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

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

ABSTRACT

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

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

General Terms: Algorithms, Performance

1. INTRODUCTION

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

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

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

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

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

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

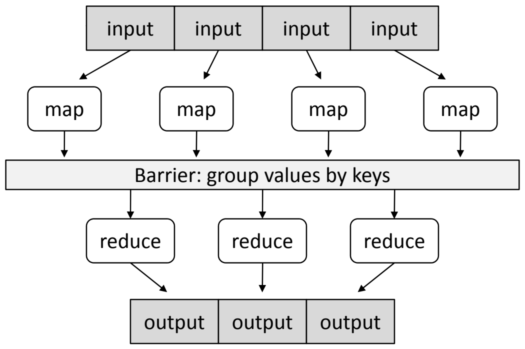

Figure 1: Illustration of MapReduce: "mappers" are applied to input records, which generate intermediate results that are aggregated by "reducers".

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

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

2. A THEORETICAL ANALYSIS

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

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

where s is the characteristic exponent. The denominator is known as the Nth generalized harmonic number, often written as $H_{N,s}$. For many types of algorithms, all occurrences of the same type of element must be processed together. That is, the element type represents the finest grain of parallelism that can be achieved. Let us assume that the amount of "work" associated with processing a type of element is

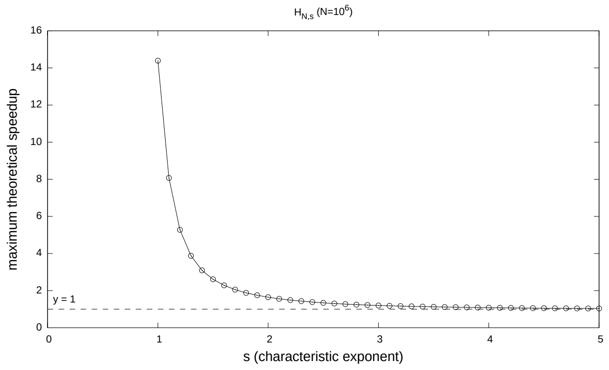

Figure 2: Plot of $H_{N,s}$, the Nth generalized harmonic number with $N = 10^6$, based on different values of s, the characteristic exponent of the Zipfian distribution. This plot shows the maximum theoretical speedup of a parallel algorithm where all instances of an element type must be processed together.

$O(n)$ with respect to the number of element instances. In other words, the amount of work necessary to process an element is proportional to its frequency. If we normalize the total amount of work to one, then the fraction of the total work that must be devoted to processing the most common element is:

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

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

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

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

Inverted indexing. The well-known MapReduce algorithm for inverted indexing begins by mapping over input documents and emitting individual postings as intermediate key-value pairs. All postings associated with each term are gathered in the reducer, where the final postings lists are created and written to disk. The most straightforward implementation of the algorithm divides the term space into roughly equal-sized partitions and assigns each to a reducer (typically, through hashing). This, however, does not take into account the inherent distribution of document frequencies (which characterizes the length of each postings list, and hence the amount of work that needs to be done for each term). Due to the Zipfian distribution of term occurrences, the reducers assigned to very frequent terms will have significantly more work than the other reducers, and therefore take much longer to complete. These are the stragglers observed in the execution of the algorithm.

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

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

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

3. PARALLEL QUERIES: A CASE STUDY

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

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

```
1: procedure Map(Term $t$, Postings $P$)
2:     $[Q_1, Q_2, \dots Q_n] \leftarrow$ LoadQueries()
3:     for all $Q_i \in [Q_1, Q_2, \dots Q_n]$ do
4:         if $t \in Q_i$ then
5:             Initialize.AssociativeArray($H$)
6:             for all $\langle a, f \rangle \in P$ do
7:                 $H\{a\} \leftarrow w_{t,q} \cdot w_{t,d}$
8:             Emit($i, H$)

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

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

3.1 Background

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

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

3.2 Experimental Results

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

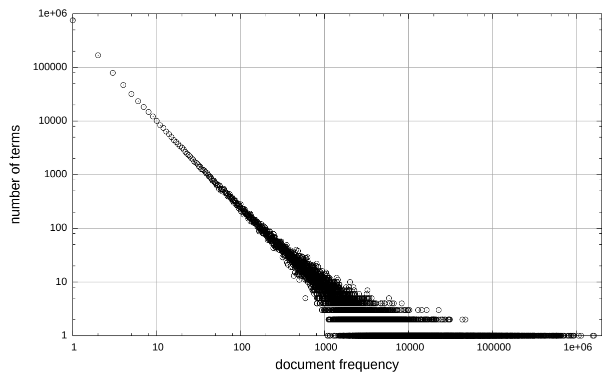

Figure 4: Distribution of document frequencies of terms in the MEDLINE collection.

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

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

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

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

- At the 467s mark, all except for 10 mappers (2% of the total) have completed.

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

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

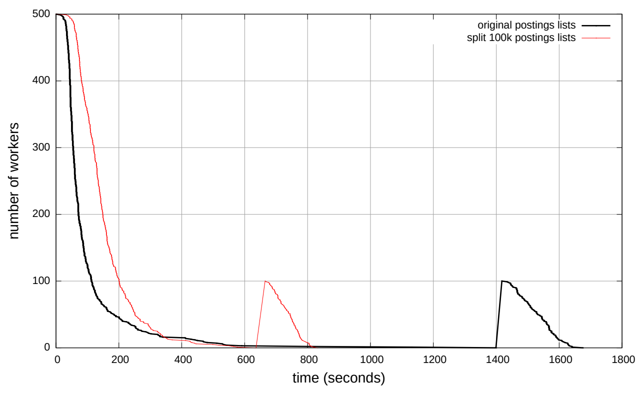

Figure 5: Progress of the MapReduce parallel queries algorithm (472 full-abstract "queries" on the MEDLINE collection), comparing original postings lists (thick black line) and postings lists split into 100k segments (thin red line).

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

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

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

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

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

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

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

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

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

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

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

4. DISCUSSION

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

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

For solving reduce-phase stragglers, once the distribution of intermediate data has been characterized, a more intelligent partitioner can better distribute load across the reducers. Unfortunately, this may still be insufficient in some cases, for example, PageRank computations. Recall that in computing PageRank all contributions from inbound links must be summed, thereby creating the requirement that all values associated with the same key (i.e., vertex in graph) be processed together. The only recourse here is a modification of the original algorithm (e.g., a multi-stage process for totaling the PageRank contributions of pages with large in-degrees).

The upshot of this discussion is that to overcome the efficiency bottleneck imposed by Zipfian distributions, developers must apply *application-specific* knowledge to parallelize the processing of common elements. This, in turn, depends on the ingenuity of the individual developer and requires insight into the problem being tackled.

5. CONCLUSION

This paper explores the stragglers problem in MapReduce caused by Zipfian distributions common in many real-world datasets. What are the broader implications of these findings? I believe that two lessons are apparent:

- First, efficient MapReduce algorithms are not quite as easy to design as one might think. The allure of MapReduce is that it presents a simple programming model for designing scalable algorithms. While this remains an accurate statement, the deeper truth is that there is still quite a bit of "art" in designing efficient MapReduce algorithms for non-toy problems—witness the case study presented in this paper, where a simple tweak to the input data cut running time in half.[5]

- Second, MapReduce algorithms cannot be studied in isolation, divorced from real-world applications—the stragglers problem is caused by properties of datasets (critically, not from the processing architecture or inherent bottlenecks in the algorithm). Furthermore, since there does not appear to be a general-purpose, universally-applicable solution to the problem, it is of limited value to discuss algorithmic performance without reference to specific applications.

Hopefully, these two lessons will be useful to future designers of MapReduce algorithms.

6. ACKNOWLEDGMENTS

This work was supported by the following sources: the Intramural Research Program of the NIH, National Library of Medicine; NSF under awards IIS-0705832 and IIS-0836560 (under the CLuE program); Google and IBM, via the Academic Cloud Computing Initiative. I am grateful to Esther and Kiri for their love.

[5]Although one might argue how obvious this solution is, I think it is safe to say that the solution requires a degree of understanding of both information retrieval algorithms and MapReduce that a novice would be unlikely to have.

7. REFERENCES

[1] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 483–485, 1967.

[2] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 2004)*, pages 137–150, San Francisco, California, 2004.

[3] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 29–43, Bolton Landing, New York, 2003.

[4] W. R. Hersh, A. Cohen, J. Yang, R. Bhupatiraju, P. Roberts, and M. Hearst. TREC 2005 Genomics Track overview. In *Proceedings of the Fourteenth Text REtrieval Conference (TREC 2005)*, Gaithersburg, Maryland, 2005.

[5] J. Lin. Scalable language processing algorithms for the masses: A case study in computing word co-occurrence matrices with mapreduce. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing (EMNLP 2008)*, pages 419–428, Honolulu, Hawaii, 2008.

[6] J. Lin. Brute force and indexed approaches to pairwise document similarity comparisons with mapreduce. In *Proceedings of the 32nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2009)*, Boston, Massachusetts, 2009.

[7] M. Newman. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics*, 46(5):323–351, 2005.

[8] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the Web. Stanford Digital Library Working Paper SIDL-WP-1999-0120, Stanford University, 1999.