# Introduction to MapReduce

Mahmoud Parsian

mahmoud.parsian@yahoo.com

LinkedIn: http://www.linkedin.com/in/mahmoudparsian

GitHub: https://github.com/mahmoudparsian/data-algorithms-book

October 10, 2015

# Contents

# 1 Introduction

This is an introductory and companion chapter for the Data Algorithms book[9] on MapReduce programming model. The purpose of this chapter is to shed some light on the concept of MapReduce programming model by some basic examples from Hadoop[1] and Spark[2]. The other purpose of this chapter is to show that MapReduce is a foundation for solving big data using modern and powerful Spark API. For solving big data problems, our experience indicate that MapReduce by itself (as a series of `map` and `reduce` functions) is not sufficient to address all types of problems. Clearly Spark API (as a superset of MapReduce paradigm) addresses the big data problem as a general compute engine (by providing basic MapReduce as well as other powerful features such as `join()`, `filter()`, `cartesian()`, and `combineByKey()`).

You should select your MapReduce paradigm based on your specific requirements (there is not a silver bullet[2] for all big data problems). But as we look at the big data platforms (Hadoop, Spark, Storm, Tez), MapReduce paradigm is a foundation for most of the data-intensive text processing. Hadoop will continue to be part of the big data echo system (because it provides MapReduce and distributed file system), but Spark is emerging as the choice of compute engine with and without using Hadoop. MapReduce as a processing paradigm can be restrictive for certain application types (such as graph and iterative algorithms), but Spark removes the restrictions and provides a simple, expressive, and powerful API for richer processing systems. For details on MapReduce basics, you should refer to Jimmy Lin's excellent book: Data-Intensive Text Processing with MapReduce[7].

To demonstrate basic MapReduce concepts and a simple MapReduce design pattern (*InMapper Combining*), at the end of this chapter, I provide 4 distinct solutions (2 in Spark and 2 in Hadoop) for the simple `"character count"`(counting frequency of all unique characters for a given corpus) problem:

- Spark

[1]The Apache Hadoop project develops open-source software for reliable, scalable, distributed computing. `http://hadoop.apache.org/`

[2]Apache Spark is a fast and general engine for large-scale data processing. `http://spark.apache.org/`

  - Using Basic[3] Design Pattern
  - Using *InMapper Combining* Design Pattern
- Hadoop
  - Using Basic Design Pattern
  - Using *InMapper Combining* Design Pattern

Informally, MapReduce is a programming paradigm that allows for massive scalability across tens, hundreds, or thousands of servers in a cluster (this cluster can be built by Hadoop or Spark or combinations). The MapReduce concept is associated with the scale-out intensive data processing solutions. Since MapReduce is a programming paradigm, therefore it can be implemented by concrete instances such as Hadoop, Spark, Tez[4] and Storm[5]. We will only focus on Hadoop and Spark. In this chapter, for our examples, we will only use the Java programming language (note that these paradigms support other programming languages such as Scala, Python, and R).

In a nutshell, MapReduce framework means "scaling-out" your solution by targeting data-intensive computations. Scaling-out means adding more servers of similar memory and CPU. Scaling-up means adding more memory and CPUs to a single server. A central goal of MapReduce framework is to scale-out and not to scale-up; this means that the distributed cluster should utilize a large number of commodity servers as opposed to a small number of high-end and specialized servers. MapReduce implementations (such as Hadoop and Spark) use a cluster of servers. The servers that execute the `map()` function are called mappers and the servers that execute the `reduce()` function are called reducers. In MapReduce implementations, there is one (or more than one) server called the master which is in charge of distributing and assigning the `map()` and `reduce()` tasks to slave servers and checking on their progress by communicating with them periodically until all the tasks are completed. Hadoop supports a limited number of primitive functionality such as `map()`, `combine()`, and `reduce()`, while Spark

[3]By "Basic" design pattern, we mean that we just emit basic `(key, value)` pairs rather than using any custom data types or functions

[4]The Apache Tez project is aimed at building an application framework which allows for a complex directed-acyclic-graph of tasks for processing data. It is currently built atop Apache Hadoop YARN. `http://tez.apache.org/`

[5]Apache Storm is a free and open source distributed realtime computation system. Storm makes it easy to reliably process unbounded streams of data, doing for realtime processing what Hadoop did for batch processing. `http://storm.apache.org`

has higher abstraction API (using RDDs – discussed in [9]) than Hadoop and provides superset functionality of MapReduce by "transformations" and "actions". Spark is a general-purpose graph execution engine for Hadoop and non-Hadoop. Spark allows users to analyze large data sets by creating DAGs (directed acyclic graph) with very high performance. It is fair to say that the Spark API is a general purpose execution engine and a superset of the Hadoop API.

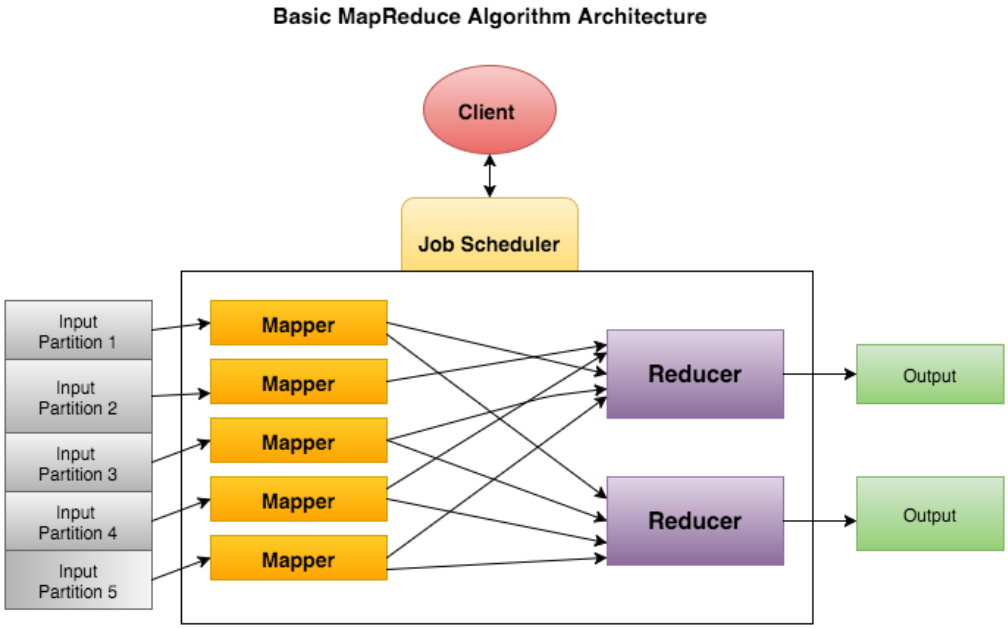



Figure 1: Basic MapReduce Algorithm Architecture

Figure 1 shows the data flow in a MapReduce framework.

## 2 Benefits of MapReduce

In a nutshell, the following are some of the key benefits of using MapReduce framework:

- concurrency and distributed computing environment

- a well-defined and simple programming interface (you just need to write `map()` and `reduce()` functions). Note that these functions may have slightly different name on concrete implementations such as Hadoop, Spark, and Storm.

- high scalability: you can add any number of commodity servers to a MapReduce cluster. This can be easily done by updating configuration files.

- fault tolerance: MapReduce framework provides graceful failure handling. This is one of the most important features of distributed computing such as Spark and Hadoop. How does Spark handle fault tolerance? "Resilient Distributed Datasets – RDDs[11] – achieve fault tolerance through a notion of lineage: if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition" (source [10]). How does Hadoop handle fault tolerance? "The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed. Any map tasks completed by the worker are reset back to their initial idle state and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to idle and becomes eligible for rescheduling" (source [1]).

- shared-nothing cluster: no need for expensive and error-prone synchronization between distributed tasks

MapReduce is a framework inspired in functional programming[6] to solve problems in which algorithmic steps can be distributed and made parallel by applying a divide and conquer approach. Apache Hadoop[7] is the de-facto industry standard for MapReduce implementation. There are several other MapReduce solutions such as Hadoop, Storm, Tez, and Spark. In MapReduce framework, the persistent storage is provided by "distributed file system" (in Hadoop it is called HDFS[8]) and the analysis (i.e., intensive computations) by MapReduce's `map()`, `reduce()`, and `shuffle()`[9] functions. To solve your problem by MapReduce, you focus on preparing your input,

[6] `http://en.wikipedia.org/wiki/Functional_programming`

[7] `http://hadoop.apache.org/`

[8] Hadoop Distributed File System

[9] The shuffle() function is one of the core functions in MapReduce framework, which enables to send the same keys (along with their associated values) to the same server

and on then providing two plug-in classes: a mapper class (which defines the `map()` function) and a reducer class (which defines the `reduce()` function). While MapReduce functionality in Hadoop is very limited (there can be only mappers and reducers), Spark on the other hand is just not limited to mappers and reducers, and it provides a superset of `map()` and `reduce()` functions. Spark has an advanced DAG execution engine that supports cyclic data flow and in-memory computing and solves Hadoop's limitations by providing a very rich functionality to the MapReduce paradigm.

MapReduce's framework offers "shared nothing"[10] architecture, which promotes scalability. What does this mean? It means that, in a given MapReduce cluster (comprised of hundreds to thousands of servers), a master or slave server should not share any hardware resource with any other server. This key design concept eliminates synchronization of resources between servers and promotes massive parallelism and distributed-ness/distribution?. Shared nothing architecture is important for back-end computing–such as indexing a search engine with billions of documents, DNA-Sequence analysis, and regression analysis–because of its scalability.

We should note that MapReduce is not a solution for every problem we face. To have a better understanding about this issue, Jimmy Lin[6] states that "Hadoop is currently the large-scale data analysis 'hammer' of choice, but there exist classes of algorithms that are not 'nails' in the sense that they are not particularly amenable to the MapReduce programming model." Furthermore, we should understand that "real-time computation on continuous, large-volume streams of data is not something that MapReduce is capable of doing. MapReduce is fundamentally a batch processing framework" (source [6]). For example, using Spark and Hadoop, a near-real-time can be achieved if your near-real-time can be defined as 5 to 60 seconds.

Input to your MapReduce solution can be expressed by a set of input files and directories (by using distributed file system - DFS). You may designate your output directory by using DFS as well. Once you have defined your Input/Output, then you may launch/submit your MapReduce job. Finally, you may browse the progress and status of your submitted job by a web browser (for example, in Hadoop, you may use `http:`

[10] A "shared nothing" architecture is a distributed computing architecture in which each node is independent and self-sufficient, and there is no single point of contention across the system. More specifically, none of the nodes share memory or disk storage. Source: `http://en.wikipedia.org/wiki/Shared_nothing_architecture`

`//<namenode-server>:50030/cluster/` to view status of all submitted MapReduce jobs and in Spark cluster, you may use `http://<spark-master>:8080/` to view status of Spark jobs – the ports mentioned in these URLs are configurable).

# 3 MapReduce's Features

MapReduce's framework provides the following features:

- Problem-focused abstraction: this enables you to focus on solving the problem in distributed computing environment without worrying on which server might have crashed or how to do synchronization between servers (you may utilize one to thousands of servers)
- Load balancing: MapReduce framework provides this as a basic service: your job load and data is distributed among all slave servers
- Analyze and process extremely large data sets
- Parallel and distributed computing by using a set of commodity servers
- Shared-nothing framework (no data synchronization, which means that each slave node can only communicate with the master node and there is no communication between slave nodes)
- Based on the functional programming concepts of higher-order functions: `map()`, `shuffle()`, and `reduce()` (note that the `shuffle()` function is a built-in feature of MapReduce framework and a developer mostly focuses on `map()` and `reduce()` functions). For example, MapReduce is the heart of Apache's Hadoop framework.
- Scalable (scale-out) - from one server to thousands
- Big persistent data storage called distributed file system (in Hadoop this is called Hadoop Distributed File System - HDFS)
- Fault-tolerant - assumes servers will crash from time to time (for example, Spark and Hadoop implementation provides a fine-grained fault-tolerance for `map()` and `reduce()` functions and failure in the middle of a long execution does not require restarting the whole job from scratch)

- Low cost – cheap – uses commodity servers, no custom hardware needed (you may use servers with minimum of 16GB of RAM to servers with 2TB of RAM – in your configuration for big servers, you may assign more `map()` and `reduce()` slots)
- Ease of use (programmer mainly focuses on MapReduce algorithm rather than spending time on scheduling and distributing job/tasks to many servers)

A MapReduce algorithm proceeds in one or more rounds by a set of servers with one or more master nodes (job scheduler, task distributor, and controller) and a big set of slave nodes (responsible for accepting tasks and task execution). Typically, each MapReduce round has three phases:

- *map* (developed by a programmer)
- *shuffle* (automatically provided by MapReduce framework implementation, this is a built-in feature, which aggregate values by keys generated by mappers)
- *reduce* (developed by a programmer)

To learn how to chain Hadoop MapReduce jobs (how to have more than one round of MapReduce jobs), you should review the Chapter 5 from "Hadoop in Action" book by Chuck Lam (`http://www.manning.com/lam/`). MapReduce framework implementation (such as Hadoop) executes a MapReduce algorithm (defined by `map()` and `reduce()` functions) in all available servers in the same way. Fortunately, in Spark, you are not limited just to the `map()` and `reduce()` functions and you may express your solution as a DAG, which may have any combinations of `map()`, `reduce()`, `join()`, `filter()`, and `sort()`.

### 3.1 Key-Value Pairs

In MapReduce, key-value pairs are the basic data structure to deal with inputs, outputs, mappers, and reducers. Keys and values may be primitives such as strings, integers, doubles, and raw bytes, or they may be arbitrarily complex data structures (tuples, lists, associative arrays, and arrays). A programmers may need to define his own custom data types for key-value pairs. To define a custom data type in Hadoop, it has to implement the

`Writable` [11] interface and in Spark it has to implement the `Serializable`[12] interface. There are many libraries (such as Cloud9[13]) to simplify the task of custom data type creation. Overall, it is much easier to define custom types in Spark than Hadoop (in Spark, your custom type has to implement `Serializable`, but in Hadoop you need to implement the `Writable`, which involves implementing several additional methods).

The following are some examples of key-value pairs:

| Key | Value |
|---|---|
| `String` | `Integer` |
| `String` | `Tuple2<Integer, Double>` |
| `Long` | `Map<String, Integer>` |
| `String` | `Tuple3<String, Integer, Double>` |
| `Tuple2<String, Integer>` | `Map<String, Integer>` |

### 3.2 MapReduce's Input Partition Phase

The input partitioner partitions data into small chunks (configured by MapReduce programmer or administrator) and sends it to mappers. Then each `map()` function performs its computation on these small input chunks. Therefore, the main function of the input reader is to divide the whole input into appropriate size 'chunks' (typically 64MB to 512MB can be configured) and the MapReduce framework assigns one chunk to each `map()` function. The MapReduce's input reader reads data from a distributed file system and generates key-value pairs for mappers. Note that concrete implementations of MapReduce paradigm (such as Hadoop, Spark, and Storm) may implement partitioning the input data differently.

[11] `org.apache.hadoop.io.Writable`
[12] `java.io.Serializable`
[13] `https://github.com/lintool/Cloud9`

### 3.3 MapReduce's Map Phase

In map phase, the partitioner generates a list of key-value pairs `(K, V)` using MapReduce's distributed file system. The key `K` can be any type, and the value `V` can be any type of arbitrary data structure or it can be any basic data type, such as Integer, String, Double, etc. The `map()` function will accept a key-value pair and may generate any number of $(K_2, V_2)$ pairs. The generated pairs $(K_2, V_2)$ will be transmitted to another server in the shuffle phase, such that the recipient server is determined solely by key $K_2$.

### 3.4 MapReduce's Shuffle Phase

The shuffle phase is a major feature of MapReduce framework, which enables parallelism and distributed computing. The main function of shuffling is to transfer data from the mappers to the reducers. In a nutshell: "after the Map phase and before the beginning of the Reduce phase is a handoff process, known as shuffle and sort. Here, data from the mapper tasks is prepared and moved to the nodes where the reducer tasks will be run. When the mapper task is complete, the results are sorted by key, partitioned if there are multiple reducers, and then written to disk."(source: `http://www.dummies.com/how-to/content/the-shuffle-phase-of-hadoops-mapreduce-application.html`).

To understand shuffle phase better, let `T` be the list of all key-value pairs that all the machines generate in the map phase. The main task of shuffle phase is to distribute the `T` across the MapReduce servers adhering to the constraint that pairs with the **same key** must be delivered to the **same server**. Therefore, if $\{(K, V_1), (K, V_2), ..., (K, V_n)\}$ are the pairs in `T` having a **common key** `K`, all of them will arrive at the **same server**.

Therefore, the shuffle phase can be summarized as:

- First, the shuffle phase accepts all of the $(K_2, V_2)$ pairs generated from the output of map phase
- Then, it groups them together by the $K_2$ (these keys are generated by all mappers
- Finally, all of the pairs from all over the cluster having the same key (i.e., $K_2$) are merged into a single $(K_2, [V_{21}, V_{22}, V_{23}, ...])$ pair, which will be processed by a single reducer (we use `[ ]` notation to denote list of items).

### 3.5 MapReduce's Reduce Phase

The Reduce phase is designed to act as an aggregation function. The master server incorporates the key-value pairs received from the previous phase into its distributed file system. Each reducer receives a key-value where "key" is a unique key (such as $K$) and the "value" is a list of values (generated by mappers). Note that, values arrive to a reducer are unsorted (there is no ordering between them). To understand this better, let `T` be the list of all key-value pairs that all the cluster nodes generated in the map phase. So we have:

$$
\begin{aligned}
\mathtt{T} = \{ & \\
& (\mathtt{K_1}, \mathtt{V_{11}}), (\mathtt{K_1}, \mathtt{V_{12}}), (\mathtt{K_1}, \mathtt{V_{13}}), ..., \\
& (\mathtt{K_2}, \mathtt{V_{21}}), (\mathtt{K_2}, \mathtt{V_{22}}), (\mathtt{K_2}, \mathtt{V_{23}}), ..., \\
& ... \\
& (\mathtt{K_m}, \mathtt{V_{m1}}), (\mathtt{K_m}, \mathtt{V_{m2}}), (\mathtt{K_m}, \mathtt{V_{m3}}), ... \\
\} &
\end{aligned}
$$

Note that each reducer key may have any number of values of the same data type. For example, the key $\mathtt{K_1}$ might have 100 values, while the key $\mathtt{K_2}$ might have 4589 values (this all depends on how mappers generated key-value pairs to be consumed by reducers). Here we have $m$ unique keys $\{\mathtt{K_1}, \mathtt{K_2}, ..., \mathtt{K_m}\}$ generated by all mappers. Therefore we will have $m$ reducers (in total) working concurrently on these lists. For example, if a reducer received key $\mathtt{K_1}$, then the values available for that reducer will be $\{\mathtt{V_{11}}, \mathtt{V_{12}}, \mathtt{V_{13}}, ...\}$ and if a reducer received key $\mathtt{K_2}$, then the values available for that reducer will be $\{\mathtt{V_{21}}, \mathtt{V_{22}}, \mathtt{V_{23}}, ...\}$ and so on. After all servers have completed the reduce phase, the current round of MapReduce terminates.

### 3.6 MapReduce's Map and Reduce Phases

MapReduce has two key functions: `map()` and `reduce()`. A programmer writes these functions. Typically, the `map()` function is a transformation function and the `reduce()` function acts as an aggregation function. For example, in the classic "word count" problem, mappers will emit `(word,`

`1)` pairs and reducers will count them up per unique word. MapReduce system (such as Hadoop) partitions input data and passes them to the `map()` function. The main task of MapReduce system is to ensure locality (each server can work on data independently). The mappers and reducers can work independently on different servers. For example, in Hadoop, output of `map()` functions are shuffled and sorted (this is the magical work of the Hadoop's MapReduce framework – and not of the programmer) and then fed to `reduce()` functions.

Also, MapReduce supports divide-and-conquer algorithm design technique: it enable us to divide a problem into smaller ones – partition input data and computations into smaller pieces – until the individual problems can be solved independently and then combined to answer the original question. MapReduce is a distributed computing framework for implementing divide-and-conquer algorithms in a scalable manner, by automatically distributing units-of-work to slave nodes in an arbitrarily large cluster of nodes. One important aspect of MapReduce is a "fault-tolerance" where MapReduce framework handles failures of individual slave nodes by redistributing the unit-of-work to other slave nodes.

MapReduce framework is depicted in the following figures:

- MapReduce framework without combiners (Figure 2)
- MapReduce framework with combiners (Figure 3)

A MapReduce cluster is comprised of a set of nodes: one master, one job tracker, and a set of slave nodes. This configuration is for Hadoop and it might be different for other MapReduce frameworks. In MapReduce, input data is initially partitioned across the nodes of a cluster and stored in a distributed file system. In Hadoop, this is called Hadoop Distributed File System (HDFS). MapReduce paradigm interprets and represents all data records as a (`key`, `value`) pairs. The MapReduce computation is expressed using two functions:

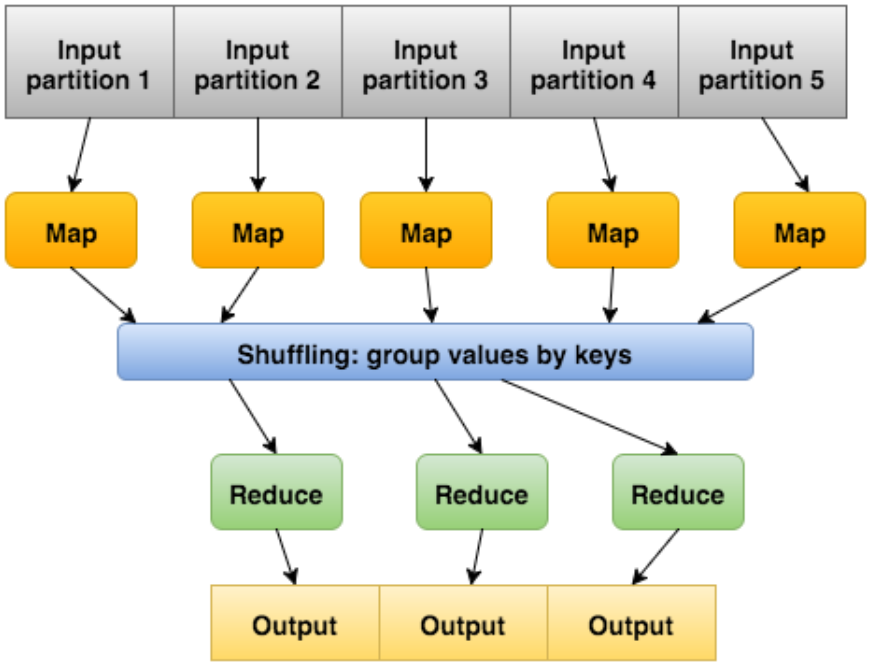



Figure 2: MapReduce framework without combiners

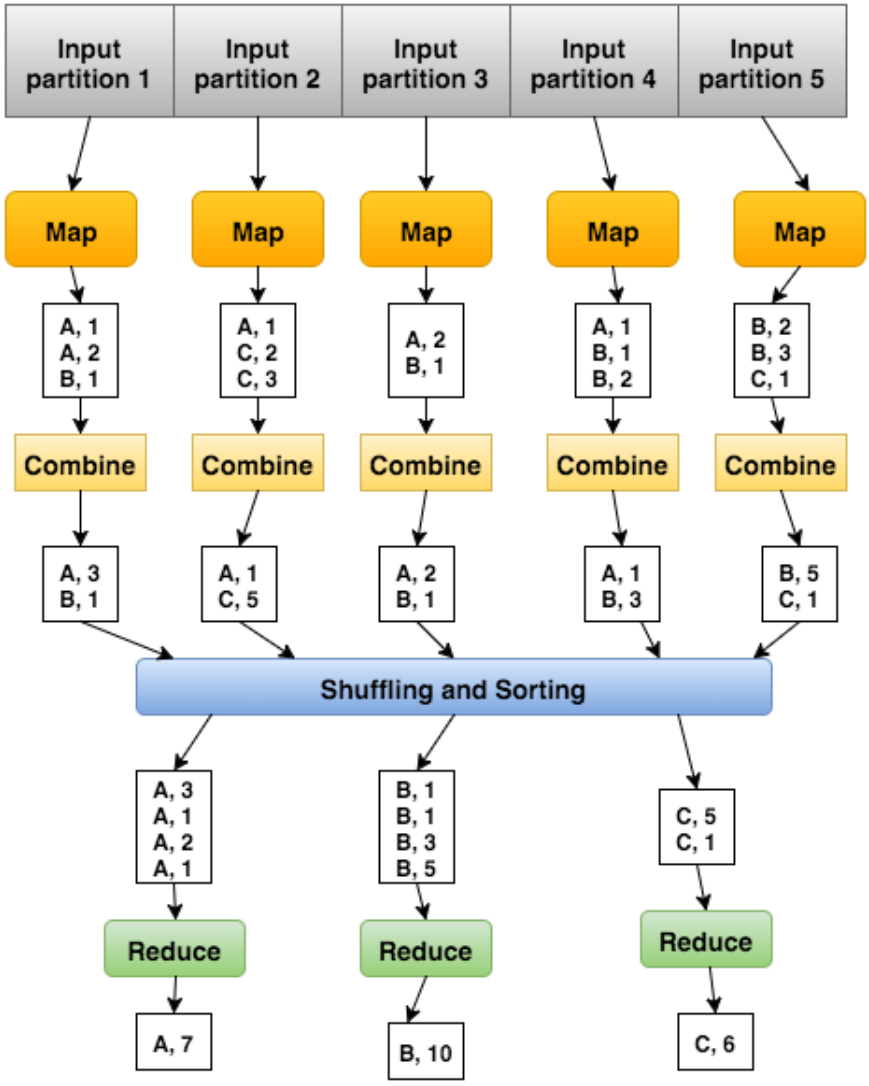



Figure 3: MapReduce framework with combiners

$$\texttt{map}(\texttt{key}_1, \texttt{value}_1) \rightarrow [(\texttt{key}_2, \texttt{value}_2)]$$

$$\texttt{reduce}(\texttt{key}_2, \texttt{list}(\texttt{value}_2)) \rightarrow [(\texttt{key}_3, \texttt{value}_3)]$$

Hadoop's MapReduce framework has a very powerful `sort` and `shuffle` feature capability, which sorts all intermediate data by the intermediate key (i.e., $\texttt{key}_2$). This sort capability enables aggregation of intermediate data and can be used as a powerful tool in de-duplication of data, such as finger prints.

# 4 MapReduce's Inputs and Outputs

According to Apache's Hadoop[14]: "The MapReduce framework operates exclusively on `<key, value>` pairs, that is, the framework views the input to the job as a set of `<key, value>` pairs and produces a set of `<key, value>` pairs as the output of the job, conceivably of different types. The `key` and `value` classes have to be serializable by the framework. For example, in Hadoop, the `key` and `value` classes need to implement the `Writable` interface. Additionally, the `key` classes have to implement the `WritableComparable` interface to facilitate sorting by the framework."

The following entries summarize Input and Output types of a MapReduce job:

- MapReduce without combiner

$$\texttt{map}(\texttt{key}_1, \texttt{value}_1) \rightarrow [(\texttt{key}_2, \texttt{value}_2)]$$

$$\texttt{reduce}(\texttt{key}_2, [\texttt{value}_2]) \rightarrow [(\texttt{key}_3, \texttt{value}_3)]$$

- MapReduce with combiner

$$\texttt{map}(\texttt{key}_1, \texttt{value}_1) \rightarrow [(\texttt{key}_2, \texttt{value}_2)]$$

$$\texttt{combine}(\texttt{key}_2, [\texttt{value}_2]) \rightarrow [(\texttt{key}_2, \texttt{value}_2)]$$

$$\texttt{reduce}(\texttt{key}_2, [\texttt{value}_2]) \rightarrow [(\texttt{key}_3, \texttt{value}_3)]$$

[14] http://hadoop.apache.org/docs/r1.1.2/mapred_tutorial.html

In MapReduce paradigm, it is recommended and desirable to design and implement MapReduce algorithms to use combine functions, because the `combine()` function provides the following benefits:

- Reduce the result size of `map()` functions, which also improves the cost of I/O network
- Perform reduce-like function in each machine (slave severs), which improves utilization of server resources
- Decrease the sort and shuffling cost

Apache Hadoop provides a combine function as a plug-in (you may set your combiner function by `Job.setCombinerClass(CombinerClassName.class)`, but does not guarantee that they will be executed. In most of the cases your `combine()` function will be the same as your `reduce()` function). You can utilize `combine()` function if the output of mappers form a monoid. Monoids and MapReduce combiners are discussed in detail in [5] and chapter 31 of Data Algorithms book[9].

### 4.1 MapReduce's Optional Functions

In MapReduce framework implementation (such as Apache's Hadoop), each `map` or `reduce` task can optionally use two additional functions: `init()` and `close()`

- `init()` : called at the start of each map or reduce task (in Hadoop, this function/method is called `setup()` – called once at the beginning of the task). This functions is used to pass parameters from MapReduce drivers to mappers or reducers.
- `close()`: called at the end of each map or reduce task (in Hadoop, this function/method is called `cleanup()` – called once at the end of the task). This function may be used in closing database connections or file handles.

In Spark, MapReduce's optional functions can be easily implemented using the `mapPartitions()` method:

**Listing 1:** MapReduce's Optional Functions in Spark

```
JavaPairRDD<K,V> pairRDD = ...;
JavaRDD<Tuple2<K2,V2>> partitions = pairRDD.mapPartitions(
   new FlatMapFunction<Iterator<Tuple2<K,V>>, Tuple2<K2,V2>>() {
   @Override
   public Iterable<Tuple2<K2,V2>> call(Iterator<Tuple2<K,V>> iter) {
      setup();   // <-- optional function
      while (iter.hasNext()) {
         // map() functionality
      }
      cleanup(); // <-- optional function
      return <the-result>;
   }
});
```

# 5 MapReduce Programming Model

MapReduce programming model in a nutshell can be expressed as:

- Input: a list of $(\texttt{key}_1, \texttt{value}_1)$ pairs
- Map: apply the `map()` function to all input pairs $(\texttt{key}_1, \texttt{value}_1)$ and generate any number $(\texttt{key}_2, \texttt{value}_2)$ pairs
- Combine: apply the `combine()` function to local values with key $\texttt{key}_2$ (this acts as a local reducer per worker node); this step creates $(\texttt{key}_2, \texttt{value}_2)$ pairs
- Shuffle: group all pairs with the same key $\texttt{key}_2$ together
- Reduce: apply the `reduce()` function to all values with $\texttt{key}_2$ and generate $(\texttt{key}_3, \texttt{value}_3)$ pairs
- Output: a list of $(\texttt{key}_3, \texttt{value}_3)$ pairs

The MapReduce programming model is based on `(key, value)` pairs. The Map function (`map()`) is provided by the programmer, accepts a chunk of input $(\texttt{key}_1, \texttt{value}_1)$ pairs, and produces a set of intermediate key-value pairs as $(\texttt{key}_2, \texttt{value}_2)$. The sort and shuffle phase, provided by the MapReduce implementation, groups together all intermediate values associated with the same intermediate $\texttt{key}_2$. The shuffle and sort phase of MapReduce (not as a responsibility of a programmer, but a magical component of a MapReduce implementation such as Hadoop and Spark) groups output of mappers by keys and sends $(\texttt{key}_2, [\texttt{value}_2])$ to reducers. Finally, in the Reduce function (`reduce()`), provided by the programmer, intermediate values for the

same $\texttt{key}_2$ are aggregated and processed together, to produce a final set (as output) $(\texttt{key}_3, \texttt{value}_3)$ pairs. Therefore, if a `map()` has produced $N$ unique keys, then $N$ reducers (`reduce()`) can run in parallel.
In MapReduce systems, the input datasets (a large set of records or documents – in giga/tera/peta bytes) are partitioned into many smaller datasets. Then these small datasets are then passed along to a pool of mappers operating concurrently. If the number of partitioned data is greater than the number of workers/mappers, then the MapReduce system will use a queuing system to process the data. For example, if you have partitioned input into 300 partitions and if you have 70 mappers, then you need five iterations to process the whole mapping:

```
1st iteration: process first 70 partitions
2nd iteration: process next  70 partitions
3rd iteration: process next  70 partitions
4th iteration: process last  70 partitions
5th iteration: process last  20 partitions
```

While mapping is not completed, at any one time, there will be at most 70 `map()` functions executing on all configured servers. Once all mapping is completed then the reducers will start to work. Note that no mapper will be idle during the mapping process. When a mapper completes a task, the master assigns another map task to it. Note that any time (during mapping phase), there will be at most 70 mappers executing. Before assigning tasks to reducers, the MapReduce system will handle one important task: to sort and group all keys generated by the mappers. This is the shuffle and sort phase in which the programmer takes no part. Sort and shuffle phase is a core functionality, which is provided by the MapReduce implementation (such as Hadoop and Spark).
The keys generated by mappers will be aggregated and then passed to reducers. For argument's sake, let's say that all mappers created 130 unique keys, then 130 independent `reduce()` functions will be called to complete the solution. For example, if the MapReduce cluster is configured to run 50 reducers at the same time, then we need three iterations to process the whole reduce (while reducing is not completed, at any one time there will be at most 50 `reduce()` functions executing on all configured servers):

```
1st iteration: process first 50 keys
2nd iteration: process next  50 keys
3rd iteration: process next  30 keys
```

Again, note that no reducer will be idle during the reducing process. The moment a reducer finishes a `reduce()`, another reduce task will be assigned (by the master) to it, with all tasks executed in parallel rather than in sequence.

## 5.1 MapReduce Programming Model in Detail

- Input: a list of $(\texttt{key}_1, \texttt{value}_1)$ pairs
- Map function:
  $\texttt{map}(\texttt{key}_1, \texttt{value}_1) \rightarrow \texttt{List}(\texttt{Tuple2}(\texttt{key}_2, \texttt{value}_2))$
  - Can be executed in parallel for each pair of $(\texttt{key}_1, \texttt{value}_1)$
  - All `map()` functions can be done in parallel
  - `map()` function must be side-effect free
  - Provided by the programmer
- Combine
  - This phase is optional and provided by the programmer
  - Combine apply `reduce()` to local values with `key` in the same local server
  - Combine can reduce network traffic by doing more in the same local server
- Shuffle: Aggregate and group all pairs with the same `key`
  - Synchronization step (prepare input for reducers) and is triggered when a Mapper completes
  - Handled by the MapReduce system automatically (programmer may inject some parameters to affect the sort and shuffle phase)
  - These keys are generated by `map()`
  - Intermediate data from Mapper is copied to Reducer machines
  - All data for a partition goes to same Reducer
- Reduce function:
  $\texttt{reduce}(\texttt{key}_2, \texttt{List}(\texttt{value}_2)) \rightarrow \texttt{List}(\texttt{Tuple2}(\texttt{key}_3, \texttt{value}_3))$
  - Can be executed in parallel for each $\texttt{key}_2$
  - a `reduce()` processes one `key` and all values associated to it,

  - `reduce()` function must be side-effect free
  - Provided by the programmer
- Output: a list of $(\texttt{key}_3, \texttt{value}_3)$ pairs

Below, we observe some notes on the "sort" phase of MapReduce framework:

- Sort intermediate data by key (these keys are generated by `map()` function)
- All records for key are then contiguous
- Sorting can not start until all Mappers are finished

## 5.2 MapReduce Programming Model Summarized

Let $D$ be a set of some big data, which has been partitioned into $n$ blocks $\{\mathtt{D}_1, \mathtt{D}_2, ..., \mathtt{D}_n\}$. These data blocks are distributed among different servers (a MapReduce framework can utilize hundreds to thousands of servers). More than one data block can go to the same server. If we have more than $n$ servers, then each data block may go to a distinct server. Then the MapReduce framework proceeds in rounds, each with 3 steps:

1. Mapper: map all $\mathtt{d} \in \mathtt{D}$ to $(\mathtt{key}(\mathtt{d}), \mathtt{value}(\mathtt{d}))$. A programmer provides the mapper.
2. Shuffle: Moves all $\{(\mathtt{K}, \mathtt{v}_1), (\mathtt{K}, \mathtt{v}_2), ..., (\mathtt{K}, \mathtt{v}_m)\}$ to the same server. Note that all pairs have the same key as $\mathtt{K}$. This part is done by the heart of MapReduce framework automatically.
3. Reducer: Transforms $\{(\mathtt{K}, \mathtt{v}_1), (\mathtt{K}, \mathtt{v}_2), ..., (\mathtt{K}, \mathtt{v}_m)\}$ to an output listed below:

   $$\mathtt{D}'_{\mathtt{K}} = \mathtt{f}(\mathtt{v}_1, \mathtt{v}_2, ..., \mathtt{v}_m)$$

   A programmer provides the reducer. A reducer may generate any number of new (key, value) pairs to be consumed by another program. This program can be another MapReduce job or it can be a simple analyzer.

# 6 MapReduce Design Patterns

What is a design pattern? "In software engineering, a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. A design pattern is not a finished design that can be transformed directly into source or machine code."[15] MapReduce design patterns are common patterns in data-related problem solving. For example, the *singleton* pattern is a design pattern that restricts the instantiation of a class to one object. In MapReduce, *InMapper Combiner* is a design pattern, which does most of the `combine()` functionality inside the mappers. Another benefit of design patterns is that it makes the objective of code easier to understand and provides known scalability issues (if any), performance profiles and limitations of solutions.

For learning design patterns in software engineering, you should look at the "Gang of Four" book[3]. For learning design patterns in MapReduce, you should look at the "MapReduce Design Patterns"[8] by Donald Miner and Adam Shook and Data Algorithms[9].

Below are the partial list of MapReduce design patterns:

- Top-10
- Secondary Sorting
- Stripes
- InMapper Combining
- Join patterns (inner-join, left-join, ...)
- Summarization Patterns
- Filtering Patterns
- Data Organization Patterns

[15] Software design pattern – Wikipedia, the free encyclopedia: `https://en.wikipedia.org/wiki/Software_design_pattern`

# 7 Hadoop MapReduce Program Components

A typical MapReduce program for Hadoop platform will have the following components:

- Driver Program
- Mapper Class
- Reducer Class

It is possible to just write one big class and put all components in it; however, I will not recommend it. To make program components smaller, modular, and manageable, I usually generate at least three pieces of source code for a Hadoop MapReduce job:

## 7.1 Driver Program

The main function of a driver program is to identify input (comprised of set of files and directories) and output directory (comprised of one of more directories). The other function of a driver program is to plug-in the mapper and reducer by registering the mapper and reducer classes. A typical driver program may look like:

**Listing 2:** MapReduce Driver in Hadoop

```
1  public class MyMapReduceJobDriver  {
2     public static void main(String[] args) {
3        MyMapReduceJobDriver driver = new MyMapReduceJobDriver();
4        driver.run(args);
5     }
6
7     void run(String[] args) {
8        // prepare input/output and additional parameters
9        String input = args[0];    // "/dir1/my/input/path";
10       String output = args[1];   // "/dir2/my/output/path";
11
12       // create a Job
13       Job job = new Job(...);
14
15       // define input/output directories to MapReduce framework
16       FileInputFormat.addInputPath(job, new Path(input));
17       FileOutputFormat.setOutputPath(job, new Path(output));
18
19       // plug in your Mapper and Reducer classes
20       job.setMapperClass(MyMapperClass.class);
21       job.setReducerClass(MyReducerClass.class);
```

```
22
23        //submit your MapReduce job
24        job.submit();
25    }
26 }
```

## 7.2 Mapper Class in Hadoop

The `map()` function of a Mapper class maps input $(key_1, value_1)$ pairs to a set of (zero, one, or more) intermediate $(key_2, value_2)$ pairs. According to Hadoop[16]: "Maps are the individual tasks which transform input records into a intermediate records. The transformed intermediate records need not be of the same type as the input records. A given input pair may map to zero or many output pairs.". For example, to find bigrams[17] for a set of documents, your mapper's input will be a line of text or a sentence (comprised of any number words) and your output will be $((\texttt{word}_1, \texttt{word}_2), \texttt{1})$, where output key is $(\texttt{word}_1, \texttt{word}_2)$ and `1` is the associated value. Therefore, for each two words next to each other we emit those two words as a key and `1` as a value (to count frequency of bigrams).

**Listing 3:** Mapper Class in Hadoop

```
1  public class MyMapperClass extends ... {
2      // called once at the beginning of the map task (optional).
3      setup() {   ...
4      }
5
6      // called once for each key-value pair in the input split.
7      map(key, value){
8          ...
9      }
10
11     // called once at the end of the map task (optional).
12     cleanup() {   ...
13     }
14 }
```

The mapper's `setup()` can be used to initialize and set up some global resources (such as file handles, cache data structures, database connection objects) to be used by the `map()`. While `cleanup()` method can be used to close and release resources defined by the `setup()` method.

[16] http://hadoop.apache.org/docs/r1.1.1/api/org/apache/hadoop/mapred/Mapper.html

[17] https://en.wikipedia.org/wiki/Bigram

## 7.3 Reducer Class in Hadoop

The `reduce()` function of a Reducer class reduces a set of intermediate values which share a `key` to a set of values (listed below). The input to a reducer will be (note that, in Hadoop, there is no order between values arriving at a reducer — you may use a "secondary sorting" technique to impose a sort order between reducer values):

$$(\texttt{key}, \{\texttt{value}_1, \texttt{value}_2, ..., \texttt{value}_n\})$$

Before the `reduce()` function is called, the MapReduce framework does perform three support functions behind the scenes (the shuffle and sort functions occur simultaneously):

- Shuffle: the Reducer copies the sorted output from each Mapper in the cluster. This function is provided by MapReduce framework.
- Sort: the framework merge sorts Reducer inputs by keys (since different Mappers may have output the same key). This function is provided by MapReduce framework.
- Secondary Sort: this is optional (if you want to sort the input values of a reducer – in Hadoop and Spark values of a reducer arrives unsorted). This can be achieved by a plug-in comparator classes (the secondary-sorting design pattern is explained in details in the first two chapters of Data Algorithms book[9]). This default implementation of this function is provided by MapReduce framework, but can be overridden by a programmer (as a plug-in class).

The output of the Reducer phase is a set of $(key_3, value_3)$ and is not re-sorted.

**Listing 4:** Reducer Class in Hadoop

```
1 public class MyReducerClass extends ... {
2     // called once at the start of the reduce task.
3     setup() {   ...
4     }
5
6     // This method is called once for each reduce key
```

```
7     reduce(key, values){
8        foreach (v : values) {
9           process(key, v) ...
10       }
11    }
12
13    // called once at the end of the reduce task.
14    cleanup() {   ...
15    }
16 }
```

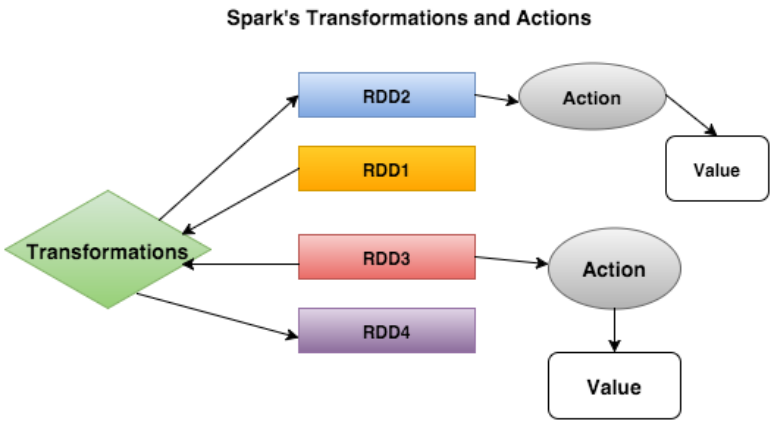
The reducer's `setup()` can be used to initialize and set up some global resources (such as file handles, cache data structures, database connection objects) to be used by the `reduce()`. While `cleanup()` method can be used to close and release resources defined by the `setup()` method.

# 8 Spark Program Components

Spark extends the popular MapReduce programming model to efficiently support more types of computations, including interactive queries, machine learning, graph algorithms, and stream processing. Since Spark has a higher abstraction layer API than Hadoop, A typical program for Spark platform can be expressed in a single driver class, which may include a series of transformations and actions (expressed as a DAG) and finally you may save any number of RDDs in the persistent layer (such as Linux file system or HDFS). Spark is a general-purpose framework for cluster computing, and is used for a diverse range of applications. For details on Spark programming model, refer to the "Learning Spark" book[4].

Figure 4 shows the Spark's Transformations and Actions.

Figure 4: Spark's Transformations and Actions




Below is a very simple Spark application (SparkErrorCount.java), which counts the number of `"error"` and `"exception"` keywords.

**Listing 5:** Sample Spark Program

```
1  import org.apache.spark.SparkConf;
2  import org.apache.spark.api.java.JavaSparkContext;
3  import org.apache.spark.api.java.function.Function;
4
5  public class SampleSparkProgram {
6    public static void main(String[] args) {
7      String logFile = "/home/mp/logs.txt"; // Should be some file on your system
8      SparkConf conf = new SparkConf().setAppName("count errors");
9      JavaSparkContext context = new JavaSparkContext(conf);
10     JavaRDD<String> logData = context.textFile(logFile).cache();
11
12     long numOfErrors = logData.filter(new Function<String, Boolean>() {
13       public Boolean call(String s) { return s.contains("error"); }
14     }).count();
15
16     long numOfExceptions = logData.filter(new Function<String, Boolean>() {
17       public Boolean call(String s) { return s.contains("exception"); }
18     }).count();
19
20     System.out.println("errors: " + numOfErrors + ", exceptions: " + numOfExceptions);
21
22     // done
23     context.close();
24   }
25 }
```

## 8.1 Working with Key-Value Pairs

Spark provides convenient structures for expressing key-value pairs. The foundation for key-value pairs are `scala.Tuple<N>` objects. For example, to have a composite value, which has 3 fields, you may specify it as `Tuple3<String, Integer, Integer>` and to specify a key with 2 fields, you may write it as `Tuple2<String, String>`. For example, in Java we may write:

```
Tuple2<String,String> k2 =
    new Tuple2<String,String>("s1", "s2");
Tuple3<String, Integer, Integer> v3 =
    new Tuple3<String, Integer, Integer>("a", "b", 2);
```

You may define your own custom key or value types by defining a class, which implements the `java.io.Serializable` interface, for example, to define a custom value class, we may write:

```
public class MyCustomValue implements java.io.Serializable {
   int id;
   String name;
   String address;
   char gender;
   <your-desired-methods>
}
```

Then, you may use `MyCustomValue` as a key or value for your Spark programs:

```
JavaSparkContext context = new JavaSparkContext();
JavaRDD<String> lines = context.textFile("/dir/data.txt");
JavaPairRDD<String, MyCustomValue> pairs = lines.mapToPair(...);
```

## 8.2 Transformations and Actions

Spark has two major types of operations:

- Transformations: transformations return a new, modified RDD based on the original. Some of the available transformations are: `foldByKey()`, `map()`, `filter()`, `sample()`, and `union()`. A transformation is a lazy (not computed immediately) operation on an RDD which returns another RDD)

- Actions: actions return a value based on some computation being performed on an RDD. Some of the available actions are `countByKey()`, `reduce()`, `count()`, `first()`, and `foreach()`.

Using Spark's transformations and actions, you may create any complex DAG, which may solve any MapReduce problems and beyond.

# 9 Character Count in MapReduce Hadoop

To demonstrate the MapReduce implementation concepts in Hadoop, we solve counting the characters for a set of documents. In simple terms, we want to find out the frequency of each unique character for a given corpus. Two MapReduce Hadoop solutions are provided for Character Count by using:

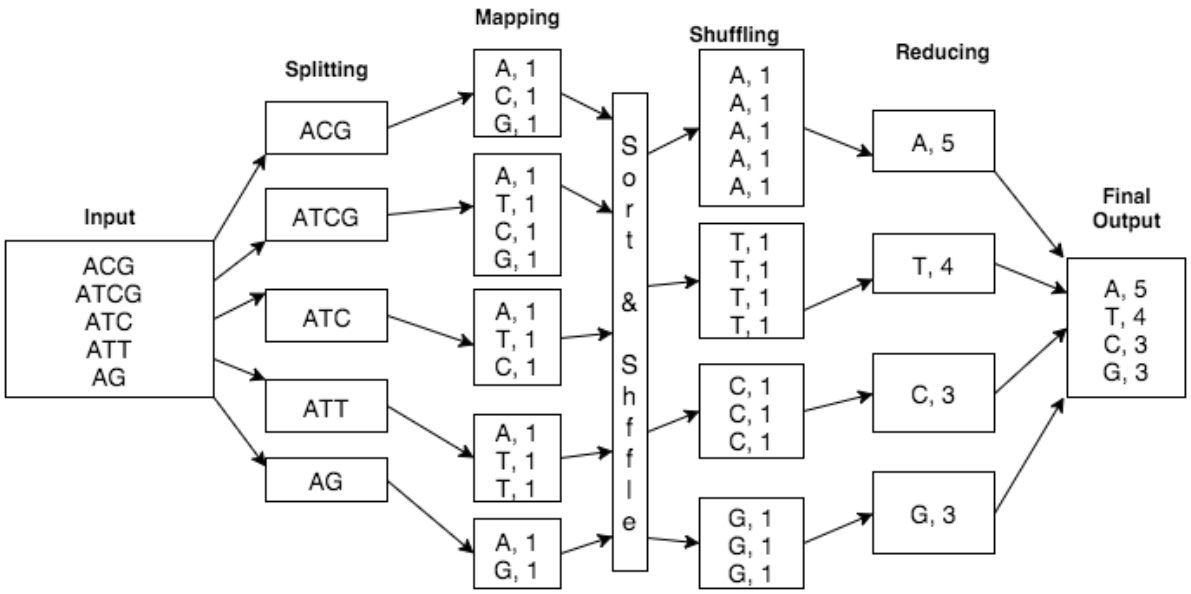
- Solution-1: Basic MapReduce Design Pattern
- Solution-2: *InMapper Combining* Design Pattern

## 9.1 Basic MapReduce Design Pattern

To count the characters, for each record of input, we split it into a set of words, and then we split each word into a character array, and finally we emit a `(K,V)` pair, where `K` is a single character from a character array and `V` is `1` (frequency count of one). Since we did not use any custom data types for emitting key-value pairs, we call this the "Basic" MapReduce design pattern. The reducer sums up the frequencies of a single unique character.

The overall MapReduce character count process is illustrated in Figure 5.

Figure 5: Character Count MapReduce Algorithm




### 9.1.1 Mapper

The Mapper implementation, via the `map()` method, processes one line at a time, as provided by the MapReduce input partitioner. It then splits the line into tokens separated by white spaces, via the `split()`, and then splits each word into a character array, and finally emits a key-value pair of `<<character>, 1>`.

**Listing 6:** map() for Character Count

```
1  // K is system assigned, ignored here
2  // V is a single record of input
3  map(K, V) {
4     String[] words = V.split(" ");
5     for (String word : words) {
6        char[] arr = word.toLowerCase().toCharArray();
7        for (char c : arr) {
8           emit(c, 1);
9        }
10    }
11 }
```

### 9.1.2 Combiner

The combiner function (as an optional function in Hadoop) is used as an optimization for the MapReduce job. The combiner function uses the output of the mapper phase and is used as a filtering or an aggregating step to lessen the number of intermediate keys that are being passed to the reducer. Hadoop implementation injects the combiner function by:

```
Job job = ...;
job.setCombinerClass(CharCountCombiner.class);
```

Note that in Hadoop, there is no function called `combine()`. In Hadoop, in most of the cases the reducer class is set to be the combiner class. You can apply combiners if and only if the output of mappers form monoids (for details on combiners and monoids, see [9] and [5]. Since the output of mappers for the "character count" forms a monoid (addition operator over a set of integers form a monoid structure) then we can write a `combine()` function. Therefore, combiner acts as a local reducer and the output of the `combiner()` is the intermediate data that is passed to the `reducer()`.

**Listing 7:** combine() for Character Count

```
1 // K is a single unique character
2 // V is a Iterable<Integer>
3 combine(K, V) {
4    Integer sum = 0;
5    for (Integer i : V) {
6       sum += i;
7    }
8    emit(K, sum);
9 }
```

Therefore, combiners reduce the amount of intermediate data that is shuffled across the network, which can improve the performance of MapReduce jobs.

### 9.1.3 Reducer

The Reducer implementation, via the `reduce()` method, processes one unique key at a time, as provided by the MapReduce sort and shuffle functions. It sums up the frequencies for a given single unique character.

**Listing 8:** reduce() for Character Count

```
// K is a single unique character
// V is a Iterable<Integer>
reduce(K, V) {
   Integer sum = 0;
   for (Integer i : V) {
      sum += i;
   }
   emit(K, sum);
}
```

### 9.1.4 Hadoop Implementation

The Hadoop implementation is provided by 3 basic Java classes (defined in the `org.dataalgorithms.bonus.charcount.mapreduce.basic` package):

- `CharCountDriver`
- `CharCountMapper`
- `CharCountReducer`

## 9.2 InMapper Combining Design Pattern

There is another way to emit the frequency of characters: build a hash table of `Map<Character,Integer>` from the characters of given mapper input partition and then emit the `(K,V)` pairs, where `K` is `Map.Entry.getKey()` and `V` is `Map.Entry.getValue()`. Therefore, the mapper phase will emit a `(K,V)` pair, where `(K,V)` is an entry of built hash table. In the *InMapper Combining* version of mapper, we use a hash table to keep track of frequencies of all unique characters. Note that the `map()` function just updates the frequencies table and does not emit any key-value pairs at all. After the mapper completes, the `cleanup()` is executed, which will emit all key-value pairs from the frequencies table. The reducers will sum up the frequencies and find the final count of characters. Not that the mapper using Hash Table (called the *InMapper Combining* design pattern – since we are performing the `combine()` inside mappers) is more efficient because it will not emit too many key-value pairs, which improve efficiency in network by sending less data.

### 9.2.1 Mapper

It is clear that the "Basic" MapReduce algorithm generates huge number of key-value pairs compared to the *InMapper Combining* design pattern. The *InMapper Combining* data structure representation is much more compact, since every entry of `Map<Character,Integer>`is equivalent to N basic key-value pairs, where N is equal to `Map.Entry.getValue()`. The *InMapper Combining* approach also generates fewer and shorter intermediate keys, and therefore the execution framework has less sorting to perform. In using the *InMapper Combining* design pattern, you need to be careful on the size of hash table to make sure it will not cause bottlenecks. For Character Cont problem, the size of hash table for each mapper will be very small (since we have a limited number of unique characters), therefore there is no performance bottleneck for the presented solution.

**Listing 9:** map() for Character Count

```
1  public class CharCountInMapperCombinerMapper {
2
3     Map<Character, Integer> map = null;
4
5     // called once at the beginning of the map task.
6     setup() {
7        map = new HashMap<Character, Integer>
8     }
9
10    // K is system assigned, ignored here
11    // V is a single record of input
12    // called once for each (K, V) pair in the input split.
13    map(K, V) {
14       String[] words = V.split(" ");
15       for (String word : words) {
16          char[] arr = word.toLowerCase().toCharArray();
17          for (char c : arr) {
18             Integer count = map.get(c);
19             if (count == null) {
20                map.put(c, 1);
21             }
22             else {
23                map.put(c, count+1);
24             }
25          }
26       }
27    }
28
29    // called once at the end of the map task.
30    cleanup() {
31       for (Map.Entry<String, String> entry : map.entrySet()) {
32          key = entry.getKey();
33          value = entry.getValue();
```

```
34          emit (key, value);
35       }
36    }
37
38 }
```

### 9.2.2 Reducer

The Reducer implementation, via the `reduce()` method just sums up the values, which are the frequency counts for each key (i.e. unique characters).

**Listing 10:** map() for Character Count

```
1  public class CharacterCountReducer {
2
3     // K is a unique character
4     // V is a Iterable<Integer>
5     reduce(K, V) {
6        int sum = 0;
7        for (Integer i : V) {
8           sum += i;
9        }
10       emit (K, sum);
11    }
12
13 }
```

### 9.2.3 Hadoop Implementation

The Hadoop implementation is provided by 3 basic Java classes (defined in the `org.dataalgorithms.bonus.charcount.mapreduce.inmapper` package):

- `CharCountInMapperCombinerDriver`
- `CharCountInMapperCombinerMapper`
- `CharCountInMapperCombinerReducer`

## 10 Character Count in Spark

Since Spark has a very rich and expressive MapReduce API, I will present 2 distinct solutions (each using a different design pattern) using different set of APIs and algorithms:

- "Basic" Design Pattern

- *InMapper Combining* Design Pattern

## 10.1 Using "Basic" Design Pattern

This solution is implemented by the `CharCount`[18] class. This solution just emits pairs of `(K,V)` where `K` is a single character and `V` is `1` (counting of a single character), then we use `reduceByKey()` to aggregate all frequencies for a unique single character. This solution uses the basic MapReduce design pattern (it just emits `(K,V)` pairs where `K` and `V` are just basic data types). The mapper breaks up the input into each individual tuple it finds and emits proper `(K,V)` pairs.

## 10.2 Using InMapper Combining Design Pattern

This solution is implemented by the `CharCountInMapperCombiner` [19] class. This solution uses the *InMapper Combining* design pattern. In MapReduce, there are at least two basic general design patterns: "Pairs" and *InMapper Combining*. For this solution, our focus will be on the *InMapper Combining* design pattern. The *InMapper Combining* design pattern uses in-mapper memory data structures to group together pairs into an associative array (we implement this associative array in Java as `Map<Character, Integer>`). Note that, both the *InMapper Combining* and Pairs design patterns will calculate the same output, but they differ on their generation of (K,V) intermediate values. The *InMapper Combining* algorithm does some preprocessing within the mapper before emitting its values to the reducer. The *InMapper Combining* algorithm is faster, while the pairs mapper scales to problems that have a large number of unique data key tuples. The *InMapper Combining* design pattern reduces the amount of data that needs to be transferred between the mapper and reducer. However, the *InMapper Combining* algorithm may runs into a problem if the number of unique keys grows too large for the associative array to fit in memory. If this is the case you will have to revert to the basic Pairs design pattern approach. Note that with *InMapper Combining*, the mappers will generate only those key-value pairs that need to be shuffled across the network to the reducers.

What are the scalability bottlenecks for the *InMapper Combining* design pattern? Using the *InMapper Combining* design pattern, we make the assumption that, at any point in time, each associative array (per mapper

[18] org.dataalgorithms.bonus.charcount.spark.basic.CharCount
[19] org.dataalgorithms.bonus.charcount.spark.inmapper.CharCountInMapperCombiner

partition) (i.e., `Map<Character, Integer>`) is small enough to fit into memory – otherwise, memory paging will significantly impact performance. For character count, the size of the associative array (per mapper partition) is bounded by the number of unique characters. Therefore, for character count problem, there is no scalability bottlenecks by using the *InMapper Combining* design pattern.

What are the advantages and disadvantages of the *InMapper Combining* design pattern? The *InMapper Combining* has the following advantages and disadvantages:

- Advantages
  - Far less sorting and shuffling of key-value pairs
  - Can make better use of combiners
- Disadvantages
  - More difficult to implement
  - Underlying object (per mapper partition) is more heavyweight
  - Fundamental limitation in terms of size of the underlying object (for the character count problem: an associative array per mapper partition)

Using the *InMapper Combining* design pattern, the Spark program presented here finds the frequency of each unique character for a given corpus. The data structure `Map<Character, Integer>` (as an associative array) is used to implement the *InMapper Combining* design pattern and it will not cause scalability issues since the number of unique characters are limited, therefore size of the associative array per map partition is small enough to fit in memory. Therefore, the *InMapper Combining* design pattern scales out well for this solution. But if you have too many unique keys, then the *InMapper Combining* design pattern might not be a good scalable solution (might have OOM problems).

Since Spark has a much higher abstraction API layer than Hadoop, a single Java driver class is used to present the entire solution. Our Spark solution is presented as a set of actions and transformations:

Step-1: handle input/output parameters

Step-2: Read input corpus and a create `JavaRDD<String>`, where each RDD element is a single record of input

Step-3: Use `mapPartitions()`, such that each partition creates a `Map<Character,Integer>`, where `Map.Entry.key` is character and `Map.Entry.value` is the frequency of that character

Step-4: Finally we merge all these hash tables to create a single frequency table.

Step-5: The last step is to save the desired result

## 10.3 High-Level Steps

**Listing 11:** Spark High-Level Steps

```
1  public static void main(String[] args) throws Exception {
2
3     // Step-1: Handle input/output parameters
4     // Step-2: Create an RDD from input
5     // Step-3: Use mapPartitions() and create Map<Character,Integer>
6     // Step-4: Merge all these hash tables to create a single frequency table
7     // Step-5: Save the desired result
8
9     // close the context and we are done
10    context.close();
11 }
```

## 10.4 Step-1: Handle Input/Output Parameters

This step reads an input and output parameters.

```
// Step-1: handle input/output parameters
String inputPath = args[0];
String outputPath = args[1];
```

## 10.5 Step-2: Create an RDD from input

This step reads an input and creates `JavaRDD<String>`, where each RDD element is a single record of input.

```
// Step-2: Create an RDD from input

// first: create a context object, which is used
```

```
// as a factory for creating new RDDs
JavaSparkContext context = new JavaSparkContext();

// next: create an RDD from input
JavaRDD<String> lines = context.textFile(inputPath);
```

## 10.6 Step-3: Map Partitions

Use `mapPartitions()`, such that each partition creates a `Map<Character,Integer>`, where `Map.Entry.key` is character and `Map.Entry.value` is the frequency of that character.

**Listing 12:** Creating the Associate Array Data Structures

```
// JavaRDD<U> mapPartitions(FlatMapFunction<java.util.Iterator<T>,U> f)
JavaRDD<Map<Character,Integer>> partitions = lines.mapPartitions(
    new FlatMapFunction<Iterator<String>, Map<Character,Integer>>() {
    @Override
    public Iterable<Map<Character,Integer>> call(Iterator<String> iter) {
        Map<Character,Integer> map = new HashMap<Character,Integer>();
        while (iter.hasNext()) {
           String record = iter.next();
           String[] words = record.split(" ");
           for (String word : words) {
              char[] arr = word.toCharArray();
              for (char c : arr) {
                 updateAndIncrement(map, c)
              }
           }
        }
        return Collections.singletonList(map);
    }
});
```

The helper method is defined below:

**Listing 13:** updateAndIncrement() Method

```
private static void updateAndIncrement(Map<Character,Integer> map, char c) {
   Integer count = map.get(c);
   if (count == null) {
      map.put(c, 1);
   }
   else {
      map.put(c, count+1);
   }
}
```

## 10.7 Step-4: Merge Frequencies

Finally we merge all these hash tables to create a single frequency table.

## 10.8 Step-5: Save Result

The last step is to save the desired result

## 10.9 Sample Run Using InMapper Combining

### 10.9.1 Input

We used a small input text file (243,373,751 bytes) to run the script (to make sure program works!).

**Listing 14:** Hadoop Input

```
# hadoop fs -ls /charcount/input/
Found 1 items
-rw-r--r-- 1 ...  243373751 ... /charcount/input/simulated_data.txt
```

### 10.9.2 The Shell Script

**Listing 15:** Script to Run Spark Program

```
# define the installation dir for hadoop
export HADOOP_HOME=/Users/mparsian/zmp/zs/hadoop-2.5.0
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export HADOOP_HOME_WARN_SUPPRESS=true
#
export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk1.7.0_60.jdk/Contents/Home
export BOOK_HOME=/Users/mparsian/zmp/github/data-algorithms-book
export SPARK_HOME=/Users/mparsian/spark-1.5.0
#export SPARK_MASTER=spark://localhost:7077
export SPARK_JAR=$BOOK_HOME/lib/spark-assembly-1.5.0-hadoop2.6.0.jar
export APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
# defines some environment for hadoop
source $HADOOP_CONF_DIR/hadoop-env.sh
#
# build all other dependent jars in OTHER_JARS
JARS=`find $BOOK_HOME/lib -name '*.jar'  ! -name '*spark-assembly-1.5.0-hadoop2.6.0.jar' `
OTHER_JARS=""
for J in $JARS ; do
   OTHER_JARS=$J,$OTHER_JARS
done
#
# define input/output for Hadoop/HDFS
INPUT=/charcount/input
```

```
OUTPUT=/charcount/output
#
# remove all files under input
$HADOOP_HOME/bin/hadoop fs -rmr $OUTPUT
#
# remove all files under output
driver=org.dataalgorithms.bonus.charcount.spark.inmapper.CharCountInMapperCombiner
$SPARK_HOME/bin/spark-submit --class $driver \
        --master yarn-cluster \
        --jars $OTHER_JARS \
        --conf "spark.yarn.jar=$SPARK_JAR" \
        $APP_JAR $INPUT $OUTPUT
```

### 10.9.3 Output

**Listing 16:** Generated Output

```
        @ : 2
        _ : 1
        f : 4416
        g : 2399
        d : 3473
        e : 4414
        b : 3484
        ' : 1
        c : 8188
        a : 8120
        n : 3463
        o : 4208
        l : 6697
        m : 4272
        . : 19634079
        j : 563
        k : 1798
        / : 2
        h : 2086
        - : 9816844
        i : 2183
        3 : 18143746
        w : 285
        v : 340
        2 : 18658461
        u : 817
        1 : 23673218
        0 : 29738144
        t : 3876
        s : 4949
        7 : 17381045
        r : 5715
        6 : 17254695
        q : 159
        5 : 17588154
```

```
        p : 6762
        4 : 17738009
        9 : 16765751
        8 : 17238669
        z : 915
        y : 694
        x : 1016
```

## 11 Source Code

All source code for this chapter are posted at GitHub:

`https://github.com/mahmoudparsian/data-algorithms-book/`

## 12 Comments and Suggestions

Your comments and suggestions are always welcome and I am sure your comments will improve the readability and accuracy of this fundamental chapter. I can be reached at: mahmoud.parsian@yahoo.com.

## 13 Acknowledgement

I would like to express my sincere gratitude to Mr. Amir Bahmani[20] for reviewing this chapter and providing useful suggestions, which improved the quality of this chapter.

[20] `http://www4.ncsu.edu/~abahman/`

## References


[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. 2008.

[2] Jr. Frederick P. Brooks. No silver bullet – essence and accident in software engineering.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[4] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning Spark*. O'Reilly Media, 2014.

[5] Jimmy Lin. Monoidify! monoids as a design principle for efficient mapreduce algorithms. 29 April 2013.

[6] Jimmy Lin. Mapreduce is good enough? pages 28–37, March 2003.

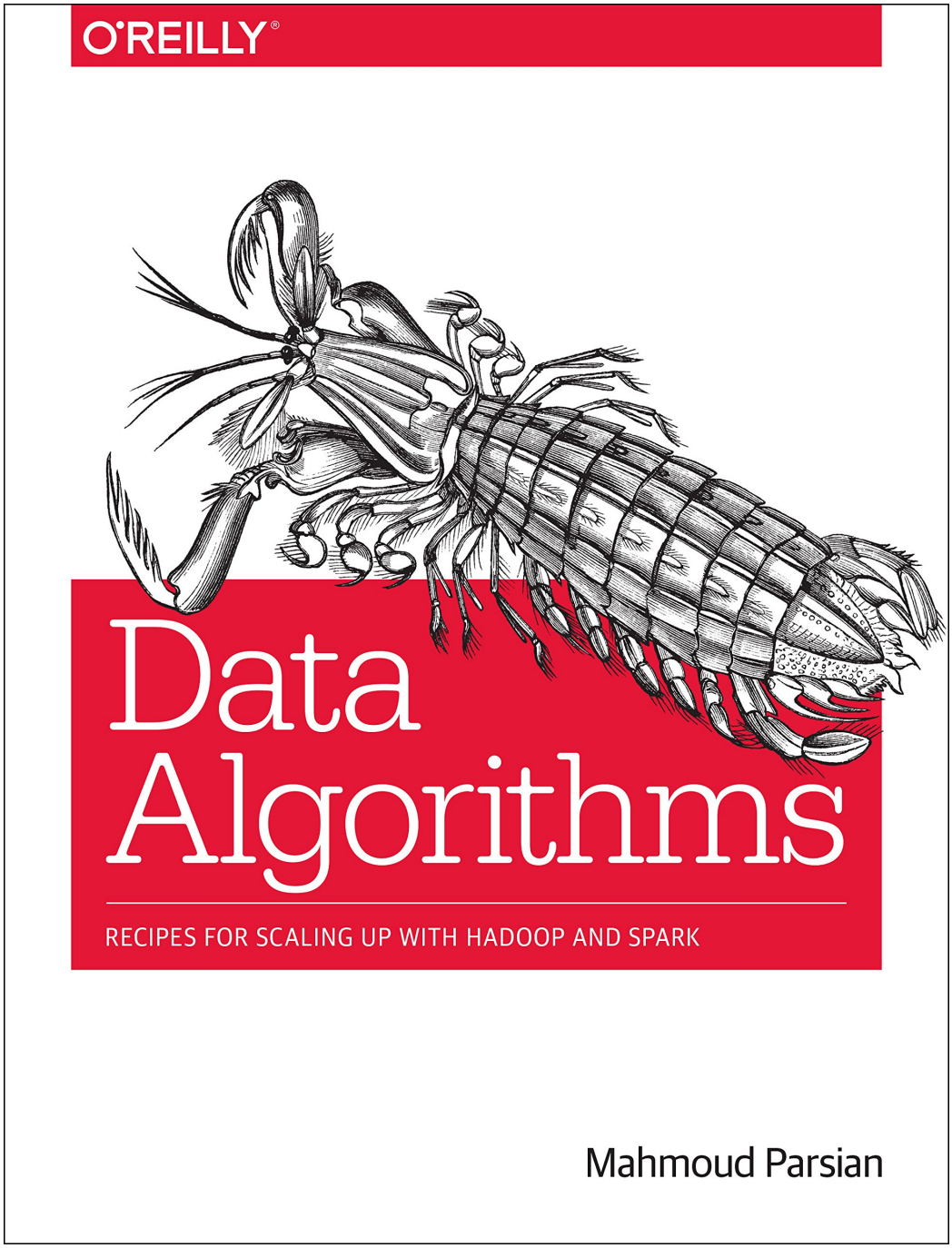
[7] Jimmy Lin and Chris Dyer. Data-intensive text processing with mapreduce. `https://lintool.github.io/MapReduceAlgorithms/ed1n/MapReduce-algorithms.pdf/`, 2013.

[8] Donald Miner and Adam Shook. *MapReduce Design Patterns*. O'Reilly Media, Inc., 2013.

[9] Mahmoud Parsian. *Data Algorithms*. O'Reilly Media, 2015.

[10] Matei Zaharia. Spark: cluster computing with working sets. 2010.

[11] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. 2012.


O'REILLY®
Data
Algorithms
RECIPES FOR SCALING UP WITH HADOOP AND SPARK
Mahmoud Parsian