MapReduce for Beginners

Dr. Bushra Anjum

Technical Lead, Amazon Prime

Amazon.com, Inc.

Relevant PDC Topics

- Algorithmic Paradigm: MapReduce framework (C)
- Architecture: Cluster computing (K)
- Distributed Algorithms (A)
- Scalability in algorithm (A)
- Programming: Parallel Programming (A)
- Algorithm: Scalability in algorithms and architectures (C)

Learning Outcomes

After finishing this chapter, the student will be able to:

1. Recognize the growing need for scalable and distributed data processing solutions and how this need is addressed by the MapReduce paradigm.
2. Describe the MapReduce programming abstraction and runtime environment.
3. Analyze the strengths and limitations of the MapReduce platform.
4. Define the problem characteristics that make it a candidate for a MapReduce solution.
5. Evaluate a given problem for its suitability to be solved using a MapReduce approach.
6. Transform a candidate problem into the map and reduce computing phases.
7. Outline the growing system of open source components, around the MapReduce framework, for large-scale data processing.

Table of Contents

Table of Figures

Background and Introduction

Professor Patrick Wolfe, executive director of the University College of London's Big Data Institute, has recently said in an interview with Business Insider [1], "The rate at which we're generating data is rapidly outpacing our ability to analyze it. The trick here is to turn these massive data streams from a liability into a strength."

Indeed the digital data has been growing at an exponential rate, doubling every two years, and it is predicted [2] that by 2020 the digital universe will contain nearly as many bits as there are stars in the physical universe!

The increase of data opens up huge learning and analysis opportunities. However, it also comes with its unique set of challenge, e.g., How to store the data? How to process it economically in an acceptable amount of time? The complexity is not limited to the enormous size of the data, but also includes other dimensions such as the speed at which the data is produced, the various formats it comes in, etc. The popular term used to describe such datasets is "Big Data". Big Data is defined by the 'three Vs' of data: volume, variety, and velocity [3]. *Volume* refers to the size of the data, *Variety* refers to the fact that the data is often coming from a variety of different sources and in many different formats, and *Velocity* relates to the speed at which the data is being generated. Storing and processing these ever growing datasets require *scalable* storage and computing solutions. Let's define scalability next.

Scalability has been classically defined as the ability to process data even when it is larger than the available capacity of a server machine. One way to achieve this is to process the dataset piece by piece, i.e., to take the first piece of data, operate on it, write the results back then take the next piece of data, operate on it, write the results back and so on.

As an example, let's assume that we are given an extensive weather dataset consisting of millions of temperature, humidity, and air pressure readings. We would like to calculate the average temperature, average humidity, and average air pressure values over all the records in the dataset. Since the dataset is large and cannot fit inside a single server, we will analyze it by first breaking it down into smaller blocks and then working on it one block at a time. So we bring the first block of data in and calculate the three statistics (average temperature, average humidity, and average air pressure) on it. Then we bring the second block and calculate and update the three statistics to reflect the data present in both blocks. Then bring the third block, and so on. After bringing in and processing the last block, we will have the average temperature, average humidity, and average air pressure statistics calculated over the entire dataset. This approach is shown in Figure 1.

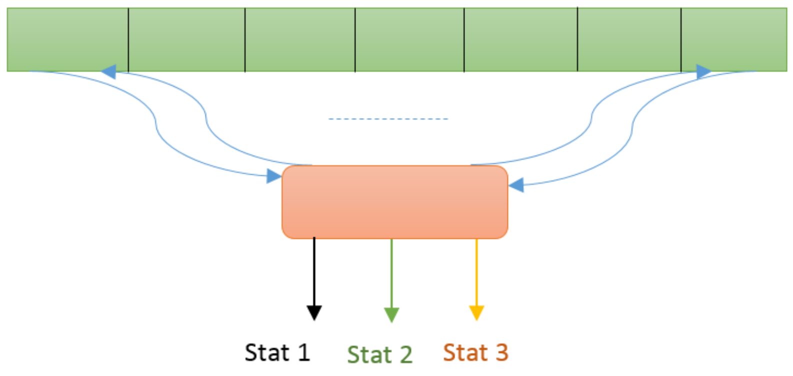

Figure 1: Distributed Processing without Parallelism

This block by block processing will give us the required results. Even so, a potential problem with this method is that as the size of the dataset increases, so does the overall processing time, as this approach is **serial** in nature. One way to scale the solution is to make this server machine better with a faster processor, more memory, etc. This approach is known as "**Scaling up**" (or vertical scaling), and it takes place through an improvement in the specification of a resource (e.g., upgrading a server with more main memory, larger hard drive, or a faster CPU, etc). Still, there are limits to the upgrades that can be done to a computer system governed by the law of diminishing returns, price and power considerations, limiting Input/Output latencies, etc.

The alternate approach is to **parallelize** the solution. With this approach we could use multiple servers where each server works on an individual block of data, while all of them operate in parallel. This approach is called "**Scaling out**" (or horizontal scaling), and it takes place through an increase in the number of resources available to solve a problem (e.g., adding more hard drives to a storage system or adding more servers to support an application, etc).

Scaling out is an excellent way to build Big Data applications, as it allows distribution of workloads to multiple servers operating in parallel. Thus by scaling out, we can use hundreds or even thousands of commodity servers and apply them all to the same problem. Such a collection of server machines, connected via hardware, networks, and software, and working in parallel on the same problem, is called a cluster.

Scaling out and parallelizing the solution is an attractive option, however, as a result of processing different data blocks on different server machines, now we have results distributed all over the cluster. In the context of our example, this means that we have an average temperature, average humidity, and average air pressure reading calculated over the first block on one server, an average temperature, average humidity, and average air pressure reading calculated over the second block on another server and so on. Whereas, we are interested in consolidated results calculated over the entire dataset. This situation is depicted in Figure 2 below.

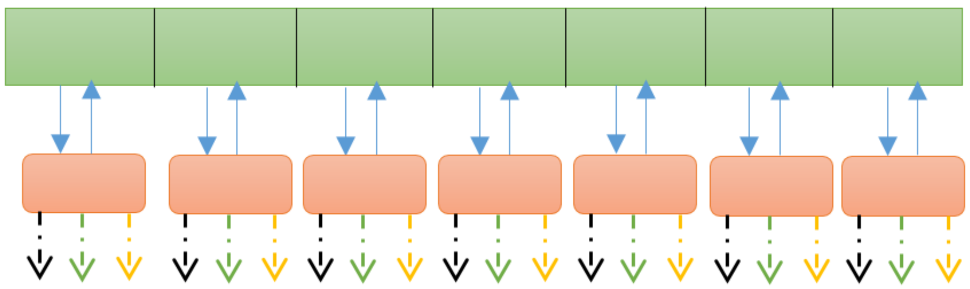

Figure 2: Distributed Processing with Parallelism

Here is an interesting insight though, the key insight that led to the development of the MapReduce paradigm. We already have individual statistics calculated per block of data. Now, we can use more servers to parallelize the consolidation task too! That is, take all those average temperatures, average humidity, and average air pressure statistics calculated per block and combine them on additional server machines, in parallel if possible. This added step is demonstrated in Figure 3 below.

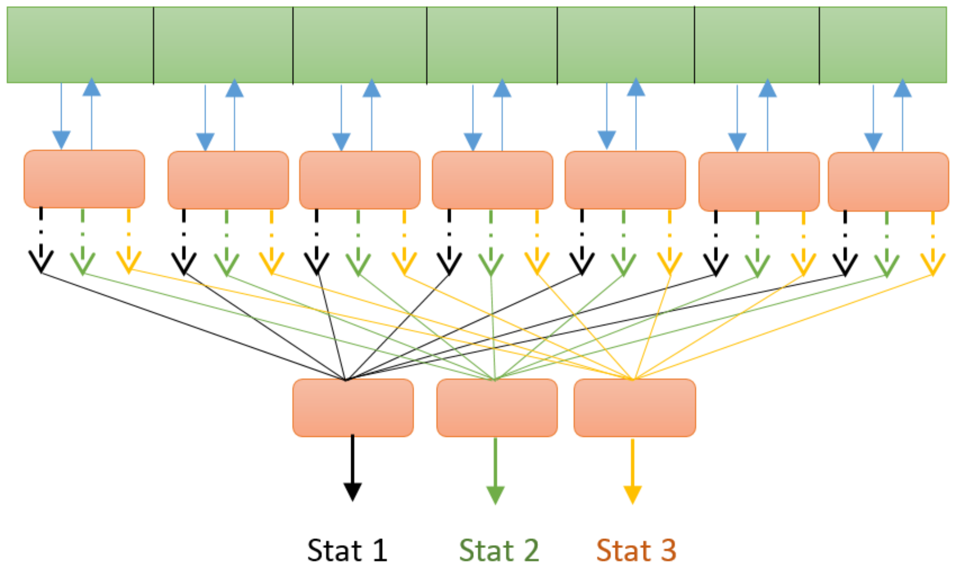

Figure 3: Multi-Stage Distributed Processing with Parallelism

Hence scaling out is part of the solution, the other part is having distributed algorithms that can run on these clusters and produce desired results. *MapReduce* is one category of such distributed algorithms. It started when Google developed and published on Google File System (GFS), a scalable and distributed data storage solution in 2003 to store the large corpus of web crawling data [4]. Then in 2004 Google further presented the MapReduce framework to help search and find the insights from the data stored in the GFS [5]. Inspired by Google's proprietary GFS and MapReduce, their open source equivalents were developed by the Apache Software Foundation and became the Hadoop project [6]. Much like Google's MapReduce is layered on top of GFS, in Hadoop, MapReduce framework is layered on top of Hadoop Distributed File System (HDFS), a distributed fault tolerant storage facility. Hadoop and its variants are currently in use at Yahoo!, Facebook, Amazon and Google-IBM NSF clusters, to name a few.

We have almost described the MapReduce paradigm here. Let's look at it formally in the next section.

MapReduce

MapReduce is a scalable distributed data processing solution that works in collaboration with a massively scalable distributed file system, such as HDFS. HDFS is responsible for taking large datasets, dividing them into smaller blocks and storing them on individual nodes of a cluster while providing additional services like availability, fault tolerance, replication, persistence, etc. MapReduce framework, which is layered on top of HDFS, is responsible for bringing the computation to the data stored in these nodes and running it in parallel.

MapReduce consists of *two separate and distinct computation phases,* the Map phase and the Reduce phase. During the first phase, the framework runs a map function (also called a mapper) in parallel on the entire dataset stored in the HDFS. In the second phase, the framework runs a reduce function (also known as a reducer) on all the data produced by the mappers during the Map phase. The output from the reducers, which is the final result of the data processing job, is written back to the HDFS cluster. As the sequence of the name MapReduce implies, the reduce task is always performed after the map task is finished.

A map function is executed in parallel on each node in the HDFS cluster that is storing a block of the input data. The mapper reads the block of data one record or one line at a time, depending on the type of data. The data is read record by record if it is structured in nature such as originating from a database, or it may be read line by line, if the data is unstructured such as log files, social media streams, etc. The mapper then processes this record or line of data and outputs the results of its processing in a specialized format. The format is essentially a list consisting of key-value pairs, i.e., {(key_1, $value_1$), (key_2, $value_2$), … , (key_n, $value_n$)}. For example, for our weather dataset, the mapper may emit the following three key-value pairs for each record of input read: **{('temperature': value_of_temperature_in_current_record), ('humidity': value_of_humidity_in_current_record), ('air pressure': value_of_airpressure_in_current_record)}**. Then begins the Reduce phase. Here several reducers work in parallel, each taking as input a 'subset' of key-value pairs produced by the map function and combining those into a final result. For weather data example, three reducers may work in parallel. The first reducer may combine all the key-value pairs that have the key 'temperature' into a single result **('temperature': avg_value_of_temperature_of_all_records).** The second reducer may combine all the key-value pairs with the key 'humidity' into a single result **(humidity: avg_value_of_humidity_of_all_records),** and a third reducer may combine all the key-value pairs with the key 'air pressure' into a single result **(air pressure: avg_value_of_airpressure_of_all_records)**. If the reader recalls, this is what we suggested in Figure 3 above.

In general, a map and a reduce function can be defined by the following mappings:

$$map:\ value_{input} \rightarrow (key_{output}, value_{intermediate})$$

$$reduce: (key_{output}, \{list(value_{intermediate})\}) \rightarrow value_{output}$$

The reader may be wondering how all the key-value pairs associated with a unique key end up at a single reducer? This functionality is provided by the "shuffle" phase of the underlying MapReduce framework.

After the mapper, and before the reducer, a background shuffle phase comes into play. It involves sorting the mapper outputs, combining all the key-value pairs with the same key into a list format {key, list(values)} and deciding on which reducers to send the list to for further processing. The shuffle phase assures that every key-value pair with the same key goes to the same reducer. It is important to mention here that a single reducer may process more than one lists but, a list corresponding to a unique key is only handled by a single reducer.

Now is a good place to call out the difference between the MapReduce abstraction (also called the programming paradigm) and the MapReduce framework (also called the runtime system). As a user of MapReduce, we load the data into the HDFS and write the MapReduce abstractions, i.e., a 'serial' map function and a 'serial' reduce function, to process the data. The MapReduce system then takes care of everything else such as taking the map function and applying it in parallel to all the input blocks, shuffling intermediate results produced by the mappers and re-routing them to the appropriate reducers, running the reducers in parallel and writing the final output back to the HDFS. The MapReduce framework also provides distributed processing services such as scheduling, synchronization, parallelization, maintaining data and code locality, monitoring, failure recovery, etc. As far as the user is concerned, all this happens automatically. Therefore one of the strengths of MapReduce, and main contributor to its widespread popularity is the ability of the framework to separate the ‘what’ of distributed processing from the ‘how’. The user focuses on the data problem they are trying to solve, and all the required aspects of distributed code execution are transparently handled for them by the framework.

Let's spend a little more time discussing the framework. The MapReduce framework uses the master-slave architecture. The master process is responsible for task scheduling, overall resource management, monitoring, and failure recovery. The slave processes are responsible for managing per node resources and job executions. As an example, we will briefly discuss YARN here, the MapReduce execution framework for Hadoop v2 [7]. YARN, which stands for 'Yet Another Resource Negotiator,' is built on top of HDFS and has three essential elements, as shown in Figure 4 below:

1. A singleton master process called the 'Resource Manager' (RM). The RM keeps track of the slave processes; which cluster node they are running at, how many resources they have available and how to assign those resources to the MapReduce tasks. RM accepts MapReduce job requests, allocate resources to the job and schedules the execution.
2. An 'Application Master' (AM) is spawned by the RM for every accepted MapReduce job request. AM has the responsibility of negotiating appropriate resources from the RM, starting the map and reduce tasks on the assigned resources and monitoring for progress.
3. A per node (or per group of nodes) slave process called the 'Node Manager' (NM) is responsible for announcing itself to the RM along with its available resources (memory, cores) and sending periodic updates.

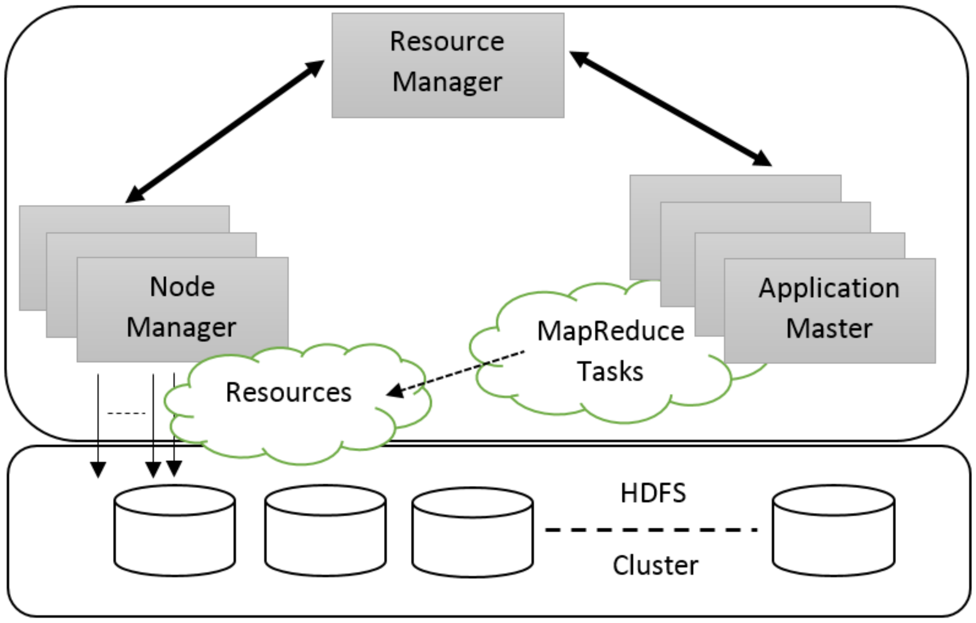

Figure 4: YARN Execution Framework for Hadoop 2.0

We will shift our focus back to the MapReduce programming paradigm, which is the original intent of this chapter. Let's take the classic example of counting word frequency and see how it can efficiently be solved using the MapReduce approach.

Example: Counting Word Frequency

The 'Word Count' example is the 'Hello World' equivalent of the MapReduce paradigm. In this example, we would count the number of occurrences of each unique word in an input dataset, possibly a huge dataset, which consists of text files.

The first step is to split the input files into smaller blocks and to store each block on a distinct node in a distributed cluster with the help of an HDFS. The mapper then looks at the block of text, one line at a time, and emits each word with a count of 1, i.e., the map function output is the key-value pair ($word_i$, 1). It is primarily marking the word as being seen once. All the mappers operate in parallel on the cluster nodes and emit similar key-value pairs. Next, the shuffle phase collects all the words emitted by the mappers, sorts them alphabetically, makes a list for each unique word and sends each list to a reducer. The output of the shuffle phase looks like:{($word_1$, 1), ($word_1$, 1), …., ($word_1$, 1)}, {($word_2$, 1), ($word_2$, 1), …., ($word_2$, 1)}, …, {($word_n$, 1), ($word_n$, 1), …., ($word_n$, 1)}. The reducer then sums the number of occurrences in the input list and emits that value as the final result. The MapReduce mappings of this problem is given below followed by a pictorial depiction of the various phases in Figure 5.

$$map:\ word_1 word_2 \dots word_n\ \rightarrow \{(word_1, 1), (word_2, 1), \dots\ (word_n, 1)\}$$

$$reduce: (word_i, \{1,1, \dots 1\}) \rightarrow word_i: \sum\nolimits_{All} 1$$

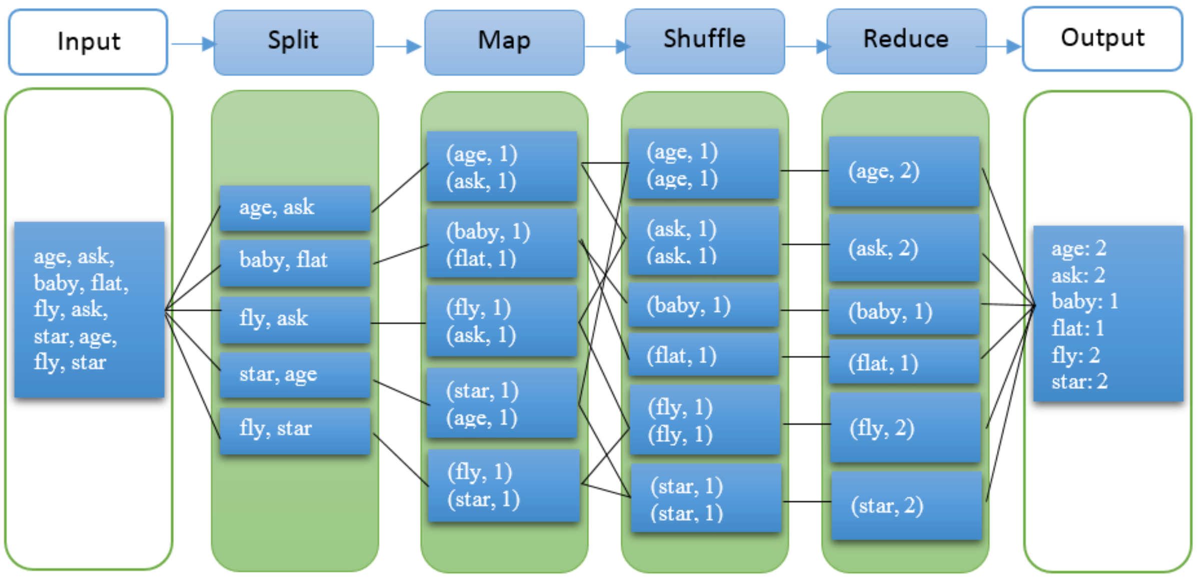

Figure 5: Counting World Frequency using MapReduce

Let's look at another example, where we combine dictionaries using the MapReduce distributed programming paradigm.

Example: Combining Dictionaries

In this example, we will take a set of translation dictionaries, English-Spanish, English-Italian, English-French, and create a dictionary file that has the English word followed by all the different translations separated by the pipe (|) character. For example, looking at Figure 6, if the input files are as shown in the blue boxes, we expect the final output as shown in the green box below. This example is a modified version of the dictionary example discussed at the DZone blog [8].

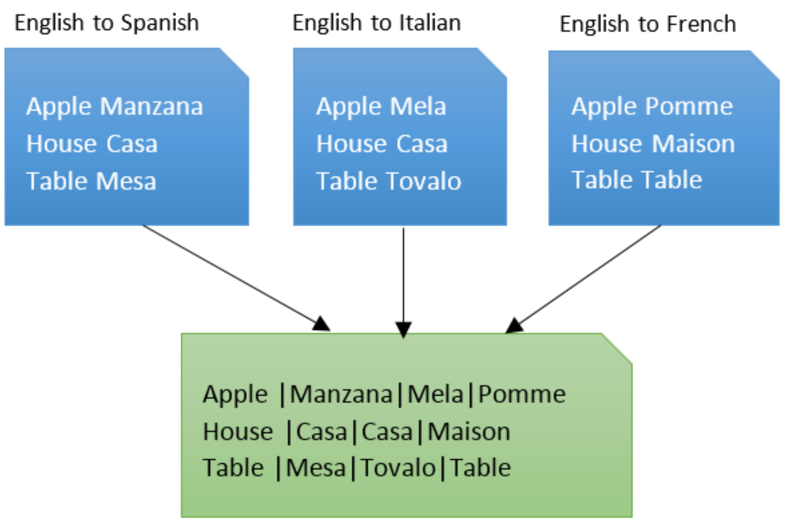

Figure 6: Combining Dictionaries using MapReduce

In this example, each dictionary will be parsed by a mapper (or a set of mappers) line by line, emitting each English word and its corresponding translation as the output of the map function. The reducer, with the help of the shuffle phase, will then receive all the translations related to a particular word and combine them into the final output. We present these mappings below.

$$map : \ word \ \ translation \rightarrow (word, translation)$$

$$reduce : (word_i, \{list(translation)\}) \rightarrow word_i |\ translation_1 | \ldots |\ translation_n$$

For this example, we also present simplified code excerpts of a Java-based implementation of the map and reduce functions. The complete code, along with running instructions can be found at [7].

```
public void map(String key_word, String value_translation, Context context)
{
    context.write(key_word, value_translation);
}
```

```
public void reduce(String key_word, Iterable<String> values, Context context)
```

```
{
    String translations = "";
    for (String value_translation : values)
    {
        translations += "|" + value_translation;
    }
    context.write(key_word, translations);
}
```

We have provided several other examples at the end of this chapter.

Strengths and Limitations of MapReduce

MapReduce is a programming model (and execution framework) for processing large datasets distributed across a cluster with a parallel, distributed algorithm. It has found merits in many applications, such as recommendation systems, processing of logs, marketing analysis, warehousing of data and fraud detection etc.

MapReduce is ideal for running batch computations over large datasets as it can easily scale to hundreds, even thousands, of server nodes. The framework takes the computation to the data rather than bringing the data from various cluster nodes to a centralized processing location. Running mappers on the same node as the data block achieves data locality, consequently conserving precious network bandwidth and allowing for faster processing. [9] The framework is designed to both take advantage of massive parallelism while at the same time hiding messy internal details (parallelization, synchronization, failure recovery, etc.) from the programmer.

Another advantage of the MapReduce programming paradigm is its flexibility. MapReduce programming has the capability to operate on different types and sources of data, whether they are structured (database records) or unstructured (from social media, email, or clickstream, etc.). MapReduce can work on all of them with the help of the various input processing libraries available with the framework.

MapReduce framework is built to be both available and resilient. The underlying HDFS ensures that when data is sent to an individual node in the entire cluster, the very same set of data is also forwarded to the other numerous nodes that make up the cluster. Thus, if there is any failure that affects a particular node, there are always other copies that can still be accessed whenever the need may arise. This replication always assures the availability of data. On top of that MapReduce framework has baked in fault tolerance. In fact this is one of the prime responsibilities of the Master process. The framework can quickly recognize failures that occur and then apply a quick and automatic recovery solution.

MapReduce works well in its domain, offline batch processing, however, it is less effective outside of it. For example, MapReduce is not an ideal solution for tasks that need a shared state or global coordination. MapReduce does not support shared mutable state. The technique is, in general, *embarrassingly parallel*. There is only a single opportunity for a global synchronization in MapReduce which is after the map phase ends and before the reduce phase begins. [10]

Also, as it is designed for large and distributed datasets, the performance is not ideal when it operates on small datasets or individual records. The MapReduce framework has considerable startup and execution costs such as setting up the parallel environment, task creation, communication, synchronization, etc. These overheads are usually negligible as the framework is optimized to conduct batch operations over a large amount of data. However, for smaller problems, it is probably going to be faster to process the data serially on a single fast processor than to go through the entire map/reduce.

MapReduce is not suited towards real-time processing of data, or iterative and interactive applications either. Both Iterative and Interactive applications require faster data sharing across parallel jobs. Unfortunately, in most current frameworks, the only way to reuse data between computations (Ex − between two MapReduce jobs) is to write it to an external stable storage system. Data sharing is slow in MapReduce due to replication, serialization, and disk IO. Apache Software Foundation introduced Spark for speeding up the Hadoop computational computing software process. [11] The main feature of Spark is its in-memory cluster computing that increases the processing speed of an application. It is built on top of Hadoop MapReduce, and it extends the MapReduce model to efficiently use more types of computations which include Interactive Queries and Stream Processing.

The Hadoop-MapReduce Ecosystem

The Hadoop platform primarily consists of two essential services: a reliable, distributed file system called HDFS and the high-performance parallel data processing engine called MapReduce. Though they form the core of the Hadoop project, they are just two parts of a growing system of open source components for large-scale data processing. Below we briefly discuss some of the related technologies. The reader is encouraged to look at thereferences for more details.

Hive [12] was originally developed at Facebook for business analysts to be able to access data on Hadoop using an SQL-like engine. Hive offers techniques to map a tabular structure onto a distributed file system like HDFS, and also allows querying of the data from this mapped tabular structure using an SQL dialect known as HiveQL. HiveQL queries are executed via MapReduce, i.e., when a HiveQL query is issued, it triggers Map and Reduce tasks to perform the operation specified in the query.

Pig [13], developed at Yahoo, is a platform for constructing data flows for extract, transform, and load (ETL) processing and analysis of large datasets. Pig uses a high-level scripting language called Pig Latin. Pig Latin queries and commands are compiled into one or more MapReduce tasks and then executed on a Hadoop cluster.

Hive is Pig's opposite. Where Hive was developed to process completely structured data, Pig can be used for both structured as well as unstructured data (a pig will eat anything!). Both Pig and Hive queries get converted into MapReduce tasks under the hood

MapReduce framework is at its best when the data is huge, and we want to batch process it offline. However, it is not suitable for real-time processing or random read and write accesses. It led to the development of Apache HBase [14], the distributed, scalable, NoSQL database for Hadoop, built on top of HDFS, that is great for quick updates and low latency data accesses. HBase is a column-oriented store and runs on top of HDFS in a distributed fashion. HBase can provide fast, random read/write access to users and applications in near real-time.

Mahout [15] is the machine learning and data mining library for Hadoop. It implements the machine learning and data mining algorithms, such as collaborative filtering, classification, clustering and dimensionality reduction using MapReduce.

Oozie [16], developed at Yahoo, is a workflow coordination service to coordinate, schedule and manage tasks executed on Hadoop. The tasks are represented as action nodes on a Directed Acyclic Graph (DAG), and the DAG sequence is used to control the subsequent actions. You can have several different action nodes within your Oozie workflows such as steps for chaining events, Pig and Hive tasks, MapReduce tasks or HDFS actions.

These Hadoop components are presented in Figure 7 below.

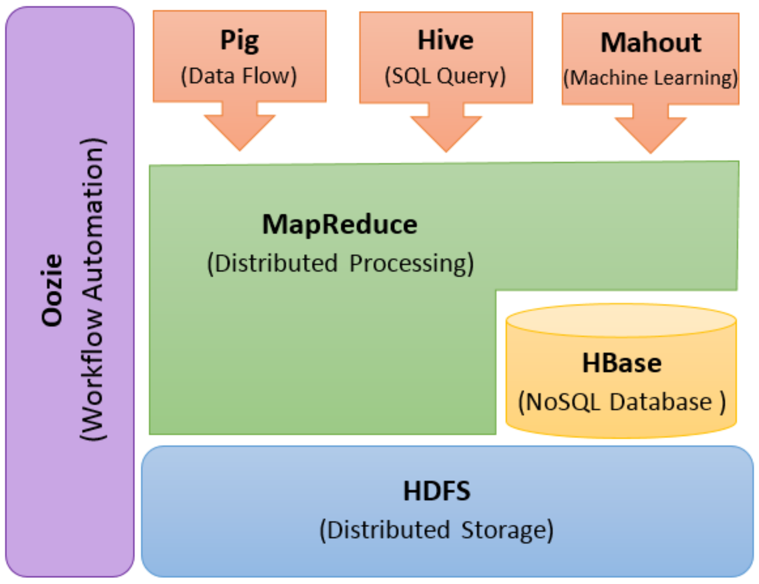

Figure 7: Simplified Hadoop-MapReduce Ecosystem

Apart from those, there are various other Apache Projects built around the Hadoop framework and have become a part of the Hadoop Ecosystem. For a complete list, see [17].

Additional Examples

We wrap this chapter by providing additional MapReduce examples.

Example: Inverted Index

An inverted index consists of a list of all the unique words that appear in any document, and for each word, a list of the documents in which it appears. The inverted index is useful for fast retrieval of relevant information. Let's look at building an inverted index for a set of tweets based on their hashtags and how we can map the solution as a MapReduce.

Input Data:

"It's not too late to vote. #ElectionDay"

"Midtown polling office seeing a steady flow of voters! #PrimaryDay"

"Today's the day. Be a voter! #ElectionDay"

"Happy #PrimaryDay"

"Say NO to corruption & vote! #ElectionDay"

"About to go cast my vote...first time #ElectionDay"

MapReduce mapping:

$$map : tweet \rightarrow (hashtag, tweet)$$

$$reduce : (hashtag, \{list(tweet)\}) \rightarrow hashtag, \{list(tweet)\}$$

Map Output:

```
("ElectionDay", "It's not too late to vote. #ElectionDay")

("PrimaryDay", "Midtown polling office seeing a steady flow of voters! #PrimaryDay")

("ElectionDay", "Today's the day. Be a voter! #ElectionDay ")

("PrimaryDay", "Happy #PrimaryDay")

("ElectionDay", "Say NO to corruption & vote! #ElectionDay")

("ElectionDay", "About to go cast my vote...first time #ElectionDay")
```

Reduce Input:

Reducer 1:

```
("ElectionDay", "It's not too late to vote. #ElectionDay")

("ElectionDay", "Today's the day. Be a voter! #ElectionDay ")

("ElectionDay", "Say NO to corruption & vote! #ElectionDay")

("ElectionDay", "About to go cast my vote...first time #ElectionDay")
```

Reducer 2:

```
("PrimaryDay", "Midtown polling office seeing a steady flow of voters! #PrimaryDay")

("PrimaryDay", "Happy #PrimaryDay")
```

Reduce Output:

```
("ElectionDay", [      "It's not too late to vote. #ElectionDay",
                       "Today's the day. Be a voter! #ElectionDay ",
                       "Say NO to corruption & vote! #ElectionDay",
                       "About to go cast my vote...first time #ElectionDay"])


("PrimaryDay", [       "Midtown polling office seeing a steady flow of voters! #PrimaryDay",
                       "Happy #PrimaryDay"])
```

Example: Relational Algebra (table JOIN)

MapReduce can be used to join two database tables based on common criteria. Let's take an example. We have two tables, where the first contains an employee's personal information primary keyed on SSN and the second table includes the employee's income again keyed on SSN. We would like to compute average income in each city in 2016. This computation requires a JOIN operation on these two tables. We will map the problem to a two-phase MapReduce solution. The first phase effectively creates a JOIN on the two tables using two map functions (one for each table), and the second phase gathers the relevant data for calculating desired statistics.

Input Data:

Table 1: (SSN, {Personal Information})

111222:(Stephen King; Sacramento, CA)

333444:(Edward Lee; San Diego, CA)

555666:(Karen Taylor; San Diego, CA)

Table 2: (SSN, {year, income})

111222:(2016,$70000),(2015,$65000),(2014,$6000),...

333444:(2016,$72000),(2015,$70000),(2014,$6000),...

555666:(2016,$80000),(2015,$85000),(2014,$7500),...

MapReduce Mapping:

Stage 1 (table JOIN)

$$map_{table1} : record_{table\ 1} \rightarrow (SSN, City)$$

$$map_{table2} : record_{table\ 2} \rightarrow (SSN, Income_{2016})$$

$$reduce : (SSN, \{City, Income_{2016}\}) \rightarrow SSN, (City, Income_{2016})$$

Stage 2

$$map : SSN, (City, Income_{2016}) \rightarrow (City, Income_{2016})$$

$$reduce : City, \{list(Income_{2016})\} \rightarrow City, avg(Income_{2016})$$

Stage 1

Map Output:

Mapper 1a: (SSN, city)

```
(111222, "Sacramento, CA")
(333444, "San Diego, CA)
(555666, "San Diego, CA)
```

Mapper 1b: (SSN, income 2016)

```
(111222, $70000)
(333444, $72000)
(555666, $80000)
```

Reduce Input: (SSN, city), (SSN, income)

```
(111222, "Sacramento, CA")
(111222, $70000)
(333444, "San Diego, CA")
(333444, $72000)
(555666, "San Diego, CA")
(555666, $80000)
```

Reduce Output: (SSN, [city, income])

```
(111222, ["Sacramento, CA", 70000])
(333444, ["San Diego, CA", 72000])
(555666, ["San Diego, CA", 80000])
```

Stage 2:

Map Input: (SSN, [city, income])

```
(111222, ["Sacramento, CA", 70000])
(333444, ["San Diego, CA", 72000])
(555666, ["San Diego, CA", 80000])
```

Map Output: (city, income)

```
("Sacramento, CA", 70000)
("San Diego, CA", 72000)
("San Diego, CA", 80000)
```

Reduce Input: (city, income)

Reducer 2a:

```
("Sacramento, CA", 70000)
```

Reducer 2b:

```
("San Diego, CA", 72000)
("San Diego, CA", 80000)
```

Reduce Output: (city, average [income])

Reducer 2a:	Reducer 2b:
`("Sacramento, CA", 70000)`	`("San Diego, CA", 76000)`

The reader is encouraged to think how the solution will differ if the employee is allowed to have multiple addresses, i.e., there can be multiple addresses per SSN in Table 1.

Advanced Example: Graph Algorithm (Single Source Shortest Path)

This example assumes that the reader has familiarity with the graph algorithms terminology, such as vertices, edges, adjacency lists, etc. MapReduce can be used to calculate statistics iteratively where each iteration can use the previous iteration's output as its input. This kind of iterative MapReduce is useful for applications including graph problems. For example, given Figure 8, we would like to calculate the single source shortest path from source vertex **'s'** to all other vertices in the graph. The shortest path is defined as a path between two vertices in a graph such that the sum of the weights of its constituent edges is minimized.

We will be using MapReduce iterative approach to solve this problem, where each iteration, starting from the origin, will be 'radiating' information 'one edge hop' distance at a time.

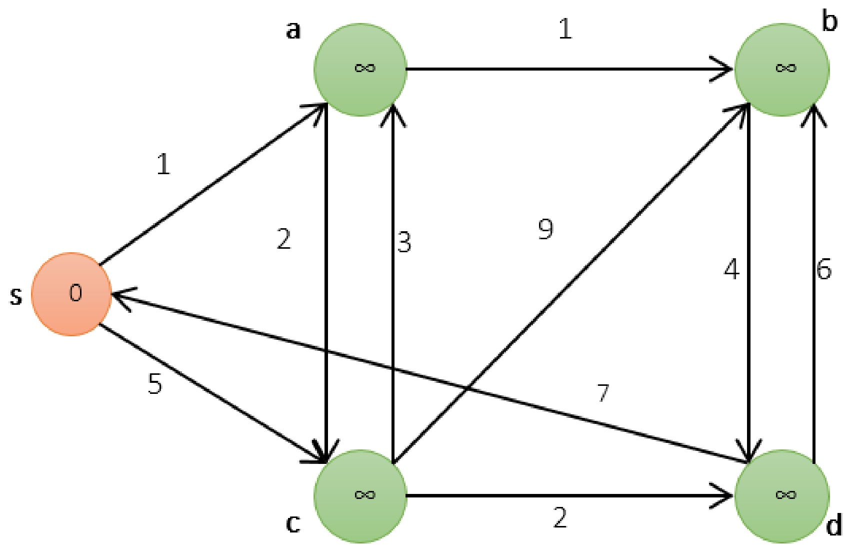

Figure 8: Input Graph for Single Source Shortest Path Problem

Input Data Format:

$$Node: < id, costFromSource, prevHopFromSource, AdjacencyList >$$
$$AdjacencyList: \{neighborNode, costToNeighborNode\}$$

Initial Input Data:

```
Node s: <s, 0, - , {(Node a, 1), (Node c, 5)}>

Node a: <a, ∞, - , {(Node b, 1), (Node c, 2)}>

Node b: <b, ∞, - , {(Node d, 4)}>
Node c: <c, ∞, - , {(Node a, 3), (Node b, 9), (Node d, 2)}>

Node d: <d, ∞, - , {(Node s, 7), (Node b, 6)}>
```

MapReduce Iteration Mapping:

$$map1 : Node.id: Node \rightarrow \{list(Node.neighborNode.id: (Node.id, SUM(Node.costToNeighborNode, Node.costFromSource)))\}$$

$$map2 : Node.id: Node \rightarrow \{list(Node.NeighborNode.id, Node.NeighborNode)\}$$

$$reduce : Node.id: \{list(prevHopFromSource, costFromSource, Node)\} \rightarrow Node.id: Node'$$
$$where$$
$$Node'.costFromSource = MIN(costFromSource),$$
$$Node'.prevHopFromSource = prevHopFromSource_{min}$$

Note: Reducer only emits value if Node structure is updated, i.e., the iteration found a new shortest path from the source.

Iteration 1:

```
Map Input: s: <s, 0, - , {(Node a, 1), (Node c, 5)}>

Map Output: (a: s, 1), (a: Node a), (c: s, 5), (c, Node c)

Reduce 1 Input: a: (s, 1, Node a)

Reduce 1 Output: a: <a, 1, s, {(Node b, 1), (Node c, 2)}>

Reduce 2 Input: c: (s, 5, Node c)

Reduce 2 Output: c: <c, 5, s, {(Node a, 3), (Node b, 9), (Node d, 2)}>
```

The reader is encouraged to continue the example and see how the solution converges in 4 iterations.

References

1. [Online]. "Mind-blowing growth & power of big data - Business Insider" Available: http://www.businessinsider.com/mind-blowing-growth-and-power-of-big-data-2015-6
2. EMC Digital Universe with Research & Analysis by IDC. The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things. 2014
3. [Online]. "Volume, velocity, and variety: Understanding the three V's of big data," in DIY-IT Available: http://www.zdnet.com/article/volume-velocity-and-variety-understanding-the-three-vs-of-big-data/
4. G. Sanjay, G. Howard, and L. Shun-Tak, "The Google File system," in ACM SIGOPS Operating Systems Review - Volume 37 Issue 5, December 2003
5. D. Jeff and G. Sanjay, "MapReduce: Simplified Data Processing on Large Clusters," in Communications of the ACM – 50th Anniversary Issue, Vol. 51 No. 1, Pages 107-113, 2008.
6. [Online]. "Apache Hadoop" Available: http://hadoop.apache.org/
7. [Online]. "Apache Hadoop YARN" Available: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html
8. [Online]. "Hadoop Basics—Creating a MapReduce Program," DZone Available: https://dzone.com/articles/hadoop-basics-creating
9. "Data-Intensive Text Processing with MapReduce" by Jimmy Lin and Chris Dyer, University of Maryland, College Park, Manuscript prepared April 11, 2010
10. "MapReduce Patterns, Algorithms, and Use Cases" by Ilya Katsov in Highly Scalable Blog, 2012 (https://highlyscalable.wordpress.com/2012/02/01/MapReduce-patterns/)
11. [Online]. "Apache Spark" Available: http://spark.apache.org/
12. [Online]. "Apache Hive" Available: https://hive.apache.org/
13. [Online]. "Apache Pig" Available: https://pig.apache.org/
14. [Online]. "Apache HBase" Available: https://hbase.apache.org/
15. [Online]. "Apache Mahout" Available: http://mahout.apache.org/
16. [Online]. "Apache Oozie" Available: http://oozie.apache.org/
17. [Online]. "The Hadoop Ecosystem Table" Available: https://hadoopecosystemtable.github.io/