Comparing Distributed Indexing: To MapReduce or Not?

Richard M. C. McCreadie
Department of Computing Science
University of Glasgow
Glasgow, G12 8QQ
richardm@dcs.gla.ac.uk

Craig Macdonald
Department of Computing Science
University of Glasgow
Glasgow, G12 8QQ
craigm@dcs.gla.ac.uk

Iadh Ounis
Department of Computing Science
University of Glasgow
Glasgow, G12 8QQ
ounis@dcs.gla.ac.uk

ABSTRACT

Information Retrieval (IR) systems require input corpora to be indexed. The advent of terabyte-scale Web corpora has reinvigorated the need for efficient indexing. In this work, we investigate distributed indexing paradigms, in particular within the auspices of the MapReduce programming framework. In particular, we describe two indexing approaches based on the original MapReduce paper, and compare these with a standard distributed IR system, the MapReduce indexing strategy used by the Nutch IR platform, and a more advanced MapReduce indexing implementation that we propose. Experiments using the Hadoop MapReduce implementation and a large standard TREC corpus show our proposed MapReduce indexing implementation to be more efficient than those proposed in the original paper.

1. INTRODUCTION

The Web is the largest known document repository, and poses a major challenge for Information Retrieval (IR) systems, such as those used by Web search engines or Web IR researchers. Indeed, while the index sizes of major Web search engines are a closely guarded secret, these are commonly accepted to be in the range of billions of documents. For researchers, the recently released TREC ClueWeb09 corpus[1] of 1.2 billion Web documents poses both indexing and retrieval challenges. In both scenarios, the ability to efficiently create appropriate index structures to allow effective and efficient search is of much value. Moreover, at such scale, the use of distributed architectures to achieve high throughput is essential.

In this work, we investigate the MapReduce programming paradigm, that has been gaining popularity in commercial settings, with implementations by Google [5] and Yahoo! [21]. Microsoft also has a similar framework for distributed operations [10]. In particular, MapReduce allows the horizontal scaling of large-scale workloads using clusters of machines. It applies the intuition that many common large-scale tasks can be expressed as map and reduce operations [5], thereby providing an easily accessible framework for parallelism over multiple machines.

However, while MapReduce has been widely adopted within Google, and is reportedly used for their main indexing process, the MapReduce framework implementation and other programs using it remain (understandably) internal only. Moreover, there have been few empirical studies undertaken into the scalability of MapReduce beyond that contained within the original MapReduce paper [5], which in particular demonstrates the scalability of the simple operations grep and sort. More recently, a MapReduce implementation has been used to sort 1 terabyte of data in approx. 1 minute [17]. However, while Dean & Ghemawat [5] suggest a simple formulation in MapReduce for document indexing, no studies have empirically shown the benefits of applying MapReduce on the important IR indexing problem.

This paper contributes a first step towards understanding the benefits of indexing large corpora using MapReduce, in comparison to other indexing implementations. In particular, we describe four different methods of performing document indexing in MapReduce, from initial suggestions by Dean & Ghemawat, to more advanced strategies. We deploy MapReduce indexing strategies in the Terrier IR platform [18], using the freely available Hadoop implementation [1] of MapReduce, and then perform experiments using standard TREC data.

The remainder of this paper is structured as follows: Section 2 describes a state-of-the art single-pass indexing strategy; Section 3 introduces the MapReduce paradigm; Section 4 describes strategies for document indexing in MapReduce; Section 5 describes our experimental setup, research questions, experiments, and analysis of results; Concluding remarks are provided in Section 6.

2. INDEXING

In the following, we briefly describe the structures involved in the indexing process (Section 2.1) and how the modern single-pass indexing strategy is deployed in the open source Terrier IR platform [18] on which this work is based (Section 2.2). We then provide details of how an indexing process can be distributed to make use of additional machines (Section 2.3).

2.1 Index Structures

To allow efficient retrieval of documents from a corpus, suitable data structures must be created, collectively known as an index. Usually, a corpus covers many documents, and hence the index will be held on a large storage device - commonly one or more hard disks. Typically, at the centre of any IR system is the *inverted index* [23]. For each term, the inverted index contains a *posting list*, which lists the documents - represented as integer document-IDs (doc-IDs) - containing the term. Each posting in the posting list also stores sufficient statistical information to score each docu-

[1] See `http://boston.lti.cs.cmu.edu/Data/clueweb09/`.

Copyright © 2009 for the individual papers by the papers' authors. Copying permitted for private and academic purposes. Re-publication of material from this volume requires permission by the copyright owners. This volume is published by its editors.

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

ment, such as the frequency of the term occurrences and, possibly, positional information (the position of the term within each document, which facilitates phrase or proximity search) [23] or field information (the occurrence of the term in various semi-structured area of the document, such as title, enabling these to be higher-weighted during retrieved). The inverted index does not store the textual terms themselves, but instead uses an additional structure known as a lexicon to store these along with pointers to the corresponding posting lists within the inverted index. A document index may also be created which stores meta-information about each document within the inverted index, such as an external name for the document (e.g. URL), and the length of the document [18]. The process of generating these structures is known as *indexing*.

2.2 Single-pass Indexing

When indexing a corpus of documents, documents are read from their storage location on disk, and then tokenised. Tokens may then be removed (stop-words) or transformed (e.g. stemming), before being collated into the final index structures [23]. Current state-of-the-art indexing uses a single-pass indexing method [8], where the (compressed) posting lists for each term are built in memory as the corpus is scanned. However, it is unlikely that the posting lists for very many documents would fit wholly in the memory of a single machine. Instead, when memory is exhausted, the partial indices are 'flushed' to disk. Once all documents have been scanned, the final index is built by merging the flushed partial indices.

In particular, the temporary posting lists held in memory are of the form list(term, list(doc-ID, Term Frequency)). Additional information such as positions or fields can also be held within each posting. As per modern compression schemes, only the first doc-ID in each posting list is absolute - for the rest, the difference between doc-IDs are instead stored to save space, using Elias-Gamma compression [6].

2.3 Distributing Indexing

The single-pass indexing strategy described above is designed to run on a single machine architecture with finite available memory. However, should we want to take advantage of multiple machines, this can be achieved in an intuitive manner by deploying an instance of this indexing strategy on each machine [22]. For machines with more than one processor, one instance per processing core is possible, assuming the local disk and memory are not saturated. As described by Ribeiro-Neto & Barbosa [20], each instance would index a subset of the input corpus to create an index for only those documents. It should be noted that if the documents to be indexed are local to the machines doing the work (*shared-nothing*), such as when each machine has crawled the documents it is indexing, then this strategy will *always be optimal* (will scale linearly with processing power). However, in practical terms, fully machine-local data is difficult to achieve when a large number of machines is involved. This stems from the need to split and distribute the corpus without overloading the network or risking un-recoverable data loss from a single point of failure.

Distributed indexing has seen some coverage in the literature. Ribeiro-Neto & Barbosa [20] compared three distributed indexing algorithms for indexing 18 million documents. Efficiency was measured with respect to local throughput of each processor, not in terms of overall indexing time. Unfortunately, they do not state the underlying hardware that they employ, and as such their results are difficult to compare to. Melnik et al. [15] described a distributed indexing regime designed for the Web, with considerations for updatable indices. However, their experiments did not consider efficiency as the number of nodes is increased.

In [5], Dean & Ghemawat proposed the MapReduce paradigm for distributing data-intensive processing across multiple machines. Section 3 gives an overview of MapReduce. Section 4 reviews prior work on MapReduce indexing, namely that of Dean & Ghemawat, who suggest how document indexing can be implemented in MapReduce, and from the Nutch IR system. Moreover, we propose a more advanced method of MapReduce indexing, which, by the experiments in Section 5, is shown to be more efficient.

3. MAPREDUCE

MapReduce is a programming paradigm for the processing of large amounts of data by distributing work tasks over multiple processing machines [5]. It was designed at Google as a way to distribute computational tasks which are run over large datasets. It is built on the idea that many tasks which are computationally intensive involve doing a 'map' operation with a simple function over each 'record' in a large dataset, emitting key/value pairs to comprise the results. The map operation itself can be easily distributed by running it on different machines processing different subsets of the input data. The output from each of these is then collected and merged into the desired results by 'reduce' operations.

By using the MapReduce abstraction, the complex details of parallel processing, such as fault tolerance and node availability, are hidden, in a conceptually simple framework [13], allowing highly distributed tools to easily be built on top of MapReduce. Indeed, various companies have developed tools to perform data mining operations on large-scale datasets on top of MapReduce implementations. Google's Sawzall [19] and Yahoo's Pig [16] are two such examples of data mining languages. Microsoft uses a distributed framework similar to MapReduce called Dryad, which the Nebula scripting language uses to provide similar data mining capabilities [10]. However, it is of note that MapReduce trades the ability to perform code optimisation (by abstracting from the internal workings) for easy implementation through its framework, meaning that an implementation in MapReduce is likely not the optimal solution, but will be cheaper to produce and maintain [11].

MapReduce is designed from a functional programming perspective, where functions provide definitions of operations over input data. A single MapReduce job is defined by the user as two functions. The map function takes in a key/value pair (of type <key1, value1>) and produces a set of intermediate key/value pairs (<key2, value2>). The outputs from the map function are then automatically grouped by their key, and then passed to the reduce function. The reduce task merges the values with the same key to form a smaller final result. A typical job will have many map tasks which each operate on a subset of the input data, and fewer reduce tasks, which operate on the merged output of the map tasks. Map or reduce tasks may run on different machines, allowing parallelism to be achieved. In common with functional programming design, each task is independent of other tasks of the same type, and there is no global state, or communication between maps or between reduces.

Counting term occurrences in a large data-set is an often-repeated example of how to use MapReduce paradigm[2] [5]. For this, the map function takes the document file-name (key1) and the contents of the document (value1) as input, then for each term in the document emits the term (key2) and the integer value '1' (value2). The reduce then sums up all of the values (many 1s) for each key2 (a term) to give the total occurrences of that term.

As mentioned above, MapReduce jobs are executed over multiple machines. In a typical setup, data is not stored in a central file store, but instead replicated in blocks (usually of 64MB) across many machines [7]. This has a central advantage that the map functions can operate on data that may be 'rack-local' or 'machine-local' - i.e. does not have to transit intra- and inter-data centre backbone links, and does not overload a central file storage service. Therefore high bandwidth can be achieved because data is always as local as possible to the processing CPUs. Intermediate results of map tasks are stored on the processing machines themselves. To reduce the size of this output (and therefore IO), it may be merged using a combiner, which acts as a reducer local to each machine. A central master machine provides job and task scheduling, which attempts to perform tasks as local as possible to the input data.

While MapReduce is seeing increasing popularity, there are only a few notable studies investigating the paradigm beyond the original paper. In particular, for machine learning [4], Chu et al. studied how various machine learning algorithms could be parallelised using the MapReduce paradigm, however experiments were only carried out on single systems, rather than a cluster of machines. In such a situation, MapReduce provides an easy framework to distribute non-cooperating tasks of work, but misses the central data locality advantage facilitated by a MapReduce framework. A similar study for natural language processing [12] used several machines, but with experimental datasets of only 88MB and 770MB, would again fail to see benefit in the data-local scheduling of tasks.

In contrast, indexing is an IO-intensive operation, where large amounts of raw data have to be read and transformed into suitable index structures. In this work, we show how indexing can be implemented in a MapReduce framework. However, the MapReduce implementation described in [5] is not available outside of Google. Instead, we use the Hadoop [1] framework, which is an open-source Java implementation of MapReduce from the Apache Software Foundation, with developers contributed by Yahoo! and Facebook, among others. In the next section, we describe several indexing strategies in MapReduce, starting from that proposed in the original MapReduce paper [5], before developing a more refined strategy inspired by the single-pass indexing described in Section 2.2.

4. INDEXING IN MAPREDUCE

In this section, we show how indexing can be performed in MapReduce. Firstly, we describe two possible interpretations of indexing as envisaged by Dean & Ghemawat in their original seminal MapReduce paper [5] (Section 4.1). Then, we describe an alternative MapReduce indexing strategy used by the Nutch IR platform, before finally showing how a more refined single-pass indexing strategy can be implemented in MapReduce (Section 4.3).

It should be noted that in MapReduce each map task is not aware of its context in the overall job. For indexing, this means that the doc-IDs emitted from the map phases cannot be globally correct. Instead, these doc-IDs start from 0 in each map. To allow the reduce tasks to calculate the correct doc-IDs, each map task produces a "side-effect" file, detailing the number of documents emitted per map. This is true for all the indexing implementations described in this section. We also note that for all our indexing implementations the number of reducers specified depicts the number of final indices generated.

4.1 Dean & Ghemawat's MapReduce Indexing Strategy

The original MapReduce paper by Dean & Ghemawat [5] presents a short description for performing indexing in MapReduce, which is directly quoted below:

"The map function parses each document, and emits a sequence of <word, document ID> pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a <word, list(document ID)> pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions."

The implicit claim being made in the original MapReduce paper [5] is that efficient indexing could be trivially implemented in MapReduce. However, we argue that this over-simplifies the details, and provides room for a useful study to allow document indexing in MapReduce to be better understood. For example, for an inverted index to be useful, the term frequencies within each document need to be stored. Though this is not accounted for in Dean & Ghemawat's paper, there are two possible interpretations on how this could be achieved within the bounds laid out in the quotation above. We detail these interpretations below in Sections 4.1.1 and 4.1.2, respectively.

4.1.1 Emitting Term,Doc-ID Tuples

The literal interpretation of the description above would be to output a set of <term, doc-ID> pairs for each token in a document. This means that if a single term appears n times in a document then the <term, doc-ID> pair will be emitted n times. This has the advantage of making the map phase incredibly simple, as it emits on a per token basis. However, this means that we will emit a <term, doc-ID> pair for every token in the collection. In general, when a map task emits lots of intermediate data, this will be saved to the machine's local disk, and then later transferred to the appropriate reducer. However, with this indexing interpretation, the intermediate map data would be extremely large - indeed, similar to the size of the corpus, as each token in the corpus is emitted along with a doc-ID. Having large amounts of intermediate map data will increase map to reducer network traffic, as well as lengthening the sort phase. These are likely to have an effect on the job's overall execution time. The reducer will - for each unique term - sort the doc-IDs, then add up the instances on a per doc-ID basis to retrieve the term frequencies. Finally, the reducer will write the completed posting list for that term to disk. Figure 1 provides a pseudo-code implementation of map and reduce functions for this strategy.

[2] A worked example and associated source code is available at `http://hadoop.apache.org/core/docs/r0.19.0/mapred_tutorial.html`

Dean & Ghemawat MapReduce Indexing - Map function pseudo-code

```
1: Input
     Key: Document Identifier, Name
     Value: Contents of the Document, DocContents
2: Output
     A list of (term,doc-ID) pairs, one for each token
     in the document
3: for each Term in the DocContents loop
4 :   Stem(Term)
5 :   deleteIfStopword(Term)
6 :   if (Term is not empty) then emit(Term, doc-ID)
7: end loop
8: Add document to the Document Index
9: if (lastMap()) write out information about the
10:   documents this map processed ("side-effect" files)
```

Dean & Ghemawat MapReduce Indexing - Reduce function pseudo-code

```
1: Input
     Key: A Term
     Value: List of (doc-ID), doc-IDs
2: Output
     Key: Term
     Value: Posting List
3 : List Posting-List = new PostingList()
4 : Sort doc-IDs
5 : for each doc-ID in doc-IDs loop
6 :   increment tf for doc-ID
7 :   correct doc-ID
8 :   add doc-ID and tf to Posting-List
9 : end loop
10: emit(Posting-List)
```

Figure 1: Pseudo-code interpretation of Dean & Ghemawat's MapReduce indexing strategy (map emitting <term,doc-ID>, Section 4.1.1).

4.1.2 Emitting Term,Doc-ID,TF Tuples

We claim that emitting once for every token extracted is wasteful of resources, causing excessive disk IO on the map by writing intermediate map output to disk, and excessive disk IO in moving map output to the reduce tasks. To reduce IO, we could instead emit <term,(doc-ID, tf)> tuples, where tf is the term frequency for the current document. In this way, the number of emit operations which have to be done is significantly reduced, as we now only emit once per unique term per document. The reduce method for this interpretation is also much simpler than the earlier interpretation, as it only has to sort instances by document to get the final posting list to write out. It should also be noted that the <term, doc-ID> strategy described earlier, can be adapted to generate tfs instead through the use of a MapReduce combiner, which performs a localised merge on each map task's output.

While the <term,(doc-ID, tf)> indexing strategy emits significantly less than that described in Section 4.1.1, we argue that an implementation in this manner would still be inefficient, because a large amount of IO is still required to store, move and sort the temporary map output data.

4.2 Nutch's MapReduce Indexing Strategy

The Apache Software Foundation's open source Nutch platform [3] also deploys a MapReduce indexing strategy, using the Hadoop MapReduce implementation. By inspection of the source of Nutch v0.9, we have determined that the MapReduce indexing strategy differs from the general outline described in Section 4.1 above. Instead of emitting terms, Nutch only tokenises the document during the map phase, hence emitting <doc-ID, analysed-Document> tuples from the map function. Each analysed-Document contains the textual forms of each term and their corresponding frequencies. The reduce phase is then responsible for writing all index structures. Compared to emitting <term,(doc-ID, tf)>, the Nutch indexing method will emit less, but the value of each emit will contain substantially more data (i.e. the textual form and frequency of each unique term in the document). We believe this is a step-forward towards reducing intermediate map output. However, there may still be scope for further reducing map task output to the benefit of overall indexing efficiency. In the next section, we develop our single-pass indexing strategy (described in Section 2.2) for the MapReduce framework, to address this issue.

4.3 Single-pass MapReduce Indexing Strategy

We now adapt the single-pass indexing strategy described in Section 2.2, for use in a MapReduce framework. The indexing process is split into m map tasks. Each map task operates on its own subset of the data, and is similar to the single-pass indexing corpus scanning phase. However, when memory runs low or all documents for that map have been processed, the partial index is flushed from the map task, by emitting a set of <term, posting list> pairs. The partial indices (flushes) are then sorted by term, map and flush numbers before being passed to a reduce task. As the flushes are collected at an appropriate reduce task, the posting lists for each term are merged by map number and flush number, to ensure that the posting lists for each term are in a globally correct ordering. The reduce function takes each term in turn and merges the posting lists for that term into the full posting list, as a standard index. Elias-Gamma compression is used as in non-distributed indexing to store only the distance between doc-IDs. Figure 2 provides a pseudo-code implementation of map and reduce functions for our proposed MapReduce indexing strategy.

The fundamental difference between this strategy and that of Dean & Ghemawat described in Section 4.1, is what the map tasks emit. Instead of emitting a batch of <term,doc-ID> pairs immediately upon parsing each document, we instead build up a posting list for each term in memory. Over many documents, memory will eventually be exhausted, at which time all currently stored posting lists will be flushed as <term,posting list> tuples. This has the positive effect of minimising both the size of the map task output, as well as the number of emits. Compared to the Dean & Ghemawat indexing strategies, far less emits will be called, but emits will be much larger. Compared to the Nutch MapReduce indexing strategy, there may more emits, however, the reduce task is operating on term-sorted data, and does not require a further sort and invert operation to generate an inverted index. Moreover, the emit values will only contain doc-IDs instead of textual terms, making them considerably smaller.

Figure 3 presents an example for a distributed setting MapReduce indexing paradigm of 200 documents. The documents are indexed by $m = 2$ map tasks, before the posting lists for each term are grouped and sorted, and then reduced to a single index. The posting lists output from each map contains only local doc-IDs. In the reduce tasks, these are merged into a list of absolute doc-IDs, by adding to each

Single-Pass MapReduce Indexing - Map function pseudo-code

```
1: Input
      Key: Document Identifier, Name
      Value: Contents of the Document, DocContents
2: Output
      Key: Term
      Value: Posting list
3: for each Term in the DocContents loop
4 :   Stem(Term)
5 :   deleteIfStopword(Term)
6 :   if (Term is not empty) then add the current
      document for that term to the in-memory
      Posting List
7: end loop
8: Add document to the Document Index
9: if (lastMap() or outOfMemory()) then
      emit(in-Memory Posting List)
10: if (lastMap()) write out information about the
11:     documents this map processed ("side-effect" files)
```

Single-Pass MapReduce Indexing - Reduce function pseudo-code

```
1: Input
      Key: A Term
      Value: List of (Posting List), PartialPostingLists
2: Output
      Key: Term
      Value: Posting List
3 : List Posting-List = new PostingList()
4 : Sort PartialPostingLists by the map and flush they
      were emitted from
5 : for each PostList in PartialPostingLists loop
6 :   for each doc-ID in PostList loop
7 :       correct doc-ID
8 :       Merge PostList into Posting-List
9 :   end loop
10: end loop
11: emit(Posting-List)
```

Figure 2: Pseudo-code for our proposed single-pass MapReduce indexing strategy (Section 4.3).

entry the number of documents processed by previous map tasks. However, note that in our indexing implementation, the doc-IDs are flush-local as well as map-local. While this is not strictly necessary, it allows smaller doc-IDs to be emitted from each map, which can be better compressed.

5. EXPERIMENTS & RESULTS

In the following experiments, we aim to determine the efficiency of multiple indexing implementations. Specifically, we investigate whether distributed indexing as laid out in the original MapReduce paper (Section 4.1) is fit for purpose. We compare this to our single-pass indexing strategy developed both for a single machine architecture (Section 2) and for MapReduce (Section 4.3). Note that in this paper we do not investigate Nutch's MapReduce indexing strategy, however we expect it to be more efficient than Dean & Ghemawat's indexing strategy, while being less efficient than our single-pass indexing strategy. We leave this for future work. Furthermore, we investigate these approaches in terms of scalability as the number of machines designated for work is increased, and experiment with various parameters

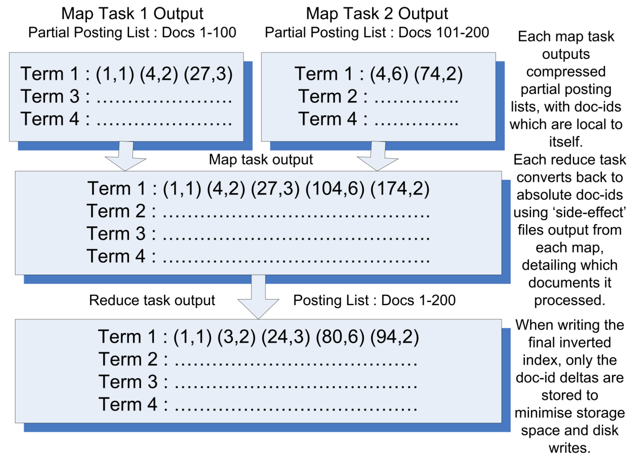

Figure 3: Correcting document IDs while merging.

in MapReduce to determine how to most efficiently apply it for indexing.

5.1 Research Questions

To measure the efficiency of our indexing implementations and therefore the suitability (or otherwise) of MapReduce for indexing, we investigate 3 important research questions, which we address by experimentation in the remainder of this section:
1. Can a practical application of the distributed indexing strategy described in Section 2 be sufficient for large-scale collections when using many machines? (Section 5.4)
2. When indexing with MapReduce, what is the most efficient number of maps and reduces to use? (Section 5.5)
3. Is MapReduce Performance Close to Optimal Distributed Indexing? (Section 5.6)

5.2 Evaluation Metrics

Research questions 1-3 require a metric for indexing performance. For this, we measure the throughput of the system, in terms of MB/s (megabytes per second). We calculate throughput as $collectionsize/timetaken$ where collection size is the compressed size on disk for a single copy of the collection in MB (megabytes). The time taken is the full time taken by the job (including setup) measured in seconds.

Research question 3 mandates suitability for indexing at a large scale. We measure suitability in terms of throughput (as above) and in terms of speedup. Speedup S_m, defined as $S_m = \frac{T_1}{T_m}$, where m is the number of machines, T_1 is the execution of the algorithm on a single machine, and T_m is the execution time in parallel, using m machines [9]. This encompasses the idea that not only should speed improve as more resources are added, but that such a speed increase should reflect the quantity of those resources. For instance, if we increase the available resources by a factor of 2, then it would be desirable to get (close to) twice the speed. This is known as linear speedup (where $S_m = m$), and is the ideal scenario for parallel processing. However, linear speedup can be hard to achieve in a parallel environment, because of the growing influence of small sequential sections of code as the number of processors increases (known as Amdahl's law [2]), or due to overheads.

5.3 Experimental Setup

Following [24], which prescribes guidelines for presenting indexing techniques, we now give details of our experimen-

Number of Machines (Cores)	1(3)	2(6)	4(12)	6(18)	8(24)
Distributed Single-Pass	2.44	4.6	12.8	12.4	12.8
Dean & Ghemawat MapReduce	1.15	1.59	4.01	4.71	6.38
MapReduce Single-Pass	2.59	5.19	9.45	13.16	17.31

Table 1: Throughput as the number of machines allocated is increased using using a variety of indexing strategies, measured in MB/sec.

tal cluster setup, consisting of 19 identical machines. Each machine has a single Intel Xeon 2.4GHz processor with 4 cores, 4GB of RAM, and contains three hard drives: One 160GB hard disk, spinning at 7200rpm with an 8MB buffer, is used for the operating system and temporary job scratch space; Two 400GB hard disks, each spinning at 7200rpm with a 16MB buffer, are dedicated for distributed file system storage. Each machine is running a copy of the open source Linux operating system Centos 5 and are connected together by a gigabit Ethernet connection on a single rack. The Hadoop (version 0.18.2) distributed file system (DFS) is running on this cluster, replicating files to the distributed file storage on each machine. Each file on the DFS is split into 64MB blocks, which are each replicated to 2 machines[3]. While each machine has four processors available at any one time, only three of these are valid targets for job execution, the last processor is left free for the distributed file system software running on each machine. As our cluster is shared by several users, job allocation is done by Hadoop on Demand (HOD) running with the Torque resource manager (version 2.1.9) rather than using a dedicated Hadoop cluster. Machines not allocated to a MapReduce job are available to be scheduled by Torque for other jobs not associated with MapReduce. However on such nodes, the fourth processor core is still free for distributed file system work[4]. We also have in the same rack a RAID5 centralised file server powered by 8 Intel Xeon 3GHz processor cores for use with non-MapReduce jobs, providing network file system (NFS) storage. For consistency, in the following experiments, we employ the standard TREC web collection .GOV2. This is an 80GB (425GB uncompressed) crawl of *.gov* Web domain comprising over 25 million documents. Before the advent of ClueWeb09, .GOV2 was the largest available TREC corpus.

5.4 Is Distributed Indexing Good Enough?

First we determine if MapReduce is necessary for large-scale indexing. If a simple distribution of the non-parallel indexing strategy described in Section 2 is sufficient to index large collections then there is no need for MapReduce. To evaluate this, we distribute the single-pass indexing strategy across n machines in our cluster, where we vary $n = \{1, 2, 4, 6, 8\}$. To provide a comparative baseline, the non-parallel single-pass indexing implementation in Terrier can index the .GOV2 corpus on a single processor core (*not* machine) in just over 1 day using the same algorithm. This translates into a throughput of approximately 1MB/sec. For distributed indexing to be sufficient for indexing large collections, throughput should increase in a (close-to) linear fashion with the number of processing cores added. As

[3]This is lower than the Hadoop default of 3, to conserve distributed file system space.

[4]Hence, as each machine always has one processing core free to handle distributed file system traffic, and the network traffic of other cluster jobs is assumed to be low, then there should be no impact on the validity of the experiments.

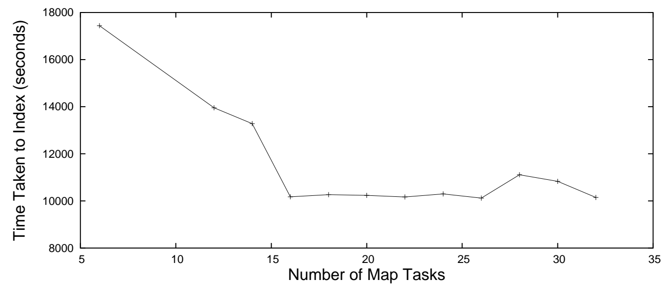

Figure 4: The effect of varying the number of map tasks on indexing time (seconds) of .GOV2 collection: 4 machines, 1 reduce task.

mentioned in Section 2.3, when distributed indexing uses machine-local data, indexing will achieve exactly linear scaling. However, unless the document data is already present on the machines (e.g. indexing takes place on the machines which crawled the documents), there would be the need to copy the required data to the indexing machines. In many other scenarios, crawling or documents corpora storage may not be on indexing machines. Moreover, local-only indexing is not resilient to machine failure. Instead, we experiment with the shared-corpus distributed indexing, where the corpus is indexed over NFS from a central fileserver. Local data (shared-nothing) indexing would require the corpus subset to be copied prior to indexing.

Table 1, row 1, shows how throughput increases as we allocate more machines (recall that each machine adds three processor cores for indexing work). Here we can see that throughput indeed increases in a reasonable fashion, However, once we allocate more than 4 machines we observe no further speed improvements. This is caused by our central file store becoming a bottleneck as it is unable to serve all the allocated machines simultaneously. We can therefore conclude that this distribution method is unsuitable for large-scale indexing using our hardware setup. Moreover, we argue that even with better hardware this issue cannot be overcome as the file server(s) will always be slower than the combination of all worker machines.

5.5 Investigating MapReduce Parameters

In Section 5.4, we showed that the distributed indexing strategy described in Section 2 is unsuitable for the scalable distributed shared-corpus indexing of large collections. However, before we can evaluate MapReduce as an alternate solution we need to investigate how to maximise its efficiency in terms of its input parameters. The fundamental parameters of a MapReduce job are m - the number of map tasks that the input data is divided across - and r, the number of reduce tasks. A higher number of map tasks means that the input collection of documents is split into smaller chunks, but also that there will be more overheads, as more tasks have to be initialised and latterly cleared. To determine what effect this has on performance, we vary m while indexing the .GOV2 corpus, using a set 4 machines. The results - in terms of indexing time - are shown in Figure 4. We see that when the number of maps is small (i.e. less than the 12 processors available from the 4 machines), parallelism is hindered, as not all processors have work to do. When the number of map tasks is ≤ 14, we also note that indexing time is still high. On examination of these jobs, we found

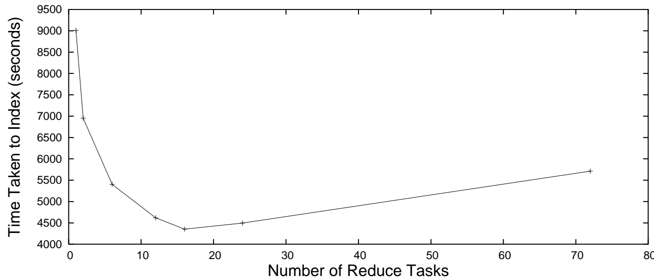

Figure 5: The effect of varying the number of reduce tasks on indexing time (seconds) of .GOV2 collection: 6 machines, 72 map tasks.

that the balance of work between map tasks was not even, with one map task taking markedly longer than the others[5]. When the number of map tasks is increased to 16, balance is restored.

In previous work [14], we have shown that the time taken by the reduce step is an important factor in determining indexing performance. Therefore, it is important to know how many reduce tasks it is is optimal to create - subject to external constraints on the number of reducers (*e.g.* having 8 query servers suggests 8 reducers are used so that 8 final indices are created). To test the effect of the number of reduce tasks on efficiency, we index .GOV2 while varying the number of reduce tasks. Here we used 6 machines and 72 map tasks. The indexing time results are shown in Figure 5. As we would expect, while the number of reduces is below the available processors (for the 6 machines allocated, 18 processors) the speed increases as we add more reducers, since we are effectively providing more parallel processing power. Once we are beyond the number of processors however, indexing time increases. This is intuitive, as there is more work to be done than available processors. Therefore, we can conclude that the number of reduce tasks should be a multiple of the number of processors. Unlike map tasks, however, there is an incentive to have less reduce tasks, resulting in fewer indices, but this needs to be traded off against the possibility of failures and the associated time wasted through re-running.

5.6 Is MapReduce Performance Close to Optimal Distributed Indexing?

We now investigate whether MapReduce is an efficient alternative to distributed indexing. Moreover, we evaluate MapReduce against optimal distributed indexing in terms of performance, i.e. the extent to which it scales close to linearly with processing power. The core advantage of MapReduce is the ability to apply the distributed file system (DFS) to avoid centralised storage of data (creating a single point of failure), and to take advantage of data locality to avoid excess network IO. This meanwhile, is at the cost of additional overheads in job setup, monitoring and control, as well as the additional IO required to replicate the data on a DFS. As the centralised file-system was identified as the bottleneck for distributed indexing, we would expect MapReduce to perform better since it uses a DFS. For evaluation, we perform a direct comparison on throughput between indexing strategies. Note that while distributed indexing creates n index shards, where n is the number of processors allocated, MapReduce instead produces r index shards where r is the number of reduce tasks created. For these experiments we always allocate 72 map tasks and 24 reduce tasks. This means that for distributed indexing a smaller number of index shards were created when indexing on $\{1, 2, 4, 6\}$ machines. However, we believe that this has no significant impact on the overall throughput.

First, we investigate whether the MapReduce indexing strategy proposed by Dean & Ghemawat is more efficient than distributed indexing. Table 1 shows how the throughput increases as we allocate more machines - in particular, row 2 shows results for Dean & Ghemawat's strategy, interpreted as emitting term <doc-ID,tf> tuples (Section 4.1.2). We also implemented the other interpretation which emits term,doc-ID tuples, however, it consumed excessive temporary storage space during operation due to its large number of emit operations. This made it impossible to determine throughput, as the worker machines ran out of disk space causing the job to fail. Our implementation of Dean & Ghemawat's indexing method also creates the additional data structures described in Section 2.1 - i.e. the lexicon and document index - and uses the compressed Terrier inverted index format. From Table 1, row 2, we can see that this implementation performs very poorly in comparison to distributed indexing. Indeed, with 8 machines it indexes only at half the speed of distributed indexing with the same number of machines. Upon further investigation, as expected, this speed degradation can be attributed to the large volume of map output which is generated by this approach. However, it should be noted that unlike distributed indexing, performance improvements do not stall after 4 machines. This would indicate that while the indexing strategy is poor, MapReduce in general will continue to garner performance improvements as more machines are added. Therefore, we believe this makes it more suitable for processing larger corpora, where larger clusters of 100s-1000s of machines are needed to index them in reasonable amounts of time.

We now experiment with our proposed implementation of single-pass indexing in MapReduce, as described in Section 4.3. Our expectation is that this strategy should prove to be more efficient as it lowers disk and network IO by building up posting lists in memory, thereby minimising map output size. Table 1, row 3 shows the throughput of the single-pass MapReduce indexing strategy. In comparison to Dean & Ghemawat's indexing strategy, we find our approach to be markedly faster. Indeed, when using 8 machines our method is over 2.7 times faster. Moreover, Figure 6 shows the speedup achieved by both approaches as the number of machines is increased. We observe that our single-pass based strategy scales close to linearly in terms of indexing time as the number of machines allocated for work is increased. In contrast, the scalability of Dean & Ghemawat's approach is noticeably worse (5.5 times for 8 processors, versus 6.8 times for single-pass based indexing). We believe that this makes our proposed strategy suitable for scaling to large clusters of machines, which is essential when indexing new large-scale collections like ClueWeb09.

[5]Hadoop actually supports *speculative execution*, where two copies of the last task, or the slowest tasks, will be started. Only output from the first successful task to complete will be used. This uses otherwise idle processing power to decrease average job duration.

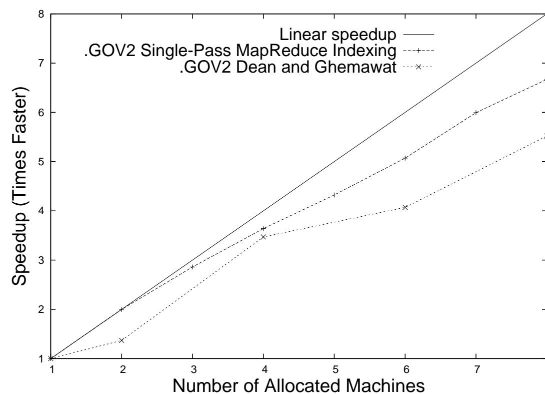

Figure 6: Speedup of .GOV2 indexing as more MapReduce machines are allocated.

6. CONCLUSION

In this paper, we detailed four different strategies for applying document indexing within the MapReduce paradigm, with varying efficiency. In particular, we firstly showed that indexing speed using a distributed indexing strategy was limited by accessing a centralised file-store, and hence the advantage of using MapReduce to allocate indexing tasks close to input data is clear. Secondly, we showed that the MapReduce indexing strategy suggested by Dean & Ghemawat in the original MapReduce paper [5] generates too much intermediate map data, causing an overall slowness of indexing. In contrast, our proposed single-pass indexing strategy is almost 3 times faster, and scales well as the number of machines allocated is increased.

Overall, we conclude that the single-pass based MapReduce indexing algorithm should be suitable for efficiently indexing larger corpora, including the recently released TREC ClueWeb09 corpus. Moreover, as a framework for distributed indexing, MapReduce conveniently provides both data locality and resilience. Finally, it is of note that an implementation of the MapReduce single-pass indexing strategy described in this paper is freely available for use by the community as part of the Terrier IR Platform[6].

7. REFERENCES

[1] Apache Software Foundation. The apache hadoop project. `http://hadoop.apache.org/`, as of 15/06/2009.

[2] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *Proc. of AFIPS*, pp. 483–485, 1967.

[3] M. Cafarella and D. Cutting. Building nutch: Open source search. *ACM Queue*, 2(2):54–61, 2004.

[4] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *Proc. of NIPS 2006*, pp. 281–288.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. of OSDI 2004*, pp. 137–150.

[6] P. Elias. Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on*, 21(2):194–203, 1975.

[6] `http://terrier.org`

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[8] S. Heinz and J. Zobel. Efficient single-pass index construction for text databases. *JASIST*, 54(8):713–729, 2003.

[9] M. D. Hill. What is scalability? *SIGARCH Comput. Archit. News*, 18(4):18–21, 1990.

[10] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. of EuroSys 2007*, pp. 59–72.

[11] R. E. Johnson. Frameworks = (components + patterns). *Commun. ACM*, 40(10):39–42, 1997.

[12] M. Laclavik, M. Seleng, and L. Hluchý. Towards large scale semantic annotation built on mapreduce architecture. In *Proc. of ICCS (3)*, pp. 331–338, 2008.

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

[14] R. McCreadie, C. MacDonald, and I. Ounis. On single-pass indexing with mapreduce. In *Proc. of SIGIR 2009*, in press.

[15] S. Melnik, S. Raghavan, B. Yang, and H. Garcia-Molina. Building a distributed full-text index for the web. In *Proc. of WWW 2001*, pp. 396–406.

[16] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proc. of SIGMOD 2008*, pp. 1099–1110.

[17] O. O'Malley and A. C. Murthy. Winning a 60 second dash with a yellow elephant. TR, Yahoo! Inc., 2009.

[18] I. Ounis, G. Amati, V. Plachouras, B. He, C. Macdonald, and C. Lioma. Terrier: A high performance and scalable information retrieval platform. In *Proc. of OSIR workshop, SIGIR-2006*, pp. 18–25.

[19] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[20] B. A. Ribeiro-Neto, E. S. de Moura, M. S. Neubert, and N. Ziviani. Efficient distributed algorithms to build inverted files. In *Proc. of SIGIR 1999*, pp. 105–112.

[21] E. Schonfeld. Yahoo! search wants to be more like google, embraces hadoop, 2008. `http://www.techcrunch.com/2008/02/20/yahoo-search-wants-to-be-more-like-google-embraces-hadoop/`, as of 15/06/2009.

[22] A. Tomasic and H. Garcia-Molina. Performance of inverted indices in shared-nothing distributed text document information retrieval systems. In *Proc. of PDIS 1993*, pp. 8–17.

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

[24] J. Zobel, A. Moffat, and K. Ramamohanarao. Guidelines for presentation and comparison of indexing techniques. *SIGMOD Record*, 25(3):10–15, 1996.