LSDS-IR'10

8th Workshop on Large-Scale Distributed Systems for Information Retrieval

Workshop co-located with ACM SIGIR 2010

Geneva, Switzerland, July 23, 2010

Workshop Chairs

Roi Blanco, Yahoo! Research, Barcelona, Spain
B. Barla Cambazoglu, Yahoo! Research, Barcelona, Spain
Claudio Lucchese, ISTI-CNR, Pisa, Italy

Steering Committee

Flavio Junqueira, Yahoo! Research, Barcelona, Spain
Fabrizio Silvestri, ISTI-CNR, Italy

Program Committee

Karl Aberer, EPFL, Switzerland
Ismail Altingovde, Bilkent University, Turkey
Ricardo Baeza-Yates, Yahoo! Research, Barcelona, Spain
Ranieri Baraglia, ISTI-CNR, Italy
Fabrizio Falchi, ISTI-CNR, Italy
Ophir Frieder, Illinois Institute of Technology, Chicago, USA
Sebastian Michel, EPFL, Switzerland
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
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
Justin Zobel, NICTA, Australia

Contents

Twitter

Abdur Chowdhury

Twitter

Dr. Abdur Chowdhury serves as Twitter's Chief Scientist. Prior to that, Dr. Chowdhury co-founded Summize, a real-time search engine sold to Twitter in 2008. Dr. Chowdhury has held positions at AOL as their Chief Architect for Search, Georgetown's Computer Science Department and University of Maryland's Institute for Systems Research. His research interest lays in Information Retrieval focusing on making information accessible.

Towards a Distributed Search Engine

Ricardo Baeza-Yates

Yahoo!

Ricardo Baeza-Yates is VP of Yahoo! Research for Europe and Latin America, leading the labs at Barcelona, Spain and Santiago, Chile. Previously full professor at Univ. of Chile and ICREA research professor at UPF in Barcelona. Co-author of Modern Information Retrieval (Addison-Wesley, 1999) among other books and publications. Member of the ACM, AMS, IEEE (Senior), SIAM and SCCC, as well as the Chilean Academy of Sciences. Awards from American Organization States, Institute of Engineers of Chile, and COMPAQ. His research interests includes algorithms and data structures, information retrieval, web mining, text and multimedia databases, software and database visualization, and user interfaces.

Query-Based Sampling using Snippets

Almer S. Tigelaar
Database Group, University of Twente,
Enschede, The Netherlands
a.s.tigelaar@cs.utwente.nl

Djoerd Hiemstra
Database Group, University of Twente,
Enschede, The Netherlands
hiemstra@cs.utwente.nl

ABSTRACT

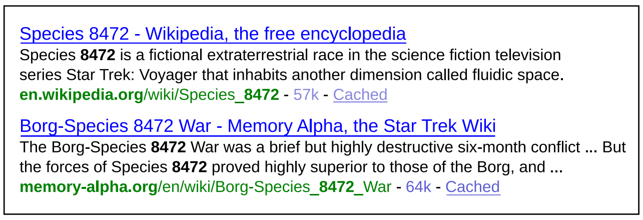
Query-based sampling is a commonly used approach to model the content of servers. Conventionally, queries are sent to a server and the documents in the search results returned are downloaded in full as representation of the server's content. We present an approach that uses the document snippets in the search results as samples instead of downloading the entire documents. We show this yields equal or better modeling performance for the same bandwidth consumption depending on collection characteristics, like document length distribution and homogeneity. Query-based sampling using snippets is a useful approach for real-world systems, since it requires no extra operations beyond exchanging queries and search results.

1. INTRODUCTION

Query-based sampling is a technique for obtaining a resource description of a search server. This description is based on the downloaded content of a small subset of documents the server returns in response to queries [8]. We present an approach that requires no additional downloading beyond the returned results, but instead relies solely on information returned as part of the results: the snippets.

Knowing what server offers what content allows a central server to forward queries to the most suitable server for handling a query. This task is commonly referred to as *resource selection* [6]. Selection is based on a representation of the content of a server: a *resource description*. Most servers on the web are uncooperative and do not provide such a description, thus query-based sampling exploits only the native search functionality provided by such servers.

In conventional query-based sampling, the first step is sending a query to a server. The server returns a ranked list of results of which the top N most relevant documents are downloaded and used to build a resource description. Queries are randomly chosen, the first from an external resource and subsequent queries from the description built so far. This repeats until a stopping criterion is reached [7, 8].


Species 8472 - Wikipedia, the free encyclopedia
Species **8472** is a fictional extraterrestrial race in the science fiction television series Star Trek: Voyager that inhabits another dimension called fluidic space.
en.wikipedia.org/wiki/Species_**8472** - 57k - Cached

Borg-Species 8472 War - Memory Alpha, the Star Trek Wiki
The Borg-Species **8472** War was a brief but highly destructive six-month conflict ... But the forces of Species **8472** proved highly superior to those of the Borg, and ...
memory-alpha.org/en/wiki/Borg-Species_**8472**_War - 64k - Cached

Figure 1: Example snippets. From top to bottom: each snippet consists of an underlined title, a two line summary and a link.

Disadvantages of downloading entire documents are that it consumes more bandwidth, is impossible if servers do not return full documents, and does not work when the full documents themselves are non-text: multimedia with short summary descriptions. In contrast, some data always comes along 'for free' in the returned search results: the snippets. A snippet is a short piece of text consisting of a document title, a short summary and a link as shown in Figure 1. A summary can be either dynamically generated in response to a query or is statically defined [16, p. 157]. We postulate that these snippets can also be used for query-based sampling to build a language model. This way we can avoid downloading entire documents and thus reduce bandwidth usage and cope with servers that return only search results or contain multimedia content. However, since snippets are small we need to see many of them. This means that we need to send more queries compared with the full document approach. While this increases the query load on the remote servers, it is an advantage for live systems that need to sample from document collections that change over time, since it allows continously updating the language model, based on the results of live queries.

Whether the documents returned in response to random queries are a truly random part of the underlying collection is doubtful. Servers have a propensity to return documents that users indicate as important and the number of in-links has a substantial correlation with this importance [1]. This may not be a problem, as it is preferable to know only the language model represented by these important documents, since the user is likely to look for those [3]. Recent work [5] focuses on obtaining uniform random samples from large search engines in order to estimate their size and overlap. Others [20] have evaluated this in the context of obtaining resource descriptions and found that it does not consistently work well across collections.

The foundational work for acquiring resource descriptions via query-based sampling was done by Callan et al. [7, 8]. They show that a small sample of several hundred documents can be used for obtaining a good quality resource description of large collections consisting of hundreds of thousands of documents. The test collection used in their research, TREC123, is not a web data collection. While this initially casts doubt on the applicability of the query-based sampling approach to the web, Monroe et al. [18] show that it also works very well for web data.

The approach we take has some similarities with prior research by Paltoglou et al. [19]. They show that downloading only a part of a document can also yield good modelling performance. However, they download the first two to three kilobytes of each document in the result list, whereas we use small snippets and thus avoid any extra downloading beyond the search results.

Our main research question is:

> "How does query-based sampling using *only snippets* compare to downloading full documents in terms of the learned language model?"

We show that query-based sampling using snippets offers similar performance compared to using full documents. However, using snippets uses less bandwidth and enables constantly updating the resource description at no extra cost. Additionally, we introduce a new metric for comparing language models in the context of resource descriptions and a method to establish the homogeneity of a corpus.

We describe our experimental setup in section 2. This is followed by section 3 which shows the results. Finally, the paper concludes with sections 4 and 5.

2. METHODOLOGY

In our experimental set-up we have one remote server which content we wish to estimate by sampling. This server can only take queries and return search results. For each document a title, snippet and download link is returned. These results are used to locally build a resource description in the form of a vocabulary with frequency information, also called a language model [7]. The act of submitting a query to the remote server, obtaining search results, updating the local language model and calculating values for the evaluation metrics is called an *iteration*. An iteration consists of the following steps:

1. Pick a one-term query.
   (a) In the first iteration our local language model is empty and has no terms. In this case we pick a random term from an external resource as query.
   (b) In subsequent iterations we pick a random term from our local language model that we have not yet submitted previously as query.
2. Send the query to the remote server, requesting a maximum number of results ($n = 10$). In our set-up, the maximum length of the document summaries may be no more than 2 fragments of 90 characters each ($s \leq 2 \cdot 90$).
3. Update the resource description using the returned results ($1 \leq n \leq 10$).
   (a) For the full document strategy: download all the returned documents and use all their content to update the local language model.
   (b) For the snippet strategy: use the snippet of each document in the search results to update the local language model. If a document appears multiple times in search results, use its snippet only if it differs from previously seen snippets of that document.
4. Evaluate the iteration by comparing the *unstemmed* language model of the remote server with the local model (see metrics described in Section 2.2).
5. Terminate if a stopping criterion has been reached, otherwise go to step 1.

Table 1: **Properties of the data sets used.**

Name	Raw	Index	#Docs	# Terms	# Unique
OANC	97M	117M	8,824	14,567,719	176,691
TREC123	2.6G	3.5G	1,078,166	432,134,562	969,061
WT2G	1.6G	2.1G	247,413	247,833,426	1,545,707
WIKIL	163M	84M	30,006	9,507,759	108,712
WIKIM	58M	25M	6,821	3,003,418	56,330

Since the snippet approach uses the title and summary of each document returned in the search result, the way in which the summary is generated affects the performance. Our simulation environment uses Apache Lucene which generates keyword-in-context document summaries [16, p. 158]. These summaries are constructed by using words surrounding a query term in a document, without keeping into account sentence boundaries. For all experiments the summaries consisted of two keyword-in-context segments of maximally ninety characters. This length boundary is similar to the one modern web search engines use to generate their summaries. One might be tempted to believe that snippets are biased due to the fact that they commonly also contain the query terms. However, in full-document sampling the returned documents *also* contain the query and have a similar bias, although mitigated by document length.

2.1 Data sets

We used the following data sets to conduct our tests:

OANC-1.1:	The Open American National Corpus: A heterogeneous collection. We use it exclusively for selecting bootstrap terms [14].
TREC123:	A heterogeneous collection consisting of TREC Volumes 1–3. Contains: short newspaper and magazine articles, scientific abstracts, and government documents [12]. Used in previous experiments by Callan et al. [7]
WT2G:	Web Track 2G: A small subset of the Very Large Corpus web crawl conducted in 1997 [13].
WIKIL:	The large Memory Alpha Wiki. `http://memory-alpha.org`
WIKIM:	The medium sized Fallout Wiki. `http://fallout.wikia.com`

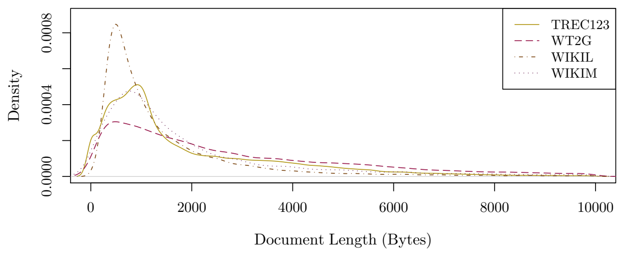


Figure 2: Kernel density plot of document lengths up to 10 Kilobytes for each collection.

The OANC is used as external resource to select a bootstrap term on the first iteration: we pick a random term out of the top 25 most-frequent terms (excluding stop words). TREC123 is for comparison with Callan's work [7]. WT2G is a representative subset of the web. It has some deficiencies, such as missing inter-server links [2]. However, since we use only the page data, this is not a major problem for this experiment.

Our experiment is part of a scenario where many sites offer searchable content. With this in mind using larger monolithic collections, like ClueWeb, offers little extra insights. After all: there are relatively few websites that provide gigabytes or terabytes of information, whereas there is a long tail that offers smaller amounts. For this purpose we have included two Wiki collections in our tests: WIKIL and WIKIM. All Wiki collection were obtained from Wikia, on October 5th 2009. Wikis contain many pages in addition to normal content pages. However, we index *only* content pages which is the reason the raw sizes of these corpora are bigger than the indices.

Table 1 shows some properties of the data sets. We have also included Figure 2 which shows a kernel density plot of the size distributions of the collections [21]. We see that WT2G has a more gradual distribution of document lengths, whereas TREC123 shows a sharper decline near two kilobytes. Both collections consist primarily of many small documents. This is also true for the Wiki collections. Especially the WIKIL collection has many very small documents.

2.2 Metrics

Evaluation is done by comparing the complete remote language model with the subset local language model each iteration. We discard stop words, and compare terms unstemmed. Various metrics exist to conduct this comparison. For comparability with earlier work we use two metrics and introduce one new metric in this context: the Jensen-Shannon Divergence (JSD), which we believe is a better choice than the others for reasons outlined below.

We first discuss the Collection Term Frequency (CTF) ratio. This metric expresses the coverage of the terms of the locally learned language model as a ratio of the terms of the actual remote model. It is defined as follows [8]:

$$CTF_{ratio}\left(\mathscr{T}, \hat{\mathscr{T}}\right) = \frac{1}{\alpha} \cdot \sum_{\mathbf{t} \in \hat{\mathscr{T}}} CTF\left(\mathbf{t}, \mathscr{T}\right) \qquad (1)$$

where $\mathscr{T}$ is the actual model and $\hat{\mathscr{T}}$ the learned model. The CTF function returns the number of times a term $\mathbf{t}$ occurs in the given model. The symbol α represents the sum of the CTF of all terms in the actual model $\mathscr{T}$, which is simply the number of tokens in $\mathscr{T}$. The higher the CTF ratio, the more of the important terms have been found.

The Kullback-Leibler Divergence (KLD) gives an indication of the extent to which two probability models, in this case our local and remote language models, will produce the same predictions. The output is the number of additional bits it would take to encode one model into the other. It is defined as follows [16, p. 231]:

$$KLD\left(\mathscr{T} \parallel \hat{\mathscr{T}}\right) = \sum_{\mathbf{t} \in \mathscr{T}} P\left(\mathbf{t} \mid \mathscr{T}\right) \cdot \log \frac{P\left(\mathbf{t} \mid \mathscr{T}\right)}{P\left(\mathbf{t} \mid \hat{\mathscr{T}}\right)} \qquad (2)$$

where $\hat{\mathscr{T}}$ is the learned model and $\mathscr{T}$ the actual model. KLD has several disadvantages. Firstly, if a term occurs in one model, but not in the other it will produce zero or infinite numbers. Therefore, we apply Laplace smoothing, which simply adds one to all counts of the learned model $\hat{\mathscr{T}}$. This ensures that each term in the remote model exists at least once in the local model, thereby avoiding divisions by zero [3]. Secondly, the KLD is asymmetric, which is expressed using the double bar notation. Manning [17, p. 304] argues that using Jensen-Shannon Divergence (JSD) solves both problems. It is defined in terms of the KLD as [9]:

$$JSD\left(\mathscr{T}, \hat{\mathscr{T}}\right) = KLD\left(\mathscr{T} \| \frac{\mathscr{T} + \hat{\mathscr{T}}}{2}\right) + KLD\left(\hat{\mathscr{T}} \| \frac{\mathscr{T} + \hat{\mathscr{T}}}{2}\right) \qquad (3)$$

The Jensen-Shannon Divergence (JSD) expresses how much information is lost if we describe two distributions with their average distribution. This distribution is formed by summing the counts for each term that occurs in either model and taking the average by dividing this by two. Using the average is a form of smoothing which avoids changing the original counts in contrast with the KLD. Other differences with the KLD are that the JSD is symmetric and finite. Conveniently, when using a logarithm of base 2 in the underlying KLD, the JSD ranges from 0.0 for identical distributions to 2.0 for maximally different distributions.

3. RESULTS

In this section we report the results of our experiments. Because the queries are chosen randomly, we repeated the experiment 30 times.

Figure 3 shows our results on TREC123 in the conventional way for query-based sampling: a metric against the number of iterations on the horizontal axis [7]. We have omitted graphs for WT2G and the Wikia collections as they are highly similar in shape.

As the bottom right graph shows, the amount of bandwidth consumed when using full documents is much larger than when using snippets. Full documents downloads each of the ten documents in the search results, which can be potentially large. Downloading all these documents also uses many connections to the server: one for the search results plus ten for the documents, whereas the snippet approach uses only one connection for transferring the search results and performs no additional downloads.

The fact that the full documents approach downloads a

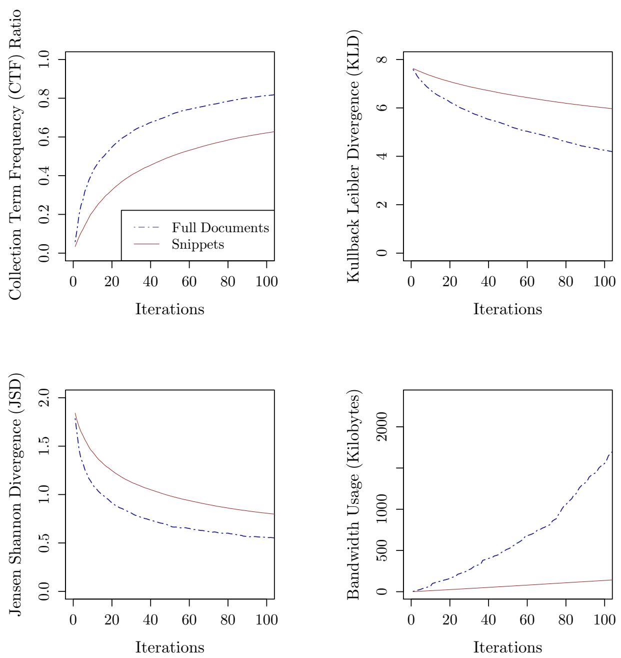


Figure 3: Results for TREC123. Shows CTF, KLD, JSD and bandwidth usage, plotted against the number of iterations. Shows both the full document and snippet-based approach. The legend is shown in the top left graph.

lot of extra information results in it outperforming the snippet approach for the defined metrics as shown in the other graphs of Figure 3. However, comparing this way is unfair. Full document sampling performs better, simply because it acquires more data in fewer iterations. A more interesting question is: how effectively do the approaches use bandwidth?

3.1 Bandwidth

Figures 4 and 5 show the metrics plotted against bandwidth usage. The graphs are 41-point interpolated plots based on experiment data. These plots are generated in a similar same way as recall-precision graphs, but they contain more points: 41 instead of 11, one every 25 kilobytes. Additionally, the recall-precision graphs, as frequently used in TREC, use the maximum value at each point [11]. We use linear interpolation instead which uses averages.

Figure 4 shows that snippets outperform the full document approach for all metrics. This seems to be more pronounced for WT2G. The underlying data reveals that snippets yield much more stable performance increments per unit of bandwidth. Partially, this is due to a larger quantity of queries. The poorer performance of full documents is caused by variations in document length and quality. Downloading a long document that poorly represents the underlying collection is heavily penalised. The snippet approach never makes very large 'mistakes' like this, because its document length is bound to the maximum summary size.

TREC123 and WT2G are very large heterogeneous test collections as we will show later. The WIKI collections are more homogeneous and have different document length dis-

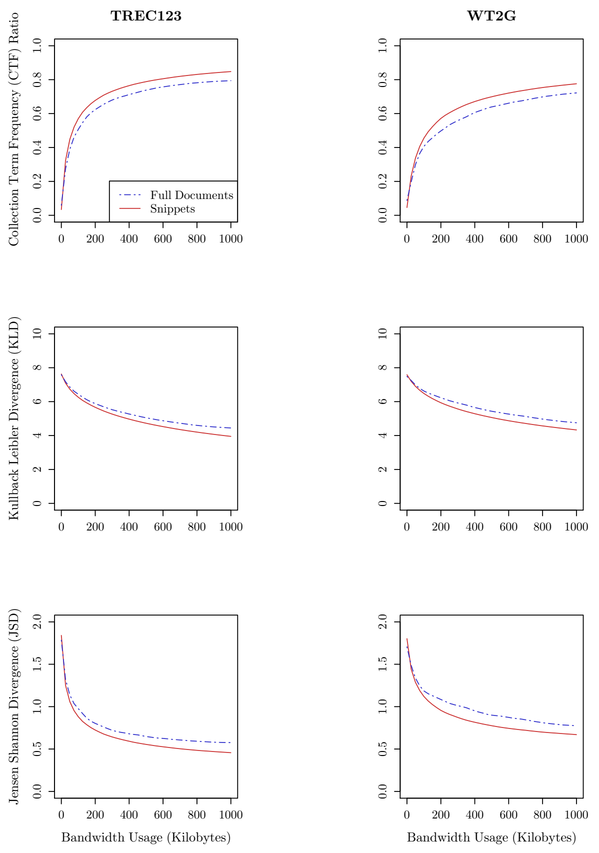


Figure 4: Interpolated plots for all metrics against bandwidth usage up to 1000 KB. The left graphs show results for TREC123, the right for WT2G. Axis titles are shown on the left and bottom graphs, the legend in the top left graph.

tribution characteristics. In Figure 5 we see that the performance of snippets on the WIKIL corpus is worse for the JSD, but undecided for the other metrics. For WIKIM performance measured with CTF is slightly better and undecided for the other metrics. Why this difference? We conducted tests on several other large size Wiki collections to verify our results. The results suggest that there is some relation between the distribution of document lengths and the performance of query-based sampling using snippets. In Figure 2 we see a peak at the low end of documents lengths for WIKIL. Collections that exhibit this type of peak all showed similar performance as WIKIL: snippets performing slightly worse especially for the JSD. In contrast, collections that have a distribution like WIKIM, also show similar performance: slightly better for CTF. Collections that have a less pronounced peak at higher document lengths, or a more gradual distribution appear to perform at least as good or better using snippets compared to full documents.

The reason for this is that as the document size decreases and approaches the snippet summary size, the full document strategy is less heavily penalised by mistakes. It can no longer download very large unrepresentative documents, only small ones. However, this advantage is offset if the document sizes equal the summary size. In that case the

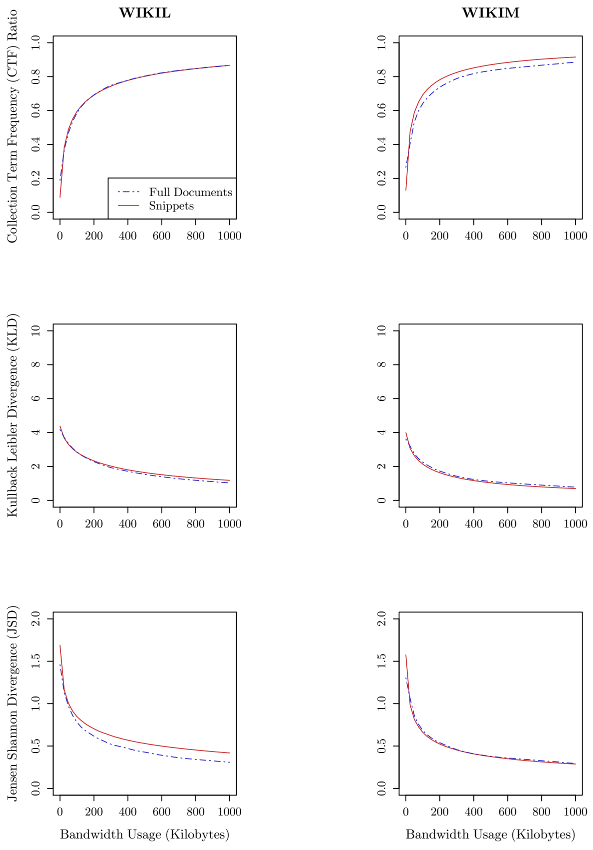


Figure 5: Interpolated plots for all metrics against bandwidth usage up to 1000 KB. The left graphs show results for WIKIL, the right for WIKIM. Axis titles are shown on the left and bottom graphs, the legend in the top left graph.

full document approach would actually use double the bandwidth with no advantage: once to obtain the search results, with summaries, and once again to download the entire documents which are the same as the summaries in the search results.

3.2 Homogeneity

While WIKIM has a fairly smooth document length distribution, the performance increase of snippets over full documents with regard to the JSD and KLD metrics is not the same as that obtained with TREC123 and WT2G. This is likely caused by the homogeneous nature of the collection. Consider that if a collection is highly homogeneous, only a few samples are needed to obtain a good representation. Every additional sample can only slightly improve such a model. In contrast, for a heterogeneous collection, each new sample can improve the model significantly.

So, how homogeneous are the collections that we used? We adopt the approach of Kilgariff and Rose [15] of splitting the corpus into parts and comparing those, with some slight adjustments. As metric we use the Jensen-Shannon Divergence (JSD) explained in Section 2.2 and also used by Eiron and McCurley [10] for the same task. The exact procedure we used is as follows:

Table 2: Collection homogeneity expressed as Jensen-Shannon Divergence (JSD): Lower scores indicate more homogeneity ($n = 100, \sigma = 0.01$).

Collection name	μ JSD
TREC123	1.11
WT2G	1.04
WIKIL	0.97
WIKIM	0.85

1. Select a random sample $\mathscr{S}$ of 5000 documents from a collection.
2. Randomly divide the documents in the sample $\mathscr{S}$ into ten bins: $s_1 \ldots s_{10}$. Each bin contains approximately 500 documents.
3. For each bin s_i calculate the Jensen-Shannon Divergence (JSD) between the *bigram* language model defined by the documents in bin s_i and the language model defined by the documents in the remaining nine bins. Meaning: the language model of documents in s_1 would be compared to that of those in $s_2 \ldots s_{10}$, et cetera. This is known as a leave-one-out test.
4. Average the ten JSD scores obtained in step 3. The outcome represents the homogeneity. The lower the number, the more self similarity within the corpus, thus the more homogeneous the corpus is.

Because we select documents from the collection randomly in step 1, we repeated the experiment ten times for each collection. Results are shown in Table 2.

Table 2 shows that the large collections we used, TREC123 and WT2G, are more heterogeneous compared to the smaller collections WIKIL and WIKIM. It appears that WIKIL is more heterogeneous than WIKIM, yet snippet-based sampling performs better on WIKIM. We conjecture that this is caused by the difference in document length distributions discussed earlier: see Figure 2. Overall, it appears that query-based sampling using snippets is better suited towards heterogeneous collections with a smooth distribution of document lengths.

4. CONCLUSION

We have shown that query-based sampling using snippets is a viable alternative for conventional query-based sampling using entire documents. This opens the way for distributed search systems that do not need to download documents at all, but instead solely operate by exchanging queries and search results. Few adjustments are needed to existing operational distributed information retrieval systems, that use a central server, as the remote search engines and the central server already exchange snippets. Our research implies that the significant overhead incurred by downloading documents in today's prototype distributed information retrieval systems can be completely eliminated. This also enables modeling of servers from which full documents can not be obtained and those which index multimedia content. Furthermore, the central server can continuously use the search result data, the snippets, to keep its resource descriptions up to date without imposing additional overhead, naturally coping with changes in document collections that occur over time. This also provides the extra iterations that snippet query-based sampling requires without extra latency.

Compared to the conventional query-based sampling approach our snippet approach shows equal or better performance per unit of bandwidth consumed for most of the test collections. The performance also appears to be more stable per unit of bandwidth consumed. Factors influencing the performance are document length distribution and the homogeneity of the data. Snippet query-based sampling performs best when document lengths are smoothly distributed, without a large peak at the low-end of document sizes, and when the data is heterogeneous.

Even though the performance of snippet query-based sampling depends on the underlying collection, the information that is used always comes along 'for free' with search results. No extra bandwidth, connections or operations are required beyond simply sending a query and obtaining a list of search results. Herein lies the strength of the approach.

5. FUTURE WORK

We believe that the performance gains seen in the various metrics leads to improved selection and merging performance. However, this is something that could be further explored. A measure for how representative the resource descriptions obtained by sampling are for real-world usage would be very useful. This remains an open problem, also for full document sampling, even though some attempts have been made to solve it [4].

An other research direction is the snippets themselves. Firstly, how snippet generation affects modeling performance. Secondly, how a query can be generated from the snippets seen so far in more sophisticated ways. This could be done by attaching a different priority to different words in a snippet. Finally, the influence of the ratio of snippet to document size could be further investigated.

6. ACKNOWLEDGEMENTS

We thank the USI Lugano Information Retrieval group for their comments, notably Mark Carman and Cyrus Hall. We also thank Dolf Trieschnigg, Kien Tjin-Kam-Jet and Jan Flokstra. This paper, and the experiments, were created using only Free and Open Source Software. Finally, we gratefully acknowledge the support of the Netherlands Organisation for Scientific Research (NWO) under project DIRKA (NWO-Vidi), Number 639.022.809.

7. REFERENCES

[1] Azzopardi, L., de Rijke, M., and Balog, K. Building simulated queries for known-item topics: An analysis using six european languages. In *Proceedings of SIGIR* (New York, NY, US, July 2007), ACM, pp. 455–462.

[2] Bailey, P., Craswell, N., and Hawking, D. Engineering a multi-purpose test collection for web retrieval experiments. *Information Processing & Management 39*, 6 (2003), 853–871.

[3] Baillie, M., Azzopardi, L., and Crestani, F. *Adaptive Query-Based Sampling of Distributed Collections*, vol. 4209 of *Lecture Notes in Computer Science*. Springer, 2006, pp. 316–328.

[4] Baillie, M., Carman, M. J., and Crestani, F. A topic-based measure of resource description quality for distributed information retrieval. In *Proceedings of ECIR* (Apr. 2009), vol. 5478 of *Lecture Notes in Computer Science*, Springer, pp. 485–497.

[5] Bar-Yossef, Z., and Gurevich, M. Random sampling from a search engine's index. *Journal of the ACM 55*, 5 (2008), 1–74.

[6] Callan, J. *Distributed Information Retrieval.* Advances in Information Retrieval. Kluwer Academic Publishers, 2000, ch. 5.

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

[8] Callan, J., Connell, M., and Du, A. Automatic discovery of language models for text databases. In *Proceedings of SIGMOD* (June 1999), ACM Press, pp. 479–490.

[9] Dagan, I., Lee, L., and Pereira, F. Similarity-based methods for word sense disambiguation. In *Proceedings of ACL* (Morristown, NJ, US, Aug. 1997), Association for Computational Linguistics, pp. 56–63.

[10] Eiron, N., and McCurley, K. S. Analysis of anchor text for web search. In *Proceedings of SIGIR* (New York, NY, US, July 2003), ACM, pp. 459–460.

[11] Harman, D. Overview of the first trec conference. In *Proceedings of SIGIR* (New York, NY, US, June 1993), ACM, pp. 36–47.

[12] Harman, D. K. *Overview of the Third Text Retrieval Conference (TREC-3).* National Institute of Standards and Technology, 1995.

[13] Hawking, D., Voorhees, E., Craswell, N., and Bailey, P. Overview of the trec-8 web track. Tech. rep., National Institute of Standards and Technology, Gaithersburg, MD, US, 2000.

[14] Ide, N., and Suderman, K. The open american national corpus, 2007.

[15] Kilgarriff, A., and Rose, T. Measures for corpus similarity and homogeneity. In *Proceedings of EMNLP* (Morristown, NJ, US, June 1998), ACL-SIGDAT, pp. 46–52.

[16] Manning, C. D., Raghavan, P., and Schütze, H. *Introduction to Information Retrieval.* Cambridge University Press, New York, NY, US, 2008.

[17] Manning, C. D., and Schütze, H. *Foundations of Statistical Natural Language Processing.* MIT Press, Cambridge, MA, US, June 1999.

[18] Monroe, G., French, J. C., and Powell, A. L. Obtaining language models of web collections using query-based sampling techniques. In *Proceedings of HICSS* (Washington, DC, US, Jan. 2002), vol. 3, IEEE Computer Society, p. 67.

[19] Paltoglou, G., Salampasis, M., and Satratzemi, M. Hybrid results merging. In *Proceedings of CIKM* (New York, NY, US, Nov. 2007), ACM, pp. 321–330.

[20] Thomas, P., and Hawking, D. Evaluating sampling methods for uncooperative collections. In *Proceedings of SIGIR* (New York, NY, US, July 2007), ACM, pp. 503–510.

[21] Venables, W. N., and Smith, D. M. *An Introduction to R*, Aug. 2009.

RankReduce – Processing K-Nearest Neighbor Queries on Top of MapReduce*

Aleksandar Stupar
Saarland University
Saarbrücken, Germany
astupar@mmci.uni-saarland.de

Sebastian Michel
Saarland University
Saarbrücken, Germany
smichel@mmci.uni-saarland.de

Ralf Schenkel
Saarland University
Saarbrücken, Germany
schenkel@mmci.uni-saarland.de

ABSTRACT

We consider the problem of processing K-Nearest Neighbor (KNN) queries over large datasets where the index is jointly maintained by a set of machines in a computing cluster. The proposed RankReduce approach uses locality sensitive hashing (LSH) together with a MapReduce implementation, which by design is a perfect match as the hashing principle of LSH can be smoothly integrated in the mapping phase of MapReduce. The LSH algorithm assigns similar objects to the same fragments in the distributed file system which enables a effective selection of potential candidate neighbors which get then reduced to the set of K-Nearest Neighbors. We address problems arising due to the different characteristics of MapReduce and LSH to achieve an efficient search process on the one hand and high LSH accuracy on the other hand. We discuss several pitfalls and detailed descriptions on how to circumvent these. We evaluate RankReduce using both synthetic data and a dataset obtained from Flickr.com demonstrating the suitability of the approach.

1. INTRODUCTION

With the success of the Web 2.0 and the wide spread usage of cell phones and digital cameras, millions of pictures are being taken and uploaded to portals like Facebook or Flickr every day[1], accumulating to billions of images[2]. Searching in these huge amounts of images becomes a challenging task. While there is an increasing trend to use social annotations, so-called tags, for image retrieval, next to the traditional image search à la Google/Bing/Yahoo which inspects the text around the web site holding the picture, there is a vital need to process similarity queries, where for a given query picture, the K most similar pictures are returned based on low level features such as color, texture, and shape [9]. The big advantage of such low level features is that they are always available, whereas tags are usually extremely scarce and text around images can often be misleading. Approaches like the work by Taneva et al. [24] use both textual and low level image descriptors to increase the diversity of returned query results. There exist plenty of fundamental prior works [16, 5, 4] on how to index feature based representations of pictures or, more generally, high dimensional vectors in a way that allows for inspecting only a small subset of all vectors to find the most similar ones.

The increasing volume of high dimensional data, however, poses novel problems to traditional indexing mechanisms which usually assume an in-memory index or optimize for local disk access. As a promising approach to process huge amounts of data on a multitude of machines in a cluster, MapReduce [10] has been proposed and continuously explored for many interesting application classes. In this paper, we investigate the usage of MapReduce for searching in high dimensional data. Although MapReduce was initially described in a generic and rather imprecise way in terms of implementation, implementations like Apache's Hadoop have proven to provide salient properties such as scalability, ease of use, and most notably robustness to node failures. This provides an excellent base to explore MapReduce for its suitability for large scale management of high dimensional data.

In this work, we propose RankReduce, an approach to implement locality sensitive hashing (LSH) [1, 8, 16], an established method for similarity search on high dimensional data, on top of the highly reliable and scalable MapReduce infrastructure. While this may seem to be straight forward at first glance, it poses interesting challenges to the integration: most of the time, we face different characteristics of MapReduce and LSH which need to be harnessed both at the same time to achieve both high accuracy and good performance. As MapReduce is usually used only to process large amounts of data in an offline fashion and not for query processing, we carefully investigate its suitability to handle user defined queries effectively demonstrating interesting insights on how to tune LSH on top of MapReduce.

The remainder of the paper is structured as follows. Section 2 gives an overview of related work, Section 3 presents our framework and gives a brief introduction to LSH and MapReduce, Section 4 describes the way queries are processed, Section 5 presents an experimental evaluation, and Section 6 concludes the paper and gives an outlook on ongoing work.

*This work has been supported by the Cluster of Excellence "Multimodal Computing and Interaction" (MMCI)

[1]http://blog.facebook.com/blog.php?post=2406207130

[2]http://blog.flickr.net/en/2009/10/12/4000000000/

2. RELATED WORK

Processing K-Nearest Neighbor queries in high dimensional data has received a lot of attention by researchers in recent years. When the dimensionality increases the distance between the closest and the farthest neighbor decreases rapidly, for most of the datasets [6], which is also known as the 'curse of dimensionality'. This problem has a direct impact on exact KNN queries processing based on tree structures, such as X-Tree [5] and K-D tree [4], rendering these approaches applicable only to a rather small number of dimensions. A better suitable approach for KNN processing in high dimensions is Locality Sensitive Hashing (LSH) [1, 8, 16]. It is based on the application of locality preserving hash functions which map, with high probability, close points from the high dimensional space to the same hash value (i.e., hash bucket). Being an approximate method, the performance of LSH highly depends on accurate parameter tuning [12, 3]. Work has also been done on decreasing the number of hash tables used for LSH, while preserving the precision, by probing multiple buckets per hash table [20]. In recent years, a number of distributed solutions where the main emphasis was put on exploring loosely coupled distributed systems in form of Peer-to-Peer networks (P2P) such as [11, 13, 14, 17, 23]) have been proposed, cf. the work by Batko et al. [2] for a discussion on the suitablity of different P2P approaches to distributed similarity search.

MapReduce is a framework for efficient and fault tolerant workload distribution in large clusters [10]. The motivation behind the design and development of MapReduce has been found in Information Retrieval with its many computationally expensive, but embarrassingly parallel problems on large datasets. One of the most basic of those problems is the inverted index construction, described in [21]. MapReduce has not yet been utilized for distributed processing of KNN queries. Some similarities with KNN processing can be found in recent work by Rares et al. [22] which describes a couple of approaches for computing set similarities on textual documents, but it does not address the issue of KNN query processing. The pairwise similarity is calculated only for documents with the same prefixes (prefix filtering), which can be considered as the LSH min-hashing technique. Lin [19] describes a MapReduce based implementation of pairwise similarity comparisons of text documents based on an inverted index.

3. RANKREDUCE FRAMEWORK

We address the problem of processing K-Nearest Neighbor queries in large datasets by implementing a distributed LSH based index within the MapReduce Framework.

An LSH based index uses *locality sensitive hash* functions for indexing data. The salient property of these functions is that they map, with high probability, similar objects (represented in the d-dimensional vector space) to the same hash bucket, i.e., related objects are more probable to have the same hash value than distant ones. The actual indexing builds several hash tables with different LSH functions to increase the probability of collision for close points. At query time, the KNN search is performed by hashing the query point to one bucket per hash table and then to rank all discovered objects in any of these buckets by their distance to the query point. The closest K points are returned as the final result.

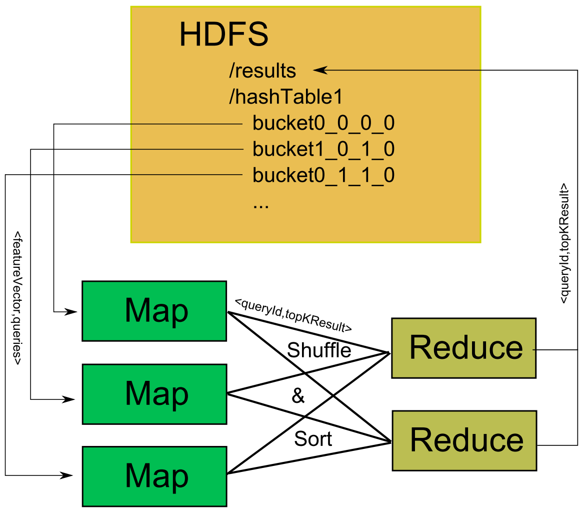


Figure 1: The RankReduce Framework

In this work, we consider the family of LSH functions based on p-stable distributions [8] which are most suitable for l_p norms. In this case, for each data point $\mathbf{v}$, the hashing scheme considers k independent hash functions of the form

$$h_{\mathbf{a},B}(\mathbf{v}) = \lfloor \frac{\mathbf{a} \cdot \mathbf{v} + B}{W} \rfloor \qquad (1)$$

where $\mathbf{a}$ is a d-dimensional vector whose elements are chosen independently from a p-stable distribution, $W \in \mathbb{R}$, and B is chosen uniformly from $[0, W]$. Each hash function maps a d-dimensional data point onto the set of integers. With k such hash functions, the final result is a vector of length k of the form $g(\mathbf{v}) = (h_{\mathbf{a_1},B_1}(\mathbf{v}), ..., h_{\mathbf{a_k},B_k}(\mathbf{v}))$.

In order to achieve high search accuracy, multiple hash tables need to be constructed. The work by Lv et al. [20] presents an approach to probe multiple buckets per hash table which, however, leads to either sampling a larger fraction of the dataset or to many fine grained accesses to small buckets. The latter causes a larger number of expensive random accesses to the underlying infrastructure, as we deal with file based indexes as opposed to in-memory accesses. Hence, we opted for using a single probe per hash table.

For maintaining the hash tables over a set of machines in a cluster, we employ MapReduce [10] which is designed to be used for large data processing in parallel. It is built on top of the Distributed File System [15], which enables distributing the data over the cluster machines in a scalable and fault tolerant way. This tight integration of MapReduce with the distributed file system enables it to move calculations where the data resides, eliminating network bandwidth bottlenecks caused by data shipping during query processing. Our implementation uses the open source software Hadoop [3], maintained by the Apache Foundation, which provides a Java based implementation of both the MapReduce framework and the Distributed File System (coined HDFS for Hadoop Distributed File System). In the last years, Hadoop gained a lot of popularity in the open source community and is also part of many research efforts investigating large data processing.

[3]http://hadoop.apache.org/

MapReduce is a fairly simple programming model, based on two developer supplied functions: *Map* and *Reduce*. Both functions are based on key-value pairs. The Map function receives a key-value pair as input and emits multiple (or none) key-value pairs as output. The output from all Map functions is grouped by key, and for each such key, all values are fed to the Reduce function, which then produces the final output from these values.

In the Hadoop implementation, the input data is grouped in so-called *input splits* (which often correspond to blocks in the distributed file system), and a number of so-called *mapper* processes call the Map function for each key-value pair in such an input split. A number of mappers can run concurrently on each node in the cluster, and the mapper processes are in addition distributed over all nodes in the cluster. Ideally, a mapper is run on the same node where the input block resides, but this is not always possible due to workload imbalance. Similarly, after all mappers have finished, dedicated *reducer* processes are run on nodes int the cluster. Each reducer handles a fraction of the output key space, copies those key-value pairs from all mappers' outputs (in the so-called *shuffle phase*), sorts them by key, and feeds the to the Reduce function. The output of the reducers is usually considered the final result but can also be used as input for following MapReduce jobs.

Figure 1 shows an illustration of our LSH integration with MapReduce. Each hash table in the LSH index is mapped to one folder in HDFS. For each bucket in such a hash table, a corresponding file is created in this folder, where the file name is created by concatenating hash values into a string, with '_' as separator. This mapping of buckets to HDFS files enables fast lookup at query time and ensures that only data that is to be probed is read from the HDFS. Placing the bucket in one file also enables block based sequential access to all vectors in one bucket, which is very important as the MapReduce framework is optimized for such block based rather than random access processing. Each of the buckets stores the complete feature vectors of all objects mapped to this bucket in a binary encoding.

Indexing of new feature vectors to the LSH index in HDFS is easily done by appending them to the end of the appropriate bucket file. This can also be done in parallel with query processing as long as different buckets are affected; as HDFS does not include a transaction mechanism, appending entries to buckets that are being queried would be possible, but with unclear semantics for running queries. As HDFS scales well with increasing cluster size, the resulting growth of the LSH index can easily be supported by adding more machines to the cluster.

While an LSH index stored in-memory has no limitation on the number of buckets, too many files in HDFS can downgrade its performance, especially if these files are much smaller than the block size (which defaults to 64MB). The number of buckets, and therefore the number of files in HDFS for the LSH index, is highly dependent on the set up of LSH parameters as choosing a bad combination of parameters can result in a large number of small files.

Inspired by in-memory indexes which can have references from buckets to materialized feature vectors, we considered storing only feature vector ids in the buckets instead of the actual feature vectors, and retrieving the full vectors only on demand at query time. However, this approach would result in poor performance due to many random accesses to the HDFS when retrieving the full vectors, so we decided to store complete feature vectors. This fact also needs to be addressed when setting up LSH parameters, while too many LSH hash tables can dramatically increase index size, as each feature vector is materialized for each hash table.

4. QUERY PROCESSING

We implemented KNN query processing as a MapReduce job. Before starting this MapReduce job, the hash values for the query documents are calculated. These values are then used for selecting the buckets from the LSH index, which are to be probed. The selected buckets are provided as input to the query processing MapReduce job, generating multiple input splits. The generated input splits are read by a custom implementation of the *InputFormat* class, which reads feature vectors stored in a binary format and provides them as the key part of the Map function input. Queries are being distributed to mappers either by putting them in the *Distributed Cache* or by putting them in HDFS file with high number of replicas. They are read once by the InputFormat implementation and reused as value part of the Map function input between the function invocations.

The input to the Map function consists therefore of the feature vector to be probed as the key and the list of queries as the value. The Map function computes the similarity of the feature vector with all query vectors. While a standard MapReduce implementation would now emit a result pair for each combination of feature vector and query vector, we employ an optimization that delays emitting results until all feature vectors in the input split have been processed. We then eventually emit the final K-Nearest Neighbor for each query vector from this input split in the form of key-value pairs. Here, the query is the key and a nearest neighbor together with its distance to the query vector is the value. To implement this delayed emitting, we store the currently best K-Nearest Neighbor for each query in-memory, together with their distances from the query points. The results are emitted at the end of processing the input split in Hadoop's cleanup method[4]. The Reduce method then reads, for each query, the K-Nearest Neighbor from each mapper, sorts them by increasing distance, and emits the best K of them as the final result for this query.

The final sort in the reducer can even be executed within Hadoop instead of inside the Reduce method, as a subtask of sorting keys in the reducer. It is possible to apply a so-called *Secondary Sort* that allows, in our application, to sort not just the keys, but also the values for the same key. Technically, this is implemented by replacing, for each (query, (neighbor, distance)) tuple that is emitted by a mapper, the key by a combined key consisting of the query and the distance. Keys are then sorted lexicographically first by query and then by distance. For assigning tuples to a Reduce method, however, only the query part of the key is taken into account. The reducer then only needs to read the first K values for each key, which then correspond to the K-Nearest Neighbor for that query.

It is worth mentioning that because one feature vector is placed in multiple hash tables, the same vector can be evaluated twice for the same query during processing. An

[4]This feature was introduced in the most recent version 0.20; before, it was only possible to emit directly from the Map function

alternative approach would be to have two MapReduce jobs for query processing instead of one, which would eliminate this kind of redundancy. The first MapReduce job would create a union between buckets that need to be probed, and the second job would use the union as an input to similarity search. However, while this would possibly save redundant computations, it has the major drawback that the results from the first job need to be written to the HDFS before starting the second job. As the overhead from multiple evaluations of the same feature vector has not been too large in our experimental evaluation (see Figure 4), we decided that it is better to probe slightly more data rather than to pay the additional I/O cost incurred by using two Map Reduce jobs.

The approach can handle multiple queries at the same time in one MapReduce job. But it is not suitable for the cases when the number of queries becomes too large, as problem of KNN queries processing becomes the problem of set similarity joins [22].

5. EXPERIMENTAL EVALUATION

For our experiments we have used Hadoop version 0.20.2 installed on three virtual machines with Debian GNU/Linux 5.0 (Kernel version: 2.6.30.10.1) as operating system. Each of the virtual machines has been configured to have 200GB hard drive, 5 GB main memory and two processors. VMware Server version 2.0.2 was used for virtualization of all machines. The virtual machines were run on a single machine with Intel Xeon CPU E5530 @2.4 GHz, 48 GB main memory, 4 TB of hard drive and Microsoft Windows Server 2008 R2 x64 as operating system. We used a single machine Hadoop installation on these virtual machines as described later on.

Datasets

As the performance of the LSH based index is highly dependent on the data characteristics [12], we conducted an experimental evaluation both on randomly generated (Synthetic Dataset) and real world image data (Flickr Dataset):

Synthetic Dataset:
For the synthetic dataset we used 32-dimensional randomly generated vectors. The synthetic dataset was built by first creating N independently generated vector instances drawn from the normal distribution $N(0, 1)$ (independently for each dimension). Subsequently, we created m near duplicates for each of the N vectors, leading to an overall dataset size of $m*N$ vectors. The rational behind using the near duplicates is that we make sure that the KNN retrieval is meaninful at all. We set m to 10 in the experiments and adapt N to the desired dataset size depending on the experiment. We generated 50 queries by using the same procedure as the original vectors were generated.

Flickr Dataset:
We used the 64-dimensional color structure feature vectors from crawled Flickr images provided by the CoPhIR data collection [7] as our real image dataset. We extracted the color structure feature vectors from the available MPEG-7 features and stored them in a binary format suitable for the experiments. As the queries, we have randomly selected 50 images from the rest of the CoPhIR data collection

As LSH is an approximate method, we measure the effectiveness of the nearest neighbor search by its *precision*, which is the relative overlap of the true K-Nearest Neighbor with the K-Nearest Neighbor computed by our method. And for the proximity measure we used Euclidean distance.

5.1 LSH Setup

Before starting the evaluation we needed to understand how to set up LSH and what consequence it may have on the index size and query performance. In our setup we consider the number of hash tables and the bucket size as LSH parameters to be tuned. The bucket size can be changed either by changing the number of concatenated hash values or by changing the parameter W in Formula 1. Because W is a continuous variable and provides a subtle control over bucket size, we first fix the number of concatenated hash values and then vary parameter W [12]. These two parameters together determine which subset of the data needs to be accessed to answer a query (one bucket per hash table). We varied the bucket size by varying parameter W for a different number of hash tables and then measured data subset probed and precision, shown in Figure 2 for synthetic dataset and in Figure 3 for the Flickr dataset. These measurements were done using 50 KNN queries for $k = 20$ on both datasets, but with reduced sizes to 100,000 feature vectors indexed. The results show that increasing the number of hash tables can decrease the data subset that needs to be probed to achieve a certain precision, resulting in less time needed for the query execution.

Realizing that each new table creates another copy of data and we may have only limited storage available, we need to tradeoff storage cost vs. execution time. Additionally, when only a fixed subset of the data should be accessed, a larger number of hash tables results in a large number of small sized buckets, which is not a good scenario for HDFS (it puts additional pressure on Hadoop's data node that manages all files). On one hand, we would like to increase the number of hash tables and to decrease the probed data subset. On the other hand, we would like to use less storage space and a smaller number of files for storage and probing. Figure 3 shows that the number of hash tables has smaller impact on precision in case of real image data. Thus, as a general rule we suggest a smaller number of hash tables with larger bucket sizes, still set to satisfy the precision threshold. Therefore we settle for a setup of four hash tables and a bucket size that allow us to get at least 70% precision.

5.2 Evaluation

We evaluate our approach and compare it to the linear scan over all data, also implemented as a MapReduce job. As we did not have a real compute cluster at hand for running the experiments, we simulate the execution in a large cluster by running the mappers and reducers sequentially on our small machine. We measure run times of their executions and the number of mappers started for each query job. To avoid the possible bottleneck of a shared hard drive between virtual machines [18], we run each experiment on a single machine Hadoop installation with one map task allowed per task tracker. This results in sequential execution of map tasks so there is no concurrent access to a shared hard drive by multiple virtual machines.

Considering that the workload for the reducers is really small for both linear scan and LSH, we only evaluate map execution times and the number of mappers run per query job. We measured the map execution times for all jobs and found that they are approximately constant, with average value per mapper being 3.256 seconds and standard devia-

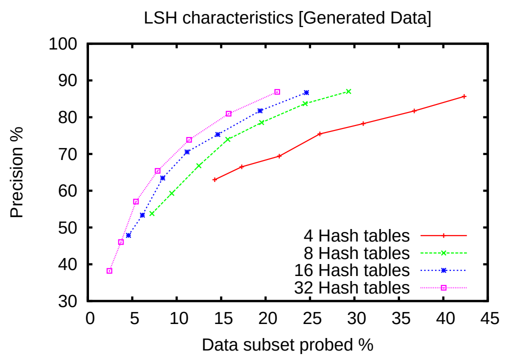


Figure 2: LSH characteristics on generated data.

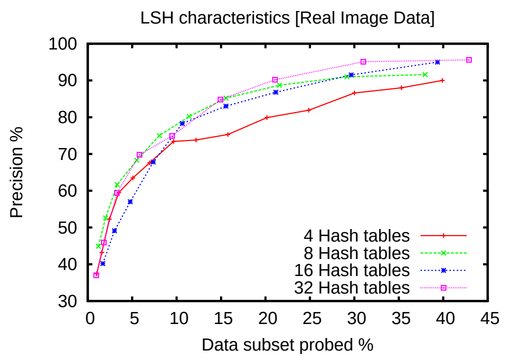


Figure 3: LSH characteristics on picture data.

tion of 1.702 seconds. Taking into account that each mapper has an approximately same data input size, defined by the HDFS' block size, approximately constant mapper execution time is well expected.

Measures of Interest

Because the execution time of the mappers is almost constant, the load of a query execution can be represented as number of mappers per query. We measured the number of mappers per query and precision for 50 KNN queries, with K=20, for both datasets, with 2GB, 4GB, and 8GB of indexed data (∼4000, ∼8000, and ∼16000 feature vectors for the real image dataset and ∼8000, ∼16000, and ∼32000 feature vectors for the synthetic dataset, respectively). The number of mappers per query for synthetic dataset is shown in Figure 5. And as we can see the number of mappers is about 3 times smaller for LSH than for linear scan. Also we can see in Figure 6, which shows the number of mappers per query for the Flickr dataset, that the difference in the number of mappers between LSH and linear scan is even bigger. The number of mappers per query is 4 to 5 times smaller for LSH than for linear scan in this case. The precision, shown in Figure 7, for generated data is over the threshold of 70% for 2GB and 4GB of indexed data, but drops down to 63.8% for 8GB. For real image data, the precision is almost constant, varying slightly around 86%.

6. CONCLUSION

In this work we described RankReduce, an approach for processing large amounts of data for K-Nearest Neighbor (KNN) queries. Instead of dealing with standard issues in distributed systems such as scalability and fault tolerance, we implement our solution with MapReduce, which provides these salient properties out of the box. The key idea of the presented approach is to use Locality Sensitive Hashing (LSH) in the Map phase of MapReduce to assign similar objects to the same files in the underlying distributed file system. While this seemed to be straight forward at first glance, there was a nontrivial conflict of opposing criteria and constraints caused by LSH and MapReduce which we had to solve to achieve accurate results with an acceptable query response time. We have demonstrated the suitability of our approach using both a synthetic and a real world dataset.

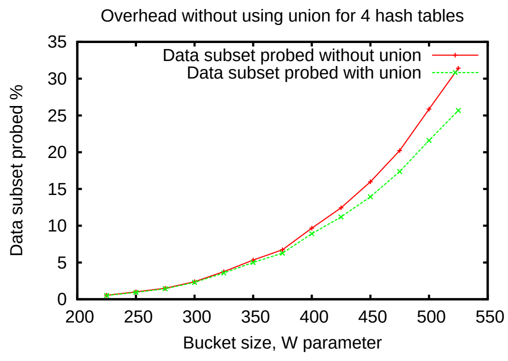


Figure 4: Overhead without using union.

Our presented approach on large scale data processing is, however, not limited to KNN search over images, but can be extended to a variety of other interesting applications, such as near duplicate detection, document classification, or document clustering.

As a first step in our future work plan to evaluate our approach on a real compute cluster, which we are currently building up, with large scale data in the order of several TB. We furthermore plan to extend our approach to video and music retrieval.

7. REFERENCES

[1] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *FOCS*, 2006.

[2] Michal Batko, David Novak, Fabrizio Falchi, and Pavel Zezula. Scalability comparison of peer-to-peer similarity search structures. *Future Generation Comp. Syst.*, 2008.

[3] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. Lsh forest: self-tuning indexes for similarity search. In *WWW*, 2005.

[4] Jon Louis Bentley. K-d trees for semidynamic point sets. In *Symposium on Computational Geometry*, 1990.

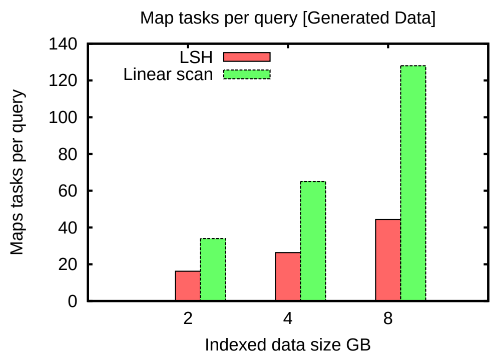


Figure 5: Map tasks per query on generated data.

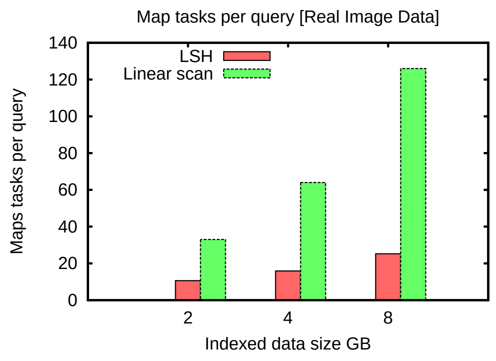


Figure 6: Map tasks per query on picture data.

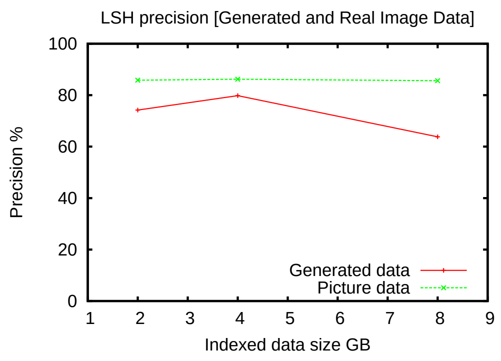


Figure 7: LSH precision on generated and picture data.

[5] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The x-tree : An index structure for high-dimensional data. In *VLDB*, 1996.

[6] Kevin S. Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *ICDT*, 1999.

[7] Paolo Bolettieri, Andrea Esuli, Fabrizio Falchi, Claudio Lucchese, Raffaele Perego, Tommaso Piccioli, and Fausto Rabitti. CoPhIR: a test collection for content-based image retrieval. *CoRR*, 2009.

[8] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Symposium on Computational Geometry*, 2004.

[9] Ritendra Datta, Dhiraj Joshi, Jia Li, and James Ze Wang. Image retrieval: Ideas, influences, and trends of the new age. *ACM Comput. Surv.*, 2008.

[10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[11] Vlastislav Dohnal and Pavel Zezula. Similarity searching in structured and unstructured p2p networks. In *QSHINE*, 2009.

[12] Wei Dong, Zhe Wang, William Josephson, Moses Charikar, and Kai Li. Modeling lsh for performance tuning. In *CIKM*, 2008.

[13] Christos Doulkeridis, Kjetil Nørvåg, and Michalis Vazirgiannis. Peer-to-peer similarity search over widely distributed document collections. In *LSDS-IR*, 2008.

[14] Fabrizio Falchi, Claudio Gennaro, and Pavel Zezula. A content-addressable network for similarity search in metric spaces. In *DBISP2P*, 2005.

[15] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 2003.

[16] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *VLDB*, 1999.

[17] Parisa Haghani, Sebastian Michel, and Karl Aberer. Distributed similarity search in high dimensions using locality sensitive hashing. In *EDBT*, 2009.

[18] Shadi Ibrahim, Hai Jin, Lu Lu, Li Qi, Song Wu, and Xuanhua Shi. Evaluating mapreduce on virtual machines: The hadoop case. In *CloudCom*, 2009.

[19] Jimmy J. Lin. Brute force and indexed approaches to pairwise document similarity comparisons with mapreduce. In *SIGIR*, 2009.

[20] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe lsh: Efficient indexing for high-dimensional similarity search. In *VLDB*, 2007.

[21] Richard M. C. McCreadie, Craig Macdonald, and Iadh Ounis. On single-pass indexing with mapreduce. In *SIGIR*, 2009.

[22] Vernica Rares, Carey Michael J., and Li Chen. Efficient parallel set-similarity joins using mapreduce. 2010.

[23] Ozgur D. Sahin, Fatih Emekçi, Divyakant Agrawal, and Amr El Abbadi. Content-based similarity search over peer-to-peer systems. In *DBISP2P*, 2004.

[24] Bilyana Taneva, Mouna Kacimi, and Gerhard Weikum. Gathering and ranking photos of named entities with high precision, high recall, and diversity. In *WSDM*, 2010.

Topic-based Index Partitions for Efficient and Effective Selective Search

Anagha Kulkarni and Jamie Callan
Language Technologies Institute
School of Computer Science
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh, PA 15213
anaghak, callan@cs.cmu.edu

ABSTRACT

Indexes for large collections are often divided into *shards* that are distributed across multiple computers and searched in parallel to provide rapid interactive search. Typically, all index shards are searched for each query. This paper investigates document allocation policies that permit searching only a few shards for each query (*selective search*) without sacrificing search quality. Three types of allocation policies (random, source-based and topic-based) are studied. K-means clustering is used to create topic-based shards. We manage the computational cost of applying these techniques to large datasets by defining topics on a subset of the collection. Experiments with three large collections demonstrate that selective search using topic-based shards reduces search costs by at least an order of magnitude without reducing search accuracy.

Categories and Subject Descriptors

H.3 [**INFORMATION STORAGE AND RETRIEVAL**]: Information Search and Retrieval

Keywords

selective searching, federated search, document clustering

1. INTRODUCTION

Traditionally, searching a collection of documents was a serial task accomplished using a single central index. However, as the document collections increased in size, it became necessary and a common practice to partition collections into multiple disjoint indexes (*shards*) [2, 1]. These distributed indexes facilitate parallelization of search which in turn brings down the query processing time. However, even in this architecture the cost associated with searching large-scale collections is high. For organizations with modest resources this becomes a challenge and potentially limits the scale of the collections that they can experiment with.

Our goal is to organize large collections into shards such that the shards facilitate a search setup where only a subset of the shards are searched for any query (*selective search*) and yet provide a performance that is at par with that provided by exhaustive search. The amount of work required per query is naturally much lower in the selective search setup and thus it does not necessitate availability of large computing clusters to work with large document collections.

We experiment with three document allocation policies random, source-based and topic-based to partition the collections into shards (Section 3). One of the main challenges that we tackle in this work is to scale the organization policies to be able to process large collections. Some of the above policies are naturally efficient but for others we propose an approximation technique that is efficient and can parallelize the partitioning process. We also establish that the approximation would not lead to any significant loss in effectiveness. The other contribution of this work is the introduction of a simple yet more accurate metric for measuring the search cost incurred for each query (Section 6.2).

2. RELATED WORK

There have been few other studies that have looked at partitioning of collections into shards. Xu and Croft [16] used a two-pass K-means clustering algorithm and a KL-divergence distance metric to organize a collection into 100 topical clusters. They also experiment with source-based organization and demonstrate that selective search performed as well as exhaustive search, and much better than a source-based organization. The datasets used in this work are small and thus it not clear whether the document organization algorithms employed in this work would scale and be effective for large-scale datasets such as the ones used in our work. Secondly, it has been a common practice in previous work to compute search cost by looking at the number of shards searched for a query which is what is used by Xu and Croft. However, in most setups the shards are of non-uniform sizes and thus this formulation of search cost does not enable an accurate analysis of the trade-off between search cost and accuracy. We remedy this by factoring in the individual shard sizes into the search cost formulation.

Larkey et al. [7] studied selective search on a dataset composed of over a million US Patents documents. The dataset was divided into 401 topical units using manually assigned patent categories, and into 401 chronological units using dates. Selective search was more effective using the topical organization than the chronological organization.

Puppin et al. [12] used query logs to organize a document collection into multiple shards. The query log covered a period of time when exhaustive search was used for each query. These training queries and the documents that they retrieved were co-clustered to generate a set of *query clusters* and a set of corresponding 16 *document clusters*. Documents that could not be clustered because they were not retrieved by any query (50% of the dataset) were put in a 17th (fall-back) cluster. Selective search using shards defined by these clusters was found to be more effective than selective search using shards that were defined randomly. The number of shards is relatively very small for a large dataset and the distribution of documents across the shards using this approach is skewed. The inability of the algorithm to partition documents that have not appeared in the query log make this technique's performance highly dependent on the query-log used for the partitioning.

Once the collection has been organized into shards, deciding which index shards to search from the given set of shards is a type of *resource selection* problem [3]. In prior research [4, 14, 13] , the resources were usually independent search engines that might be uncooperative. Selectively searching the shards of a large index is however an especially *cooperative* federated search problem where the federated system can define the resources (shards) and expect complete support from them.

3. DOCUMENT ALLOCATION POLICIES

Our goal is to investigate document allocation policies that are effective, scalable, and applicable in both research and commercial environments. Although we recognize the considerable value of query logs and well-defined categories, they are not available in all environments, thus our research assume access only to the document contents to develop the allocation techniques. This work studies random, source-based, and topic-based allocation policies.

3.1 Random Document Allocation

The random allocation policy assigns each document to one of the shards at random with equal probability. One might not expect a random policy to be effective, but it was a baseline in prior research [12]. Our experimental results show that for some of the datasets random allocation is more effective than one might expect.

3.2 Source-based Document Allocation

Our datasets are all from the Web. The source-based policy uses document URLs to define shards. The document collection is sorted based on document URLs, which arranges documents from the same website consecutively. Groups of M/K consecutive documents are assigned to each shard (M: total number of documents in the collection, K: number of shards). Source-based allocation was used as a baseline in prior research [16].

3.3 Topic-based Document Allocation

The *Cluster Hypothesis* states that *closely associated documents tend to be relevant to the same request* [15]. Thus if the collection is organized such that each shard contains a similar set of documents, then it is likely that the relevant documents for any given query will be concentrated in just a few shards. Cluster-based and category-based document allocation policies were effective in prior research [16, 12, 7].

We adapt K-means clustering [8] such that it would scale to large collections and thus provide an efficient approach to topical sharding of datasets.

Typically, a clustering algorithm is applied to the entire dataset in order to generate clusters. Although the computational complexity of the K-means algorithm [8] is only linear in the number of documents (M), applying this algorithm to large collections is still computationally expensive. Thus, we sample a subset (S) of documents from the collection ($|S| << |M|$), using uniform sampling without replacement. The standard K-means clustering algorithm is applied to S and a set of K clusters is generated. The remaining documents in the collection ($M - S$) are then projected onto the space defined by the K clusters. Note that the process of assigning the remaining documents in the collection to the clusters is parallelizable. Using this methodology large collections can be efficiently partitioned into shards.

We use the negative Kullback-Liebler divergence (Equation 1) to compute the similarity between the unigram language model of a document D ($p_d(w)$), and that of a cluster centroid C^i ($p_c^i(w)$). (Please refer to [11] for the derivation.) Using maximum likelihood estimation (MLE), the cluster centroid language model computes to, $p_c^i(w) = c(w, C^i)/\sum_{w'} c(w', C^i)$ where $c(w, C^i)$ is the occurrence count of w in C^i. Following Zhai and Lafferty [17], we estimate $p_d(w)$ using MLE with Jelinek-Mercer smoothing which gives $p_d(w) = (1 - \lambda)\ c(w, D)/\sum_{w'} c(w', D) + \lambda\ p_B(w)$. The term $p_B(w)$ is the probability of the term w in the background model. The background model is an average of the K centroid models. Note that the background model plays the role of inverse document frequency for the term w.

$$KL(C^i||D) = \sum_{w \in C^i \bigcap D} p_c^i(w)\ log \frac{p_d(w)}{\lambda\ p_B(w)} \qquad (1)$$

We found this version of KL-divergence to be more effective than the variant used by Xu and Croft [16].

4. SHARD SELECTION

After index shards are defined, a resource selection algorithm is used to determine which shards to search for each query. Our research used ReDDE [14], a widely used algorithm that prioritizes shards by estimating a query specific distribution of relevant documents across shards. To this end, a *centralized sample index*, CS, is created, one that combines samples from every shard R. For each query, a retrieval from the central sample index is performed and the top N documents are assumed to be relevant. If n_R is the number of documents in N that are mapped to shard R then a score s_R for each R is computed as $s_R = n_R * w_R$, where the shard weight w_R is the ratio of size of the shard $|R|$ and the size of its sample. The shard scores s_R are then normalized to obtain a valid probability distribution which is used to rank the shards. In this work, we used a variation of ReDDE, which produced better results in preliminary experiments. Rather than weight each retrieved sampled document equally, we use the document score assigned by the retrieval algorithm to weight the document.

Selective search of index shards is a cooperative environment where global statistics of each shard are readily available. Thus merging the document rankings generated by searching the top ranked shards is straightforward.

Table 1: Datasets and Query Sets

Dataset	Number of Documents	Number of Words (billion)	Vocabulary Size (million)	Avg Doc Len	Query Set	Avg Qry Len	Avg Number of Rel Docs Per Qry
Gov2	25,205,179	23.9	39.2	949	701-850	3.1	179 (+/- 149)
Clue-CatB	50,220,423	46.1	96.1	918	TREC09:1-50	2.1	80 (+/- 49)
Clue-CatA-Eng	503,903,810	381.3	1,226.3	757	TREC09:1-50	2.1	114 (+/- 64)

5. DATASETS

Three large datasets were used in this work: Gov2, the CategoryB portion of ClueWeb09 (Clue-CatB) and the English portion of ClueWeb09 (Clue-CatA-Eng). The summary statistics of these datasets are given in Table 1.

The Gov2 TREC corpus [5] consists of 25 million documents from the US government domains, such as .gov and .us, and also from government related websites, such as, www.ncgov.com and www.youroklahoma.com [1]. TREC topics 701-850 were used for evaluation with this dataset. The statistics for these queries are provided in the Table 1.

The ClueWeb09 is a newer dataset that consists of 1 billion web pages that were crawled between January and February 2009. Out of the 10 languages present in the dataset we use the English portion in this work. The Clue-CatB dataset consists of the first 50 million English pages and the Clue-CatA-Eng consists of all the English pages in the dataset (over 500 million). For evaluation with both Clue-CatB and Clue-CatA-Eng datasets we use the 50 queries that were used in the Web track at TREC 2009.

6. EXPERIMENTAL METHODOLOGY

The three datasets were converted to Indri[2] indexes after stoplisting and stemming with the Krovetz stemmer.

6.1 Sample size and OOV terms

Using a subset instead of the entire collection to learn the clusters reduces the computational cost however it also introduces the issue of out-of-vocabulary (OOV) terms during inference. Depending upon the size of the subset (S) that was used for learning, the remaining documents in the collection are bound to contain terms that were not observed in S and thus are absent from the learned clusters or topic models. In such a situation, inference must proceed using the seen terms and ignore the OOV terms. However, the inference quality can potentially degrade because of the discounting of the OOV terms. It is important to select a sample size that leads to a small percentage of OOV terms per document.

Figure 1 (x-axis in log domain) demonstrates that the average percentage of OOV terms per document is low even for small sample sizes. Note that the drop in the average values isn't linear in the sample size; as more documents are seen, the percentage of unseen terms does not decrease proportionately. Heaps' law [6] offers an explanation for this trend – when examining a corpus, the rate at which vocabulary is discovered tapers off as the examination continues. Thus after a certain point increasing the sample size has little effect on the percentage of OOV terms per document.

We leverage these observations to make our experimental

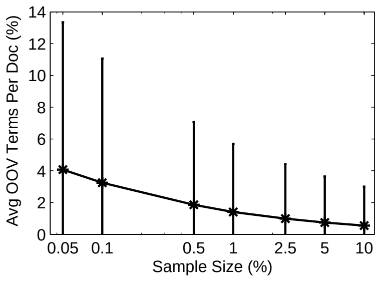


Figure 1: Sample size vs. percentage of OOV terms per document, on average, for the Clue-CatB Dataset.

methodology efficient. For Gov2 and Clue-CatB datasets we sample 0.1% (25K and 50K documents) and for Clue-CatA-Eng dataset we sample 0.01% (50K documents) of the entire collection using uniform sampling. These samples are used by K-means for cluster learning.

6.2 Setup

The Gov2 and Clue-CatB datasets were each partitioned into 100 shards while the Clue-CatA-Eng dataset was organized into 500 shards using each of the document allocation techniques. The top 10 terms for nine of the 100 topical shards of the Clue-CatB dataset are given in Table 2. These are the terms that explain the majority of the probability mass in the language models for each of these topical clusters. For these nine shards and for most of the other 91 shards a semantically coherent topic emerges from these terms.

A language modeling and inference network based retrieval model, Indri [9], was used for our experiments. Modeling dependencies among the query terms has been shown to improve adhoc retrieval performance [10]. We investigate if this holds for selective search as well. Thus document retrieval was performed using the simple bag-of-words query representation and also with the full-dependence model query representation. The Indri query language, which supports structured queries, was used for the dependence model queries. For each query the set of shards was ranked using the variant of ReDDE algorithm described in Section 4 and the top T shards were searched to generate the merged ranked list of documents.

The precision at rank 10 metric (P10) was used to compare the search accuracy of exhaustive search with that of selective search. We define the search cost of a query to be the percentage of documents that were searched. For exhaustive search the cost is 100% while for selective search the cost depends on the number of shards that were searched and the fraction of documents that were present in these shards.

[1]http://www.mccurley.org/trec/
[2]http://www.lemurproject.org/indri/

Table 2: Top terms from topical shards of the Clue-CatB dataset.

Topic A	Topic B	Topic C	Topic D	Topic E	Topic F	Topic G	Topic H	Topic I
state	policy	recipe	music	game	law	entertain	price	health
politics	privacy	food	record	play	patent	com	accessory	care
election	information	cook	song	casino	attorney	news	com	center
party	terms	com	album	free	com	sports	size	service
war	service	home	wikipedia	online	legal	advertise	product	school
america	site	new	edit	com	lawyer	home	clothing	child
government	rights	make	rock	puzzle	www	blog	item	home
vote	copyright	make	com	download	california	list	ship	program
new	return	oil	band	poker	home	business	home	educate
president	com	cup	video	arcade	case	search	costume	parent

Table 3: P10 values for selective search on Gov2 with bag-of-words query. ▼ denotes significantly worse P10 than exhaustive search ($p < 0.05$).
Exhaustive search: P10=0.530, Cost=100%

	Rand	Source	K-means
1 Shard	▼0.169	▼0.236	0.491
Cost (%)	1.00	1.00	1.24
3 Shards	▼0.302	▼0.419	0.511
Cost (%)	3.00	3.00	3.62
5 Shards	▼0.338	▼0.456	0.520
Cost (%)	5.00	5.00	6.00
10 Shards	▼0.384	▼0.492	0.533
Cost (%)	10.00	10.00	11.38
15 Shards	▼0.411	0.507	0.530
Cost (%)	15.00	15.00	15.40

Table 4: P10 values for selective search on Gov2 with dependence model query. ▼ denotes significantly worse P10 than exhaustive search ($p < 0.05$).
Exhaustive search: P10=0.580, Cost=100%

	Rand	Source	K-means
1 Shard	▼0.165	▼0.255	▼0.504
Cost(%)	1.00	1.00	1.26
3 Shards	▼0.304	▼0.443	▼0.552
Cost (%)	3.00	3.00	3.62
5 Shards	▼0.357	▼0.491	0.575
Cost (%)	5.00	5.00	6.00
10 Shards	▼0.419	0.556	0.583
Cost (%)	10.00	10.00	11.38
15 Shards	▼0.442	0.560	0.584
Cost (%)	15.00	15.00	15.63

7. RESULTS AND DISCUSSION

The selective search results for the Gov2 dataset with bag-of-words query representation are provided in Table 3.

Selective search on shards defined by K-means provides search accuracy that is statistically indistinguishable from that of exhaustive search when the search cost is 1.24% of that of exhaustive search. For source-based shards the top 15 shards have to be searched to obtain comparable search accuracy, however, even this leads to an order of magnitude reduction in search cost.

Recall that the samples that were used to define the K-means clusters were quite small, 0.1% and 0.01% of the collection. These results show that an exact clustering solution that uses the entire collection is not necessary for selective search to perform at par with the exhaustive search. An efficient approximation to topic-based techniques can partition large collection effectively and facilitate selective search.

Table 4 provides selective search results for the Gov2 dataset with dependence model queries. As observed by Metzler and Croft in [10], the dependence model queries lead to better search performance than bag-of-words queries – an improvement of 10% is obtained for exhaustive search and for many of the selective search settings as well. Selective search proves to be as capable as exhaustive search in leveraging the information about query term dependence. The trends observed in Table 4 are similar to those observed in Table 3 – topic-based shards provide the cheapest setup for obtaining selective search accuracies that are comparable to those of exhaustive search. However, the absolute search cost for the selective search to be statistically indistinguishable from exhaustive search goes up from 1.24% (bag-of-words) to 6%. Nevertheless, the search cost (6%) is still an order of magnitude smaller than the cost for exhaustive search. In the interest of space we report only dependence model results henceforth, due to their higher accuracy.

Results for the Clue-CatB dataset and the Clue-CatA-Eng datasets are provided in Tables 5 and 6. The topic-based technique perform as well as the exhaustive search by searching only the top ranked shard which is less than 2% and 0.5% of the documents for Clue-CatB and Clue-CatA-Eng, respectively. Searching the top 3 shards provides nearly 10% and 30% improvement over exhaustive search for Clue-CatB and Clue-CatA-Eng, respectively, and the latter is found to be statistically significant. To the best of our knowledge these results provide an evidence for the first time that selective search can consistently improve over exhaustive search while searching a small fraction of the collection.

For both the datasets, selective search loses this advantage over the exhaustive search by searching more shards. This indicates that a smaller but tightly focused search space can be better than a larger search space. This also implies that searching a fixed number of shards for each query might not be ideal. This is an interesting topic for future research in selective search. The source-based shards continue to provide a competitive baseline for both the datasets.

A query-level analysis of the effectiveness of different methods at minimizing the number of queries harmed by selective search revealed that 86% or more queries did as well or improved over exhaustive search accuracy when performing selective search using topic-based shards. While for

Table 5: P10 values for selective search on Clue-CatB with dependence model query. ▼ denotes significantly worse P10 than exhaustive search and ▲ denotes significantly better P10 than exhaustive search ($p < 0.05$).
Exhaustive search: P10=0.300, Cost=100%

	Rand	Source	K-means
1 Shard	▼0.080	▼0.156	0.302
Cost (%)	1.00	1.00	1.63
3 Shards	▼0.180	0.244	0.330
Cost (%)	3.00	3.00	4.99
5 Shards	▼0.212	0.278	0.314
Cost (%)	5.00	5.00	7.85
10 Shards	0.252	0.304	0.292
Cost (%)	10.00	9.80	14.69
15 Shards	0.254	0.306	0.294
Cost (%)	15.00	15.00	21.84

Table 6: P10 values for selective search on Clue-CatA-Eng with dependence model query. ▼ denotes significantly worse P10 than exhaustive search and ▲ denotes significantly better P10 than exhaustive search ($p < 0.05$).
Exhaustive search: P10=0.142, Cost=100%

	Rand	Source	K-means
1 Shard	▼0.024	▼0.056	0.152
Cost (%)	0.20	0.20	0.32
3 Shards	▼0.046	0.112	▲0.182
Cost (%)	0.60	0.60	1.12
5 Shards	▼0.066	0.120	0.174
Cost (%)	1.00	1.00	2.08
10 Shards	▼0.088	0.168	0.160
Cost (%)	2.00	2.01	4.81
15 Shards	0.114	0.174	0.146
Cost (%)	3.00	3.00	7.40

source-based 60% or more queries were found to perform well with selective search.

The selective search performance for the Clue datasets becomes comparable to that of exhaustive search much earlier in terms of shard cutoff than that for the Gov2 dataset. We believe this could be an artifact of the differences in the topical diversity of the datasets – the ClueWeb-09 dataset is much more diverse than the Gov2 dataset. As a result the topical shards of the ClueWeb-09 dataset are more dissimilar to each other than those for the Gov2 dataset. This could have an effect of concentrating similar documents in fewer shards for Clue datasets. Thus searching the top ranked shard is sufficient to retrieve most of the relevant documents. The topical diversity and the topically focused shards must also help reduce the errors during shard ranking.

The Clue datasets and the Gov2 dataset are also different in terms of the level of noise that is present in these datasets. Clue datasets have high percentage of noise while Gov2 is relatively clean. This could be one of the reasons why selective search is able to provide a significant improvement over exhaustive search for the Clue datasets. Selective searching of shards provides a natural way to eliminate some of the noise from the search space which improves the search accuracy by reducing the false positives from the final results.

More generally, these results reveal that each of the document allocation policies, more or less, converges to the exhaustive search performance, however, at very different rates. Topic-based converges the fastest and random converges the slowest.

8. CONCLUSIONS

This work demonstrated that exhaustive search of document collection is not always necessary to obtain competitive search accuracy. To enable this we partitioned the dataset into distributed indexes or shards, and then selectively searched a small subset of these shards. An important step in this process is the allocation of documents to various shards. We investigated three types of document allocation policies: random, source-based and topic-based.

Empirical results on three large datasets demonstrated that selective search of topic-based shards provides at least an order of magnitude reduction in search costs with no loss of accuracy, on average. 86% or more queries did as well or improved over exhaustive search accuracy when performing selective search using topic-based shards for all the three datasets. Although previous work hasn't reported this number anecdotal results suggest that this is much more stable than prior research. The results also demonstrate for the first time that selective search can consistently improve over exhaustive search while searching only a small fraction of the collection if a good document allocation policy has been employed to create the shards.

The topic-based document allocation technique studied in this work has two useful properties – scalability and generality. Scalability is achieved by using sampling-based approximation of K-means clustering to efficiently partition a large collection into topical shards. Our experiments show that even relatively small samples provide good coverage and statistics of corpus vocabulary. Generality is provided by the K-means clustering used to define topics, because it does not require any specific resources such as training data, query logs, click-through data, or predefined categories. Existing techniques such as caching that make use of resources like query-logs and click-through data to reduce search cost, can be used in combination with the techniques studied in this paper to further lower the search cost.

9. ACKNOWLEDGMENTS

This work was in part supported by the NSF grants IIS-0841275 and IIS-0916553. Any opinions, findings, conclusions and recommendations expressed in this paper are the authors' and do not necessarily reflect those of the sponsors.

10. REFERENCES

[1] R. Baeza-Yates, V. Murdock, and C. Hauff. Efficiency trade-offs in two-tier web search systems. In *Special Interest Group on Information Retrieval*, pages 163–170, Boston, MA, USA, 2009. ACM.

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

[3] J. Callan. Distributed information retrieval. In *Advances in Information Retrieval*, pages 127–150. Kluwer Academic Publishers, 2000.

[4] J. P. Callan, Z. Lu, and W. B. Croft. Searching distributed collections with inference networks. In

Special Interest Group on Information Retrieval, pages 21–28, New York, NY, USA, 1995. ACM.

[5] C. Clarke, N. Craswell, and I. Soboroff. Overview of the TREC 2004 Terabyte track. In *TREC*, 2004.

[6] J. Heaps. *Information Retrieval – Computational and Theoretical Aspects*. Academic Press Inc., New York, NY, 1978.

[7] L. S. Larkey, M. E. Connell, and J. Callan. Collection selection and results merging with topically organized U.S. patents and TREC data. In *Conference on Information and Knowledge Mangement*, pages 282–289, New York, NY, USA, 2000. ACM.

[8] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297. University of California Press, 1967.

[9] D. Metzler and W. B. Croft. Combining the language model and inference network approaches to retrieval. *Inf. Process. Manage.*, 40(5):735–750, 2004.

[10] D. Metzler and W. B. Croft. A markov random field model for term dependencies. In *Special Interest Group on Information Retrieval*, pages 472–479, New York, NY, USA, 2005. ACM.

[11] P. Ogilvie and J. Callan. Experiments using the lemur toolkit. In *TREC*, pages 103–108, 2001.

[12] D. Puppin, F. Silvestri, and D. Laforenza. Query-driven document partitioning and collection selection. In *InfoScale*, page 34, New York, NY, USA, 2006. ACM.

[13] M. Shokouhi. Central-rank-based collection selection in uncooperative distributed information retrieval. In *The 29th European Conference on Information Retrieval*, Rome, Italy, 2007.

[14] L. Si and J. Callan. Relevant document distribution estimation method for resource selection. In *Special Interest Group on Information Retrieval*, pages 298–305, New York, NY, USA, 2003. ACM.

[15] C. J. van Rijsbergen. *Information Retrieval*. Butterworths, 1979.

[16] J. Xu and W. B. Croft. Cluster-based language models for distributed retrieval. In *Special Interest Group on Information Retrieval*, pages 254–261, New York, NY, USA, 1999. ACM.

[17] C. Zhai and J. Lafferty. A study of smoothing methods for language models applied to information retrieval. *ACM Trans. Inf. Syst.*, 22(2):179–214, 2004.

Scaling Out All Pairs Similarity Search with MapReduce

Regular Paper

Gianmarco De Francisci Morales
IMT Institute for Advanced Studies
Lucca, Italy
gianmarco.dfmorales@imtlucca.it

Claudio Lucchese
ISTI-CNR
Pisa, Italy
claudio.lucchese@isti.cnr.it

Ranieri Baraglia
ISTI-CNR
Pisa, Italy
ranieri.baraglia@isti.cnr.it

ABSTRACT

Given a collection of objects, the All Pairs Similarity Search problem involves discovering all those pairs of objects whose similarity is above a certain threshold. In this paper we focus on document collections which are characterized by a sparseness that allows effective pruning strategies.

Our contribution is a new parallel algorithm within the MapReduce framework. The proposed algorithm is based on the inverted index approach and incorporates state-of-the-art pruning techniques. This is the first work that explores the feasibility of index pruning in a MapReduce algorithm. We evaluate several heuristics aimed at reducing the communication costs and the load imbalance. The resulting algorithm gives exact results up to 5x faster than the current best known solution that employs MapReduce.

1. INTRODUCTION

The task of discovering similar objects within a given collection is common to many real world applications and machine learning problems. To mention a few, recommendation and near duplicate detection are a typical examples.

Item-based and user-based recommendation algorithms require to find, respectively, similar objects to those of interest to the user, or other users with similar tastes. Due the the number of users and objects present in recommender systems, e.g. Amazon, similarity scores are usually computed off-line.

Near duplicate detection is commonly performed as a preprocessing step before building a document index. It may be used to detect redundant documents, which can therefore be removed, or it may be a hint for spam websites exploiting content repurposing strategies. Near duplicate detection finds application also in the area of copyright protection as a tool for discovering plagiarism, for both text and multimedia content.

In this paper, we focus on document collections. The reason is that documents are a particular kind of data that exhibits a significant sparseness: only a small subset of the whole lexicon occurs in any given document. This sparsity allows to exploit indexing strategies that reduce the potentially quadratic number of candidate pairs to evaluate.

Furthermore, we are interested in discovering only those pairs of documents with high similarity. If two documents are not similar, they usually do not contribute to any of the applications we mentioned above. By setting a minimum similarity threshold, we can also embed aggressive pruning strategies.

Finally, the size of the collection at hand poses new interesting challenges. This is particularly relevant for Web-related collections, where the number of documents involved is measured in billions. This implies an enormous number of potential candidates.

More formally, we address the all pair similarity search problem applied to a collection $\mathcal{D}$ of documents. Let $\mathcal{L}$ be the lexicon of the collection. Each document d is represented as a $|\mathcal{L}|$-dimensional vector, where $d[i]$ denotes the number of occurrences of the i-th term in the document d. We adopt the cosine distance to measure the similarity between two documents. Cosine distance is a commutative function, such that $\cos(d_i, d_j) = \cos(d_j, d_i)$.

DEFINITION 1. *Given a collection $\mathcal{D} = \{d_1, \ldots, d_N\}$ of documents, and a minimum similarity threshold σ, the* All Pairs Similarity (APS) *problem requires to discover all those document pairs $d_i, d_j \in \mathcal{D}$, such that:*

$$\cos(d_i, d_j) = \frac{\sum_{0 \le t < |\mathcal{L}|} d_i[t] \cdot d_j[t]}{\|d_i\|\|d_j\|} \ge \sigma$$

We normalize vectors to unit-magnitude. In this special case, the cosine distance becomes simply the dot product between the two vectors, denoted as $\text{dot}(d_i, d_j)$.

The main contribution of this work is a new distributed algorithm that embeds state-of-the-art pruning techniques. The algorithm is designed within the MapReduce framework, with the aim of exploiting the aggregated computing and storage capabilities of large clusters.

The rest of this paper is organized as follows: in Section 2 we introduce a few concepts needed for the description of the proposed algorithm. We also describe the two most relevant contributions to our work. Section 3 incrementally describes our proposed algorithm and heuristics, highlighting the strengths and weaknesses for each strategy. Section 4 presents the results of our experimental evaluation. Finally, in Section 5 we summarize our contribution and present some ideas for future work.

2. BACKGROUND

A serial solution. The most efficient *serial* solution to the APS problem was introduced in [3]. The authors use an *inverted index* of the document collection to compute similarities. An inverted index stores an *inverted list* for each term of the lexicon, i.e. the list of documents containing it, together with the weight of the term in each document. More formally, the inverted list of the term t is defined as $I_t = \{\langle d_i, d_i[t]\rangle | d_i[t] > 0\}$.

It is evident that two documents with non-zero similarity must occur in the same inverted list at least once. Therefore, given a document d_i, by processing all the inverted lists I_t such that $d_i[t] > 0$, we can detect all those documents d_j that have at least one term in common with d_i, and therefore similarity greater than 0.

This is analogous to information retrieval systems, where a query is submitted to the inverted index to retrieve matching/*similar* documents. In this case, a full document is used as a query.

Actually, index construction is performed incrementally, and simultaneously to the search process. The matching and indexing phase are performed one after the other. The current document is first used as a query to the current index. Then it is indexed, and it will be taken into account to answer subsequent document similarity queries. Each document is thus matched only against its predecessors, and input documents can be discarded once indexed.

Usually the matching phase dominates the computation because its complexity is quadratic with respect to the length of the inverted lists. In order to speed-up the search process, various techniques to prune the index have been proposed [2, 3].

We focus on the first technique proposed in [3]. Let $\hat{d}$ be an artificial document such that $\hat{d}[i] = \max_{d \in \mathcal{D}} d[i]$. The document $\hat{d}$ is an upper-bounding pivot: given a document d_i, if $\cos(d_i, \hat{d}) < \sigma$ then there is no document $d_j \in \mathcal{D}$ being sufficiently similar to d_i. This special document $\hat{d}$ is exploited as follows.

Before indexing the current document d_i, the largest b such that $\sum_{0 \le t < b} d_i[t] \cdot \hat{d}[t] < \sigma$ is computed. The terms $t < b$ of a document are stored in a remainder collection named $\mathcal{D}_R$, and only the terms $t \ge b$ of the current document are inserted into the inverted index. The pruned index provides partial scores upon similarity queries.

The authors prove that for each document d_i currently being matched, their algorithm correctly generates all the candidate pairs (d_i, d_j) using only the indexed components of each d_j. For such documents the remainder portion of d_j is retrieved from $\mathcal{D}_R$ to compute the final similarity score.

Finally, the authors propose to leverage the possibility of reordering the terms in the lexicon. By sorting the terms in each document by frequency in descending order, such that $d[0]$ refers to the most frequent term, most of the pruning will involve the longest lists.

A parallel solution.
When dealing with large datasets, e.g. collections of Web documents, the costs of serial solutions are still not acceptable. Furthermore, the index structure can easily outgrow the available memory. The authors of [5] propose a parallel distributed solution based on the MapReduce framework [4].

MapReduce is a distributed computing paradigm inspired by concepts of functional languages. More specifically, MapReduce is based on two higher order functions: Map and Reduce. The Map function applies a User Defined Function (UDF) to each key-value pair in the input, which is treated as a list of independent records. The result is a second list of intermediate key-value pairs. This list is sorted and grouped by key, and used as input to the Reduce function. The Reduce function applies a second UDF to every intermediate key with all its associated values to produce the final result.

The signatures of the functions that compose the phases of a MapReduce computation are as follows:

$$\begin{aligned} \mathit{Map}: &\quad [\langle k_1, v_1\rangle] &\rightarrow&\quad [\langle k_2, v_2\rangle] \\ \mathit{Reduce}: &\quad \{k_2 : [v_2]\} &\rightarrow&\quad [\langle k_3, v_3\rangle] \end{aligned}$$

where curly braces "{ }″ square brackets "[ ]" and angle brackets "⟨ ⟩" indicate respectively a map/dictionary, a list and a tuple.

The Map and Reduce function are purely functional and thus without side effects. For this reason they are easily parallelizable. Fault tolerance is easily achieved by just re-executing the failed function. MapReduce has become an effective tool for the development of large-scale applications running on thousand of machines, especially with the release of the open source implementation Hadoop [1].

Hadoop is an open source MapReduce implementation written in Java. Hadoop provides also a distributed file system called HDFS, that is used as a source and sink for MapReduce executions. HDFS deamons run on the same machines that run the computations. Data is split among the nodes and stored on local disks. Great emphasis is placed on data locality: the scheduler tries to run mappers (task executing the Map function) on the same nodes that hold the input data. This helps to reduce network traffic.

Mappers sort and write intermediate values on the local disk. Each reducer (task executing the Reduce function) pulls the data from various remote disks. Intermediate key-value pairs are already partitioned and sorted by key by the mappers, so the reducer just merge-sorts the different partitions to bring the same keys together. This phase is called *shuffle*, and is the most expensive in terms of I/O operations. The MapReduce data flow is illustrarted in Figure 1.

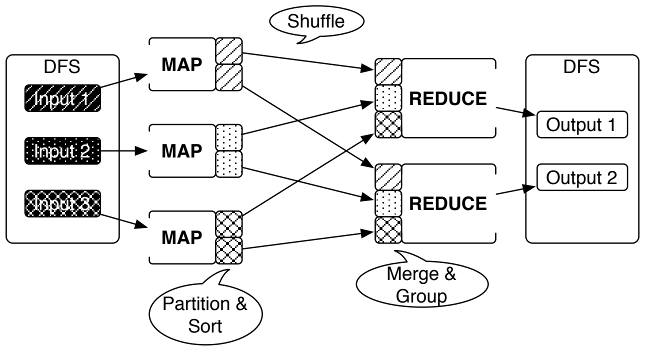


Figure 1: Data flow in a MapReduce job

Since building and querying incrementally a single shared index in parallel is not a scalable solution, a two phase algorithm is proposed in [5]. In the first phase an inverted index of the collection is built (indexing phase), and in the second phase the similarity score is computed directly from the index (similarity phase). Each phase is implemented as a MapReduce execution.

We describe this algorithm in more detail in the following section. The algorithm is used throughout the paper as a baseline for the evaluation of our proposed solutions. Indeed, the authors of [5] propose an algorithm for computing the similarity of every pair of documents. For this reason, we add a final filtering phase that discards the documents that do not satisfy the threshold.

3. ALGORITHM

In this section we describe the algorithm used to solve the APS problem using the MapReduce framework. We start from a basic algorithm and propose variations to reduce its cost. The main idea we try to exploit is that many of the pairs are not above the similarity threshold, so they can be pruned early from the computation. This fact is already exploited in state-of-the-art serial algorithms [2, 3]. Our goal is to embed these techniques into the MapReduce parallel framework.

3.1 Indexed Approach (Version 0)

A simple solution to the pairwise document similarity problem [5] can be expressed as two separate MapReduce jobs:

1. Indexing: for each term in the document, the mapper emits the term as the key, and a tuple consisting of document ID and weight as the value, i.e. the tuple $\langle d, d[t]\rangle$. The MapReduce runtime automatically handles the grouping by key of these tuples. The reducer then writes them to disk to generate the inverted lists.
2. Similarity: for each inverted list I_t, the mapper emits pairs of document IDs that are in the same list as keys. There will be $m \times (m-1)/2$ pairs where $m = |I_t|$ is the inverted list length. The mapper will associate to each pair the product of the corresponding term weights. Each value represents a single term's contribution to the final similarity score. The MapReduce runtime sorts and groups the tuples and then the reducer sums all the partial similarity scores for a pair to generate the final similarity score.

This approach is very easy to understand and implement, but suffers from various problems. First, it generates and evaluates all the pairs that have one feature in common, even if only a small fraction of them are actually above the similarity threshold. Second, the load is not evenly distributed.

The reducers of the similarity phase can only start after all the mappers have completed. The time to process the longest inverted list dominates the pair generation performed by the mappers. With real-world data, which follows a Zipfian or Power-law distribution, this means that the reducers usually have to wait for a single mapper to complete. This problem is exacerbated by the quadratic nature of the problem: a list twice as long takes about four times more to be processed.

A document frequency cut has been proposed to help reducing the number of candidate pairs [5]. This technique removes the 1% most frequent terms from the computation. The rationale behind this choice is that because these terms are frequent, they do not help in discerning documents. The main drawback of this approach is that the resulting similarity score is not exact.

3.2 Pruning (Version 1)

To address the issues in the previous approach, we employ the pruning technique described in Section 2. As a result, during the indexing phase, a smaller pruned index is produced. On the one hand, this reduces the number of candidate pairs produced, and therefore the volume of data handled during the MapReduce shuffle. On the other hand, by sorting terms by their frequency, the pruning significantly shortens the longest inverted lists. This decreases the cost of producing a quadratic number of pairs from these lists.

This pruning technique yields correct results when used in conjunction with dynamic index building. However, it also works when the index is built fully before matching, and only the index is used to generate candidate pairs. To prove this, we show that this approach generates every document pair with similarity above the threshold.

Let d_i, d_j be two documents and let b_i, b_j be, respectively, the first indexed features for each document. b_i and b_j are the boundaries between the pruned and indexed part as shown in Figure 2. Without losing generality, let $b_j \succ b_i$ (recall that features are sorted in decreasing order of frequency, so b_j is less frequent than b_i). We can compute the similarity score as the sum of two parts:

$$\text{dot}(d_i, d_j) = \sum_{0 \le t < b_j} d_i[t] \cdot d_j[t] + \sum_{b_j \le t < |\mathcal{L}|} d_i[t] \cdot d_j[t]$$

While indexing, we keep an upper bound on the similarity between the document and the rest of the input. This means that $\forall \dot{d} \in \mathcal{D}$, $\sum_{0 \le t < b_j} \dot{d}[t] \cdot d_j[t] < \sigma$. Thus, if the two documents are above the similarity threshold $\text{dot}(d_i, d_j) \ge \sigma$, then it must be that $\sum_{b_j \le t < |\mathcal{L}|} d_i[t] \cdot d_j[t] > 0$. If this is the case, then $\exists\, t \succ b_j \,|\, (d_i \in I_t \wedge d_j \in I_t)$. Therefore, our strategy will generate the pair (d_i, d_j) when scanning list I_t.

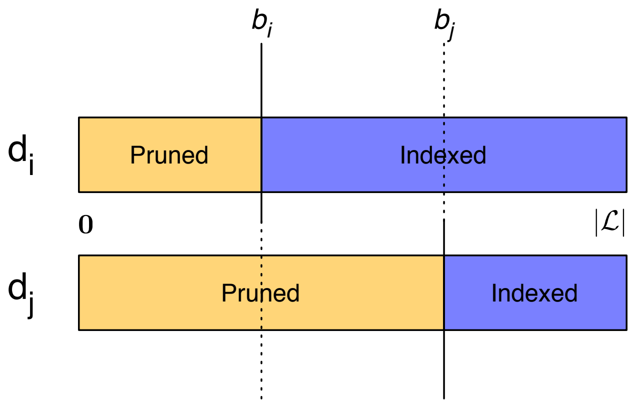


Figure 2: Pruned Document Pair: the left part (orange/light) has been pruned, the right part (blue/dark) has been indexed.

The Reduce function in the similarity phase receives a reduced number of candidate pairs, and computes a partial similarity score. Due to index pruning, no partial scores will be produced from the inverted lists $\{I_t \,|\, 0 \le t < b_j\}$, since these inverted lists will not contain both documents. Therefore, the reducer will have to retrieve the original documents, and compute the contribution up to term b_j in order to produce the exact similarity score.

We chose to distribute the input locally on every node[1]. The performance penalty of distributing the input collection is acceptable for a small number of nodes, but can become a bottleneck for large clusters. Furthermore, the input is usually too big to be kept in memory, so we still have to perform 2 random disk I/O per pair.

Finally, to improve the load balancing we employ a simple bucketing technique. During the indexing phase, we randomly hash the inverted lists to different buckets. This spreads the longest lists uniformly among the buckets. Each bucket will be consumed by a different mapper in the similarity phase. While more sophisticated strategies are possible, we found that this one works well enough in practice.

3.3 Flagging (Version 2)

In order to avoid the distribution of the full document collection, we propose a less aggressive pruning strategy. Our second approach consists in flagging the index items instead of pruning them. At the same time, the flagged parts of the documents are written as a side effect file by the mappers of the indexing phase. This *"remainders"* file is then distributed to all the nodes, and made available to the reducers of the similarity phase. The remainders file is normally just a fraction of the size of the original input (typically 10%), so distributing it is not a problem. During pair generation in the similarity phase, a pair is emitted only if at least one of the two index items is not flagged.

Our pair generation strategy emits all the pairs for the features from b_i to $|\mathcal{L}|$, so we just need to add the dot product of the remainders. The remainders file is small enough to be easily loaded in memory in one pass during the setup of the reducer. Thus, for each pair we only need to perform two in-memory lookups and compute their dot product. This process involves no I/O, so it is faster than the previous version.

The main drawback of this version is that it generates more pairs than version 1. This leads to unnecessary evaluation of pairs and consequently to wasted effort.

3.4 Using Secondary Sort (Version 3)

This version tries to achieve the benefits of both previous versions. Observe that for every pair (d_i, d_j) one of the two documents has been pruned up to a term that precedes the other (remember that features are sorted according to their frequency). Let this document be the LPD (Least Pruned Document) of the pair. Let the other document be the MPD (Most Pruned Document). In Figure 2, d_i is the LPD and d_j is the MPD.

We use version 1 pair generation strategy and version 2 remainder distribution and loading. To generate the partial scores we lack (from 0 to b_j), we just need to perform the dot product between the whole LPD and the remainder of the MPD. The catch is to have access to the whole LPD without doing random disk I/O and without keeping the input in memory.

Our proposed solution is to shuffle the input together with the generated pairs and route the documents where they are needed. In order to do that, we employ Hadoop's Secondary Sort feature. Normally, MapReduce sorts the intermediate records by key before starting the reduce phase. Using secondary sort we ask Hadoop to sort the records also by a secondary key while the grouping of values is still performed only by primary key. Instead of using the whole pair as a key, we use the LPD as the primary key and the MPD as the secondary key.

As a result, input values for the reducer are grouped by LPD, and sorted by both LPD and MPD, so that partial scores that belong to the same pair are adjacent. The LPD document from the original input that we shuffled together with the pairs is in the same group. In addition, we impose the LPD document itself to sort before every other pair using a fake minimum secondary key. This allows us to have access to the document before iterating over the values, and therefore to perform the dot products on the fly. This is a representation of the input for the reduce of the similarity phase:

$$\underbrace{\langle \mathbf{d_i} \rangle; \langle (d_i, d_j), W_{ij}^A \rangle; \langle (d_i, d_j), W_{ij}^B \rangle; \langle (d_i, d_k), W_{ik}^A \rangle; \ldots}_{group\ by\ key\ d_i}$$
$$\underbrace{\langle \mathbf{d_j} \rangle; \langle (d_j, d_k), W_{jk}^A \rangle; \langle (d_j, d_k), W_{jk}^B \rangle; \langle (d_j, d_l), W_{jl}^A \rangle; \ldots}_{group\ by\ key\ d_j}$$

First, we load the document $\mathbf{d_i}$ in memory. Then, for each stripe of equal consecutive pairs (d_i, d_j), we sum the partial scores W_{ij}^X for each common term X. Finally, we compute the dot product between the LPD and the remainder of the MPD, which is already loaded in memory from the remainders file. We repeat this cycle until there are no more values. After that we can discard the LPD from memory and proceed to the next key-values group.

4. EXPERIMENTAL RESULTS

In this section we describe the performance evaluation of the proposed algorithms. We ran the experiments on a 5-node cluster. Each node is equipped with two Intel Xeon E5520 CPUs clocked at 2.27GHz. Each CPU features 4 cores and Hyper-Threading for a total of 40 virtual cores. Each node has a 2TiB disk, 8GiB of RAM, and Gigabit Ethernet.

On each node, we installed Ubuntu 9.10 Karmic, 64-bit server edition, Sun JVM 1.6.0_20 HotSpot 64-bit server, and Hadoop 0.20.1 from Cloudera (CDH2).

We used one of the nodes to run Hadoop's master daemons (Namenode and JobTracker), and the rest were configured as slaves running Datanode and TaskTracker daemons. Two of the cores on each slave machine where reserved to run the daemons, the rest were equally split among map and reduce slots (7 each), for a total of 28 slots for each phase.

We tuned Hadoop's configuration in the following way: we allocated 1GiB of memory to each daemon and 400MiB to each task, we changed the HDFS block size to 256MiB and the file buffer size to 128KiB. We also disabled speculative execution and enabled JVM reuse and map output compression.

For each algorithm, we wrote an appropriate combiner to reduce the shuffle size (a combiner is a reduce-like function that runs inside the mapper to aggregate partial results). In our case, the combiners perform the sums of partial scores in the values, according to the same logic used in the reducer. We also implemented raw comparators for every key value used in the algorithms in order to get better performance (raw comparators are used to compare keys during sorting without deserializing them into objects).

[1]using Hadoop's Distributed Cache feature

# documents	17,024				30,683				63,126			
# terms	183,467				297,227				580,915			
# all pairs	289,816,576				941,446,489				3,984,891,876			
# similar pairs	94,220				138,816				189,969			
algorithm version	v0	v1	v2	v3	v0	v1	v2	v3	v0	v1	v2	v3
# evaluated pairs (M)	109	65	82	65	346	224	272	224	1,519	1,035	1,241	1,035
# partial scores (M)	838	401	541	401	2,992	1,588	2,042	1,588	12,724	6,879	8,845	6,879
index size (MB)	46.5	40.9	46.5	40.9	91.8	82.1	91.7	82.1	188.6	170.3	188.6	170.3
remainder size (MB)			4.7	4.7			8.2	8.2			15.6	15.6
running time (s)	3,211	1,080	625	554	12,796	4,692	3,114	2,519	61,798	24,124	17,231	12,296
avg. map time (s)	413	197	272	177	2,091	1,000	1,321	855	10,183	5,702	7,615	5,309
stdv. map time (%)	137.35	33.53	34.97	25.74	122.18	31.52	34.08	35.27	129.65	24.52	30.27	24.43
avg. reduce time (s)	57	558	35	79	380	2,210	191	220	1,499	11,330	1,112	1,036
stdv. reduce time (%)	18.76	5.79	13.59	14.66	48.00	5.62	23.56	14.61	13.55	2.46	8.37	9.51

Table 1: Statistics for various versions of the algorithm

We used different subsets of the TREC WT10G Web corpus. The dataset has 1,692,096 english language documents. The size of the entire uncompressed collection is around 10GiB.

We performed a preprocessing step to prepare the data for analysis. We parsed the dataset, removed stopwords, performed stemming and vectorization of the input. We extracted the lexicon and the maximum weight for each term. We also sorted the features inside each document in decreasing order of document frequency, as required by the pruning strategy.

4.1 Running Time

We evaluated the running time of the different algorithm versions while increasing the dataset size. For all the algorithms, the indexing phase took always less than 1 minute in the worst case. Thus we do not report indexing times, but only similarity computation times, which dominate the whole computation.

We set the number of mappers to 50 and the number of reducers to 28, so that the mappers finish in two waves and all the reducers can run at the same time and start copying and sorting the partial results while mappers are still running. For all the experiments, we set the similarity threshold to 0.9.

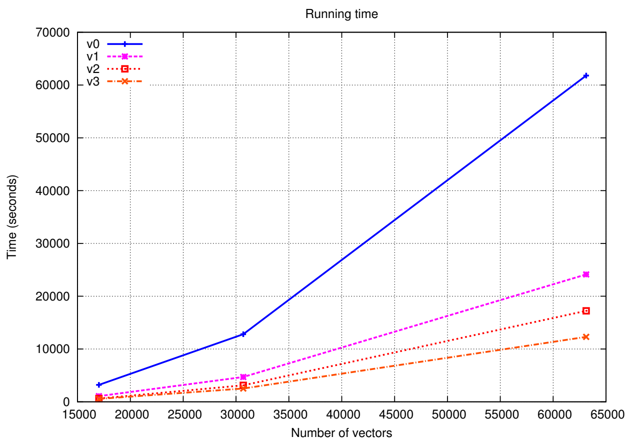


Figure 3: Computation times for different algorithm versions with varying input sizes

Figure 3 shows the comparison between running times for the different algorithms. The algorithms are all still quadratic, so doubling the size of the input roughly multiplies by 4 the running time. All the advanced versions outperform the basic indexed approach. This can easily be explained once we take into accounts the effects of of the pruning and bucketing techniques we applied.

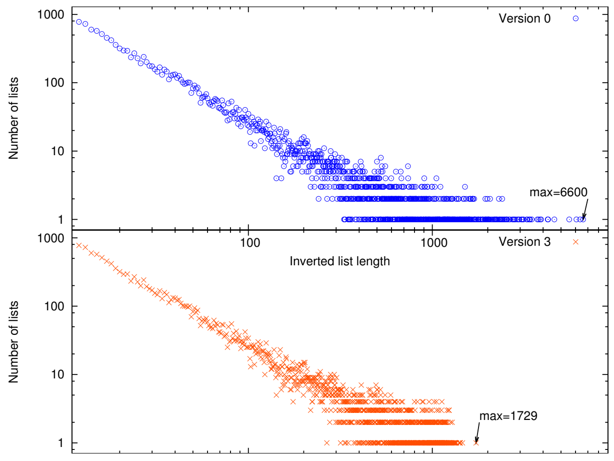


Figure 4: Index size distribution with and without pruning

Figure 4 shows the effects of pruning. The maximum length of the inverted lists is drastically reduced in version 3 compared to version 0. This explains their different running times, as the algorithm is dominated by the traversal of the longest inverted list. Figure 5 shows the effects of bucketing. The load is evenly spread across all the mappers, so that the time wasted waiting for the slowest mapper is minimized. It is evident also from Table 1 that the standard deviation of map running times is much lower when bucketing is enabled.

On the largest input, version 3 is 5x faster than version 0, 2x faster than version 1 and 1.4x faster than version 2. This is caused by the fact that version 3 does not access the disk randomly like version 1 and evaluates less pairs than version 2. Exact times are reported in Table 1.

Version 3 outperforms all the others in almost all aspects. The number of evaluated pairs and the number of partial scores are the lowest, together with version 1. Version 3 has also the lowest average map times. The standard deviation

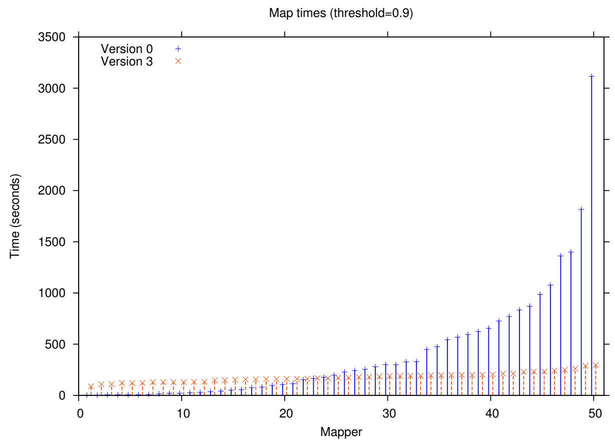


Figure 5: Map time distribution with and without bucketing

of map times for versions 1, 2 and 3 is much lower than version 0 thanks to bucketing.

For average reduce time, things change with different input sizes. For small inputs the overhead of version 3 does not pay back, and version 2 has the best trade-off between algorithm complexity and number of partial scores. For large inputs the smaller number of partial scores of version 3 gives it an edge over other versions. Version 1 is the slowest because of disk access and version 0 also scales poorly because of the large number of partial scores.

5. CONCLUSIONS AND FUTURE WORK

The All Pairs Similarity Search problem is a challenging problem that arises in many applications in the area of information retrieval, such as recommender systems and near duplicate detection. The size of Web-related problems mandates the use of parallel approaches in order to achieve reasonable computing times. In this

We presented a novel exact algorithm for the APS problem. The algorithm is based on the inverted index approach and is developed within the MapReduce framework. To the best of our knowledge, this is the first work to exploit well known pruning techniques from the literature adapting them to the MapReduce framework. We evaluated several heuristics aimed at reducing the cost of the algorithm. Our proposed approach runs up to 5x faster than the simple algorithm based on inverted index.

In our work we focused on scalability with respect to the input size. We believe that the scalability of the algorithm with respect to parallelism level deserves further investigation. In addition, we believe that more aggressive pruning techniques can be embedded in the algorithms. Adapting these techniques to a parallel environment such as MapReduce requires further study. We also want to investigate the application of our algorithm to other kinds of real world data, like social networks.

6. ACKNOWLEDGEMENTS

This work was partially supported both by the EU-FP7-215483 (S-Cube) and the POR-FESR 2007-2013 (VISITO Tuscany) projects.

References

[1] Apache Software Foundation. Hadoop: A framework for running applications on large clusters built of commodity hardware, 2006.

[2] A. Awekar and N. F. Samatova. Fast Matching for All Pairs Similarity Search. In *WI-IAT '09: Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology*, pages 295–300. IEEE Computer Society, 2009.

[3] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 131–140. ACM, 2007.

[4] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI '04: Proceedings of the 6th Symposium on Opearting Systems Design and Implementation*, pages 137–150. USENIX Association, December 2004.

[5] T. Elsayed, J. Lin, and D. W. Oard. Pairwise document similarity in large collections with MapReduce. In *HLT '08: Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies*, pages 265–268. Association for Computational Linguistics, 2008.

Efficient Dynamic Pruning with Proximity Support

Nicola Tonellotto
Information Science and Technologies Institute
National Research Council
Via G. Moruzzi 1, 56124 Pisa, Italy
nicola.tonellotto@isti.cnr.it

Craig Macdonald, Iadh Ounis
Department of Computing Science
University of Glasgow
Glasgow, G12 8QQ, UK
{craigm,ounis}@dcs.gla.ac.uk

ABSTRACT

Modern retrieval approaches apply not just single-term weighting models when ranking documents - instead, proximity weighting models are in common use, which highly score the co-occurrence of pairs of query terms in close proximity to each other in documents. The adoption of these proximity weighting models can cause a computational overhead when documents are scored, negatively impacting the efficiency of the retrieval process. In this paper, we discuss the integration of proximity weighting models into efficient dynamic pruning strategies. In particular, we propose to modify document-at-a-time strategies to include proximity scoring without any modifications to pre-existing index structures. Our resulting two-stage dynamic pruning strategies only consider single query terms during first stage pruning, but can early terminate the proximity scoring of a document if it can be shown that it will never be retrieved. We empirically examine the efficiency benefits of our approach using a large Web test collection of 50 million documents and 10,000 queries from a real query log. Our results show that our proposed two-stage dynamic pruning strategies are considerably more efficient than the original strategies, particularly for queries of 3 or more terms.

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

General Terms: Algorithms, Performance, Experimentation

Keywords: Dynamic Pruning, Efficient Proximity

1. INTRODUCTION

In most information retrieval (IR) systems, the relevance score for a document given a query follows the general outline given by the best match strategy: a score is calculated for each query term occurring in the document. These scores are then aggregated by a summation to give the final document relevance score. However, there are many queries where the relevant documents contain the query terms in close proximity. Hence, modern retrieval systems apply not just single-term weighting models when ranking documents. Instead, proximity weighting models are commonly applied, which highly score the co-occurrence of pairs of query terms in close proximity to each other in documents [8].

Dynamic pruning strategies reduce the scoring of documents, such that efficient retrieval can be obtained, without impacting on the retrieval effectiveness before rank K - such strategies are *safe-up-to-rank-K*. However, when additional proximity scores must be calculated for each document, the computational overhead impacts the efficiency of the retrieval process. While pruning techniques have been studied to efficiently score documents without considering term proximity [4, 20], there are very few proposals considering efficient top K retrieval where proximity is considered [19, 21, 22]. Moreover, these proposals require modifications of the index structure to implement efficient scoring strategies. Indeed, such modifications include sorting the posting lists by frequency or impact [2, 10], or using additional index structures containing the intersection of pairs of posting lists [19, 21, 22]. However, these can lead to negative effects on other aspects of the IR system, such as the compression of index structures or the impossibility to use other existing ranking strategies.

This work contributes a study into the behaviour of dynamic pruning strategies when combined with proximity weighting models. In particular, we analyse two existing document-at-a-time (DAAT) dynamic pruning strategies, namely MAXSCORE [20] and WAND [4], that can efficiently score documents without decreasing the retrieval effectiveness at rank K, nor requiring impact sorted indices. Moreover, we propose a runtime modification of these strategies to take into account proximity scores. We generate at runtime the posting lists of the term pairs, and transparently include the processing of these pair posting lists in the MAXSCORE and WAND strategies. Next, we propose a reorganisation of these strategies to increase their efficiency. Using thorough experiments on a 50 million document corpus and 10,000 queries from a real query log, we evaluate the proposed modification to determine their efficiency.

The remainder of this paper is structured as follows: In Section 2, we describe the state-of-the-art approaches to efficient ranking and the current existing solutions taking into account proximity scoring. In Section 3, we describe in detail the proposed framework to support proximity scores in DAAT strategies, and in Section 4, we evaluate the efficiency of the proposed modification. We provide concluding remarks in Section 5.

2. BACKGROUND

In the following, we outline the state-of-the-art strategies of dynamic pruning, followed by a discussion on proximity weighting models.

2.1 Dynamic Pruning

The algorithms to match and score documents for a query fall into two main categories [16]: in *term-at-a-time (TAAT)* scoring, the query term posting lists are processed and scored in sequence, so that documents containing query term t_i gain a partial score before scoring commences on term t_{i+1}. In contrast, in *document-at-a-time (DAAT)* scoring, the query term postings lists are processed in parallel, such that all *postings* of document d_j are considered before scoring commences on d_{j+1}. Compared to TAAT, DAAT has a smaller memory footprint than TAAT, due to the lack of maintaining intermediate scores for many documents, and is reportedly applied by large search engines [1]. An alternative strategy to DAAT and TAAT is called score-at-a-time [2], however this is suitable only for indices sorted or partially sorted by document importance, which must be calculated before the actual query processing. The algorithms from the family of threshold algorithms [10] work similarly.

Efficient dynamic pruning strategies do not rank every document in the collection for each user query; they manage to rank only the documents that will have a chance to enter in the top-K results returned to the users. These strategies are safe-up-to-rank-K [20], meaning that the ranking of documents up to rank K will have full possible effectiveness, but with increased efficiency. Dynamic pruning strategies rely on maintaining, at query scoring time, a threshold score that documents must overcome to be considered in the top-K documents. To guarantee that the dynamic pruning strategy will provide the correct top-K documents, an *upper bound* for each term on its maximal contribution to the score of any document in its posting list is used. In this paper, we focus on two state-of-the-art safe-up-to-rank-K DAAT dynamic pruning strategies, namely MaxScore and Wand.

The MaxScore strategy maintains, at query scoring time, a sorted list containing the current top-K documents scored so far. The list is sorted in decreasing order of score. The score of the last top-K document is a *threshold* score that documents must overcome to be considered in the top-K documents. A new document is given a *partial* score while the posting lists with that document are processed. A document scoring can terminate early when it is possible to guarantee that the document will never obtain a score greater than that of the current threshold. This happens when the current document score plus the upper bounds of terms yet to be scored is not greater than the threshold.

The Wand strategy maintains the same top-K documents list and the threshold score, but, for any new document, it calculates an *approximate* score, summing up some upper bounds for the terms associated with the document. If this approximate score is greater than the current threshold, then the document is fully scored. It is then inserted in the top-K candidate document set if this score is greater than the current threshold, and the current threshold is updated. If the approximate score check fails, the next document is processed. The selection of the next document to score is optimised [4] – however, for our purposes, it is of note that the set of postings lists are sorted by the document identifier (docid) they currently represent. More details on the Wand document selection strategy, which uses the skipping [16] of postings in the posting lists to reduce disk IO and increase efficiency, is presented in Appendix A.

The MaxScore and Wand dynamic pruning strategies can both enhance retrieval efficiency, whilst ensuring that the top K documents are fully scored – i.e. that the retrieval effectiveness at rank K is not at all negatively impacted. Generally, speaking, Wand is more efficient [14], due to its ability to skip postings for unimportant query terms. Note that both strategies examine at least one term from each document, and hence cannot benefit efficiency for single term queries.

2.2 Proximity

There are many queries where the relevant documents contain the query terms in close proximity. Hence, modern retrieval systems apply not just single-term weighting models when ranking documents. Instead, proximity weighting models are commonly applied, which highly score the co-occurrence of pairs of query terms in close proximity to each other in documents [8]. Hence, some scoring proximity (or term dependence) models have recently been proposed that integrate single term and proximity scores for ranking documents [5, 15, 18]. In this manner, the basic ranking model of an IR system for a query Q can be expressed as:

$$\text{score}_Q(d,Q) = \omega\, S(d) + \kappa \sum_{t \in Q} \text{score}(tf_d, *_d, t) + \phi\, \text{prox}(d,Q)$$

where $S(d)$ is the combination of some query independent features of document d (e.g. PageRank, URL length), and $\text{score}(tf_d, *_d, t)$ is the application of a weighting model to score tf_d occurrences of term t in document d. $*_d$ denotes any other document statistics required by a particular weighting model (e.g. document length). $\text{prox}(d,Q)$ represents some proximity document scoring function. The influence of the various features is influenced using weights ω, κ and ϕ.

However, none of the proximity weighting models proposed have been designed for efficient document scoring. The main approaches to integrate proximity weighting models into pruning strategies require modifications to the index structure to include information on the proximity scores upper bounds. In [19, 21, 22], the authors detail several approaches to leverage early termination when proximity scores are included in the ranking model. While these strategies alter the index structure (e.g. by adding term-pair inverted indices), we aim to exploit the proximity scores without modifying the index structure (other than keeping position occurrence information in the standard inverted index posting list). In particular, we use the sequential term dependence model of Markov Random Fields (MRF) [15], which has been shown to be effective at modelling the proximity of query term occurrences in documents. In MRF, the proximity score is calculated as follows:

$$\text{prox(d,Q)} = \sum_{p=(t_i,t_{i+1}) \in Q} \Big(\text{score}\big(pf(t_i,t_{i+1},d,k_1), l_d, p\big) + \text{score}\big(pf(t_i,t_{i+1},d,k_2), l_d, p\big) \Big)$$

where $pf(t_i, t_{i+1}, d, k)$ represents the number of occurrences of the pair of sequential query terms (t_i, t_{i+1}) occurring in document d in windows of size k (abbreviated as pair frequency pf_d). Following [15], we set $\kappa = 1, \phi = 0.1$, and $k_1 = 2$ and $k_2 = 8$ to account for the proximity of two terms as an exact phrase, and proximity at distance 8, respectively. $\text{score}(pf_d, l_d, p)$ is implemented using Dirichlet language modelling [11], but where pair frequency takes the role of term frequency. However, for the background statistics of the language model, in contrast to term weighting, when using proximity weighting, it is common to assume a constant frequency for the pair in the collection [13][1].

[1]As implemented by the authors of MRF in the Ivory retrieval system, see `www.umiacs.umd.edu/~jimmylin/ivory`

3. FRAMEWORK

The integration of proximity weighting models within efficient dynamic pruning strategies requires the materialisation of term pair posting lists and their integration into the existing dynamic pruning decision mechanism. In the following we discuss how we proposed to address both aspects.

3.1 Term pair posting lists

Most dynamic pruning algorithms use posting list iterators – object-oriented interfaces to a posting list, allowing a posting to be read, or to be moved on to the next posting. With a standard inverted index, one posting list's iterator represents the documents in which a single query term occurs, ordered by docid.

Proximity weighting models require knowledge of the occurrence of pairs of query terms in a document. The posting list of pairs of terms can be constructed either statically (i.e., at indexing time, calculating the intersections of all pairs of term posting lists) or dynamically (i.e., at retrieval time, generating term pair postings on the fly). Previous approaches [19, 21, 22] investigated different methodologies to statically calculate these intersections. However, the static approach has two drawbacks. Firstly, storing new posting lists requires additional space on disk, and secondly, the pairs of terms whose posting lists must be intersected must be known in advance (e.g. by identifying popular phrases in the corpus [22]), to avoid generating a large number of new, potentially useless posting lists. While these drawbacks may be lightened by caching solutions to store paired posting lists [12], even in this case, there is always a relative consumption of disk or memory resources.

Instead, the pair posting lists can be built dynamically. Given two single term iterators on postings lists, there is a valid term *pair posting* each time they point to the same docid. In order to transparently include these pair postings in existing DAAT strategies, we must be sure that they are ordered by docid. A pair posting list is illustrated in Figure 1, based on the postings for terms t_1 and t_2. In our proposed approach, to avoid additional I/O operations at runtime, only the single term posting lists are responsible for reading from disk and decompressing the single postings, while the pair posting docid is updated each time a new single posting is read with the minimum of the current single term docids. The pair posting is valid only when the docids of the underlying single term posting lists are equal (i.e., in Figure 1, only two valid postings exist, namely docid 1 and docid 8.). When a term posting list ends, all the associated pair posting lists end as well. Overall, the pair posting list is docid-sorted and cannot skip over potentially useful term pair postings, however, a number of invalid pair postings will occur (e.g. (8,2) and (9,14) in Figure 1).

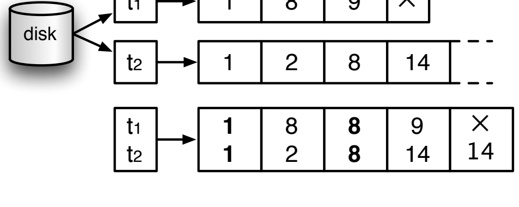


Figure 1: The dynamic creation of a pair posting list for terms t_1 and t_2. Bold entries are valid pair postings, while × indicates the end of a posting list.

3.2 Dynamic pruning with proximity

The dynamic pair posting lists can be directly put into work in existing DAAT strategies without modification. When a term pair posting is selected for scoring, it is necessary to calculate the exact value for the pair frequency at window size k, by comparing the lists of positions stored in both term postings. With dynamic pruning strategies (MaxScore and Wand), this computation can be avoided if the posting is not considered for scoring. Moreover, both the MaxScore and Wand pruning strategies require upper bounds on the score contributions of single terms. Hence, when using proximity, we need also to provide upper bounds on the score contributions of pairs as well. In [4], the authors proposed using a dynamic estimation of the inverse document frequency of pairs to determine the upper bound (the particular proximity weighting model is not defined, but assumed to be similar to [5]). In [14], we proposed a new approximation for upper bounds of the Markov Random Fields, requiring only the knowledge of the maximum term frequency of the postings in the two term posting lists.

We now describe how proximity weighting is achieved using the dynamic pruning strategies. In particular, the MaxScore strategy must always know the minimum docid in the currently processed posting lists set (which can be obtained by maintaining a heap), while the Wand strategy must have access to the posting lists sorted by docid (i.e., in the worst case, every posting in each posting list must be removed and inserted in a sorted set). However, when proximity is considered, many extra pair postings must be considered (i.e., $|Q|$ single term postings, plus an additional $2(|Q| - 1)$ pair postings) – causing the efficiency of Wand to be hindered. Moreover, both strategies must make additional checks to ensure that only 'valid' pair postings are considered, which can cause a performance bottleneck.

To deal with these limitations, we propose a modification that can be applied to both MaxScore and Wand pruning strategies, whereby the processing of single terms is separated from that of term pairs during each document scoring. We refer to these *two-stage* strategies as MaxScoreP and WandP. In particular, if a pair posting is updated after each term posting update, we will generate two potentially invalid pair postings. With the proposed modification, we update the pair postings only after all single terms have been moved to their respective next posting. This implies that we can generate only one pair posting instead of two each time both of the single term posting iterators advance. Hence, MaxScoreP and WandP process the single term posting lists according to their respective algorithms, however the term pairs are subsequently processed in a second stage using early termination, according to the MaxScore strategy. The use of early termination of proximity scoring is motivated by the fact that the pair frequency of a pair posting is expensive to compute (in comparison to term frequency, which is directly recorded in the posting) – hence early termination can reduce the unnecessary pair frequency and proximity score calculations.

In summary, we propose to implement proximity scoring using only normal index structures at retrieval time, and in such a way to integrate directly with existing DAAT dynamic pruning strategies. Moreover, we propose a modification to these dynamic pruning strategies, where the single terms are processed first according to the original dynamic pruning strategy, while the terms pairs are processed according to the MaxScore strategy. This can significantly reduce the performance impact of additional data structures required at runtime, and, as will be shown in Section 4, leads to improved retrieval efficiency. Moreover, both of the MaxScoreP and WandP modified strategies remain safe-up-to-rank-K.

4. EVALUATION

In the following experiments, we want to evaluate the benefits of the proposed modification of the dynamic pruning strategies. We are mainly interested in efficiency, because all strategies are safe-up-to-rank-K – hence have no impact on effectiveness. We tackle the following research questions:

1. How do MaxScore and Wand compare when applying the Markov Random Fields proximity weighting model?
2. Do the proposed modifications benefit the efficiency when applying proximity?
3. What impact does the length of the query have?

Experiments are performed using a 230GB 50 million English document subset of the TREC ClueWeb 09 (CW09B) corpus [6]. Documents are ranked using the Dirichlet LM weighting model [11] (with parameter setting $\mu = 4000$) and the Markov Random Fields proximity weighting (see Section 2.2). No query-independent features are used (i.e., $\omega = 0$). CW09B is indexed using the Terrier IR platform [17][2], applying Porter's English stemmer and removing standard stopwords. In the posting list, docids are encoded using Elias Gamma-encoded deltas [9] and term frequencies using Elias Unary [9]. The positions of occurrences of the term within the document are also recorded in each posting, using Elias Gamma-encoded deltas. Each posting list also includes skip points [16], one every 10,000 postings. The resulting size of the inverted index is 72GB.

For testing retrieval efficiency, we extract a stream of user queries from a real search engine log. In particular, we select the first 10,000 queries of the MSN 2006 query log [7], applying Porter's English stemmer and removing standard stopwords (empty queries are removed). The experiments measure the average query response time for each dynamic pruning strategy, broken down by the number of query terms (1, 2, 3, 4 and more than 4). The number of documents retrieved for each query is $K = 1,000$. All experiments are made using a dual quad-core Intel Xeon 2.6GHz, with 8GB RAM and a 2TB SATA2 disk containing the index.

In the following, we compare five strategies, namely: an exhaustive "Full" DAAT strategy, which fully scores every posting for all query terms and pairs; the original MaxScore and Wand dynamic pruning strategies without any modification; and the proposed two-stage MaxScoreP and WandP dynamic pruning strategies which integrate the modification proposed in Section 3.2. Every strategy uses dynamic pair posting lists, as discussed in Section 3.1.

Table 1 details the average query response time for both the original and two-stage strategies per number of query terms. From this table, we note that the average response time are reduced by approximately 22%-30% by applying the original MaxScore, but for the original Wand the average response times are worse than for Full DAAT scoring. This counters the normal efficiency of Wand, and supports

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

	# of query terms				
	1	2	3	4	> 4
# queries	3456	3149	1754	853	550
Full	**0.53**	2.76	5.57	9.95	16.42
Original strategies					
MaxScore	**0.53**	2.07	3.92	6.45	12.68
Wand	**0.53**	3.01	5.78	10.67	18.42
Two-stage strategies					
MaxScoreP	**0.53**	1.90	3.32	5.27	8.51
WandP	**0.53**	**1.67**	**2.73**	**4.46**	**7.77**

Table 1: Average response times (in seconds), original strategies and two-stage strategies.

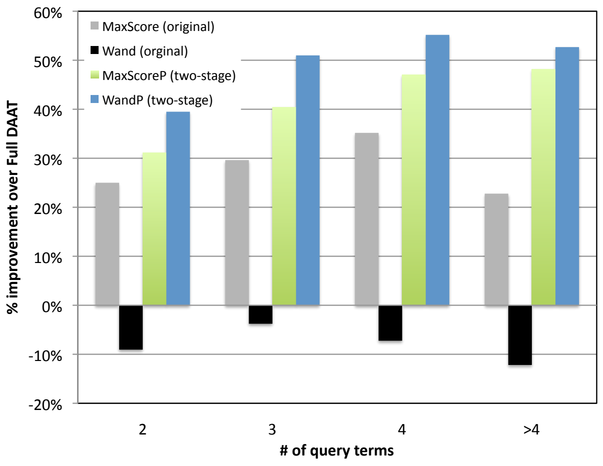


Figure 2: The relative impact on the average response time of the proposed strategies w.r.t. the Full DAAT strategy.

our assertion that Wand is not suitable for proximity scoring in its normal form. Indeed, when using the pair posting lists, there is a higher number of posting lists to maintain in the docid sorted set of posting lists used by Wand, in addition to the check that each pair posting is valid.

In contrast, the two-stage pruning strategies perform better in comparison to their original versions. MaxScoreP exhibits improvements of average response times varying from 31% for two terms, up to 48% for more than four terms, while WandP benefits vary from 39% to 58%. Moreover, we note that WandP exhibits better efficiency than MaxScoreP, in common with MaxScore versus Wand for non-proximity queries [14]. For single term queries, no proximity is applied, and, as expected, all dynamic pruning strategies are equally efficient to Full DAAT scoring.

Figure 2 summarises the percentage differences of the dynamic pruning strategies with respect to the Full DAAT scoring, for varying lengths of query. As already reported, the two-stage MaxScoreP and WandP strategies outperform their original equivalents. Moreover, their benefits increase as the length of the queries (and hence pairs of terms) increase, up to 40-55% improvements for queries with 3 or more terms.

5. CONCLUSIONS

In this work, we examined how to efficiently score documents using dynamic pruning strategies when using the Markov Random Field proximity weighting model [15]. In particular, we discussed how pair posting lists could be used to allow proximity scoring without changes to the underlying index. Moreover, the most efficient way to score documents using DAAT dynamic pruning strategies was discussed. We proposed that dynamic pruning should be performed in a two-stage process (as exemplified by the MaxScoreP and WandP strategies), whereby only single query terms are processed and pruned in the first stage. The pair posting lists are only considered during a second stage, since they are omitted from consideration in the first stage.

We performed large-scale experiments comparing the orginal and proposed two-stage dynamic pruning strategies, using a corpus of 50 million documents, and 10,000 user queries from a real query log. Our results demonstrated the benefit of the two-stage versions of the dynamic pruning strategies, particularly for queries of 3 or more terms, and are a promising start for the future efficient examination of other proximity weighting models.

Dynamic pruning techniques have previously been shown to be readily appliable in distributed retrieval settings [3]. Similarly, we infer that our strategies can also be applied in distributed retrieval without requiring adaptation.

6. REFERENCES

[1] A. Anagnostopoulos, A. Z. Broder and D. Carmel. Sampling search-engine results. In *Proc. of WWW 2005*, 245–256.

[2] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impact scores. In *Proc. of SIGIR 2006*, 372–379.

[3] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Plachouras and L. Telloli. On the feasibility of multi-site web search engines. In *Proc. of CIKM 2009*, 425–434.

[4] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. of CIKM 2006*, 426–434.

[5] S. Büttcher, C. L. A. Clarke and B. Lushman. Term proximity scoring for ad-hoc retrieval on very large text collections. In *Proc. of SIGIR 2006*, 621–622.

[6] C. L. A. Clarke, N. Craswell and I. Soboroff. Overview of the TREC 2009 Web track. In *Proc. of TREC 2009.*

[7] N. Craswell, R. Jones, G. Dupret and E. Viegas. *Proc. of the Web Search Click Data Workshop at WSDM 2009.*

[8] W. B. Croft, D. Metzler and T. Strohman. Search Engines: Information Retrieval in Practice Addison Wesley, 2009.

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

[10] R. Fagin, A. Lotem and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.* 66(4):614–656, 2003.

[11] J. Lafferty and C. Zhai. A study of smoothing methods for language models applied to information retrieval. In *Proc. of SIGIR 2001*, 334–342.

[12] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. In *Proc. of WWW 2005*, 257–266.

[13] C. Macdonald and I. Ounis. Global Statistics in Proximity Weighting Models. In *Proc. of Web N-gram Workshop at SIGIR 2010.*

[14] C. Macdonald, N. Tonellotto and I. Ounis. Upper Bound Approximations for Dynamic Pruning. *Manuscript submitted for publication*, 2010.

[15] D. Metzler and W. B. Croft. A Markov random field model for term dependencies. In *Proc. of SIGIR 2005*, 472–479.

[16] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *Transactions on Information Systems*, 14(4):349–379, 1996.

[17] 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 at SIGIR 2006.*

[18] J. Peng, C. Macdonald, B. He, V. Plachouras and I. Ounis. Incorporating term dependency in the DFR framework. In *Proc. of SIGIR 2007*, 843-844.

[19] R. Schenkel, A. Broschart, S. Hwang, M. Theobald and M. Gatford. Efficient text proximity search. In *Proc. of SPIRE 2007*, 287–299.

[20] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Information Processing and Management*, 31(6):831–850, 1995.

[21] M. Zhu, S. Shi, M. Li and J.-R. Wen. Effective top-k computation in retrieving structured documents with term-proximity support. In *Proc. of CIKM 2007*, 771–780.

[22] M. Zhu, S. Shi, N. Yu and J.-R. Wen. Can phrase indexing help to process non-phrase queries? In *Proc. of CIKM 2008*, 679–688.

APPENDIX

A. WAND NEXT DOCUMENT SELECTION

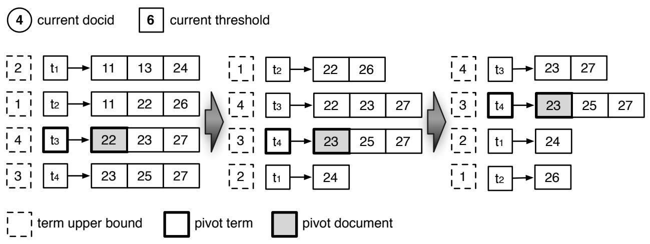


Figure 3: How the WAND strategy selects the next document to score

The selection of the next document performed in the Wand strategy is explained with the help of Figure 3. The posting lists are maintained in increasing order of docid. Then a *pivot term* is computed, i.e. the first term for which the accumulated sum of upper bounds of preceding terms and itself exceeds the current threshold (e.g., term t_3 with accumulated score of 7). The corresponding docid identifies the *pivot document*, i.e. the smallest docid having a chance to overcome the current threshold. If the current docids of the previous terms are equal to the pivot document docid, the document is fully scored. Otherwise, one of the preceding terms posting list is moved to the pivot document docid, and the procedure is repeated. In the example, at the third step a good candidate document is found (23) and it is fully processed. This implementation can benefit from skipping every posting list to a particular document, however the selection of the right document requires some additional CPU time.

Performance evaluation of large-scale Information Retrieval systems scaling down

Position paper

Fidel Cacheda, Víctor Carneiro, Diego Fernández, Vreixo Formoso
Facultad de Informática
Campus de Elviña s/n
15071, A Coruña, Spain
{fidel.cacheda, victor.carneiro, dfernandezi, vformoso}@udc.es

ABSTRACT

The performance evaluation of an IR system is a key point in the development of any search engine, and specially in the Web. In order to get the performance we are used to, Web search engines are based on large-scale distributed systems and to optimise its performance is an important aspect in the literature.

The main methods, that can be found in the literature, to analyse the performance of a distributed IR system are: the use of an analytical model, a simulation model and a real search engine. When using an analytical or simulation model some details could be missing and this will produce some differences between the real and estimated performance. When using a real system, the results obtained will be more precise but the resources required to build a large-scale search engine are excessive.

In this paper we propose to study the performance by building a scaled-down version of a search engine using virtualization tools to create a realistic distributed system. Scaling-down a distributed IR system will maintain the behaviour of the whole system and, at the same time, the computer requirements will be softened. This allows the use of virtualization tools to build a large-scale distributed system using just a small cluster of computers.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Performance evaluation (efficiency and effectiveness)*

General Terms

Information Retrieval

Keywords

Distributed Information Retrieval, Performance evaluation, Scalability

1. INTRODUCTION

Web search engines have changed our perspective of the search process because now we consider normal being able to search through billions of documents in less than a second. For example, we may not quite understand why we have to wait so long in our council for a certificate as they just have to search through a "few" thousand/million records.

However, Web search engines have to use a lot of computational power to get the performance we are used to. This computational power can only be achieved using large-scale distributed architectures. Therefore, it is extremely important to determine the distributed architectures and techniques that allow clear improvements in the system performance.

The performance of a Web search engine is determined basically by two factors:

- Response time: the time it takes to answer the query. This time includes the network transfer times that, in Internet, will take a few hundred milliseconds; and the processing time in the search engine, that is usually limited to 100 milliseconds.
- Throughput: the number of queries the search engine is able to process per second. This measure usually has to maintain a constant ratio, but also deals with peak loads.

From the user's point of view, only the response time is visible and it is the main factor, keeping constant the quality of the results: the faster the search engine is able to answer the better. From the search engine point of view both measures are important. Once an upper limit has been set up for the response time (e.g. a query should be answered in less than 100 milliseconds), the objective is to maximise the throughput.

From the search engine point of view another two factors have to be taken into account:

- Size: the number of documents indexed by the search engine. Not so long ago, Google published in its main page the number of Web pages indexed. Nowadays, the main commercial search engines do not make public detailed figures, although the estimations are in the order of 20 billion documents.

- Resources: the number of computers used by the search engine. This could be considered from the economical perspective as the cost of the distributed system. In [2] Baeza-Yates et al. estimate that, a search engine will need about 30 thousand computers to index 20 billion documents and obtain a good performance.

If we want to compare different distributed indexing models or test some new techniques for a distributed search engine (e.g. a new cache policy), usually we will fix the size of the collection and the resources and then, measure the performance in terms of response time and throughput.

Ideally, we would need a replica of a large-scale IR system (for example, one petabyte of data and one thousand computers) to measure performance. However, this would be extremely expensive and no research group, or even commercial search engine, can devote such amount of resources only to evaluation purposes.

In this article we present a new approach for performance evaluation of large-scale IR systems based on scaling-down. We consider that, creating a scaled-down version of an IR system, will produce valid results for the performance analysis, using very few resources. This is an important point for commercial search engines (from the economical point of view), but it is more important for the research groups because this could open the experimentation on large-scale IR to nearly any group.

The rest of the paper is organised as follows. In Section 2 we present the main approaches for performance evaluation. Section 3 analyses our proposal and Section 4 concludes this work and describes some ideas for future works.

2. PERFORMANCE EVALUATION

In the literature there are many articles that evaluate the performance of a search engine or one of its components. We do not intend to present an exhaustive list of papers about performance evaluation but to present the main methods used by the researchers, specially of a large-scale IR system.

The main methods to test the performance of a search engine are the following:

- An analytical model.
- A simulation model.
- A real system or part of a real system.

A clear example of study based on an analytical model can be found in [6]. In this work, Chowdhury et al. use the queueing network theory to model a distributed search engine. The authors model that the processing time in a query server is a function of the number of documents indexed. They build a framework in order to analyse distributed architectures for search engines in terms of response time, throughput and utilization. To show the utility of this framework, they provide a set of requirements and study different scalability strategies.

There are many works based on simulation that study the performance of a distributed IR system. [3] is one of the first. In this work, Burkowski uses a simple simulation model to estimate the response time of a distributed search engine, and uses one server to estimate the values for the simulation model (e.g. the reading time from disk is approximated as a Normal distribution). Then, the simulation model represents a clusters of servers and estimates the response times using local index organisation (named uniform distribution). However, the network times are not considered in the simulation model.

Tomasic and Garcia-Molina [12] also used a simulation model to study the performance of several parallel query processing strategies using various options for the organization of the inverted index. They use different simulation models to represent the collection documents, the queries, the answer set and the inverted lists.

Cacheda et al. in [4] include also a network model to simulate the behaviour of the network in a distributed IR system. They compare different distribution architectures (global and local indexing), identify the main problems and present some specific solutions, such as, the use of partial result sets or the hierarchical distribution for the brokers.

Other authors use a combination of both approaches. For example, in [10], Ribeiro-Neto and Barbosa use a simple analytical model to estimate the processing time in a distributed system. This analytical model calculates the seek time for a disk, the reading time from disk of an inverted list, the time to compare and swap two terms and the transfer time from one computer to another. In their work, they include a small simulator to represent the interference among the various queries in a distributed environment. They compare the performance of a global index and a local index and study the effect of the network and disk speed.

Some examples of works experimenting with a real IR system could be [1] or [9]. In the first work, Badue et al. study the imbalance of the workload in a distributed search engine. They use a configuration of 7 index servers and one broker to index a collection of 10 million Web pages. In their work, the use of a real system for testing was important to detect some important factors for imbalance in the index servers. They state that the correlations between the term frequency in a query log and the size of its inverted list lead to imbalances in query execution times, because these correlations affect the behaviour of the disk caching.

Moffat et al. in [9] study a distributed indexing technique named *pipelined distribution*. In a system of 8 servers and one broker, they index the TREC Terabyte collection [7] to run their experiments. The authors compare three distributed architectures: local indexing (or document partitioning), global indexing (or term partitioning) and pipelining. In their experiments the pipelined distribution outperforms the term partitioning, but not the document partitioning due to a poor workload balancing. However, they also detect some advantages over the document distribution: a better use of memory and fewer disk seeks and transfers.

The main drawback for an analytical model is that it cannot represent all the characteristics of a real IR system. Some features have to be dropped to keep the model simple and easy to implement.

Using a simulation model, we can represent more complex behaviours than an analytical model. For example, instead of assuming a fixed transfer time for the network, we can simulate the behaviour of the network (e.g. we could detect a network saturation). But, again, not all the features of a real system could be implemented. Otherwise, we will end up with a real IR system and not a simulation model.

In both cases, it is important to use a real system to estimate the initial values of the model (analytical or simulated) and, in fact, this is a common practise in all the research works. In a second step, it is also common to compare the results of the model with the response obtained from a real system, using a different configuration, in order to validate the model.

However, when the models are used to extrapolate the behaviour of a distributed IR system, for example increasing the number computers, the results obtained may introduce a bigger error than expected. For example, a simulation model of one computer, when compared with a real system, has an accuracy of 99% (or an error of 1%). But, what is the expected error when simulating a system with 10 computers: 1% or 10%?

This problem is solved by using a real system for the performance evaluation. But, in this case, the experiments will be limited by the resources available in the research groups. In fact, many researchers run their experiments using 10-20 computers. Considering the size of data collections and the size of commercial search engines, this could not be enough to provide interesting results for the research community. In this sense, the analytical or simulation models allow us to go further, at the risk of increasing the error ratio of our estimations.

3. OUR PROPOSAL

In this article we propose to use a scaled-down version of a search engine to analyse the performance of a large-scale search engine.

Scaling down has been successfully applied in many other disciplines, and it is specially interesting when the development of a real system is extremely expensive. For example, in the shipping industry the use of scale models in basins is an important way to quantify and demonstrate the behaviour of a ship or structure, before building a real ship [5].

The use of a wind tunnel is also quite common in the aeronautical or car industries. Specially in the former, the scaled-down models of planes or parts of a plane are important to analyse the performance of the structure [13]. Also in architecture scaled-down models are used to test and improve the efficiency of a building [8].

In the world of search engines, is it possible to build a scaled-down version of a search engine?

Let us say that we want to study the performance of a large-scale search engine, composed of 1000 computers, with the following parameters:

- The size of the collection is 1 petabyte.
- Each computer has a memory of 10 gigabytes.
- Each computer has a disk of 1 terabyte.
- The computers are interconnected using a high speed network (10Gbits/second).

From our point of view, maintaining the 1000 computers as the core of the distributed system, if we apply a scale factor of 1:1000 we will have a scaled-down version of the search engine with the following parameters:

- The size of the collection is 1 gigabyte.
- Each computer has a memory of 10 megabytes.
- Each computer has a disk of 1 gigabyte.
- The computers are interconnected using a high speed network (10Mbits/second).

One important point is that the scale factor does not apply to the number of computers. The computers constitute the core of the distributed system and therefore cannot be diminished, instead they are scaled-down. This is equivalent to build a scaled-down version of a building: the beams are scaled-down but not diminished.

In this way, we expect to obtain a smaller version of the large-scale search engine, but with the same drawbacks and benefits.

The next step is how to build this scaled-down version of a search engine.

The first and trivial solution is to use 1000 computers with the requirements stated previously. These would be very basic computers nowadays but, anyway, it could be quite complicated to obtain 1000 computers for a typical research group. It could be a little bit easier for a commercial search engine if they could have access to obsolete computers from previous distributed systems, but it is not straightforward.

A more interesting solution would be to use virtualization tools to create the cluster of computers. Virtualization is a technology that uses computing resources to present one or many operating systems to user. This technology is based on methodologies like hardware and software partitioning, partial or complete machine simulation and others [11]. In this work, we are interested in the virtualization at the hardware abstraction layer to emulate a personal computer. Some well-known commercial PC emulators are KVM[1], VMware[2], VirtualBox[3] or Virtual PC[4].

With this technology it could be possible to virtualize a group of scaled-down computers using just one real computer. In this way, with a small cluster of computers (e.g. 20 computers) we could virtualize the whole scaled-down search engine.

For example, using a cluster of 20 computers, will require that each computer virtualizes 50 computers with 10 megabytes of memory and 1 gigabyte of disk. Roughly speaking, this would take half a gigabyte of memory and 50 gigabytes of disk from the real machine, which should be easily handled by any modern computer.

To the best of our knowledge, all the virtualization tools allow you to set the size of memory and disk for the virtualized host. Also, some of them (e.g. VMware) can set a limit for the network usage, in terms of average or peak, and for the CPU speed.

From our point of view, these requirements should be enough to scale-down a host of a distributed search engine. Some other parameters could also be considered when scaling down a search engine, such as, disk or memory speed. We are aware of some solutions in this sense[5], but these low level parameters could be quite hard to virtualize. However, we doubt about the usefulness of these parameters in the performance analysis, while the performance of the real computer is not degraded. In any case, in future works it

[1]http://www.linux-kvm.org/
[2]http://www.vmware.com/
[3]http://www.virtualbox.org/
[4]http://www.microsoft.com/windows/virtual-pc/
[5]http://sourceforge.net/apps/trac/ioband/

would be interesting to study in depth the effect of these low level parameters in the performance analysis of a scaled-down search engine.

The use of virtualization is not only interesting to reduce the resources required to build a scaled-down version of a search engine. It could be very appealing to test the effect of new technologies in the performance of a large-scale search engine. For example, let us say that we want to compare the performance of the new SSD (Solid State Drive) memories versus the traditional hard drives. To build a whole search engine using SSD memories would be a waste of resources until the performance has been tested. But, buying a cluster of computers with SSD memory and building a scaled-down version of the search engine is feasible and not very expensive. In this way, we could test and compare the performance of this new technology.

4. CONCLUSIONS

This paper presents a new approach to the performance evaluation of large-scale search engines based on a scaled-down version of the distributed system.

The main problem, when using an analytical or simulation model for evaluation purposes, is that some (important) details could be missing in order to make the model feasible and so, the estimations obtained could differ substantially from the real values.

If we use a real search engine for performance evaluation, the results obtained will be more precise but will depend on the resources available. A distributed system composed of a few computers does not constitute a large-scale search engine and the resources required to build a representative search engine are excessive for most researchers.

We suggest to build a scaled-down version of a search engine using virtualization tools to create a realistic cluster of computers. By using a scaled-down version of a computer we expect to maintain the behaviour of the whole distributed system at the same time that the hardware requirements are softened. This would be the key to use the virtualization tools to build a large distributed system using a small cluster of computers.

This research is at an early stage, but we strongly believe that this would be a valid technique to analyse the performance of a large-scale distributed IR system.

In the near future we plan to develop a scaled-down search engine using a small cluster of computers. We would like to compare the performance of the scaled-down search engine with an equivalent real search engine to test the accuracy of this methodology.

5. ACKNOWLEDGEMENTS

This work has been partially supported by the Spanish Ministry of Education and Science under project TIN2009-14203.

6. REFERENCES

[1] C. S. Badue, R. Baeza-Yates, B. Ribeiro-Neto, A. Ziviani, and N. Ziviani. Analyzing imbalance among homogeneous index servers in a web search system. *Inf. Process. Manage.*, 43(3):592–608, 2007.

[2] R. A. Baeza-Yates, C. Castillo, F. Junqueira, V. Plachouras, and F. Silvestri. Challenges on distributed web retrieval. In *ICDE*, pages 6–20, 2007.

[3] F. J. Burkowski. Retrieval performance of a distributed text database utilizing a parallel processor document server. In *DPDS '90: Proceedings of the second international symposium on Databases in parallel and distributed systems*, pages 71–79, New York, NY, USA, 1990. ACM.

[4] F. Cacheda, V. Carneiro, V. Plachouras, and I. Ounis. Performance analysis of distributed information retrieval architectures using an improved network simulation model. *Inf. Process. Manage.*, 43(1):204–224, 2007.

[5] J.-H. Chen and C.-C. Chang. A moving piv system for ship model test in a towing tank. *Ocean Engineering*, 33(14-15):2025 – 2046, 2006.

[6] A. Chowdhury and G. Pass. Operational requirements for scalable search systems. In *CIKM '03: Proceedings of the twelfth international conference on Information and knowledge management*, pages 435–442, New York, NY, USA, 2003. ACM.

[7] C. L. A. Clarke, N. Craswell, and I. Soboroff. Overview of the trec 2004 terabyte track. In *TREC*, 2004.

[8] J.-H. Kang and S.-J. Lee. Improvement of natural ventilation in a large factory building using a louver ventilator. *Building and Environment*, 43(12):2132 – 2141, 2008.

[9] A. Moffat, W. Webber, J. Zobel, and R. A. Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Inf. Retr.*, 10(3):205–231, 2007.

[10] B. Ribeiro-Neto and R. A. Barbosa. Query performance for tightly coupled distributed digital libraries. In *Proceedings of the third ACM conference on Digital libraries*, pages 182–190, Pittsburgh, Pennsylvania, United States, 1998. ACM Press.

[11] N. Susanta and C. Tzi-Cker. A survey on virtualization technologies. Technical report.

[12] A. Tomasic and H. Garcia-Molina. Query processing and inverted indices in shared: nothing text document information retrieval systems. *The VLDB Journal*, 2(3):243–276, 1993.

[13] S. Zhong, M. Jabbal, H. Tang, L. Garcillan, F. Guo, N. Wood, and C. Warsop. Towards the design of synthetic-jet actuators for full-scale flight conditions. *Flow, Turbulence and Combustion*, 78(3):283 – 307, 2007.