Abstract

Recent innovations in Big Data have enabled major strides forward in our ability to glean important insights from massive amounts of data, and to use these insights to make better decisions. Underlying many of these innovations is a computational paradigm known as MapReduce, which enables computational processes to be scaled up to very large sizes and to take advantage of cloud computing. While very powerful, MapReduce also requires a nontrivial shift in algorithm design strategies. In this paper we provide an overview of MapReduce and types of problems it is suited for. We discuss general strategies for designing MapReduce-based algorithms and provide an illustration using social media analytics.

Keywords: big data; analytics; MapReduce; algorithm

1. Introduction

Many new tools and technologies have arisen over the past several years to process Big Data. Although the specific technologies and applications for Big Data vary widely, one of the major enablers underlying many Big Data platforms is the MapReduce computational paradigm. Many MapReduce applications come from e-commerce and social media companies such as Amazon [1], Google [2, 3], Facebook [4], Twitter [5, 6]. Hadoop, one of the most widely used Big Data analytics platforms today, has MapReduce as its basis [7]. This paradigm is also used in scientific computing on large data sets [8, 9]. MapReduce provides an abstraction consisting of two functional interfaces: Map and Reduce.  MapReduce computational infrastructures automatically handle the tasks of partitioning data, parallelizing the processes, and routing the data over multiple nodes, leaving only the task of implementing the Map and Reduce functions for the algorithm developer.

MapReduce was originally introduced by Google in 2004 [2]. It was devised as a programming abstraction to enable writing code for the execution of very large parallel computational tasks that run on thousands of commodity machines, all without having to handle details involved with the parallelization and infrastructure management. MapReduce was inspired by functional programming (FP), an alternative programming paradigm that involves
constructing programs purely by composition of functions rather than executing sequential tasks [10]. MapReduce is based on two specific functions commonly used in FP, \textit{map} and \textit{reduce}. \textit{Map} applies a unary function to each element in a list to obtain a new list, while \textit{reduce} (also known as \textit{fold}) applies a binary function on elements of a list and sequentially reduces the elements into a single value [10]. To illustrate \textit{map}, consider the task of taking a list of numbers and computing a new list consisting of their squares. In the example shown below, \textit{map} applies the function \textit{square} to the list \([1, 2, 3, 4, 5]\) to obtain a new list containing the squared values:

\[
\text{square}(x) = x \times x \\
\text{map}(\text{square}, [1, 2, 3, 4, 5]) = [1, 4, 9, 16, 25]
\]

The \textit{reduce} function reduces a list of elements into a single element using a specified binary function. To illustrate \textit{reduce}, consider the tasks of computing the sum or product for a given list of numbers. We point out that binary arithmetic operators such as \textit{\texttt{+}} and \textit{\texttt{*}} are binary function shorthand (e.g. \texttt{+(x, y) = x + y}), and any binary function may be expressed in such a manner. Using \textit{reduce} it is quite trivial to compute sums and products, enabling the \textit{sum} and \textit{product} functions to be succinctly defined in terms of \textit{reduce}. These ideas are illustrated below:

\[
\text{reduce}(+, [1, 2, 3, 4, 5]) = 1 + 2 + 3 + 4 + 5 = 15 \\
\text{reduce}(*, [1, 2, 3, 4, 5]) = 1 \times 2 \times 3 \times 4 \times 5 = 120 \\
\text{sum}(\text{Numbers}) = \text{reduce}(+, \text{Numbers}) \\
\text{product}(\text{Numbers}) = \text{reduce}(*, \text{Numbers})
\]

\textit{Map} and \textit{reduce} are frequently composed together in practice to succinctly accomplish a wide variety of computations. For example the sum of squares, a building block of many statistical computations, may be expressed succinctly using \textit{map} and \textit{reduce}:

\[
\text{sum-of-squares}(\text{Numbers}) = \text{reduce}(+, \text{map}(\text{square}, \text{Numbers}))
\]

Big Data is essentially about data transformation processes. Large-scale data preparation and analysis tasks may be thought of as transformation processes which begin with raw data sets as input and end with the desired results as output. Generally there are three structural forms that may occur between the input and output: one-to-one, one-to-many, and many-to-one correspondence. While one-to-many transforms are typically associated with data generation, one-to-one and many-to-one transforms are central to the preparation and analytical tasks of Big Data. In the examples above it is readily seen that \textit{map} and \textit{reduce} are structural transformation functions. In particular, \textit{map} is associated with one-to-one transforms, while \textit{reduce} is associated with many-to-one transforms. MapReduce extends the structural transformation ideas of the FP \textit{map} and \textit{reduce} to provide a programming interface for implementing algorithms. One must simply implement their algorithm in terms of two analogous functions – \textit{Map} and \textit{Reduce} – and the algorithm may then be run within a MapReduce infrastructure. Algorithm developers are thus freed from having to handle the tasks of managing the parallelization, data partitioning, routing, and other aspects of scaling up the computation and may instead focus on the core algorithm design.

In MapReduce the raw input is in the form of \((\text{key}, \text{value})\) tuples. The Map function is applied to each \((\text{key}, \text{value})\) pair and in turn emits an intermediate \((\text{key}, \text{value})\) pair. A single key may be present in multiple \((\text{key}, \text{value})\) pairs emitted by the Map function. Prior to being sent to the Reduce function, emitted \((\text{key}, \text{value})\) pairs are aggregated together by key in a process known as “shuffle.” In this process all \((\text{key}, \text{value})\) pairs with matching keys are grouped together and turned into a single \((\text{key}, <\text{list of values}>)\) tuple. Each of these new tuples will then be sent to the Reduce function. The Reduce function takes as input a \((\text{key}, <\text{list of values}>)\) tuple and applies its operation, resulting in the final output for the given key. Reduce is called on each \((\text{key}, <\text{list of values}>),\) resulting in a final \((\text{key}, \text{value})\) tuple emitted for each key.

We illustrate this idea with a simple and well-known MapReduce example: counting word occurrences over an entire corpus of documents [2]. The raw input is a set of \((\text{ID}, \text{Content})\) pairs, where the \text{ID} represents a document ID and \text{Content} is a list of words containing the text of the document. The Map and Reduce functions are as follows:
Map(ID, Content):
  For each Word in Content:
   Emit(Word, 1)

Reduce(Word, OccurrenceList):
  Count <- sum(OccurrenceList)
  Emit(Word, Count)

In this example, Map is applied once to each document. It goes through a document word-by-word and emits each word encountered together with the number 1 as a new (key, value) pair. Prior to going to Reduce, the resultant (key, value) pairs are aggregated together by matching key into (key, [list of values]) pairs. For instance, if the word “mountain” occurs 4 times over the entire corpus (with each occurrence possibly in a different document) then the pair (mountain, 1) will be encountered and emitted by Map a total of 4 times. Prior to going to Reduce, these 4 pairs will be aggregated into the pair (mountain, [1,1,1,1]) and the same will happen to all other intermediate (key, value) pairs. Each new aggregated pair will then go to Reduce, where it’s list of 1’s will be totaled to give the grand count for each word. The final output is a (Word, Count) pair for each word found in the corpus.

2. Object Word Sentiment Analysis: A Social Media Example

Social media sources such as blogs, Facebook, Twitter, and others produce a continual stream of user-originated text content which contains many valuable insights. Broadly speaking, sentiment analysis is the application of algorithmic methods to a body of text to determine a measure of emotional charge. There are many types of sentiment analysis. Some are very sophisticated and involve tasks such as recognizing sarcasm within the text, or detecting and analyzing the sentiment of viewpoints. Another useful and much simpler form involves applying a scoring method to determine the overall tone (polarity) and intensity of a text passage. In this type of analysis, higher scores typically correspond to more positive sentiments while lower or negative scores correspond to more negative ones. A typical approach for this type of sentiment analysis involves using a predefined list of common words paired with their corresponding numerical sentiment scores [11]. Words in this list mainly consist of adjectives and verbs. To obtain the overall score for a passage of text, its words are simply extracted and then the sentiment scores for each word occurring in both the predefined list and the text passage are averaged together. The simplicity and broad applicability of this approach make it useful for tasks such as monitoring the changing mood of a community through their Tweets, or gauging customer reactions to a new product through blog posts.

One of the limitations of this approach is that it provides only a high-level analysis of a text passage. While it may (for instance) be able to distinguish between happy, neutral, or unhappy Tweets, it is not able to tell what people are positively or negatively reacting to. As a more substantive example of MapReduce, we show how to use ideas from this rudimentary form of sentiment analysis to infer sentiments of specific unscored words by analyzing a corpus of text passages. In the context of social media using (for instance) a set of recent Tweets of Facebook posts, this form of analysis may be used to reveal what a community is currently happy or unhappy about.

In considering this problem we first note that many words in a text passage are not generally of interest (such as those already in the predefined sentiment scored words, and stop words such as “a”, “the”, “I”, “at”, etc.). Accordingly, such words should first be filtered out from consideration. To solve our main problem, we utilize the principle of context: words of unknown sentiment that occur near known sentiment-scored words will generally reflect similar sentiments. For a given word we may thus use all of its contexts of occurrence over the entire corpus of documents (which in our case may be many millions of Tweets or Facebook posts) to infer a very accurate sentiment score. If we do this for all words within a large corpus of documents, it is clear that significant computational resources are required. The following illustrates a sequential implementation of this algorithm:
document-score-metrics(Text):
    WeightTotal <- 0
    ScoreTotal <- 0
    for each Word in Text:
        if Word is in Predefined Scored Words:
            WeightTotal <- WeightTotal + 1
            ScoreTotal <- ScoreTotal + score(Word)
    return (WeightTotal, ScoreTotal)

score-object-words(Corpus):
    WordWeights <- Empty dictionary of key : value pairs
    WordScores <- Empty dictionary of key : value pairs
    for each (ID, Text) in Corpus:
        (Weight, Score) <- document-score-metrics(Text)
        for each Word in Text:
            if Word not in [Predefined Scored Words UNION Stop Words]:
                if WordScores[Word] does not exist:
                    WordScores[Word] <- 0
                    WordWeights[Word] <- 0
    AvgScores <- Empty dictionary of key : value pairs
    for each Word in keys(WordScores):
    return AvgScores

For our purposes, we will consider two words to occur in the same context each time they occur together in the same document. Lines 1-8 define a simple auxiliary function which computes two score metrics: 1) the total number of words in the text which are also found in the predefined scored word list, and 2) the sum of the scores of all these words. The former may be regarded as a weight and the latter as a total. These are returned as a pair. Lines 9-24 contain the main logic for assigning a sentiment score to each word of interest based on its contexts over the entire corpus of documents. The variables WordScores and WordWeights are used to store the total sum of sentiment scores and the total weight of occurrences for each word, respectively, over the entire corpus. The nested for loops in lines 12-20 will go through every word of every document sequentially and total up the word weights and word scores. It is readily apparent that this will quickly become the bottleneck as the number of documents grows large. Finally, lines 21-23 go through each word extracted from the corpus and an average of the word totals is obtained, resulting in a final sentiment score for each word.

In order for this algorithm to be applicable to real-world problems it must be implemented in a manner that allows it to operate on data at Web scale. To accomplish this we may thus re-implement this algorithm using MapReduce. A key insight enabling us to do this is the fact that in our algorithm there is no dependence of one word on another, nor is one document dependent on another. The basic intuition for decomposing our algorithm into Map and Reduce functions is to exploit these non-dependencies. A MapReduce implementation is shown below:

Map(DocID, Text):
    (Weight, Score) <- document-score-metrics(Text)
    for each Word in Text:
        if Word not in [Predefined Scored Words UNION Stop Words]:
            Emit(Word, (Weight, Score))

Reduce(Word, ScorePairs):
    TotalWeight <- 0
In lines 25-29 the Map function takes a single document as input in the form of a \((\text{DocID}, \text{Text})\) pair and computes the score metrics based on the \textit{document-score-metrics} function in lines 1-8. For each admissible word it emits the word paired with the document total (line 28). Prior to going to Reduce, each emitted word will have all its corresponding score metric pairs put into a list through the shuffle process, and the word will go into the Reduce function paired with this list. In line 30 the Reduce function takes a single word and a list of score metric pairs as input. These score pairs contain all \((\text{weight}, \text{score})\) pairs emitted for the word across the entire corpus. In lines 33-36 the average sentiment score for the word is computed, and then finally emitted in line 37.

In the example above it is seen that each call to Map processes a single document, and each call to Reduce processes a single word. The data partitioning and routing is handled by the computational infrastructure and is dependent on the number of machines available. However, it is readily possible for each document to be processed via Map on a unique machine and likewise, for each word to be processed via Reduce on a unique machine.

We mention one further observation about the implementation above. In lines 27-29, each admissible word of a document will be emitted each time it is encountered within the document along with its payload of metrics. If certain words occur with a very high frequency over the corpus it is conceivable that these may result in a bottleneck in the Reduce function, since some calls to Reduce may contain an inordinate number of score pair metrics. Accordingly, we may opt to rewrite the Map function to do more processing in order to ensure a more even workload balance. In this alternate version we keep track of each word encountered in the document and accumulate its score metric pairs. This results in only a single emission for each word for each document, rather than potentially very large numbers of emissions as found in our earlier Map implementation. This alternate version of Map is shown below:

```plaintext
38 Map(DocID, Text):
39   (Weight, Score) <- document-score-metrics(Text)
40   WordWeights <- Empty dictionary of key : value pairs
41   WordScores <- Empty dictionary of key : value pairs
42   for each Word in Text:
43     if Word not in [Predefined Scored Words UNION Stop Words]:
44       if WordScores[Word] does not exist:
45         WordScores[Word] <- 0
46         WordWeights[Word] <- 0
49     for each Word in keys(WordScores):
50       Emit(Word, (WordWeights[Word], WordScores[Word]))
```

3. MapReduce Algorithm Design Strategies and Limitations

After presenting several MapReduce examples, a few remarks may be made on algorithm design strategies. The first is that the crucial bridge between the Map and Reduce functions lies in the intermediate keys emitted by the Map function. This intermediate key is the primary data aggregation mechanism of MapReduce. Accordingly, an overarching strategy is to use Map to separate and preprocess data, and also to designate related data as such using the intermediate key. Then, related data is processed via Reduce to obtain the final result. We have also observed another principle to be mindful of from the examples: load balancing. The execution time of the overall MapReduce process can be no shorter than the longest Map or Reduce task. If certain Map or Reduce calls are inordinately long,
this may result in a performance bottleneck. The general mitigation strategy is thus to allocate data processing in the Map and Reduce functions to balance the processing as much as possible based on the data being used.

We also mention certain limitations of MapReduce, which have likely already become noticed to the reader. In the examples we have shown, the parallelism afforded resulted from the independence of most processing steps (e.g. documents and words were not dependent on one another and may be processed in parallel). Many large-scale data processing and analysis tasks possess this property and when it is present, a problem can generally be effectively solved using MapReduce. However, certain types of problems are not amenable to this. In a nutshell, such problems generally require close coordination of processing steps. These include tasks that are inherently recursive, require global algorithm states, or involve using results from earlier stages of computation in later stages (e.g. dynamic programming). Also included are tasks that involve iterative refinement or improvement. Fortunately, however, many if not most Big Data tasks do not involve this type of processing coordination, making MapReduce useful on a wide range of Big Data problems. Finally, it should be mentioned that MapReduce execution necessarily entails some overhead processing. Thus, while large tasks can be handled effectively through parallel decomposition, smaller tasks able to be completed on a single machine sequentially may take longer with MapReduce.

4. Conclusion

In this paper we have provided an overview of MapReduce, a powerful computational paradigm well-suited to many Big Data tasks and enjoying widespread usage today. MapReduce involves implementing an algorithm in terms of two functions, Map and Reduce, and frees algorithm developers from managing the details of parallel computation. We discussed an example from social media analytics, walking through the process of turning a sequential algorithm into one implemented in MapReduce. This also served to illuminate the types of problems for which MapReduce is well-suited, as well as those for which it is not. In general, MapReduce is well-suited to processing tasks where relatively few processing steps depend implicitly or explicitly on one another. As this property is highly pervasive in Big Data problems across many domains, MapReduce is an effective approach for many of today’s Big Data problems. Accordingly, MapReduce forms the backbone for many Big Data applications at forward-leaning companies such as Amazon, Google, Facebook, Twitter, and many others.

References