MapReduce for information retrieval evaluation: “Let’s quickly test this on 12 TB of data”

Djoerd Hiemstra and Claudia Hauff

University of Twente, The Netherlands

Abstract. We propose to use MapReduce to quickly test new retrieval approaches on a cluster of machines by sequentially scanning all documents. We present a small case study in which we use a cluster of 15 low cost machines to search a web crawl of 0.5 billion pages showing that sequential scanning is a viable approach to running large-scale information retrieval experiments with little effort. The code is available to other researchers at: `http://mirex.sourceforge.net`

1 Introduction

A lot of research in the field of information retrieval aims at improving the *quality* of search results. Search quality might for instance be improved by new scoring functions, new indexing approaches, new query (re-)formulation approaches, etc. To make a scientific judgment of the quality of a new search approach, it is good practice to use benchmark test collections, such as those provided by CLEF [10] and TREC [13]. The following steps typically need to be taken: *1)* The researcher codes the new approach by adapting an existing search system, such as Lemur [6], Lucene [8], or Terrier [11]; *2)* The researcher uses the system to create an inverted index on the documents from the test collection; *3)* The researcher puts the queries to the experimental search engine and gathers the top X search results (a common value for CLEF experiments is $X = 1000$); *4)* The researcher compares the top X to a golden standard by computing standard evaluation measures such as mean average precision. In our experience, Step 1, actually coding the new approach, takes by far the most effort and time when conducting an information retrieval experiment. Coding new retrieval approaches into existing search engines like Lemur, Lucene and Terrier is a tedious job, even if the code is maintained by members of the same research team. It requires detailed knowledge of the existing code of the search engine, or at least, knowledge of the part of the code that needs to be adapted. Radical new approaches to information retrieval, i.e., approaches that need information that is not available from the search engines inverted index, require reimplementing part of the indexing functionality. Such radical new approaches are therefore not often evaluated, and most research is done by small changes to the system.

Of in total 36 CLEF 2009 working note papers that were submitted for the Multilingual Document Retrieval (Ad Hoc) and Intellectual Property (IP) tracks, 10 papers used Lemur, 10 used Lucene, 3 used Terrier, 1 used all three, 5 did not

mention the engine, and 7 used another engine (Cheshire II, Haircut, Open Text SearchServer, MG4J, Zettair, JIRS, and MonetDB/HySpirit). In several cases a standard engine was used, and then a lot of post-processing was done on top, completely changing the retrieval approach in the end. Post-processing on top of an existing engine tests the ability of new retrieval approaches to rerank results from the existing engine, but it does not test the ability of new approaches to find *new* information.

In his WSDM keynote lecture, Dean [3] describes how MapReduce [4] is used at Google for experimental evaluations. New ranking ideas are tested off-line on human rated query sets similar to the queries from CLEF and TREC. Running such off-line tests has to be easy for the researchers at Google, possibly at the expense of the efficiency of the prototype. So, it is okay if it takes hours to run for instance 10,000 queries, as long as the experimental infrastructure allows for fast and easy coding of new approaches. A similar experimental setup was followed by Microsoft at TREC 2009: Craswell et al. [2] use DryadLINQ [15] on a cluster of 240 machines to run web search experiments. Their setup sequentially scans all document representations, providing a flexible environment for a wide range of experiments. The researchers plan to do many more to discover its benefits and limitations.

The work at Google and Microsoft shows that sequential scanning over large document collections is a viable approach to experimental information retrieval. Some of the advantages are:

1. Researchers spend less time on coding and debugging new experimental retrieval approaches;
2. It is easy to include new information in the ranking algorithm, even if that information would not normally be included in the search engine's inverted index;
3. Researchers are able to oversee all or most of the code used in the experiment;
4. Large-scale experiments can be done in reasonable time.

We show that indeed sequential scanning is a viable experimental tool, even if only a few machines are available. In Section 2 we describe the MapReduce search system. Sections 3 and 4 contain experimental results and concluding remarks.

2 Sequential Search in MapReduce

MapReduce is a framework for batch processing of large data sets on clusters of commodity machines [4]. Users of the framework specify a *mapper* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reducer* function that processes intermediate values associated with the same intermediate key. The pseudo code in Figure 1 outlines our sequential search implementation. The implementation does a single scan of the documents, processing all queries in parallel.

```
mapper (DocId, DocText) =
  FOREACH (QueryID, QueryText) IN Queries
    Score = experimental_score(QueryText, DocText)
    IF (Score > 0)
    THEN OUTPUT(QueryId, (DocId, Score))

reducer (QueryId, DocIdScorePairs) =
  RankedList = ARRAY[1000]
  FOREACH (DocId, Score) IN DocIdScorePairs
    IF (NOT filled(RankedList) OR
      Score > smallest_score(RankedList))
    THEN ranked_insert(RankedList, (DocId, Score))
  FOREACH (DocId, Score) IN RankedList
    OUTPUT(QueryId, DocId, Score)
```

Fig. 1. Pseudo code for linear search

The *mapper* function takes pairs of *document identifier* and *document text* `(DocId, DocText)`. For each pair, it runs all benchmark queries and outputs for each matching query the *query identifier* as key, and the pair *document identifier* and *score* as value. In the code, `Queries` is a global constant per experiment. The MapReduce framework runs the mappers in parallel on each machine in the cluster. When the map step finishes, the framework groups the intermediate output per key, i.e., per `QueryId`. The *reducer* function then simply takes the top 1000 results for each query identifier, and outputs those as the final result. The reducer function is also applied locally on each machine (that is, the reducer is also used as a *combiner* [4]), making sure that at most 1000 results have to be sent between machines after the map phase finishes.

3 Case Study: ClueWeb09

The ClueWeb09 test collection consists of 1 billion web pages in ten languages, collected in January and February 2009. The dataset is used by several tracks of the TREC conference [13]. We used the English pages from the collection, about 0.5 billion pages equaling 12.5 TB (2.5 TB compressed). It is hard to handle collections of the size of ClueWeb09 on a single machine, at least one needs to have a lot of external storage space. We ran our experiments on a small cluster of 15 machines; each machine costs about €1000. The cluster runs Hadoop version 0.19.2 out of the box [14].

Time to code the experiment After gaining some experience with Hadoop by having M.Sc. students doing practical assignments, we wrote the code for sequential search, and for anchor text extraction in less than a day. Table 1 gives some idea of the size of the source code compared to that of experimental

search systems. Note that this is by no means a fair comparison: The existing systems are general purpose information retrieval systems including a lot of functionality, whereas the linear search system only knows a single trick. The table also ignores the code needed to read the web archive “warc” format used by ClueWeb09, which consists of another 4 files, 915 lines, and 30 kb of code.[1] Still, in order to adapt the systems below, one at least has to figure out what code to adapt.

Table 1. Size of code base per system

Code base	#files	#lines	size (kb)
MapReduce anchors & search	2	350	13
Terrier 2.2.1	300	59,000	2,000
Lucene 2.9.2	1,370	283,000	9,800
Lemur/Indri 4.11	1,210	540,000	19,500

Time to run the experiment Anchor text extraction on all English documents of ClueWeb09 takes about 11 hours on our cluster. The anchor text representation contains text for about 87 % of the documents, about 400 GB in total. A subsequent TREC run using 50 queries on the anchor text representation takes less than 30 minutes. Our linear search system implements a fairly simple language model with a length prior without stemming or stop words. It achieves expected precision at 5, 10 and 20 documents retrieved of respectively 0.42, 0.39, and 0.35 (MTC method), similar to the best runs at TREC 2009 [1].

Figure 2 shows how the system scales when processing up to 5,000 queries, using random sets of queries from the TREC 2009 Million Query track. Reported times are full Hadoop job times including job setup and job cleanup averaged over three trials. Processing time increases only slightly if more queries are processed. Whereas the average processing time per query is about 35 seconds per query for 50 queries, it goes down to only 1.6 second per query for 5,000 queries. For comparison, the graph shows the performance of “Lemur-one-node”, i.e., Lemur version 4.11 running on *one fourteenth* of the anchor text representation on a single machine. A distributed version of Lemur searching the full full anchor text representation would not do faster: It would be as fast as the slowest node, it would need to send results from each node to the master, and to merge the results. Lemur-one-node takes 3.3 seconds per query on average for 50 queries, and 0.44 seconds on average for 5,000 queries. The processing times for Lemur were measured after flushing the file system cache. Although Lemur cannot process queries in parallel, the system's performance benefits from receiving many queries. Lemur's performance scales sublinearly because it caches intermediate results. Still, at 5,000 queries Lemur-one-node is only 3.6 times faster than the MapReduce system. For experiments at this scale, the benefits of the full, distributed Lemur are probably negligible.

[1] The warc reader was kindly provided by Mark J. Hoy, Carnegie Mellon University.

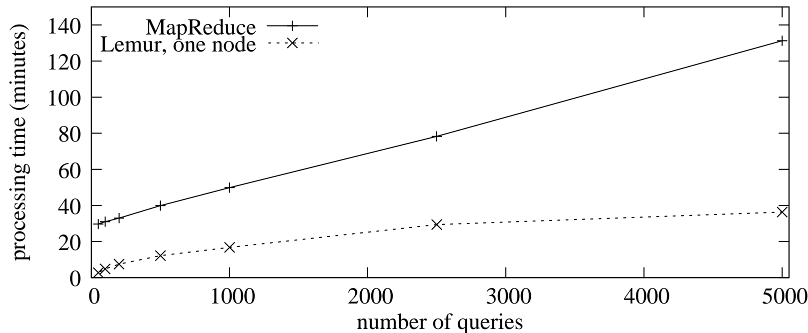

Fig. 2. Processing time for query set sizes

We also tried to index all anchor text data on a single machine and run a fraction of the queries on each of the 14 machines. Unfortunately, Lemur crashed when trying to index all data, and we did not pursue this approach any further.

On (not) computing global statistics Our language modeling approach does not use smoothing, so it does not need global statistics to compute IDF-like (inverse document frequency) weights. Global statistics can be incorporated by doing one initial pass over the corpus to collect global statistics for all queries [2]. Post-hoc experiments using standard approaches like linear interpolation smoothing, Dirichlet prior smoothing, and Okapi's BM25 [9] show that these approaches – that *do* use global statistics – consistently perform equally or worse than our simple weighting approach. Presumably, on document collections of this scale, IDF-like weighting is unnecessary.

Related work The idea to use sequential scanning of documents on many machines in parallel is certainly not new. Salton and Buckley [12] analyzed such a method for the Connection Machine, a multi-processor parallel computing machine. We also know of at least one researcher who used sequential scanning over ten years ago for his thesis [5]. Without high-level programming paradigms like MapReduce, however, efficiently implementing sequential scanning is not a trivial task, and without a cluster of machines the approach does not scale to large collections. A similar approach using MapReduce was taken by Lin [7], who used Hadoop MapReduce for computing pairwise document similarities. Our implementation resembles Lin's brute force algorithm that also scans document representations linearly. Our approach is simpler because our preprocessing step does not divide the collection into blocks, nor does it compute document vectors.

4 Conclusion

A faster turnaround of the experimental cycle can be achieved by making coding of experimental systems easier. Faster coding means one is able to do more

experiments, and more experiments means more improvement of retrieval performance. We implemented a full experimental retrieval system with little effort using Hadoop MapReduce. Using 15 machines to search a web crawl of 0.5 billion pages, the proposed MapReduce approach is less than 10 times slower than a single node of a distributed inverted index search system on a set of 50 queries. If more queries are processed per experiment, the processing times of the two systems get even more close. The code used in our experiment is open source and available to other researchers at: `http://mirex.sourceforge.net`

Acknowledgments

Many thanks to Sietse ten Hoeve, Guido van der Zanden, and Michael Meijer for early implementations of the system. The research was partly funded by the Netherlands Organization for Scientific Research, NWO, grant 639.022.809. We are grateful to Yahoo Research, Barcelona, for sponsoring our cluster.

References

1. C.L.A. Clarke, N. Craswell, and I. Soboroff. Overview of the TREC 2009 web track. In *Proceedings of the 18th Text REtrieval Conference (TREC)*. 2009.
2. N. Craswell, D. Fetterly, M. Najork, S. Robertson, and E. Yilmaz. Microsoft Research at TREC 2009: Web and relevance feedback tracks. In *Proceedings of the 18th Text REtrieval Conference (TREC)*. 2009.
3. J. Dean. Challenges in building large-scale information retrieval systems. In *Proceedings of the 2nd Conference on Web Search and Data Mining (WSDM)*, 2009.
4. J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implemention (OSDI)*, 2004.
5. D. Hiemstra. Using Language Models for Information Retrieval. *Ph.D. thesis*, 2001.
6. Lemur Toolkit. `http://www.lemurproject.org/`
7. J. Lin. Brute Force and Indexed Approaches to Pairwise Document Similarity Comparisons with MapReduce. In *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, 2009.
8. Lucene Search Engine. `http://lucene.apache.org`
9. C.D. Manning, P. Raghavan and H. Schütze. Introduction to Information Retrieval. *Cambridge University Press*, 2008.
10. C. Peters, Th. Deselaers, N. Ferro, J. Gonzalo, G.J.F. Jones, M. Kurimo, Th. Mandl, A. Peñas, and V. Petras, editors. Evaluating Systems for Multilingual and Multimodal Information Access. In *Lecture Notes in Computer Science 5706*, 2009.
11. Terrier IR Platform. `http://ir.dcs.gla.ac.uk/terrier/`
12. G. Salton and C. Buckley. Parallel text search methods. *Communications of the ACM 31*(2), 1988.
13. E.M. Voorhees and D.K. Harman, editors. *TREC Experiment and Evaluation in Information Retrieval.* MIT Press, 2008.
14. Tom White. *Hadoop: The Definitive Guide.* O'Reilly Media, 2009.
15. Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. Kumar, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th Symposium on Operating System Design and Implemention (OSDI)*, 2008.