Contents lists available at ScienceDirect

Information Processing and Management

journal homepage: www.elsevier.com/locate/infoproman

MapReduce indexing strategies: Studying scalability and efficiency

Richard McCreadie *, Craig Macdonald, Iadh Ounis

Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, United Kingdom

ARTICLE INFO

Article history:
Received 11 February 2010
Received in revised form 17 August 2010
Accepted 19 December 2010
Available online xxxx

Keywords:
MapReduce
Indexing
Efficiency
Scalability
Hadoop

ABSTRACT

In Information Retrieval (IR), the efficient indexing of terabyte-scale and larger corpora is still a difficult problem. MapReduce has been proposed as a framework for distributing data-intensive operations across multiple processing machines. In this work, we provide a detailed analysis of four MapReduce indexing strategies of varying complexity. Moreover, we evaluate these indexing strategies by implementing them in an existing IR framework, and performing experiments using the Hadoop MapReduce implementation, in combination with several large standard TREC test corpora. In particular, we examine the efficiency of the indexing strategies, and for the most efficient strategy, we examine how it scales with respect to corpus size, and processing power. Our results attest to both the importance of minimising data transfer between machines for IO intensive tasks like indexing, and the suitability of the per-posting list MapReduce indexing strategy, in particular for indexing at a terabyte-scale. Hence, we conclude that MapReduce is a suitable framework for the deployment of large-scale indexing.

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. However, Dean and Ghemawat (2010) have admitted to holding more than “a dozen distinct datasets at Google [of] more than 1PB in size”. For researchers, the terabyte-scale 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 facilitate effective and efficient search is an important problem. Moreover, at such scale, highly distributed architectures are needed to achieve sufficient indexing throughput.

In this work, we investigate distributed indexing with regard to the MapReduce programming paradigm, that has been gaining popularity in commercial settings, with implementations by Google (Dean & Ghemawat, 2004) and Yahoo! (Schonfeld, 2008). Microsoft has also developed a framework which can be configured to provide similar functionality (Isard, Budiu, Yu, Birrell, & Fetterly, 2007). Importantly, 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 (Dean & Ghemawat, 2004), 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 original MapReduce framework implementation and other programs using it remain (understandably) internal only. Moreover, there have been few systematic studies undertaken into the scalability of MapReduce beyond that contained

* Corresponding author.
E-mail addresses: richardm@dcs.gla.ac.uk (R. McCreadie), craig.macdonald@glasgow.ac.uk (C. Macdonald), iadh.ounis@glasgow.ac.uk (I. Ounis).

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

0306-4573/$ - see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.ipm.2010.12.003

within the original MapReduce paper (Dean & Ghemawat, 2004), which in particular demonstrates the scalability for only the simple operations grep and sort. Indeed, with ever-increasing Web corpora needing to be indexed by information retrieval systems, and the associated uptake of distributed processing frameworks like MapReduce, the case for a detailed study into its viability for such a critical IR task is clear.

This paper represents an important step towards understanding the benefits of indexing large corpora using MapReduce. In particular, we describe four different methods of performing document indexing in MapReduce, from initial suggestions by Dean and Ghemawat, to more advanced strategies employed by the Nutch and Terrier IR systems. We implement these MapReduce indexing strategies within the Terrier IR platform (Ounis et al., 2006), using the freely available Hadoop (Apache Software Foundation, 2010) MapReduce implementation. Moreover, we perform experiments over multiple standard TREC test corpora to determine the most efficient indexing strategy examined. We then leverage the most efficient of these to evaluate indexing scalability in MapReduce, both in terms of corpus size and horizontal hardware scaling.

The remainder of this paper is structured as follows: Section 2 describes the state-of-the-art in document indexing; Section 3 introduces the MapReduce paradigm; Section 4 describes our four 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 (Ounis et al., 2006) 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 multiple 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* (Witten, Moffat, & Bell, 1999). 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 document, 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) (Witten et al., 1999) or field information (the occurrence of the term in various semi-structured areas 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 or meta index may also be created to store information about the indexed documents, such as an external name for the document (e.g. URL), or the length of the document (Ounis et al., 2006), so that they can be effectively ranked and presented to the user. The process of generating all of these data structures is known as *indexing*, and typically happens offline, before being used to determine search results.

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 (e.g. low information bearing stop-words), or transformed in such a way that retrieval is benefited, (e.g. stemming) before being collated into the final index structures (Witten et al., 1999). Current state-of-the-art indexing uses a single-pass indexing method (Heinz & Zobel, 2003), 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 some form of compression, e.g. Elias-Gamma compression (Elias, 1975). During a flush, these compressed posting lists are written to disk verbatim (bit-for-bit correct), while during merging, the flush-local doc-IDs are aligned such that these are correct across the entire index.

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 (Tomasic & Garcia-Molina, 1993). For machines with more than one processor, one instance per processing core is possible, assuming that local disk and memory are not saturated. As described by Ribeiro-Neto and Barbosa (Ribeiro-Neto, de Moura, Neubert, & Ziviani, 1999), 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 (which we refer to as a *shared-nothing* approach), 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 and Barbosa (Ribeiro-Neto et al., 1999) 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, Raghavan, Yang, and Garcia-Molina (2001) described a distributed indexing regime designed for the Web, with considerations for updatable indices. However, their experiments did not consider scalability of the indexing approach as the number of nodes is increased.

In (Dean & Ghemawat, 2004), Dean and 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 and Ghemawat, who suggest how document indexing can be implemented in MapReduce. Furthermore, we also describe the more complex MapReduce document indexing strategies implemented by the Nutch and Terrier IR systems.

3. MapReduce

MapReduce is a programming paradigm for the processing of large amounts of data by distributing work tasks over multiple processing machines (Dean & Ghemawat, 2004). It was designed at Google as a way to distribute computational tasks which are run over large datasets. The core idea is that many types of tasks that 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, each processing their own subset 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 (Manning, Raghavan, & Schütze, 2008), allowing highly distributed tools to be easily built on top of MapReduce. Indeed, various companies have developed tools to perform operations on large-scale datasets on top of MapReduce implementations. Google's Sawzall (Pike, Dorward, Griesemer, & Quinlan, 2005) and Yahoo's Pig (Olston, Reed, Srivastava, Kumar, & Tomkins, 2008) are examples of data flow languages which are built upon MapReduce. Microsoft uses a distributed framework called Dryad which can provide similar functionality (Yu et al., 2008). Indeed, the Nebula scripting language uses the Dryad framework to provide data mining capabilities (Isard et al., 2007). 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 unlikely the optimal solution, but will be cheaper to produce and maintain (Johnson, 1997).

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 sorted by their key (key2). The reduce function collects the map output and then merges the values with the same key to form a smaller final result.

A single MapReduce job has many map *tasks* that each operate on a subset of the input data, using the map function defined for that job. The MapReduce framework queues tasks such that each processor receives only a single task at any one time. When all map tasks have finished, a smaller number of reduce tasks then 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. Fig. 1 shows the main stages of a MapReduce job. In particular, in the figure, the output of three map tasks is sorted, then processed by two reduce tasks.

A *partitioner* decides which reduce task processes each <key2, value2> pair. Usually, the partitioner splits map output between reduce tasks as a function of the value of key2. For instance, in Fig. 1, the partitioner is responsible for deciding which of the two reduce tasks should process each output pair from a map task. Moreover, if the ordering of values in each reduce task is important, then the output can be partitioned using both key2 and value2. Notably, the map output collected by each reduce task is subjected to a secondary sort as it is being merged together, hence allowing both map tasks and intermediate sorting to be performed in parallel.

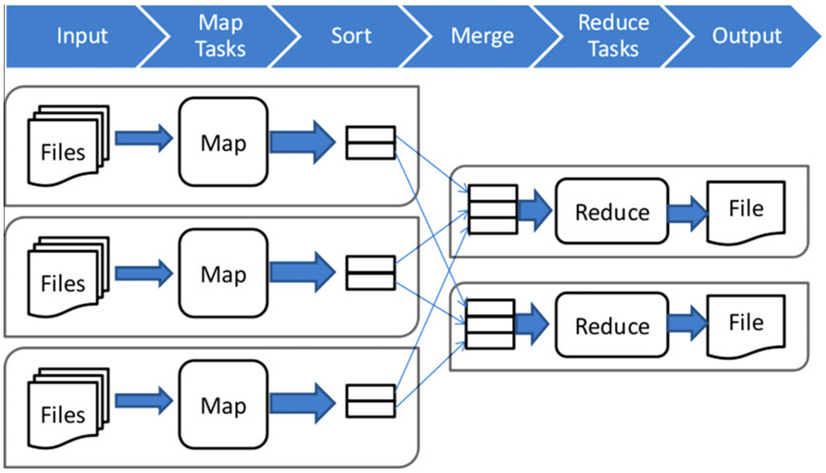

Fig. 1. A graphical illustration of the phases of a typical MapReduce job.

Word Counting - Map function pseudo-code

```
1: Input
      Document File-name, Name
      Contents of the Document, Contents
2: Output
      A list of (Word,1) pairs, one for each
      word in the document, Instances
3: for each Word in the Document loop
4:    emit(Word, 1)
5: end loop
```

Word Counting - Reduce function pseudo-code

```
1: Input
      A Word
      List of 1s, Instances
2: Output
      The Word and the size of
      The Instances, Result
3: Integer result = 0
4: for each int 'i' in Instances loop
5:    result = result + i
6: end loop
7: emit(Word, Result)
```

Fig. 2. Pseudo-code for the word-counting operation, expressed as map and reduce functions.

Counting term occurrences in a large data-set is an often-repeated example of how to use the MapReduce paradigm[2] (Dean & Ghemawat, 2004). 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 function then sums up all of the values (many 1s) for each key2 (a term) to give the total occurrences of that term. Fig. 2 provides the pseudo-code for the word counting map and reduce functions. Note that this is a very simple example for illustrative purposes.

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 64 MB) across many machines (Ghemawat, Gobioff, & Leung, 2003). 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 hence does not overload a central file storage service. Hence, good throughput can be achieved because data is always as local as possible to the processors doing the work. Intermediate results of map tasks are stored on the processing machines themselves. To reduce the size of this output (and therefore the network traffic), it may be merged using a *combiner*, which acts as a reduce task 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 (Dean & Ghemawat, 2004). In particular, for machine learning (Chu et al., 2006), 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 (Laclavik, Seleng, & Hluchý, 2008) used several machines, but with small experimental datasets of only 88 MB and 770 MB, it would again fail to see benefit in the data-local scheduling of tasks. More recently, Pavlo et al. (2009) performed an analysis of MapReduce in comparison to parallel databases, which

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

indicated that MapReduce was inefficient in comparison for data analysis. However, Dean and Ghemawat later responded (Dean & Ghemawat, 2010), highlighting faulty assumptions about MapReduce and poor implementations as the reason for the lacklustre performance. Importantly, they point out that the input-output data format needs to be as efficient as possible (Google Inc., 2010), to avoid needless data transfer between map and reduce tasks, and that often the final reduce task outputs may not need to be merged upon job completion, as further jobs can be modified to access sharded data structures.

In an IR system, indexing is a disk-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 (Dean & Ghemawat, 2004) is not available outside of Google. Instead, we use the Hadoop (Apache Software Foundation, 2010) 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 (Dean & Ghemawat, 2004).

4. Indexing in MapReduce

In this section, we describe multiple existing approaches to indexing under MapReduce. Firstly, we describe two possible interpretations of indexing as envisaged by Dean and Ghemawat in their original seminal MapReduce paper (Dean & Ghemawat, 2004) (Section 4.1), of which one is currently employed by the Ivory IR system (Lin, Metzler, Elsayed, & Wang, 2009) (Section 4.1.2). Then, we describe two alternative MapReduce indexing strategies used by the Nutch (Section 4.2) and Terrier IR platforms (Section 4.3), respectively.

4.1. MapReduce indexing as envisaged by Dean and Ghemawat

The original MapReduce paper by Dean and Ghemawat (2004) 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 paper (Dean & Ghemawat, 2004) is that efficient indexing could be trivially implemented in MapReduce. However, we argue that this claim 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 and 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. Per-token indexing

The literal interpretation of the description above would be for the map function to output a set of <term, doc-ID> pairs for each token in a document. Then, the reduce function aggregates for each unique term the doc-IDs in which that term occurs to obtain the term frequencies, before writing the completed posting list for that term to disk. Fig. 3 provides a pseudo-code implementation of the map and reduce functions for this strategy.

Per-Token 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 Token in the DocContents loop
4 :   Stem(Token)
5 :   deleteIfStopword(Token)
6 :   if (Token is not empty) then emit(Token, 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)
```

Per-Token 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)
```

Fig. 3. Pseudo-code interpretation of the per-token indexing strategy (map emitting <term,doc-ID>, Section 4.1.1).

Importantly, for this strategy, if a single term appears *tf* times in a document then the <term, doc-ID> pair will be emitted *tf* times. This strategy has the advantage of a very simple map phase, as it emits on a per-token basis. Hence, we refer to this as the *per-token* indexing strategy. Indeed, this strategy 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 reduce task. Hence, 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-reduce network traffic, as well as lengthening the sort phase. These are likely to have an effect on the job's overall execution time.

4.1.2. Per-term indexing (Ivory)

In the per-token indexing strategy above, one emit is made for every token in the collection. We claim that this is excessive, and will cause too much disk input/output (IO) when writing intermediate map output to disk, and again when moving map output to the reduce tasks. To reduce this effect, we could instead emit <term,(doc-ID, *tf*)> tuples, where *tf* is the term frequency of the term in the 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 task for this interpretation is also much simpler than the earlier interpretation, as it only has to sort instances by document to obtain the final posting list sorted by ascending doc-ID. Indeed, as this strategy emits once per-term, we refer to this as the *per-term* indexing strategy.

It is of note that this approach is implemented by the Ivory IR system (Lin et al., 2009) for inverted indexing. The pseudo-code implementation of the map and reduce functions for this strategy are shown in Fig. 4. It should also be noted that the per-token strategy can be adapted to generate *tf*s instead, through the use of a combiner function (see Section 3), 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 is still inefficient, because a large amount of disk IO is required to store, move and sort the temporary map output data.

4.2. Per-document indexing (Nutch)

The Apache Software Foundation's open source Nutch platform (Cafarella & Cutting, 2004) 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 indexing strategy differs from the general outline described in Section 4.1 above. Instead of emitting terms, this strategy only tokenises the document during the map phase, hence emitting <doc-ID, Document> tuples from the map function. Each document contains the textual forms of each term and their corresponding frequencies. The reduce phase is then responsible for writing all index structures. Pseudo-code implementations of map and reduce functions for this strategy are shown in Fig. 5. As one emit is made per-document, we refer to this strategy as *per-document* indexing.

Compared to emitting <term,(doc-ID, *tf*)>, this indexing strategy 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 that this is a step-forward towards reducing intermediate map output, since the larger document-level emits can achieve higher lev-

Per-Term 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,tf)) pairs, one for each term
      in the document
3: for each Token in the DocContents loop
 4 :    Stem(Token)
 5 :    deleteIfStopword(Token)
 6 :    if (Token is not empty) then increment stored tf
        for the Term=Token
7: end loop
8: Add document to the Document Index
9: for each Term where tf does not equal 0 loop
10 :    emit(term, (doc-ID,tf))
11: end loop
12: if (lastMap()) write out information about the
13:     documents this map processed ("side-effect" files)
```

Per-Term Indexing -
Reduce function pseudo-code

```
1: Input
      Key: A Term
      Value: List of (doc-ID,tf) pairs, 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
7 :    correct doc-ID
8 :    add doc-ID and tf to Posting-List
9 : end loop
10: emit(Posting-List)
```

Fig. 4. Pseudo-code interpretation of the per-term indexing strategy (map emitting <term,(doc-ID,tf)>, Section 4.1.2).

Per-Document Indexing - Map function pseudo-code

```
1: Input
       Key: Document Identifier, Name
       Value: Contents of the Document, DocContents
2: Output
       Key: A Doc-ID
       Value: A compressed Document, Document
3: for each Token in the DocContents loop
4 :    Stem(Token)
5 :    deleteIfStopword(Token)
6 :    if (Token is not empty) then increment stored tf
       for the Term=Token
7: end loop
8: Add document to the Document Index
9: for each Term where tf does not equal 0 loop
10 :   add term to compressed Document representation
11: end loop
12: emit(Doc-ID,Document)
```

Per-Document Indexing - Reduce function pseudo-code

```
1: Input
       Key: A Doc-ID
       Value: A compressed Document, Document
2: Output
       Key: Term
       Value: Posting List
3: index(Document) into in-memory Posting-Lists
4: if (lastMap())
5 :    for each Posting-List in-memory loop
6 :        emit(Posting-List)
7 :    end loop
```

Fig. 5. Pseudo-code interpretation of per-document style indexing (map emitting <Doc-ID,Document>, Section 4.2).

els of compression than single terms. A central advantage of the Nutch approach is that documents are sorted by document name (e.g. URL), allowing anchor text from other documents to be indexed on the same reduce task. However, a study of anchor text indexing techniques is outwith the scope of this study. Hence, we examine the scalability of this technique for indexing document content only.

4.3. Per-posting list indexing (Terrier)

Lastly, we examine the MapReduce indexing strategy we previously proposed in (McCreadie, Macdonald, & Ounis, 2009, 2009) and that is currently employed by the Terrier IR platform. This approach is based upon the single-pass indexing strategy described earlier in Section 2.2. The indexing process is split into multiple map tasks. Each map task operates on its own subset of the data, and is similar to the single-pass indexing corpus scanning phase. In particular, as documents are processed by each map task, compressed posting lists are built in memory for each term. 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 for all terms currently in memory. These flushed partial indices 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. Fig. 6 provides a pseudo-code implementation of map and reduce functions for this indexing strategy.

The fundamental difference between this strategy and those described earlier, is what the map tasks emit. In contrast to the per-token, per-term and per-document indexing strategies, this strategy builds up a posting list for each term in memory. Over many documents, memory will eventually be exhausted, at which time all currently stored compressed posting lists will be flushed as <term,posting list> tuples. Hence, we refer to this as a *per-posting list* indexing strategy. This has the positive effect of minimising both the size of the map task output, as well as the frequency and number of emits. Moreover, as the posting lists themselves are compressed using Elias-Gamma compression, the size of the emits will be very small.[3] Compared to the per-token and per-term indexing strategies, the per-posting list strategy emits far less, but the size of each emit will be much larger. Compared to the per-document indexing strategy, there will likely be less emits (dependent on memory available), and as the reduce task is operating on term-sorted data, it 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.

4.4. Discussion

A summary of the map output types of the various MapReduce indexing algorithms is provided in Table 1. We have implemented all of these algorithms within the auspices of the Terrier IR system. In the following, we discuss some notable

[3] While Hadoop provides GZip compression to reduce the size of map outputs, using Elias-Gamma compression of doc-IDs and term frequencies in the posting lists is superior to GZip alone.

Per-Posting List 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 into the in-memory
      Posting List for that term
7: end loop
8: Add document to the Document Index
9: if (lastMap() or outOfMemory()) then
      for each Term in the in-memory Posting Lists
      emit(Term, Posting List)
10: if (lastMap()) write out information about the
11:   documents this map processed ("side-effect" files)
```

Per-Posting List 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)
```

Fig. 6. Pseudo-code for the per-posting list indexing strategy (Section 4.3).

Table 1
Summary of the map output types of the various MapReduce indexing strategies.

Algorithm	Map output		Reduce partitioning	
	Key	Value	Term	Map
Per-token	Term	doc-ID	✓	✓
Per-term (Ivory)	Term	(doc-ID, tf)	✓	✓
Per-document (Nutch)	doc-ID	Document	×	✓
Per-posting list (Terrier)	Term	Posting list (compressed)	✓	✓

implementation details, which remain consistent across our implementations of these algorithms. Firstly, it should be reiterated 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.

Secondly, we note that if only one reduce task is used, then a single index would be generated. However, because of the lack of parallelism, such a scenario is slow, and should be avoided (McCreadie, Macdonald, & Ounis, 2009). Instead, multiple reduce tasks should be used to ensure high parallelism, and hence high efficiency (White, 2009). Intuitively, there are two partitioning schemes that could be applied for indexing, depending upon the desired output index format. In particular, *by-map* partitioning will result in all posting lists from one map task going to a particular reduce task. The effect of this that each reduce task will create an index for a subset of the document in the corpus, in a similar fashion to *by-document* partitioning in distributed indexing (Cacheda, Plachouras, & Ounis, 2005). An alternative is *by-term* partitioning (Brin & Page, 1998) of the reduce tasks, with the final inverted index being split across the files created by each reduce task. Table 1 also shows the support of the various MapReduce indexing strategies for the by-term and by-map partitioning schemes. It is of note that the per-document strategy does not support by-term partitioning, as each map output is the aggregation of multiple terms, while, in contrast, the other strategies can support both by-term and by-document partitioning.

In our implementation of by-term partitioning, terms are partitioned alphabetically by the first letter they contain. Hence, for an English corpus, one reduce task could handle terms for one or more of the 26 characters. An alternative is to randomly partition terms amongst the reduce tasks.[4] However, while this ensures a good balance of reduce task workload, the posting lists for lexicographically adjacent terms are not located adjacently in the inverted index – this may be a desirable quality for the inverted index to have for further processing, but is not required for efficient retrieval. Hence, we employ lexicographic by-term partitioning in this paper.

Lastly, we note that in keeping with the MapReduce paradigm, all of the indexing strategies described earlier are fully fault tolerant. If any map or reduce task fails then an identical task will be added to the queue of tasks for that job. Multiple

[4] Indeed, from inspection of the source of the Ivory IR system v0.2, we note that it adopts this partitioning approach.

Table 2
Statistics for 4 TREC test corpora of varying size used during experimentation.

Corpus	# Docs	Total pointers	Total # of tokens	Total # of unique terms	Compressed size	Uncompressed size (GB)
WT2G	247,491	62,921,466	159,712,427	1,002,586	603 MB	2
.GOV	1,247,753	280,988,127	911,881,613	2,788,457	4.5 GB	18
.GOV2	25,205,179	4,715,752,485	16,464,764,334	15,466,803	81 GB	426
ClueWeb09_English_1	50,220,423	14,885,245,469	35,089,166,902	74,238,222	229.9 GB	1422.3

failures by any single machine cause that machine to be black-listed and its work to be transferred elsewhere. Note that if a task should repeatedly fail then the job will be terminated, as a complete index is unable to be built.

5. Experiments and results

In the following experiments, we aim to determine the efficiency of multiple indexing implementations in MapReduce. In particular, we compare each of the four indexing approaches described in Section 4 in terms of processing time consumed at each stage of the indexing process. Furthermore, we also investigate how MapReduce indexing scales as we increase corpus size and horizontally scale hardware.

5.1. Research questions

To measure the efficiency of our implementations of the MapReduce indexing strategies and therefore the suitability (or otherwise) of MapReduce for indexing, we investigate four important research questions, which we address by experimentation in the remainder of this section:

1. Which of the four indexing strategies described earlier in Section 4 is the most efficient? (Section 5.4)
2. How does MapReduce indexing scale with corpus size? (Section 5.5)
3. Does MapReduce scale close to linearly as hardware is scaled horizontally? (Section 5.6)
4. What effect does the number of reduce tasks have on indexing performance? (Section 5.7)

5.2. Evaluation metrics

In this paper, we employ multiple metrics for measuring indexing performance. To compare how the four different indexing strategies described earlier (see Section 4) perform in detail, we measure the average time taken over all map and reduce tasks. In this way, we examine any tradeoff between map and reduce phase durations for each strategy, as well as their overall performance. However, to give a general overall measure of indexing performance, 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, measured in seconds.

Research questions 2, 3 and 4 address the suitability for indexing at a large-scale. We measure horizontal scalability (i.e. as more resources hardware is added) in terms of speed-up (S_m). Intuitively, speed-up ensures that not only should speed improve as more resources are added, but that such a speed increase should reflect the quantity of those resources. In particular, speed-up $S_m = \frac{T_1}{T_m}$, 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 (Hill, 1990). For instance, if we increase the available resources by a factor of 2, then it would be desirable to observe a 50% reduction in job duration, i.e. a doubling of speed. Linear speed-up (i.e. $S_m = m$) is the ideal scenario for parallel processing. However, it is 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 (Amdahl, 1967)), or due to overheads in job scheduling, monitoring and control.

5.3. Experimental setup

Following (Zobel, Moffat, & Ramamohanarao, 1996), which prescribes guidelines for evaluating indexing techniques, we now give details of our experimental cluster setup, consisting of 30 identical machines. Each machine contains two quad-core AMD Opteron processors, 16 GB of RAM, and contains one 250 GB SATA hard disk (manufacturer: Seagate, model number: ST32502N), of which 95 GB is available for MapReduce files. All machines run the Linux Centos 5.3 operating system and are connected together by a gigabit Ethernet switch 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 64 MB blocks, which are each replicated to two machines.[5] The machines are part of a dedicated Hadoop cluster which

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

Table 3
The total volume of data emitted from the Map tasks in addition to the average time in seconds for completion of the map/reduce tasks and total indexing time taken, for the four indexing algorithms described in Section 4 over two TREC test corpora. The lowest time or output size for each corpus is highlighted.

Corpus	Algorithm	Average map time	Map output MB	Average reduce time	Total time
WT2G	Per-token	9.937	2826	14.461	152
	Per-term	3.055	1372	5.730	107
	Per-document	**2.691**	459	5.423	**81**
	Per-posting list	3.216	**449**	**0.923**	**81**
.GOV	Per-token	63.463	16,486	74.961	600
	Per-term	26.303	6258	29.538	342
	Per-document	**11.792**	2008	24.038	187
	Per-posting list	17.654	**1619**	**5.153**	**142**

can process multiple jobs simultaneously (assuming there are enough processing cores to service them all). However, to avoid any possibility of interference between jobs, for example from network overloading, we run each job sequentially. Machines not allocated to a job are left idle, however these machines are still part of the distributed file system and as such can accept requests for data from each job. No other processes were running on the cluster during our experiments.

For reproducibility, in the following experiments, we employ four standard TREC Web corpora. In particular, we use three gigabyte-scale TREC corpora, i.e. WT2G, .GOV and .GOV2, as well as one terabyte-scale corpus ClueWeb09_English_1. The statistics of each corpora are presented in Table 2, including number of documents, number of tokens, number of unique terms, and the number of pointers (posting list entries). Note that ClueWeb09_English_1 is a subset of 50 million English documents from the ClueWeb09 corpus.[6] This subset was used for multiple tracks in TREC 2009 (Clarke, Craswell, & Soboroff, 2009), and was known as ClueWeb09 'B' set.

In the remainder of this paper, we use by-term partitioning for all indexing strategies, apart from the per-document strategy which does not support this partitioning scheme (see Section 4.4). Furthermore, in the following experiments (unless otherwise stated), each MapReduce job was allocated all 30 machines, 240 map tasks and 26 reduce tasks. Notably, the number of map tasks chosen was based upon the maximum number of tasks that could be run in parallel, i.e. 240 tasks since each of our 30 machines contributed 8 processors. The small number of reduce tasks was mandated by our by-term partitioning strategy, i.e we have one reduce task per initial character in the English alphabet (see Section 4.4). Lastly, we note that the output of each map task is automatically and transparently compressed by Hadoop using GZip, to reduce disk IO and network traffic between map and reduce tasks.

5.4. Algorithmic indexing efficiency

We experiment to determine the efficiency of multiple MapReduce indexing strategies, as per our first research question (see Section 5.1). To evaluate this, for each indexing strategy, we record the average time to complete each of the MapReduce map and reduce phases. In particular, in each map task, documents are being read from disk, tokenised, and possibly aggregated. Reduce tasks are responsible for writing the final index structures. Furthermore, we also measure the volume of map output data generated, as well as the overall time taken by each indexing job.

Table 3 shows the average time taken by the map and reduce phases, the total time taken and the volume of map output data when indexing the TREC WT2G and .GOV corpora with 30 machines, 240 map tasks and 26 reduce tasks. Note that for this test we report performance only on the smaller gigabyte-scale corpora, since evaluation of the per-token strategy proved to be impractical for indexing at a larger scale, i.e. the volume of local storage required exceeded that available to us.

Firstly, we observe that per-token indexing is, as expected, the slowest overall, likely due to the large volume of emitting which has to take place during the map phase. This also negatively impacts the reduce phase, as large amounts of data need to be transferred to the reducer, before being merged together into the index structures.

Secondly, we can see that per-term style indexing is faster both for the map and reduce phases, and overall (e.g. 107 vs. 152 s for WT2G). This is due to the reduced map output size, which is half that of the per-token strategy. However, per-term indexing is still slower than the per-document and per-posting list strategies, which exhibit smaller map output sizes and smaller overall indexing times.

Lastly, we compare the per-document and per-posting list indexing strategies. We note that per-document indexing produces shorter map phases, as less processing takes place in each map task. In contrast, the map phase of per-posting list indexing is slower because posting lists are built by each mapper. However, by making this extra processing at the map phase, the size of the map output is reduced (e.g. from 459 MB to 449 MB for WT2G and 2008 MB to 1619 MB for .GOV) and moreover, the reduce tasks are markedly faster, resulting in an overall shorter indexing job. In particular, in our implementation of the per-document indexing strategy, each reduce task corresponds to running a single-pass indexer on already tokenised documents. In contrast, the reduce tasks for the per-posting list indexing strategy work with much less data, and are merely correcting the doc-IDs before writing to the final inverted index files (see Section 4.3). As such, although the dif-

[6] This cluster did not have sufficient hard disk space to store the minimum 2 copies of the entire ClueWeb09 corpus on the DFS.

ference in processing time reported for these two strategies is not large, we would expect the disparity in performance to continue to escalate as the corpus size grows. This is because under the per-posting list strategy, the savings that result from posting-list compression will become more pronounced as posting lists become larger, whilst per-document compression will remain roughly constant.

In summary, the simple per-token and per-term indexing strategies as interpreted from the original MapReduce paper (Dean & Ghemawat, 2004) (of which the latter is employed by the Ivory IR system (Lin, 2009)) are markedly less efficient than more recent per-document and per-posting list indexing approaches deployed by the Nutch and Terrier IR platforms. Of these two, the per-posting list implementation is overall more efficient. As such, in the remainder of these experiments we deploy the per-posting list MapReduce indexing implementation only.

5.5. Scaling to larger corpora

In Section 5.4, we determined that the per-posting list MapReduce indexing strategy was the most efficient of those examined. In this section, we investigate our second research question (see Section 5.1), to determine how this strategy scales as corpora size increases. In particular, we compare indexing performance over three gigabyte-scale TREC test corpora: WT2G, .GOV and .GOV2, as well as one terabyte-scale TREC corpus, i.e. ClueWeb09_English_1. For MapReduce to be viable as a platform for distributed indexing, overall indexing throughput should be maintained as corpus size increases.

Table 4 shows the indexing throughput for each of the four TREC test corpora based upon its compressed index size. From the results in the table, we observe that throughput shows an unexpectedly large variance over the test corpora. To examine this in more detail, we provide an analysis of each indexing job. In particular, Fig. 7a–d show the time taken by each individual map when indexing the four investigated corpora.

Taking each corpus in turn, Fig. 7a illustrates the reason for the low throughput for WT2G. In particular, we can see that many of the map tasks do not start until the job is well under way. This is due to the manner in which Hadoop assigns map tasks to available machines. In particular, Hadoop only assigns two map tasks to a given machine in a 5 s iteration. For small jobs, such as indexing WT2G where each map task only lasts 10 s, this is a significant overhead to the startup of the indexing job. This issue was also identified by O'Malley and Murthy (2009) when deploying MapReduce for sorting large datasets – it is of note that future versions of Hadoop have optimisations to decrease this latency.

From Table 4, we see that the indexing of the .GOV corpus had 4.5 times the throughput of WT2G. However, on examination of Fig. 7b, we see that it is similarly affected by latency in task startup. In contrast, for the .GOV2 and ClueWeb09_English_1 corpora (Fig. 7c and d), map tasks took significantly longer, and hence, startup latency overheads were negligible, with overall indexing times and throughputs being significantly higher. Indeed, for the two largest corpora, indexing throughput was approx. 65–75 MB/s.

For .GOV and .GOV2, there is a noticeable downwards trend in map duration for the later map tasks. This is explained by the file size distributions in these corpora, which decreases for later bundles of documents. In our implementation, corpora files are split evenly across map tasks – a better splitting would spread files evenly across the map tasks based on their size. In contrast, for ClueWeb09_English_1, the distribution of work over the map tasks is fairly even, with the exception of the last few map tasks, which are smaller files containing Wikipedia articles.

To summarise, our indexing implementation in MapReduce shows promising throughput, which is increased as collection size increases, due to the reduced overheads in starting short-lived map tasks. Indeed, this supports the notion that MapReduce indexing is scalable, as the rate at which a corpus is indexed does not degrade for larger corpora.

5.6. Horizontal hardware scaling

We now investigate how well indexing under MapReduce performs as we scale hardware horizontally with processing power, i.e add more machines for work (third research question, see Section 5.1). As mentioned in Section 2.3, when distributed indexing uses machine-local data, optimal linear scaling can be achieved. The alternative to shared-nothing is to use a "shared-everything" setting, where central fileserver(s) serve the corpora to be processed by multiple indexing machines. However, such a centralised approach can suffer from a single point of failure. Moreover, centralised fileservers are often a bottleneck – indeed, we have previously shown that, for indexing, a single RAID-based fileserver scaled to only 4 machines (3 processors per machine) (McCreadie et al., 2009).

As noted in Section 3, a core advantage of MapReduce is the ability to apply the distributed file system (DFS) to avoid centralised storage of data and, moreover, to take advantage of data locality to avoid excess network IO. Using MapReduce for indexing cannot be as fast as a shared-nothing indexing setting, due to overheads in job setup, monitoring and control

Table 4
Throughput for the per-posting list MapReduce indexing strategy over four TREC test corpora using 30 machines, 240 map tasks and 26 reduce tasks.

Corpus	Compressed size (MB)	Time taken (s)	Throughput (MB/s)
WT2G	614	81	7.17
.GOV	4608	142	32.45
.GOV2	82,944	1306	63.51
ClueWeb09_English_1	235,417	3167	74.33

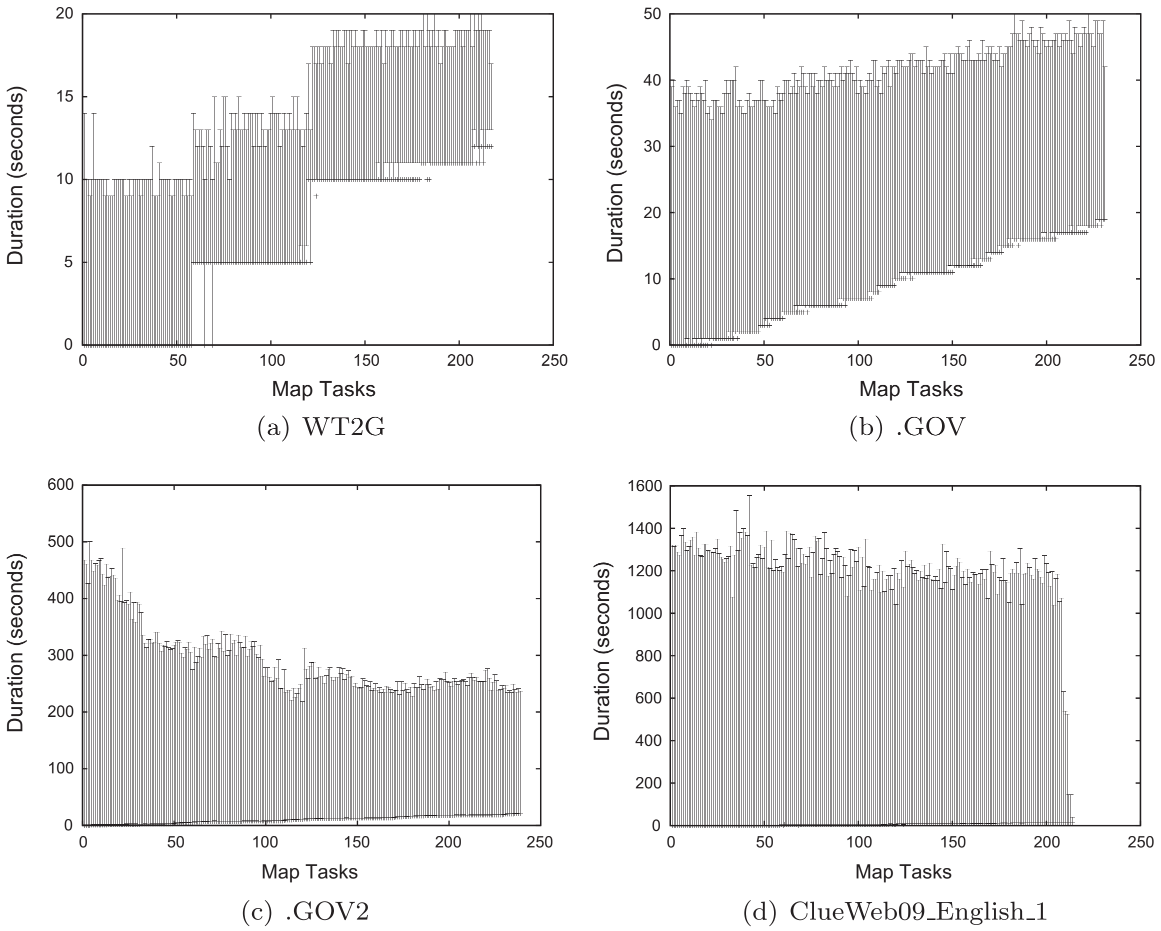

Fig. 7. The time taken (s) by each individual map task for each of the four TREC test corpora investigated when indexing using the per-posting list indexing strategy over 30 machines and 240 map tasks.

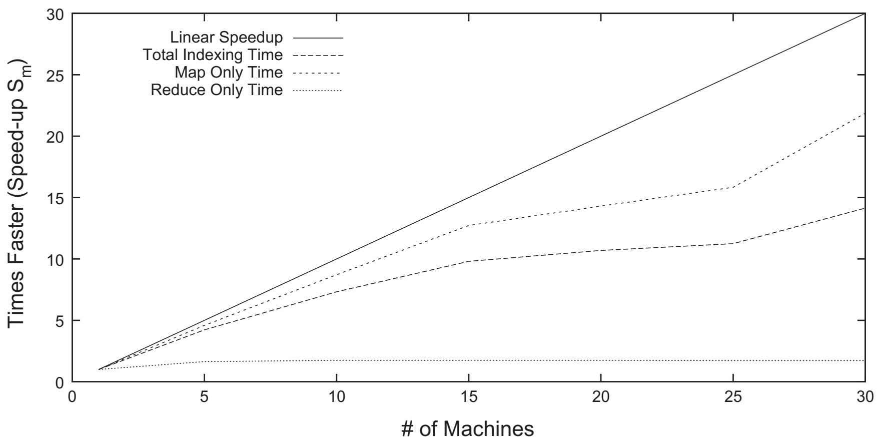

Fig. 8. The speed-up observed upon indexing with the per-posting list indexing strategy over the ClueWeb09_English_1 corpus using {1,5,10,15,20,25,30} machines, when allocating 240 map tasks and 26 reduce tasks, both for the indexing process as a whole and the individual map and reduce phases.

(c.f. Amdahl's law). Hence, we evaluate MapReduce indexing in terms of the extent to which throughput increases in a linear fashion with the machines allocated for work.

In this section, we measure the extent to which indexing under MapReduce scales linearly with processing power. In particular, we measure the speed-up gained as more machines are allocated for map and reduce indexing tasks. Fig. 8 reports

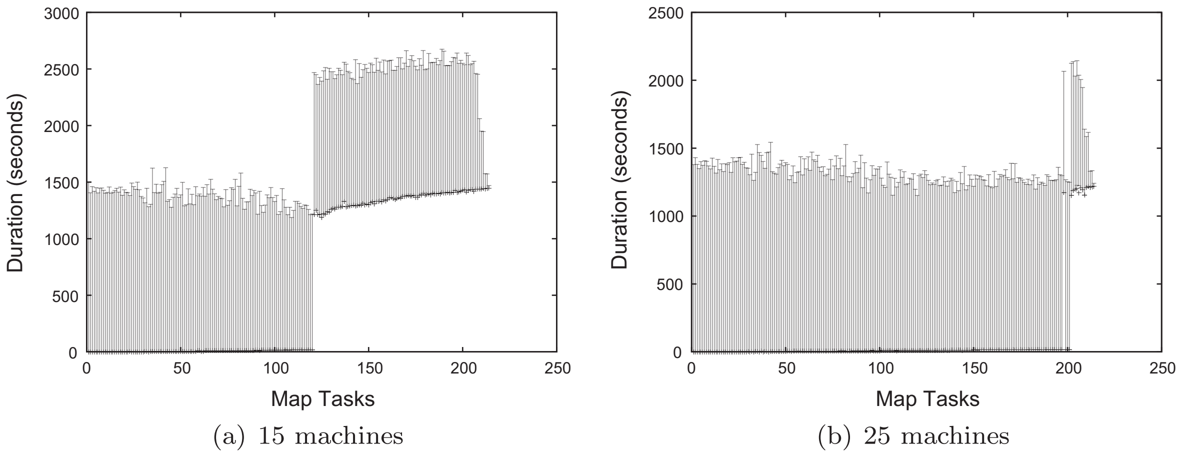

Fig. 9. The time taken (s) by each individual map task in the job as a whole when indexing with the per-posting list strategy on the ClueWeb09_English_1 corpus when using 15 and 25 machines over 240 map tasks.

the speed-up gained upon indexing the largest corpus investigated, i.e. the ClueWeb09_English_1 corpus, using $\{1,5,10,15,20,25,30\}$ machines, when allocating 240 map tasks and 26 reduce tasks, both for the indexing process as a whole and the individual map and reduce phases.

From Fig. 8, we observe that, as expected, MapReduce indexing as a whole scales sub-linearly. However, if we deconstruct the indexing process into map and reduce tasks, we observe that the map tasks scale better with processing power than the job as a whole. Indeed this is close enough to perfect linear scaling that we would deem it satisfactory for large-scale indexing. However, as more machines are allocated, the reduce task does not scale with processing power in this setting, thereby decreasing overall scalability.

In explanation, recall that the total number of reduce tasks allocated is 26 – one for terms starting with each letter of the English alphabet. Hence, when five or more machines are allocated, at least 40 processors will be available for reducing – more than the number of reduce tasks. In these cases, once the reduce phase is underway, many processing cores will be left idle. However, this is not an insurmountable issue, as by splitting the map output by multiple characters instead of just the first, the number of final output indices, and therefore reduce tasks required increases, hence supporting a higher degree of parallelism.[7] In particular, under such a 'prefix-tree' approach, the partitioner would allocate terms to reducers based on the expected size of the posting lists for terms with similar prefixes. The expected distribution would need to be obtained from a source outwith the corpus being indexed, as the partitioner of each map task does not have access to the global term prefix distribution.

From Fig. 8, we also note that speed-up does not increase smoothly as the number of machines is increased. Indeed, between 15 and 25 machines, speed-up is less than at other points in the figure. To explain this issue, we refer to Fig. 9, which shows the time taken by each of the 240 map tasks when indexing the ClueWeb09_English_1 corpus with both 15 and 25 machines. From this, we note that for both indexing jobs, not all map tasks can be processed at once, as the number of map tasks exceeds the number of processing cores available. Instead, in both cases, as many map tasks as possible are run, and then the remaining are held until a processor becomes free. Unfortunately, this results in a similar processing time for both jobs, even though the latter task has 40% more processors available. This is a direct result from the majority of cores are left idle during the second 'batch' of jobs for the 25 machine task. From this, we can see that it is important to set the number of map tasks with reference to the number of processing cores available. This contrasts to normal Hadoop configuration, where the number of map tasks is a constant, or defined according to the number of splits of the input data that can be obtained.

Overall, we conclude that the map task within the indexing process scales in a fashion that is suitable for large-scale indexing. Indeed, this is an important result, as typically the map phase dominates the indexing job in terms of time taken. Hence this is an considerable step toward showing that MapReduce indexing is scalable. Moreover, we have shown that, ideally, the number of map tasks should be approximately equal to the number of processors available for work (for fault tolerance, leaving a few processors free allows single task failures to be quickly re-run (White, 2009)). However, with only 26 reduce tasks, scaling beyond 26 processors did not decrease reduce phase time. In the next section, we examine how varying the number of reduce tasks affects indexing time.

5.7. Reducer scaling

In this section, we examine how well the reduce phase of a MapReduce indexing job scales with regard to the number of processors. In particular, we evaluate the total speed-up observed for the reduce phase as we increase the number of reduce

[7] At the time of writing, Terrier's standard lexicon format allows for an inverted index split across a maximum of 32 files.

tasks. Ideally, speed-up should be close to linear with the number of reduce tasks allocated, assuming that map phase output is evenly split across reduce tasks.

Fig. 10 shows the speed-up observed upon indexing the ClueWeb09_English_1 corpus on 30 machines using {1,4,8,12,16,10,24,26} reduce tasks using the per-posting list indexing strategy. In particular, we can see that from 1 to 26 reduce tasks, close-to linear speed-up is achieved.

To examine the reasons that exact linear speed-up is not achieved, we examine the distribution of all reduce task durations. Fig. 11 shows the distribution of reduce task durations for indexing the ClueWeb09_English_1 corpus. From the figure, we observe that not all reduce tasks take a uniform duration. This is expected, as the initial character of indexed terms will not have a uniform distribution. Indeed, it can be seen that the reduce task for character 'e' takes the longest, as it is the most frequent initial character in the ClueWeb09_English_1 corpus. Similarly, '*x*', '*y*' and '*z*' have smaller reduce tasks, due to their infrequent nature.

Overall, as both the map (see Section 5.6) and reduce tasks have the potential to scale in a fashion, which is suitable for large-scale indexing. Hence, we conclude that MapReduce in general is suitable for indexing terabyte and larger scale collections.

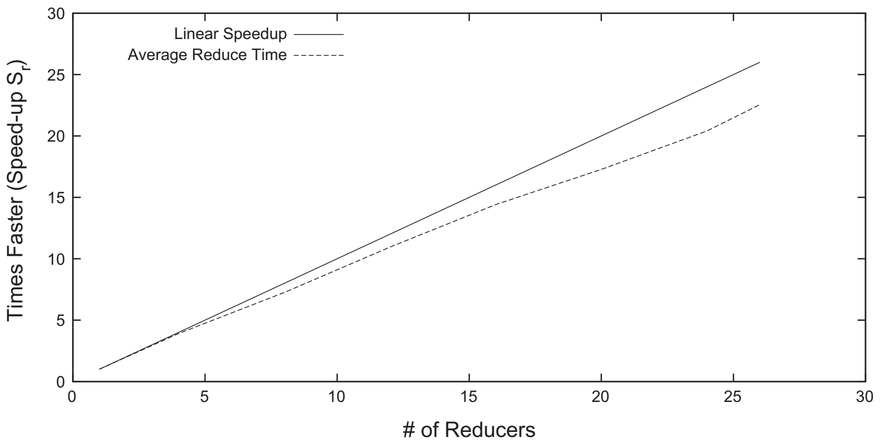

Fig. 10. The speed-up observed upon indexing the ClueWeb09_English_1 corpus on 30 machines using {1,4,8,12,16,10,24,26} reduce tasks.

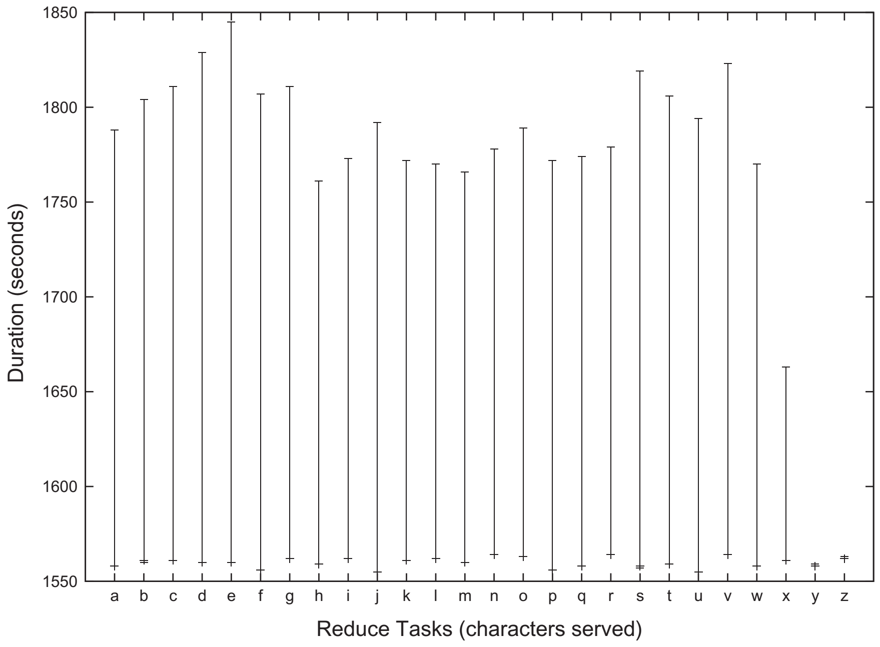

Fig. 11. The distribution of reduce task durations for indexing the ClueWeb09_English_1 corpus with the per-posting list indexing strategy using 30 machines and 26 reduce tasks.

6. Conclusion

In this paper, we investigated the common IR process of indexing within the context of the distributed processing paradigm MapReduce. In particular, we detailed four different strategies for applying document indexing within the MapReduce paradigm.

Through experimentation on four standard TREC test corpora of varying size and using the Java Hadoop implementation of MapReduce, we experimented upon those four indexing strategies. We firstly showed that of the four, the two interpreted from Dean and Ghemawat's original MapReduce paper (Dean & Ghemawat, 2004), i.e. per-token and per-term indexing, generate too much intermediate map data, causing an overall slowness of indexing. Indeed, we conclude that these strategies are impractical for indexing at a large-scale.

Moreover, while the per-document indexing strategy employed by the Nutch IR platform proved to be more efficient, it is compromised by lengthy reduce phases resulting from their increased responsibilities. In contrast, the per-posting list indexing strategy that we previously proposed, and that is employed by the Terrier IR platform, proved to be the most efficient indexing strategy. This is due to its use of local machine memory and posting list compression techniques to minimise map-to-reduce traffic.

Furthermore, we examined how per-posting list indexing scaled with both corpus size and horizontal hardware scaling. In particular, we showed that while our specific implementation of the indexing strategy was suboptimal, the algorithm overall scales in a close to linear fashion as we scale hardware horizontally and that variance observed in indexing throughput could be attributed to time lost as Hadoop assigned tasks to machines, which is negligible on terabyte-scale corpora.

Overall, we conclude that the per-posting list MapReduce indexing strategy should be suitable for efficiently indexing larger corpora, including the full billion document TREC ClueWeb09 collection. Indeed, we have shown indexing performance on a 50 million document subset of that 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 per-posting list MapReduce indexing strategy described in this paper is now freely available for use by the community as part of the Terrier IR Platform.[8]

Acknowledgements

We are thankful to Mark Meenan and Derek Higgins of the University of Glasgow Computing Service for timely and exclusive access to their computer cluster.

References

Apache Software Foundation. Apache Hadoop Project. (2010). <http://hadoop.apache.org/> Accessed 08.02.10.
Amdahl, G. (1967). Validity of the single processor approach to achieving large-scale computing capabilities (pp. 483–485).
Brin, S., & Page, L. (1998). The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems, 30*(1–7), 107–117.
Cacheda, F., Plachouras, V., & Ounis, I. (2005). A case study of distributed information retrieval architectures to index one terabyte of text. *Information Processing & Management, 41*(5), 1141–1161.
Cafarella, M., & Cutting, D. (2004). Building Nutch: Open source search. *Queue, 2*(2), 54–61.
Clarke, C. L. A., Craswell, N., & Soboroff, I. (2009). Preliminary report on the TREC 2009 Web track. In *TREC'09: Notebook of the 18th text retrieval conference, Gaitherburg, USA*.
Chu, C.-T., Kim, S. K., Lin, Y.-A., Yu, Y., Bradski, G. R., Ng, A. Y., et al (2006). Map-reduce for machine learning on multicore. In *NIPS: Proceedings of the 20th annual conference on neural information processing systems* (pp. 281–288). Canada: Vancouver.
Dean, J., & Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. In *OSDI: Proceedings of the 6th symposium on operating systems design, San Francisco, USA* (pp. 137–150).
Dean, J., & Ghemawat, S. (2010). Mapreduce: A flexible data processing tool. *Communications of the ACM, 53*(1), 72–77.
Elias, P. (1975). Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory, 21*(2), 194–203.
Ghemawat, S., Gobioff, H., & Leung, S.-T. (2003). The Google file system. *SIGOPS Operating Systems Review, 37*(5), 29–43.
Heinz, S., & Zobel, J. (2003). Efficient single-pass index construction for text databases. *JASIST, 54*(8), 713–729.
Hill, M. D. (1990). What is scalability? *SIGARCH Computer Architecture News, 18*(4), 18–21.
Google Inc. Protocol Buffers: Google's data interchange format. (2010). <http://code.google.com/p/protobuf/> Accessed 26.01.10.
Isard, M., Budiu, M., Yu, Y., Birrell, A., & Fetterly, D. (2007). Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys'07: Proceedings of the 2nd ACM SIGOPS/EuroSys European conference on computer systems, Lisbon, Portugal* (pp. 59–72).
Johnson, R. E. (1997). Frameworks = (components + patterns). *Communications of the ACM, 40*(10), 39–42.
Laclavik, M., Seleng, M., & Hluchý, L. (2008). Towards large scale semantic annotation built on mapreduce architecture. In *ICCS'08: Proceedings of the 8th international conference on computational science, Krakow, Poland* (pp. 331–338).
Lin, J., Metzler, D., Elsayed, & T., Wang, L. (2009). Of Ivory and smurfs: Loxodontan MapReduce experiments for Web search. In *TREC'09: Proceedings of the 18th text retrieval conference, Gaithersburg, USA* (pp. 125–134).
Lin, J. (2009). Data-intensive text processing with MapReduce. Tutorial at *SIGIR '09: The 32nd international ACM SIGIR conference on research and development in information retrieval, Boston, USA.*
Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to information retrieval*. Cambridge University Press.
McCreadie, R. M. C., Macdonald, C., & Ounis, I. (2009). On single-pass indexing with MapReduce. In *SIGIR '09: Proceedings of the 32nd international ACM SIGIR conference on research and development in information retrieval, Boston, USA* (pp. 742–743).
McCreadie, R. M. C., Macdonald, C., & Ounis, I. (2009). Comparing distributed indexing: To MapReduce or not? In *LSDS-IR'09: Proceedings of the 2009 workshop on large-scale distributed systems for information retrieval, Boston, USA* (pp. 41–48).
Melnik, S., Raghavan, S., Yang, B., & Garcia-Molina, H. (2001). Building a distributed full-text index for the web. In *WWW '01: Proceedings of the 10th international conference on World Wide Web, Hong Kong, China* (pp. 396–406).
O'Malley, O., & Murthy, A. C. (2009). Winning a 60 second dash with a yellow elephant. In *Proceedings of sort benchmark*.

[8] http://terrier.org.

Olston, C., Reed, B., Srivastava, U., Kumar, R., & Tomkins, A. (2008). Pig Latin: A not-so-foreign language for data processing. In *SIGMOD'08: Proceedings of 34th ACM international conference on management of data (industrial track), Vancouver, Canada* (pp. 1099–1110).

Ounis, I., Amati, G., Plachouras, V., He, B., Macdonald, C., & Lioma, C. (2006). Terrier: A high performance and scalable information retrieval platform. In *OSIR'06: Proceedings of 2nd international workshop on open source information retrieval, Seattle, USA* (pp. 18–25).

Pavlo, A., Paulson, E., Rasin, A., Abadi, D. J., DeWitt, D. J., Madden, S., et al. (2009). A comparison of approaches to large-scale data analysis. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on management of data, Rhode Island, USA* (pp. 165–178).

Pike, R., Dorward, S., Griesemer, R., & Quinlan, S. (2005). Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming, 13*(4), 277–298.

Ribeiro-Neto, B. A., de Moura, E. S., Neubert, M. S., & Ziviani, N. (1999). Efficient distributed algorithms to build inverted files. In *SIGIR '99: Proceedings of the 22nd annual international ACM SIGIR conference on research and development in information retrieval, Berkeley, USA* (pp. 105–112).

Schonfeld, E. (2008). Yahoo search wants to be more like Google, embraces Hadoop, Febuary 2008. <http://www.techcrunch.com/2008/02/20/yahoo-search-wants-to-be-more-like-google-embraces-hadoop/> Accessed 09.02.10.

Tomasic, A., & Garcia-Molina, H. (1993). Performance of inverted indices in shared-nothing distributed text document information retrieval systems. In *PDCS'93: Proceedings of the 2nd international conference on parallel and distributed information systems, Massachusetts, USA* (pp. 8–17).

White, T. (2009). *Hadoop: The definitive guide*. O'Reilly Media.

Witten, I. H., Moffat, A., & Bell, T. C. (1999). *Managing gigabytes: Compressing and indexing documents and images*. Morgan Kaufman.

Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, U., Kumar, P., et al. (2008). DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI'08: Proceedings of the 8th symposium on operating system design and implementation, San Diego, CA* (pp. 1–14).

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