Brute-Force Approaches to Batch Retrieval: Scalable Indexing with MapReduce, or Why Bother?

Tamer Elsayed,[1] Ferhan Ture,[1] Jimmy Lin[2]
[1]Department of Computer Science
[2]The iSchool, College of Information Studies
University of Maryland
College Park, Maryland 20742
{telsayed,fture}@cs.umd.edu, jimmylin@umd.edu

Abstract

Modern information retrieval research has evolved a standard workflow that involves first indexing a document collection and then running *ad hoc* queries sequentially to evaluate retrieval effectiveness using standard test collections. This paper explores how aspects of this workflow might change in a MapReduce cluster-based environment. First, we present and evaluate two algorithms for inverted indexing that take advantage of the programming model's sorting mechanism to different extents. The running times of both algorithms scale linearly in terms of collection size up to 102 million web pages. Second, we show that it is possible to efficiently perform batch query evaluation with MapReduce by scanning all postings lists in parallel, as opposed to sequentially accessing each postings list. Third, we explore an approach that forgoes inverted indexing altogether and simply computes all query–document scores from document vectors themselves. Experimental results challenge us to think differently about previous assumptions in information retrieval, and show that brute force approaches are surprisingly compelling under certain circumstances: parallel scan of postings can effectively take advantage of large clusters and parallel scan of documents fits naturally with ranking functions that use document-level features.

1 Introduction

Inverted indexing and ranked retrieval form the foundations of modern Information Retrieval (IR). The standard research workflow consists of these two steps in sequence. Prior to running retrieval experiments, a document collection must first be indexed. In a standard inverted index, each term is associated with a list of postings, each of which consists of a document id and some payload information (e.g., term frequency). Inverted indexes are typically a fraction of the original collection in size (depending on the payload), but are usually too large to hold in memory completely (leaving aside caching of postings or results). Ranked retrieval involves random access to postings that correspond to query terms (residing on disk) and systematically traversing the postings to compute query–document scores based on a ranking model. In the academic community, it remains the norm that indexing and retrieval are performed on individual machines. However, the continual growth of collection sizes means that single-machine solutions are becoming increasingly impractical. We must turn to distributed algorithms that take advantage of clusters.

This paper reexamines the standard workflow of indexing and retrieval in a cluster-based environment built around the open-source Hadoop implementation of MapReduce [10], a framework for processing large amounts of data on commodity clusters. We explicitly wanted to provide a "big picture" of running retrieval experiments from end to end in such an environment. The first question we tackle is how to efficiently build inverted indexes using the MapReduce framework. We present two algorithms that take advantage of the programming model to different extents and evaluate their performance on segments of the ClueWeb09 web crawl (up to 102 million pages). Having built inverted indexes, we turn our attention to the "bread and butter" of modern IR research: batch *ad hoc* retrieval. Typically, retrieval is performed sequentially, one query at a time, and involves random access to on-disk postings lists. However, this underutilizes available processing capacity in a cluster. As an alternative, we present an approach (called *PScan*) that involves a parallel scan of all postings, where all queries in a testset are evaluated concurrently. This brute force approach is taken one step further by dispensing with the inverted index altogether. With MapReduce, we can exploit the aggregate disk bandwidth of all cluster nodes to compute query–document scores directly from the document vectors (we call this *DScan*). Experiments show that these two non-traditional approaches are compelling for different reasons: parallel scan of postings lets us take advantage of large clusters and parallel scan of documents lets us easily work with ranking functions that use document-level features.

We view this work as having three contributions. First, we present and evaluate two scalable algorithms for inverted indexing in MapReduce. Although the basic ideas behind these two algorithms are not new, to our knowledge, their formulations in MapReduce are novel. Furthermore, we are not aware of any previously-published work that has presented experiments on inverted indexing at the scale of the collection sizes we report. Second, this paper critically examines the standard workflow in academic IR, which focuses on batch *ad hoc* retrieval experiments, performed repeatedly. We challenge the conventional wisdom of query evaluation as random postings lookup and traversal. Going even further, we entertain the radical suggestion of dispensing entirely with the inverted index. Experimental results challenge us to rethink rarely-questioned assumptions in information retrieval. Finally, to help move the community forward, implementations of algorithms in this paper are part of Ivory, a new open-source toolkit for web-scale retrieval that we are excited to share.

The remainder of this paper is organized as follows. We begin in Section 2 with a brief overview of MapReduce. Section 3 describes two inverted indexing algorithms, which we evaluate in Section 4 on a web-scale collection. Section 5 focuses on the PScan algorithm for batch retrieval, where all postings lists are processed in parallel. Section 6 explores the DScan algorithm for batch retrieval that dispenses entirely with the inverted index and computes query–document scores directly from document vectors. We conclude in Section 7 with a summary of our contributions.

2 Distributed Processing and MapReduce

It is fairly clear that web-scale collections have outgrown the capabilities of individual machines, necessitating the use of clusters to tackle basic problems in information retrieval. Distributed computations are inherently difficult to organize, manage, and reason about. With traditional programming models such as MPI, the developer must explicitly handle a range of system-level details, ranging from synchronization to data distribution to fault tolerance. Recently, MapReduce [10] has emerged as an attractive alternative: its functional abstraction provides an easy-to-understand model for designing scalable and 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, nodes in a graph) to generate partial results, which are then aggregated in some fashion. Taking inspiration from higher-order functions in functional programming, MapReduce provides an abstraction for programmer-defined "mappers" (that specify the per-record computation) and "reducers" (that specify result aggregation), with the following signatures:

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

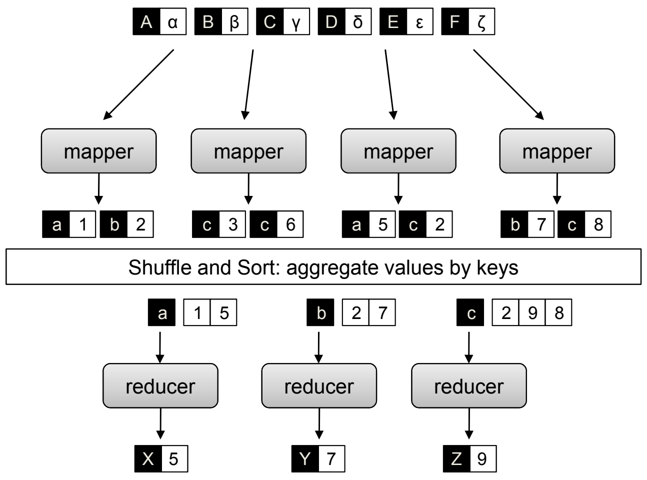

Figure 1: Illustration of MapReduce. Mappers are applied to all input key-value pairs, which generate an arbitrary number of intermediate key-value pairs. Reducers are applied to all values associated with the same key. Between the map and reduce phases lies a barrier that involves a large distributed sort and group by.

where k's indicate keys, v's indicate values, parentheses indicate a single record, and square brackets indicate a list of records. 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 an arbitrary number of final key-value pairs as output. A partitioning function divides up the intermediate key space by assigning intermediate keys to reducers (in the simplest case, using a hash function), and intermediate keys are guaranteed to be processed in sorted order in each reducer. This two-stage processing structure is illustrated in Figure 1.

Under the MapReduce programming model, a developer needs only to provide implementations of the mapper and reducer. On top of a distributed file system [13], the execution framework (i.e., "runtime") transparently handles all other aspects of execution on clusters ranging from a few to a few thousand cores, on gigabytes to petabytes of data. It is responsible, among other things, for scheduling (moving code to data), handling faults, and the large distributed sorting and shuffling problem between the map and reduce phases whereby intermediate key-value pairs must be grouped by key.

As an optimization, MapReduce supports the use of "combiners", which are similar to reducers except that they operate directly on the output of mappers; one can think of them as "mini-reducers". Combiners operate in isolation on each node in the cluster and cannot use partial results from other nodes. Since the output of mappers (i.e., the key-value pairs) must ultimately be shuffled to the appropriate reducer over a network, combiners allow a programmer to aggregate partial results, thus reducing network traffic. In cases where an operation is both associative and commutative, reducers can directly serve as combiners, although in general this is not the case.

The final component of MapReduce is the "partitioner", which is responsible for dividing up the intermediate key space and assigning intermediate key-value pairs to reducers. The default partitioner computes the hash value of the key and then takes the mod of that value with the number of reducers. This randomly assigns approximately the same number of keys to each reducer.

The open-source Hadoop implementation of MapReduce has given researchers a powerful tool for tackling large-data problems in areas such as machine learning [8, 35, 26], text processing [4, 12, 11, 18], databases [7, 27, 1] graph analysis [14, 20], and bioinformatics [30, 16], just to name a few. This paper explores running retrieval experiments from end to end in this environment.

3 Inverted Indexing

Inverted indexing on a single machine is well understood and efficient algorithms can be found in IR textbooks [34, 22, 6]. Distributed indexing has received attention in the research community since the 1990s [32, 28, 24] and is a problem that search engine companies have solved [5, 3]. However, since companies are often hesitant to publish proprietary details, there is surprisingly little published results dealing with the performance of web-scale indexing algorithms. Furthermore, there does not seem to be much recent research activity directly focused on distributed indexing of large collections.

Much has changed since the above publications. In terms of hardware, computers have obviously gotten faster, with more memory and bigger disks—but crucially, many assumptions about the relationship between processor, memory, and disk no longer hold. In terms of software, we have seen the advent of programming models such as MapReduce that allow us to organize and reason about computations on a massive scale. MapReduce provides not only a framework for concisely expressing inverted indexing algorithms, but also a much deeper analytics pipeline in which indexing might only form a component. For these reasons, we think it is worth revisiting the problem of distributed indexing in the context of MapReduce (specifically, Hadoop). We are aware of one other recent research paper on inverted indexing with MapReduce [23], which we explicitly discuss later.

A baseline inverted indexing algorithm in MapReduce is shown in Figure 2. This algorithm can be attributed to material prepared by Google employees for a university course in MapReduce and for other training purposes.[1] Input to the mappers consists of document ids[2] (keys) paired with the document content (values). Inside the mapper, each document is tokenized, stemmed, and filtered for stopwords. From the processed terms, a histogram H of term frequencies is built—the details of document processing are hidden in line 4 of the mapper code. The algorithm then iterates over all terms: for each, a posting consisting of the document id and the term frequency is created (line 7 in the mapper pseudo-code). The mapper then emits an intermediate key-value pair with the term as the key and the posting as the value. In this simple case, the payload of each posting contains only the term frequency *tf*, but this can easily be augmented with term positions to build positional indexes.

In the shuffle and sort phase, the MapReduce execution framework performs a large, distributed "group by" of the postings by term. Without any additional effort by the programmer, all postings associated with the same term are brought together in the reducer, which gathers them and writes the postings to disk. The reducer begins by initializing an empty list and then iteratively appends incoming values (i.e., postings). The postings are then sorted (by document id), compressed, and written to disk (appropriately compressed).

The MapReduce programming model provides a very concise expression of the inverted indexing algorithm. Its implementation is similarly concise: the basic algorithm can be implemented in as few as a couple dozen lines of code in Hadoop (with minimal document processing). Such an implementation can be completed as a week-long programming assignment in a course for advanced undergraduates or first-year graduate students [15, 17]. In a non-MapReduce indexer, a significant fraction of the code is devoted to grouping postings by term, given constraints imposed by memory and disk (e.g., memory capacity is limited, disk seeks are slow, etc.). In MapReduce, the programmer does not need to worry about any of these issues—most of the heavy lifting is performed by the execution framework.

3.1 Design Space of Algorithms

Of course, the baseline inverted indexing algorithm in MapReduce represents a single point in the possible design space of MapReduce algorithms for inverted indexing. We briefly discuss alternatives here before describing the two approaches explored in this paper.

[1]http://code.google.com/edu/parallel/

[2]We assume that documents are sequentially numbered.

```
class Mapper
    method Map(docid $n$, doc $d$)
        $H \leftarrow$ new AssociativeArray                ▷ histogram to hold term frequencies
        for all term $t \in$ doc $d$ do    ▷ processes the doc, e.g., tokenization and stopword removal
            $H\{t\} \leftarrow H\{t\} + 1$
        for all term $t \in H$ do
            Emit(term $t$, posting $\langle n, H\{t\}\rangle$)            ▷ emits individual postings
class Reducer
    method Reduce(term $t$, postings $[\langle n_1, f_1\rangle \ldots]$)
        $P \leftarrow$ new List
        for all $\langle n, f\rangle \in$ postings $[\langle n_1, f_1\rangle \ldots]$ do
            $P$.Append($\langle n, f\rangle$)                          ▷ appends postings unsorted
        $P$.Sort()                                                      ▷ sorts for compression
        Emit(term $t$, postingsList $P$)
```

Figure 2: Pseudo-code of the baseline inverted indexing algorithm in MapReduce.

Given an existing single-machine indexer, one simple way to take advantage of MapReduce is to leverage reducers to merge indexes built on local disk. This might proceed as follows: an existing indexer is embedded inside the mapper, and mappers are applied over the entire document collection. Each indexer operates independently and builds an index on local disk for the documents it encounters. Once the local indexes have been built, compressed postings are emitted as values, keyed by the term. In the reducer, postings from each locally-built index are merged into a final index.[3]

Another relatively straightforward adaptation of a single-machine indexer is demonstrated by Nutch.[4] Its algorithm processes documents in the map phase, and emits pairs consisting of docids and analyzed document contents. The sort and shuffle phase in MapReduce is used essentially for document partitioning, and the reducers build each individual index partition independently. In contrast with the above approach, Nutch basically embeds a traditional indexer in the reducers, instead of the mappers. With this approach, the number of reducers specifies the number of document partitions—which limits the degree of parallelization that can be achieved.

We decided not to pursue the two approaches discussed above since they seemed like incremental improvements over existing indexing methods. Instead, we implemented and evaluated two distinct algorithms that make fuller use of the MapReduce programming model. The first is a scalable variant of the baseline inverted indexing algorithm in MapReduce, in which the mappers emit individual postings. The second is an algorithm in which the mappers emit partial lists of postings. The algorithms primarily differ in how postings are sorted: by the execution framework (in the first algorithm) or by the indexing code itself (in the second algorithm). Detailed descriptions of both are provided below, followed by a general discussion of their relative merits.

3.2 Emitting Individual Postings

The starting point of our first algorithm, based on mappers emitting individual postings, is an observation about a significant bottleneck in the baseline algorithm in Figure 2: it assumes that there is sufficient memory to hold all postings associated with the same term before sorting them. Since the MapReduce execution framework makes no guarantees about the ordering of values associated with the same key, the reducer must first buffer all postings and then perform an in-memory sort before the

[3]Indri is capable of distributed indexing using exactly this approach, albeit outside of the MapReduce framework.

[4]http://lucene.apache.org/nutch/

```
1: class Mapper
2:     method Map(docid $n$, doc $d$)
3:         $H \leftarrow$ new AssociativeArray
4:         for all term $t \in$ doc $d$ do                          ▷ builds a histogram of term frequencies
5:             $H\{t\} \leftarrow H\{t\} + 1$
6:         for all term $t \in H$ do
7:             Emit(tuple $\langle t, n \rangle$, tf $H\{t\}$)          ▷ emits individual postings, with a tuple as the key
1: class Partitioner
2:     method Partition(tuple $\langle t, n \rangle$, tf $f$)
3:         return Hash($t$) mod $NumOfReducers$              ▷ keys of same term are sent to same reducer
1: class Reducer
2:     method Initialize
3:         $t_{prev} \leftarrow \emptyset$
4:         $P \leftarrow$ new PostingsList
5:     method Reduce(tuple $\langle t, n \rangle$, tf $[f]$)
6:         if $t \neq t_{prev} \wedge t_{prev} \neq \emptyset$ then
7:             Emit(term $t$, postings $P$)                          ▷ emits postings list of term $t_{prev}$
8:             $P$.Reset()
9:         $P$.Append($\langle n, f \rangle$)                              ▷ appends postings in sorted order
10:        $t_{prev} \leftarrow t$
11:    method Close
12:        Emit(term $t$, postings $P$)                          ▷ emits last postings list from this reducer
```

Figure 3: Pseudo-code of the inverted indexing algorithm based on emitting individual postings (IP).

postings can be written out to disk. Of course, as collections grow in size there may not be sufficient memory to perform this sort (bound by the term with the largest *df*).

Since the MapReduce programming model guarantees that keys arrive at each reducer in sorted order, we can overcome the scalability bottleneck by letting the execution framework do the sorting. Instead of emitting key-value pairs of the form:

$$(\text{term } t, \text{posting } \langle docid, f \rangle)$$

we emit intermediate key-value pairs of the form:

$$(\text{tuple } \langle t, docid \rangle, \text{tf } f)$$

In other words, the key is a tuple containing the term and the document number, while the value is the term frequency. We need to redefine the sort order so that keys are sorted first by term t, and then by docid n. Additionally, we need a custom partitioner to ensure that all tuples with the same term are shuffled to the same reducer. Having implemented these two changes, the MapReduce execution framework ensures that the postings arrive in the correct order. This, combined with the fact that reducers can hold state across multiple keys, allows compressed postings to be written with minimal memory usage.

The revised MapReduce inverted indexing algorithm is shown in Figure 3. The mapper remains unchanged for the most part, other than differences in the intermediate key-value pairs. The key space of the intermediate output is partitioned by term; that is, all keys with the same term are sent to the same reducer. This is guaranteed by the partitioner. The reducer contains two additional methods:

Initialize, which is called before keys are processed, and Close, which is called after the final key is processed (both API hooks in Hadoop). The Reduce method is called for each key (i.e., $\langle t, n \rangle$), and by design, there will only be one value associated with each key. For each key-value pair, a posting can be directly added to the postings list. Since the postings are guaranteed to arrive in the correct order, they can be incrementally encoded in compressed form—thus ensuring a small memory footprint. Finally, when all postings associated with the same term have been processed (i.e., $t \neq t_{prev}$), the entire postings list is written out to disk. The final postings list must be written out in the Close method.

Since each reducer writes its output in a separate file, our final index will be split across r files, where r is the number of reducers. However, there is no need to consolidate the r files, since we can keep track of which index file a term's postings list is found in (for looking up postings during retrieval). It is important to note that these files are stored on the distributed file system (HDFS for Hadoop), which means that data comprising each index file are stored in blocks, distributed and replicated throughout nodes in the cluster.

Two more details complete the description of this indexing algorithm: positional information and parameter settings for postings list compression. First, positional indexes can be built by simply replacing the intermediate value f (term frequency) with an array of term positions (gap-compressed with γ codes [34]). Second, parameters must be appropriately set for compression of the postings lists. One common practice is to use Golomb compression on first order document id differences (i.e., *d*-gaps) [34, 36]. The difficulty, however, is that Golomb compression requires two parameters: the size of the document collection and the number of postings for a particular term (i.e., document frequency or *df*). The first is easy to obtain and can be passed into the reducer as a constant. The *df* of a term, however, is not known until all the postings have been processed—and unfortunately, the parameter must be known before any posting is encoded. A two-pass solution that involves first buffering the postings (in memory) would suffer from the memory bottleneck we have been trying to avoid in the first place.

To get around this problem, we need to somehow inform the reducer of a term's *df* before any of its postings arrive. The solution is to have the mappers emit special keys of the form $\langle t, * \rangle$ to communicate partial document frequencies. The mapper holds an in-memory associative array that keeps track of how many documents a term has been observed in (i.e., the local document frequency of the term for the subset of documents processed by the mapper). Once the mapper has processed all input records, special keys of the form $\langle t, * \rangle$ are emitted with the partial *df* as the value.

To ensure that these special keys arrive first, we define the sort order of the tuple so that the special symbol $*$ precedes all documents. Thus, for each term, the reducer will first encounter a series of $\langle t, * \rangle$ keys, representing partial *dfs* originating from each mapper. Summing all these partial contributions will yield the term's *df*, which can then be used to set the Golomb compression parameter. This allows the postings to be encoded in one pass.

3.3 Emitting Lists of Postings

The key difference in the second algorithm is that instead of emitting individual postings, the mappers emit compressed partial lists of postings. The basic idea is to replace a "flush to disk" operation in a traditional single-machine indexer with a "flush to intermediate key-value pairs" in MapReduce. This algorithm is shown in Figure 4.

The initial steps of the algorithm are quite similar to a traditional single-machine indexer that performs in-memory inversion. Each mapper preserves state across input documents in the data structure M, which holds a dictionary of terms that have been encountered so far and pointers to lists of postings for each term. As each document is processed, postings are added to M until the mapper runs out of memory. This triggers a call to Flush, in which the global data structure M is converted into m Golomb-compressed, document-sorted postings lists (one for each unique term in the processed

```
class Mapper
    method Initialize
        $M \leftarrow$ new AssociativeArray                    ▷ holds partial lists of postings
    method Map(docid $n$, doc $d$)
        $H \leftarrow$ new AssociativeArray                    ▷ builds a histogram of term frequencies
        for all term $t \in$ doc $d$ do
            $H\{t\} \leftarrow H\{t\} + 1$
        for all term $t \in H$ do
            $M\{t\}$.Add(posting $\langle n, H\{t\}\rangle$)          ▷ adds a posting to partial postings lists
        if MemoryFull() then
            Flush()
    method Flush                          ▷ flushes partial lists of postings as intermediate output
        for all term $t \in M$ do
            $P \leftarrow$ SortAndEncodePostings($M\{t\}$)
            Emit(term $t$, postingsList $P$)
        $M$.Clear()
    method Close
        Flush()
```

```
class Reducer
    method Reduce(term $t$, postingsLists $[P_1, P_2, \ldots]$)
        $P_f \leftarrow$ new List                        ▷ temporarily stores partial lists of postings
        $R \leftarrow$ new List                              ▷ stores merged partial lists of postings
        for all $P \in$ postingsLists $[P_1, P_2, \ldots]$ do
            $P_f$.Add($P$)
            if MemoryNearlyFull() then
                $R$.Add(MergeLists($P_f$))
                $P_f$.Clear()
        $R$.Add(MergeLists($P_f$))
        Emit(term $t$, postingsList MergeLists($R$))      ▷ emits fully merged postings list of term $t$
```

Figure 4: Pseudo-code of the inverted indexing algorithm based on emitting lists of postings (LP).

documents). These partial postings lists are emitted as values, keyed by the terms. In our actual implementation, positional information is also encoded in the postings lists, but this detail is omitted from the pseudo-code for presentation purposes.

In the reduce phase, all partial postings lists associated with the same term are brought together by the execution framework. The reducer must then merge all these partial lists (arbitrarily ordered) into a final postings list. For this, we adopted a two-pass approach. In the first pass, the algorithm reads postings lists (let's call them $p_1, p_2, \ldots$) into memory until memory is nearly exhausted. These are then merged to create a new postings list (let's call this p_a). The partial postings lists are in compressed form, which means we can store quite a few of them in memory. The memory needed for merging is relatively modest for two reasons: First, we know how many postings are in $p_1, p_2, \ldots$, so we can compress p_a incrementally—very few postings are actually materialized. Second, the d-gaps in p_a are smaller post-merging, so compression becomes more efficient. At the end of the first pass, we obtain a smaller number of partial postings lists ($p_a, p_b, \ldots$ in R), which are then merged in a second pass into a single postings list. This is emitted as the final value, keyed by the term, and written to disk. As in the previous algorithm, the key space is partitioned by term. The final index will be split across r

files, where r is the number of reducers, stored on the distributed file system.

This algorithm requires the mappers and reducers to more actively manage their memory footprint, which can be controlled by a few parameters. In the mapper, we must decide at what point to flush postings to intermediate key-value pairs. We elected to express this as a fraction of total memory consumed and the number of documents processed. However, improper settings can have a significant impact on performance: too conservative, and the algorithm generates more partial postings lists than necessary (more merging later on); too aggressive, and the algorithm may run out of memory. In the reducer, we must decide how many partial postings lists to merge at once. We elected to control this as a parameter expressed in terms of the fraction of total memory used.

In theory, this algorithm could benefit from the use of combiners to perform local aggregation on the output of the mappers. The reducer itself could be used as a combiner to consolidate partial postings lists emitted by the mappers; this has the effect of modestly reducing the number of intermediate keys. In practice, however, the addition of combiners actually hurts indexing performance and increases running time, since combiners compete with mappers for memory, resulting in suboptimal memory-access patterns.

3.4 Discussion

For convenience, we refer to the first algorithm as IP (individual postings) and the second algorithm as LP (lists of postings). As previously mentioned, the algorithms primarily differ in how postings are sorted. In the IP algorithm, the MapReduce execution framework handles both grouping postings by term and sorting terms by document. In the LP algorithm, MapReduce handles grouping of partial postings lists by term, but not sorting postings themselves; the mapper and reducer code shoulders the burden of sorting postings by document.

The advantage with the IP algorithm is simplicity, since a great deal of the complexity involved in inverted indexing is offloaded to mechanisms that are built into the MapReduce programming model. The other advantage is scalability "for free"—the sorting and grouping of postings will scale as long as the MapReduce implementation itself scales (which is not a worry since Hadoop has successfully scaled to sorting petabyte-sized datasets[5]). The downside, however, is inefficiency. In the IP algorithm, each individual posting is materialized and emitted as an intermediate key-value pair: the term is duplicated for every posting and the postings cannot benefit from gap-based compression. This results in a tremendous amount of intermediate data that needs to be copied across the network.

The LP algorithm manifests exactly the opposite set of tradeoffs. Since the mappers and reducers explicitly handle the sorting of postings by document, we can take advantage of best practices such as gap-based compression to significantly reduce the amount of intermediate data. Furthermore, since the mappers generate far fewer intermediate key-value pairs, less effort is spent by the execution framework in sorting and grouping terms (the keys). The tradeoff, however, is more complex code. In terms of programming effort, the LP algorithm was more difficult to implement and debug than the IP algorithm. For example, we spent a lot of time trying out different approaches to merging the partial postings lists in the reducer: seemingly simple tweaks translated into orders of magnitude differences in running times.

In addition, setting the various parameters in the LP algorithm required some amount of trial and error—having "more knobs to twiddle", in this case, was a disadvantage. First, we had to decide an allocation of memory for each map task, but this is dependent on the amount of memory on each cluster node and the number of concurrent tasks that each node can run (since they are drawing from the same resources). We had to be careful in budgeting physical memory, since performance would significantly degrade if the processes started consuming swap. Once deciding the memory allocation,

[5]http://developer.yahoo.net/blogs/hadoop/2009/05/hadoop_sorts_a_petabyte_in_162.html

we had to decide on specific settings inside the mapper and reducer (discussed in the previous section). All of this required hand-tuning.

Finally, a word on the novelty of these two algorithms. The ideas behind both the IP and LP algorithms are not new. IP is similar to sort-based index construction and LP is similar to merge-based index construction [36, 6]: both are well-known in the information retrieval literature. Our specific formulations in MapReduce, however, are novel, and we believe there is significant value in understanding how existing indexing strategies must be adapted to the MapReduce framework. To our knowledge, the IP algorithm is the first that addresses the scalability bottleneck in the baseline MapReduce inverted indexing algorithm. It was first described in a previous publication [19], but that paper lacked the rigorous evaluation we present here. The LP algorithm is similar to an algorithm proposed by McCreadie et al. [23], in that mappers in both cases emitted partial postings lists. However, details differ quite a bit, as McCreadie et al. build a separate (and independent) index partition in each reducer. This limits the degree of parallelization that can be achieved—it may not be useful for applications to have hundreds of partitions, but MapReduce jobs can routinely run hundreds and even thousands of reducers. In contrast, our algorithm builds a single unified index, and is able to scale out to an arbitrary number of reducers. In addition, the previous work only reported on collections that were much smaller than the ones we explore here.

4 Indexing Performance

4.1 Experimental Setup

Experiments were run on a cluster owned by Google and managed by IBM, shared among a few universities as part of NSF's CLuE (Cluster Exploratory) Program and the Google/IBM Academic Cloud Computing Initiative. The cluster used in our experiments contained 280 physical nodes; each node 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) was virtualized; each physical node runs one virtual machine hosting Linux. Experiments used Java 1.6 and Hadoop version 0.20.1. The cluster was configured to run a maximum of three map tasks and two reduce tasks simultaneously. Although the cluster is a shared resource, all experiments for which we report timing results were run during periods when there were no other large jobs competing for capacity.

Since more detailed specifications of the cluster machines (e.g., exact processor model) were not available to us, we decided to informally run our own performance benchmarks. An individual cluster node obtained a composite score of 442 on NIST's SciMark 2.0 benchmark,[6] averaged over 3 trials. For comparison, a laptop with a 2.6 GHz Core 2 Duo (T7800, released 2007) processor and 2 GB of RAM scored 494 on the same test (once again, averaged over three trials). SciMark consists of five computational kernels: FFT, Gauss-Seidel relaxation, Sparse matrix-multiply, Monte Carlo integration, and dense LU factorization. Note that this benchmark is primarily used to measure the performance of scientific and engineering applications, so the focus is on processor speed (which is only one component of overall performance). Nevertheless, we suspect that the cluster consists of previous-generation machines. Performance figures presented below should be interpreted with this important caveat. The 280-node cluster contained 560 cores, which, with current dual-processor quad-core configurations, could fit into 70 machines—a far more modest cluster with today's technology, not to mention that modern processors would be substantially faster.

We performed experiments on the ClueWeb09 collection, a best-first web crawl by CMU in early 2009. The collection contains one billion pages in ten languages totaling around 25 terabytes. Of those, about 500 million pages are in English, divided into ten roughly equal-sized segments. Our indexing experiments specifically focused on the first two English segments: the first contains 50.2m documents

[6]http://math.nist.gov/scimark2/

Alg.	Time	Intermediate Pairs	Intermediate Size
IP	38.5 min	13×10^9	306×10^9 bytes
LP	29.6 min	614×10^6	85×10^9 bytes

Table 1: Comparison of the IP and LP indexing algorithms on the first ClueWeb09 segment.

(totaling 1.53 TB uncompressed, 247 GB compressed) and the second contains 51.6m documents (totaling 1.44 TB uncompressed, 225 GB compressed).

4.2 Preprocessing

Prior to indexing, we first preprocessed the collection. This consisted of three major stages, all conceived as MapReduce jobs implemented in Java. In the first stage, all documents were parsed into document vectors (with stemming and stopword removal), represented as associative arrays from terms to term frequencies. At the same time we built a table of document lengths, necessary for retrieval later. In the second stage, we constructed a mapping from terms to integers (term ids), sorted by ascending document frequency, i.e., term 1 was the term with the highest *df*, term 2 was the term with the second highest *df*, etc. In this process, we discarded all terms that occurred ten or fewer times in the collection, since these rare terms are mostly misspelled words and are unlikely to be part of real-world user queries. The resulting dictionary was then compressed with front-coding [34]. Finally, in the third stage a new set of document vectors were generated in which terms were replaced with the integer term ids. Furthermore, within each document the terms were sorted in increasing term id, so that we were able to encode gap differences (using γ codes). The final result is a compact representation of the original document collection. The first and third stages are parallel operations with mappers and no reducers; the second stage uses a single reducer to build the term id mapping.

There were three primary reasons for building and separately storing this compressed representation of the document collection. First, for evaluating indexing performance, we wished to factor out the time taken to process the documents: parsing, tokenization, stemming, etc. Second, materializing the document vectors is necessary if the retrieval model performs relevance feedback. Third, this representation serves as the input to the brute force query-evaluation algorithm we describe in Section 6.

For the first English segment of ClueWeb09, the entire preprocessing pipeline took 54.3 minutes (averaged over two trials): 19.6 minutes for the first stage, 8.0 minutes for the second stage, and 26.7 minutes for the third. The parsed document vectors were 115 GB; replacing terms with term ids and gap compression reduced the size down to 64 GB.

4.3 Efficiency Results

We have implemented both the IP and LP indexing algorithms in Java. Starting from the compact representation of the collection, the running times of the IP and LP algorithms on the first English segment of ClueWeb09 are shown in Table 1, each averaged over three trials (cf. Figure 5). The third and fourth columns of the table show the number of intermediate key-value pairs and the total size of the intermediate data generated by the two approaches. The final size of postings lists is 64 GB, containing full positional information. Both algorithms construct a single, monolithic index (i.e., the document collection is *not* partitioned). We can see that the LP algorithm is relatively space efficient, generating only about a third more intermediate data than the final size of the postings, whereas the IP algorithm generates nearly five times more intermediate data.

For both algorithms, the MapReduce job decomposed into 2901 map tasks and 200 reduce tasks, each utilizing an allocated maximum heap size of 2 GB. Note that in the reduce phase we do not take advantage of all available cluster capacity. For the LP algorithm, mappers were set to flush postings

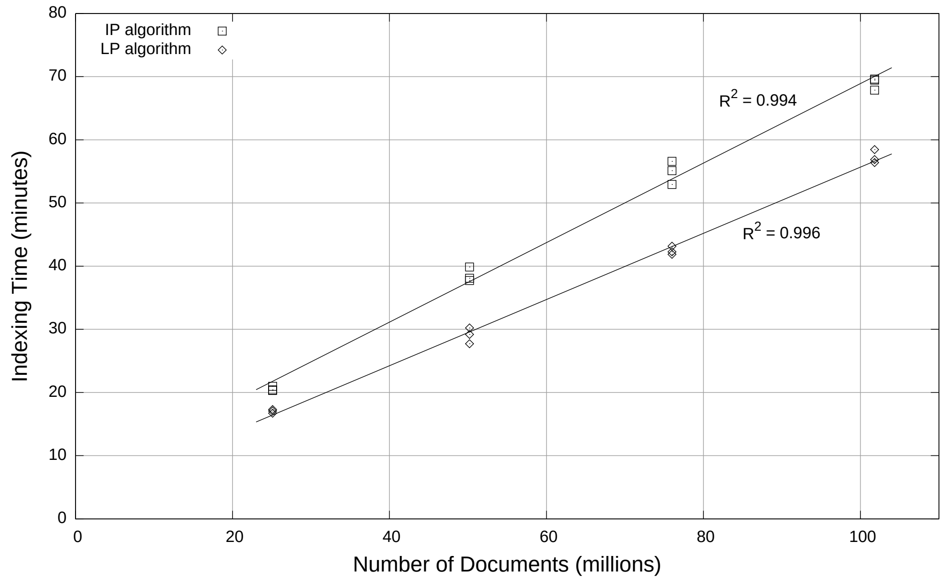

Figure 5: Running time of the LP and IP algorithms on the first two English segments of ClueWeb09.

as soon as memory was 90% full, or when the mapper had processed 50k documents; on the reducer end, the memory threshold was set to 90% as well. The correctness of the constructed indexes was verified using the 50 queries from the TREC 2009 web track. We had previously participated in the evaluation, reporting competitive results, which we were able to replicate.

Scaling characteristics of both indexing algorithms are presented in Figure 5; in addition to plotting the above results, we also show running times for half of the first ClueWeb09 segment (25.1m documents), the first and half of the second segment (76.0m documents), and the first two segments (101.8m documents). We emphasize that in all cases we are constructing a single monolithic (i.e., non-partitioned) index. The figure shows three trials each for the IP and LP algorithms. The graph also shows linear regressions through the running times: very high R^2 values demonstrate that both algorithms scale linearly with collection size, which is a very desirable property. We did not examine even larger collections because real-world retrieval engines adopt a document-partitioned architecture [2, 36], such that the bottleneck is in building the index for a *single* partition—building multiple partitions is parallelizable. Partition sizes, of course, are collection specific, but we find it unrealistic that one would in reality want to build even larger partitions (since query evaluation time would be dominated by traversal of the longest posting).

5 To Seek or Not To Seek?

Having indexed the document collection, researchers can proceed to focus on the central problem in IR: ranking documents in response to a user's query based on a particular retrieval model. An empirical discipline built around test collections is at the core of our field. The basic experimental cycle consists of developing or modifying the retrieval model, running a batch of *ad hoc* queries, and evaluating the quality of results based on some standard metric such as mean average precision. Traditionally, batch evaluation is performed sequentially, one query at a time, and the evaluation of each query consists of fetching postings that correspond to the query terms and traversing the postings to compute query–document scores.

How long does a retrieval experiment take? We started with an index of the first ClueWeb09 segment (50.2 million documents), copied it out of Hadoop's distributed file system (HDFS) onto the local disk

Queries	Seek	PScan	DScan
Web09	332s [04]	246s [09]	499s [028]
Robust04	825s [22]	280s [72]	604s [025]
Efficiency500	-	393s [48]	2061s [083]
Efficiency1000	-	764s [57]	3350s [331]
Efficiency1500	-	1018s [79]	4939s [930]

Table 2: Total running time (in seconds) of different queries on the first segment of ClueWeb09 (standard deviation in brackets).

of one cluster node, and timed some retrieval experiments using a single thread. This additional step of copying the index has the effect of consolidating all data onto the local disk of a single node; recall that files stored in HDFS are spread across many nodes in the cluster. Random access to HDFS data, therefore, incurs multiple roundtrip network latencies (first to contact the master for metadata and block location and then to contact the actual datanode) as well as overhead associated with streaming data across the network. HDFS was not designed with low-latency random access in mind, which is exactly the dominant data access pattern for query evaluation. Prior to retrieval, we constructed a postings forward index to store the file location and byte offset position of each postings list. This allowed us to, given a query term, seek to its postings.

We used 50 queries from the TREC 2009 web track and 100 queries from the TREC 2004 robust track. Documents were ranked with *BM25* [29]. Running times (not including startup costs of the retrieval engine) for retrieving 1000 hits are presented in Table 2, under the column "Seek". Our retrieval engine implements a document-at-a-time query evaluation strategy with a max_score optimization [33]. We would characterize this as a reasonable, but certainly not spectacular, implementation—comparisons with published figures support this assertion (recall that we are searching a single index). However, these results provide a baseline for comparison.

Time required for retrieval places an upper bound on how quickly one can explore the solution space in search of better retrieval models. That is, research is limited by the speed in which one can perform batch retrieval and evaluate the results. For large collections and sophisticated models that require non-trivial amounts of training [21], the experimental turnaround times are significant.

5.1 Parallel Postings Scan

Can we better exploit resources in a cluster-based environment to improve turnaround time for batch retrieval experiments? The fundamental problem with the sequential approach is that we are bound by disk seek latencies on a single node. However, with MapReduce, we can easily harness the aggregate throughput of all disks by scanning the postings in parallel. This can be implemented as follows: we map over all postings lists, with terms as keys and postings lists as values. The mapper loads all queries at once. When processing a (term, postings list) input key-value pair, if the term is *not* found in any of the queries, no action is performed. Otherwise, the postings list is emitted as an intermediate value, keyed by the query id of the query that contains the term (multiple intermediate key-value pairs are emitted if the query term exists in multiple queries). In the reducer, all postings corresponding to all query terms are brought together, and query evaluation proceeds as usual. In effect, we replace random postings lookup with a parallel scan of *all* postings. The shuffle and sort mechanism in MapReduce copies the necessary postings across the network and makes sure all postings with the same query id are gathered together. This allows us to maximally parallelize query evaluation: we can have as many reducers as there are queries. Lin [18] described a similar approach which involves the shuffling of accumulators containing partial scores, but the downside of computing scores in the mapper is the inability to apply any early-termination heuristics (e.g., max_score).

Queries	PScan	DScan
Web09	1.74×10^9	0.91×10^9
Robust04	3.66×10^9	2.57×10^9
Efficiency500	29.35×10^9	12.97×10^9
Efficiency1000	62.62×10^9	26.80×10^9
Efficiency1500	92.98×10^9	41.37×10^9

Table 3: The amount of intermediate data (bytes) in the PScan and DScan algorithms.

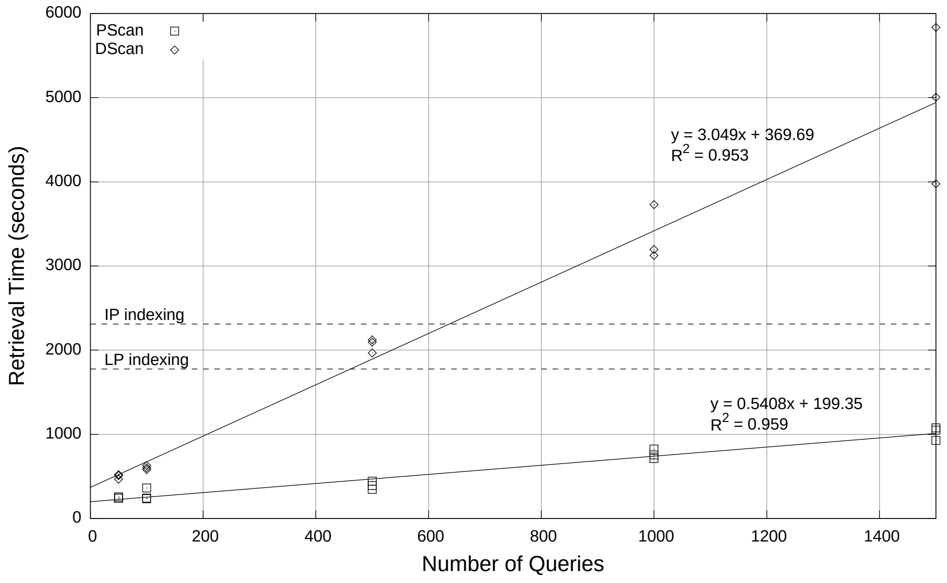

Figure 6: Running times of the PScan and DScan retrieval algorithms, with indexing times shown for reference.

We have implemented exactly this parallel query evaluation algorithm (which we call parallel postings scan, or *PScan* for short). Running times (in seconds) for the same conditions as before are shown in Table 2, under the column "PScan". We report averages over three trials on the cluster described in Section 4.2, with standard deviation of the running times shown in square brackets. The query evaluation algorithm in the reducer is the same as the algorithm in the single-threaded version, making the comparison fair. For all experiments, we ran a reducer per query (50 for Web09, 100 for Robust04), retrieving the top 1000 hits with *BM25*. On average, we observed a 26% reduction in running time for Web09 queries, compared to sequential query evaluation ("Seek" column), and a 66% reduction in running time for Robust04 queries. Although copying postings across the network may seem like a waste, these results show that the ability to evaluate all queries in parallel (and thereby making more effective use of resources) more than makes up for the additional network traffic. Note that although we can evaluate queries in parallel, there are still non-trivial startup costs associated with initializing each reducer. In particular, our current algorithm requires that all document lengths be read into memory (the reducers load this in as "side data"). Table 3 shows the amount of intermediate data that is shuffled from the mappers to the reducers (i.e., the size of all postings corresponding to query terms).

To further explore the scalability of this algorithm, we created three different sets of queries from the TREC 2006 efficiency task consisting of the first 500, 1000, 1500 queries. Running times of the PScan retrieval algorithm are shown in the last three rows of Table 2 (three trials, 1000 hits per query, *BM25*). In Figure 6, we plot the running times of the PScan algorithm (squares) vs. the number of queries for all of our testsets. Although the query sets are slightly different in nature (Robust04 queries were from newswire topics; the rest were web queries), the results nevertheless give us an idea of how the algorithm scales. As expected, running time increases linearly with number of queries, and we obtain a very good R^2 fit. For reference, running times of the LP and IP indexing algorithms are shown in dotted lines.

5.2 Discussion

How does the PScan algorithm compare with alternative strategies that might make better use of cluster resources? Of course, we could simply run r retrieval engines in parallel, where r is the number of nodes in our cluster ($r = 280$). However, this would require copying the index to local disks, which in our case would have taken a non-trivial amount of time. Furthermore, we would create complex data management issues with multiple versions of indexes that need to be kept in sync and up to date. As an alternative, we might copy the index onto network-attached storage (e.g., an NFS mount), shared by query engines running on all the cluster nodes. We expect that this would require a high-end filer in order to cope with the load (which we do not have access to).[7] While both approaches are certainly feasible, they are more complex than our PScan algorithm.

How does our PScan algorithm compare with current models of distributed retrieval? The standard solution is a broker-mediated, document-partitioned architecture [2, 36], where the document collection is divided into a number of independent and separately-indexed partitions.[8] A server is responsible for searching each partition, the results of which are merged by the broker. There are a few issues we see with this approach. To make maximal use of our cluster would require 280 partitions; while we can imagine modifying our inverted indexing algorithm to generate all partition indexes in parallel, we cannot avoid having to copy 280 separate indexes out of HDFS onto the local disks of the respective machines. Furthermore, with 280 partition servers, the query broker will need to contend with a non-trivial amount of network traffic. Finally, document-partitioned indexes require some mechanism for coordinating global term statistics across multiple partitions, which adds yet another layer of complexity. We believe that our algorithm is much simpler, both conceptually and architecturally.

One might object that given the amount of hardware resources we had access to, the performance gains are not particularly impressive. However, recall that in our experiments we only ran one reducer per query, and each reducer had substantial startup costs associated with loading "side data". The reducer startup cost could be amortized by running fewer reducers (each evaluating more queries), which means that one would expect to observe substantial performance gains even for modest clusters. Since the query evaluation algorithm is the same, we are only trading time to scan all postings (bound by disk bandwidth) and to copy them across the network with the ability to parallelize query evaluation. We further note that our approach takes full advantage of today's multi-core processors. Normally, parallel postings seeks (by threads on multiple cores) would be contending for disk (if the postings are not already in memory or cached), since there are generally more cores than disks. In the PScan algorithm, the postings are guaranteed to be already in memory by the time they arrive at the reducer, so we are no longer disk bound. Overall, we do not see a convincing downside to the PScan approach for batch query evaluation.

[7]As an alternative, we have been experimenting with reading postings directly from HDFS, which is a potential solution to this problem [19]. However, not enough is known about HDFS latency in this usage scenario so we set aside the possibility in this work.

[8]We leave aside term partitioning, since previous work has shown document partitioning to be superior [25].

6 Why Bother Indexing?

The performance of the PScan query evaluation algorithm illustrates the power of brute force scan-based approaches that take advantage of the aggregate throughput of many disks. Taking this to the logical end, could we dispense completely with inverted indexing and run batch *ad hoc* retrieval directly on the documents?

6.1 Parallel Document Scan

We implemented and evaluated a brute force algorithm that operates directly over the compressed document vectors created as part of preprocessing for inverted indexing (see Section 4.2). This approach, which we call *DScan*, is very simple. We map over all document vectors, and inside each mapper we load all queries into memory at once. For each document, we compute its score with respect to every query; the top m hits for every query are maintained in a set of priority queues in the mapper. Once each mapper has finished processing its assigned block of documents, the contents of the priority queues are emitted as key-value pairs, keyed by the query id. After shuffling and sorting, the reducer receives the top m hits emitted by every mapper for each query id, dumps those results in another priority queue, and then extracts the top k final results (which are written to disk). In our implementation $k = m$, which is conservative, since the final results are unlikely to draw deeply from the partial results of each mapper. This brute-force DScan algorithm is not novel: we are aware of at least one TREC 2009 team that adopted the same approach [9] (cf. [31]). However, Craswell et al. [9] did not provide a thorough comparison with alternative batch retrieval strategies as we have in this work.

Results of the DScan algorithm on the cluster described in Section 4.2 are presented in Table 2, under the column "DScan". Running times (in seconds) averaged over three trials are shown, along with the standard deviations in square brackets (same queries as before). Table 3 shows the amount of intermediate data that is shuffled from the mappers to the reducers. Running times vs. number of queries is also plotted in Figure 6. Although significant variance was observed in the different trials due to cluster idiosyncrasies, a linear regression nevertheless models the scaling characteristics of the algorithm quite well. The DScan algorithm is always slower than the PScan algorithm, but faster than the sequential query evaluation algorithm on Robust04. However, recall that the DScan algorithm operates directly over raw document vectors: by the time it takes to index the collection, we could have already completed a batch *ad hoc* retrieval run on 500 queries. This is illustrated in Figure 7, which shows the total running time of *ad hoc* retrieval using both PScan (including the time taken by the LP indexing algorithm) and DScan, ignoring the common preprocessing time.

6.2 Discussion

We were surprised at how competitive the DScan algorithm was, particularly for smaller testsets. For larger testsets, it might not make sense because indexing time can be amortized across many batch runs. However, we believe that the DScan algorithm can be more competitive than our results suggest. The algorithm is not bound by disk bandwidth, but rather processor floating-point performance. This conclusion was reached by a simple experiment: we ran a MapReduce job over the document vectors of the first ClueWeb09 English segment that did not do anything other than decoding the vectors (no output, no reducers). This took less than four minutes, which quantifies the overhead of scanning the compressed collection representation. Most of the rest of the time, we infer, is spent on actually computing scores. As our previous benchmarks have shown, the raw performance of each processor in the cluster is comparable to that of a three year old commodity laptop. This suggests that with current-generation hardware, DScan running times may be reduced substantially. While the DScan algorithm is not novel, we are not aware of any previous studies comparing different retrieval approaches (and factoring in indexing costs) in the way that we have.

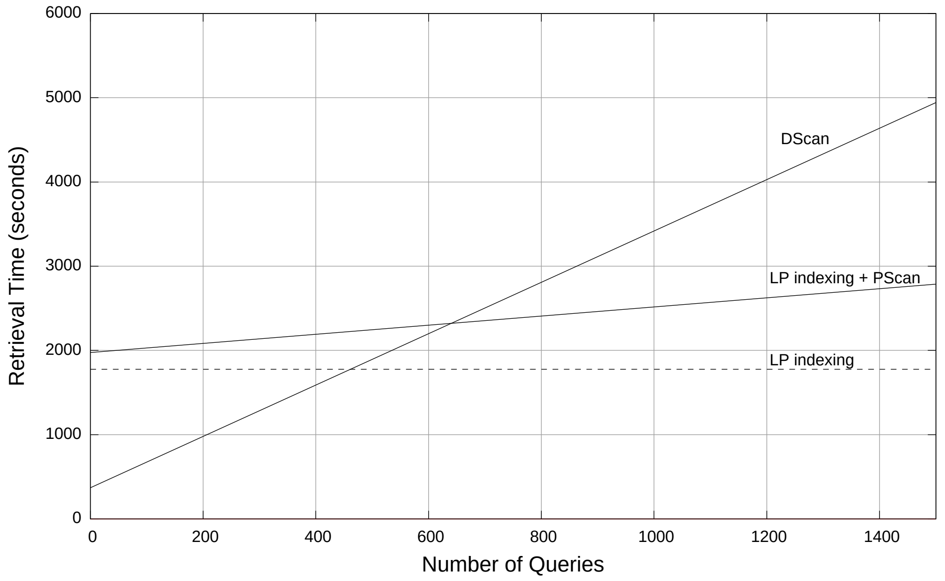

Figure 7: Total running times (ignoring preprocessing) of *ad hoc* retrieval using the PScan and DScan algorithms. Notice that PScan requires the indexing step.

Although DScan is slower than PScan, it has one substantial advantage that we should point out: modern retrieval algorithms, especially machine-learned models [21], take into account a multitude of document features beyond simple *tfs* and *dfs*—features such as PageRank, HITS, document quality scores, etc. At ranking time, the retrieval algorithm needs access to these features—this is a challenge since features do not generally fit into memory, necessitating some type of distributed low-latency key-value store (and most researchers do not have access to such infrastructure). This problem does not exist for the DScan algorithm, since these additional features can be stored as part of the compressed document vectors and readily accessible by the ranking function. This means that as researchers are developing new learning-to-rank algorithms, they can focus on the features themselves, as opposed to how to make them available.

7 Conclusions

To summarize, we see three major contributions of our work: First, we describe and evaluate the performance of two alternative MapReduce inverted indexing algorithms. Our results show that inverted indexing fits well in the MapReduce framework and that our algorithms scale linearly, at least in the collection sizes we examined (up to 102m documents, 3 TB). These results are not surprising, but we are not aware of any other work that has presented similarly detailed evaluations of alternative MapReduce indexing algorithms at this scale. In qualitative terms, we found that these two algorithms are instructive in teaching us about tradeoffs in designing scalable MapReduce algorithms.

Second, we reexamine aspects of the standard IR research workflow and demonstrate alternative approaches to batch *ad hoc* query evaluation that involve brute force sequential scans: over postings (PScan) and over document vectors (DScan). There are compelling merits to both, which provide interesting models for future IR research. These results, coupled with results from the indexing experiments, provide an accurate estimate on the time cost of running IR experiments from end to end. Of course, neither of the approaches are applicable to interactive retrieval, but even when online search is the end goal, batch runs still form an important part of the development process.

Finally, implementations of algorithms in this paper are part of Ivory, an open-source toolkit for web-scale retrieval, that we are excited to share with the community.[9] We hope that this contributes a small step in moving the field forward.

8 Acknowledgments

This work was supported in part by the NSF under award IIS-0836560 and IIS-0916043; Google and IBM, via the Academic Cloud Computing Initiative (ACCI). Any opinions, findings, conclusions, or recommendations expressed in this paper are the authors' and do not necessarily reflect those of the sponsors. The last author is grateful to Esther and Kiri for their loving support.

References

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. In *Proceedings of the 35th International Conference on Very Large Data Base (VLDB 2009)*, pages 922–933, Lyon, France, 2009.

[2] R. Baeza-Yates, C. Castillo, F. Junqueira, V. Plachouras, and F. Silvestri. Challenges on distributed web retrieval. In *Proceedings of the IEEE 23rd International Conference on Data Engineering (ICDE 2007)*, pages 6–20, Istanbul, Turkey, 2007.

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

[4] T. Brants, A. C. Popat, P. Xu, F. J. Och, and J. Dean. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 858–867, Prague, Czech Republic, 2007.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of the Seventh International World Wide Web Conference (WWW 7)*, pages 107–117, Brisbane, Australia, 1998.

[6] S. Büttcher, C. L. A. Clarke, and G. V. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines.* MIT Press, Cambridge, Massachusetts, 2010.

[7] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-Reduce-Merge: Simplified relational data processing on large clusters. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 1029–1040, Beijing, China, 2007.

[8] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Ng, and K. Olukotun. Map-Reduce for machine learning on multicore. In *Advances in Neural Information Processing Systems 19 (NIPS 2006)*, pages 281–288, Vancouver, British Columbia, Canada, 2006.

[9] N. Craswell, D. Fetterly, M. Najork, S. Robertson, and E. Yilmaz. Microsoft Research at TREC 2009. In *Proceedings of the Eighteenth Text REtrieval Conference (TREC 2009)*, Gaithersburg, Maryland, 2009.

[10] 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.

[9] Available for download at http://ivory.cc/.

[11] C. Dyer, A. Cordova, A. Mont, and J. Lin. Fast, easy, and cheap: Construction of statistical machine translation models with MapReduce. In *Proceedings of the Third Workshop on Statistical Machine Translation at ACL 2008*, pages 199–207, Columbus, Ohio, 2008.

[12] T. Elsayed, J. Lin, and D. Oard. Pairwise document similarity in large collections with MapReduce. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics (ACL 2008), Companion Volume*, pages 265–268, Columbus, Ohio, 2008.

[13] 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.

[14] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A peta-scale graph mining system—implementation and observations. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining (ICDM 2009)*, pages 229–238, Miami, Floria, 2009.

[15] A. Kimball, S. Michels-Slettvet, and C. Bisciglia. Cluster computing for Web-scale data processing. In *Proceedings of the 39th ACM Technical Symposium on Computer Science Education (SIGCSE 2008)*, pages 116–120, Portland, Oregon, 2008.

[16] B. Langmead, M. C. Schatz, J. Lin, M. Pop, and S. L. Salzberg. Searching for SNPs with cloud computing. *Genome Biology*, 10(R134), 2009.

[17] J. Lin. Exploring large-data issues in the curriculum: A case study with MapReduce. In *Proceedings of the Third Workshop on Issues in Teaching Computational Linguistics (TeachCL-08) at ACL 2008*, pages 54–61, Columbus, Ohio, 2008.

[18] 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)*, pages 155–162, Boston, Massachusetts, 2009.

[19] J. Lin, D. Metzler, T. Elsayed, and L. Wang. Of Ivory and Smurfs: Loxodontan MapReduce experiments for web search. In *Proceedings of the Eighteenth Text REtrieval Conference (TREC 2009)*, Gaithersburg, Maryland, 2009.

[20] J. Lin and M. Schatz. Design patterns for efficient graph algorithms in MapReduce. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs Workshop (MLG-2010)*, pages 78–85, Washington, D.C., 2010.

[21] T.-Y. Liu. Learning to rank for information retrieval. *Foundations and Trends in Information Retrieval*, 3(3):225–331, 2009.

[22] C. Manning, P. Raghavan, and H. Schütze. *An Introduction to Information Retrieval.* Cambridge University Press, 2008.

[23] R. M. C. McCreadie, C. Macdonald, and I. Ounis. Comparing distributed indexing: To MapReduce or not? In *Proceedings of the 7th Workshop on Large-Scale Distributed Systems for Information Retrieval (LSDS-IR'09) at SIGIR 2009*, 2009.

[24] S. Melnik, S. Raghavan, B. Yang, and H. Garcia-Molina. Building a distributed full-text index for the web. In *Proceedings of the 10th International World Wide Web Conference (WWW 10)*, pages 396–406, Hong Kong, Hong Kong, 2001.

[25] A. Moffat, W. Webber, and J. Zobel. Load balancing for term-distributed parallel retrieval. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2006)*, pages 348–355, Seattle, Washington, 2006.

[26] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo. PLANET: Massively parallel learning of tree ensembles with MapReduce. In *Proceedings of the 35th International Conference on Very Large Data Base (VLDB 2009)*, pages 1426–1437, Lyon, France, 2009.

[27] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 35th ACM SIGMOD International Conference on Management of Data*, pages 165–178, Providence, Rhode Island, 2009.

[28] B. Ribeiro-Neto, E. S. Moura, M. S. Neubert, and N. Ziviani. Efficient distributed algorithms to build inverted files. In *Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 1999)*, pages 105–112, Berkeley, California, 1999.

[29] S. E. Robertson, S. Walker, S. Jones, M. Hancock-Beaulieu, and M. Gatford. Okapi at TREC-3. In *Proceedings of the 3rd Text REtrieval Conference (TREC-3)*, pages 109–126, Gaithersburg, Maryland, 1994.

[30] M. C. Schatz. CloudBurst: Highly sensitive read mapping with MapReduce. *Bioinformatics*, 25(11):1363–1369, 2009.

[31] D. Shasha and T.-L. Wang. New techniques for best-match retrieval. *ACM Transactions on Information Systems*, 8(2):140–158, 1990.

[32] A. Tomasic and H. Garcia-Molina. Performance of inverted indices in shared-nothing distributed text document information retrieval systems. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 8–17, San Diego, California, 1993.

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

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

[35] J. Wolfe, A. Haghighi, and D. Klein. Fully distributed EM for very large datasets. In *Proceedings of the 25th International Conference on Machine Learning (ICML 2008)*, pages 1184–1191, Helsinki, Finland, 2008.

[36] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(6):1–56, 2006.