Pairwise Document Similarity in Large Collections with MapReduce

Tamer Elsayed,* Jimmy Lin,† and Douglas W. Oard†
Human Language Technology Center of Excellence and
UMIACS Laboratory for Computational Linguistics and Information Processing
University of Maryland, College Park, MD 20742
{telsayed,jimmylin,oard}@umd.edu

Abstract

This paper presents a MapReduce algorithm for computing pairwise document similarity in large document collections. MapReduce is an attractive framework because it allows us to decompose the inner products involved in computing document similarity into separate multiplication and summation stages in a way that is well matched to efficient disk access patterns across several machines. On a collection consisting of approximately 900,000 newswire articles, our algorithm exhibits linear growth in running time and space in terms of the number of documents.

1 Introduction

Computing pairwise similarity on large document collections is a task common to a variety of problems such as clustering and cross-document coreference resolution. For example, in the PubMed search engine,[1] which provides access to the life sciences literature, a "more like this" browsing feature is implemented as a simple lookup of document-document similarity scores, computed offline. This paper considers a large class of similarity functions that can be expressed as an inner product of term weight vectors.

For document collections that fit into random-access memory, the solution is straightforward. As collection size grows, however, it ultimately becomes necessary to resort to disk storage, at which point aligning computation order with disk access patterns becomes a challenge. Further growth in the document collection will ultimately make it desirable to spread the computation over several processors, at which point interprocess communication becomes a second potential bottleneck for which the computation order must be optimized. Although tailored implementations can be designed for specific parallel processing architectures, the MapReduce framework (Dean and Ghemawat, 2004) offers an attractive solution to these challenges. In this paper, we describe how pairwise similarity computation for large collections can be efficiently implemented with MapReduce. We empirically demonstrate that removing high frequency (and therefore low entropy) terms results in approximately linear growth in required disk space and running time with increasing collection size for collections containing several hundred thousand documents.

2 MapReduce Framework

MapReduce builds on the observation that many tasks have the same structure: a computation is applied over a large number of records (e.g., documents) to generate partial results, which are then aggregated in some fashion. Naturally, the per-record computation and aggregation vary by task, but the basic structure remains fixed. Taking inspiration from higher-order functions in functional programming, MapReduce provides an abstraction that involves the programmer defining a "mapper" and a "reducer", with the following signatures:

$$\text{map: } (k_1, v_1) \rightarrow [(k_2, v_2)]$$
$$\text{reduce: } (k_2, [v_2]) \rightarrow [(k_3, v_3)]$$

Key/value pairs form the basic data structure in MapReduce. The "mapper" is applied to every input

*Department of Computer Science
†The iSchool, College of Information Studies
[1]http://www.ncbi.nlm.nih.gov/PubMed

Proceedings of ACL-08: HLT, Short Papers (Companion Volume), pages 265–268,
Columbus, Ohio, USA, June 2008.

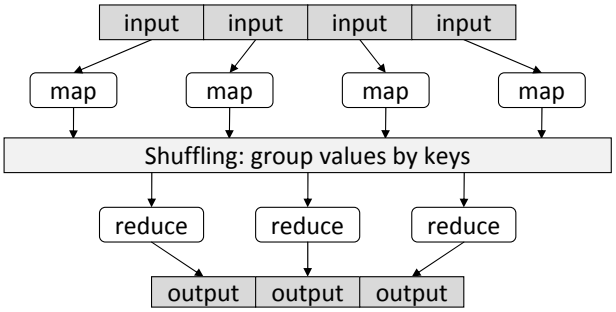

Figure 1: Illustration of the MapReduce framework: the "mapper" is applied to all input records, which generates results that are aggregated by the "reducer".

key/value pair to generate an arbitrary number of intermediate key/value pairs. The "reducer" is applied to all values associated with the same intermediate key to generate output key/value pairs (see Figure 1).

On top of a distributed file system (Ghemawat et al., 2003), the runtime transparently handles all other aspects of execution (e.g., scheduling and fault tolerance), on clusters ranging from a few to a few thousand nodes. MapReduce is an attractive framework because it shields the programmer from distributed processing issues such as synchronization, data exchange, and load balancing.

3 Pairwise Document Similarity

Our work focuses on a large class of document similarity metrics that can be expressed as an inner product of term weights. A document d is represented as a vector W_d of term weights $w_{t,d}$, which indicate the importance of each term t in the document, ignoring the relative ordering of terms ("bag of words" model). We consider symmetric similarity measures defined as follows:

$$sim(d_i, d_j) = \sum_{t \in V} w_{t,d_i} \cdot w_{t,d_j} \qquad (1)$$

where $sim(d_i, d_j)$ is the similarity between documents d_i and d_j and V is the vocabulary set. In this type of similarity measure, a term will contribute to the similarity between two documents only if it has non-zero weights in both. Therefore, $t \in V$ can be replaced with $t \in d_i \cap d_j$ in equation 1.

Generalizing this to the problem of computing similarity between *all* pairs of documents, we note that a term contributes to *each* pair that contains it.[2] For example, if a term appears in documents x, y, and z, it contributes *only* to the similarity scores between (x, y), (x, z), and (y, z). The list of documents that contain a particular term is exactly what is contained in the postings of an inverted index. Thus, by processing all postings, we can compute the entire pairwise similarity matrix by summing term contributions.

Algorithm 1 Compute Pairwise Similarity Matrix

```
1: ∀i, j : sim[i, j] ⇐ 0
2: for all t ∈ V do
3:     p_t ⇐ postings(t)
4:     for all d_i, d_j ∈ p_t do
5:         sim[i, j] ⇐ sim[i, j] + w_{t,d_i} · w_{t,d_j}
```

Algorithm 1 formalizes this idea: $postings(t)$ denotes the list of documents that contain term t. For simplicity, we assume that term weights are also stored in the postings. For small collections, this algorithm can be run efficiently to compute the entire similarity matrix in memory. For larger collections, disk access optimization is needed—which is provided by the MapReduce runtime, without requiring explicit coordination.

We propose an efficient solution to the pairwise document similarity problem, expressed as two separate MapReduce jobs (illustrated in Figure 2):

1) Indexing: We build a standard inverted index (Frakes and Baeza-Yates, 1992), where each term is associated with a list of docid's for documents that contain it and the associated term weight. Mapping over all documents, the mapper, for each term in the document, emits the term as the key, and a tuple consisting of the docid and term weight as the value. The MapReduce runtime automatically handles the grouping of these tuples, which the reducer then writes out to disk, thus generating the *postings*.

2) Pairwise Similarity: Mapping over each posting, the mapper generates key tuples corresponding to pairs of docids in the postings: in total, $\frac{1}{2}m(m-1)$ pairs where m is the posting length. These key tuples are associated with the product of the corresponding term weights—they represent the individ-

[2] Actually, since we focus on symmetric similarity functions, we only need to compute half the pairs.

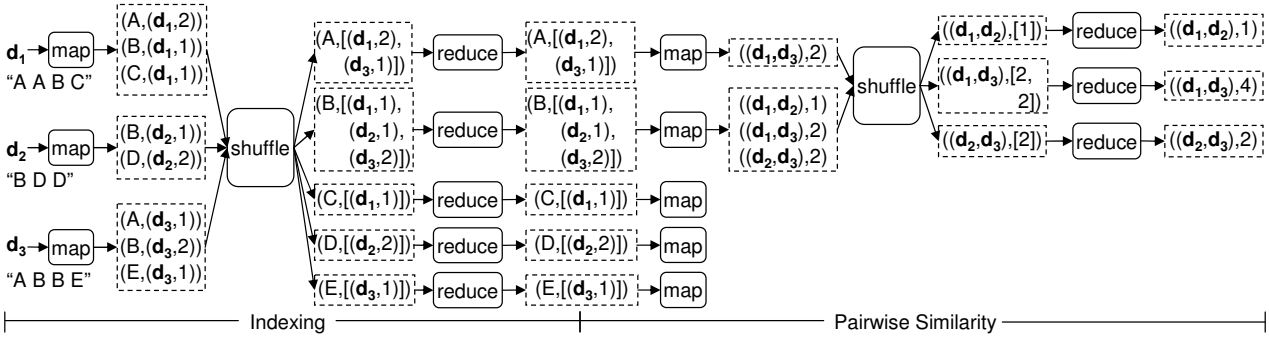

Figure 2: Computing pairwise similarity of a toy collection of 3 documents. A simple term weighting scheme ($w_{t,d} = tf_{t,d}$) is chosen for illustration.

ual term contributions to the final inner product. The MapReduce runtime sorts the tuples and then the reducer sums all the individual score contributions for a pair to generate the final similarity score.

4 Experimental Evaluation

In our experiments, we used Hadoop version 0.16.0,[3] an open-source Java implementation of MapReduce, running on a cluster with 20 machines (1 master, 19 slave). Each machine has two single-core processors (running at either 2.4GHz or 2.8GHz), 4GB memory, and 100GB disk.

We implemented the symmetric variant of Okapi-BM25 (Olsson and Oard, 2007) as the similarity function. We used the AQUAINT-2 collection of newswire text, containing 906k documents, totaling approximately 2.5 gigabytes. Terms were stemmed. To test the scalability of our technique, we sampled the collection into subsets of 10, 20, 25, 50, 67, 75, 80, 90, and 100 percent of the documents.

After stopword removal (using Lucene's stopword list), we implemented a *df*-cut, where a fraction of the terms with the highest document frequencies is eliminated.[4] This has the effect of removing non-discriminative terms. In our experiments, we adopt a 99% cut, which means that the most frequent 1% of terms were discarded (9,093 terms out of a total vocabulary size of 909,326). This technique greatly increases the efficiency of our algorithm, since the number of tuples emitted by the mappers in the pairwise similarity phase is dominated by the length of the longest posting (in the worst case, if a term appears in all documents, it would generate approximately 10^{12} tuples).

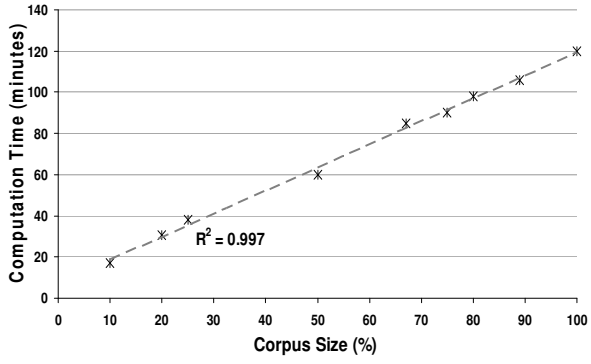

Figure 3: Running time of pairwise similarity comparisons, for subsets of AQUAINT-2.

Figure 3 shows the running time of the pairwise similarity phase for different collection sizes.[5] The computation for the entire collection finishes in approximately two hours. Empirically, we find that running time increases linearly with collection size, which is an extremely desirable property. To get a sense of the space complexity, we compute the number of intermediate document pairs that are emitted by the mappers. The space savings are large (3.7 billion rather than 8.1 trillion intermediate pairs for the entire collection), and space requirements grow linearly with collection size over this region ($R^2 = 0.9975$).

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

[4] In text classification, removal of rare terms is more common. Here we use *df*-cut to remove common terms.

[5] The entire collection was indexed in about 3.5 minutes.

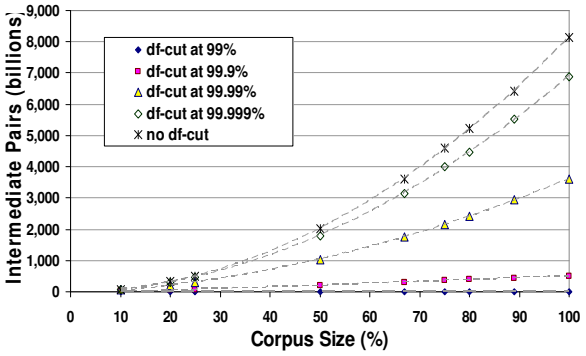

Figure 4: Effect of changing *df*-cut thresholds on the number of intermediate document-pairs emitted, for subsets of AQUAINT-2.

5 Discussion and Future Work

In addition to empirical results, it would be desirable to derive an analytical model of our algorithm's complexity. Here we present a preliminary sketch of such an analysis and discuss its implications. The complexity of our pairwise similarity algorithm is tied to the number of document pairs that are emitted by the mapper, which equals the total number of products required in $O(N^2)$ inner products, where N is the collection size. This is equal to:

$$\frac{1}{2}\sum_{t \in V} df_t(df_t - 1) \qquad (2)$$

where df_t is the document frequency, or equivalently the length of the postings for term t. Given that tokens in natural language generally obey Zipf's Law, and vocabulary size and collection size can be related via Heap's Law, it may be possible to develop a closed form approximation to the above series.

Given the necessity of computing $O(N^2)$ inner products, it may come as a surprise that empirically our algorithm scales linearly (at least for the collection sizes we explored). We believe that the key to this behavior is our *df*-cut technique, which eliminates the head of the *df* distribution. In our case, eliminating the top 1% of terms reduces the number of document pairs by several orders of magnitude. However, the impact of this technique on effectiveness (e.g., in a query-by-example experiment) has not yet been characterized. Indeed, a *df*-cut threshold of 99% might seem rather aggressive, removing meaning-bearing terms such as "arthritis" and "Cornell" in addition to perhaps less problematic terms such as "sleek" and "frail." But redundant use of related terms is common in news stories, which we would expect to reduce the adverse effect on many applications of removing these low entropy terms.

Moreover, as Figure 4 illustrates, relaxing the *df*-cut to a 99.9% threshold still results in approximately linear growth in the requirement for intermediate storage (at least over this region).[6] In essence, optimizing the *df*-cut is an efficiency vs. effectiveness tradeoff that is best made in the context of a specific application. Finally, we note that alternative approaches to similar problems based on locality-sensitive hashing (Andoni and Indyk, 2008) face similar tradeoffs in tuning for a particular false positive rate; cf. (Bayardo et al., 2007).

6 Conclusion

We present a MapReduce algorithm for efficiently computing pairwise document similarity in large document collections. In addition to offering specific benefits for a number of real-world tasks, we also believe that our work provides an example of a programming paradigm that could be useful for a broad range of text analysis problems.

Acknowledgments

This work was supported in part by the Intramural Research Program of the NIH/NLM/NCBI.

References

A. Andoni and P. Indyk. 2008. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *CACM*, 51(1):117–122.

R. Bayardo, Y. Ma, and R. Srikant. 2007. Scaling up all pairs similarity search. In *WWW '07*.

J. Dean and S. Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. In *OSDI '04*.

W. Frakes and R. Baeza-Yates. 1992. *Information Retrieval: Data Structures and Algorithms*.

S. Ghemawat, H. Gobioff, and S. Leung. 2003. The Google File System. In *SOSP '03*.

J. Olsson and D. Oard. 2007. Improving text classification for oral history archives with temporal domain knowledge. In *SIGIR '07*.

[6] More recent experiments suggest that a *df*-cut of 99.9% results in almost no loss of effectiveness on a query-by-example task, compared to no *df*-cut.