Of Ivory and Smurfs: Loxodontan MapReduce Experiments for Web Search

Jimmy Lin,[1] Donald Metzler,[2] Tamer Elsayed,[1] and Lidan Wang[1]
[1]University of Maryland, College Park
[2]Yahoo! Research
jimmylin@umd.edu, metzler@yahoo-inc.com, {telsayed,lidan}@cs.umd.edu

ABSTRACT

This paper describes Ivory, an attempt to build a distributed retrieval system around the open-source Hadoop implementation of MapReduce. We focus on three noteworthy aspects of our work: a retrieval architecture built directly on the Hadoop Distributed File System (HDFS), a scalable MapReduce algorithm for inverted indexing, and webpage classification to enhance retrieval effectiveness.

1. INTRODUCTION

It is commonly acknowledged that web-scale collections have outgrown the capabilities of individual machines, necessitating the use of clusters to tackle basic problems in information retrieval. Although search engine and other internet companies have long recognized and adapted to this fact, the academic community is just beginning to transition from single-machine to cluster-based systems. One previous impediment to progress was the availability of data: the largest collections available to researchers could be comfortably indexed on a typical server-class machine, obviating the need for clusters. The release of the 25 terabyte, one billion page ClueWeb09 collection, however, has forced researchers to think more seriously about cluster-based distributed retrieval solutions. This is a good sign, as it will propel the field forward.

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 [5] 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) 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). Key-value pairs form the processing primitives. 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.

Under this framework, a programmer needs only to provide implementations of the mapper and reducer. On top of a distributed file system [6], the execution framework transparently handles all other aspects of execution on clusters ranging from a few to a few thousand cores. 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.

Hadoop,[1] the open-source implementation of MapReduce, has gained immense popularity as an accessible, cost-effective framework for processing large datasets.[2] This paper describes an attempt to build a distributed retrieval system around the Hadoop ecosystem. Retrieval systems designed to run on single machines make certain assumptions about characteristics of system resources (latency, bandwidth, capacity) and relationships between them. We used this opportunity to rethink some of these assumptions in a distributed environment, as the first step in building a scalable information retrieval toolkit for the future.

The system we have developed is called Ivory, which integrates Metzler's SMRF (Search using Markov Random Fields) retrieval engine [14, 13].[3] Ivory has been released under an open source license and can be freely downloaded from the web. This paper discusses three noteworthy aspects of our work: a retrieval architecture built directly on HDFS (Section 2), a scalable MapReduce algorithm for inverted indexing (Section 3), and post-processing of results to suppress adult content, spam, and low quality pages (Section 4). Experimental results are discussed in Section 5.

2. RETRIEVAL ARCHITECTURE

Given a user query, retrieval involves fetching postings lists corresponding to query terms and computing query-document scores according to the specified retrieval model. The postings list for each query term must be traversed, in a manner determined by the organization of the index and the query evaluation strategy.

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

[2]To be precise, MapReduce is used to refer to the programming model in general, while Hadoop refers to the specific open-source implementation. Along the same lines, the distributed file system (DFS) is used to refer to the underlying storage substrate in general, while GFS [6] and HDFS are used to refer to specific implementations.

[3]In the Maryland tradition of whimsical titles for TREC papers: The mascot for Hadoop is an elephant, and African elephants belong to the genus *Loxodonta*. And yes, it is clear that Hadoop is an *African* elephant and not of the *Asian* variety.

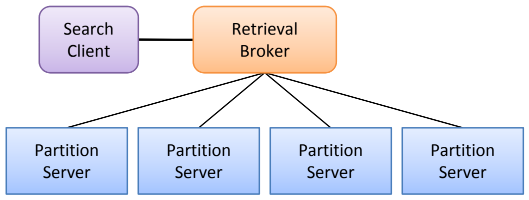


Figure 1: Illustration of a simple broker-mediated, document-partitioned retrieval architecture.

Beyond collections of a certain size, it is not practical to store the entire index on a single machine. The standard distributed solution is a broker-mediated, document-partitioned retrieval architecture, illustrated in Figure 1. The entire document collection is divided into a number of partitions (sometimes called "shards"), and indexes are built for each partition separately; a server is responsible for searching each index, independent of the others. The interactions between a search client and the partition servers are mediated by the broker. In the standard query-response cycle, the client issues a query to the broker, which then distributes the query to all partition servers in parallel. Each server computes a ranked list on its assigned document partition independently, and the results are passed back to the broker. The broker merges the results and returns the final ranked list to the client. Although this "vertical" document partitioning strategy is often used in conjunction with a "horizontal" tiered partitioning strategy (i.e., by document quality), we do not consider that additional complexity in this work.

2.1 The Distributed Environment

In the early stages of our project, we noticed a fundamental mismatch between the standard document-partitioned retrieval architecture and characteristics of the MapReduce environment.

First, consider the problem of query evaluation on each individual partition. MapReduce, which was designed for batch processing, is not appropriate for this task. In Hadoop, it can take tens of seconds for mappers to even launch, since tasks must be queued at the jobtracker before they can be assigned to individual workers. Furthermore, the current design of Hadoop limits the rate at which new map tasks can be spawned. For the sub-second query latency expected by searchers today, there is no obvious way to implement workable retrieval algorithms in MapReduce.

Moreover, the MapReduce software ecosystem presents additional challenges for real-time retrieval algorithms. An integral component of MapReduce is the underlying distributed file system (DFS), which was designed around a number of assumptions about the workload. Since it is assumed that MapReduce jobs perform batch-oriented processing of large datasets, the distributed file system was optimized for high sustained throughput and not low-latency random access.

The DFS employs a simple master-slave architecture and stores files in fixed-size blocks. The master (called the namenode in HDFS) stores metadata and namespace mappings to data blocks, which are themselves stored on the local disks of the slaves (called datanodes in HDFS). The master only communicates metadata; data transfer occurs directly between the application client and the relevant datanode. To the extent possible, the MapReduce scheduler starts map tasks on the machines that hold the data block to be processed, thus guaranteeing high sustained throughput since the task reads from local disk.

This design makes it difficult to achieve low-latency random access to DFS data from an arbitrary application client (e.g., a partition server). To access a random position in a file (e.g., looking up a postings list), the client must first contact the namenode to locate the relevant data block. Then, the client must contact the appropriate datanode to obtain the requested data. In addition to a disk seek on the datanode, the entire process involves round-trip communications with multiple machines and data transfer over the network. This problem cannot be solved by simply running the application client on the datanode that has the block stored locally. The distributed file system, by design, spreads data blocks across nodes in the cluster (to ensure reliability, to provide better locality, etc.), and therefore, for even moderately-large files, no single datanode will hold the entirety of a file's contents.

The design of the distributed file system is directly at odds with the requirements for query evaluation, since low-latency random access to postings is necessary. Even though MapReduce provides a nice framework for building inverted indexes, the above discussion suggests that the DFS makes a poor storage substrate for retrieval. This is indeed the conventional wisdom.

The typical solution to this problem is to employ a separate architecture for retrieval. Once indexes have been built using Hadoop and written out to HDFS, they are then copied over to another cluster (onto standard POSIX file systems) to support retrieval. Typically, this involves copying individual partition indexes onto the local disk of the corresponding partition server. An example is Katta, which is a system for managing distributed Lucene indexes.[4] This solution, while certainly workable, suffers from two major drawbacks, discussed below.

First, this solution requires the maintenance of two separate architectures: one for batch processing and another for real-time querying. This also requires splitting hardware resources, making it difficult to bring all available capacity to bear on a large problem. Although it is possible for the same physical machines to serve "double duty", such a setup may have unpredictable performance effects as multiple processes are competing for the same cores, memory, disk, and network. Furthermore, maintaining independent architectures will inevitably require keeping multiple copies of the data. For example, the collection needs to reside in HDFS to support indexing, but a separate copy may be needed on the retrieval cluster so that users can examine results.

Second, the two-architectures solution results in a complex workflow that necessitates copying large indexes over the network, thus complicating data management. Such a setup requires a good mechanism for versioning and metadata control, because duplicate data may be residing on independent systems at any given time. Workflow management is notoriously difficult in a rapidly-evolving research environment. Furthermore, the non-trivial latencies involved in copying indexes over the network to local disks is not conducive to the rapid turnaround times needed for IR experiments.

[4]http://katta.sourceforge.net/

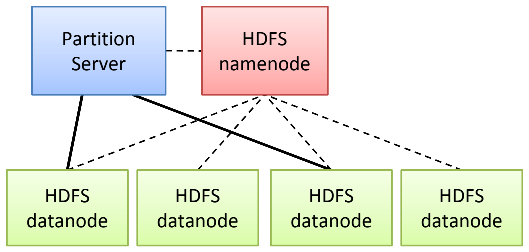


Figure 2: Illustration of Ivory's distributed architecture that involves reading postings directly from HDFS (data transfer shown as solid lines; metadata communication shown as dotted lines).

2.2 Challenging Conventional Wisdom

In developing the Ivory system, we decided to challenge conventional wisdom and explore whether it was indeed feasible to "run" query evaluation algorithms directly on HDFS-stored indexes. In addition, we wondered whether it was possible to use the same Hadoop cluster for both batch-oriented processing (e.g., indexing) and for real-time services (e.g., retrieval).

Despite the discussion above, there were two additional observations that led us to believe that such an architecture was at least worth trying. The first bit of evidence comes from BigTable [4], which is a sparse, distributed, persistent multidimensional sorted map built on top of the Google File System. BigTable is used for a number of production services with low latency requirements (e.g., Google Earth). Although very different from the distributed retrieval architecture we explore here, BigTable demonstrates that there is no principled reason why DFS latencies cannot be hidden by higher-level applications. The second bit of evidence comes from physical cluster architecture—as it turns out, bandwidth between a machine and the disks of any other rack-local machine is surprisingly competitive to the bandwidth of local disks (since for the most part, rack-level switches are not oversubscribed internally). A recent monograph by Barroso and Hölzle [2] discusses these observations in more detail. Operationally, this means that reading data off the disk of another machine on the same rack isn't much slower than reading data off the local disk.

The retrieval component in Ivory comes from Metzler's SMRF (Search using Markov Random Fields) engine, which was used in a number of previous studies examining the effectiveness of Markov Random Fields for information retrieval, but has not been available as open-source software until now. The major modification to the previous implementation was to fetch postings directly from HDFS instead of local disk. This is shown in Figure 2, which focuses on an individual partition server. As is standard in most retrieval engines, the vocabulary is held in memory. With front-coding, this is relatively easy to accomplish, even for large collections. The vocabulary holds byte offsets into HDFS-stored index files that correspond to locations of postings lists. The fetching of a postings list involves first contacting the namenode for the block location, and then contacting the datanode itself for the actual data—which is no different from any other HDFS read.

Within a Hadoop cluster environment, we still need to address the issue of how partition servers and the broker are initialized—given that the only point of contact between a client and the Hadoop cluster is the jobtracker. The solution we devised involves embedding servers in MapReduce jobs (albeit degenerate ones in most cases).

Partition servers can be spawned as a MapReduce job that runs mappers but no reducers. Embedded inside each mapper is a server that handles queries over a TCP connection and accesses postings directly on HDFS (as described above). To start multiple partition servers, we create a MapReduce job that maps over a configuration file specifying the locations of the partition indexes. By appropriately configuring the job, a number of mappers equal to the number of document partitions is spawned. Each mapper reads in the location of the partition index, initializes a query engine, and then launches into an infinite service loop waiting for incoming TCP connections. The Hadoop execution framework is in essence co-opted into serving as a simple scheduler. However, we have little control over which cluster nodes the mappers are launched on. Fortunately, this situation is easy to rectify: when each mapper launches, it first writes its host information into a known DFS location. After all the partition servers have been initialized, the broker can be launched as a 1-mapper/0-reducer MapReduce job, reading the host information of all the partition servers and completing the distributed broker architecture.

Our solution addresses many of the issues with the two-architectures solution discussed in Section 2.1. Instead of maintaining a Hadoop cluster for indexing and another cluster for retrieval, we can accomplish both within a homogeneous environment. This allows us to better utilize available hardware resources and simplifies data management and workflow. The potential downside is, of course, degraded query performance due to reading postings remotely. Section 5.1 reports the performance of this architecture.

3. INVERTED INDEXING

Dean and Ghemawat's original paper [5] showed that MapReduce was designed from the very beginning with inverted indexing as an application. Although very little space was devoted to describing the algorithm, it is relatively straightforward to fill in the missing details: this basic MapReduce algorithm for inverted indexing is shown in Figure 3. Input to the mappers consists of document numbers[5] (keys) paired with the document content (values). Inside the mapper, each document is tokenized, stemmed, and filtered for stopwords. Terms are processed sequentially to build a histogram of term frequencies (implemented as an associative array). The algorithm then iterates over all terms: for each, a posting consisting of the document number and the term frequency is created (denoted by $\langle n, H\{t\}\rangle$ in the 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 *tf*, but this can easily be augmented with term position information to build positional indexes.

In the sort and shuffle phase, the MapReduce runtime performs a large, distributed "group by" of the postings by term. Without any additional effort by the programmer, the execution framework brings together all postings associated

[5]We assume that documents are sequentially numbered from 1 to n, where n is the number of documents in the collection.

```
class Mapper
    method Map(docno $n$, doc $d$)
        $H \leftarrow$ new AssociativeArray
        for all term $t \in$ doc $d$ do
            $H\{t\} \leftarrow H\{t\} + 1$
        for all term $t \in H$ do
            Emit(term $t$, posting $\langle n, H\{t\}\rangle$)
class Reducer
    method Reduce(term $t$, postings $[\langle n_1, f_1\rangle \ldots]$)
        $P \leftarrow$ new List
        for all posting $\langle n, f\rangle \in$ postings $[\langle n_1, f_1\rangle \ldots]$ do
            $P$.Append($\langle n, f\rangle$)
        $P$.Sort()
        Emit(term $t$, postings $P$)
```

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

with the same term. This tremendously simplifies the task of the reducer, which gathers the postings and writes them to disk. The reducer begins by initializing an empty list and then appends all postings associated with the same term (key) to the list. The postings are then sorted (depending on type of index, by document number or term frequency) and written to disk (appropriately compressed).

The MapReduce programming model provides a very concise expression of the inverted indexing algorithm, and can be implemented in a couple of dozen lines of code in Hadoop. Such an implementation can be successfully completed as a programming assignment in a computer science course for advanced undergraduates and first-year graduate students [7, 9], which illustrates the simplicity of the the algorithm. In a traditional indexer (i.e., not implemented in MapReduce), significant attention must be devoted to the task of grouping postings by term, given constraints imposed by memory and disk (that memory capacity is limited, disk seeks are slow, sequential operations are preferred, etc.). In MapReduce, the programmer does not need to worry about any of these issues—the heavy lifting of grouping postings is handled by the execution framework.

3.1 Scalable MapReduce Indexing Algorithm

There is, however, a significant bottleneck in the basic MapReduce algorithm for inverted indexing: it assumes that there is sufficient memory to hold all postings associated with the same term. 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 postings can be written out to disk.

Since Ivory builds document-sorted indexes, we restrict our attention to the problem of sorting postings by ascending document number. Since the execution framework guarantees that keys arrive at each reducer in sorted order, one way to overcome the scalability bottleneck is to let the MapReduce runtime do the sorting. Instead of emitting key-value pairs of the following type:

(term t, posting $\langle n, f\rangle$)

We emit intermediate key-value pairs of the type:

(tuple $\langle t, n\rangle$, tf f)

```
class Mapper
    method Map(docno $n$, doc $d$)
        $H \leftarrow$ new AssociativeArray
        for all term $t \in$ doc $d$ do
            $H\{t\} \leftarrow H\{t\} + 1$
        for all term $t \in H$ do
            Emit(tuple $\langle t, n\rangle$, tf $H\{t\}$)
class Reducer
    method Initialize
        $t_{prev} \leftarrow \emptyset$
        $P \leftarrow$ new PostingsList
    method Reduce(tuple $\langle t, n\rangle$, tf $[f]$)
        if $t \neq t_{prev} \wedge t_{prev} \neq \emptyset$ then
            Emit(term $t$, postings $P$)
            $P$.Reset()
        $P$.Add($\langle n, f\rangle$)
        $t_{prev} \leftarrow t$
    method Close
        Emit(term $t$, postings $P$)
```

Figure 4: Pseudo-code of a scalable inverted indexing algorithm in MapReduce (slightly simplified from the actual algorithm in Ivory).

In other words, the key is a tuple containing the term and the document number, and 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 docno n. Additionally, we need a custom partitioner to ensure that all tuples with the same term are shuffled to the same reducer. With these two changes, the MapReduce execution framework ensures that the postings arrive in the correct order. This, combined with reducers preserving state across multiple keys, allows compressed postings to be written with minimal memory usage.

The revised MapReduce inverted indexing algorithm is shown in Figure 4. The mapper remains unchanged for the most part, other than differences in the intermediate key-value pairs. 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. 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 HDFS. The final postings list must be written out in the Close method.

In our algorithm, the key space is partitioned by term; that is, all keys with the same term are sent to the same reducer. Since in Hadoop each reducer writes its output in a separate file on HDFS, our final index will be split across r files, where r is the number of reducers. In another MapReduce pass over these files, we construct a postings forward index to store the byte offset position of each postings list. This is used during retrieval to fetch postings that correspond to query terms. There is no need to consolidate the r files, since the postings forward index can keep track of which file a term's postings list is found in.

Three more details complete the description of Ivory's MapReduce indexing algorithm: positional information, document length data, and parameter setting for Golomb compression. First, positional indexes can be built by simply replacing the intermediate value f (term frequency) with an array of term positions; otherwise, no additional modifications are needed to the algorithm.

Second, since almost all retrieval models take into account document length, this information needs to be computed. Although it is straightforward to express this computation as another MapReduce job, this task can actually be folded into the inverted indexing process. When processing the terms in each document, the document length is known, and can be written out as "side data" directly to HDFS. We take advantage of the ability for a mapper to hold state across the processing of multiple documents in the following manner: an in-memory associative array is created to store document lengths, which is populated as each document is processed. When the mapper finishes processing input records, document lengths are written out to HDFS (i.e., in the CLOSE method). Thus, document length data ends up in m different files, where m is the number of mappers; these files are then consolidated into a more compact representation.

Finally, parameters must be appropriately set for compression of the postings lists. The prescribed best practice is to use Golomb compression on first order document number differences (i.e., *d*-gaps) [16, 17]. The difficulty, however, is that Golomb compression requires two parameters: the size of the document collection and the number of postings for a particular postings list (i.e., *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 postings are encoded. A two-pass solution that involves first buffering the postings (in memory) would suffer from the memory bottleneck we've 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 mapper emit special keys of the form $\langle t, * \rangle$ to communicate partial document frequencies. This is accomplished in a manner similar to the computation of document lengths. 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.

3.2 Merging Results Across Partitions

The broker in a distributed document-partitioned architecture is responsible for merging results from each of the partition servers. We explored two separate algorithms for accomplishing this.

The first approach, which we call the *independent fusion* strategy, is to view results merging as a federated search problem, treating each partition as an independent collection. This approach simplifies index construction, but makes document scores across partitions difficult to compare directly. To correct for this, raw scores are normalized, per partition, using the z-score transformation as follows [8]:

$$S^* = (S - \mu_o)/\sigma$$

where S is the raw score, μ_o is the sample mean of the raw scores, σ^2 is the sample variance, and S^* is the normalized score. The normalized scores are now considered samples of a standard normal distribution. The broker returns a combined ranked list by sorting all of the returned documents from all partitions based on their normalized scores.

The other strategy for merging results is called *global statistics*, which involves distributing global collection statistics to each of the partition indexes. First, each of the partition indexes are built independently. Then, a MapReduce job maps over *all* the partition indexes to compute global statistics (the global *df* and *cf* for each term and the size of the entire collection). Finally, global statistics are propagated back to each partition index. This is also accomplished with MapReduce: we map over each postings list, and inside each mapper the global statistics are loaded into memory. A new version of the index is written with the updated statistics (no reducers are required). This simple process is repeated for each partition. Given that MapReduce can take advantage of the aggregate disk throughput of multiple machines, these MapReduce jobs are surprisingly fast.

The advantage of the *global statistics* approach is that document scores generated in each partition are exactly the same as document scores in a single global index that spans all partitions—at least for the retrieval models used in our experiments (*bm25* and query-likelihood). Thus, no additional score manipulation is necessary, and the broker simply resorts results from the partition servers and returns the final reranked list to the client.

3.3 Alternative Algorithm Designs

Our inverted indexing algorithm in MapReduce represents a single point in the design space of possible approaches to the task. We discuss alternatives here, which primarily vary in the extent to which they take advantage of the large distributed group and sort operations built into the MapReduce execution framework.

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 (i.e., index construction may involve multiple flushes to local disk and on-disk merge sorts outside of MapReduce). 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 and written out as the final index. We did not pursue this option since it seemed like an incremental improvement over known indexing algorithms, and instead opted to develop an indexer from scratch to more fully explore the MapReduce programming model.

Another relatively straightforward adaptation of a single-machine indexer is demonstrated by Nutch.[6] 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 independently. In this approach, the number of reducers specifies the number of partitions—which limits the degree of parallelization that can be achieved.

Next, reconsider our critique of Dean and Ghemawat's MapReduce algorithm shown in Figure 3. Although we pointed out the scalability bottleneck associated with sorting the postings in the reducer, in actuality, there is no principled reason why this needs to be an in-memory sort. Instead, one could implement a multi-pass on-disk merge sort within the reducer. However, this is exactly what the MapReduce execution framework does in the sort and shuffle phase, so it makes sense to offload the processing.

Finally, we note that independently and roughly concurrently, McCreadie et al. [12] proposed a MapReduce inverted indexing algorithm based on emitting partial postings lists. The reducer receives partial postings lists and merges them into final postings lists.

Abstractly, inverted indexing can be viewed as a massive group and sort of individual postings. MapReduce indexing algorithms vary in what component performs these operations: the mappers and reducers, the execution framework, or a combination of both. In the first approach, the developer must shoulder at least some of the burden of grouping and sorting key-value pairs, but can take advantage of application-specific optimizations (e.g., efficient δ compression schemes). The downside, however, is added code complexity and potential scalability bottlenecks that may not be apparent. We have taken the second approach, and completely offloaded the grouping and sorting operations onto the MapReduce execution framework. Although this does not allow us to take advantage of application-specific optimizations, it does significantly simplify code. Moreover, scalability is ensured since we are taking advantage of mechanisms built directly into the programming model. Nevertheless, there is likely to be a middle ground (the third option) that balances simplicity and efficiency—which seems like a promising direction for future work.

4. ADULT, SPAM, AND QUALITY

Given the large size of the ClueWeb09 collection, we hypothesized that traditional retrieval models would return a large amount of spam, adult material, and generally low quality documents that would severely degrade retrieval effectiveness. However, to properly test our hypothesis, we would need highly accurate spam, adult, and document quality classifiers or predictors. Rather than build classifiers ourselves, we used Yahoo!'s proprietary adult, spam, and document quality classifiers to post-process the ranked lists produced using Ivory.

Due to their proprietary nature, we are unable to provide the exact details of how these classifiers work, other than to say that they are machine-learned models that make use of many features and were trained using a very large amount of manually labeled data. We normalized the output of these classifiers to provide a score between 0 and 1, with 0 denoting not spam / not adult / low quality and 1 denoting spam / adult / high quality.[7] As a reference, Qi and Davison [15] provide a recent survey on web page classification.

Given the lack of proper training data on the ClueWeb09 collection, we utilized the output of the classifiers in a simple, heuristic manner. We assumed that spam and adult documents would never be judged relevant, so we used the spam and adult classifiers to filter such documents from the result set. Furthermore, we used the output of the document quality classifier to adjust the original document scores assigned by Ivory. Results were rescored as follows:

$$S'(Q,D) = \begin{cases} S(Q,D)\cdot f_q(D)^{\alpha_q} & f_a(D) < \tau_a \wedge f_s(D) < \tau_s \\ -\infty & otherwise \end{cases}$$

where $S'(Q,D)$ is the new score, $S(Q,D)$ is the original score, $f_s(D)$ is the spam classifier score, $f_a(D)$ is the adult classifier score, $f_q(D)$ is the document quality classifier score, τ_s is the spam threshold, τ_a is the adult threshold, and α_q is quality score adjustment factor. The free parameters are τ_s, τ_a, and α_q: different settings will lead to different degrees of filtering and reranking.

We considered two different settings for these parameters. The first, which we call *conservative*, corresponds to $\tau_a = 0.9, \tau_s = 0.9, \alpha_q = 0.1$. The second, which we call *moderate* and uses $\tau_a = 0.75, \tau_s = 0.75, \alpha_q = 0.25$. These settings were manually chosen after some preliminary experiments on a small development set of queries. To ensure that we return 1000 documents per query, we post-processed the top 2000 ranked documents.

5. RESULTS

Experiments were run on a cluster provided 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 99 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.

Since more detailed specifications of the cluster machines were not available, we decided to informally run our own performance benchmarks. An individual cluster node achieved a composite score of 442 on NIST's SciMark 2.0 benchmark,[8] averaged over 3 trials. For comparison, a laptop with a 2.6 GHz Core 2 Duo (T7800) processor[9] 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). However, Lin [10] reported that on a brute-force task involving repeated computation of dot products, each cluster node was significantly slower than the same laptop. While it is true that our applications are

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

[7]Note that the scales are reversed for quality, compared to spam/adult.

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

[9]2007 technology

Queries	Model	HDFS	local
Robust04	*bm25*	5.45s	8.25s
Robust04	QL	6.65s	10.0s
Web09	*bm25*	4.73s	6.65s
Web09	QL	5.60s	7.42s

Table 1: Average per-query running time on the first segment of ClueWeb09, comparing indexes stored on HDFS with indexes stored on local disk.

ID	Model	P@5	P@10
UMHOObm25GS	*bm25* (global)	0.1040	0.1420
UMHOObm25IF	*bm25* (fusion)	0.1240	0.1640
UMHOOqlGS	QL (global)	0.0920	0.1180
UMHOOqlIF	QL (fusion)	0.0800	0.1080

Table 2: Official retrieval effectiveness for baseline category A submissions based on trec_eval. Results of post-processing are shown in Table 3.

primarily IO-bound and not processor-bound, we suspect that the cluster consists of previous-generation machines. Performance figures presented below should be interpreted with this important caveat. The 99-node cluster contained 198 cores, which, with current dual-processor quad-core configurations, could fit into 25 machines—a far more modest cluster with today's technology, not to mention that modern processors would be substantially faster.

5.1 Efficiency

On the 99-node cluster, indexing time for the first English segment of the ClueWeb09 collection (∼50 million pages) was 145 minutes (averaged over three trials; the fastest and slowest running times differed by less than 10 minutes). The size of the full positional index was around 66 GB.

On the retrieval end, we compared the performance of two variants of our query engine: one that reads indexes from local disk, and one that reads indexes from HDFS (the architecture discussed in Section 2.2). Both conditions utilized a single processor core on the cluster, and therefore performance differences can be attributed to the different methods of postings access. Average time per query (across three trials) is shown in Table 1, for both queries from this year's web track (50 queries) and the 2004 robust track (100 queries) on the index built from the first English segment of ClueWeb09. We compared *bm25* and query-likelihood, and in each case fetched 2000 hits.

These performance results were surprising in that reading postings from local disk was actually slower than reading postings over HDFS. One benefit of HDFS is the ability to read postings corresponding to different query terms *in parallel*, since they may involve accessing different datanodes. Reading multiple postings in parallel doesn't make much sense in a single machine environment unless there are multiple disks, and even then, it requires the retrieval engine to model that fact explicitly. In contrast, parallel reads are transparently handled by the HDFS API. The HDFS experiments also benefited from caching, which makes repeated access of postings faster (for common query terms, and also across multiple experimental runs). Although HDFS itself does not provide caching, since it resides on top of Linux, caching is performed transparently at the OS level—we can take advantage of the aggregate Linux buffer caches of all HDFS datanodes "for free". For this reason, the HDFS results are perhaps overly optimistic; more experiments are required to tease apart the various factors that influence performance.

Nevertheless, results show that our distributed architecture is not only feasible, but may provide additional performance advantages over separate batch and real-time architectures. In addition, we expect random access latencies to improve over time as developers continue to improve HDFS.

5.2 Effectiveness

Our official category A submissions were divided into two types: baseline runs and post-processed runs. The baselines examined four conditions: {*bm25*, query-likelihood}×{global statistics, independent fusion} (the latter describes the results merging strategies outlined in Section 3.2). The English portion of the ClueWeb09 collection was divided into ten different segments, each of which formed a partition in our architecture. For *bm25*, we used $k_1 = 0.5$ and $b = 0.3$. For query likelihood, we used Dirichlet smoothing with $\mu = 1000$. Official results for baseline runs based on trec_eval are shown in Table 2. We were quite surprised that the independent fusion approach was more effective than the global statistics approach; this may be due to a bug, since the task of propagating global statistics back to the individual partition indexes introduced an additional layer of complexity. However, see additional discussions below.

The post-processed runs used the filtering and reranking strategy described in Section 4; official results based on trec_eval are shown in Table 3. The second column of the table shows which of the baseline runs were post-processed. Spam, adult, and quality scores were found in Yahoo!'s metadata store for approximately 95% of the URLs retrieved by the baseline runs. Note that the scores were computed over the version of the document at the time of run submission, which may differ from the crawled version in the ClueWeb09 collection.

Based on the Wilcoxon signed-rank test, we observe large and statistically-significant improvements ($p < 0.01$) in P@5 and P@10 for yhooumd09BGC and yhooumd09BGM, post-processed versions of the baseline *bm25* (with global statistics) run. Furthermore, moderate filtering was found to be more effective than conservative filtering. Moderate filtering was 10% better than conservative filtering for P@5 and 6% better for P@10 (both *n.s.*). Table 4 shows the queries from the yhooumd09BGM run that were the most improved, in terms of absolute P@10, as the result of post-processing. In almost every case, these queries initially retrieved no relevant items in the top 10, but found 7 or more after post-processing.

Somewhat surprisingly, the improvements observed for the yhooumd09BFM run were not statistically significant. One possible explanation for this is that the baseline system (i.e., UMHOObm25IF) retrieved many non-relevant documents that also happened to not be spam, adult, or low quality, thereby nullifying the effect of the filtering and reranking. Another possible explanation is that many of the z-transformed scores were close to zero, which caused our document quality score adjustments to have a negligible effect on the ranking.

Given the success of this simple, heuristic strategy, it is likely that a more formal learning to rank approach could

ID	Base	Setting	P@5	P@10
yhooumd09BFM	UMHOObm25IF: *bm25* (fusion)	Moderate	0.1520 (+23%)	0.1640 (0%)
yhooumd09BGC	UMHOObm25GS: *bm25* (global)	Conservative	0.3880 (+273%)*	0.3820 (+169%)*
yhooumd09BGM	UMHOObm25GS: *bm25* (global)	Moderate	0.4280 (+312%)*	0.4040 (+185%)*

Table 3: Official retrieval effectiveness for post-processed category A runs based on trec_eval (relative gains shown in parentheses). A single asterisk denotes a statistically-significant difference according the Wilcoxon signed-rank test at the $p < 0.01$ level.

		StatMAP Method				MTC Method			
ID	Model	MAP	MRP	MP@30	MnDCG	eMAP	eRprec	eP5	eP10
UMHOObm25B	*bm25*	0.2037	0.2848	0.3967	0.3718	0.0461	0.1048	0.3496	0.3849
UMHOOqlB	QL	0.1874	0.2761	0.3779	0.3416	0.0436	0.1027	0.2810	0.3395
UMHOOsd	MRF	$0.2142^{\triangle\circ}$	$0.3023^{\triangle\circ}$	$0.4272^{\triangle\bullet}$	$0.3885^{\blacktriangle\bullet}$	$0.0476^{\blacktriangle\bullet}$	$0.1068^{\blacktriangle\bullet}$	$0.3458^{\triangle\bullet}$	$0.3999^{\triangle\bullet}$
UMHOOsdp	MRF pruned	$0.2138^{\triangle\circ}$	$0.2993^{\triangle\circ}$	$0.4251^{\triangle\bullet}$	$0.3860^{\triangle\bullet}$	$0.0476^{\blacktriangle\bullet}$	$0.1068^{\blacktriangle\bullet}$	$0.3436^{\triangle\bullet}$	$0.3991^{\triangle\bullet}$

Table 5: Retrieval effectiveness for category B runs. Comparing the two MRF models to the two baseline models: $^{\blacktriangle}$ indicates significantly better than *bm25* ($p < 0.05$), $^{\triangle}$ indicates *n.s.*; $^{\bullet}$ indicates significantly better than QL ($p < 0.05$), $^{\circ}$ indicates *n.s.* (all significance tests performed with the Wilcoxon signed-rank test).

Query	Before	After
diversity	0.0	1.0
inuyasha	0.0	1.0
atari	0.0	1.0
dogs for adoption	0.0	1.0
dinosaurs	0.0	0.9
espn sports	0.1	0.9
euclid	0.0	0.8
appraisals	0.0	0.7
hoboken	0.0	0.7
the secret garden	0.0	0.7

Table 4: Queries most improved as the result of post-processing in terms of P@10.

result in even better retrieval effectiveness [11]. It would have been difficult to take such an approach this year, given the lack of training data on the ClueWeb09 collection, but it should be possible, at least to some extent, for future tasks that make use of the data.

Results for category B runs are shown in Table 5, based both on the statistical evaluation (StatMAP) method [1] and the Minimal Test Collection (MTC) method [3]. The first two models used features based on single term occurrences (*bm25* and query-likelihood), while UMHOOsd combined term-dependence features such as ordered and unordered phrases with individual term occurrences using the Markov Random Field (MRF) retrieval framework. The single-term, ordered, and unordered clique types used in the MRF were assigned weights of 0.82, 0.09, 0.09, respectively. In order to consider retrieval efficiency, in run UMHOOsdp we pruned cliques based on *idf*: if a term's *idf* was less than 0.12 then cliques containing the term were pruned. Although the pruning threshold of 0.12 is relatively conservative for web-scale collections, we did see a drop in query evaluation time compared to the full MRF model, without a significant impact on effectiveness. Our simple pruning technique was performed at query time and hence could be adapted to query-dependent characteristics.

Details of significance testing comparing the two MRF models and the two baseline models are also shown in Table 5. The pruned and unpruned MRF models were statistically indistinguishable, but both MRF models were significantly better than both baseline models for many metrics.

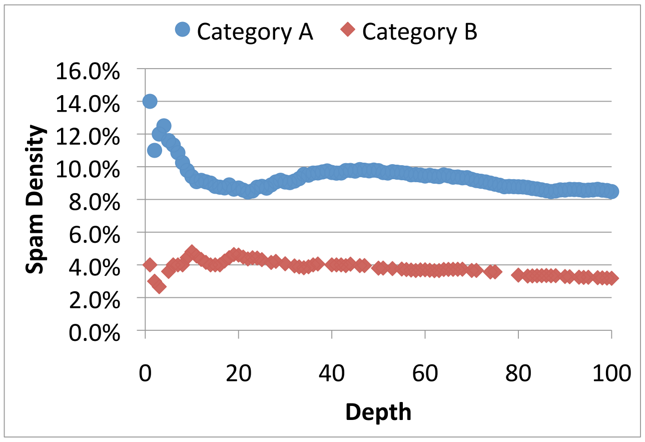


Figure 5: Spam density as a function of rank depth for category A (circle) and category B (diamond).

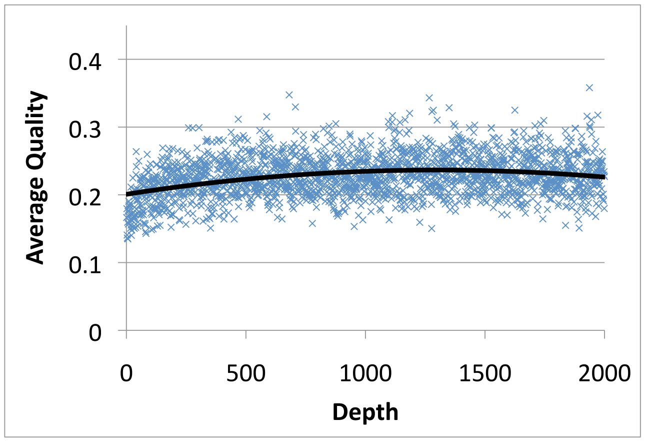
5.3 Category A vs. Category B Quality

In addition to using the Yahoo! classifiers to improve retrieval effectiveness, we also used them to compare the quality of the category A and category B document sets. In our first experiment, we compared the *spam density* of documents retrieved from category A and category B using the 50 queries. Spam density is defined as the percentage of results returned, up to a certain rank depth, that is filtered as spam. Figure 5 plots the spam density as a function of rank depth for both sets of documents (category A run with *bm25*, global statistics vs. category B run with *bm25*). First, the plot clearly shows that category A has a much higher spam density than category B across all ranks. This is not unexpected, as the ClueWeb09 collection represented a best-first crawl, so the larger category A document set contained documents that were lower in quality. Another interesting characteristic of the plot is that the spam den-

Spam	
Category A	Category B
appraisals (20.0%)	air travel information (10.0%)
poker tournaments (19.5%)	cheap internet (6.7%)
elliptical trainer (13.6%)	website design hosting (6.3%)
used car parts (12.7%)	cell phones (4.9%)
cell phones (12.4%)	poker tournaments (4.7%)

Adult	
Category A	Category B
french lick resort and casino (0.25%)	the current (1.85%)
toilet (0.15%)	toilet (0.45%)
cheap internet (0.15%)	french lick resort and casino (0.30%)
inuyasha (0.15%)	inuyasha (0.25%)
the secret garden (0.15%)	the secret garden (0.25%)

Table 6: Queries with highest density of spam (top) and adult content (bottom).



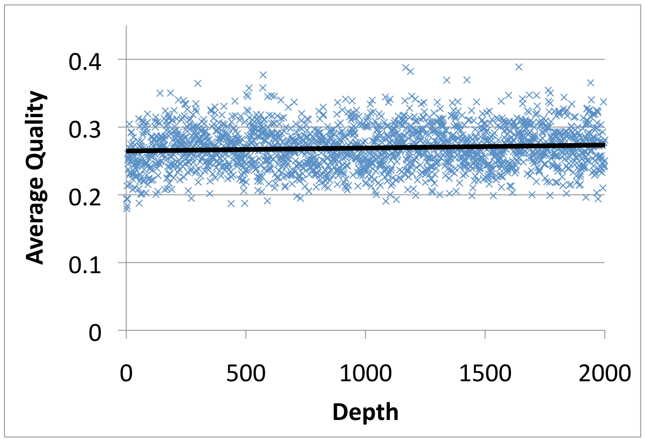


Figure 6: Average result quality vs. rank depth for category A (left) and category B (right); higher is better.

sity tends to be the highest at the top ranks and decreases farther down the ranked list. This suggests that traditional information retrieval models such as *bm25* are highly susceptible to spam and that spammers are very good at getting their documents ranked highly when ranking is based on text alone. Table 6 (top) shows the individual queries with the highest spam density for categories A and B (up to 2000 hits). The query-by-query analysis shows overlap in the spammable queries and reaffirms that spam is much more prevalent in category A.

We also measured the *adult density*, which is the percentage of results, to a fixed rank depth, that is filtered as adult. While the spam density for certain queries was often very high (up to 20%), the adult densities were significantly lower, which is either an artifact of the data collection (low adult coverage) or of the queries themselves. Table 6 (bottom) shows the 5 queries with the highest adult density for the two document sets (up to 2000 hits). Somewhat interestingly, category B tends to have larger adult densities than category A, which would indicate that category B may contain a larger fraction of adult pages than category A or that the adult pages are simply more 'retrievable' in category B. While most of the high adult density queries contain terms that may lead to adult results, it was surprising to see the rather innocuous query "the current" make the list.

The difference in document quality may also explain why the independent fusion approach to results merging was more effective than global statistics. If the average quality, spam density, and adult density of each segment of the ClueWeb09 collection were equal, then one would expect the use of global statistics to be more effective. On the other hand, if there is a high variance in quality across the segments, then independent fusion will rank the best documents from each partition highly, some of which will be higher quality (i.e., those returned from the high quality segment) than others. For example, consider an index with just two segments, where segment X is full of spam and segment Y has no spam. In addition, suppose spam documents rank very highly for certain queries. In this case, global statistics may return mostly documents from segment X, while independent fusion will return a mixture of documents from X and Y, and therefore have better retrieval effectiveness.

Finally, we compare the *average quality* of the results retrieved for the two document sets. Figure 6 plots the average quality as a function of rank depth for category A (*bm25*, global statistics) on the left and category B (*bm25*) on the right; higher is better. Note that the vertical axes are on the same scale, so points on the two plots can be meaningfully compared. Trendlines are added to the plots for illustrative purposes to aid in comparison. These plots show results to depth 2000, which, as we described earlier, is the number of baseline results we used for filtering and reranking. These plots show that the results retrieved from category B are consistently higher quality than the results retrieved from category A. The other point to notice is that the category B trendline is relatively flat, indicating almost constant document quality across all depths, while the category A trendline is more quadratic, increasing until around depth 1000 and then decreasing. The shape of the category A curve can be explained, in part, by Figure 5, which shows that spam density is higher early in the ranked list. Since spam plays a role in determining document quality, it is natural for the average quality curve to be inversely related to the spam density curve in this way.

6. CONCLUSIONS

The transition from single-machine to cluster-based architectures in information retrieval research is inevitable, and the availability of the ClueWeb09 collection propels the academic community in the right direction. This development provides an opportunity to reexamine many aspects of information retrieval in a distributed processing environment for web-scale collections. In Ivory, we have explored three such aspects: an HDFS-based retrieval architecture, scalable indexing algorithms with MapReduce, and webpage classification. There is, of course, much more work to be done.

7. 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 first author is grateful to Esther and Kiri for their loving support.

8. REFERENCES

[1] J. Allan, J. A. Aslam, B. Carterette, V. Pavlu, and E. Kanoulas. Million query track 2008 overview. In *Proceedings of the Seventeenth Text REtrieval Conference (TREC 2007)*, Gaithersburg, Maryland, 2008.

[2] L. A. Barroso and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool Publishers, 2009.

[3] B. Carterette, J. Allan, and R. Sitaraman. Minimal test collections for retrieval evaluation. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2006)*, pages 268–275, Seattle, Washington, 2006.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating System Design and Implementation (OSDI 2006)*, pages 205–218, Seattle, Washington, 2006.

[5] 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.

[6] 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.

[7] 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.

[8] T. Leek, R. Schwartz, and S. Sista. Probabilistic approaches to topic detection and tracking. In J. Allan, editor, *Topic Detection and Tracking: Event-based Information Organization*, pages 67–84. Kluwer Academic Publishers, 2002.

[9] 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.

[10] 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.

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

[12] 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.

[13] D. Metzler. *Beyond Bags of Words: Effectively Modeling Dependence and Features in Information Retrieval*. PhD thesis, University of Massachusetts, Amherst, 2007.

[14] D. Metzler and W. B. Croft. A Markov random field model for term dependencies. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2005)*, pages 472–479, Salvador, Brazil, 2005.

[15] X. Qi and B. D. Davison. Web page classification: Features and algorithms. *ACM Computing Surveys*, 41(2), 2009.

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

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