MapReduce in Distributed Text Processing

By

Sagar Shah

Independent Study Report EECE 8850-02

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

December 15, 2019

Nashville, Tennessee

Submitted to:

_____

Dr. Theodore Bapty, Ph.D.

MapReduce for Distributed Text Processing

Sagar Shah
Electrical and Computer Engineering Department
Vanderbilt University
Nashville, USA
sagar.k.shah@vanderbilt.edu

Abstract

More data means better insights. Taking advantage of this means processing large-scale data. Processing 'big-data' requires distributed computation which brings about its own challenges in the form of scheduling, load balancing, synchronization, fault handling and so on. MapReduce has been the de-facto standard to address these issues. In this study, we will establish the need and requirements for MapReduce, learn about the relevant design patterns and discuss it applications in the real-world. Finally, we will compare the concrete implementation of the MapReduce programming model.

I. INTRODUCTION

The proliferation of five V's, velocity, volume, value, variety and veracity of data in the digital society has brought up tremendous amount of computational challenges. No single machine would suffice the needs of computation. Such challenges are dealt with using a 'big idea' with it being "Scale out, not up"[1]. A commodity low-end servers working together than a single high-end machine is a more efficient approach. MapReduce [2] has for long been the programming model to express such needs of distributed computing. The Hadoop MapReduce[2] is one implementation and Apache Spark[3] is another that extends this model. This report is a study of map reduce, its design patterns and its implementation as the foundation to data-intensive distributed computing. Along with the introduction the report is organized in five sections:

The second section is on the basics of MapReduce. It starts with establishing the need for MapReduce by raising issues that arise at the implementation level of a MapReduce paradigm. It is followed by an explanation of what MapReduce in the functional programming parlance. The next subsection determines the major components of a MapReduce implementation. While the user is isolated from the underlying implementation there are several responsibilities that an execution framework should carry out. This helps in improving the intuition of MapReduce programming. This is succeeded by a discussion of the basic implementation of the MapReduce with the Hadoop Distributed File System (HDFS) as the underlying storage.

The third section is about the scalable approaches for processing huge amounts of text. The third section focuses on arriving at a scalable design pattern by suggesting a pattern and iteratively correcting it by addressing its limitations. As many as four different map reduce design pattern or algorithms are discussed in this section with each of them either building on the previous by addressing some limitation or by proposing a solution for a different type of a problem.

The fourth section looks at a practical application of MapReduce in the form of inverted indexing technique. Inverted indexing is a very useful technique for crawling the web. It proposes two implementations - one a baseline and another an improvement over it, a scalable implementation.

The fifth section is an introduction to a data processing framework in the form of Apache Spark. It discusses key components of the Spark's architecture and then offers a quick overview of its fundamental unit - Resilient Distributed Datasets. The section ends with a comparison between Apache Spark and Hadoop MapReduce which offers insights into both the architecture's strengths and weaknesses.

This report is partly based on work [1] carried out by Dr. Jimmy Lin of University of Waterloo and Dr. Chris Dyer of Carnegie Mellon University and official documentation of Apache Spark [3][4] and Hadoop MapReduce[2].

II. MAPREDUCE

A. Problem

A fundamental approach to deal with large-scale problem is to decompose the problem into smaller pieces and deal with each problem piece by piece. This is synonymous with the divide and conquer approach. At the implementation level, however, there are several issues that needs to be addressed first as described in [1].

- What is the optimal approach to decompose a problem so that they can be executed on parallel nodes?

- How can the distribution of tasks across different machines be decided given different tasks can be suited to different computational resources?
- How to synchronize tasks distributed on different machines?
- How to resolve dependencies i.e., share partial results from one to another?
- How to achieve all of the above during a software or a hardware failure?

MapReduce addresses these problems by isolating the system-level details such as scheduling, synchronization, data starvation and so on from the user.

B. About MapReduce

MapReduce is a programming paradigm that deals with expressing computation of enormous amounts of data distributed in large-scale clusters. It is also an implementation for data processing on a cluster of servers. MapReduce has its roots in functional programming. In functional programming, for a given sequence of values *map* means applying a function f on each of them. In other words, map is a transformation action. *fold* in functional programming takes two arguments - a function g and an initial value. The function g is applied to the initial value and the result is stored in the intermediate variable. This variable and the next item in the sequence are fed as the arguments to the function. This continues until the entire sequence is traversed. The fold operation can be seen as an aggregation action.

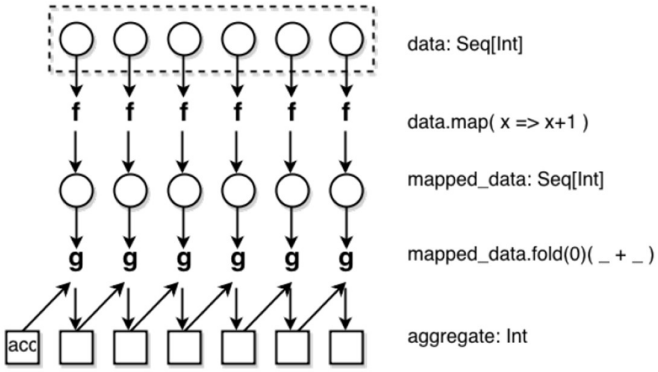

Fig. 1. Map and Fold example [5]

The Map in the MapReduce paradigm roughly corresponds to map Similarly, Reduce corresponds to fold operation. As we can see since a function is applied to all the values in isolation, it becomes a contender of the distribution of such operations across different machines. The fold operation requires at least some items in the list before the operation can be performed. In reality, it is possible to divide sequence into smaller sets to perform fold operation. The signature of map and reduce function are given as follows:

map: (k1, v1) ! [(k2, v2)]

reduce: (k2, [v2]) ! [(k3, v3)]

where [] represent a sequence.

C. Components of MapReduce

As per the design of MapReduce algorithms, some form of key-value structure on arbitrary datasets must be imposed. The four components of MapReduce help in achieving that. The first two, mappers and reducers are mandatory components even if one of them is identity whereas the last two, combiners and partitioners are optional.

Mapper: A mapper is an object that implements the map functionality. A mapper is applied to a sequence of key-value pairs which generates immediate key-value pairs independent of the input ones. For example, in a word count program a mapper would take document id and document value key-value pair as input, tokenize the document and return an intermediate key-value pair of word as the key and value as the integer one indicating the occurrence of a word.

Reducer: A reducer is an object that implements the reduce functionality. A reducer is applied to values related with a given intermediate key generated by the mapper. For example, the intermediate key-value pairs generated by the mapper in the last program the reducer sums up all the counts related with each word and returns a key-value pair with word as the key and count as the value.

It is important to mention that between mapper and reducer there also exists a shuffle and sort functionality which aggregates the values by keys from the intermediate key-values generated by the mapper.

Combiners: Combiners are a sort of an optimization in MapReduce that does a local aggregation before the shuffle and sort phase. For example, in the word count problem a local count for a word (key) processed by the mapper can be aggregated and passed on. This way the number of intermediate key-value pairs are reduced by the count of duplicate keys. The combiners are often known as 'mini-reducers'. Combiners can output any number of key-value pairs, but the type of keys and values must be the same as that of the final output.

Partitioners: Partitioners divide up the intermediate key sets and assign intermediate key-value pairs to the reducers. The partitioner essentially specifies the task to which a key-pair should be copied to. In its simplest implementation, a partitioner hashes the key and takes the mod of the value with the number of reducers which depending upon accuracy of the hash function. This technique may not be the most efficient always as evenly partioning the key set doesn't guarantee the number of key-value pairs sent to the reducers are equal.

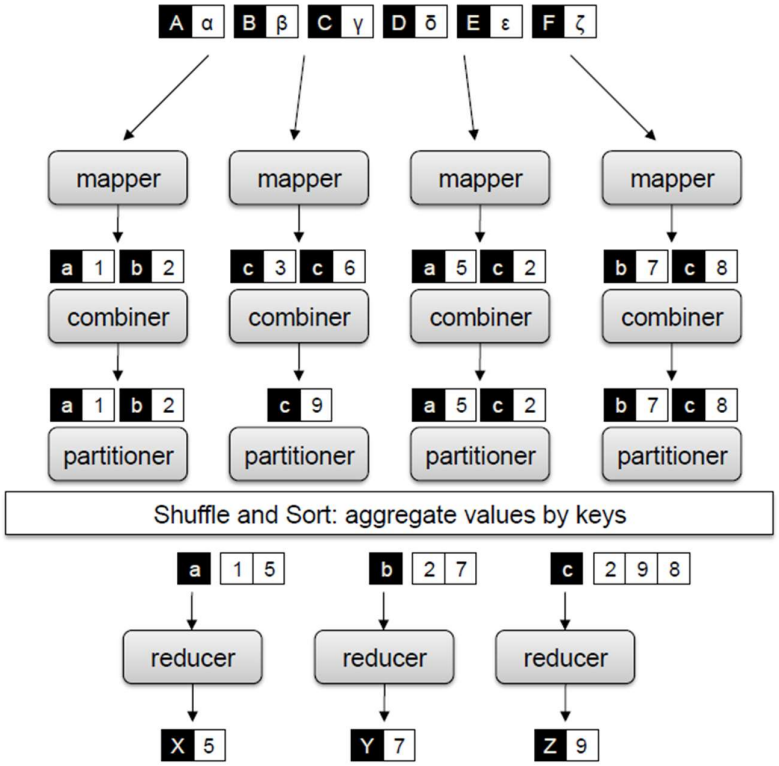

Fig. 2. A complete flow of MapReduce paradigm [1]

D. Responsibilities of execution framework

The user/developer submits a MapReduce program to an execution framework and expects a result without worrying about the underlying implementation details. Internally, the execution framework has several responsibilities that it carries out. Most of them are:

1) Scheduling

1. Often in cases where the number of map/reduce tasks that the cluster can handle is outnumbered by the number of tasks submitted. The role of the execution framework is to maintain a task queue and track the progress of running tasks. It should also be able to assign the waiting task to the node in the cluster as soon as it available.
2. Another aspect of scheduling also involves coordination among tasks belonging to different jobs.

2) Data/Code co-location

For computation, the data needs to be fed to the code. Rather than moving the data to the code, code is moved to the data. In effect, the scheduler starts the task on the node that holds the block of data. If, however, that is

not possible the task is started on a different node and the data is streamed to that location. In that case, the framework ensures that the priority is given to a node in same rack as the one with the data.

3) Synchronization

The synchronization between mappers and reducers is achieved by introducing a barrier between map and reduce phases of processing. A MapReduce job can have up to m x r distinct copy operations since each mapper's intermediate output could be received by a different reducer. The synchronization is achieved by halting or not starting the reduce operation until all the mappers have finished their job as well as shuffling and sorting is done with. An optimization over this could involve copying key-value pairs over the network to the reducers as soon as a mapper is finished.

4) Error and Fault Handling

The execution framework should be able to handle both software and hardware failures.

E. Hadoop Distributed File System

1) HDFS

As storage requirements of a system increases, it is a better choice to distribute the storage capacity rather than increase it. In other words, scaling out is better than scaling up. Such systems are known as Distributed File Systems (DFS). The idea with DFS is to divide the data into large blocks and distribute it across machines. The Hadoop Distributed File System (HDFS) is an implementation of DFS. Hadoop has a master-slave architecture. A master maintains the file namespace information such as metadata, directory structure, file to block mapping, location of blocks, and access permissions. A slave manages actual data blocks. The master is known as namenode whereas the slave is known as datanode. The master or the namenode has several responsibilities such as namespace management, file operations' coordination and maintaining overall health of the system.

- Namespace management: namenode manages metadata, directory structure, file to block mapping, location of blocks, and access permissions. For example, clients requesting read access to a file on a datanode first reaches the namenode. The namenode returns the block id and its location id along with a read access permission.
- Health maintenance: The namenode receives periodic heartbeat messages from the datanodes and checks if they are under-replicated in which case it orders creation of new replicas. It also manages load on the datanodes by moving blocks of data between the nodes
- Coordination: The namenode not only returns the file location to a client based on its request but also allocates blocks on suitable datanodes for write operations. It also lazily reclaims physical storage when a file is deleted.

2) Hadoop Cluster Architecture

As we can see in the Fig. 3. there are three major components: namenode, data node and job submission node. We are already familiar with the first two. The third one, job submission node runs a jobtracker which is the point of contact for clients willing to execute a MapReduce job. It monitors the status of the MapReduce jobs and is responsible for execution of mappers and reducers. The slave machine runs a datanode daemon and a tasktracker which is responsible for running the user code. Let us understand how a Hadoop MapReduce job is executed in the HDFS environment.

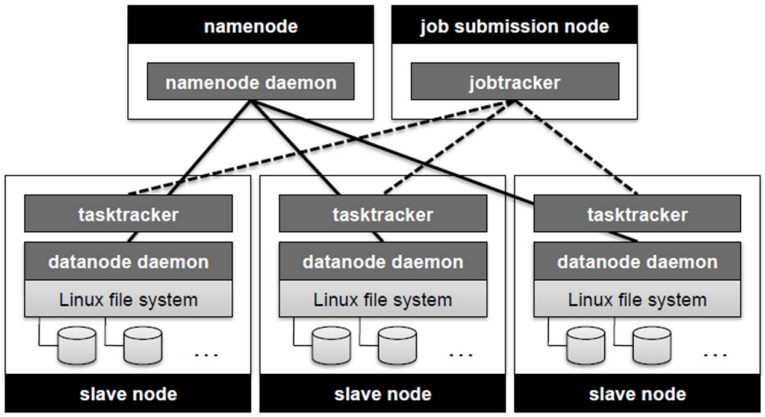

Fig. 3. HDFS Architecture[1]

The MapReduce job is made up of map and reduce tasks. Tasktracker sends heartbeat messages to the jobtracker which in turn responds with task allocation information if it is available. The number of reducers is controlled by the user whereas number of mappers depends upon number of input files and number of HDFS data blocks occupied by the files. The process of feeding key-value pairs to a map task is known as input split. The jobtracker schedules the map tasks such that it as close to the data as possible i.e. in the order of on the node to intra-rack to inter-rack node. Mapper objects' API provides a hook to specify user-defined code before instantiation and after termination. The initialization code allows the state to be preserved across multiple input key-value pairs which is exploited MapReduce design patterns.

III. MAPREDUCE DESIGN PATTERNS

There are several things a programmer cannot control such as where a mapper or reducer runs in the cluster. It cannot constrain reducers and mappers' time limit. It cannot explicitly specify which block of a data is processed by a specific mapper or which intermediate key-value pairs are processed by a specific reducer. However, there are a few things a programmer can control in MapReduce such as

- Specifying code before a map or a reduce task starts and after a map or reduce task ends.
- Construction of complex data structures of keys and values for storing and communicating intermediate results.
- Retaining status in mappers and reducers across multiple inputs or intermediate keys.
- Controlling the order of keys that are received by the reducer.
- Control partitioning of keys thus controlling a set of keys to a particular reducer.

These are the aspects that can be exploited in designing MapReduce algorithms. The following subsections deal with different MapReduce design pattern that can improve scalability of the system and ensure there are less bottlenecks in the system.

A. Local Aggregation

Local aggregation of intermediate results is an important aspect in designing efficient algorithms. This subsection discusses approaches of local aggregation by iteratively improving on a design.

1) Algorithm 1

The first approach is based off the simple word count algorithm described in the previous section where a mapper generates intermediate key-value pair of all unique keys with a count of 1 as their value and reducer then accumulating the result for each key. As in the pseudo code in Fig. 4 we see that an associate array is placed inside the Mapper whose state is preserved across multiple input-key value pairs. The mapper tallies up the count of unique words within a document instead of emitting an intermediate key-value pair for each word it emits the key-value pair for unique word with a count of its appearance.

```
1: class MAPPER
2:     method MAP(docid a, doc d)
3:         H ← new ASSOCIATIVEARRAY
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1              ▷ Tally counts for entire document
6:         for all term t ∈ H do
7:             EMIT(term t, count H{t})
```

Fig. 4. Word count mapper using associative array[1]

2) Algorithm 2

In this approach, the ability to specify user code before a map task starts and after its terminates is leveraged. A mapper's Initialize method is called, which is an API hook for user-specified code, before any input key-value pairs, are fed to it. The initialize method in this approach is used to initialize an associative array which is used to accumulate count of occurrence of words. This is possible because states can be preserved across multiple calls of the Map method. In this case, we initialize an associative array for holding term counts. The key-value pairs are only emitted when the mapper has processed all documents. This is achieved using another API hook known as Close which can be used to execute a termination code. This is essentially like using a combiner to locally aggregate the result. Thus, this pattern is known as 'in-mapper combining'.

```
1: class Mapper
2:     method Initialize
3:         H ← new AssociativeArray
4:     method Map(docid a, doc d)
5:         for all term t ∈ doc d do
6:             H{t} ← H{t} + 1                ▷ Tally counts across documents
7:     method Close
8:         for all term t ∈ H do
9:             Emit(term t, count H{t})
```

Fig. 5. Word count mapper "in-mapper combining"[1]

3) Advantages

There are a couple of advantages in using the "in-mapper combining".

- MapReduce is not explicit about the semantics of a combiner. In fact, Hadoop makes no guarantees about how many times or if at all the combiner is applied or not. Also, combiner is accepted as optimization to the Hadoop framework which may or may not use it. The control is not in the user's hands. While using a combiner after mapper such indeterminacy is unacceptable. With the in-mapper combining local aggregation can be performed without uncertainty.
- While combiners reduce the intermediate data that is passed around in the network, it doesn't reduce the number of key-value pairs emitted by the mapper. It processes the output from the mapper and shortens it but in the first place this generation already has its own overhead. Besides, serialization and deserialization also have their own costs. However, with the in-mapper combining approach the mappers generate lesser number of key-value pairs which in turn also has reduced cost of serialization and deserialization cost.

4) Disadvantages

- Ideally, a functional program doesn't preserve states. In the in-mapper combining, however, that is the case. Besides, preserving states constrains the input key-value pair to be in order which otherwise may lead to ordering bugs. Such problems are hard to troubleshoot on large datasets.
- Scalability is an issue since the mapper must store intermediate results until the mapper has completely processed all the key-value pairs. For a very large dataset the associative array can become extremely large so much that it exceeds the memory impacting the performance of the application.

5) Correctness of Algorithm

The use of combiners should be smart otherwise they can lead to problems. In Hadoop especially, combiners used as optional optimizations and so they cannot be used to determine the correctness of a pattern. In any MapReduce program, the input key-value type of the reducer must match the output type of the mapper key-value pair which means the input and output of the combiner should be of the same type. This constrain will form the basis of how a MapReduce algorithm is designed. Let us take an example that we can run through the design of algorithms. Consider a large dataset with string as input keys and integers as input values. The output should be used to calculate the mean. Now let us walk through different approaches and eliminate or optimize them iteratively.

a) Approach 1

We can have an identity mapper that emits the string and integer input key-value pair as it is. The reducer can then compute the sum and count of the integers associated with a key which can be used to emit the string and the average value as the output. The pseudo code of the algorithm is the following.

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, integer r)

1: class Reducer
2:     method Reduce(string t, integers [r1, r2, ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all integer r ∈ integers [r1, r2, ...] do
6:             sum ← sum + r
7:             cnt ← cnt + 1
8:         r_avg ← sum/cnt
9:         Emit(string t, integer r_avg)
```

Fig. 6. Mapper to compute mean of values related to a key[1]

The problem with above algorithm is that like before, the shuffling of the key-value pairs from mappers to reduce would be highly inefficient. Using a combiner reduces shuffling but yields incorrect results.

b) Approach 2

In this approach a combiner is used with a slight modification. The combiner has string-integer as its input key-value from which it calculates the sum of encountered integers and the number of them. With the string as the key and a pair of sums and count it emits the output key-value. This approach looks correct, however, like mentioned earlier the type input and the output of the combiner be the same which is not the case here. The correctness of the algorithm dependent upon the number of times the combiner runs – in this case it is one. The pseudo code of the algorithm is as follows:

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, integer r)
1: class Combiner
2:     method Combine(string t, integers [r_1, r_2, ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all integer r ∈ integers [r_1, r_2, ...] do
6:             sum ← sum + r
7:             cnt ← cnt + 1
8:         Emit(string t, pair (sum, cnt))            ▷ Separate sum and count
1: class Reducer
2:     method Reduce(string t, pairs [(s_1, c_1), (s_2, c_2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s_1, c_1), (s_2, c_2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         r_avg ← sum/cnt
9:         Emit(string t, integer r_avg)
```

Fig. 7. Algorithm to compute mean of values related to a key using combiner[1]

c) Approach 3

This approach builds on the last approach with a slight modification to the mapper. Now, the mapper emits a pair of the integer and 1 as the value alongside the string as key. The combiner then accepts the input, accumulates the partial sums and partial counts and emits them as a pair. The reducer again accepts those input, calculates the sum of the partial sums and partial counts and then at the end computes the mean of specific to that string. It emits string as the key and mean as the output. The pseudo code is as follows:

```
1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, pair (r, 1))
1: class Combiner
2:     method Combine(string t, pairs [(s_1, c_1), (s_2, c_2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s_1, c_1), (s_2, c_2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         Emit(string t, pair (sum, cnt))
1: class Reducer
2:     method Reduce(string t, pairs [(s_1, c_1), (s_2, c_2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s_1, c_1), (s_2, c_2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         r_avg ← sum/cnt
9:         Emit(string t, integer r_avg)
```

Fig. 8. Algorithm with modfied mapper to compute mean of values related to a key using combiner [1]

The algorithm can be verified without running the combiners in which case the output of the mapper would be fed directly to the reducers (obviously after shuffle and sort phase). Since the reducer is like combiner it would deal

with the input in the same way except there would be more key-value pairs in the input. The mean would be calculated at the end by dividing the sum with the count. Hence, this algorithm should work correctly. Let us consider one more approach that exploits algorithm 2.

d) Approach 4

In this approach an associative array as in algorithm 2 is created which is used to store the partial sums and counts related with the string in the memory. The rest sequence of executions is all same. This approach has the same advantages and disadvantages of the algorithm 2.

B. Pairs and Stripes

The design of pairs and stripes algorithm is built on the work carried out in [6][7]. To explain this approach let us take an example of building word co-occurrence matrix from a corpus which would be used throughout the description. It is a n x n square matrix where n is the number of unique words. For a given context, such as a sentence, paragraph, a document or some number of words, a cell m_{ij} in square matrix represents the number of times a word w_i co-occurs with word w_j. We know that the computation of a word co-occurrence matrix can become very slow if it is too big to fit in a memory as it is then paged to disk. The pairs and stripes approach can deal with this problem. [1]

1) Pairs

Let us look at the pseudo-code in Fig. 9.

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term w ∈ doc d do
4:             for all term u ∈ Neighbors(w) do
5:                 Emit(pair (w, u), count 1)              ▷ Emit count for each
   co-occurrence
1: class Reducer
2:     method Reduce(pair p, counts [c1, c2, ...])
3:         s ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             s ← s + c                                   ▷ Sum co-occurrence counts
6:         Emit(pair p, count s)
```

Fig. 9. Calculation of word co-occurrence using pairs approach[1]

For each unique word in a document a count of the word for a neighbor is emitted out by the mapper. In other words, a pair of a unique word with a neighbor as key and count of 1 as value is emitted out by the mapper. The reducer then for every such unique pair accumulates the count of their appearance. Thus with the unique pair of word and neighbor as key and their number of cooccurrences as the value are emitted as final-key value pair.

2) Stripes

In this algorithm, four each unique word an associative array is initialized. The associative array tallies the words that co-occur with the word. To elaborate, for each unique word a count of occurrence of a unique neighbour with that unique word is calculated. The word as the key and the associative array as the value is emitted out as the intermediate key-value pair. This associative array is known as stripe. The reducer element-wise sums all the associative arrays with the same key, accumulating counts that correspond to the same cell in the co-occurrence matrix. A final key-value is of the form (word: associative array). The pseudo code as follows:

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term w ∈ doc d do
4:             H ← new AssociativeArray
5:             for all term u ∈ Neighbors(w) do
6:                 H{u} ← H{u} + 1          ▷ Tally words co-occurring with w
7:             Emit(Term w, Stripe H)
1: class Reducer
2:     method Reduce(term w, stripes [H1, H2, H3, ...])
3:         Hf ← new AssociativeArray
4:         for all stripe H ∈ stripes [H1, H2, H3, ...] do
5:             Sum(Hf, H)                              ▷ Element-wise sum
6:         Emit(term w, stripe Hf)
```

Fig. 10. Calculation of word co-occurrence using stripe approach[1]

3) Comparison between Pairs and Stripes

- On the basis of intermediate key-value pairs, the number is much lower with the stripes approach than the pairs approach since in the stripes approach the first element of the pair is repeated for every co-occurring word pair. However, the serialization and deserialization overhead is higher with the stripes approach than the pairs approach as stripes uses associative arrays which is a complex data structure.
- In term of scalability, the stripes approach assumes that any point in time the associative array is small enough to fit into memory. The size of it is dependent upon the number of unique words in the corpus which is unbounded according to Heap's Law[8]. Therefore for a very large corpora (~ PBs) this can become a bottleneck. The pairs approach doesn't need intermediate data in memory and has no such limitation.

In an experiment performed by Jimmy Lin and Chris Dyer the stripes approach was found faster than the pairs approach.[6][7]

C. Order inversion

For the calculation of the word co-occurrence matrix we have used absolute counts. However, some words are more frequent than the other. This is not taken into consideration with the absolute counts. The solution to this problem is using relative frequencies. Relative frequency is the count of the joint event divided by a marginal, where a marginal is the sum of the counts of a given word co-occurring with everything else. It is given by the following equation:

$$(w_j|w_i) = \frac{N(w_i, w_j)}{\sum_{w'} N(w_i, w')}$$

The computation of relative frequencies can be approached using stripes as well as pairs.

With the stripes approach, counts of all words that co-occur with a given word are available in the associative array. The sum of all the counts gives the marginal and dividing all the joint counts by the marginal returns the frequency of all the words.

Using the pairs approach requires more modification. The reducer can preserve states across multiple input key-value pairs. Inside the reducer object the count of words co-occurring with a particular word can be stored in the memory which is equivalent to building an associative array in the stripes approach. Before feeding to the reducer, the pair is sorted by the first element and then the second word. With this ordering, it can be detected if pairs associated with a word have been encountered. It is important that a key-value pair of the format for any (word, *) are all fed to the same reducer. This requires a custom partitioner that partitions pairs based on the first element of a pair. This approach however suffers the drawbacks of having a large corpus. This basic approach can be iterated over to reduce the drawbacks.

In the next approach, the mapper is modified so that it emits an additional key also known as the "special key" of the form $(w_i, *)$, with 1 as the value which stands for the word pair's contribution to the marginal. The partial marginal counts are aggregated before passing to the reducers using the in-mapper combiner technique. In the reducer part, the special keys are processed before the other key-value pairs representing the joint counts. To illustrate this, let us consider an example. The following is a sequence of key-value pairs that the reducer can encounter.

key	values	
(dog, ∗)	[6327, 8514, ...]	compute marginal: $\sum_{w'} N(\text{dog}, w') = 42908$
(dog, aardvark)	[2,1]	$f(\text{aardvark}\|\text{dog}) = 3/42908$
(dog, aardwolf)	[1]	$f(\text{aardwolf}\|\text{dog}) = 1/42908$
...		
(dog, zebra)	[2,1,1,1]	$f(\text{zebra}\|\text{dog}) = 5/42908$
(doge, ∗)	[682, ...]	compute marginal: $\sum_{w'} N(\text{doge}, w') = 1267$

Fig. 11. Example sequence of key-value pairs for the reducer[1]

When the reducer receives a special key along with a list of partial marginal contribution from the map phase, it accumulates them to find the marginal. When a normal key is received with a list of partial joint counts from the map phase, the reducer accumulates them to find the final joint count. Total joint counts divided by the marginal returns the relative frequency. When a special key is encountered again, the accumulation of marginal contribution restarts from 0. Thus, this algorithm has minimal memory requirement. Since, no individual co-occurring counts is

required, the scalability bottleneck is eliminated from the algorithm. This design pattern is also known as 'order inversion'. This is so because with correct coordination, the result of a computation can be accessed in the reducer before processing the data needed for that computation.

D. Value to key conversion (Secondary Sorting)

The motivation to use this algorithm comes from situations where not only sorting by key but sorting by value is also required. While some implementations [9] provide an in-built functionality others don't [10]. Consider a situation where many sensors are sending the data with timestamps and readings. In order to capture the sensor activity an intermediate key might look like:

Sensor ID → (Timestamp, Reading)

However, since MapReduce doesn't guarantee the order of values related to a key, the timestamps of the reading won't be sorted. Buffering the data is an option but it doesn't scale well. The solution to this is using a composite key where timestamp is moved out of the value and placed alongside a key in the following format:

(Sensor ID, Timestamp) → Reading

Sorting the intermediate keys by the first element i.e. sensor id and then by timestamp should be easy now. This requires implementing a custom partitioner though. Implementing a secondary sorter in the reducer is faster but isn't scalable. This approach is slightly slower but is scalable. Besides, implementing ternary, quaternary and so on sorters is relatively easier with this approach.

IV. INVERTED INDEXING FOR TEXT PROCESSING WITH MAPREDUCE

One of the most important requirements for processing text is in Web search. In fact, Web search is a classic big-data problem. A short query requires results with a latency in a few hundredths of milliseconds. Search engines use a data-structure called inverted index which for a given term provides access to the list of term containing documents. Documents can be web pages, pdfs, code fragments or any other web object. A web search is composed of three steps: web content gathering also known as crawling, inverted indexing and document ranking also known as query retrieval. MapReduce offers the most suitable solution for inverted indexing.

1) Inverted index

A data structure that stores a map of words or numbers to its locations in a document or a set of documents is known as an inverted index. For example, this is how inverted index might look

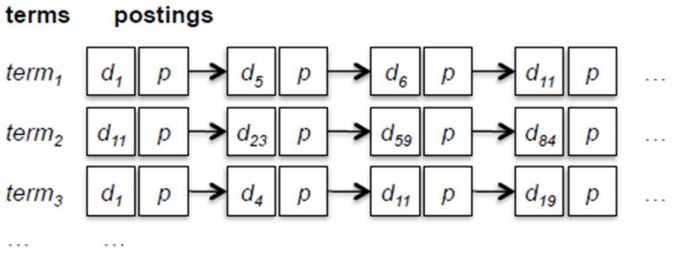

Fig. 12. Inverted example index[1]

From the example, it can also be described as a postings list comprising of postings, each of which consists of a document id and a payload such as frequency of the term in a document. Given a query, retrieval involves fetching postings lists associated with query terms and traverses the postings to compute the result set. For example, a Boolean retrieval requires operations such as OR and AND on postings lists, can be achieved in an efficient manner since the postings are sorted by document id.

2) Implementation 1

In this approach, documents are pre-processed in which things like HTML tags removal, tokenization, case folding and stop word removal are carried out. Input to the mapper consists of document ids (keys) paired with the actual content (values). Post this, the frequency of unique term is calculated and stored in an associative array. The pseudo code for the following process is as follows:

```
1: class Mapper
2:     procedure Map(docid n, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             Emit(term t, posting ⟨n, H{t}⟩)
1: class Reducer
2:     procedure Reduce(term t, postings [⟨n1, f1⟩, ⟨n2, f2⟩ ...])
3:         P ← new List
4:         for all posting ⟨a, f⟩ ∈ postings [⟨n1, f1⟩, ⟨n2, f2⟩ ...] do
5:             Append(P, ⟨a, f⟩)
6:         Sort(P)
7:         Emit(term t, postings P)
```

Fig. 13. Inverted indexing pseudo code – Implementation 1 [1]

Once the processing is finished, for all terms a term as a key and a posting as a pair of document id and the associative array are emitted out. This is followed by the shuffle and sort phase in which the execution framework collects all the postings that are related with a term and sorts them for each term. So now the input to the reducer is a term as a key and all associated postings as a value. On the reducer's front, it maintains a new list for each term and add all the corresponding postings to the list. After adding them it sorts the list in-memory by their document id and emits the entire postings list as the value and the term as the key.

However, this approach also suffers from the same scalability issue that we have been discussing in section III. The approach doesn't account for the fact that there may not be sufficient memory to store all the postings associated with a term. Besides, the sorting in-memory also has the same limitation. The next approach tries to address this problem.

3) Implementation 2

In this approach, the onus of sorting the postings is laid on the MapReduce framework. Intermediate key-value pairs of the type (tuple <*t, docid*>, tf f) are emitted. A tuple of the term and its document id becomes the key while the term frequency becomes the value. This is similar to the value to key pattern encountered in the second section. The pseudo code for this algorithm is as follows:

```
1: class Mapper
2:     method Map(docid n, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             Emit(tuple ⟨t, n⟩, tf H{t})
1: class Reducer
2:     method Initialize
3:         t_prev ← ∅
4:         P ← new PostingsList
5:     method Reduce(tuple ⟨t, n⟩, tf [f])
6:         if t ≠ t_prev ∧ t_prev ≠ ∅ then
7:             Emit(term t_prev, postings P)
8:             P.Reset()
9:         P.Add(⟨n, f⟩)
10:        t_prev ← t
11:    method Close
12:        Emit(term t, postings P)
```

Fig. 14. Inverted indexing pseudo code – Implementation 2 [1]

It should be noted that this pattern would require defining a custom partitioner. The custom partitioner ensures that all the tuples with the same term are shuffled to the same reducer. The fact that state could be preserved in a reducer across multiple keys, means that the posting lists can be created with minimal memory usage. On the reducer part, an initialize method is called that maintains a new postings list for sharing all the postings while running the reduce method. For each key <t, n> there is only one value associated with each key. For each key-value pair , the postings are directly added to the list. The postings are guaranteed to arrive in the order sorted by document id. Once the last

posting associated with a term is processed the shared posting list is written to the disk. This is accomplished by calling the close method which emits the term as key and posting as value.

V. INTRODUCTION TO SPARK

A. What is Spark?

Apache Spark is a collection of services to facilitate distributed data processing.[4] Spark is an extension of the MapReduce model with support for interactive queries and real-time stream processing.

- In terms of performance, Spark's ability to run computations in the memory makes it unique and that is one of its main features.
- Spark as mentioned provides support for a variety of workloads including batch processing, iterative computation, interactive querying and real-time streaming.

Apache spark is a combination of closely integrated yet distributed components. Spark is responsible for scheduling, distributing and managing workloads on the slave machines in its own cluster or to another spark unit managing another cluster.

B. Spark Components

1) Spark Core

The Spark Core is the foundation of Apache Spark. It provides and manages components responsible for task scheduling, memory management, fault recovery and storage interactions. It contains the main programming abstraction of RDDs (resilient distributed datasets) which are explained in details later. RDD is a collection of items distributed on different machines and can be manipulated in parallel.

2) Spark SQL

Spark SQL is used to work with structured data. It can be queried using traditional SQL or Apache's Hive Query Language (HiveQL). Alongside SQL, this component also supports regular data manipulations on RDDs that is available in the spark context. This allows complex analytics to be run within an application.

3) Spark Streaming

Spark streaming originally designed for providing fault tolerance, throughput and scalability to the Spark Core supports stream processing of real time data. For example, continuous log of web pages can be manipulated to perform useful transformations. It makes it easy to move between applications that act upon data stored in memory, on disk or the one arriving in real-time.

4) MLib

MLLib is a library that contains machine learning algorithms for classification, regression, clustering and collaborative filtering. It also supports functionalities such as model evaluation and data import alongside a generic gradient descent optimization algorithm.

5) GraphX

GraphX is a library used for graph processing. With this library, graphs can be created, vertices and edges be attached and algorithms be applied on them.

6) Cluster Managers

Spark comes with its own cluster manager in the form of Standalone schedule. Besides, spark has support for cluster managers such as Hadoop YARN or Mesos.

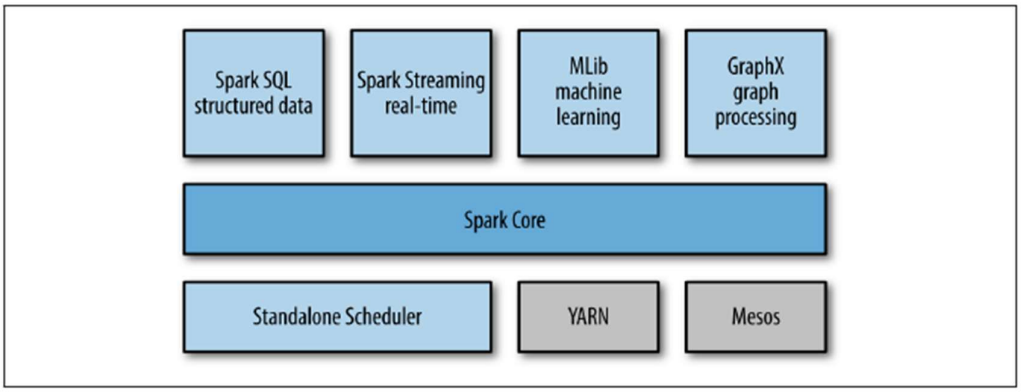

Fig. 15. The Spark Stack [4]

C. Resilient Distributed Datasets

A resilient distributed data (RDD) is an immutable distributed collection of objects [7]. A formal definition given by its creators [3] is as follows: *RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.*

An RDD is internally split into multiple partitions and can be computed on different nodes. Before any RDD is created a Spark context has to be created which is the starting point for any Spark application. The services of Spark's execution engine can only be availed after creating the Spark Context. There are two major operations that can be defined for any RDD:

- Transformation

To transform is to operate on an RDD and return a new RDD. Transformation uses lazy evaluation i.e. this operation is not performed until an action is called. Most transformations are element wise i.e. they operate one element at a time.

Map and filter are the two most common transformations performed on an RDD. The map() transformation applies a function to every element in the RDD and returns a new RDD consisting of the value of all elements in the resulting RDD. The filter() transform takes a function as an argument and returns a new RDD consisting of only those values that matched the filtering criteria. Few common transformations are listed below:

Function name	Purpose	Example	Result
`map()`	Apply a function to each element in the RDD and return an RDD of the result.	`rdd.map(x => x + 1)`	`{2, 3, 4, 4}`
`flatMap()`	Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words.	`rdd.flatMap(x => x.to(3))`	`{1, 2, 3, 2, 3, 3, 3}`
`filter()`	Return an RDD consisting of only elements that pass the condition passed to `filter()`.	`rdd.filter(x => x != 1)`	`{2, 3, 3}`
`distinct()`	Remove duplicates.	`rdd.distinct()`	`{1, 2, 3}`
`sample(withReplacement, fraction, [seed])`	Sample an RDD, with or without replacement.	`rdd.sample(false, 0.5)`	Nondeterministic

Fig. 16. Transformations with Spark [4]

- Action

The most common action transformation is reduce(). Reduce transformation takes two functions of a type from the RDD and operates on them returning an intermediate RDD which is then supplied to the same function along with the next in the function until all the elements in the RDD have been traversed. Following is the list of actions available in Apache Spark:

Function name	Purpose	Example	Result
`take(num)`	Return num elements from the RDD.	`rdd.take(2)`	`{1, 2}`
`top(num)`	Return the top num elements the RDD.	`rdd.top(2)`	`{3, 3}`
`takeOrdered(num)(ordering)`	Return num elements based on provided ordering.	`rdd.takeOrdered(2)(myOrdering)`	`{3, 3}`
`takeSample(withReplacement, num, [seed])`	Return num elements at random.	`rdd.takeSample(false, 1)`	Nondeterministic
`reduce(func)`	Combine the elements of the RDD together in parallel (e.g., sum).	`rdd.reduce((x, y) => x + y)`	9
`fold(zero)(func)`	Same as `reduce()` but with the provided zero value.	`rdd.fold(0)((x, y) => x + y)`	9
`aggregate(zeroValue)(seqOp, combOp)`	Similar to `reduce()` but used to return a different type.	`rdd.aggregate((0, 0))((x, y) => (x._1 + y, x._2 + 1), (x, y) => (x._1 + y._1, x._2 + y._2))`	(9, 4)
`foreach(func)`	Apply the provided function to each element of the RDD.	`rdd.foreach(func)`	Nothing

Fig. 17. Action with Spark [7]

D. Comparison between Apache Spark and Hadoop MapReduce

The following table offers a qualitative comparison between two data processing frameworks – Apache Spark and Hadoop MapReduce.

Sr. No.	Parameters	Apache Spark	Hadoop MapReduce
1	Workload variety	Suitable for iterative as well as batch processing	Suitable for batch processing
2	Speed performance	Much faster than MapReduce while performing operation on data in-memory. Slightly faster than MapReduce for disk operations	Has higher latency because of disk input-output operation. Also because of greater network traffic
3	Scalability	Scalable up to 1000 nodes in single cluster	Scalable up to 1000 nodes in single cluster
4	Fault Tolerance	Has intrinsic support for fault tolerance in the form of RDDs	Replication
5	Architectural complexity	Has several components that make the system work. However, that is abstracted from the user and hence is not a concern	Has fewer components to manage.
6	Memory requirements	Larger RAM. Costlier than MapReduce	Smaller than used in spark clusters.
7	Hardware	Mid-to-high level hardware requirement	Works with commodity hardware
8	Scheduler	Comes with a standalone scheduler along with support for other schedulers such as Hadoop YARN and Mesos.	No built-in support. External integration.

9	Compatibility	Compatible with file formats supported by HDFS cluster.	Compatible with all data sources and file formats
10	Querying	Support SQL as well HiveQL	Hive Query Language can be applied
11	Ease of use	Easy to write and debug	Difficult to write code
12	Code verbosity	Needs fewer number of lines of code	More number of lines to code for a task.
13	Machine Learning Integration	Has built-in support	Can be integrated with Apache Mahout
14	Interactive Mode	Has interactive mode	Doesn't have any interactive mode
15	Language Support	Supports Java, Scala, Python and R	Written in Java. Support available in C++, Ruby, Python, Perl and Groovy

Table 1: Comparison between Apache Spark and Hadoop MapReduce [11]

VI. CONCLUSION

MapReduce, the programming model remains fundamental to implementation of data processing frameworks. This report discusses the role of MapReduce in data-intensive distributed computing. We see several approaches in coming up with a scalable design pattern. As many as four algorithms were iteratively worked through to address different problems. Local aggregation is a good technique to address word count and mean value count problem whereas pairs and stripes is suitable for word co-occurrence matrix. Order inversion is a natural extension of pairs and stripes problem in the context of relative frequency computation. Secondary sorting algorithm turns out to be scalable for ternary and even quaternary sorting. These algorithms become useful in implementing inverted indexing for web search. The solutions were all achievable with the Hadoop's MapReduce implementation but finally we see how Apache Spark is a natural successor to MapReduce. Along with its core engine it offers several functionalities such as SQL querying, graph processing, machine learning algorithms' library, all, built-in. The resilient distributed datasets becomes its core abstraction which reduces the latency by up to 5 times [12]. Besides writing code becomes much simpler with Apache Spark than with Hadoop MapReduce. PageRank uses graph analysis to measure web-page quality based on web search. It would have been interesting to explore MapReduce algorithms in the context of graph processing and structured querying.

ACKNOWLEDGEMENT

This work is carried out under guidance of Dr. Theodore Bapty at Vanderbilt University. I am extremely grateful to you, Dr. Bapty for your patience and encouragement.

REFERENCES

[1] J. Lin and C. Dyer, *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool Publishers, 2010.

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, 2008.

[3] M. Zaharia *et al.*, "Apache Spark: A Unified Engine for Big Data Processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016.

[4] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, *Learning Spark Lightning-Fast Data Analysis*, First. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472., 2015.

[5] P. Michiardi, "Scalable Algorithm Design."

[6] C. Dyer, a Cordova, a Mont, and J. Lin, "Fast, easy, and cheap: Construction of statistical machine translation models with MapReduce," *Comput. Linguist.*, 2008.

[7] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, 2002.

[8] N. Alon, Y. Matias, and M. Szegedy, "The space complexity of approximating the frequency moments," *J. Comput. Syst. Sci.*, 1999.

[9] S. Ghemawat, H. Gobioff, and S. T. Leung, "The google file system," in *Operating Systems Review (ACM)*, 2003.

[10] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–10.

[11] "MapReduce vs Apache Spark- 20 Useful Comparisons To Learn." [Online]. Available: https://www.educba.com/mapreduce-vs-apache-spark/. [Accessed: 15-Dec-2019].

[12] J. Shi *et al.*, "Clash of the titans: Mapreduce vs. spark for large scale data analytics," in *Proceedings of the VLDB Endowment*, 2015.