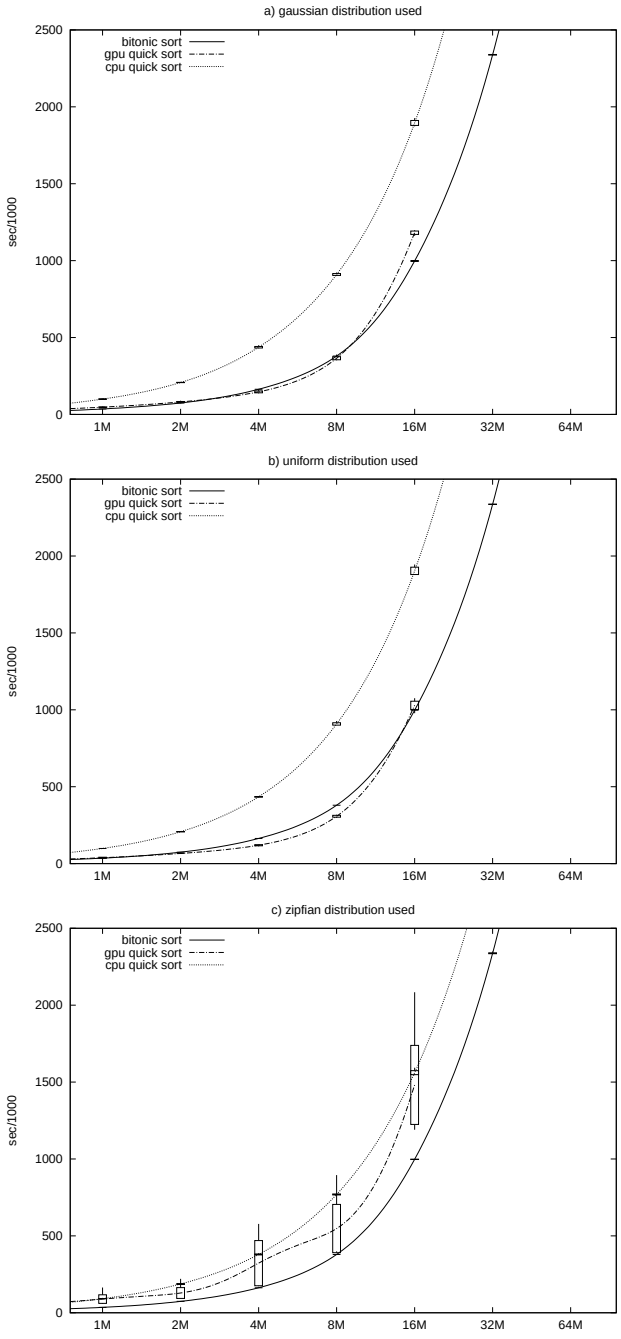


Figure 4: Squared white areas represent the variance obtained for several running for each size of the problem. Vertical lines point out the maximum and the minimum value obtained.

As suggest by Helman *et al.* [16] a deeper evaluation of the algorithm can be conducted by using arrays generated by different distributions. This type of test puts in evidence the behavior of the algorithms regarding to the variance of the times obtained in different contexts. For the two distribution tested, bitonic sort algorithm has a better behavior with respect to variance. Obviously, this result is caused by the type of algorithm used. Quick sort is a data-dependent approach whereas sorting network are based on a fixed schema of comparisons that does not vary with respect to data-distribution.

Consideration on the results obtained from these preliminary test suggest us that ideal-cache model does not seem sufficiently accurate to abstract GPU's architecture. If theoretical results lead to better performance for GPU-based quick sort, from the tests conducted, it arises that bitonic sort has a better performance-trend by increasing the size of the problem. This consideration is enforced by the analysis of the data-transfer: we strongly believe that by improving the data-transfer bandwidth, bitonic sort can reach better results without increasing theoretical $W()$ and $Q()$ metrics.

7. FUTURE WORKS

Preliminary results show that the number of transfers is not the only aspect to take into consideration for minimizing communication overhead. Another important factor is the *transfer bandwidth* that is relevant to achieve better results. Results show that the ideal-cache model is not able to fully describe and capture all the aspects determining the performance for such kind of architectures. Probably different kinds of performance metrics are needed to evaluate an algorithms on these novel hardware resources.

Furthermore, since indexing is a tuple-sorting operation we will extend our solution to include the sorting of tuples of integers. In this paper, in fact, we assume the tuples are sorted by using multiple passes on the dataset. We reserve to future work the extension to tuples.

Of course, we have to run more tests to enforce the results obtained and to analyze more in deep the causes of the waste of performance that affect our algorithm.

8. ACKNOWLEDGMENTS

This research has been supported by the Action IC0805: Open European Network for High Performance Computing on Complex Environments.

9. REFERENCES

[1] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[2] R. Baeza-Yates, C. Castillo, F. Junqueira, V. Plachouras, and F. Silvestri. Challenges on distributed web retrieval. In *ICDE*. IEEE, 2007.

[3] L. A. Barroso, J. Dean, and Holzle. Web search for a planet: The google cluster architecture. *Micro, IEEE*, 23(2):22–28, 2003.

[4] L. A. Barroso and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool, San Rafael, CA, USA, 2009.

[5] K. E. Batcher. Sorting networks and their applications. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, New York, NY, USA, 1968. ACM.

[6] G. Bilardi and A. Nicolau. Adaptive bitonic sorting: An optimal parallel algorithm for shared memory machines. Technical report, Ithaca, NY, USA, 1986.

[7] J. Bovay, B. H. Brent, H. Lin, and K. Wadleigh. Accelerators for high performance computing

investigation. White paper, High Performance Computing Division - Hewlett-Packard Company, 2007.

[8] D. Cederman and P. Tsigas. A practical quicksort algorithm for graphics processors. In *ESA '08: Proceedings of the 16th annual European symposium on Algorithms*, pages 246–258, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] J. D. Davis, J. Laudon, and K. Olukotun. Maximizing cmp throughput with mediocre cores. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 51–62, Washington, DC, USA, 2005. IEEE Computer Society.

[10] M. Dowd, Y. Perl, L. Rudolph, and M. Saks. The periodic balanced sorting network. *J. ACM*, 36(4):738–757, 1989.

[11] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, Washington, DC, USA, 1999. IEEE Computer Society.

[12] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: High performance graphics coprocessor sorting for large database management. In *ACM SIGMOD International Conference on Management of Data*, Chicago, United States, June 2006.

[13] N. K. Govindaraju, N. Raghuvanshi, and D. Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 611–622, New York, NY, USA, 2005. ACM.

[14] A. Greß and G. Zachmann. Gpu-abisort: Optimal parallel sorting on stream architectures. In *The 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS '06)*, page 45, Apr. 2006.

[15] S. Heinz, J. Zobel, and H. E. Williams. Burst tries: A fast, efficient data structure for string keys. *ACM Transactions on Information Systems*, 20(2):192–223, 2002.

[16] D. R. Helman, D. A. B. Y, and J. J. Z. A randomized parallel sorting algorithm with an experimental study. Technical report, Journal of Parallel and Distributed Computing, 1995.

[17] U. J. Kapasi, W. J. Dally, S. Rixner, P. R. Mattson, J. D. Owens, and B. Khailany. Efficient conditional operations for data-parallel architectures. In *In Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 159–170. ACM Press, 2000.

[18] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: Media processing with streams. *IEEE Micro*, 21(2):35–46, 2001.

[19] P. Kipfer, M. Segal, and R. Westermann. Uberflow: a gpu-based particle engine. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 115–122, New York, NY, USA, 2004. ACM Press.

[20] P. Kipfer and R. Westermann. Improved GPU sorting. In M. Pharr, editor, *GPUGems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 733–746. Addison-Wesley, 2005.

[21] N. Lester. Fast on-line index construction by geometric partitioning. In *In Proceedings of the 14th ACM Conference on Information and Knowledge Management (CIKM 2005*, pages 776–783. ACM Press, 2005.

[22] C. D. Manning, P. Raghavan, and H. Schütze. *An Introduction to Information Retrieval*. 2008.

[23] M. A. Nichols, H. J. Siegel, H. G. Dietz, R. W. Quong, and W. G. Nation. Eliminating memory for fragmentation within partitionable simd/spmd machines. *IEEE Trans. Parallel Distrib. Syst.*, 2(3):290–303, 1991.

[24] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH Conference on Graphics Hardware*, pages 41–50. Eurographics Association, 2003.

[25] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for gpu computing. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[26] R. Sinha and A. Wirth. Engineering burstsort: Towards fast in-place string sorting. In *WEA*, pages 14–27, 2008.

[27] R. Sinha, J. Zobel, and D. Ring. Cache-efficient string sorting using copying. *ACM Journal of Experimental Algorithmics*, 2006.

[28] E. Sintorn and U. Assarsson. Fast parallel gpu-sorting using a hybrid algorithm. *Journal of Parallel and Distributed Computing*, In Press, Corrected Proof.

[29] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, San Francisco, CA, 1999.

Comparing Distributed Indexing: To MapReduce or Not?

Richard M. C. McCreadie
Department of Computing Science
University of Glasgow
Glasgow, G12 8QQ
richardm@dcs.gla.ac.uk

Craig Macdonald
Department of Computing Science
University of Glasgow
Glasgow, G12 8QQ
craigm@dcs.gla.ac.uk

Iadh Ounis
Department of Computing Science
University of Glasgow
Glasgow, G12 8QQ
ounis@dcs.gla.ac.uk

ABSTRACT

Information Retrieval (IR) systems require input corpora to be indexed. The advent of terabyte-scale Web corpora has reinvigorated the need for efficient indexing. In this work, we investigate distributed indexing paradigms, in particular within the auspices of the MapReduce programming framework. In particular, we describe two indexing approaches based on the original MapReduce paper, and compare these with a standard distributed IR system, the MapReduce indexing strategy used by the Nutch IR platform, and a more advanced MapReduce indexing implementation that we propose. Experiments using the Hadoop MapReduce implementation and a large standard TREC corpus show our proposed MapReduce indexing implementation to be more efficient than those proposed in the original paper.

1. INTRODUCTION

The Web is the largest known document repository, and poses a major challenge for Information Retrieval (IR) systems, such as those used by Web search engines or Web IR researchers. Indeed, while the index sizes of major Web search engines are a closely guarded secret, these are commonly accepted to be in the range of billions of documents. For researchers, the recently released TREC ClueWeb09 corpus[1] of 1.2 billion Web documents poses both indexing and retrieval challenges. In both scenarios, the ability to efficiently create appropriate index structures to allow effective and efficient search is of much value. Moreover, at such scale, the use of distributed architectures to achieve high throughput is essential.

In this work, we investigate the MapReduce programming paradigm, that has been gaining popularity in commercial settings, with implementations by Google [5] and Yahoo! [21]. Microsoft also has a similar framework for distributed operations [10]. In particular, MapReduce allows the horizontal scaling of large-scale workloads using clusters of machines. It applies the intuition that many common large-scale tasks can be expressed as map and reduce operations [5], thereby providing an easily accessible framework for parallelism over multiple machines.

However, while MapReduce has been widely adopted within Google, and is reportedly used for their main indexing process, the MapReduce framework implementation and other programs using it remain (understandably) internal only. Moreover, there have been few empirical studies undertaken into the scalability of MapReduce beyond that contained within the original MapReduce paper [5], which in particular demonstrates the scalability of the simple operations grep and sort. More recently, a MapReduce implementation has been used to sort 1 terabyte of data in approx. 1 minute [17]. However, while Dean & Ghemawat [5] suggest a simple formulation in MapReduce for document indexing, no studies have empirically shown the benefits of applying MapReduce on the important IR indexing problem.

This paper contributes a first step towards understanding the benefits of indexing large corpora using MapReduce, in comparison to other indexing implementations. In particular, we describe four different methods of performing document indexing in MapReduce, from initial suggestions by Dean & Ghemawat, to more advanced strategies. We deploy MapReduce indexing strategies in the Terrier IR platform [18], using the freely available Hadoop implementation [1] of MapReduce, and then perform experiments using standard TREC data.

The remainder of this paper is structured as follows: Section 2 describes a state-of-the art single-pass indexing strategy; Section 3 introduces the MapReduce paradigm; Section 4 describes strategies for document indexing in MapReduce; Section 5 describes our experimental setup, research questions, experiments, and analysis of results; Concluding remarks are provided in Section 6.

2. INDEXING

In the following, we briefly describe the structures involved in the indexing process (Section 2.1) and how the modern single-pass indexing strategy is deployed in the open source Terrier IR platform [18] on which this work is based (Section 2.2). We then provide details of how an indexing process can be distributed to make use of additional machines (Section 2.3).

2.1 Index Structures

To allow efficient retrieval of documents from a corpus, suitable data structures must be created, collectively known as an index. Usually, a corpus covers many documents, and hence the index will be held on a large storage device - commonly one or more hard disks. Typically, at the centre of any IR system is the *inverted index* [23]. For each term, the inverted index contains a *posting list*, which lists the documents - represented as integer document-IDs (doc-IDs) - containing the term. Each posting in the posting list also stores sufficient statistical information to score each docu-

[1]See `http://boston.lti.cs.cmu.edu/Data/clueweb09/`.

LSDS-IR Workshop. July 2009. Boston, USA.

ment, such as the frequency of the term occurrences and, possibly, positional information (the position of the term within each document, which facilitates phrase or proximity search) [23] or field information (the occurrence of the term in various semi-structured area of the document, such as title, enabling these to be higher-weighted during retrieved). The inverted index does not store the textual terms themselves, but instead uses an additional structure known as a lexicon to store these along with pointers to the corresponding posting lists within the inverted index. A document index may also be created which stores meta-information about each document within the inverted index, such as an external name for the document (e.g. URL), and the length of the document [18]. The process of generating these structures is known as *indexing*.

2.2 Single-pass Indexing

When indexing a corpus of documents, documents are read from their storage location on disk, and then tokenised. Tokens may then be removed (stop-words) or transformed (e.g. stemming), before being collated into the final index structures [23]. Current state-of-the-art indexing uses a single-pass indexing method [8], where the (compressed) posting lists for each term are built in memory as the corpus is scanned. However, it is unlikely that the posting lists for very many documents would fit wholly in the memory of a single machine. Instead, when memory is exhausted, the partial indices are 'flushed' to disk. Once all documents have been scanned, the final index is built by merging the flushed partial indices.

In particular, the temporary posting lists held in memory are of the form list(term, list(doc-ID, Term Frequency)). Additional information such as positions or fields can also be held within each posting. As per modern compression schemes, only the first doc-ID in each posting list is absolute - for the rest, the difference between doc-IDs are instead stored to save space, using Elias-Gamma compression [6].

2.3 Distributing Indexing

The single-pass indexing strategy described above is designed to run on a single machine architecture with finite available memory. However, should we want to take advantage of multiple machines, this can be achieved in an intuitive manner by deploying an instance of this indexing strategy on each machine [22]. For machines with more than one processor, one instance per processing core is possible, assuming the local disk and memory are not saturated. As described by Ribeiro-Neto & Barbosa [20], each instance would index a subset of the input corpus to create an index for only those documents. It should be noted that if the documents to be indexed are local to the machines doing the work (*shared-nothing*), such as when each machine has crawled the documents it is indexing, then this strategy will *always be optimal* (will scale linearly with processing power). However, in practical terms, fully machine-local data is difficult to achieve when a large number of machines is involved. This stems from the need to split and distribute the corpus without overloading the network or risking un-recoverable data loss from a single point of failure.

Distributed indexing has seen some coverage in the literature. Ribeiro-Neto & Barbosa [20] compared three distributed indexing algorithms for indexing 18 million documents. Efficiency was measured with respect to local throughput of each processor, not in terms of overall indexing time. Unfortunately, they do not state the underlying hardware that they employ, and as such their results are difficult to compare to. Melnik et al. [15] described a distributed indexing regime designed for the Web, with considerations for updatable indices. However, their experiments did not consider efficiency as the number of nodes is increased.

In [5], Dean & Ghemawat proposed the MapReduce paradigm for distributing data-intensive processing across multiple machines. Section 3 gives an overview of MapReduce. Section 4 reviews prior work on MapReduce indexing, namely that of Dean & Ghemawat, who suggest how document indexing can be implemented in MapReduce, and from the Nutch IR system. Moreover, we propose a more advanced method of MapReduce indexing, which, by the experiments in Section 5, is shown to be more efficient.

3. MAPREDUCE

MapReduce is a programming paradigm for the processing of large amounts of data by distributing work tasks over multiple processing machines [5]. It was designed at Google as a way to distribute computational tasks which are run over large datasets. It is built on the idea that many tasks which are computationally intensive involve doing a 'map' operation with a simple function over each 'record' in a large dataset, emitting key/value pairs to comprise the results. The map operation itself can be easily distributed by running it on different machines processing different subsets of the input data. The output from each of these is then collected and merged into the desired results by 'reduce' operations.

By using the MapReduce abstraction, the complex details of parallel processing, such as fault tolerance and node availability, are hidden, in a conceptually simple framework [13], allowing highly distributed tools to easily be built on top of MapReduce. Indeed, various companies have developed tools to perform data mining operations on large-scale datasets on top of MapReduce implementations. Google's Sawzall [19] and Yahoo's Pig [16] are two such examples of data mining languages. Microsoft uses a distributed framework similar to MapReduce called Dryad, which the Nebula scripting language uses to provide similar data mining capabilities [10]. However, it is of note that MapReduce trades the ability to perform code optimisation (by abstracting from the internal workings) for easy implementation through its framework, meaning that an implementation in MapReduce is likely not the optimal solution, but will be cheaper to produce and maintain [11].

MapReduce is designed from a functional programming perspective, where functions provide definitions of operations over input data. A single MapReduce job is defined by the user as two functions. The map function takes in a key/value pair (of type <key1, value1>) and produces a set of intermediate key/value pairs (<key2, value2>). The outputs from the map function are then automatically grouped by their key, and then passed to the reduce function. The reduce task merges the values with the same key to form a smaller final result. A typical job will have many map tasks which each operate on a subset of the input data, and fewer reduce tasks, which operate on the merged output of the map tasks. Map or reduce tasks may run on different machines, allowing parallelism to be achieved. In common with functional programming design, each task is independent of other tasks of the same type, and there is no global state, or communication between maps or between reduces.

Counting term occurrences in a large data-set is an often-repeated example of how to use MapReduce paradigm[2] [5]. For this, the map function takes the document file-name (key1) and the contents of the document (value1) as input, then for each term in the document emits the term (key2) and the integer value '1' (value2). The reduce then sums up all of the values (many 1s) for each key2 (a term) to give the total occurrences of that term.

As mentioned above, MapReduce jobs are executed over multiple machines. In a typical setup, data is not stored in a central file store, but instead replicated in blocks (usually of 64MB) across many machines [7]. This has a central advantage that the map functions can operate on data that may be 'rack-local' or 'machine-local' - i.e. does not have to transit intra- and inter-data centre backbone links, and does not overload a central file storage service. Therefore high bandwidth can be achieved because data is always as local as possible to the processing CPUs. Intermediate results of map tasks are stored on the processing machines themselves. To reduce the size of this output (and therefore IO), it may be merged using a combiner, which acts as a reducer local to each machine. A central master machine provides job and task scheduling, which attempts to perform tasks as local as possible to the input data.

While MapReduce is seeing increasing popularity, there are only a few notable studies investigating the paradigm beyond the original paper. In particular, for machine learning [4], Chu et al. studied how various machine learning algorithms could be parallelised using the MapReduce paradigm, however experiments were only carried out on single systems, rather than a cluster of machines. In such a situation, MapReduce provides an easy framework to distribute non-cooperating tasks of work, but misses the central data locality advantage facilitated by a MapReduce framework. A similar study for natural language processing [12] used several machines, but with experimental datasets of only 88MB and 770MB, would again fail to see benefit in the data-local scheduling of tasks.

In contrast, indexing is an IO-intensive operation, where large amounts of raw data have to be read and transformed into suitable index structures. In this work, we show how indexing can be implemented in a MapReduce framework. However, the MapReduce implementation described in [5] is not available outside of Google. Instead, we use the Hadoop [1] framework, which is an open-source Java implementation of MapReduce from the Apache Software Foundation, with developers contributed by Yahoo! and Facebook, among others. In the next section, we describe several indexing strategies in MapReduce, starting from that proposed in the original MapReduce paper [5], before developing a more refined strategy inspired by the single-pass indexing described in Section 2.2.

4. INDEXING IN MAPREDUCE

In this section, we show how indexing can be performed in MapReduce. Firstly, we describe two possible interpretations of indexing as envisaged by Dean & Ghemawat in their original seminal MapReduce paper [5] (Section 4.1). Then, we describe an alternative MapReduce indexing strategy used by the Nutch IR platform, before finally showing how a more refined single-pass indexing strategy can be implemented in MapReduce (Section 4.3).

It should be noted that in MapReduce each map task is not aware of its context in the overall job. For indexing, this means that the doc-IDs emitted from the map phases cannot be globally correct. Instead, these doc-IDs start from 0 in each map. To allow the reduce tasks to calculate the correct doc-IDs, each map task produces a "side-effect" file, detailing the number of documents emitted per map. This is true for all the indexing implementations described in this section. We also note that for all our indexing implementations the number of reducers specified depicts the number of final indices generated.

4.1 Dean & Ghemawat's MapReduce Indexing Strategy

The original MapReduce paper by Dean & Ghemawat [5] presents a short description for performing indexing in MapReduce, which is directly quoted below:

"The map function parses each document, and emits a sequence of <word, document ID> pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a <word, list(document ID)> pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions."

The implicit claim being made in the original MapReduce paper [5] is that efficient indexing could be trivially implemented in MapReduce. However, we argue that this oversimplifies the details, and provides room for a useful study to allow document indexing in MapReduce to be better understood. For example, for an inverted index to be useful, the term frequencies within each document need to be stored. Though this is not accounted for in Dean & Ghemawat's paper, there are two possible interpretations on how this could be achieved within the bounds laid out in the quotation above. We detail these interpretations below in Sections 4.1.1 and 4.1.2, respectively.

4.1.1 Emitting Term,Doc-ID Tuples

The literal interpretation of the description above would be to output a set of <term, doc-ID> pairs for each token in a document. This means that if a single term appears n times in a document then the <term, doc-ID> pair will be emitted n times. This has the advantage of making the map phase incredibly simple, as it emits on a per token basis. However, this means that we will emit a <term, doc-ID> pair for every token in the collection. In general, when a map task emits lots of intermediate data, this will be saved to the machine's local disk, and then later transferred to the appropriate reducer. However, with this indexing interpretation, the intermediate map data would be extremely large - indeed, similar to the size of the corpus, as each token in the corpus is emitted along with a doc-ID. Having large amounts of intermediate map data will increase map to reducer network traffic, as well as lengthening the sort phase. These are likely to have an effect on the job's overall execution time. The reducer will - for each unique term - sort the doc-IDs, then add up the instances on a per doc-ID basis to retrieve the term frequencies. Finally, the reducer will write the completed posting list for that term to disk. Figure 1 provides a pseudo-code implementation of map and reduce functions for this strategy.

[2]A worked example and associated source code is available at `http://hadoop.apache.org/core/docs/r0.19.0/mapred_tutorial.html`

Dean & Ghemawat MapReduce Indexing - Map function pseudo-code

```
1: Input
       Key: Document Identifier, Name
       Value: Contents of the Document, DocContents
2: Output
       A list of (term,doc-ID) pairs, one for each token
       in the document
3: for each Term in the DocContents loop
4 :    Stem(Term)
5 :    deleteIfStopword(Term)
6 :    if (Term is not empty) then emit(Term, doc-ID)
7: end loop
8: Add document to the Document Index
9: if (lastMap()) write out information about the
10:    documents this map processed ("side-effect" files)
```

Dean & Ghemawat MapReduce Indexing - Reduce function pseudo-code

```
1: Input
       Key: A Term
       Value: List of (doc-ID), doc-IDs
2: Output
       Key: Term
       Value: Posting List
3 : List Posting-List = new PostingList()
4 : Sort doc-IDs
5 : for each doc-ID in doc-IDs loop
6 :    increment tf for doc-ID
7 :    correct doc-ID
8 :    add doc-ID and tf to Posting-List
9 : end loop
10: emit(Posting-List)
```

Figure 1: Pseudo-code interpretation of Dean & Ghemawat's MapReduce indexing strategy (map emitting <term,doc-ID>, Section 4.1.1).

4.1.2 Emitting Term,Doc-ID,TF Tuples

We claim that emitting once for every token extracted is wasteful of resources, causing excessive disk IO on the map by writing intermediate map output to disk, and excessive disk IO in moving map output to the reduce tasks. To reduce IO, we could instead emit <term,(doc-ID, tf)> tuples, where tf is the term frequency for the current document. In this way, the number of emit operations which have to be done is significantly reduced, as we now only emit once per unique term per document. The reduce method for this interpretation is also much simpler than the earlier interpretation, as it only has to sort instances by document to get the final posting list to write out. It should also be noted that the <term, doc-ID> strategy described earlier, can be adapted to generate tfs instead through the use of a MapReduce combiner, which performs a localised merge on each map task's output.

While the <term,(doc-ID, tf)> indexing strategy emits significantly less than that described in Section 4.1.1, we argue that an implementation in this manner would still be inefficient, because a large amount of IO is still required to store, move and sort the temporary map output data.

4.2 Nutch's MapReduce Indexing Strategy

The Apache Software Foundation's open source Nutch platform [3] also deploys a MapReduce indexing strategy, using the Hadoop MapReduce implementation. By inspection of the source of Nutch v0.9, we have determined that the MapReduce indexing strategy differs from the general outline described in Section 4.1 above. Instead of emitting terms, Nutch only tokenises the document during the map phase, hence emitting <doc-ID, analysed-Document> tuples from the map function. Each analysed-Document contains the textual forms of each term and their corresponding frequencies. The reduce phase is then responsible for writing all index structures. Compared to emitting <term,(doc-ID, tf)>, the Nutch indexing method will emit less, but the value of each emit will contain substantially more data (i.e. the textual form and frequency of each unique term in the document). We believe this is a step-forward towards reducing intermediate map output. However, there may still be scope for further reducing map task output to the benefit of overall indexing efficiency. In the next section, we develop our single-pass indexing strategy (described in Section 2.2) for the MapReduce framework, to address this issue.

4.3 Single-pass MapReduce Indexing Strategy

We now adapt the single-pass indexing strategy described in Section 2.2, for use in a MapReduce framework. The indexing process is split into m map tasks. Each map task operates on its own subset of the data, and is similar to the single-pass indexing corpus scanning phase. However, when memory runs low or all documents for that map have been processed, the partial index is flushed from the map task, by emitting a set of <term, posting list> pairs. The partial indices (flushes) are then sorted by term, map and flush numbers before being passed to a reduce task. As the flushes are collected at an appropriate reduce task, the posting lists for each term are merged by map number and flush number, to ensure that the posting lists for each term are in a globally correct ordering. The reduce function takes each term in turn and merges the posting lists for that term into the full posting list, as a standard index. Elias-Gamma compression is used as in non-distributed indexing to store only the distance between doc-IDs. Figure 2 provides a pseudo-code implementation of map and reduce functions for our proposed MapReduce indexing strategy.

The fundamental difference between this strategy and that of Dean & Ghemawat described in Section 4.1, is what the map tasks emit. Instead of emitting a batch of <term,doc-ID> pairs immediately upon parsing each document, we instead build up a posting list for each term in memory. Over many documents, memory will eventually be exhausted, at which time all currently stored posting lists will be flushed as <term,posting list> tuples. This has the positive effect of minimising both the size of the map task output, as well as the number of emits. Compared to the Dean & Ghemawat indexing strategies, far less emits will be called, but emits will be much larger. Compared to the Nutch MapReduce indexing strategy, there may more emits, however, the reduce task is operating on term-sorted data, and does not require a further sort and invert operation to generate an inverted index. Moreover, the emit values will only contain doc-IDs instead of textual terms, making them considerably smaller.

Figure 3 presents an example for a distributed setting MapReduce indexing paradigm of 200 documents. The documents are indexed by $m = 2$ map tasks, before the posting lists for each term are grouped and sorted, and then reduced to a single index. The posting lists output from each map contains only local doc-IDs. In the reduce tasks, these are merged into a list of absolute doc-IDs, by adding to each

Single-Pass MapReduce Indexing - Map function pseudo-code

```
1: Input
       Key: Document Identifier, Name
       Value: Contents of the Document, DocContents
2: Output
       Key: Term
       Value: Posting list
3: for each Term in the DocContents loop
4 :    Stem(Term)
5 :    deleteIfStopword(Term)
6 :    if (Term is not empty) then add the current
       document for that term to the in-memory
       Posting List
7: end loop
8: Add document to the Document Index
9: if (lastMap() or outOfMemory()) then
       emit(in-Memory Posting List)
10: if (lastMap()) write out information about the
11:    documents this map processed ("side-effect" files)
```

Single-Pass MapReduce Indexing - Reduce function pseudo-code

```
1: Input
       Key: A Term
       Value: List of (Posting List), PartialPostingLists
2: Output
       Key: Term
       Value: Posting List
3 : List Posting-List = new PostingList()
4 : Sort PartialPostingLists by the map and flush they
       were emitted from
5 : for each PostList in PartialPostingLists loop
6 :    for each doc-ID in PostList loop
7 :        correct doc-ID
8 :        Merge PostList into Posting-List
9 :    end loop
10: end loop
11: emit(Posting-List)
```

Figure 2: Pseudo-code for our proposed single-pass MapReduce indexing strategy (Section 4.3).

entry the number of documents processed by previous map tasks. However, note that in our indexing implementation, the doc-IDs are flush-local as well as map-local. While this is not strictly necessary, it allows smaller doc-IDs to be emitted from each map, which can be better compressed.

5. EXPERIMENTS & RESULTS

In the following experiments, we aim to determine the efficiency of multiple indexing implementations. Specifically, we investigate whether distributed indexing as laid out in the original MapReduce paper (Section 4.1) is fit for purpose. We compare this to our single-pass indexing strategy developed both for a single machine architecture (Section 2) and for MapReduce (Section 4.3). Note that in this paper we do not investigate Nutch's MapReduce indexing strategy, however we expect it to be more efficient than Dean & Ghemawat's indexing strategy, while being less efficient than our single-pass indexing strategy. We leave this for future work. Furthermore, we investigate these approaches in terms of scalability as the number of machines designated for work is increased, and experiment with various parameters

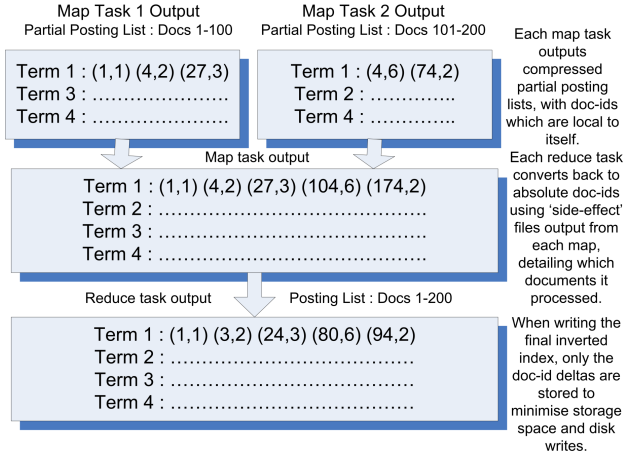


Figure 3: Correcting document IDs while merging.

in MapReduce to determine how to most efficiently apply it for indexing.

5.1 Research Questions

To measure the efficiency of our indexing implementations and therefore the suitability (or otherwise) of MapReduce for indexing, we investigate 3 important research questions, which we address by experimentation in the remainder of this section:

1. Can a practical application of the distributed indexing strategy described in Section 2 be sufficient for large-scale collections when using many machines? (Section 5.4)
2. When indexing with MapReduce, what is the most efficient number of maps and reduces to use? (Section 5.5)
3. Is MapReduce Performance Close to Optimal Distributed Indexing? (Section 5.6)

5.2 Evaluation Metrics

Research questions 1-3 require a metric for indexing performance. For this, we measure the throughput of the system, in terms of MB/s (megabytes per second). We calculate throughput as $collectionsize/timetaken$ where collection size is the compressed size on disk for a single copy of the collection in MB (megabytes). The time taken is the full time taken by the job (including setup) measured in seconds.

Research question 3 mandates suitability for indexing at a large scale. We measure suitability in terms of throughput (as above) and in terms of speedup. Speedup S_m, defined as $S_m = \frac{T_1}{T_m}$, where m is the number of machines, T_1 is the execution of the algorithm on a single machine, and T_m is the execution time in parallel, using m machines [9]. This encompasses the idea that not only should speed improve as more resources are added, but that such a speed increase should reflect the quantity of those resources. For instance, if we increase the available resources by a factor of 2, then it would be desirable to get (close to) twice the speed. This is known as linear speedup (where $S_m = m$), and is the ideal scenario for parallel processing. However, linear speedup can be hard to achieve in a parallel environment, because of the growing influence of small sequential sections of code as the number of processors increases (known as Amdahl's law [2]), or due to overheads.

5.3 Experimental Setup

Following [24], which prescribes guidelines for presenting indexing techniques, we now give details of our experimen-

Number of Machines (Cores)	1(3)	2(6)	4(12)	6(18)	8(24)
Distributed Single-Pass	2.44	4.6	12.8	12.4	12.8
Dean & Ghemawat MapReduce	1.15	1.59	4.01	4.71	6.38
MapReduce Single-Pass	2.59	5.19	9.45	13.16	17.31

Table 1: Throughput as the number of machines allocated is increased using using a variety of indexing strategies, measured in MB/sec.

tal cluster setup, consisting of 19 identical machines. Each machine has a single Intel Xeon 2.4GHz processor with 4 cores, 4GB of RAM, and contains three hard drives: One 160GB hard disk, spinning at 7200rpm with an 8MB buffer, is used for the operating system and temporary job scratch space; Two 400GB hard disks, each spinning at 7200rpm with a 16MB buffer, are dedicated for distributed file system storage. Each machine is running a copy of the open source Linux operating system Centos 5 and are connected together by a gigabit Ethernet connection on a single rack. The Hadoop (version 0.18.2) distributed file system (DFS) is running on this cluster, replicating files to the distributed file storage on each machine. Each file on the DFS is split into 64MB blocks, which are each replicated to 2 machines[3]. While each machine has four processors available at any one time, only three of these are valid targets for job execution, the last processor is left free for the distributed file system software running on each machine. As our cluster is shared by several users, job allocation is done by Hadoop on Demand (HOD) running with the Torque resource manager (version 2.1.9) rather than using a dedicated Hadoop cluster. Machines not allocated to a MapReduce job are available to be scheduled by Torque for other jobs not associated with MapReduce. However on such nodes, the fourth processor core is still free for distributed file system work[4]. We also have in the same rack a RAID5 centralised file server powered by 8 Intel Xeon 3GHz processor cores for use with non-MapReduce jobs, providing network file system (NFS) storage. For consistency, in the following experiments, we employ the standard TREC web collection .GOV2. This is an 80GB (425GB uncompressed) crawl of *.gov* Web domain comprising over 25 million documents. Before the advent of ClueWeb09, .GOV2 was the largest available TREC corpus.

5.4 Is Distributed Indexing Good Enough?

First we determine if MapReduce is necessary for large-scale indexing. If a simple distribution of the non-parallel indexing strategy described in Section 2 is sufficient to index large collections then there is no need for MapReduce. To evaluate this, we distribute the single-pass indexing strategy across n machines in our cluster, where we vary $n = \{1, 2, 4, 6, 8\}$. To provide a comparative baseline, the non-parallel single-pass indexing implementation in Terrier can index the .GOV2 corpus on a single processor core (*not* machine) in just over 1 day using the same algorithm. This translates into a throughput of approximately 1MB/sec. For distributed indexing to be sufficient for indexing large collections, throughput should increase in a (close-to) linear fashion with the number of processing cores added. As

[3]This is lower than the Hadoop default of 3, to conserve distributed file system space.

[4]Hence, as each machine always has one processing core free to handle distributed file system traffic, and the network traffic of other cluster jobs is assumed to be low, then there should be no impact on the validity of the experiments.

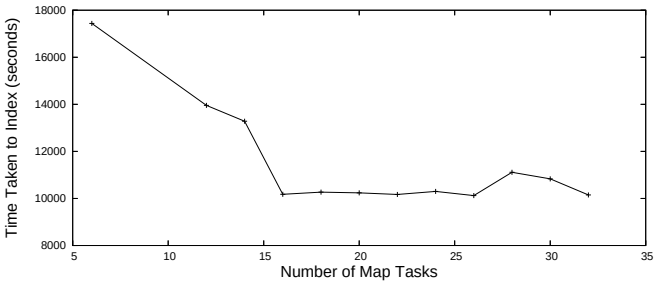


Figure 4: The effect of varying the number of map tasks on indexing time (seconds) of .GOV2 collection: 4 machines, 1 reduce task.

mentioned in Section 2.3, when distributed indexing uses machine-local data, indexing will achieve exactly linear scaling. However, unless the document data is already present on the machines (e.g. indexing takes place on the machines which crawled the documents), there would be the need to copy the required data to the indexing machines. In many other scenarios, crawling or documents corpora storage may not be on indexing machines. Moreover, local-only indexing is not resilient to machine failure. Instead, we experiment with the shared-corpus distributed indexing, where the corpus is indexed over NFS from a central fileserver. Local data (shared-nothing) indexing would require the corpus subset to be copied prior to indexing.

Table 1, row 1, shows how throughput increases as we allocate more machines (recall that each machine adds three processor cores for indexing work). Here we can see that throughput indeed increases in a reasonable fashion, However, once we allocate more than 4 machines we observe no further speed improvements. This is caused by our central file store becoming a bottleneck as it is unable to serve all the allocated machines simultaneously. We can therefore conclude that this distribution method is unsuitable for large-scale indexing using our hardware setup. Moreover, we argue that even with better hardware this issue cannot be overcome as the file server(s) will always be slower than the combination of all worker machines.

5.5 Investigating MapReduce Parameters

In Section 5.4, we showed that the distributed indexing strategy described in Section 2 is unsuitable for the scalable distributed shared-corpus indexing of large collections. However, before we can evaluate MapReduce as an alternate solution we need to investigate how to maximise its efficiency in terms of its input parameters. The fundamental parameters of a MapReduce job are m - the number of map tasks that the input data is divided across - and r, the number of reduce tasks. A higher number of map tasks means that the input collection of documents is split into smaller chunks, but also that there will be more overheads, as more tasks have to be initialised and latterly cleared. To determine what effect this has on performance, we vary m while indexing the .GOV2 corpus, using a set 4 machines. The results - in terms of indexing time - are shown in Figure 4. We see that when the number of maps is small (i.e. less than the 12 processors available from the 4 machines), parallelism is hindered, as not all processors have work to do. When the number of map tasks is ≤ 14, we also note that indexing time is still high. On examination of these jobs, we found

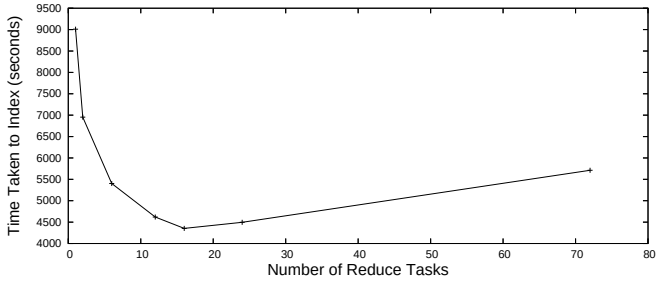


Figure 5: The effect of varying the number of reduce tasks on indexing time (seconds) of .GOV2 collection: 6 machines, 72 map tasks.

that the balance of work between map tasks was not even, with one map task taking markedly longer than the others[5]. When the number of map tasks is increased to 16, balance is restored.

In previous work [14], we have shown that the time taken by the reduce step is an important factor in determining indexing performance. Therefore, it is important to know how many reduce tasks it is is optimal to create - subject to external constraints on the number of reducers (*e.g.* having 8 query servers suggests 8 reducers are used so that 8 final indices are created). To test the effect of the number of reduce tasks on efficiency, we index .GOV2 while varying the number of reduce tasks. Here we used 6 machines and 72 map tasks. The indexing time results are shown in Figure 5. As we would expect, while the number of reduces is below the available processors (for the 6 machines allocated, 18 processors) the speed increases as we add more reducers, since we are effectively providing more parallel processing power. Once we are beyond the number of processors however, indexing time increases. This is intuitive, as there is more work to be done than available processors. Therefore, we can conclude that the number of reduce tasks should be a multiple of the number of processors. Unlike map tasks, however, there is an incentive to have less reduce tasks, resulting in fewer indices, but this needs to be traded off against the possibility of failures and the associated time wasted through re-running.

5.6 Is MapReduce Performance Close to Optimal Distributed Indexing?

We now investigate whether MapReduce is an efficient alternative to distributed indexing. Moreover, we evaluate MapReduce against optimal distributed indexing in terms of performance, i.e. the extent to which it scales close to linearly with processing power. The core advantage of MapReduce is the ability to apply the distributed file system (DFS) to avoid centralised storage of data (creating a single point of failure), and to take advantage of data locality to avoid excess network IO. This meanwhile, is at the cost of additional overheads in job setup, monitoring and control, as well as the additional IO required to replicate the data on a DFS. As the centralised file-system was identified as the bottleneck for distributed indexing, we would expect MapReduce to perform better since it uses a DFS. For evaluation, we perform a direct comparison on throughput between indexing strategies. Note that while distributed indexing creates n index shards, where n is the number of processors allocated, MapReduce instead produces r index shards where r is the number of reduce tasks created. For these experiments we always allocate 72 map tasks and 24 reduce tasks. This means that for distributed indexing a smaller number of index shards were created when indexing on $\{1, 2, 4, 6\}$ machines. However, we believe that this has no significant impact on the overall throughput.

First, we investigate whether the MapReduce indexing strategy proposed by Dean & Ghemawat is more efficient than distributed indexing. Table 1 shows how the throughput increases as we allocate more machines - in particular, row 2 shows results for Dean & Ghemawat's strategy, interpreted as emitting term <doc-ID,tf> tuples (Section 4.1.2). We also implemented the other interpretation which emits term,doc-ID tuples, however, it consumed excessive temporary storage space during operation due to its large number of emit operations. This made it impossible to determine throughput, as the worker machines ran out of disk space causing the job to fail. Our implementation of Dean & Ghemawat's indexing method also creates the additional data structures described in Section 2.1 - i.e. the lexicon and document index - and uses the compressed Terrier inverted index format. From Table 1, row 2, we can see that this implementation performs very poorly in comparison to distributed indexing. Indeed, with 8 machines it indexes only at half the speed of distributed indexing with the same number of machines. Upon further investigation, as expected, this speed degradation can be attributed to the large volume of map output which is generated by this approach. However, it should be noted that unlike distributed indexing, performance improvements do not stall after 4 machines. This would indicate that while the indexing strategy is poor, MapReduce in general will continue to garner performance improvements as more machines are added. Therefore, we believe this makes it more suitable for processing larger corpora, where larger clusters of 100s-1000s of machines are needed to index them in reasonable amounts of time.

We now experiment with our proposed implementation of single-pass indexing in MapReduce, as described in Section 4.3. Our expectation is that this strategy should prove to be more efficient as it lowers disk and network IO by building up posting lists in memory, thereby minimising map output size. Table 1, row 3 shows the throughput of the single-pass MapReduce indexing strategy. In comparison to Dean & Ghemawat's indexing strategy, we find our approach to be markedly faster. Indeed, when using 8 machines our method is over 2.7 times faster. Moreover, Figure 6 shows the speedup achieved by both approaches as the number of machines is increased. We observe that our single-pass based strategy scales close to linearly in terms of indexing time as the number of machines allocated for work is increased. In contrast, the scalability of Dean & Ghemawat's approach is noticeably worse (5.5 times for 8 processors, versus 6.8 times for single-pass based indexing). We believe that this makes our proposed strategy suitable for scaling to large clusters of machines, which is essential when indexing new large-scale collections like ClueWeb09.

[5]Hadoop actually supports *speculative execution*, where two copies of the last task, or the slowest tasks, will be started. Only output from the first successful task to complete will be used. This uses otherwise idle processing power to decrease average job duration.

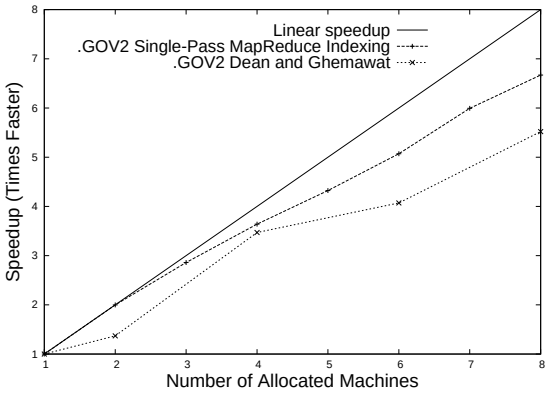


Figure 6: Speedup of .GOV2 indexing as more MapReduce machines are allocated.

6. CONCLUSION

In this paper, we detailed four different strategies for applying document indexing within the MapReduce paradigm, with varying efficiency. In particular, we firstly showed that indexing speed using a distributed indexing strategy was limited by accessing a centralised file-store, and hence the advantage of using MapReduce to allocate indexing tasks close to input data is clear. Secondly, we showed that the MapReduce indexing strategy suggested by Dean & Ghemawat in the original MapReduce paper [5] generates too much intermediate map data, causing an overall slowness of indexing. In contrast, our proposed single-pass indexing strategy is almost 3 times faster, and scales well as the number of machines allocated is increased.

Overall, we conclude that the single-pass based MapReduce indexing algorithm should be suitable for efficiently indexing larger corpora, including the recently released TREC ClueWeb09 corpus. Moreover, as a framework for distributed indexing, MapReduce conveniently provides both data locality and resilience. Finally, it is of note that an implementation of the MapReduce single-pass indexing strategy described in this paper is freely available for use by the community as part of the Terrier IR Platform[6].

7. REFERENCES

[1] Apache Software Foundation. The apache hadoop project. `http://hadoop.apache.org/`, as of 15/06/2009.

[2] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *Proc. of AFIPS*, pp. 483–485, 1967.

[3] M. Cafarella and D. Cutting. Building nutch: Open source search. *ACM Queue*, 2(2):54–61, 2004.

[4] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *Proc. of NIPS 2006*, pp. 281–288.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. of OSDI 2004*, pp. 137–150.

[6] P. Elias. Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on*, 21(2):194–203, 1975.

[6] `http://terrier.org`

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[8] S. Heinz and J. Zobel. Efficient single-pass index construction for text databases. *JASIST*, 54(8):713–729, 2003.

[9] M. D. Hill. What is scalability? *SIGARCH Comput. Archit. News*, 18(4):18–21, 1990.

[10] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. of EuroSys 2007*, pp. 59–72.

[11] R. E. Johnson. Frameworks = (components + patterns). *Commun. ACM*, 40(10):39–42, 1997.

[12] M. Laclavik, M. Seleng, and L. Hluchý. Towards large scale semantic annotation built on mapreduce architecture. In *Proc. of ICCS (3)*, pp. 331–338, 2008.

[13] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[14] R. McCreadie, C. MacDonald, and I. Ounis. On single-pass indexing with mapreduce. In *Proc. of SIGIR 2009*, in press.

[15] S. Melnik, S. Raghavan, B. Yang, and H. Garcia-Molina. Building a distributed full-text index for the web. In *Proc. of WWW 2001*, pp. 396–406.

[16] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proc. of SIGMOD 2008*, pp. 1099–1110.

[17] O. O'Malley and A. C. Murthy. Winning a 60 second dash with a yellow elephant. TR, Yahoo! Inc., 2009.

[18] I. Ounis, G. Amati, V. Plachouras, B. He, C. Macdonald, and C. Lioma. Terrier: A high performance and scalable information retrieval platform. In *Proc. of OSIR workshop, SIGIR-2006*, pp. 18–25.

[19] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[20] B. A. Ribeiro-Neto, E. S. de Moura, M. S. Neubert, and N. Ziviani. Efficient distributed algorithms to build inverted files. In *Proc. of SIGIR 1999*, pp. 105–112.

[21] E. Schonfeld. Yahoo! search wants to be more like google, embraces hadoop, 2008. `http://www.techcrunch.com/2008/02/20/yahoo-search-wants-to-be-more-like-google-embraces-hadoop/`, as of 15/06/2009.

[22] A. Tomasic and H. Garcia-Molina. Performance of inverted indices in shared-nothing distributed text document information retrieval systems. In *Proc. of PDIS 1993*, pp. 8–17.

[23] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.

[24] J. Zobel, A. Moffat, and K. Ramamohanarao. Guidelines for presentation and comparison of indexing techniques. *SIGMOD Record*, 25(3):10–15, 1996.

Strong Ties vs. Weak Ties: Studying the Clustering Paradox for Decentralized Search

Weimao Ke
Laboratory of Applied Informatics Research
School of Information and Library Science
& Translational and Clinical Sciences Institute
University of North Carolina at Chapel Hill
wke@unc.edu

Javed Mostafa
Laboratory of Applied Informatics Research
School of Information and Library Science
& Translational and Clinical Sciences Institute
University of North Carolina at Chapel Hill
jm@unc.edu

ABSTRACT

We studied decentralized search in information networks and focused on the impact of network clustering on the findability of relevant information sources. We developed a multi-agent system to simulate peer-to-peer networks, in which peers worked with one another to forward queries to targets containing relevant information, and evaluated the effectiveness, efficiency, and scalability of the decentralized search. Experiments on a network of 181 peers showed that the *RefNet* method based on topical similarity cues outperformed *random walks* and was able to reach relevant peers through short search paths. When the network was extended to a larger community of 5890 peers, however, the advantage of the *RefNet* model was constrained due to noise of many topically irrelevant connections or weak ties.

By applying topical clustering and a *clustering exponent* α to guide network rewiring, we studied the role of *strong ties* vs. *weak ties*, particularly their influence on distributed search. Interestingly, an inflection point was discovered for α, below which performance suffered from many remote connections that disoriented searches and above which performance degraded due to lack of *weak ties* that could move queries quickly from one segment to another. The inflection threshold for the 5890-peer network was $\alpha \approx 3.5$. Further experiments on larger networks of up to 4 million peers demonstrated that clustering optimization is crucial for decentralized search. Although overclustering only moderately degraded search performance on small networks, it led to dramatic loss in search efficiency for large networks. We explain the implication on scalability of distributed systems that rely on clustering for search.

Categories and Subject Descriptors

H.3.4 [**Information storage and retrieval**]: Systems and Software—*Distributed systems, Information networks*

LSDS-IR Workshop. July 2009. Boston, USA.

General Terms

Algorithms, Performance, Experimentation

Keywords

clustering, decentralized search, P2P IR, resource discovery, referral network, agent, weak tie, strong tie, topical locality

1. INTRODUCTION

Information exists in many distributed networked environments, where a centralized repository is hardly possible. In a *peer-to-peer* (P2P) environment, individual peers host separate collections and interact with one another for information sharing and retrieval [18], exemplifying a large, dynamic, and heterogeneous networked information space. Efficient network navigation is critically needed in today's distributed environments, e.g., to route queries to relevant information sources or to deliver information items to peers of interest.

Research has found *clustering* useful for information retrieval. The *Cluster Hypothesis* states that relevant documents are more similar to one another than to non-relevant documents and therefore closely related documents tend to be relevant to the same requests [29]. Traditional IR research utilized document-level clustering to support exploratory searching and to improve retrieval effectiveness [12, 9, 14]. Distributed information retrieval, particularly unstructured peer-to-peer IR, relied on peer-level clustering for better decentralized search efficiency. Topical segmentation based techniques such as semantic overlay networks (SONs) have been widely used for efficient query propagation and high recall [3, 7, 17, 8]. Hence, overall, clustering was often regarded as beneficial whereas the potential *negative* impact of clustering (or over-clustering) on retrieval has rarely been scrutinized.

Research on complex networks indicated that a proper degree of network clustering with some presence of remote connections has to be maintained for efficient searches [15, 25, 30, 16, 24, 6]. Clustering reduces the number of "irrelevant" links and aids in creating topical segments useful for orienting searches. With very strong clustering, however, a network tends to be fragmented into local communities with abundant *strong ties* but few *weak ties* to bridge remote parts [10]. Although searches might be able to move gradually to targets, necessary "hops" become unavailable.

We refer to this phenomenon as the *Clustering Paradox*, in which neither strong clustering nor weak clustering is de-

sirable. In other words, trade-off is required between *strong ties* for search orientation and *weak ties* for efficient traversal. In Granovetter's terms, whereas *strong ties* deal with local connections within small, well-defined groups, *weak ties* capture between-group relations and serve as bridges of social segments [10]. The *Clustering Paradox*, seen in light of strong ties and weak ties, has received attention in complex network research and requires further scrutiny in a decentralized IR context.

In this study, we examined network characteristics and search optimization in a fully decentralized retrieval context. We focused on the effect of network clustering, i.e., strong ties vs. weak ties, on the efficient findability of relevant information sources. Outcome of this research will provide guidance on how an information network can be structured or self-organized to better support efficient discovery of relevant information sources that are highly distributed.

2. RELATED WORK

In an open, dynamic information space such as a peer-to-peer network, people, information, and technologies are all mobile and changing entities. Identifying where relevant collections are for the retrieval of information is essential. Without global information, decentralized methods have to rely on local intelligence of distributed peers to collectively construct paths to desired targets.

2.1 P2P Information Retrieval

In some respect, decentralized IR in networks is concerned with the cost of traversing a network to reach desired information sources. Unstructured or loosely structured peer-to-peer networks represent a connected space self-organized by individuals with local objectives and constraints, exhibiting a topological underpinning on which all can collectively scale [1, 18].

While federated IR research has made advances in enabling searches across hundreds of repositories, a P2P network usually has a much larger number of participants who dynamically join and leave the network, and only offer idle computing resources for sharing and searching [34]. Usually there is no global information about available collections; seldom is there centralized control or a central server for mediating [18, 8].

Recent years have seen growing popularity of peer-to-peer (P2P) networks for large scale information sharing and retrieval [18]. With network topology and placement of content tightly controlled, *structured* peer-to-peer networks have the advantage of search efficiency [27, 21, 5, 19, 26]. However, their ability to handle unreliable peers and a transient population was not sufficiently tested. *Unstructured* overlay systems work in an indeterministic manner and have received increased popularity for being fault tolerant and adaptive to evolving system dynamics [18, 8].

As the peer-to-peer paradigm becomes better recognized for IR research, there have been ongoing discussions on the applicability of existing P2P search models for IR, the efficiency and scalability challenges, and the effectiveness of traditional IR models in such environments [33]. Some researchers applied Distributed Hashing Tables (DHTs) techniques to *structured* P2P environments for distributed retrieval and focused on building an efficient indexing structure over peers [5, 19, 26]. Others, however, questioned the sufficiency of DHTs for dealing with high dimensionality of IR in dynamic P2P environments [3, 18, 17]. For information retrieval based on a large feature space, which often requires frequent updates to cope with a transient population, it is challenging for distributed hashing to work in a traffic- and space-efficient manner.

2.2 Clustering and Decentralized Search

In recent years, topical segmentation based techniques such as semantic overlay networks (*SONs*) have been widely used for P2P IR, in which peers containing similar information formed semantic groups for efficient searches [3, 7, 28, 17, 20]. Clustering, often in the form of hierarchical segments, was the key idea for bringing similar peers together in a more organized way so that topically relevant peers or information sources can be quickly identified. Existing P2P IR research, however, often assumed the unitary benefit of clustering and rarely scrutinized its potential negative impact on decentralized search.

Research on complex networks has found that efficient searching in some properly clustered networks is more promising than in others. Kleinberg (2000) studied decentralized search in small world using a two dimensional model, in which peers had rich connections with immediate neighbors and sparse associations with remote ones [15]. The probability p_r of connecting to a neighbor beyond the immediate neighborhood was proportional to $r^{-\alpha}$, where r was the topical (search) distance between the two and α a constant called *clustering exponent*[1]. It was shown that only when *clustering exponent* $\alpha = 2$, search time (i.e., search path length) was optimal and bounded by $c(\log N)^2$, where N was the network size and c was some constant [15].

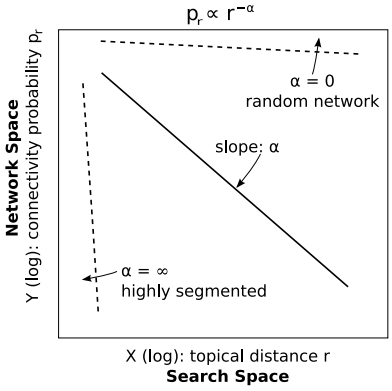


Figure 1: Network Clustering

The *clustering exponent* α, as shown in Figure 1, describes a correlation between the network (topological) space and the search (topical) space [15, 6]. When α is large, weak ties (long-distance connections) are rare and strong ties dominate [10]. The network becomes highly segmented. When α is small, connectivity has little dependence on topical closeness – local segments become less visible as the network is built on increased randomness. In this way, the *clustering exponent* α influences the formation of local clusters and overall network clustering.

It was further demonstrated that optimal value of α for search depends on dimensionality of the search space. Specif-

[1]The *clustering exponent* α is also known as the *homophily exponent* [30, 24].

ically, when $\alpha = d$ on a d-dimension space, decentralized search is optimal. Further studies conducted by various research groups have shown consistent results [30, 16, 24, 6]. These findings require closer scrutiny in an IR context where some assumptions might be voilated, e.g, when orthogonal feature dimensions cannot be precisely defined.

3. APPROACH OVERVIEW

We have developed a decentralized search architecture named *RefNet* for finding distributed information sources in a simulated networked environment. We relied on multi-agent systems to study the problem of decentralized search and focused on the impact of clustering in an information retrieval context. Similar agent-based approaches have been adopted by various research groups to study efficient information retrieval, resource discovery, service location, and expert finding in decentralized peer-to-peer environments [25, 32, 36, 35]. One common goal was to efficiently route a query to a relevant agent or peer[2]. We illustrate the conceptual model in Figure 2 and elaborate on major components.

Assume that agents or peers, representatives of information seekers, providers (sources), and mediators, reside in an n dimensional space. An agent's location in the space represents its information topicality. Therefore, finding relevant sources for an information need is to route the query to agents in the *relevant* topical space. To simplify the discussion, assume all agents can be characterized using a two-dimensional space. Figure 2 visualizes a 2D representation of the conceptual model. Let agent A_u be the one who has an information need whereas agent A_v has the relevant information. The problem becomes how agents in the connected society, without global information, can collectively construct a short path to A_v. In Figure 2, the query traverses a referral chain $A_u \rightarrow A_b \rightarrow A_c \rightarrow A_d \rightarrow A_v$ to reach the target. While agents A_b and A_d help move the query on the horizontal dimension, agent A_c primarily works on the vertical dimension and has a remote connection for the query to jump.

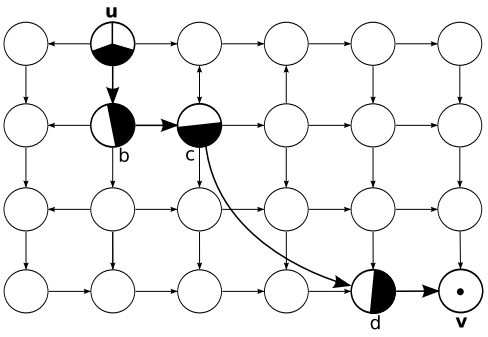


Figure 2: Conceptual Model of RefNet. A circle represents an agent or peer. The black/white segments of each circle illustrate agent representation according to its topical dimensions (coverage).

3.1 Local Indexing & Classification

For decentralized search, direction matters. Pointing to the right direction to the relevant topical space means the agents or peers have some ability to differentiate items on certain dimensions. For instance, one should be able to tell if a query is related to mathematics or not in order to route the query properly on that dimension. Each agent derives clusters or major topics from its local information collection through *document clustering*[3]. The local index provides the basis of an agent's "knowledge" and enables abstraction of queries. Now, when a query is routed to it, the agent will be able to tell what it is about and assign a label to it through *query classification* based on identified clusters [23]. The label associated with the query serves as a clue for potential referral directions.

3.2 Neighbor Selection

Pointing to the right direction also requires that each agent or peer knows which neighbor(s) should be contacted given a labeled query. Therefore, there should be a mechanism of mapping classification output to a potential *good* neighbor. By *good neighbor*, we mean agents on a short path to the targeted information space – either the neighbor is likely to have a relevant information collection to answer the query directly or in a neighborhood closer to relevant targets. Agents explore their neighborhoods through interactions and develop knowledge of who serves or connect to what types of information collections.

3.3 Network Clustering and Rewiring

Network topology plays an important role in decentralized search. Topical segmentation based techniques such as semantic overlay networks (SONs) have been widely used for efficient peer-to-peer information retrieval [8]. Through self-organization, similar peers form topical partitions, which provide some association between the topological (network) space and the topical space to guide searches. Research has found that such an association, in the form of a *clustering exponent* α that defines an inverse relationship between connectivity probability and topical distance, is critical for efficient navigation in networks without global information [15, 16, 6]. The RefNet framework has a mechanism for clustering-based rewiring, which influences the balance of *strong ties* vs. *weak ties* for efficient routing, as illustrated in Figure 1.

4. ALGORITHMIC DETAIL

In the previous section, we proposed and described a conceptual model for decentralized search of relevant information sources. Figure 3 illustrates how various components work together within each agent. This section will elaborate on specific algorithms used in the *RefNet* model for decentralized search.

We used the Vector-Space Model (VSM) for information (document and query) representation [2]. Given that information is highly distributed, a global thesaurus was not assumed. Instead, each agent had to parse information items it individually had and produced a local thesaurus. This thesaurus was then used to represent each information item using the TF*IDF (Term Frequency * Inverse Document

[2]In this paper, the terms *agent* and *peer* are exchangeable.

[3]Note that *document clustering* refers to mining a peer's local collection of documents to identify significant topics and topical overlap whereas *network clustering* is to determine how similar peers connect to each other to form groups and is the main focus of this study.

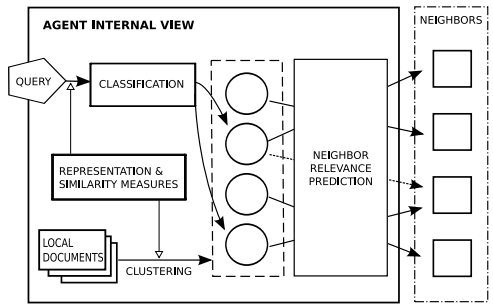


Figure 3: Agent Internal View

Frequency) weighing scheme. Note that for the DF component of TF*IDF, values were computed within the information space of an agent. This was to follow the assumption that global information was not available to individuals and it is impossible to aggregate all documents in the network to get global DF values.

Provided TF*IDF representation, pair-wise similarity values were computed based on the Cosine coefficient that measures cosine of the angle between a pair of vectors [2]. For document clustering, we used the well-known K-means method based on cosine similarities [11].

Section 4.1 elaborates on a centroid-based method for query classification. Section 4.2 introduces a single-perceptron neural network (NN) algorithm for neighbor relevance prediction given query classification output. Section 4.3 discusses the formula for rewiring based on a *clustering exponent* α. For comparison, we also adopted a *Random Walk* model. The only difference was that in *Random Walk*, an agent simply ignored the neighbor selection step in Section 4.2 and forwarded a query to a random neighbor.

4.1 Centroid-based Query Classification

Given limited information each agent has, many widely appreciated classification methods, such as the Support Vector Machine (SVM), require a fair amount of training data and are therefore not applicable [23]. In this study, we used a simple centroid-based approach that produced competitive decentralized search results on a benchmark news collection [13].

Suppose an agent had k identified clusters/classes. Each class, $c \in [c_1, c_2..c_k]$, contained a set of documents $[d_1, d_2..d_n]$. Let $W_{d|i}$ denote the weight of the i^{th} term in document d. The weight of the i^{th} term in class centroid c was computed by:

$$W_{c|i} = \frac{\sum_{d=1}^{n_c} W_{d|i}}{n_c} \quad (1)$$

where n_c was the number of documents in class c. To classify a query, the query was first locally vectorized using the TF*IDF method and then compared to each class using the cosine similarity measure. The relevance of the classes to the query was sorted using the similarity scores.

4.2 Neural-Net for Neighbor Prediction

After query classification, the relevance (or similarity) of a query to each class was known. The topical relevance scores were then used to infer which neighbor was the best neighbor to contact if the current agent did not have relevant information. We assumed that the association between the classification output (a vector of topics' relevance scores) and the prediction (a vector of neighbors' relevance scores) is linear. A single perceptron neural network (NN) is suitable for the estimation of linear associations [23]. In this study, we implemented a feedforward perceptron NN with backprop and a sigmoid signal transfer function (please refer to [22] for details). To initialize learning, agents interact with their neighbors and learn about their topicality by using local documents as queries.

4.3 Peer Clustering and Network Rewiring

We introduced a clustering exponent α to rewire (reconnect peers through self-organization) a network and studied its impact on decentralized search. First, for each peer, some random peers were picked and added to its existing neighbors. Then, the current peer (i) queried all these neighbors (j) to determine their topical distance r_{ij} by sending them local documents as queries. Finally, the following connectivity probability function was used by the peer to decide who should remain as neighbors:

$$P_{ij} \propto r_{ij}^{-\alpha} \quad (2)$$

where α is the *clustering exponent* (or *homophily exponent*) and r_{ij} the pairwise topical distance. The finalized neighborhood size depended on the number of neighbors before rewiring. With a positive α value, the larger the topical distance, the less likely two peers will connect. Large α values lead to a highly clustered network while small values produce many topically remote connections or weak ties.

5. EXPERIMENTAL SETUP

We constructed a peer-to-peer network by using a large scholarly communication data collection and treating each unique scholar as a peer, who possessed a local collection of documents published by the scholar (author). The task involved finding a peer with relevant topic(s) in the network, given a query. Applications of this framework include, but are not limited to, distributed IR, P2P resource discovery, expert location in work settings, and reviewer finding in scholarly networks. However, we focused on the general decentralized search problem in large networked environments.

5.1 Data Collection

Data used in the experiments were from the TREC Genomics track 2004 benchmark collection, a Medline subset of about 4.5 million citations from 1994 to 2003. The data collection included metadata about publication titles, abstracts, and authors. We chose six scholars in the medical informatics domain and identified their direct co-authors (1^{st} degree) who published 10 to 80 articles in the TREC collection, resulting in a small network of 181 peers. Then the network was extended to the 2^{nd} degree (co-authors' co-authors) to total 5890 peers for experiments on a larger scale. Both networks had a diameter (the longest of all shortest pairwise paths) of 8 and roughly followed a power-law degree distribution with irregularities on the tail. For each peer, which represented a scholar/author, all articles (with titles and abstracts) authored or co-authored by the scholar were loaded as the local information collection.

5.2 Relevant Peers and Tasks

Relevant peers or information sources are considered few, if not rare, given a particular information need. To operationalize it, we defined a relevant peer as one of those who have the most similar information to a query. Specifically, we considered those scholars whose topical (cosine) similarity to a given query was ranked above the fifth percentile. Hence, for evaluation purposes, peers were sampled to estimate a threshold similarity score for each query, which was then used in experiments to judge whether a relevant peer had been found. We retrieved citations to articles published in the Journal of the American Medical Informatics Association (JAMIA) in the Genomics track collection and used all (498) articles with titles and abstracts as simulated queries.

5.3 Software and Hardware Setup

We developed a multi-agent system called RefNet, which takes advantage of the JADE [4] agent platform and the Weka machine learning framework [31]. RefNet has integrated the two major software packages (both in Java) to facilitate research experiments on decentralized search in networked environments.

Experiments were conducted on a Linux cluster of 9 nodes, each has Dual Intel Xeon e5405 (2.0 Ghz) Quad Core Processors (8 processors), 8 GB fully buffered system memory, and a Fedora 7 installation. The nodes were connected internally through a dedicated 1Gb network switch. The agents were equally distributed among the 72 processors, each of which loaded an agent container in Java, reserved 1GB memory, and communicated to each other. The Java Runtime Environment version for this study was 1.6.0_07.

5.4 Simulation Procedures

We ran experiments on the proposed RefNet model and a random-walk model and conducted comparative analyses. In both models, agents tried to forward a query to one another until one of the following conditions was met: 1) a relevant peer was found, or 2) the search path length reached its defined maximum. When concluded, the query would follow the search path in the reverse order back to the querying peer. Multiple runs were conducted in each parameter configuration. In each run, the 498 queries were submitted to the network one after another.

After experiments on initial co-authorship networks, we introduced the *clustering exponent* α to rewire the networks and studied its impact on decentralized search. Twenty random peers were added to each existing neighborhood, which was finalized based on the connectivity probability function defined in Section 4.3. It was further required that the final neighborhood size, for each peer, was in the range between 3 and 100.

5.5 Evaluation

The dependent variables of this study were effectiveness and efficiency of decentralized searches. We used completion rate of all tasks to measure retrieval effectiveness, $R_c = \frac{N_S}{N_T}$, where N_T is the total number of queries and N_S the number of them with a relevant peer found within given parameter limits.

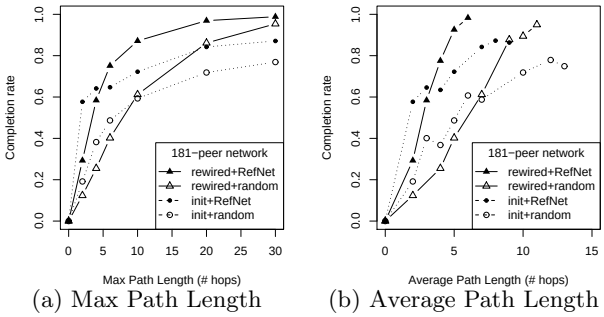
For efficiency, the maximum search path length L_{max} was controlled in each experiment and the actual path length of each task was measured. We computed average length of all searches in each experiment run, i.e., $\bar{L} = \frac{\sum_{i=1}^{N} L_i}{N_T}$, where L_i was the path length of the i_{th} query and N_T the total number of queries. With shorter path lengths, the entire distributed system is considered more efficient given fewer peers involved in computation.

For scalability, we ran experiments on different network sizes: 181 peers and 5890 peers. Effectiveness vs. efficiency patterns were compared. Various *clustering exponent* α values were controlled in experiments to examine its impact on the above variables. We further investigated the scaling of clustering impact in very large networks of up to 4 million peers based on synthetic data.

6. EXPERIMENTAL RESULTS

In this section, we present effectiveness and efficiency results on initial and rewired networks of 181 and 5890 peers, focus on the impact of clustering on decentralized search, and examine how the impact of network clustering scales.

6.1 181-Peer Network



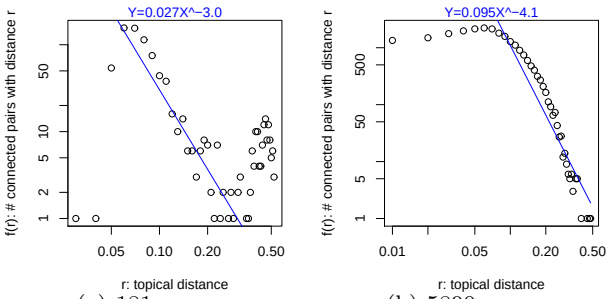
(a) Max Path Length (b) Average Path Length

Figure 4: Completion Rate (Y) vs. Path Length (X) on 181 Peers

Figure 4 shows experimental results on 181-peers networks. With the initial network (dotted lines), the *RefNet* model consistently outperformed random walks, especially within small path lengths. For instance, within two hops, RefNet already achieved a completion rate of more than 50% while random-walk was still at 20%. Increasing the path length helped both models but neither reached a completion rate higher than 90%, suggesting that there were particular characteristics of the initial network that disoriented some searches after a long path.

Clustering analysis, as plotted in Figure 5 (a) on log/log coordinates, showed that the association between connectivity frequency and topical distance has a power-law region (in the middle) with irregularities. We believe that *RefNet* searches were well guided by the network in most instances (when routed through peers with regular clustering-guided connections) but was lost in others (disoriented in regions where irregular connections dominated).

To demonstrate potential utility of network clustering, we rewired the network (throug self-organization) based on the connectivity probability function described in Section 4.3. Experimental results with clustering exponent $\alpha = 3.0$ are shown as solid lines in Figure 4, in which proper network clustering better guided RefNet search and further improved the results – a higher than 95% completion rate was already achieved at max search path length 20 (Figure 4 (a)) or average path length 5 (Figure 4 (b)).

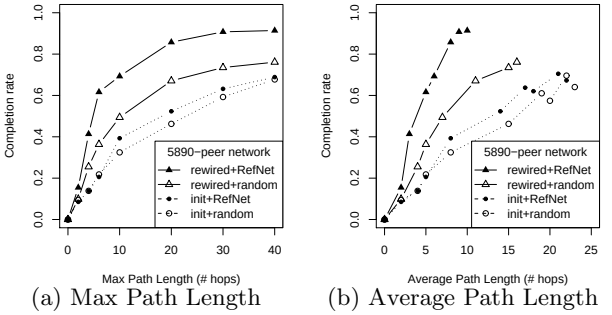


(a) 181 peers (b) 5890 peers

Figure 5: Initial Network Clustering: Connectivity (Y) vs. Topical Distance (X). Compare to Figure 1.

6.2 5890-Peer Network

On the initial 5890-peer network, experimental results indicated that the *RefNet* model had limited advantage over *random walk*, as shown by dotted lines in Figures 6 (a) and (b). Further analysis revealed that the network was insufficiently clustered. As shown in Figure 5 (b) on log/log coordinates, the correlation between connectivity and topical distance departed quite a bit from a power-law function (linear on log/log) with which efficient searches can be well-guided [15, 16, 24]. The curve suggests that there were too many topically remote connections that disoriented searches as peers were more likely to connect to topically irrelevant neighbors.



(a) Max Path Length (b) Average Path Length

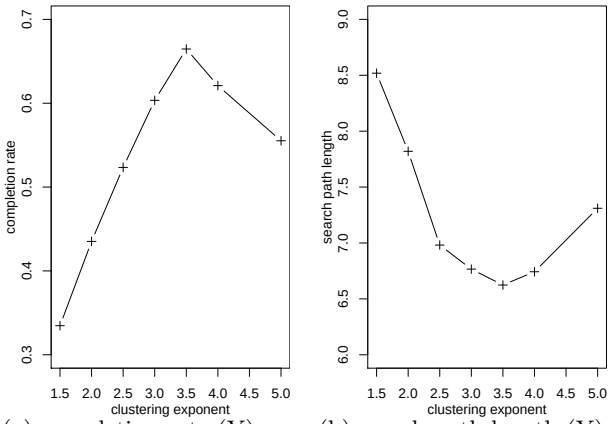
Figure 6: Completion Rate (Y) vs. Path Length (X) on 5890 Peers

Again, we used the method discussed in Section 4.3 to fine tune the 5890-peer network for a proper level of clustering. As shown by solid lines in Figure 6, given *clustering exponent* $\alpha = 4.0$, the *RefNet model* performed much better and achieved above 90% completion rate within a max path length of 40 (Figure 6 (a)) and with an average path length of about 10 (Figure 6 (b)).

6.3 Impact of Clustering

In the results above, we have demonstrated that some level of network clustering improved decentralized search of relevant peers or information sources. It is unclear yet how much clustering is enough or how much is too much. Setting max search path length at 10, experiments based on various clustering exponent α values on the 5890-peer network produced results shown in Figures 7 (a) and (b).

Given a constant max search path length at 10, Figure 7 (a) shows completion rate vs. clustering exponent α results,



(a) completion rate (Y) (b) search path length (Y)

Figure 7: Impact of Clustering Exponent α (X)

in which best completion rate was achieved at $\alpha \approx 3.5$, which also enabled optimal search path length in Figure 7 (b). Both smaller and larger α values resulted in less optimal searches. As discussed, smaller α values produced less visible topical segments and more remote connections that disoriented searches. Larger α values, on the other hand, led to an over-clustered and fragmented network without sufficient *weak ties* for searches to move fast.

This result, obtained in a decentralized information retrieval context, is consistent with findings from previous research on complex networks with simpler representations of the search (topical) space [15, 16, 24]. The *Clustering Paradox* suggests that when we use clustering-based techniques (e.g., topical segmentation and semantic overlay in P2P networks), some balance between *strong ties* and *weak ties* should be maintained.

Previous research also suggested that the optimal clustering exponent (the absolute value) is equal to the number of dimensions that describe topical distances among peers [15, 16]. We observed that the 181-peer network was optimal at $\alpha \approx 3.0$. With a larger number of peers and more diverse contents, the 5890-peer network seemed to require a little higher dimensionality to accurately depict all pairwise relationships, thus a slightly larger optimal clustering exponent $\alpha \approx 3.5$.

6.4 Scaling of Clustering Impact

One may argue that the impact of network clustering on decentralized search is small especially in the case of over-clustering – in Figure 7, for instance, there were roughly 10% loss in completion rate (effectiveness) and an increase of 1 in average search path length (efficiency) when α increased from 3.5 (optimum) to 5.0. Nonetheless, we will show in very large networks, the *Clustering Paradox* has a huge impact on search efficiency.

Relying on a 2-dimensional network model used in previous research [15, 16, 6], we ran decentralized search simulations on various network size scales $N \in [10^4, .., 4 \times 10^6]$ and with clustering exponent $\alpha \in [0, 4]$ (see [15] for detailed configurations). Results indicated that while optimum α approaches 2 with increased network size, there is a dramatic constrast between optimal clustering and overclustering in very large networks (see steeper curves in log-transformed Figure 8).

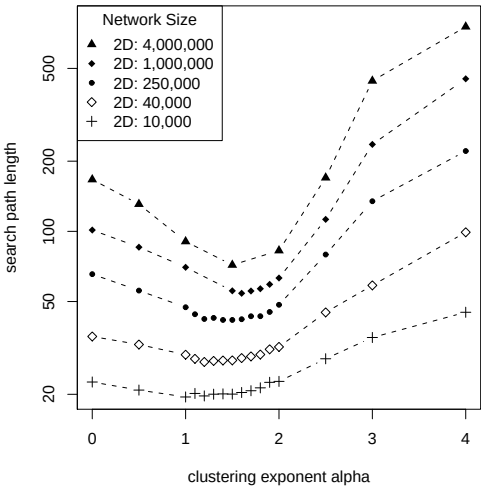


Figure 8: Scaling of Clustering Impact (100% completion rate). Note that search path length (Y) is log transformed.

On smaller scales (e.g., in the 10^4-peer network), as shown in Figure 8, optimization curves are much flatter. Overclustering in small networks only resulted in a moderate increase of search path length. However, in the network of four million peers, as shown in Figure 8, when α increased from 2 (nearly optimum) to 4, the average search path length increased from roughly 80 to more than 700 – a huge loss in search efficiency. Seen in this light, methods achieving good results on small or medium network sizes will not necessarily function well on large scales. Little performance disadvantage in small networks might become too big to ignore in large networks. Scrutiny of the *Clustering Paradox* for network optimization is crucial for scalability of decentralized search.

7. CONCLUSION

In this paper, we presented a multi-agent framework for information retrieval in distributed networked environments and focused on the impact of network clustering on decentralized search. Particularly, we studied search optimization in the face of the *Clustering Paradox*, in which either too little or too much clustering leads to degraded findability of relevant information sources. Experiments showed that the similarity based *RefNet* model outperformed random walks on the initial 181-peer network and did not show much advantage on the initial 5890-peer network, which was shown to have too many topically remote connections or *weak ties* that disoriented searches.

By introducing a *clustering exponent* α to guide network rewiring, we studied the impact of clustering and found that a balanced level of network clustering produced optimal results. Particularly, in the network of 5890 scholars, relevant peers were best findable at $\alpha \approx 3.5$. Smaller α values resulted in less visible topical segments and many remote connections that disoriented searches. Larger α values, on the other hand, led to an over-clustered and fragmented network with rich *strong ties* but scant *weak ties* for searches to move fast.

Further experiments on various larger networks of up to 4 million peers demonstrated that clustering optimization is crucial for decentralized search. Although overclustering only moderately degraded search performance on small networks, it led to dramatic loss in search efficiency for large networks. So did weak clustering. Search methods that work well on small scales might function badly in large networks, in which little performance disadvantage in small networks might become too big to ignore. As many research rely on clustering for decentralized search (e.g., in semantic overlay networks for P2P), scrutiny of the *Clustering Paradox* is crucial for scalability of existing methods.

Acknowledgments

We appreciate valuable discussions with Gary Marchionini, Munindar P. Singh, Diane Kelly, Jeffrey Pomerantz, and Simon Spero, and constructive comments from LSDS-IR'09 reviewers. We thank the NC Translational and Clinical Sciences (TraCS) Institute for support.

8. REFERENCES

[1] L. A. N. Amaral, A. Scala, M. Barthélémy, and H. E. Stanley. Classes of small-world networks. *Proceedings of the National Academy of Sciences of the United States of America*, 97(21):11149–11152, 2000.

[2] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley Longman Publishing, 2004.

[3] M. Bawa, G. S. Manku, and P. Raghavan. Sets: search enhanced by topic segmentation. In *SIGIR '03: Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pages 306–313, New York, NY, USA, 2003. ACM.

[4] F. L. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.

[5] M. Bender, S. Michel, P. Triantafillou, G. Weikum, and C. Zimmer. Improving collection selection with overlap awareness in p2p search engines. In *SIGIR '05: Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 67–74, New York, NY, USA, 2005. ACM.

[6] M. Boguñá, D. Krioukov, and K. C. Claffy. Navigability of complex networks. *Nature Physics*, 5(1):74 –80, 2009.

[7] A. Crespo and H. Garcia-Molina. Semantic overlay networks for p2p systems. In *Agents and Peer-to-Peer Computing*, pages 1–13, 2005.

[8] C. Doulkeridis, K. Norvag, and M. Vazirgiannis. Peer-to-peer similarity search over widely distributed document collections. In *LSDS-IR '08: Proceeding of the 2008 ACM workshop on Large-Scale distributed systems for information retrieval*, pages 35–42, New York, NY, USA, 2008. ACM.

[9] G. Fischer and A. Nurzenski. Towards scatter/gather browsing in a hierarchical peer-to-peer network. In *P2PIR '05: Proceedings of the 2005 ACM workshop on Information retrieval in peer-to-peer networks*, pages 25–32, New York, NY, USA, 2005. ACM.

[10] M. S. Granovetter. The strength of weak ties. *American Journal of Sociology*, 78(6):1360–1380, May 1973.

[11] J. Han, M. Kamber, and A. L. H. Tung. *Spatial Clustering methods in data mining: a survey*. CRC, New York, 2001.

[12] M. A. Hearst and J. O. Pedersen. Reexamining the cluster hypothesis: Scatter/Gather on retrieval results. In *SIGIR '96: Proceedings of the 19th annual international ACM SIGIR conference on research and development in information retrieval*, pages 76–84, New York, NY, USA, 1996. ACM Press.

[13] W. Ke, J. Mostafa, and Y. Fu. Collaborative classifier agents: studying the impact of learning in distributed document classification. In *JCDL '07: Proceedings of the 7th ACM/IEEE-CS joint conference on Digital libraries*, pages 428–437, New York, NY, USA, 2007. ACM.

[14] W. Ke, C. R. Sugimoto, and J. Mostafa. Dynamicity vs. effectiveness: Studying online clustering for scatter/gather. In *SIGIR '09: Proceedings of the 32th annual international ACM SIGIR conference on research and development in information retrieval*, Boston, MA, 2009. ACM Press.

[15] J. M. Kleinberg. Navigation in a small world. *Nature*, 406(6798), August 2000.

[16] D. Liben-Nowell, J. Novak, R. Kumar, P. Raghavan, and A. Tomkins. Geographic routing in social networks. *Proceedings of the National Academy of Sciences of the United States of America*, 102(33):11623–11628, 2005.

[17] J. Lu and J. Callan. User modeling for full-text federated search in peer-to-peer networks. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 332–339, New York, NY, USA, 2006. ACM.

[18] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and Tutorials*, 7:72–93, 2005.

[19] T. Luu, F. Klemm, I. Podnar, M. Rajman, and K. Aberer. Alvis peers: a scalable full-text peer-to-peer retrieval engine. In *P2PIR '06: Proceedings of the international workshop on Information retrieval in peer-to-peer networks*, pages 41–48, New York, NY, USA, 2006. ACM.

[20] P. Raftopoulou and E. G. Petrakis. A measure for cluster cohesion in semantic overlay networks. In *LSDS-IR '08: Proceeding of the 2008 ACM workshop on Large-Scale distributed systems for information retrieval*, pages 59–66, New York, NY, USA, 2008. ACM.

[21] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, New York, NY, USA, 2001. ACM.

[22] R. D. Reed and R. J. Marks. *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. MIT Press, Cambridge, MA, USA, 1998.

[23] F. Sebastiani. Machine learning in automated text categorization. *ACM Computing Surveys*, 34(1):1–47, 2002.

[24] O. Simsek and D. Jensen. Navigating networks by using homophily and degree. *Proceedings of the National Academy of Sciences*, 105(35):12758–12762, 2008.

[25] M. P. Singh, B. Yu, and M. Venkatraman. Community-based service location. *Communications of the ACM*, 44(4):49–54, 2001.

[26] G. Skobeltsyn, T. Luu, I. P. Zarko, M. Rajman, and K. Aberer. Web text retrieval with a p2p query-driven index. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 679–686, New York, NY, USA, 2007. ACM.

[27] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM.

[28] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 175–186, New York, NY, USA, 2003. ACM.

[29] C. J. van Rijsbergen and K. Sparck-Jones. A test for the separation of relevant and non-relevant documents in experimental retrieval collections. *Journal of Documentation*, 29(3):251–257, 1973.

[30] D. J. Watts, P. S. Dodds, and M. E. J. Newman. Identity and Search in Social Networks. *Science*, 296(5571):1302–1305, 2002.

[31] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, San Francisco, 2nd edition, 2005.

[32] B. Yu and M. P. Singh. Searching social networks. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 65–72, New York, NY, USA, 2003. ACM.

[33] I. P. Zarko and F. Silvestri. The CIKM 2006 workshop on information retrieval in peer-to-peer networks. *SIGIR Forum*, 41(1):101–103, 2007.

[34] D. Zeinalipour-Yazti, V. Kalogeraki, and D. Gunopulos. Information retrieval techniques for peer-to-peer networks. *Computing in Science and Engineering*, 6(4):20–26, 2004.

[35] H. Zhang and V. Lesser. A reinforcement learning based distributed search algorithm for hierarchical peer-to-peer information retrieval systems. In *AAMAS '07: Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 1–8, New York, NY, USA, 2007. ACM.

[36] J. Zhang and M. S. Ackerman. Searching for expertise in social networks: a simulation of potential strategies. In *GROUP '05: Proceedings of the 2005 international ACM SIGGROUP conference on Supporting group work*, pages 71–80, NY, USA, 2005. ACM.

The Curse of Zipf and Limits to Parallelization: A Look at the Stragglers Problem in MapReduce

Jimmy Lin
The iSchool, College of Information Studies, University of Maryland
National Center for Biotechnology Information, U.S. National Library of Medicine
jimmylin@umd.edu

ABSTRACT

This paper explores the problem of "stragglers" in MapReduce: a common phenomenon where a small number of mappers or reducers takes significantly longer than the others to complete. The effects of these stragglers include unnecessarily long wall-clock running times and sub-optimal cluster utilization. In many cases, this problem cannot simply be attributed to hardware idiosyncrasies, but is rather caused by the Zipfian distribution of input or intermediate data. I present a simple theoretical model that shows how such distributions impose a fundamental limit on the amount of parallelism that can be extracted from a large class of algorithms where all occurrences of the same element must be processed together. A case study in parallel *ad hoc* query evaluation highlights the severity of the stragglers problem. Fortunately, a simple modification of the input data cuts end-to-end running time in half. This example illustrates some of the issues associated with designing efficient MapReduce algorithms for real-world datasets.

Categories and Subject Descriptors: H.3.3 [Information Storage and Retrieval]: Information Search and Retrieval

General Terms: Algorithms, Performance

1. INTRODUCTION

The only practical solution to large data problems today is to distribute computations across multiple machines in a cluster. With traditional parallel programming models (e.g., MPI, pthreads), the developer shoulders the burden of explicitly managing concurrency. As a result, a significant amount of the developer's attention must be devoted to managing system-level details (e.g., synchronization primitives, inter-process communication, data transfer, etc.). MapReduce [2] presents an attractive alternative: its functional abstraction provides an easy-to-understand model for designing scalable, distributed algorithms.

Taking inspiration from higher-order functions in functional programming, MapReduce provides an abstraction for programmer-defined "mappers" and "reducers". Key-value pairs form the processing primitives in MapReduce. The mapper is applied to every input key-value pair to generate an arbitrary number of intermediate key-value pairs. The reducer is applied to all values associated with the same intermediate key to generate output key-value pairs. This two-stage processing structure is illustrated in Figure 1.

Under the MapReduce framework, a programmer needs only to provide implementations of the mapper and reducer. On top of a distributed file system [3], the runtime transparently handles all other aspects of execution on clusters ranging from a few to a few thousand processors. The runtime is responsible for scheduling, coordination, handling faults, and the potentially very large sorting problem between the map and reduce phases whereby intermediate key-value pairs must be grouped by key. The availability of Hadoop, an open-source implementation of the MapReduce programming model, coupled with the dropping cost of commodity hardware and the growing popularity of alternatives such as utility computing, has brought data-intensive distributed computing within the reach of many academic researchers [5].

This paper explores a performance issue frequently encountered with MapReduce algorithms on natural language text and other large datasets: the "stragglers problem", where the distribution of running times for mappers and reducers is highly skewed. Often, a small number of mappers takes significantly longer to complete than the rest, thus blocking the progress of the entire job. Since in MapReduce the reducers cannot start until all the mappers have finished, a few stragglers can have a large impact on overall end-to-end running time. The same observation similarly applies to the reduce stage, where a small number of long-running reducers can significantly delay the completion of a MapReduce job. In addition to long running times, the stragglers phenomenon has implications for cluster utilization—while a submitted job waits for the last mapper or reducer to complete, most of the cluster is idle. Of course, interleaving the execution of multiple MapReduce jobs alleviates this problem, but presently, the scheduler in the open-source Hadoop implementation remains relatively rudimentary.

There are two main reasons for the stragglers problem. The first is idiosyncrasies of machines in a large cluster—this problem is nicely handled by "speculative execution" [2], which is implemented in Hadoop. To handle machine-level variations, multiple instances of the same mapper or reducer are redundantly executed in parallel (depending on the availability of cluster resources), and results from the

LSDS-IR Workshop. July 2009. Boston, USA.

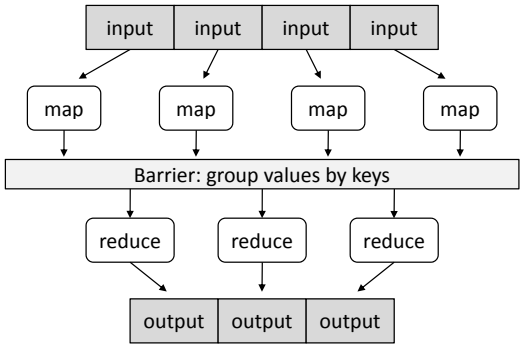


Figure 1: Illustration of MapReduce: "mappers" are applied to input records, which generate intermediate results that are aggregated by "reducers".

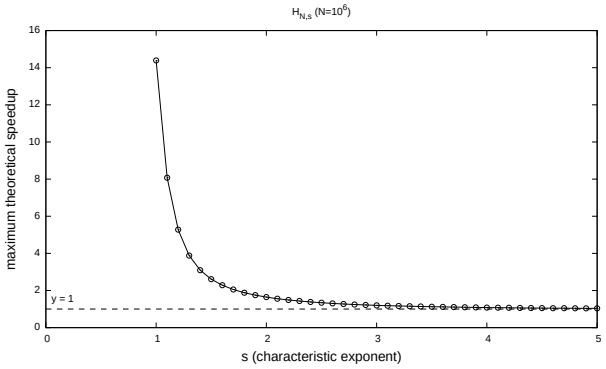


Figure 2: Plot of $H_{N,s}$, the Nth generalized harmonic number with $N = 10^6$, based on different values of s, the characteristic exponent of the Zipfian distribution. This plot shows the maximum theoretical speedup of a parallel algorithm where all instances of an element type must be processed together.

first instance to finish are used (the remaining results are discarded). This, however, does not address skewed running times caused by the distribution of input or intermediate data. To illustrate this, consider the simple example of word count using MapReduce (for example, as the first step to computing collection frequency in a language-modeling framework for information retrieval): the mappers emit key-values pairs with the term as the key and the local count as the value. Reducers sum up the local counts to arrive at the global counts. Due to the distribution of terms in natural language, some reducers will have more work than others (in the limit, imagine counting stopwords)—this yields potentially large differences in running times, independent of hardware idiosyncrasies.

This paper explores the stragglers problem in MapReduce, starting first with a theoretical analysis in Section 2. A specific case study in parallel *ad hoc* query evaluation is discussed in Section 3; for that specific task, a simple manipulation of the input data yields a significant decrease in wall-clock running time. I conclude with some thoughts on the design of MapReduce algorithms in light of these findings.

2. A THEORETICAL ANALYSIS

It is a well-known observation that word occurrences in natural language, to a first approximation, follow a Zipfian distribution. Many other naturally-occurring phenomena, ranging from website popularity to sizes of craters on the moon, can be similarly characterized [7]. Loosely formulated, Zipfian distributions are those where a few elements are exceedingly common, but contain a "long tail" of rare events. More precisely, for a population of N elements, ordered by frequency of occurrence, the frequency of the element with rank k can be characterized by:

$$p(k; s, N) = \frac{k^{-s}}{\sum_{n=1}^{N} 1/n^s} \quad (1)$$

where s is the characteristic exponent. The denominator is known as the Nth generalized harmonic number, often written as $H_{N,s}$. For many types of algorithms, all occurrences of the same type of element must be processed together. That is, the element type represents the finest grain of parallelism that can be achieved. Let us assume that the amount of "work" associated with processing a type of element is $O(n)$ with respect to the number of element instances. In other words, the amount of work necessary to process an element is proportional to its frequency. If we normalize the total amount of work to one, then the fraction of the total work that must be devoted to processing the most common element is:

$$\frac{1}{\sum_{n=1}^{N} 1/n^s} = \frac{1}{H_{N,s}} \quad (2)$$

Even if we assume unlimited resources and the ability to process *all* element types in parallel, the running time of an algorithm would still be bound by the time required to process the most frequent element. In the same spirit as Amdahl's Law [1], Zipf's Law places a theoretical upper bound on the amount of parallelism that can be extracted for certain classes of algorithms. In short, an algorithm is only as fast as the slowest of the sub-tasks that can run in parallel.

As a concrete example, suppose $N = 10^6$ and $s = 1$. The most frequently-occurring element would be observed about 6.95% percent of the time—which means that the maximum theoretical speedup of any parallel algorithm is about a factor of 14 (assuming that all instances of the same element must be processed together). For this class of algorithms, the maximum theoretical speedup is simply $H_{N,s}$. Figure 2 plots $H_{N,s}$ with varying values of s for $N = 10^6$. This simple theoretical model shows that Zipfian distributions present a serious impediment to the development of efficient parallel algorithms for a large class of real-world problems.

This analysis can be directly applied to MapReduce algorithms. In a common scenario, each key corresponds to a single type of element, and the fraction of total values associated with each key can be characterized by a Zipfian distribution. In other words, a few keys have a large number of associated values, while a very large number of keys have only a few values. Take the two following examples:

Inverted indexing. The well-known MapReduce algorithm for inverted indexing begins by mapping over input documents and emitting individual postings as intermediate key-value pairs. All postings associated with each term are

gathered in the reducer, where the final postings lists are created and written to disk. The most straightforward implementation of the algorithm divides the term space into roughly equal-sized partitions and assigns each to a reducer (typically, through hashing). This, however, does not take into account the inherent distribution of document frequencies (which characterizes the length of each postings list, and hence the amount of work that needs to be done for each term). Due to the Zipfian distribution of term occurrences, the reducers assigned to very frequent terms will have significantly more work than the other reducers, and therefore take much longer to complete. These are the stragglers observed in the execution of the algorithm.

Computing PageRank over large graphs. Although Google recently revealed that it has developed infrastructure specifically designed for large-scale graph algorithms (called Pregel), the implementation of PageRank [8] in MapReduce is nevertheless instructive. The standard iterative MapReduce algorithm for computing PageRank maps over adjacency lists associated with each vertex (containing information about outlinks); PageRank contributions are passed onto the target of those outlinks, keyed by the target vertex id. In the reduce stage, PageRank contributions for each vertex are totaled.[1] In the simplest implementation, vertices are distributed across the reducers such that each is responsible for roughly the same number of vertices (by hashing). However, the highly skewed distribution of incoming links to a page presents a problem: the vast majority of pages have few inbound links, while a few (e.g., the Google homepage) have many orders of magnitude more. In computing PageRank, the amount of work required to process a vertex is roughly proportional to the number of incoming links. The stragglers are exactly those assigned to process vertices in the graph with large in-degrees.

In practice, faster speed-ups than predicted are possible because in most cases some amount of local aggregation can be performed, thus making the skewed key-value distributions less pronounced. In MapReduce, this is accomplished with "combiners", which can dramatically increase efficiency. Nevertheless, the stragglers problem remains both common and severe.

In both of the examples presented above, the stragglers problem is most severe in the reduce stage of processing, since the distribution of the intermediate data can be characterized as Zipfian. However, depending on the distribution of input key-value pairs, it is also possible to have the stragglers problem in the map phase—in fact, the case study presented in the next section examines such a case.

3. PARALLEL QUERIES: A CASE STUDY

This paper presents a case study of the stragglers problem that builds on the parallel queries algorithm described in SIGIR 2009 [6]. This piece is meant to serve as a companion to the paper in the main conference proceedings. The problem explored here nicely illustrates how the running time of a MapReduce algorithm is dominated by the slowest parallel sub-task. Fortunately, for this problem, a simple manipulation of the input data cuts running time in half.

[1]This algorithm sketch ignores details such as handling of dangling links and the jump factor.

```
1: procedure Map(Term t, Postings P)
2:     [Q_1, Q_2, ... Q_n] ← LoadQueries()
3:     for all Q_i ∈ [Q_1, Q_2, ... Q_n] do
4:         if t ∈ Q_i then
5:             Initialize.AssociativeArray(H)
6:             for all ⟨a, f⟩ ∈ P do
7:                 H{a} ← w_{t,q} · w_{t,d}
8:             Emit(i, H)

1: procedure Reduce(Qid i, [H_1, H_2, H_3, ...])
2:     Initialize.AssociativeArray(H_f)
3:     for all H ∈ [H_1, H_2, H_3, ...] do
4:         Merge(H_f, H)
5:     Emit(i, H_f)
```

Figure 3: Pseudo-code of the parallel queries algorithm in MapReduce.

3.1 Background

Computing pairwise similarity on document collections is a task common to a variety of problems such as clustering, unsupervised learning, and text retrieval. One algorithm presented in [6] treats this task as a very large *ad hoc* retrieval problem (where query-document scores are computed via inner products of weighted feature vectors). This proceeds as follows: First, the entire collection is divided up into individual blocks of documents; these are treated as blocks of "queries". For each document block, the parallel retrieval algorithm in Figure 3 is applied. The input to each mapper is a term t (the key) and its postings list P (the value). The mapper loads all the queries at once and processes each query in turn. If the query does not contain t, no action is performed. If the query contains t, then the corresponding postings must be traversed to compute the partial contributions to the query-document score. For each posting element, the partial contribution to the score ($w_{t,q} \cdot w_{t,d}$) is computed and stored in an associative array H, indexed by the document id a—this structure holds the accumulators. The mapper emits an intermediate key-value pair with the query number i as the key and H as the value. The result of each mapper is all partial query-document scores associated with term t for all queries that contain the term.

In the reduce phase, all associative arrays belonging to the same query are brought together. The reducer performs an element-wise sum of all the associative arrays (denoted by Merge in the pseudo-code): this adds up the contributions for each query term across all documents. The final result is an associative array holding complete query-document scores. In effect, this algorithm replaces random access to the postings with a parallel scan of all postings. In processing a set of queries, each postings list is accessed only once—each mapper computes partial score contributions for *all* queries that contain the term. Pairwise similarity for the entire collection can be computed by running this algorithm over all blocks in the collection. For more details, please refer to additional discussions of this algorithm in the SIGIR 2009 main proceedings paper.

3.2 Experimental Results

Experiments using the algorithm presented in Figure 3 were conducted on a collection of 4.59m MEDLINE abstracts (from journals in the life and health sciences domain),

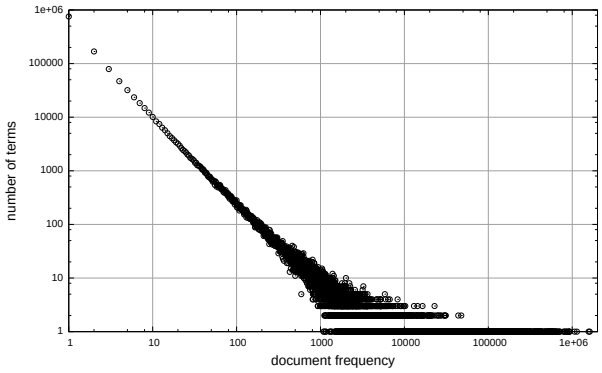


Figure 4: Distribution of document frequencies of terms in the MEDLINE collection.

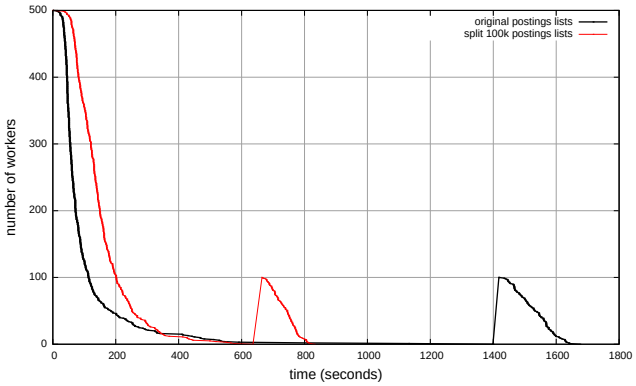


Figure 5: Progress of the MapReduce parallel queries algorithm (472 full-abstract "queries" on the MEDLINE collection), comparing original postings lists (thick black line) and postings lists split into 100k segments (thin red line).

used in the TREC 2005 genomics track [4]. From the list of relevant documents for the evaluation, 472 (approximately one tenth) were selected to serve as the "queries". The entire text of each abstract was taken verbatim as the query. On average, each query contained approximately 120 terms after stopword removal. All runs were performed on a large cluster running the Hadoop implementation of MapReduce (version 0.17.3) with Java 1.5; jobs were configured with 500 mappers and 200 reducers. The parallel queries algorithm was implemented in pure Java. Although the cluster contains a large number of machines, each individual machine is relatively slow, based on informal benchmarks.[2]

An inverted index was built from the entire collection.[3] Figure 4 shows the document frequencies of all 1.3m unique terms in the collection: on the x-axis, the document frequency of a term, and on the y-axis, the number of terms that have that *df*. Approximately 753k terms appear only once, and approximately 168k terms appear only twice. The term with the largest *df* appeared in 1.62m documents (or about one in three documents). This distribution is fairly typical of information retrieval text collections.

The thick line in Figure 5 plots the progress of the MapReduce job on the unmodified index of the document collection: wall-clock time (in seconds) on the x-axis and number of active workers (either mappers or reducers) on the y-axis. At the beginning of the job, 500 mappers are started by the MapReduce runtime; the number of running mappers decreases as the job progresses. When all the mappers have finished, 200 reducers start up after a short lull (during this time the framework is copying intermediate key-value pairs from mappers to reducers). Finally, the job ends when all reducers complete and the results have been written to disk.

The map phase of execution completes in 1399s; the reduce phase begins at the 1418s mark, and the entire job finishes in 1675s. The long tail of the plot in the map phase graphically illustrates the stragglers problem. Consider the following statistics, which can be interpreted as checkpoints corresponding to 98.0%, 99.0%, and 99.8% mapper progress:

- At the 467s mark, all except for 10 mappers (2% of the total) have completed.
- At the 532s mark, all except for 5 mappers (1% of the total) have completed.
- At the 1131s mark, all except for one mapper have finished—the last mapper would run for about another 4.5 minutes before finishing.

Since all mappers must complete before the reducers can begin processing intermediate key-value pairs, the running time of the map phase is dominated by the slowest mapper, which as shown here takes significantly longer than the rest of the mappers. Repeated trials of the same experiment produced essentially the same behavior (not shown).

The behavior of the parallel queries algorithm is explained by the empirical distribution of the length of the postings lists (see Figure 4). There are a few very long postings lists (corresponding to common terms such as "gene" or "patient"), while the vast majority of the postings lists are very short. Since for each query term the postings must be traversed to compute partial document score contributions, the amount of work involved in processing a query term is on the order of the length of the postings list. Therefore, the mappers assigned to process common query terms will run for a disproportionately long time—these are exactly the stragglers observed in Figure 5. These empirical results are consistent with the theoretical model presented in Section 2. However, the situation is not quite as grim in some cases—in a retrieval application, user queries are less likely to contain common terms since they are typically less helpful in specifying an information need.

A natural solution to the stragglers problem, in this case, is to break long postings lists into shorter ones. Exactly such a solution was implemented: postings lists were split into 100k segments, so that terms contained in more than 100k documents were associated with multiple postings lists. Furthermore, the ordering of all the postings segments were randomized (as opposed to alphabetically sorted by term, as in the original inverted index).[4] Note that this modification to the inverted index did not require any changes to

[2]The hardware configuration is the same as the setup in [6].

[3]The tokenizer from the open-source Lucene search engine was adapted for document processing. A list of stopwords from the Terrier search engine was used.

[4]This algorithm was straightforwardly implemented in MapReduce, by mapping over the original inverted index and writing a new copy.

Postings	98.0%	99.0%	99.8%	100.0%
original	467s	532s	1131s	1399s
segmented	425s	498s	576s	636s

Table 1: Progress of mappers in the parallel queries algorithm, comparing original postings lists and postings lists split into 100k segments.

the parallel queries algorithm. Although partial score contributions for a single query term might now be computed in different mappers, all partial scores will still be collected together in the same reducer during the element-wise sum across associative arrays keyed by the same query id.

The thin (red) line in Figure 5 plots the progress of the MapReduce job on the modified index. The map phase of execution completes in 636s; the reduce phase begins at the 665s mark, and the entire job finishes in 831s. With this simple modification to the inverted index, the same algorithm runs in about half the wall-clock time. Table 1 compares the 98.0%, 99.0%, 99.8%, and 100.0% progress of the mappers for both the original and segmented postings list. Multiple trials of the same experiment gave rise to similar results.

As a follow up, additional experiments were conducted with even smaller postings segments, but finer splits did further decrease running time. This is likely due to the distribution of query terms—some postings lists will never be traversed no matter how finely segmented they are, since the query documents contain only a subset of all terms in the collection (and by implication, some mappers will do little work regardless).

Why does this simple manipulation of the input data work so well? It is important to note that the technique does not actually reduce the number of computations required to produce query-document scores. The total amount of work (i.e., area under the curve) is the same—however, with the segmented index, work is more evenly distributed across the mappers. In essence, splitting long postings lists is a simple, yet effective, approach to "defeating" Zipfian distributions for this particular application.

4. DISCUSSION

The limits on parallelization imposed by Zipfian distributions is based on one important assumption—that all instances of the same type of element must be processed together. The simple approach to overcoming the stragglers problem is to devise algorithms that do not depend on this assumption. In the parallel queries case, this required only manipulating input data and no modifications to the algorithm: query-document scores could be computed independently, since the associative arrays in which they are held would eventually be brought together and merged in the reduce stage.

Although the parallel queries case study dealt with stragglers in the map phase due to distributional characteristics of the input data, the same principles can be applied to stragglers in the reduce phase as well. However, there are additional complexities that need to be considered. Often, it is known in advance that intermediate data can be characterized as Zipfian—but devising algorithms to address the issue may require actually knowing which elements occupy the head of the distribution. This in turn requires executing the algorithm itself, which may be slow precisely because of the stragglers problem. This chicken-and-egg dependency can be broken by sampling strategies, but at the cost of greater complexity in algorithm design.

For solving reduce-phase stragglers, once the distribution of intermediate data has been characterized, a more intelligent partitioner can better distribute load across the reducers. Unfortunately, this may still be insufficient in some cases, for example, PageRank computations. Recall that in computing PageRank all contributions from inbound links must be summed, thereby creating the requirement that all values associated with the same key (i.e., vertex in graph) be processed together. The only recourse here is a modification of the original algorithm (e.g., a multi-stage process for totaling the PageRank contributions of pages with large in-degrees).

The upshot of this discussion is that to overcome the efficiency bottleneck imposed by Zipfian distributions, developers must apply *application-specific* knowledge to parallelize the processing of common elements. This, in turn, depends on the ingenuity of the individual developer and requires insight into the problem being tackled.

5. CONCLUSION

This paper explores the stragglers problem in MapReduce caused by Zipfian distributions common in many real-world datasets. What are the broader implications of these findings? I believe that two lessons are apparent:

- First, efficient MapReduce algorithms are not quite as easy to design as one might think. The allure of MapReduce is that it presents a simple programming model for designing scalable algorithms. While this remains an accurate statement, the deeper truth is that there is still quite a bit of "art" in designing efficient MapReduce algorithms for non-toy problems—witness the case study presented in this paper, where a simple tweak to the input data cut running time in half.[5]

- Second, MapReduce algorithms cannot be studied in isolation, divorced from real-world applications—the stragglers problem is caused by properties of datasets (critically, not from the processing architecture or inherent bottlenecks in the algorithm). Furthermore, since there does not appear to be a general-purpose, universally-applicable solution to the problem, it is of limited value to discuss algorithmic performance without reference to specific applications.

Hopefully, these two lessons will be useful to future designers of MapReduce algorithms.

6. ACKNOWLEDGMENTS

This work was supported by the following sources: the Intramural Research Program of the NIH, National Library of Medicine; NSF under awards IIS-0705832 and IIS-0836560 (under the CLuE program); Google and IBM, via the Academic Cloud Computing Initiative. I am grateful to Esther and Kiri for their love.

[5] Although one might argue how obvious this solution is, I think it is safe to say that the solution requires a degree of understanding of both information retrieval algorithms and MapReduce that a novice would be unlikely to have.

7. REFERENCES

[1] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 483–485, 1967.

[2] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 2004)*, pages 137–150, San Francisco, California, 2004.

[3] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 29–43, Bolton Landing, New York, 2003.

[4] W. R. Hersh, A. Cohen, J. Yang, R. Bhupatiraju, P. Roberts, and M. Hearst. TREC 2005 Genomics Track overview. In *Proceedings of the Fourteenth Text REtrieval Conference (TREC 2005)*, Gaithersburg, Maryland, 2005.

[5] J. Lin. Scalable language processing algorithms for the masses: A case study in computing word co-occurrence matrices with mapreduce. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing (EMNLP 2008)*, pages 419–428, Honolulu, Hawaii, 2008.

[6] J. Lin. Brute force and indexed approaches to pairwise document similarity comparisons with mapreduce. In *Proceedings of the 32nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2009)*, Boston, Massachusetts, 2009.

[7] M. Newman. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics*, 46(5):323–351, 2005.

[8] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the Web. Stanford Digital Library Working Paper SIDL-WP-1999-0120, Stanford University, 1999.

Are Web User Comments Useful for Search?

Wai Gen Yee, Andrew Yates, Shizhu Liu, and Ophir Frieder
Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616 USA
+1-312-567-5205
waigen@ir.iit.edu, ayates@iit.edu, sliu28@iit.edu, ophir@ir.iit.edu

ABSTRACT
We consider the potential impact of comments on search accuracy in social Web sites. We characterize YouTube comments, showing that they have the potential to distinguish videos. Furthermore, we show how they could be incorporated into the index, yielding up to a 15% increase in search accuracy.

Categories and Subject Descriptors
H.3.3 [**Information Systems**]: Information Search and Retrieval – *search process*.

General Terms
Measurement, Performance, Experimentation.

Keywords
search, comments, YouTube.

1. INTRODUCTION
The popularity of modern Web information sharing sites (e.g., online newspapers, shopping or video sharing sites), where users can post comments about the subject matter has increased the need for effective content search functionality. Work has been done on keyword (or "tag")-based search (e.g., [2][3]), but little work has been done on using user comments to improve search accuracy. We consider the impact of comments on search accuracy in the context of YouTube video search.

Search in YouTube currently involves comparing a query to a video's title, description and keywords. Comments are not factored into search ostensibly because they are unreliable indicators of content and do not exist when a video is first posted.

Note that external, non-YouTube search engines, such as Google, also do not index video comments (but do index video title, description and keywords). We confirmed this via informal experiments where we issued video comments chosen for their apparent selectivity as queries. The results of these queries did not include the corresponding videos. On the other hand, queries consisting of any combination of title, description or keywords of a video returned the corresponding video.

If content is poorly described by the title/description/keywords, however, comment information may supplement or replace traditional forms of search. The title/description/keywords of a Westminster Kennel Show video, for example, may fail to mention "dog" (not to mention particular dog breeds), and thus not turn up in the results for "dog show." Searching through comment information will almost certainly solve this problem.

In this paper, we explore the "nature" of user comments and how they may aid in search. Specifically, we analyze the term distributions of user comments and attempt to apply this information to improve search accuracy.

The hazard associated with the use of comments to improve search accuracy is that they may contain noisy terms that hurt performance as well as significantly increase the size of the index. Our experimental results, however, suggest that while some queries are negatively affected by comments, overall, they can improve query accuracy by nearly 15%. Furthermore, we apply techniques that can reduce the cost of using comments by up to 70%.

2. ANALYSIS OF THE YOUTUBE DATA
We crawled YouTube during February, 2009 and collected the text associated with the 500 most popular and 3,500 random videos. Popular videos were identified using the YouTube API. Random videos were retrieved by randomly selecting results of queries consisting of terms selected randomly from the SCOWL English word list [12]. For each video, we retrieved several information "fields," including:

- Title – A title assigned to the video by the user who posted it.
- Description – A video description by the user who posted it.
- Keywords – Video "tags" by the user who posted it.
- Comments – Comments by viewers of the video.

In total, for the 4,000 videos, we retrieved over 1 million comments made by over 600,000 users. We refer to the random 3,500 videos and the popular 500 videos together as the "small" data set.

We also similarly created a "large" data set, also consisting of a random and a popular part, crawled in May, 2009. This data set consists of 10,000 randomly crawled videos and 1,500 popular videos. The four data sets are thus:

- rand3500: This data set contains data on 3,500 videos, randomly crawled from YouTube in February, 2009. This

data was found on YouTube by issuing random queries from the SCOWL word list [12].

- pop500: This data set contains data on the 500 most popular videos according to YouTube as of February, 2009.
- rand10K: This data set contains data on 10,000 videos randomly crawled from YouTube (is the same way that rand3500 was collected) in May, 2009.
- pop1500: This data set contains data on the 1,500 most popular videos according to YouTube as of May, 2009.

In our experiments, we pre-processed the data using the Porter stemming algorithm [10]. We also tried a more conservative stemming algorithm [11] in anticipation of problems with overstemming from the unique language usage found in video comments. However, the different stemmer had little effect on the final results. We also remove stop words using the Lucene stop word list.

2.1 Basic Statistics

As shown in Table 1a, popular videos have more than 3 times the number of viewers than do random videos and more than 6 times the number of comments. Comment length for both types of videos is about 12 to 15 terms. On average, there are 2,280 terms describing a video from the rand3500 data set and 12,132 terms describing a video in the pop500 data set.. In the large data set, there is an even greater disparity between the random and popular videos, with more viewers and more comments.

The length statistics of the title, description and keyword fields, shown in Table 2, indicate that on average only 34 to 58 terms are used to describe a (random) video (assuming that comments are not used to describe videos). Including the comment field in the search returns a potential richer database of information because the average number of comment terms is at least 1,485.

Table 1. Average values for various comment statistics.

	#Views/Video	#Comments/Video	Comment Len
Popular	247,226	1,011	12
Random	71,654	152	15
Average	93,601	259	13

a. Small data set.

	#Views/Video	#Comments/Video	Comment Len
Popular	874,805	2,425	10
Random	62,807	135	11
Average	168,720	434	11

b. Large data set.

Table 2. Average lengths for non-comment video fields.

	Title	Description	Keywords
Popular	5	33	13
Random	5	44	9
Average	5	43	10

a. Small data set.

	Title	Description	Keywords
Popular	5	42	14
Random	5	24	10
Average	5	26	11

b. Large data set.

3. MEASURING INFORMATION CONTENT

As demonstrated in opinion-mining applications, many comments often describe something's "quality," rather than its "content" (e.g., how good a product is rather than what the product is) [1]. If we assume that quality-based comments come largely from a restricted vocabulary (i.e., adjectives, such as "good" or "bad"), then comments will have only a limited ability to distinguish one video from another apart from the subjective impression it left on the viewer. Specifically, comments from different videos in this case will have similar term distributions and therefore have poor discriminating power from the perspective of a search system. Furthermore, because queries generally contain content-based terms, they do not "match" the quality-based terms in the comments. In other words, comments contain little information useful to search.

To measure the discriminating power of each field, we compute each field's language model and then compute the average *KL*-divergence [13] of the individual field values to its corresponding language model. This metric is one way of identifying the potential of the field to distinguish one video from others in a search system [5].

The results shown in Table 3 confirm that the comment field is generally the least discriminating based on *KL*-divergence. For the most part, the title and the keyword fields are the most discriminating.

Table 3. *KL*-divergences for each video field.

Data Set	Title	Desc	Keywds	Comments	All
rand3500	6.77	6.14	6.82	4.98	5.19
pop500	5.46	5.14	5.35	5.68	2.59
rand10K	7.26	6.29	7.23	5.26	5.06
pop1500	5.89	5.38	5.72	5.38	2.33

4. DISTILLING INFORMATION CONTENT FROM COMMENTS

A consideration of the relative length of the average comment field explains its low *KL*-divergence. Intuitively, as a document (i.e., the comment field) gets longer, its divergence from the "background" language model decreases. (In separate experiments – not shown – we verified this phenomenon on the comment field and on the WT10G Web corpus.) In other words, the comment field becomes the language model if its size relative to the other fields is great enough.

We contend that, as a document gets longer, however, it will contain more discriminating information – as well as less discriminating information. To verify this, we identify the terms "most associated" with the comment field and see if these terms are unique to the field. We do this by pruning all but the "top terms" of each video's comment field and compare these terms to the background language model. We identify top terms with a variation of TF-IDF score (where TF measures the number of times a term appears in the video's comment field and IDF measures the number of videos' comment fields in which the term appears, as analogous to the typical definition of TF-IDF). We consider the top 68 unique terms to make the number comparable to that which is typically available in the title, description and

keyword fields, combined. (Recall our discussion on the results shown in Table 2.)

As shown in Figure 1, the *KL*-divergence of the top 68 comment terms increases quickly with the number of comment terms. The *KL*-divergence stabilizes at approximately 7.6 when the number of comment terms reaches 250 (when most of the terms are unique to the comment). This *KL*-divergence exceeds that of all the other fields (Table 3), indicating its potential in discriminating videos.

This result shows that longer comment fields contain more discriminating information. However, it is also likely that the rate of discriminating terms in comment fields decreases with comment length. Therefore, while we claim that longer comment fields contain more discriminating information, the rate at which we yield this information should decrease as the comment field gets longer. In any case, the long comment fields are more discriminating than the other fields.

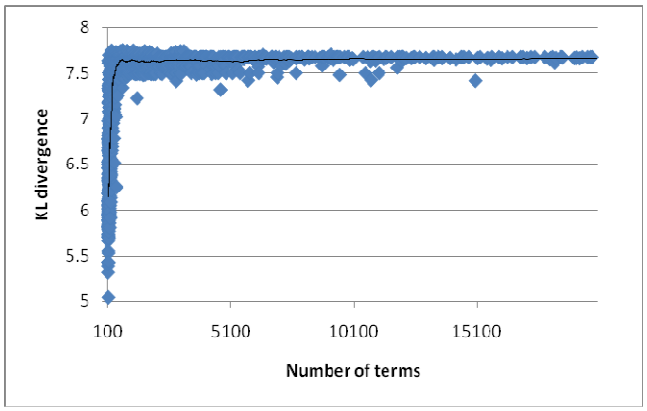


Figure 1. *KL*-divergences of the top 68 terms in each comment field as a function of number of terms in the comment field with the rand3500 data set (the trendline indicates the 50-point moving average).

Note that we only consider comment fields with at least 100 terms. With fewer terms, the comments often lacked 68 *unique* terms, making their *KL*-divergences as a function of length unstable, obscuring the results. Also, experiments with different numbers of top terms yielded similar, predictable results.

Table 4. Overlap percentage of top 30 terms and various fields with the rand3500 data set.

N	Title	Description	Keywords	Comments
10	12.58%	22.44%	31.93%	52.05%
20	10.05%	18.71%	30.24%	52.24%
30	8.14%	15.49%	27.90%	52.41%

4.1 Potential Impact of Comments on Query Accuracy

To estimate the potential of using comment terms to improve search accuracy, we use a technique described in [4] that effectively identifies the terms that are most likely to occur in a query that retrieves a given document. For each video, we extract the top *N* of these terms and calculate their overlap with the various video information fields. Note that the overlap is not necessarily disjoint, so the overlap percentages may exceed 100%.

The results in Table 4 show that most of these terms come from the comments. Of course, the comment field contains many more terms than the other fields, so the overlap will be greater. (For example, the title field's overlap is limited because titles generally contain fewer than 30 terms.) But the point is that it is exactly the size of the comment field that is the source of its potential. Although it contains many meaningless terms, it also contains a lion's share of the top terms. This suggests including comment terms in queries can improve search accuracy.

4.2 Waiting for Comments

One of the problems with using comments in search is that they take time for users to generate. In the results discussed in Section 4, we need about 250 comment terms before the *KL*-divergence stabilizes. If we assume each comment is 13 terms long, then we would need about 20 comments to yield 250 terms.

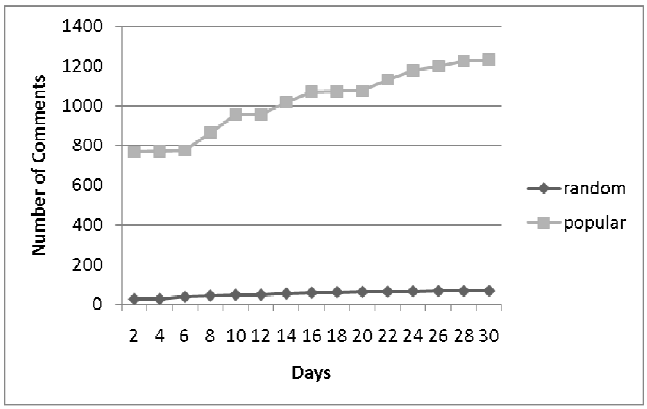


Figure 2. Number of comments as a function of time for the small data set.

Based on our data, popular and random videos receive approximately 20 and 1.4 comments per day, respectively. Therefore, popular videos collect enough comments in one day and random videos require about 2 weeks to yield enough terms to be useful for search. In Figure 2, we show the number of comments for the data set as a function of time. Popular videos are commented at a higher rate as expected, but both types of videos have a consistent increase in the number of comments.

Table 5. Analysis of DTC results on the rand3500 data set with length 3 top-IDF queries.

MRR Improvement	# of Videos	Avg(Len(DT))	Avg(Len(C))	$\|C \cap K\| / \|K\|$	$\|DT \cap K\| / \|K\|$
-1.00 - -0.75	40	19.225	719.475	0.2734	0.5314
-0.75 - -0.50	67	22.8209	345.6268	0.2904	0.4992
-0.50 - -0.25	188	27.1436	386.6968	0.2372	0.5081
-0.25 – 0.00	165	48.3273	118.8242	0.1764	0.5383
0	2576	33.5093	277.5613	0.2557	0.5617
0.00 - 0.25	152	58.4145	304.3158	0.3600	0.4049
0.25 - 0.50	151	45.4172	533.2848	0.4766	0.4402
0.50 - 0.75	38	66.2105	492.9474	0.3900	0.5529
0.75 - 1.00	116	37.2931	895.0776	0.6260	0.3291

5. EXPERIMENTAL RESULTS

5.1 Data Set and Metrics

We use data sets mentioned in Section 1 for our experiments. We simulate user queries by removing the keyword field from the video data set and using them to generate known-item queries. From the keyword set, we generate queries in two ways:

- top-IDF – Top-IDF queries are generated by the top *K* terms in the keyword field, where IDF is computed based on keyword fields.
- random – Random queries are generated by randomly picking *K* terms from the keyword field.

In the alternatives above, we use *K* values 2, 3, and 4 as these are the most common query lengths in Web and P2P applications [7][8].

We generate queries in this way because keywords are meant to help users index content. Top-IDF queries are meant to simulate users who generate very specific queries and their generation is similar to the query generation techniques described in [4][9]. Random queries are appropriate if we assume that all keywords are appropriate for queries.

Note that the choice of using the keyword field to create queries is somewhat arbitrary. Recent work shows that the terms used as keywords do not necessarily match those used in user queries [20]. For example, people tagging music would use the terms associated with genre, such as "pop," whereas people generally do not search for music via genre – title and artist are more likely in cases of known-item search. In future work, we will consider queries generated by other techniques described in [4][9].

Because we use keywords to generate queries, we also strip the keywords from the data set that we index. If we did not do this, then the baseline query accuracy would be so high – given our experimental setup – that we would not be able to reasonably run experiments that would show any meaningful positive change. One might worry that stripping keywords from the data set will result in an artificially low baseline for performance because keywords are expected to match queries very precisely. However, referring again to the results from [20], keywords do not necessarily match query behavior. Furthermore, the title and description fields were shown in Table 3 to be at least as discriminating of the video as the keywords, so these fields could have been chose as well as sources for queries.

In any case, our goal is to show whether the addition of comments can improve query performance over not using them. We could have therefore generated queries from any field provided that we remove that field from the data that is indexed. A positive result, therefore, would suggest that indexing comments in addition to all of the other fields is beneficial to query accuracy.

Because we are assuming known-item search, we use *MRR* as our main performance metric, defined as the average reciprocal rank of the desired result over all queries:

$$MRR = \frac{1}{N_Q}\sum_{i=1}^{N_Q}\frac{1}{r_i}$$

In the expression above, N_Q is the number of queries issued and r_i is the rank of the known item in the result set of query *i*. *MRR* is a metric that ranges from 0 to 1, where *MRR* = 1 indicates ideal ranking accuracy.

The data are indexed in the Terrier search engine [6]. Each of the videos is searched for via their respective queries.

Table 6. Query performance with and without comments with various query lengths on the rand3500 data set.

Query Type	Query Length	DT	DTC	Pct Change
top-IDF	2	0.5912	0.6045	2.24%
top-IDF	3	0.6645	0.6761	1.74%
top-IDF	4	0.7064	0.7136	1.01%
random	2	0.4697	0.4761	1.36%
random	3	0.5736	0.5839	1.80%
random	4	0.6377	0.6459	1.29%

5.2 Basic Results

In our first experiment, we test the impact of indexing queries. We issue the queries twice. First, we issue the queries on an index that contains the title and description (a setup that we refer to as DT) of each video, but not the comments. Second, we issue

the queries on an index that contains the title, description, and comments (a setup that we refer to as DTC) of each video.

In Table 6, we show query performance for different query lengths when the index does not and does contain comments. The results show that there is a consistent difference in *MRR* in the range of about 1% to 2% when using the comments on the rand3500 data set.

We search for the source of the performance improvement by dividing the results of each query into buckets based on the impact that the comment field has on the query results and then search for correlations between the change in *MRR* and "features" of the video data to which the query corresponds. The goal is to find some correlation between *MRR* improvement and a video feature. We considered several features of the video data, including, the lengths of the various fields in terms of the number of unique terms and the similarities between the fields.

A subset of our correlation analysis is shown in Table 5. Each bucket corresponds to a 0.25 point difference in *MRR*. We see that approximately 450 videos have their *MRR*s improved and about the same number have their *MRR*s worsened. Most videos (2,576 or about 75%) are not affected by the addition of comments.

We see that the length of the title and description fields have little impact on *MRR*. There is no clear correlation between them and change in *MRR*.

On the other hand, both the length of the comment field and the similarity between the comment and keyword fields are correlated with *MRR* change. Note that the similarity between the comment and keyword field is measured by how much the comment field covers the keyword field:

$$\frac{|C \cap K|}{|K|}$$

The coefficient of correlation between the similarity of the comment and keyword fields and the change in *MRR* is 0.7589. The coverage of the keyword field is also related to the length of the comments. If we remove the comment length of the first row of Table 5, then the coefficient of correlation between the change in *MRR* and the length of the comment field is 0.7214. (With the first row, the coefficient of correlation is 0.2964.) Finally, the coefficient of correlation between the length of the comment field and the similarity between the comment and keyword fields is 0.9351 without the first row of data and 0.7552 with the first row of data.

There is also a negative correlation between the similarity of the title and description fields with the keyword field (|DT∩K| / |K|) and *MRR* (-0.5177) and between |DT∩K| / |K| and |C∩K| / |K| (-0.8077). These results show that in the cases where titles and descriptions do not contain enough information to match the queries, then the long comment field is able to compensate. (We observe, for example, some videos with non-English descriptions and English keywords.)

The conclusion that we draw from these results is that comments help:

- *MRR* improves when the comments contain keywords (equivalently, query terms, since we generate queries from the keywords).
- Comments are particularly important when the title and description do not contain the appropriate terms that match the query.
- Longer comment fields are more likely to contain keywords.

So, despite all of the irrelevant terms contained in the comments – particularly long comments – the existence of the relevant terms helps.

In our next experiments, we run the same test on the pop500 data set. The results of this experiment show how comments affect the search for videos that users actually want to find (i.e., popular videos).

Table 7. Query performance with and without comments with various query lengths on the pop500 data set.

Query Type	Query Length	DT	DTC	Pct Change
top-idf	2	0.5193	0.6239	20.14%
top-idf	3	0.5984	0.6709	12.12%
top-idf	4	0.6561	0.7150	8.99%
random	2	0.4895	0.5455	11.44%
random	3	0.5592	0.6010	7.48%
random	4	0.6105	0.6650	8.93%

Table 8. Analysis of DTC results on the pop500 data set with length 3 top-IDF queries.

MRR Improvement	# of Videos	Avg(Len(C))	|C∩K| / |K|
-1.00 - -0.75	18	2267.7	0.5071
-0.75 - -0.50	10	2842.0	0.6795
-0.50 - -0.25	45	2481.2	0.6927
-0.25 – 0.00	16	2328.4	0.7800
0	267	3337.4	0.6546
0.00 - 0.25	41	3668.8	0.6731
0.25 - 0.50	38	4103.5	0.7710
0.50 - 0.75	10	8933.5	0.8430
0.75 - 1.00	53	5230.6	0.8204

Our results are shown in Table 7. Comments are much more effective on popular videos. For top-IDF queries, the *MRR* improvement ranges from 9% to 20%. For random queries, the *MRR* improvement ranges from 7% to 9%. Results are somewhat better for shorter queries and for top-IDF queries.

In Table 8, we again search for features that are correlated to the change in *MRR*. First, we notice that a greater percentage of videos are affected by the comments in the pop500 data set than in the rand3500 data set (about 47% versus 26%). Of the affected videos, 89 videos' *MRR*s worsened and 142 videos' *MRR*s improved with the use of comments.

We again see a correlation between the similarity between the comment and keyword fields and the change in *MRR*. The coefficient of correlation between these two variables is even greater than that of the rand3500 data set: 0.8295 versus 0.7589. The correlation between the length of the comment field and the change in *MRR* is 0.7404 with the pop500 data set versus 0.7214 with the rand3500 data set.

We summarize the performance results on the rand3500 and pop500 data sets in Table 9. We see that comments are clearly more effective on popular data. The change in *MRR* is greater and the number of videos whose *MRR* improves is greater. This is likely because of the similarity between the comment and keyword fields.

Table 9. Summary of the performance differences between experiments on rand3500 and pop500 data sets with length 3 top-IDF queries.

Metric \ Data Set	rand3500	pop500
MRR change	0.0175	0.1212
Pct of video *MRR*s improved	0.1306	0.2840
Pct of video *MRR*s worsened	0.1314	0.1780
Correl(*MRR* change, len(C))	0.7214	0.7404
Correl(*MRR* change, |C∩K| / |K|)	0.7589	0.8295

Table 10. Query performance with and without comments with various query lengths on the rand10K data set with top-IDF queries.

Query Length	DT	DTC	Pct Change
2	0.6271	0.6442	2.65%
3	0.6842	0.7052	2.98%
4	0.7199	0.7388	2.56%

Table 11. Analysis of DTC results on the rand10K data set with length 3 top-IDF queries.

MRR Improvement	# of Videos	Avg(Len(C))	|C∩K| / |K|
-1.00 - -0.75	121	838.7686	0.3614
-0.75 - -0.50	167	481.4551	0.3633
-0.50 - -0.25	421	411.7316	0.3627
-0.25 – 0.00	552	246.5326	0.2078
0	7248	291.2323	0.2947
0.00 - 0.25	538	480.7993	0.4043
0.25 - 0.50	461	553.7852	0.4892
0.50 - 0.75	121	724.9669	0.5224
0.75 - 1.00	349	874.0029	0.6521

5.2.1 Results on Larger Data Sets

To simplify our explication, in this section, we only report results using the top-IDF-type queries. Also, as done above, if no query length is specified, we use queries of length 3.

As shown in Table 10, comments also improve the query performance in the rand10K data set. *MRR* improvements are about 3%, which is similar to the improvements with the smaller, rand3500 data set (Table 6).

An analysis of the *MRR* change table for rand10K (Table 11) reveals that there is a again correlation between the length of the comments and the change in *MRR* (0.7515), and the similarity between the comment and keyword fields and the change in *MRR* (0.7064). In this case, most of the videos (72%) are unaffected by comments, however, while 13% have their *MRR*s worsened and 15% have their *MRR*s improved.

Table 12. Query performance with and without comments with various query lengths on the pop1500 data set with top-IDF queries.

Query Length	DT	DTC	Pct Change
2	0.5596	0.5991	6.59%
3	0.6228	0.6465	3.67%
4	0.6592	0.6818	3.31%

Table 13. Analysis of DTC results on the pop1500 data set with length 3 top-IDF queries.

MRR Improvement	# of Videos	Avg(Len(C))	|C∩K| / |K|
-1.00 - -0.75	86	1966.686	0.6623
-0.75 - -0.50	49	2972.674	0.7259
-0.50 - -0.25	128	2573.914	0.7766
-0.25 – 0.00	78	2611.885	0.7254
0	706	2323.965	0.7587
0.00 - 0.25	165	2789.746	0.7826
0.25 - 0.50	121	2619.744	0.8201
0.50 - 0.75	24	3690.25	0.8609
0.75 - 1.00	136	3170.044	0.8430

Again, as shown in Table 12, the improvement in MRR with popular data is greater than that with random data. With the pop1500 data set, the percentage MRR improvement ranges from 3% to 7% compared with 3% for the rand10K data set. In this case, 47% of the videos MRRs are unaffected by the comments, 23% are worsened, and 30% are improved.

The coefficient of correlation between MRR change and comment length is 0.6769 and the coefficient of correlation between MRR change and similarity of comment and keyword fields is 0.9192. Again, long comment fields are able to substitute for keywords in search.

The fact that MRR is better for popular data has been shown in other work (e.g., [9]). This is clearly due to the fact that popular data have more comments. This result is significant as it shows that increasing the number of comments does not only increase the ability for videos to naively match queries, but also increases the ability for queries to distinguish the relevant videos.

5.3 Improving our Results

Our next goal is to improve on our improvement-in-MRR results based on our observations. If we detect a correlated feature, we use the correlation in our indexing strategy.

Our main observation is that as the length of the comments field increases, so does its effectiveness in search. Therefore, we should only index comments if they are above a certain length.

We also acknowledge that there is a correlation between the change in MRR and the similarity between the comment and keyword fields. However, as there is also a correlation between comment length and similarity, we roughly cover both features by considering just the comment length.

Our first strategy is to index comments only if they are above a given length threshold. We refer to this strategy as "length-selective indexing." We show the experimental results in Table 14, where the threshold is in terms of number of terms (words).

The performance of length-selective indexing is negative. MRR consistently decreases with increasing thresholds. The problem with this strategy is that it creates a situation where certain videos are too eager to match queries. In other words, videos that have their comments indexed are ranked higher than other videos compared with the base case regardless of whether they are relevant to the query or not. Because videos are only relevant to a single query (by definition of MRR), MRR must decrease with this type of indexing.

Table 14. Percentage change in MRR with length-selective indexing on the rand3500 data set.

Len(C) Threshold	Pct Change in MRR
0	0
50	-0.19%
100	-0.40%
150	-0.64%
200	-0.97%
250	-1.09%
300	-1.25%
350	-1.38%
400	-1.60%
450	-1.76%
500	-1.95%

This was not a problem in the case where all videos' comments were indexed because the "eager matching" problem is offset by the fact that all videos (with comments) have additional terms associated with them. We expect that the additional terms contained in the comments are more likely to help match relevant queries.

5.3.1 Comment Pruning

The problem with length-selective indexing is that it un-uniformly adds "noise" to the description of videos making them match irrelevant queries. If noise were applied uniformly to all videos, then such a problem would be attenuated. The problem is that noise still causes the incorrect matching of query to results.

This observation inspires a solution whereby we index each video with its comments, but then prune away noise from the comments, leaving only the most relevant terms in the description of each video. This solution is expected to do two things:

1. Reduce the irrelevant matches of a query, and
2. Decrease the size of the index.

The technique we use to prune the comment field is that which was proposed to shrink indices in [4], known as document-centric pruning. With document-centric pruning, each term in each document is ranked based on its contribution to the *KL*-divergence [13] of the document to the background language model. The lower-ranked terms are removed from the documents before they are indexed. This technique was shown to be able to shrink the index by up to 90% with little loss in precision.

In these experiments, we prune a percentage of the comments of each video. We assume that there is a "fixed rate" at which terms that are useful to search accuracy appear in the comments. If this rate is *r*, then a comment field of length len(C) will have *r*len(C) useful terms. If we pick a pruning rate of *r*, then all of the terms left in the comment field will be useful.

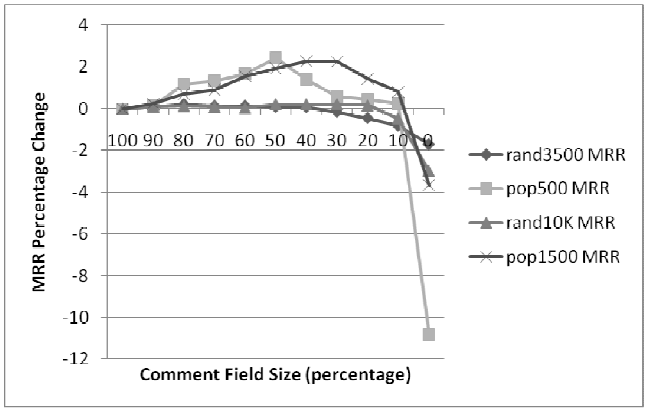


Figure 3. Percentage change in MRR for different comment field sizes for various data sets.

In Figure 3, we see the effect that comment pruning has on MRR. The data on the left of the figure corresponds to a complete comment fields, whereas the data on the right corresponds to no comments. We see that pruning initially increases the MRR for all data sets. MRR then drops dramatically as the comment field size decreases to zero.

The effect of pruning is more pronounced for the popular data sets than for the random data sets. With the random data set, the maximum MRR percentage increase is about 0.7% (60% pruning on the rand10K data set), while with the popular data set, the maximum MRR percentage increase is 2.4% (50% pruning with the pop500 data set).

The reason for this is that the random data sets' comment fields contain so few comments in the first place. They are therefore less likely to contain terms that make eagerly match irrelevant results. Second of all, the *MRR* improvement with using comments with random videos is low in the first place, suggesting the marginal impact that such comments have. We do not expect there to be much of an increase in performance with pruning.

Based on these results, a pruning rate of 50% is reasonable choice. We are able to eliminate half of the index overhead introduced by the comments and are safe from losing *MRR*

performance. *MRR* starts to decrease first with the rand3500 data set with 70% pruning.

6. RELATED WORK

In [14], the authors consider the impact that indexing blog comments have on query recall. Their conclusion is that recall is boosted by the comments, that they are useful. This result is expected, but little consideration was given to the precision of the results.

In [17], it was shown the comments do have discriminating power. The authors clustered Blog data and by using high weights for comments, were able to improve the purity and decrease the entropy of their clusters significantly.

Much of the work on "social" Web sites – where users are free to modify the metadata associated with shared data – focus on "tag" analysis, where a tag is a keyword that a user can associate with data to, say, make it easier to index. Findings related to tag analysis are they indicate data popularity and are useful in describing content [15][16][18][19]. This is somewhat orthogonal to our goal of determining if *casual* user comments can help improve search accuracy.

Table 15. Summary of potential improvement with comments.

Data Set	No Comments	Best *MRR*	Percent Change
rand3500	0.6645	0.6775	1.96%
pop500	0.5984	0.6872	14.84%
rand10K	0.6842	0.7068	3.30%
pop1500	0.6228	0.6612	6.17%

7. CONCLUSION

Our results show that comments indeed improve the quality of search compared with just using titles and descriptions to describe videos. They are particularly useful with popular videos, where the *MRR* is lower than with random videos (Table 15).

This result is not a given, however, as some queries actually do worse with comments. The reason for these cases of decreased accuracy is that the videos with fewer comments become "buried" by those with more comments in search results.

The problem of skew in result sets toward videos with larger comment fields can be addressed by well-known index pruning techniques – which also shrink the size of the index. Index pruning technique work by removing terms deemed less distinguishing or relevant to the particular "document." Applying index pruning to the comments further improves accuracy by up to about 2% (with a decrease in index size of up to 70%). Overall, accuracy improved by up to about 15% as shown in Table 15.

Our ongoing work includes further analyses and characterizations of comment terms and their impact on search accuracy. For example, our observation that comments work best when they contain query terms (Section 5.2) and when the title and description fields do not may suggest that we should only index comments when they are "different" than the title and description.

8. REFERENCES

[1] Jindal, N. and and Liu, B, "Identifying Comparative Sentences in Text Documents," In *Proc. ACM SIGIR*, 2006.

[2] Li, X., Guo, L. and Zhao Y., "Tag-based Social Interest Discovery," In *Proc. WWW*, 2008.

[3] Heymann, P., "Can Social Bookmarks Improve Web Search?" In *Proc. ACM Conf. Web Search and Data Mining* (WSDM), 2008.

[4] Buttcher, S. and Clarke, C. L. A., "A Document Centric Approach to Static Index Pruning in Text Retrieval Systems," In *Proc. ACM CIKM*, 2006.

[5] Ponte, J. M. and Croft, W. B., "A language modeling approach to information retrieval," In *Proc. ACM SIGIR*, 1998.

[6] Terrier Search Engine Web Page. http://ir.dcs.gla.ac.uk/terrier/

[7] Beitzel, S. M., Jensen, E. C., Chowdhury, A., Grossman, D. A., Frieder, O. "Hourly analysis of a very large topically categorized web query log." In Proc. *ACM SIGIR*, 2004, pp. 321-328.

[8] Yee, W. G., Nguyen, L. T., and Frieder, O., "A View of the Data on P2P File-sharing Systems." In Jrnl. Amer. Soc. of Inf. Sys. and Tech (JASIST), to appear.

[9] Azzopardi, L., de Rijke, M., Balog, K. "Building Simulated Queries for Known-Item Topics: An Analysis Using Six European Languages." In *Proc. ACM SIGIR*, 2007.

[10] Porter Stemming Web Site. http://tartarus.org/~martin/PorterStemmer/

[11] Jenkins, M.-C., Smith, D., "Conservative Stemming for Search and Indexing." In *Proc. ACM SIGIR*, 2005.

[12] Atkinson, K. SCOWL Word List. http://wordlist.sourceforge.net/

[13] Kullback, S., "Information theory and statistics." John Wiley and Sons, NY, 1959.

[14] Mishne, G., Glance, N. "Leave a Reply: An Analysis of Weblog Comments." In *Third Workshop on the Weblogging Ecosystem*, 2006.

[15] Bao, S., Xue, G., Wu, X., Yu, Y., Fei, B., Su, Z. "Optimizing Web Search Using Social Annotations." In *Proc. WWW*, 2007.

[16] Zhou, D., Bian, J., Zheng, S., Zha, H., Giles, C. L. "Exploring Social Annotations for Information Retrieval." In *Proc. WWW*, 2008.

[17] Li, B., Xu, S., Zhang, J. "Enhancing Clustering Blog Documents by Utilizing Author/Reader Comments." In *Proc. ACM Southeast Regional Conf.*, 2007.

[18] Dmitriev, P. A., Eiron, N., Fontoura, M., Shekita, E. "Using Annotations in Enterprise Search." In *Proc. WWW*, 2006.

[19] Heymann, P., Koutrika, G., Garcia-Molina, H.."Can Social Bookmarks Improve Web Search?" In *Proc. Int'l. Conf. on Web Search and Web Data Mining*, 2008.

[20] Bischoff, K., Firan, C. S., Nejdl, W., Paiu, R. "Can All Tags be Used for Search?" In *Proc. ACM CIKM*, 200

Peer-to-Peer clustering of Web-browsing users

Patrizio Dazzi
ISTI-CNR
Pisa, Italy
p.dazzi@isti.cnr.it

Pascal Felber
University of Neuchâtel
Neuchâtel, Switzerland
pascal.felber@unine.ch

Le Bao Anh
EPFL
Lausanne, Switzerland
lbanh@ifi.edu.vn

Lorenzo Leonini
University of Neuchâtel
Neuchâtel, Switzerland
lorenzo.leonini@unine.ch

Matteo Mordacchini
ISTI-CNR
Pisa, Italy
m.mordacchini@isti.cnr.it

Raffaele Perego
ISTI-CNR
Pisa, Italy
r.perego@isti.cnr.it

Martin Rajman
EPFL
Lausanne, Switzerland
Martin.Rajman@epfl.ch

Étienne Rivière
NTNU
Trondheim, Norway
etriviere@gmail.com

ABSTRACT

For most users, Web-based centralized search engines are the access point to distributed resources such as Web pages, items shared in file sharing-systems, etc. Unfortunately, existing search engines compute their results on the basis of structural information only, e.g., the Web graph structure or query-document similarity estimations. Users expectations are rarely considered to enhance the subjective relevance of returned results. However, exploiting such information can help search engines satisfy users by tailoring search results. Interestingly, user interests typically follow the clustering property: users who were interested in the same topics in the past are likely to be interested in these same topics also in the future. It follows that search results considered relevant by a user belonging to a group of homogeneous users will likely also be of interest to other users from the same group. In this paper, we propose the architecture of a novel peer-to-peer system exploiting collaboratively built search mechanisms. The paper discusses the challenges associated with a system based on the interest clustering principle. The objective is to provide a self-organized network of users, grouped according to the interests they share, that can be leveraged to enhance the quality of the experience perceived by users searching the Web.

Categories and Subject Descriptors

H.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Clustering, Information filtering*

General Terms

Algorithms, Measurement, Performance

Keywords

Peer-to-peer, Interest-based clustering, User profiling

LSDS-IR Workshop. July 2009. Boston, USA.

1. INTRODUCTION

The access to distributed resources such as Internet pages or shared files usually require the use of a search tool, e.g., a centralized search engine such as Yahoo! or Google. These search engines compute and rank their results on the basis of several different pieces of information taken from the Web graph structure, such as document PageRank [5], and from statistical estimates of query-document similarities like the TF/IDF metric. The profile of users and their tastes are rarely taken into account to enhance the users' search experience, although they provide more accurate results with respect to the interests of that particular user and such a profiling would yield better results in many situations. A first such situation occurs when there exist results along several different domains for a given ambiguous query, e.g., the request for the keyword "jaguar" can give results about cars, animals, operating systems, and several other unrelated subjects. The fact that the user issuing the query usually browses the Internet for new car models would help in determining automatically the *interest domain* to which her query likely belongs. The second and more common situation where user-centric profiling information would be effective is when a search system attempts to offer *suggestions* along with the results of a given query, e.g., when the keywords were not selective enough. Users typically use ambiguous and too general queries, instead of selective ones that would filter out the results. Some tools included in modern search engine (such as the "Searches related to:" tool in Google) propose more targeted searches, but do not take into consideration the users' expectation and search history in the process. Only the frequency of the queries is used. For many users, and given the typically skewed, long-tail distribution of interests observed for Web content, the suggested queries may not give more satisfactory results than the ones that are already proposed on the first results pages of the search engine. This calls for new tools that can take advantage of user-centric and interest-profiling information to enhance the search engines capabilities with interest-awareness for better-tailored search.

An interesting observation is that, among groups of Internet users, interests for data typically follow the clustering property [1, 2, 14, 19]: two users who were interested to same topics in the past are more likely to be interested to the

same topics in the future. More, it is likely that elements searched by users interested in one such domain will also interest other users from that group in their future searches, either as additional results, or as suggestions to replace the missing selectiveness of their query by the mean of interest scoping based on collaboratively-built knowledge. Grouping users with similar interests has already been successfully exploited for greatly increasing the chances to locate new data when using unreliable search mechanisms such as flooding or random walks [7, 10, 14, 20, 23].

In this paper, we propose the general architecture of a system that pushes further the idea of exploiting collaboratively built search mechanisms, based on interest clustering and obtained through recommendations among users. The envisioned system is meant to be used either as a stand-alone search engine, e.g., for a peer-to-peer file sharing system, or perhaps more interestingly, as a companion for some existing search engine. We will concentrate on the description of the challenges that the design of such a companion system raises and on the discussion of the corresponding technical choices. Our overall objective is to construct a self-organized network of peers, each peer being attached to a single-user, with users (i.e., peers) grouped according to their shared interests. Obviously, each peer can participate and belong to several groups, for each of the main interest domains it has been assigned to. Thereafter, the knowledge at the neighbors from that peer is leveraged (1) to enhance the search recall by proposing the resources that were deemed interesting for the same requests by interest-neighbors, and (2) to deal with ambiguous queries by comparing the results return by some search engine with the interest-communities the querying user belongs to, and by re-ranking results accordingly.

The proposed system is a two layers, peer-to-peer (P2P), fully decentralized system. This choice is due a series of considerations about the system goals. First of all, such systems allow proposing a service without any centralized authority (e.g., a single server that would store all profiles and browsing histories of users) with the service implemented through the collaboration of the peers. It is also potentially more difficult for a node to cheat and bias the results given by the system, as a single node will only have limited impact for suggesting search results to its neighbors. Moreover, it is possible, as we will see, to prevent peers from disturbing the system by faking statistics about sites' popularity, while it is impossible to detect that form of cheating with a centralized server that could modify the order of sites, e.g., based on commercial reasons. Next, P2P systems allow us to solve the important problem of scale, as they do not require the over- and proportional-provisioning of resources that would be required with a centralized approach. The more peers participate, the more power is added to the system. A P2P approach scales well to large numbers of peers and it does not suffer from the *bootstrap problems* [12] (i.e., the difficulty, for a centralized and stand-alone service, to attract enough users to fully sustain its specific functionalities—here, the P2P system can be used in conjunction with an existing search engine). Finally, P2P systems are known to deal gracefully with system dynamism at no or very little additional cost, whereas centralized systems need expensive and complex techniques to ensure continuous operation under node and link failures.

Such a system poses several design and engineering challenges. This is why our envisioned system is based on a two level P2P organized network. First, it is necessary to construct the interest-based network, so that peers are effectively grouped with other peers that share similar interest in their various interest domains. This requires maintaining a representation of these interests (*user profiling*), to compare these profiles to determine their similarity (*similarity metrics*) and finally to propose distributed algorithms that cluster peers in interest-based groups based on this metric (*clustering algorithm*). Moreover, from an orthogonal point of view, the system has to care about security and privacy. Indeed, users would not want to use such a system if it allows others to spy on their browsing activity, or if malicious peers can extract the content of their cache in plain text during proximity evaluation.

The remaining of this paper is as follows. First, Section 2 discusses related work. Next, Section 3 reviews the various issues posed by the system construction and Section 4 elaborates on what should be the adequate system architecture, as well as the role of each of its components. Section 5 presents in more details each component and discusses the different issues that are to be faced by the implementer. Section 6 presents future work and concludes.

2. RELATED WORK

Many independent studies have observed the characteristics of accesses to distributed data in various contexts. The most important of these characteristics in the context of this paper are: clustering of the graph that links users based on their shared interests, correlation between past and futures accesses by users or by groups of users that share similar interests, skewness of the distribution of interests per peer, skewness of the distribution of accesses per data element. Skewness usually relates to Zipf-shape distributions, which are a feature of access behaviors amongst large groups of humans [28]. We first review the work related to the detection and use of interest correlation between users in large-scale systems.

The presence of communities amongst user interests and accesses in Web search traces [1, 2], peer-to-peer file sharing systems [14] or RSS news feeds subscriptions [19] can be exhibited.

The existence of a correlation of interests amongst a group of distributed users has been leveraged in a variety of contexts and for designing or enhancing various distributed systems. For peer-to-peer file sharing systems that include file search facilities (e.g., Gnutella, eMule, ...), a sound approach to increase recall and precision of the search is to group users based on their past search history or based on their current cache content [10, 13, 23]. Interestingly, the small-world [18] aspects of the graph of shared interests[1] linking users with similar profiles is observed and can be exploited not only for file sharing systems, but also in re-

[1]Small-world aspects for the shared interests graph are: (i) a high clustering, (ii) a low diameter due to the existence of a small proportion of long links, i.e., links to *exotic* domains that are distant from the common interests of the node and its regular neighbors and that act as cross-interest-domain links, and (iii) the possibility to navigate through the graph of interest proximity amongst peers and effectively find short path between two interest domains based only on the one-to-one distance relationships amongst these domains, i.e., without global knowledge of the graph.

searcher communities or in web access patterns [14]. Another potential use of interest clustering is to form groups of peers that are likely to be interested in the same content in the future, hence forming groups of subscribers in a content-based publish-subscribe [7]. Finally, interest correlation can be used to help bootstrapping and self-organization of dissemination structures such as network-delay-aware trees for RSS dissemination [20]. Finally, user interest correlation can be used for efficiently prefetching data in environments where access delays and resource usage constraints can be competing [26], as it is an effective way of predicting future accesses of the users with good accuracy.

The correlation between the users' past and present accesses has been used for user-centric ranking. In order to improve the personalization of search results, the most probable expectations of users are determined using their search histories stored on a centralized server [24,25]. Nevertheless, the correlation between users with similar search histories is not leveraged to improve the quality of result personalization, hence making the approach sound only for users with sufficiently long search histories.

An alternative class of *clustering search engines* uses semantic information in order to *cluster* results according to the general domain they belong in (and not as in our approach to cluster users based on their interests). This can be seen as a centralized, server-side and user-agnostic approach to the use of characteristics of distributed accesses to improve user experience. The clustering amongst data elements is derived from their vocabulary. It presents the user with results along different interest domains and can help her to disambiguate these results from a query that may cover several domains, e.g., the query word "apple" can relate to both *food/fruits* and *computers* domains. Examples of such systems are EigenCluster [8], Grouper [27], SnakeT [9] or TermRank [11]. Nonetheless, these systems simply modify the presentation of results so that the user decides herself in which domain the interesting results may fall–these results are not in any way automatically tailored to her expectations. They do not also consider the clustering of interest amongst users, but only the clustering in content amongst the data.

Aspects related to the distribution of the popularity of te elements or to the number of interest domains of the users are of particular importance in the peer-to-peer context, where the responsibility for these elements (or for these users) has to be distributed amongst a large set of nodes or servers. To achieve scalability, it is necessary to balance the load evenly amongst nodes. This is usually achieved by letting all peers interested in one element serve that element (e.g., as in the first versions of Gnutella) at the cost of reducing availability of unpopular resources, or in a more structured manner, to map elements to one or several nodes [17]. An example of a system using reorganization of the data responsibility to cope with skewed dynamic load is Mercury [3]. An example of using data replication for load balancing is given by the Beehive [21] system. An example of replication or split of the responsibility for a group of users is the publish/subscribe system SplitStream [6], which is based on the Pastry [22] DHT.

3. CHALLENGES

This Section lists the various research challenges that are associated with our system proposal. A clear definition of these challenges helps in defining and justifying the corresponding architecture, described in the next Section.

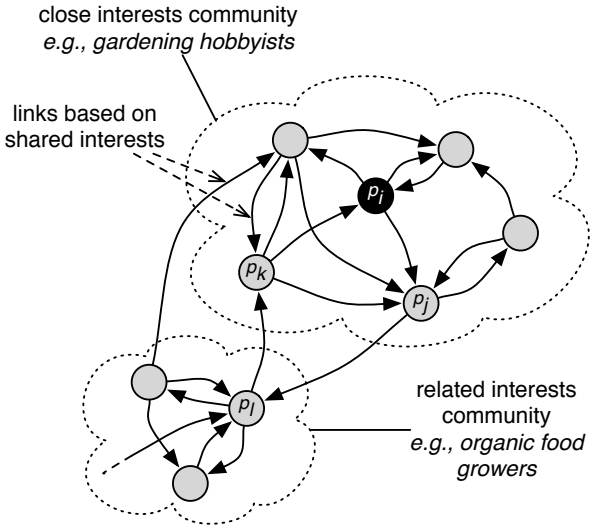


Figure 1: Interest-based network: general principle

3.1 Construction of an interest-based network

We present the general principle of the construction and use of a network of peers based on shared interest, in the context of Web search enhancing mechanisms. Figure 1 presents a coarse view of such a network. This example is purely fictional but helps for presenting the global idea.

Peers (e.g., peer p_i and p_j) are linked as they share interests for the same kind of content (or, more correctly, have been interested in the same content in the past and therefore are considered to have a high probability to be again common interests in content in future, i.e. when issuing searches for new content). Users are grouped by the means of a clustering protocol, in two different ways. First, each peer decides independently to which peer it is linked. These one-to-one relationships are chosen based on an interest-based distance, amongst the peer it encounters. Second, based on these one-to-one relationships and on their associated distances, peers are grouped in collectively-known and maintained interest groups (i.e., as collectively recognized communities of interests). An important point here is that a peer is not part of one single group but can participate in as many groups it requires to *cover* its interests. For instance, peer p_j is interested in, i.e. has been accessing resources about, both gardening and to a lesser extent to organic food production. It hence has links to members of these two groups and is "officially" part of one, but may as well be part of a completely different group, say, one grouping researchers that often search the web for new information retrieval papers.

Note that the labels given here are only for the sake of simplifying explanations: there is no automatic labeling, nor is there any ontology, in the system. The process of creating, deleting, merging or splitting interest-based clusters is completely automatic and solely based on statistical properties.

The task of creating such a network requires the following mechanisms: creating profiles of users that represent their interests, finding a way to construct the one-to-one links

with peers that are "close" in terms of shared interest, and to that extent, to define a metric of interest proximity.

3.1.1 User Profiling

User interests ideally represent the comprehensive set of thematic areas that are (most often) covered by the documents or items the user is accessing or is likely to access. The automated detection of interest domains is not easy, indeed, as typically user interests are dynamic and time-dependent. Typically, users can be interested in a topic only for a certain period due to either some personal reasons (e.g. who recently lost her/his job looks for a new position) or environmental ones (e.g. who lives or will visit Italy looks at Italian weather forecasting sites). Moreover, due to the fact users have different interests, users can, in their daily conducted browsing activity, access to web resources that are very different in content and heterogeneous in type. As a consequence, it makes the user behavior analysis for interest detection even more complex.

For the implementer, user profiles also have to respect two important properties: they have to be small and lightweight to allow a fast transmission and computation, and they have to hide as much as possible the plain content that is represented while allowing the comparison of what they represent (as part of the similarity metric computation). This calls for the use of space-constrained representation. A typical example is to use Bloom filters [4] that map a large set of elements[2] to a fixed size bit vector. Each element is hashed using k hash functions, setting the corresponding bits. Inclusion tests are made by testing for these same k bits, which require to know the resource name beforehand (hence adding intrinsic privacy support to the structure: it is impossible to reverse the process and obtain a plain text list of the visited web pages, for instance). Interestingly, while inclusion tests can yield false positives, comparing the size of two sets encoded with bloom filters (or, the size of their union/intersection, or their Jaccard similarity) gives good approximations. Counting filters and compact approximators are two possible alternatives that gives better precision (at the cost of a larger space usage). Time issues also have to be taken into account for the profiles: how much time, or how many elements, are to be kept in one profile, are particularly sensitive settings.

3.1.2 Similarity Metrics

The measure of similarity between two users, represented by some interest profiles, will eventually be used to form clusters of users, grouped together based on affinities. In this process, what matters is to be able to distinguish between two potential neighbors, which one is *closest* in terms of interest. The presence of interest, first, is denoted by accesses by both peers to the same elements (e.g. two users frequently visit a gardening-related webpage after looking up for information on their favorite search engine, or access Web pages that are described by similar keywords). Simply using the number of common elements has been successfully used in the context of P2P file sharing systems [23] or Web cache design [13]. Nevertheless, this poses the problem of the skewness in the number of elements represented by the profiles (which is due to the skewness in the number of accesses by each peer [1]), unless the profiles are kept to a fixed (or maximum) number of represented elements.

Moreover, it is important to note that a common interest for non-popular resources, or to several of them, represents future shared interests with higher accuracy. Therefore, a good metric has to take into account the popularity of each element that is encoded in each profile to weight the calculation. Nevertheless, this information is not available only at the couple of peers that are computing their interest-distance. The information about their local accesses would bias unpopular elements that are by indeed popular amongst these two peers, consider them as popular and reduce their weight, hence loosing the benefits of using a popularity-aware similarity metric. This requires some *global knowledge* about accesses, i.e., statistics about each page usage based on all accesses from all peers (or from an unbiased subset of these accesses).

3.2 Membership, Trust and Privacy

Other important challenges that are faced in constructing the envisioned system lie in the three closely related aspects of membership, trust and user privacy. All require carefully algorithmic designs that take them into account from the beginning.

Membership relates to the following problem: it is necessary to restrict peers from sending arbitrary data to the network to bias the view of other peers, e.g. by participating in multiple interests domains which they would not normally be part of. Moreover, each peer (user) has to be given limited (but fairly distributed) *credits* to participate in the creation of the global statistics.

Trust is linked to membership and is twofold: (1) for the collection of global data coming from interest based communities (2) for peers and users, the confidence they have in using these statistics as a basis for creating one-to-one relationships.

Finally, privacy is of particular concern. As the information that is shared to allow the creation of interest-based links is typically personal (URLs of visited Web pages, bookmarks, etc.) it is required that no peer can easily gather statistics about one particular user, in particular recreating in plain text the list of visited Web pages. This means that (1) a peer that manages information for one given page (typically, as a result of a routing process in the distributed index) does not need to know the original peer's IP who issued the information. Also, (2) a peer on that routing path should not be able to spy on the information that is sees when sending it to next hops.

4. SYSTEM ARCHITECTURE

Based on the challenges presented in the previous Section, we sketch here a distributed system architecture that has the necessary features and solidity. An overview of the architecture is given by Figure 2.

4.1 Network Architecture

We consider the following network setting: a large number of *regular peers* are simply accessing the network resources and issuing the queries, and some peers that dedicate some of their processing and network capacities to the well-functioning of the network, that we denote *backbone peers*. Regular peers are unreliable, not trustable and can leave

[2]These elements can be visited Web pages URL or their representing keywords (*snippet*), bookmark tags from an online annotation service such as `http://delicious.com/`, or any other information that represents the user interests.

Criteria	Information kept	Memory	Structure principle	View, reach	Churn rate	Trustability	Usage
Indexing layer	Pages frequencies, membership info., group mngmt info.	Long-lasting	Indexing distributed data structure (e.g. DHT)	Global view, routable	Low and fair	Correct	Reliable services for interest-proximity layer
Interest-proximity layer	Local browsing history, queries cache	Null to short-lasting	Self-emerging, proximity-based	Random-walks, local explorations	High and unfair	Unknown	Statistics collection for indexing layer, querying

Table 1: Interest-proximity layer and indexing layers: different peer characteristics and service objectives call to adapted structures.

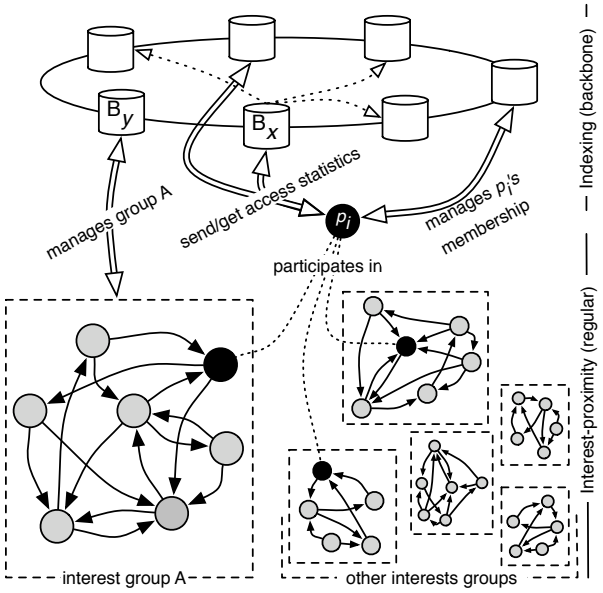


Figure 2: Two tier architecture and functional relationships between the interest-based layer and the more stable indexing layer using backbone peers.

or join the network at any time. Backbone peers are more reliable, and tend to leave the system gracefully. In their majority they are considered to be well behaving and trustable. Backbone peers can be, for instance, machines dedicated by ISPs or companies to the well functioning of the system. As the number of the regular peers grows, the number of backbone peers is expected to grow accordingly. We recap in Table 1 the characteristics of the two sets of peers, where we also derive the characteristics of the infrastructure-related aspects that better tie to each of them. The remaining of this Section explains each such decision and characteristic.

In our envisioned scenario, we face the problem of dealing with two types of information. The first level of information is related with global statistics (i.e., globally maintained Web sites popularity measurement). This information is used to bootstrap the interest-proximity layer, by allowing nodes to find similar peers when they are not still part of any group. These statistics are derived from data about groups. We also need to keep at a global level the identifiers and the signatures of existing groups, i.e.,the more relevant sites that distinguish the groups members. This data is used to ease all the global operation related with groups, e.g. retrieving existing groups and their features (i.e. signature), easy join/leave operations and derive global visit scores for the sites. The other level of information that is kept in the system is the local data associated with every user. This data is exploited by users to join to the network and find other group of peers sharing similar interests. Due to privacy concerns and the high variability of this information, it is not maintained at a global level. Instead, it is stored locally in every peer and only what the users allow to use is shared and used to compute similarities and contribute to the computation of group signatures.

Given the above remarks, we believe that the different nature and use of the two kinds of information present in the system require different ways of dealing with them.

In order to reflect these different needs, the overall network architecture of the system is composed of two different layers. One layer is the *backbone* layer, composed of the eponym stable and reliable peers. Due to their stability, they constitute the layer upon which global operations are possible. They are in charge of maintaining a long-lasting index that stores information concerning the global data of the network (hence, called *global* information).

In particular, the backbone layer stores the statistics about visited sites both at a global and at community levels. For this purpose it makes use of structured indices (i.e. DHT), in order to have an easy deterministic global store and retrieve operations. This information are critical to allow fast and fair community creations and maintenance.

The backbone network does not take in charge the formation and maintenance processes for the registered communities itself. Instead, this is delegated to the interest-proximity layer. Indexing the regular peers content is not possible, nor would be routing deterministically amongst those. More, the churn rate of these peers can be high enough to incur significant costs for the maintenance of such an index. But, while being self-structured and with no global view of the network at any, or from any, of its peer, this layer can still successfully leverage the indexed information from the backbone layer. The backbone layer has the responsibility to maintain this global data and the overall information (i.e. the signature) of each community. This data changes over time and is thus periodically refreshed. Nonetheless, is is not changed each time a peer performs a new action (i.e., visits a new site or increases the statistics about old sites).

The second layer is composed of more volatile, unreliable peers. These peers are attached to the users of the system. They usually have a high churn rate, since they constantly and unpredictably connect and disconnect to the network. Due to their nature, they are better organized using a self-

structured network, i.e., not trying to implement a globally coherent routing substrate amongst them. Those networks are more suitable to deal with less reliable peers since they can be more easily maintained. We are interested in forming community of users based on their respective profiles. Performing such a task using only a index-based substrate (i.e. the backbone layer) will involve a very high degree of requests and update activities for this layer. Network organizations based on a self-emerging paradigm have proven to be more suitable for the creation of spontaneous communities. Hence, the network is based on a gossip-style management system. P2P Gossip-based solutions for membership management have proved to be very efficient [15, 16]. This communication and information dissemination style is used by volatile peers to start building the "core" of a community that can be later used to build a stable community, whose data can then be stored in the backbone layer, allowing peers joining the system later on to speed-up their search for suitable communities. Since this network is formed by grouping together peers with similar interests, we call it the *interest-proximity* layer.

In the interest-proximity layer, communities are formed using similarities among volatile peer profiles. More similar peers can get in touch and connect to form neighborhood of similar users. Similarity is computed on the basis of user profiles, that consist of the sites visited by them during their browsing history.

5. COMPONENTS AND ALGORITHMS

5.1 Profile Creation

Profiles of peers are created based on the users' interests, represented by recently accessed resources.[3] The popularity of resources in a distributed system, e.g. the Web, is usually very sparse, typically following a Zipf-like [28] distribution: the popularity of the i^{th} least popular element is proportional to $i^{-\alpha}$ (the bigger α is, the sparsest is the distribution—α is typically around 1 for web pages popularity [1]). This means that in order to effectively decide that two users both accessing the same resource denote some kind of proximity in interest, one needs to make sure that that particular resource is not simply a vastly popular resource (e.g., `www.weather.com` or some search engine), that denotes less shared interest than mid-popular ones [2].

As a result of the aforementioned observations, it is necessary to track statistics about the popularity of pages to carefully select and weight more those that that convey more proximity of interest. To that extent, it is necessary to keep pages frequencies, based on the number of accesses by users (or on an unbiased sample or these). Clearly, the loose self-emerging but not controlled structure, with no indexing or routing mechanism, of the interest-proximity layer is not adapted for this matter: the accesses of neighbors in the interest-vicinity of some peer to Web content is itself biased by that interest proximity used to build the network. Instead, it is necessary to propose a more global and organized view of the system that would allow disposing of this information (equivalent to inverse document frequencies for data mining). This requires the different architectural choices presented in the previous section.

Based on Web sites' popularity that follow Zipf-like distributions, we extract and exploit what we call the MRFVS: the middle-range frequently visited sites. Namely, the sites that a user visited with high frequency individually but that are not the most frequent ones over all accesses by all users. The surrounding idea is that on one side the sites that are too frequently visited are not eligible for representing the user because they are accessed nearly by everyone but on the other side, the sites that are accessed only a few time are not sufficiently frequent in the user browsing activity for being considered as interesting from the point of view of the user.

The assumptions stating that MRFVS are the most relevant among the whole set of sites is not new, indeed, it is well-stated that the significance of the items in a Zipf-like distribution follows a Gaussian distribution that is maximized in the area we are considering for extracting the MRFVS. Nevertheless, in order to be able to exploit that information two issues have to be addressed: i) to decides the proper range of sites to consider and ii) to store the global statistical information for allowing to each single peer to extract from her/his browsing history the sites belonging to the ones globally considered as relevant. The former issue is a still open issue; the naive solution for finding the thresholds indicating the range is an iterative process that empirically decide the proper values. The latter issue can be addressed storing global statistical information about the MRFVS in the global index structure, which stores the (compacted and anonymized) list of sites belonging to at least one community signature, i.e., the sites representing the interests of that community.

5.2 Profile Similarity

Once a suitable method to describe each user interests is found and is coded in the peer profiles, we have to put attention on choosing a proper function to compare profiles. This is a particular relevant point, since this function determines the relationships between peers on the basis of their interests. As cited in the introduction, using simple functions, like counting the number of common items in the profiles, is not enough.

Instead, more accurate, although simple functions, should be used. Our proposal is to use a metric that takes into account the size of each profile, such as the Jaccard similarity, $\frac{|A \cap B|}{|A \cup B|}$, that have proven to be effective for that matter [10, 20].

We propose also to weight the mid-popular (MRVFS) elements in the profile at the moment of the calculation of the similarity metric. The ratio between popularity and weight in the similarity computation for pages is similar to the use TF-IDF (Term Frequency - Inverse Document Frequency) in data mining algorithms, except that the content of the document itself is not indexed.

5.3 Community Creation

As soon as each peer is able to compute its interest-based

[3]The information sources we use for extracting the data related to user browsing activity are, essentially, the history of visited web sites, the bookmarks saved by the user, the submitted queries and a either implicit or explicit user provided relevance feedback. Such gathered information have to be considered in a proper way, hence taking care of the time elapsed since the site have been visited. Indeed most recently visited sites are very important ones whereas the ones visited since a long time can be considered as not very important ones.

distance to any other peer, based on both its profile and that peer's profile, and based on information about the popularity of elements that compose the profiles (either directly or indirectly), its objective is to *group* with other peers that have close-by interests, in order to form the basis for *interests communities*. This process is done in a self-organizing and completely decentralized manner. Each peer knows a set of other peers, called its interest neighbors, and tries periodically to choose new such neighbors that are closer to its interest than the previous ones. This is simply done by learning about new peers from some other peer or from a bootstrap mechanism, then retrieving their profile, and finally choosing the C nearest neighbors in the union of present and potential neighbors.

The bootstrap mechanism leans on the backbone network. A peer p that joins the network can ask to the backbone layer to send it links to peers belonging to the most similar communities available in the network. The similarity is computed between p's profile and the communities' signatures. The backbone layer selects the most suitable candidate communities, sends their IDs and the computed similarity to p, which, in turn, select the best and proper communities to join, on the basis of thresholds on the similarity score. For the selected communities, the backbone layer retrieves some contact peers inside each community and puts p in contact with those. The join process can then continue using the volatile network. Indeed, the selected peers send to p some other links from their neighbor list, these links lying inside the chosen community. The process stops once p reaches the desired number of neighbors for all the communities it has joined, and the gossip exchanges of neighborhoods information helps with maintaining a high quality of peers neighborhoods thereafter.

When a peer enters the network, it is put in contact with one or more peers already taking part in the interest-proximity network. They use the profile similarity function to compute how similar they are. Moreover, the peers contacted by p use the same similarity function to determine which are, among their neighbors, the most similar to p and route the join request of p toward them. All the peers that receive that request will react using the same protocol described above. This mechanism will lead p to learn the existence of its most similar nodes in the network and allow it to connect with them. In doing this process, the involved peers can only use global usage statistics to compare their respective profiles. Since close similarity scores can be obtained by using different sets of sites in the peers' profiles, p can use the information given by its newly added neighbors to group them on the basis of the most common visited sites. These groups try to reflect how neighbors are divided, considering p's different interests.

Since p is not yet part of any community, it can try to create a new one, starting from its neighbors' groups. Groups represent the seeds of new possible communities. If the cardinality of the group neighborhood exceeds a given threshold, p can start a new community creation election process. Using the public profiles of its neighbors, it constructs the signature of the new potential community. Then, it asks its neighbors whether or not they want to join the new community. In the case votes for the adhesions are over a given threshold, the new community can be built. p can request a new identifier to the backbone network, spreads it among the other community members and then sends the signature to the backbone layer. This layer will then keep the information about the signature and the frequencies associated with the signature's sites.

Related with the communities' maintenance processes are also the split and fusion processes. The split process happens when a community has grown too big. This kind of evaluation is performed locally, at the interest-proximity layer level. The initiative can be taken by any peer of the community that, by simply checking its neighbors table, discovers that it has too high a number of neighbors for that community. It can then decide to initiate a split.

The split proceeds as follows. The first step consists in computing the similarity with the other community members and tox take the first C (with C large enough) of them as the possible candidates for building the new community. Then, it computes the signature of the new community and sends it to the new potential members asking them to cast votes for the community creation. If the number of adhesions is sufficient, the new community can be created, by communicating it to the backbone layer. As a consequence, the signature of the old community has to change. This operation could be done by the backbone layer. Updates of local routing tables can be done through the interest-proximity layer, via messages propagated by the community members to their old neighbors. In principle, a node can take part to both the old and the new communities. The split simply give more "specialization" to the neighbors, by better focusing their interests and, thus, the relationship among them.

The opposite operation of a split is a fusion or merge operation. In this case, a peer may observe that the number of neighbors, over some period of time for a given community have fallen under a given threshold. Hence, it may start a merge process. It works in a similar way as the join process done by a single peer. The difference, in this case, is that instead of using the peer profile, the community signature is used. Once the most similar other community is found, the new signature is computed and an election request to the peers of both communities is sent. In case the two communities decide to merge, a request for a new community identifier is sent to the backbone network that register also the new community signature.

5.4 Community Signature

The signature of a community represents the cumulative (related with the neighborhood clustering for dealing with heterogeneity) profile of the set of peer that has joined that community. As for a single-peer's profile, it is represented by the sites with mid-frequency, considering all the sites visited its members.

The signature is created at the creation of a community. Peers that have agreed to become part of it communicate their visited sites (again, in an aggregated and privacy-preserving form) to the backbone network. In this case, they communicate also the sites that have no relevance for the community, with a frequency equal to 0. This is done just to avoid further coming nodes to be restricted to the nodes added so far and be able to increase the relevance of nodes that have not yet considered relevant inside a community.

A community signature is maintained by the backbone layer. It stores the community identifier and associates to it the information about the community-visited sites. This information consists in the identifiers of the sites in the back-

bone network, since other stable peers are in charge maintaining up-to-date data about sites frequencies.

Every time a peer joins or leaves a community, it may introduce changes to the community signature. For this purpose, every given amount of time T, members of a community have to start a renewal process of the community signature. The signature has to be re-computed using the new data and, when done, peers check whether they still belong to that community or not. In case they do not, they start searching new suitable communities, using the interest-proximity layer.

6. CONCLUSIONS AND FUTURE WORK

The focus of this paper is on addressing the problem of clustering Web users in a purely decentralized way. This is particularly useful for enabling an automated creation of communities made from users sharing common interests. In this paper we presented the overall architecture of a peer-to-peer system exploiting collaboratively built search mechanisms. The architecture is based on two different network layers: the indexing layer and the interest-proximity layer. The first one represent the so-called backbone of the network, namely a set of "institutional", reliable and trusted peers provided by ISPs that are not expected to churn or to exploit in an improper way the privacy related data. The second layer consists of churn-prone unreliable and untrusted peers connected in a self-emerging topology according to interest-based communities of users.

We have discussed in this paper the main challenges faced in designing the architecture of our collaborative search system. These issues include the creation of user profiles and their maintenance, the similarity metrics for computing how close users are in terms on interests, and issues related to the security and privacy. We presented our main vision and proposed a set of solutions for each challenge. Yet, much work remains to be performed, both from a design and implementation point of view, toward our vision of a peer-to-peer system maintaining Web-user clusters in a scalable, precise, secure, and privacy-preserving manner. Besides the actual implementation of the system, we are currently validating our algorithms using real-world Web browsing data and working on improving the privacy-preserving and security features.

Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Program FP7/2007-2013 under grant agreement 215483 (S-Cube). This work was conducted during the tenure of Étienne Rivière's "Alain Bensoussan" ERCIM fellowship.

7. REFERENCES

[1] L. A. Adamic and B. A. Huberman. Zipf's law and the internet. *Glottometrics*, 3:143–150, 2002.

[2] R. Akavipat, L.-S. Wu, F. Menczer, and A. Maguitman. Emerging semantic communities in peer web search. In *Proc. of P2PIR'06*, pages 1–8, 2006.

[3] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: supporting scalable multi-attribute range queries. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 353–366, New York, NY, USA, 2004. ACM Press.

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

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 1998.

[6] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: high-bandwidth multicast in cooperative environments. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 298–313, New York, NY, USA, 2003. ACM Press.

[7] R. Chand and P. A. Felber. Semantic peer-to-peer overlays for publish/subscribe networks. In *Proceedings of Europar'05, European Conference on Parallel Processing*, pages 1194–1204, Lisboa, Portugal, Sept. 2005.

[8] D. Cheng, R. Kannan, S. Vempala, and G. Wang. A divide-and-merge methodology for clustering. *ACM Trans. Database Syst.*, 31(4):1499–1525, 2006.

[9] P. Ferragina and A. Gulli. A personalized search engine based on web-snippet hierarchical clustering. *Softw. Pract. Exper.*, 38(2):189–225, 2008.

[10] P. Fraigniaud, P. Gauron, and M. Latapy. Combining the use of clustering and scale-free nature of user exchanges into a simple and efficient p2p system. In *Proc. of EuroPar'05*, 2005.

[11] F. Gelgi and H. D. S. Vadrevu. Term ranking for clustering web search results. In *Proceedings of the 10th International Workshop on Web and Databases (WebCD 2007)*, Beijing, China, jun 2007.

[12] H. Gylfason, O. Khan, and G. Schoenebeck. Chora: Expert-based p2p web search. In *Proc. of AAMAS'06*, Hakodate, Japan, may 2006.

[13] S. B. Handurukande, A.-M. Kermarrec, F. Le Fessant, L. Massoulié, and S. Patarin. Peer sharing behaviour in the edonkey network, and implications for the design of server-less file sharing systems. In *Proceedings of Eurosys'06*, Leuven, Belgium, apr 2006.

[14] A. Iamnitchi, M. Ripeanu, and I. Foster. Small-world file-sharing communities. *ArXiv Computer Science e-prints*, July 2003.

[15] M. Jelasity and O. Babaoglu. T-Man: Gossip-based overlay topology management. In *Proceedings of Engineering Self-Organising Applications (ESOA'05)*, July 2005.

[16] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. Gossip-based peer sampling. *ACM Trans. Comput. Syst.*, 25(3):8, 2007.

[17] D. R. Karger and M. Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '04)*, 2004.

[18] J. Kleinberg. Navigation in a small world. *Nature*, 406, 2000.

[19] H. Liu, V. Ramasubramanian, and E. G. Sirer. Client behavior and feed characteristics of RSS, a publish-subscribe system for web micronews. In *Proceedings of the 2005 Internet Measurement Conference (IMC 2005)*, Oct. 2005.

[20] J. A. Patel, E. Rivière, I. Gupta, and A.-M. Kermarrec. Rappel: Exploiting interest and network locality to improve fairness in publish-subscribe systems. *Computer Networks*, 2009. In Press.

[21] V. Ramasubramanian and E. G. Sirer. Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, Berkeley, CA, USA, 2004.

[22] A. Rowstron and P. Druschel. Pastry: scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proc. of Middleware'01*, pages 329–350, Nov. 2001.

[23] K. Sripanidkulchai, B. Maggs, and H. Zhang. Efficient content location using interest-based locality in peer-to-peer systems. In *IEEE Infocom, San Francisco, CA, USA*, Mar. 2003.

[24] B. Tan, X. Shen, and C. Zhai. Mining long-term search history to improve search accuracy. In *Proc. of SIGKDD'06*, pages 718–723, Philadelphia, PA, USA, 2006.

[25] J. Teevan, S. T. Dumais, and E. Horvitz. Personalizing search via automated analysis of interests and activities. In *Proc. of SIGIR-IR'05*, pages 449–456, Salvador, Brazil, 2005.

[26] T. Wu, S. Ahuja, and S. Dixit. Efficient mobile content delivery by exploiting user interest correlation. In *Proc. WWW (Poster)*, 2003.

[27] O. Zamir and O. Etzioni. Grouper: a dynamic clustering interface to web search results. *Computer Networks*, 31(11-16):1361–1374, 1999.

[28] G. K. Zipf. *Human Behavior and the Principle of Least Effort*. Harvard University Press, 1949.