Brute Force and Indexed Approaches to Pairwise Document Similarity Comparisons with 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 computing pairwise similarity on document collections, focusing on the application of "more like this" queries in the life sciences domain. Three MapReduce algorithms are introduced: one based on brute force, a second where the problem is treated as large-scale *ad hoc* retrieval, and a third based on the Cartesian product of postings lists. Each algorithm supports one or more approximations that trade effectiveness for efficiency, the characteristics of which are studied experimentally. Results show that the brute force algorithm is the most efficient of the three when exact similarity is desired. However, the other two algorithms support approximations that yield large efficiency gains without significant loss of effectiveness.

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

General Terms: Algorithms, Performance

1. INTRODUCTION

Computing pairwise similarity on document collections is a task common to a variety of problems such as clustering, unsupervised learning, and text retrieval. This work explores "more like this" queries in the context of PubMed®, a Web-based search engine for the life sciences literature maintained by the National Library of Medicine (NLM). PubMed provides access to MEDLINE®, an authoritative collection of approximately 17 million bibliographic records (abstracts and metadata). Whenever the user views an abstract in PubMed (e.g., as the result of a search), the right hand side of the interface is automatically populated with titles of related articles (and appropriate links), computed using a probabilistic content-similarity algorithm [10]. Previous work [9] has shown the usefulness of this feature as a browsing tool. To cope with the high volume of abstract views, related article suggestion is implemented as a simple lookup of similarity scores computed offline.

To get a sense of the scale of the pairwise similarity comparison problem, consider a naïve approach: As a preliminary experiment, Indri was used "out of the box" to index a collection of 4.59m MEDLINE abstracts. Treating the entire text of each abstract as a "query", retrieving the top 100 hits takes about a minute per abstract on a laptop with a dual-core 2.6 GHz processor and a single hard drive in Windows XP. The number of cores is unimportant since disk access is the bottleneck. Extrapolating from this naïve solution, pairwise similarity comparison on this collection would take 77k machine hours. This naïve approach is slow primarily due to the tremendous number of disk seeks required to look up postings. Furthermore, postings are repeatedly retrieved for common terms and then discarded, resulting in a tremendous amount of wasted effort.

It would be desirable to leverage multiple machines in a cluster to tackle this problem. The challenge, however, lies in organizing and coordinating such distributed computations. This paper explores three different algorithms for pairwise similarity comparisons with MapReduce, a framework for large-scale distributed computing. All three algorithms convert similarity computations into sequential operations over large files, thereby reducing disk seeks: one algorithm is based on a brute force approach (Section 3.1), while two algorithms operate on an inverted index (Section 3.2). All three algorithms support one or more approximations that provide a tradeoff between efficiency and effectiveness, explored in Section 4. Experiments show that the brute force algorithm is the most efficient if exact similarity is desired. The indexed approaches, on the other hand, support approximations that yield increased efficiency without significant loss of effectiveness. A discussion of the brute force and indexed approaches in Section 5 is couched within a well-known tradeoff in distributed computing: gains from parallelization vs. cost of coordination.

2. MAPREDUCE AND HADOOP

The only practical solution to large data problems today is to distribute computations across multiple machines. With traditional parallel programming models (e.g., MPI), the developer shoulders the burden of explicitly managing concurrency. As a result, a significant amount of attention must be devoted to managing system-level details. MapReduce [4] presents an attractive alternative: its functional abstraction provides an easy-to-understand model for designing scalable, distributed algorithms.

MapReduce builds on the observation that many information processing tasks have the same basic structure: a computation is applied over a large number of records (e.g., Web pages) to generate partial results, which are then aggregated in some fashion. Taking inspiration from higher-order func-

SIGIR'09, July 19–23, 2009, Boston, Massachusetts, USA.

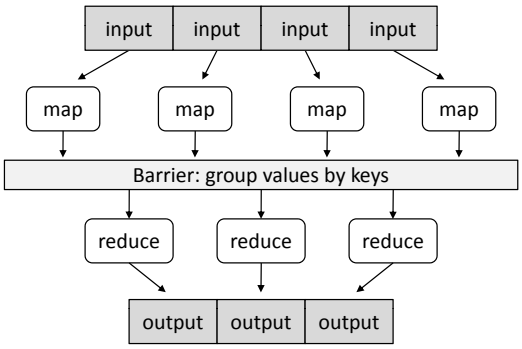

Figure 1: Illustration of MapReduce: "mappers" are applied to input records, which generate intermediate results that are aggregated by "reducers".

tions in functional programming, MapReduce provides an abstraction for programmer-defined "mappers" (that specify the per-record computation) and "reducers" (that specify result aggregation). 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 this framework, a programmer needs only to provide implementations of the mapper and reducer. On top of a distributed file system [6], the runtime transparently handles all other aspects of execution on clusters ranging from a few to a few thousand cores. The runtime is responsible for scheduling, coordination, and handling faults. In the distributed file system, data blocks are stored on the local disks of machines in the cluster—the MapReduce runtime attempts to schedule mappers on machines where the necessary data resides, thus moving the code to the data. The runtime also manages the potentially very large sorting problem between the map and reduce phases whereby intermediate key-value pairs must be grouped by key.

This paper uses Hadoop, an open-source implementation of the MapReduce programming model, and HDFS, an open-source distributed file system that provides the underlying storage substrate. As a clarification, MapReduce is used to refer to the computational model, and Hadoop to refer to the specific open-source implementation.

3. ALGORITHMS

This work focuses on the class of document similarity metrics that can be expressed as an inner product of term weights. Under a "bag of words" model, a document d is represented as a vector of term weights $w_{t,d}$. The similarity score between two documents is computed as follows:

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

where V is the vocabulary. Since a term will contribute to the similarity score only if it has non-zero weights in both documents, $t \in V$ can be replaced with $t \in d_i \cap d_j$ above.[1]

[1]This formulation can be straightforwardly extended to language modeling approaches; see [15] for a sample derivation.

```
1: procedure MAP(a, d)
2:     [⟨b_1, e_1⟩, ⟨b_2, e_2⟩, ... ⟨b_n, e_n⟩] ← LOADDOCUMENTS()
3:     for all ⟨b, e⟩ ∈ [⟨b_1, e_1⟩, ⟨b_2, e_2⟩, ... ⟨b_n, e_n⟩] do
4:         s ← COMPUTESCORE(d, e)
5:         if s > 0 then
6:             EMIT(b, ⟨a, s⟩)
```

```
1: procedure REDUCE(b, [⟨a_1, s_1⟩, ⟨a_2, s_2⟩ ...])
2:     INITIALIZE.PRIORITYQUEUE(Q)
3:     for all ⟨a, s⟩ ∈ [⟨a_1, s_1⟩, ⟨a_2, s_2⟩ ...] do
4:         if Q.SIZE() < k then
5:             Q.INSERT(a, s)
6:         else if s > Q.MIN() then
7:             Q.EXTRACTMIN()
8:             Q.INSERT(a, s)
9:     EMIT(b, Q.EXTRACTALL())
```

Figure 2: Pseudo-code of brute force (BF) algorithm for pairwise similarity in MapReduce.

The problem of computing pairwise document similarity can be succinctly defined as follows: for every document d in collection C, compute the top k ranking for similar documents according to a particular term weighting model. Typically, k is small; in the case of the PubMed application, $k = 5$ since each abstract view is populated with five related article suggestions. Note that this formulation is different from the all pairs similarity search problem defined by Bayardo et al. [3], where the goal is to find all document pairs above a similarity threshold. Here, accurate document rankings are desired, making the problem more difficult.

This section presents three MapReduce algorithms for solving the pairwise similarity problem. Because Hadoop is primarily designed for batch-oriented processing, it does not presently support low-latency random disk access. Therefore, it is not possible to quickly "look up" postings for a term, a functionality central to nearly all retrieval systems. Instead, the three algorithms organize similarity computations into sequential operations over large files.

3.1 Brute Force

As a baseline, one might consider a brute force (BF) approach: simply compute the inner product of every document vector with every other document vector. This approach proceeds as follows: First, in a preprocessing step, divide the collection into blocks and compute document vectors (tokenization, stopword removal, etc.). Then, the algorithm shown in Figure 2 is applied. Inside each mapper, pairs each consisting of a document id and a document vector $\langle b, e \rangle$[2] (for a particular block) are loaded in memory. The mapper computes the similarity score of all documents in the block with a single document d, emitting non-zero scores as intermediate key-value pairs. Since the mapper is applied to all documents in the collection, intermediate output will contain scores for all documents in the block with the entire collection. In the reducer, all scores associated with a document id are brought together; the top k results are stored in a priority queue and then emitted as the final output. Pairwise similarity for the entire collection can be computed by running this algorithm over all blocks in the collection. The block size is limited by the memory available to each mapper for holding the $\langle b, e \rangle$ pairs.

The brute force approach can be optimized by reducing

[2]In this paper, $\langle \ldots \rangle$ denotes a tuple, and $[\ldots]$ denotes a list.

```
1: procedure MAP(a, d)
2:     INITIALIZE.ASSOCIATIVEARRAY(H)
3:     for all t ∈ d do
4:         H{t} ← H{t} + 1
5:     for all t ∈ H do
6:         EMIT(t, ⟨a, H{t}⟩)
1: procedure REDUCE(t, [⟨a1, f1⟩, ⟨a2, f2⟩ ...])
2:     INITIALIZE.LIST(P)
3:     for all ⟨a, f⟩ ∈ [⟨a1, f1⟩, ⟨a2, f2⟩ ...] do
4:         APPEND(P, ⟨a, f⟩)
5:     SORT(P)
6:     EMIT(t, P)
```

Figure 3: Pseudo-code of the inverted indexing algorithm in MapReduce.

the number of intermediate key-value pairs that are generated. In Hadoop, mappers can preserve state across different input $\langle a, d\rangle$ pairs. Therefore, instead of emitting all non-zero document scores between every d and every e, the mapper can keep track of top k scoring documents for every e (with a priority queue). Once the mapper is finished processing all $\langle a, d\rangle$ input pairs, the contents of the priority queue can then be emitted as intermediate key-value pairs. This in essence moves part of the reducer functionality into the mapper, dramatically reducing the size of the intermediate output and the time spent shuffling key-value pairs across the network. Experiments in this paper use this optimized version of the brute force algorithm.

3.2 Using an Inverted Index

Instead of directly processing the text collection, the problem of pairwise similarity comparisons can be solved by first building an inverted index and then using it for the similarity computations. The pseudo-code for building a standard inverted index with MapReduce is shown in Figure 3 (cf. [4]). Input to the indexer consists of document ids (keys) and content (values). In each mapper, the document text is first tokenized. Stemmed term occurrences are stored in a histogram H. After this histogram has been built, the mapper then iterates over all terms. For each term, a pair consisting of the document id and the term frequency is created. Each pair, denoted by $\langle a, H\{t\}\rangle$ in the pseudo-code, represents an individual posting. The mapper then emits an intermediate key-value pair with the term as the key and the posting as the value. The reducer gathers up all postings, sorts them by term frequency, and emits the complete postings list. The final key-value pairs (terms and associated postings lists) make up the inverted index. Typically, computing term weights also requires information about document lengths, which is straightforwardly expressed as another MapReduce algorithm (not shown here).

This paper introduces two algorithms for solving the pairwise similarity comparison problem with an inverted index: parallel queries (PQ) and postings Cartesian product (PCP). These two algorithms are described in turn.

The first algorithm, parallel queries (PQ), is intuitively simple: treat pairwise similarity comparisons as a very large *ad hoc* retrieval problem. This proceeds as follows: First, divide up the entire collection into individual blocks of documents; these are treated as blocks of "queries". For each document block, apply the parallel retrieval algorithm shown in Figure 4. The input to each mapper is a term t (the key)

```
1: procedure MAP(t, P)
2:     [Q1, Q2, ... Qn] ← LOADQUERIES()
3:     for all Qi ∈ [Q1, Q2, ... Qn] do
4:         if t ∈ Qi then
5:             INITIALIZE.ASSOCIATIVEARRAY(H)
6:             for all ⟨a, f⟩ ∈ P do
7:                 H{a} ← wt,q · wt,d
8:             EMIT(i, H)
1: procedure REDUCE(i, [H1, H2, H3, ...])
2:     INITIALIZE.ASSOCIATIVEARRAY(Hf)
3:     for all H ∈ [H1, H2, H3, ...] do
4:         MERGE(Hf, H)
5:     EMIT(i, Hf)
```

Figure 4: Pseudo-code of the parallel queries (PQ) algorithm in MapReduce.

```
1: procedure MAP(t, P)
2:     for all ⟨ai, fi⟩ ∈ P do
3:         INITIALIZE.ASSOCIATIVEARRAY(H)
4:         for all ⟨aj, fj⟩ ∈ P do
5:             if ai ≠ aj then
6:                 H{aj} ← wt,ai · wt,aj
7:         EMIT(ai, H)
1: procedure REDUCE(ai, [H1, H2, H3, ...])
2:     INITIALIZE.ASSOCIATIVEARRAY(Hf)
3:     for all H ∈ [H1, H2, H3, ...] do
4:         MERGE(Hf, H)
5:     EMIT(ai, Hf)
```

Figure 5: Pseudo-code of the postings Cartesian product (PCP) algorithm in MapReduce.

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.[3] 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.

The second algorithm, postings Cartesian product (PCP),

[3]Note that the original query document is also scored, but a simple filter takes care of this.

computes pairwise similarity directly from the postings. This approach, shown in Figure 5, is a refinement of the algorithm described in [5]. The basic idea is to generate partial score contributions for a particular term by taking the Cartesian product of each postings list with itself; the resulting document pairs account for all similarity comparisons in which that term contributes. In the pseudo-code: the input to each mapper is a term t and its associated postings list P. The mapper implements an outer loop (iterating over $\langle a_i, f_i \rangle$) and an inner loop (iterating over $\langle a_j, f_j \rangle$); for each posting in the outer loop, partial score contributions for each posting in the inner loop are stored in an associative array (serving as accumulators). This associative array, which holds all partial scores for documents in the collection with respect to a_i for term t, is emitted as the intermediate result. The reducer is exactly the same as the PQ algorithm in Figure 4; it performs an element-wise sum of all the associative arrays, adding up term contributions across documents.

In practical terms, the PCP algorithm as described in Figure 5 cannot be directly implemented. It attempts to process the entire collection at once, and generates far too much intermediate output. Instead, pairwise similarity is computed on a block of documents at a time, by modifying the outer loop in line 2 of the mapper. Assuming documents are numbered sequentially, only those that fall within a range of ids (corresponding to the current block) are considered. The effect of this implementation is that the PCP and PQ algorithms generate the same intermediate output: the restrictions imposed on the outer loop in the PCP algorithm serve the same purpose as lines 2–4 in the PQ mapper (Figure 4). The differences in control logic, however, have an effect on algorithm running time. More importantly, the two algorithms support different approximations (see below).

3.3 Approximations

To improve the efficiency of the algorithms described in this section, three approximations are introduced that reduce the amount of computations required. The first controls the number of accumulators (implemented as associative arrays) in the PQ and PCP algorithms. In both, an accumulator is created for every posting, which results in a tremendous amount of data that needs to be shuffled between the map and reduce stages. This is inefficient because in practice only a small number of highly-similar documents are desired (five in the specific PubMed application).

Methods for limiting accumulators have received much attention in the literature [11, 12, 2, 8, 13], but these techniques cannot be easily adapted to the algorithms presented here. Nearly all techniques for accumulator management are based on an ordering of query terms, in query-at-a-time approaches (e.g., evaluating query terms in increasing *df*), or considering all query terms at once for a particular document, in document-at-a-time approaches. In both the PQ and PCP algorithms, postings are processed independently in parallel, so there is no mechanism to communicate score contributions from different query terms. Crucially, since document scores are not computed until the reduce stage, algorithms that depend on knowing the current score of the kth ranking document [2, 13] are not applicable.

Nevertheless, recognizing the importance of accumulator management, both PQ and PCP implementations adopt a simple hard limit strategy—query processing stops as soon as a predetermined accumulator limit has been reached. The postings are frequency-sorted, so the accumulators to a first approximation hold the documents with the largest contributions. It is noted that this approach has a number of well-known issues, such as stopping in the middle of a run of postings with the same *tf*. However, experiments show that this simple approach works well, trading a minor decrease in effectiveness for great gains in efficiency.

With the BF and PQ algorithms, document similarity scores can be approximated by considering only the top n terms in a document (referred to as a term limit). There are many different ways to select terms, but as an initial implementation this work explored selection based on document frequency. For the PQ algorithm, term limit is implemented as a filtering step when queries are initially loaded into memory; only the n terms with the lowest document frequencies are retained for pairwise similarity computation. For the BF algorithm, the term limit is similarly applied to the document vectors loaded into memory for each block.

The final approximation, applicable to the PQ and PCP algorithms, is a *df* limit, where all terms with document frequencies greater than a threshold are simply ignored. Since both algorithms map over postings lists, this is easy to implement as a check on *df*. The idea behind this approximation is that terms with high *df* generate a disproportionate amount of intermediate data relative to their score contributions, and therefore a global *df* limit would yield greater efficiency at an acceptable cost in effectiveness.

4. EXPERIMENTS

All three MapReduce algorithms described in the previous section, as well as the three approximations, have been implemented in Java for Hadoop. The actual PubMed term weighting model [10] was used to mirror the motivating application as closely as possible. Since the term weighting model is unimportant for the purposes of this work, the reader is referred to the cited publication for details. This section first examines the impact of the approximations on effectiveness, and then presents efficiency results.

Experiments were run on a large cluster of approximately 480 machines provided by Google and managed by IBM, shared among a few universities. Each machine has two single-core processors (2.8 GHz), 4 GB memory, and two 400 GB hard drives. The entire software stack (down to the operating system) is virtualized; each physical machine runs one virtual machine hosting Linux. Experiments used Java 1.5 and Hadoop version 0.17.3. To estimate the performance of each individual processor core on the cluster machines, the BF algorithm (applied to a small subset of abstracts) was used as a benchmark. These results were compared against an equivalent run on the laptop whose specifications are described in the introduction—whereas mappers ran in parallel on the cluster, the same exact number of mappers ran sequentially on the laptop (over the same data). Comparison of cumulative running times suggests that a single core in the laptop is approximately twice as fast as a single core in each cluster machine.

4.1 Effectiveness

To assess the characteristics of the accumulator, term, and *df* limits, effectiveness experiments were conducted with the test collection from the TREC 2005 genomics track [7], which used a ten-year subset of MEDLINE (1994–2003) containing 4.59m records. For the evaluation, fifty topics were

developed based on interviews with biologists and can be considered representative of real needs.

The genomics test collection was adapted for evaluating the quality of pairwise similarity comparisons by treating each relevant abstract as a test abstract. Abstracts relevant to the same topic were assumed to be relevant to the test abstract. The underlying relevance model is that if a user were examining a MEDLINE abstract in response to an information need, it would be desirable for the system to present related articles that were also relevant to the information need. Across 49 topics (one had no relevant abstracts), there were a total of 4584 (topic, relevant abstract) pairs. The mean number of relevant abstracts per topic is 94 with a median of 35 (minimum, 2; maximum, 709).

Precision at five (P5) and precision at twenty (P20) were selected for evaluating the quality of the results. Since the task is precision-oriented, precision at a fixed cutoff is appropriate. The selected cutoff values are meaningful: Since each abstract view in PubMed displays the top five related articles, P5 directly quantifies the user's browsing experience. Each result page in PubMed displays twenty hits, so P20 reflects the quality of a typical result page. For both P5 and P20, micro- and macro-averaged values were computed. Since each relevant abstract served as a test, the number of tests varied across topics. Micro-averaged scores were directly computed from all test abstracts, thus placing greater emphasis on topics with many relevant abstracts. Macro-averaged scores, on the other hand, were arrived at by first computing the average on test abstracts within a topic, and then averaging per-topic scores. This places equal emphasis on each topic, but the contributions of test abstracts are greater for topics with fewer relevant abstracts.

Figure 6 shows the impact that different approximations have on effectiveness.[4] The top graph shows the effects of the accumulator limit, applicable to both the PQ and PCP algorithms. The right edge of the graph shows effectiveness without accumulator limits (i.e., exact similarity), since the most common term occurs in ∼1.6m documents. For this and all subsequent experiments, the Wilcoxon signed rank test was applied to assess statistical significance with respect to the exact similarity condition. Following standard notation, ** next to a data point in the graph indicates significance with $p < 0.01$; * indicates significance with $p < 0.05$; unmarked points indicate no significant differences. Macro P20 appears to be more sensitive to the accumulator limit, but for the other three metrics, a limit of 80k does not result in a significant loss of precision and represents a good setting. With accumulator limit of 80k, the decrease in micro P5 is 1.0% and 1.3% for macro P5 (both *n.s.*).

The middle graph examines the impact of the term limit, applicable to the BF and PQ algorithms. Micro and macro P5 values (top and bottom sets of lines) are shown for an accumulator limit of 80k (circles) and no accumulator limit (square), i.e., exact similarity. Due to space constraints, P20 is omitted. For reference, each relevant abstract averages 127 terms after stopword removal. The closeness of the circle and square lines shows that the accumulator limit has minimal impact on precision. Significant differences (compared to exact similarity) are annotated in the same way as the top graph. A term limit of 80 appears to be a good setting, since that setting does not result in a significant

[4]To facilitate readability, full-size versions of all graphs in this paper are available from the author's homepage.

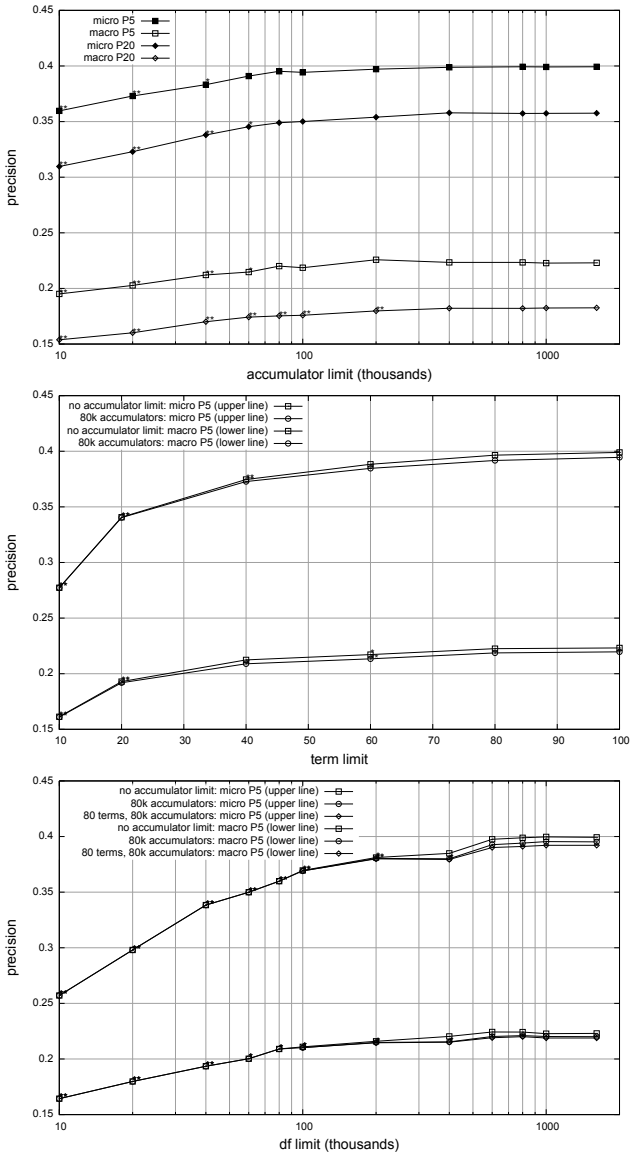

Figure 6: Effectiveness-efficiency tradeoffs for accumulator limit (top), term limit (middle), and *df* limit (bottom).

loss of precision compared to exact similarity. With a term limit of 80 and an accumulator limit of 80k, there is an 1.9% decrease in both micro and macro P5 (both *n.s.*).

The bottom graph examines the impact of the *df* limit, applicable to the PQ and PCP algorithms. Micro and macro P5 values are shown for no accumulator limit (squares), 80k accumulators (circles), and 80k accumulators with an 80-term limit (diamonds). Once again, the closeness of the precision curves shows that the previously-discussed approximations have minimal impact on effectiveness. With a *df* limit of 600k, decreases in precision (compared to exact similarity) are not significant—this represents a good tradeoff point. With a *df* limit of 600k and 80k accumulators, there is a 1.7% decrease in micro P5 and 1.2% decrease in macro P5 from the exact similarity condition (both *n.s.*). Imposing an additional 80-term limit further decreases micro and macro P5 scores by 0.6% and 0.5%, respectively (both *n.s.*).

4.2 Efficiency

Having characterized the impact on effectiveness of the accumulator, term, and *df* limits, the efficiency gains for these approximations can now be meaningfully measured. These efficiency experiments used the same collection of 4.59m MEDLINE abstracts as the effectiveness experiments.

As discussed in Section 3, it is not feasible to compute pairwise similarity on the entire collection at once. Each algorithm divides up the collection into blocks: the BF algorithm loads a block of document vectors into memory and maps over the entire collection; the PQ algorithm loads a block of queries into memory and maps over the postings; the PCP algorithm has an outer loop that controls the documents that are considered for each block. To facilitate meaningful comparison, a block size of 10k was adopted for all three algorithms (dividing the entire collection into 459 blocks). Due to the running times of the algorithms and the number of contrastive experiments conducted, it was not feasible to compute pairwise similarity on the entire collection. Instead, the algorithms were run on a 9 block sample, consisting of blocks 0, 50, 100, 150, 200, 250, 300, 350, 400. This provided a representative sample of 90k documents, or about 2% of the entire collection. Each algorithm, run on a particular block, represents a single Hadoop job.

Running time provides the basic measure of efficiency. One issue that became evident in initial experiments was the prevalence of "stragglers". Due to the Zipfian distribution of terms, completion of a Hadoop job would often be delayed by one or two reducers that take significantly longer than the others (this is a common problem, see [4]). To quantify the impact of stragglers, two separate time measurements were taken for the PCP and PQ algorithms: 100% completion time (end-to-end job completion), and 99% completion time, defined as the time it takes for 99% of the reducers to finish. For the BF algorithm, the 100% completion time was taken, along with 99% completion time for the mappers (the reduce phase for the BF algorithm is very short, usually lasting less than a minute). There are often large differences between the 99% and 100% completion times. In an operational setting, it can be argued that the 99% completion time is more meaningful since Hadoop jobs can be staggered (starting a job before the previous one has finished).

However, completion time may not be an entirely accurate measure of efficiency, and is subject to some variance, for at least two reasons: first, it depends on the overall cluster load; second, it depends on idiosyncrasies of the particular machine that happens to be running the mappers and reducers. To supplement completion time measurements, the amount of intermediate output generated by the mappers was recorded for the PQ and PCP approaches (this measure isn't meaningful for the BF algorithm since it generates little intermediate output).

All Hadoop jobs in the efficiency experiments were configured with 200 mappers and 200 reducers. The number of mappers represents only a hint to the runtime, since to a large extent the structure of the input data (e.g., disk layout) determines the data splits that serve as input to the mappers. In the experiments, the actual number of mappers was around 240. On the other hand, the number of reducers can be precisely controlled. To help the reader get a better sense of these numbers, a recommended configuration of Hadoop is two mappers per processor core (to achieve good IO balance). Therefore, 15 of today's server-class machines

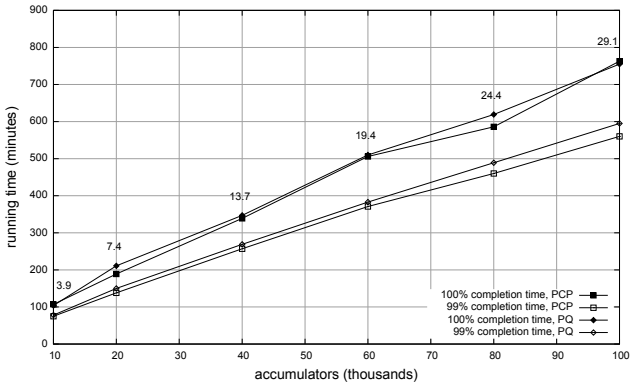

Figure 7: Running time of the PQ and PCP algorithms with different accumulator limits.

(with two quad-core processors) provide sufficient processing capacity to run 240 mappers simultaneously.

Before presenting results, a few notes on running times: it is difficult to interpret these measurements in an absolute sense. The experiments were run on a large cluster, but utilized only a small fraction of available processing capacity; furthermore, the cluster is a virtualized environment (which incurs some overhead) and is shared among a few universities (although these experiments were conducted at times when there were no concurrent jobs for the most part). Finally, little attention was paid to optimizing the Java code used in these experiments—in particular, more aggressive optimization of the inner loops involved in all three algorithms could yield significant performance gains. Like many information retrieval experiments, results are primarily meant for comparison with contrastive conditions.

Figure 7 plots the running times of both the PQ and PCP algorithms for different accumulator limits (on the 9 block sample). Size of the intermediate output is shown as annotations on the data points, measured in millions of bytes per document (same for both PQ and PCP). The graph suggests that the PCP algorithm is marginally more efficient than the PQ algorithm (evident in the 99% completion times), which makes sense since the latter has more complex control logic, necessary for checking the presence of query terms (hash lookups in the PQ implementation). Table 1 provides a summary of effectiveness and efficiency results for various algorithms at key parameter settings. The second column shows decreases in micro and macro P5 scores, compared to exact similarity. The third column lists the 99% and 100% completion times. Condition (0) represents the brute force algorithm with no approximations (i.e., exact similarity). Conditions (1) and (2) represent the PQ and PCP algorithms with 80k accumulators. The drops in effectiveness for both conditions are not significant.

Extrapolating trends for both the PQ and PCP algorithms, it appears that if exact similarity is desired, the brute force algorithm is more efficient than either of the indexed approaches, since the accumulator limit corresponding to exact similarity would be around 1600k, more than an order of magnitude larger than the largest limit considered in Figure 7 (see additional discussion in Section 5).

The effects of different term limits on efficiency are shown in Figure 8 for both the brute force and PQ algorithms (vertical scale is the same as Figure 7 to support easy compar-

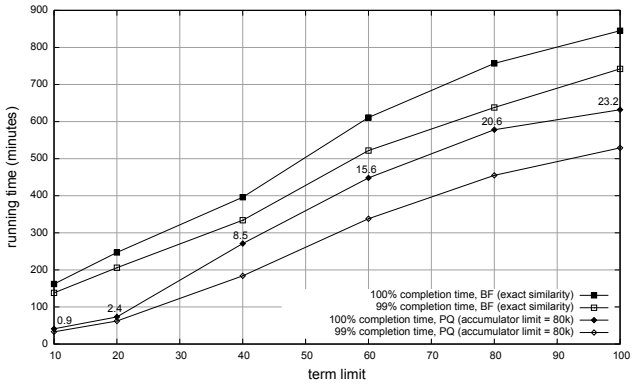

Figure 8: Running time of the PQ and BF algorithms with different term limits.

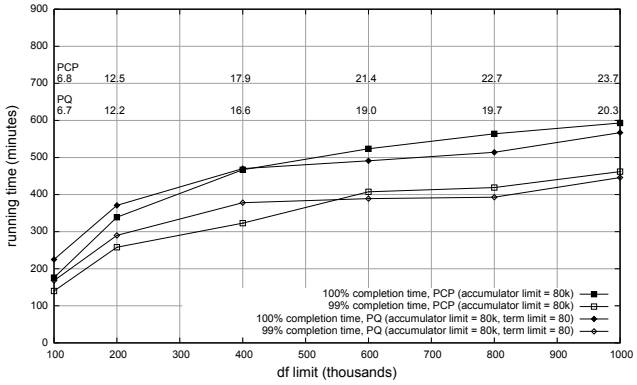

Figure 9: Running time of PQ and PCP algorithms with different *df* limits.

ison). Size of intermediate output is conveyed in the same manner as in the previous figure (applicable only to PQ). Across these documents, average length is 91 terms after stopword removal.[5] With the term limit approximation, the PQ algorithm is much more efficient than the BF algorithm across all settings. Condition (3) in Table 1 represents the BF algorithm at a term limit of 80; condition (4) represents PQ at a term limit of 80 and 80k accumulators. For both conditions, decreases in precision are not significant.

Finally, the effects of the *df* limit are shown in Figure 9, for PCP with 80k accumulators, and PQ with 80k accumulators and an 80-term limit. The second represents the most aggressive approximation thus far—taking advantage of all three approximations presented in this paper. Size of intermediate output is shown on the graph above the data points. For both sets of experiments, a *df* limit of 600k does not result in significantly lower precision—shown as conditions (5) and (6) in Table 1.

Results suggest that for low *df* limits, the PQ algorithm with the addition of the term limit (diamonds) actually takes longer to run. This is most likely because the PQ algorithm is inherently less efficient than the PCP algorithm due to its more complex control logic (as seen in Figure 7), and there is a large overlap in the terms that are discarded by

[5]This is shorter than the average length of relevant abstracts from the TREC 2005 genomics track because some MEDLINE citations lack abstract text, and are less likely to be relevant.

Setting	Effectiveness	Efficiency
0: BF exact	0.3993, 0.2230	862m, 1058m
1: PQ a80k	−1.0%, −1.3%	489m, 619m
2: PCP a80k	−1.0%, −1.3%	460m, 586m
3: BF t80	−0.2%, −0.7%	638m, 757m
4: PQ a80k,t80	−1.9%, −1.9%	455m, 578m
5: PCP a80k,df600k	−1.7%, −1.2%	408m, 524m
6: PQ a80k,t80,df600k	−2.3%, −1.7%	389m, 491m
7: PCP a60k	−2.1%, −3.7%	371m, 506m
8: PQ a80k,t60	−4.3%, −3.7%	338m, 448m
9: PCP a80k,df200k	−3.6%, −4.8%	258m, 339m

Table 1: Summary of algorithms at key settings. Effectiveness reported as relative differences in micro, macro P5; efficiency reported as 99%, 100% completion times. (a=accumulator, t=term, df=*df*)

the *df* and term limits. As a result, at lower *df* limits the PQ algorithm does more work with little additional benefit. Considering the tradeoff, it might not be worthwhile to apply *both* approximations at the same time.

Thus far, presentation of results has focused on settings where efficiency gains do not come at a significant cost in effectiveness. In Table 1, these are conditions (1) through (6), or the second block of rows in the table. However, more aggressive settings can further reduce the running time, with a relatively small (albeit statistically significant) cost in effectiveness. A few examples of these settings are shown as conditions (7), (8), and (9). In summary, the accumulator, term, and *df* limits provide useful "knobs" that can be adjusted to achieve the desired balance between effectiveness and efficiency, depending on the application and computational resources available.

5. DISCUSSION

Experimental results appear to support the following findings: First, if exact similarity is desired, the brute force approach is the most efficient of the three algorithms. Second, by limiting accumulators, both the PQ and PCP algorithms can achieve large efficiency gains without significant decreases in effectiveness. Third, additional gains in efficiency can be achieved with the term and *df* limits, although there is overlap in the computations they save.

The inverted index is perhaps the single most important data structure in text retrieval, yet a brute force approach without indexes is surprisingly efficient for pairwise similarity comparison. This result can be understood in terms of the most important tradeoff in distributed computing for divide-and-conquer strategies: the gains that can be achieved by breaking a large problem into smaller parallel pieces must be balanced against the overhead of coordinating those parallel operations. The brute force approach converts pairwise similarity comparison into an embarrassingly parallel problem that requires very little coordination between concurrent processes. This comes at a cost of many unnecessary computations—inner products between documents that have no terms in common. On the other hand, both the PQ and PCP algorithms require significant coordination between the various workers operating in parallel—this manifests in the shuffling of partial document scores from the mappers to the reducers, where final query-document scores are computed. Although each worker avoids unnecessary

computations with the help of the inverted index, this comes at a cost of intermediate results that need to be communicated across the network, which is the slowest component in a cluster. Experiments show that the size of the intermediate output can be nearly two orders of magnitude larger than the collection or the inverted index itself.

One obvious optimization is to apply standard index compression techniques to reduce the size of the intermediate output [14], e.g., *d*-gap/Golomb encoding of accumulators or quantized impacts. These optimizations represent tradeoffs in processing cycles (sorting, encoding, and decoding) for space, and will likely increase efficiency. However, it is unlikely that these optimizations alone will make the PQ and PCP algorithms competitive with the brute force algorithm if exact similarity is desired. Furthermore, compression-based optimizations are unlikely to affect the general findings reported in this paper, since all contrastive conditions would benefit. The general shape of the performance curves in Section 4.2 would not change (only the scales). The implementation of these optimizations is left for future work.

The three algorithms in this paper share in the general idea of organizing pairwise similarity computations into sequential operations over large files (either the collection itself or the inverted index). This design was necessary since HDFS, the file system underlying Hadoop, does not provide low-latency random access to data. As a result, it is not practical for a mapper or reducer to access "side data" stored on disk. Furthermore, by design, mappers and reducers are not able to directly share information with each other except through the sorting and shuffling of intermediate key-value pairs. These limitations restrict the design space of possible algorithms. The difficulty in developing a "smarter" accumulator limit, already discussed in Section 3.3, exemplifies this problem, but the issues are more general. MapReduce challenges the algorithm designer to think differently, since many constructs taken for granted in a single-processor environment (e.g., global variables or shared state) may not be readily available. Currently, there is little in the way of accumulated wisdom and best practices on the design of large-scale distributed algorithms. Hopefully, this work will make a small contribution by providing experience that may be useful for a broader range of text processing problems.

As future work it would be worth comparing the algorithms presented here with an entirely different approach based on locality-sensitive hashing [1]. Such randomized algorithms have been applied successfully in related problems that primarily depend on a similarity score threshold (e.g., duplicate detection), but it is not clear if they are adequate to produce accurate document rankings.

Finally, it might be argued that MapReduce is not necessary for the size of the problem examined in this paper, since the entire collection could reasonably fit into main memory on a single machine. While this is true, there are inherent limitations to algorithms that depend on in-memory processing, whereas the algorithms presented here have no obvious bottlenecks when scaling out to much larger collections.

6. CONCLUSION

This paper describes three algorithms for computing pairwise similarity on document collections and presents experimental results for the application of "more like this" queries in the life sciences domain. The contributions of the work lie in both algorithm design and empirical analysis of performance in a real-world application. The algorithms come with a number of approximations that provide the ability to control the balance between effectiveness and efficiency, making them adaptable to a wide range of usage scenarios. The Hadoop implementation of the MapReduce programming model provides a good framework for organizing distributed computations across a cluster, but it requires researchers to approach problems in a different way. Nevertheless, Hadoop provides unprecedented opportunities for researchers to tackle interesting problems at scale.

7. ACKNOWLEDGMENTS

This work was supported by 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 loving support.

8. REFERENCES

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

[2] V. Anh, O. de Kretser, and A. Moffat. Vector-space ranking with effective early termination. In *SIGIR*, 35–42, 2001.

[3] R. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, 131–140, 2007.

[4] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 137–150, 2004.

[5] T. Elsayed, J. Lin, and D. Oard. Pairwise document similarity in large collections with MapReduce. In *ACL, Companion Volume*, 265–268, 2008.

[6] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *SOSP*, 29–43, 2003.

[7] W. Hersh, A. Cohen, J. Yang, R. Bhupatiraju, P. Roberts, and M. Hearst. TREC 2005 Genomics Track overview. In *TREC*, 2005.

[8] N. Lester, A. Moffat, W. Webber, and J. Zobel. Space-limited ranked query evaluation using adaptive pruning. In *WISE*, 470–477, 2005.

[9] J. Lin and M. Smucker. How do users find things with PubMed? Towards automatic utility evaluation with user simulations. In *SIGIR*, 19–26, 2008.

[10] J. Lin and W. J. Wilbur. PubMed related articles: A probabilistic topic-based model for content similarity. *BMC Bioinformatics*, 8:423, 2007.

[11] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM TOIS*, 14(4):349–379, 1996.

[12] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *JASIS*, 47(10):749–764, 1996.

[13] T. Strohman and W. Croft. Efficient document retrieval in main memory. In *SIGIR*, 175–182, 2007.

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

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