Introduction to Hadoop

Big Data Analysis

What is Hadoop?

Hadoop is an open-source framework for developing and executing distributed applications that process large quantities of data. Hadoop is known for being:

- **Accessible**: The Hadoop framework leverages large clusters of commodity machines either "on-premise" or in the cloud.
- **Robust**: Since the Hadoop framework leverages commodity hardware, the framework is architected to gracefully handle most hardware failures.
- **Scalable**: Hadoop scales linearly by adding more nodes to the cluster in order to handle large volumes of data.
- **Efficient**: The Hadoop framework utilizes the *MapReduce* paradigm (i.e., a divide-and-conquer approach) to quickly and efficiently execute data processing tasks across a cluster of nodes in parallel.

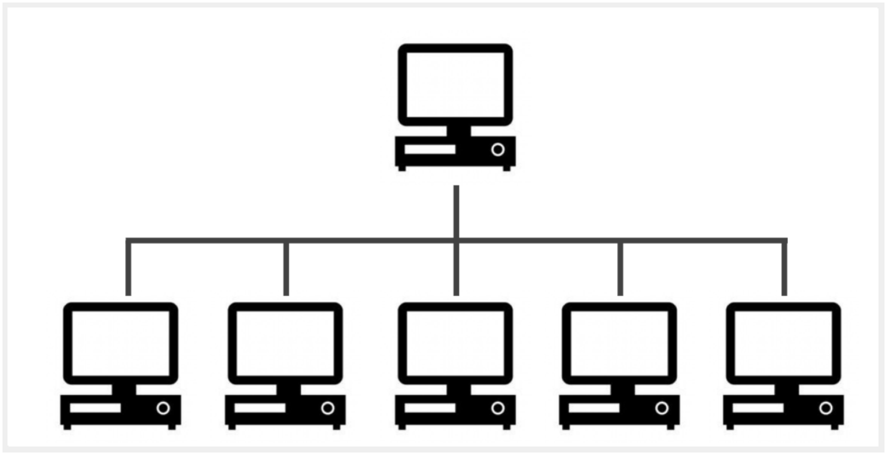


Figure 1: Hadoop cluster

A Hadoop cluster (Figure 1) is composed of many machines that store and process large quantities of data in parallel. A client application sends data processing jobs into this cluster of computers, and the tasks that constitute the job are executed across the nodes. The results are then sent back to the client application once the processing is complete.

Building Blocks of Hadoop

The two main components of Hadoop are:

1) **Hadoop Distributed File System (HDFS)**: HDFS is the *data repository* that stores all the data that is processed by the Hadoop application.

2) **MapReduce**: MapReduce is the *data processing paradigm* that is leveraged by a Hadoop application to process the data stored in HDFS.

Collectively, HDFS and MapReduce provide support for distributed storage and distributed computation within a Hadoop cluster.

Hadoop Distributed File System (HFDS)

HDFS is a filesystem that is oriented towards large-scale distributed data processing. Big data sets (e.g., 100 TB) can be stored in HDFS as a single file. When a file is stored in HDFS, it is split into data blocks (Figure 2). These data blocks are replicated across multiple machines and across a cluster of nodes called *DataNodes*.

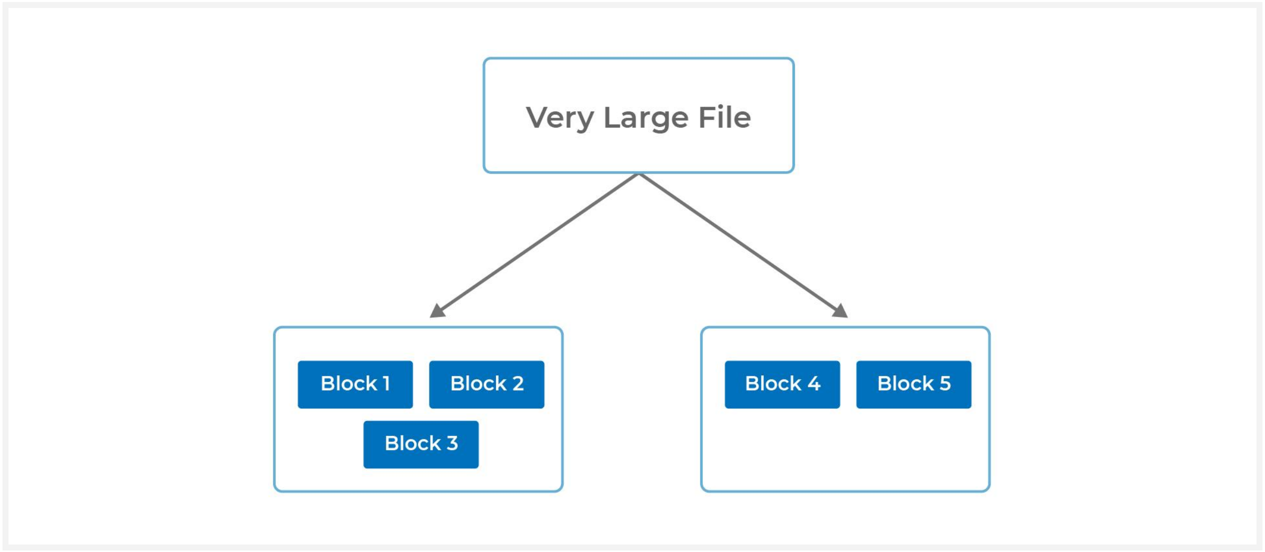


Figure 2: Hadoop data distribution

DataNodes and NameNodes

A master node, called *NameNode*, keeps track of the file metadata, i.e., which data blocks make up a file and which DataNodes store those data blocks. The NameNode monitors the health of the overall distributed file system. In contrast, the DataNodes hold the actual data blocks. A DataNode communicates with other DataNodes to replicate its data blocks for redundancy, and it constantly reports back to the NameNode to keep the metadata current.

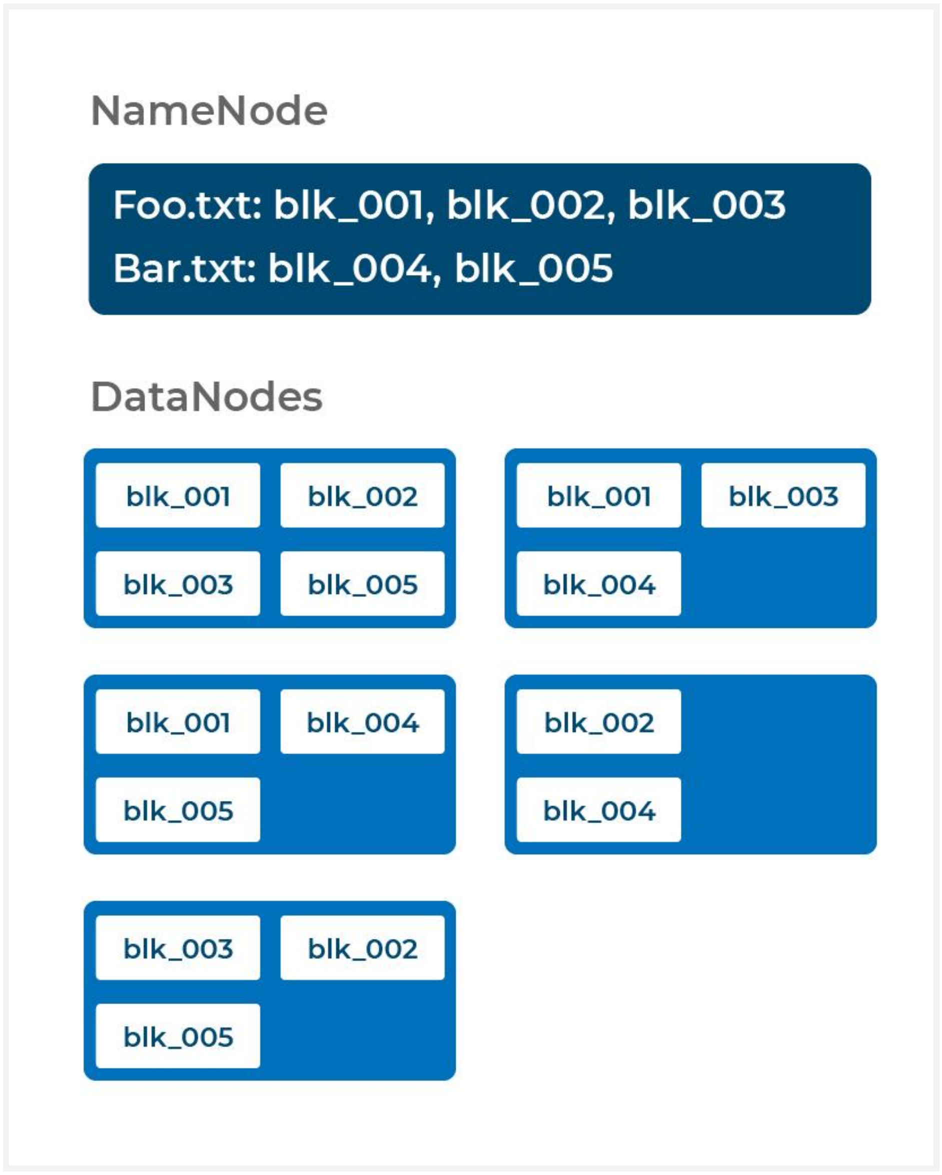


Figure 3: Constructors of the NameNode and DataNodes

The above diagram (Figure 3) shows the constructors of the NameNode and DataNodes, respectively. The NameNode holds the metadata for the two files (i.e., Foo.txt and Bar.txt), and the DataNodes holds the actual blocks - each data block will be 64MB or 128MB in size, and each block is replicated three times on the cluster.

Secondary NameNode (SNN)

A separate construct known as the *Secondary NameNode* (SNN) acts as an assistant to the NameNode for monitoring the state of the Hadoop cluster. Similar to the NameNode, each cluster has one SNN which typically resides on its own machine (Figure 4).

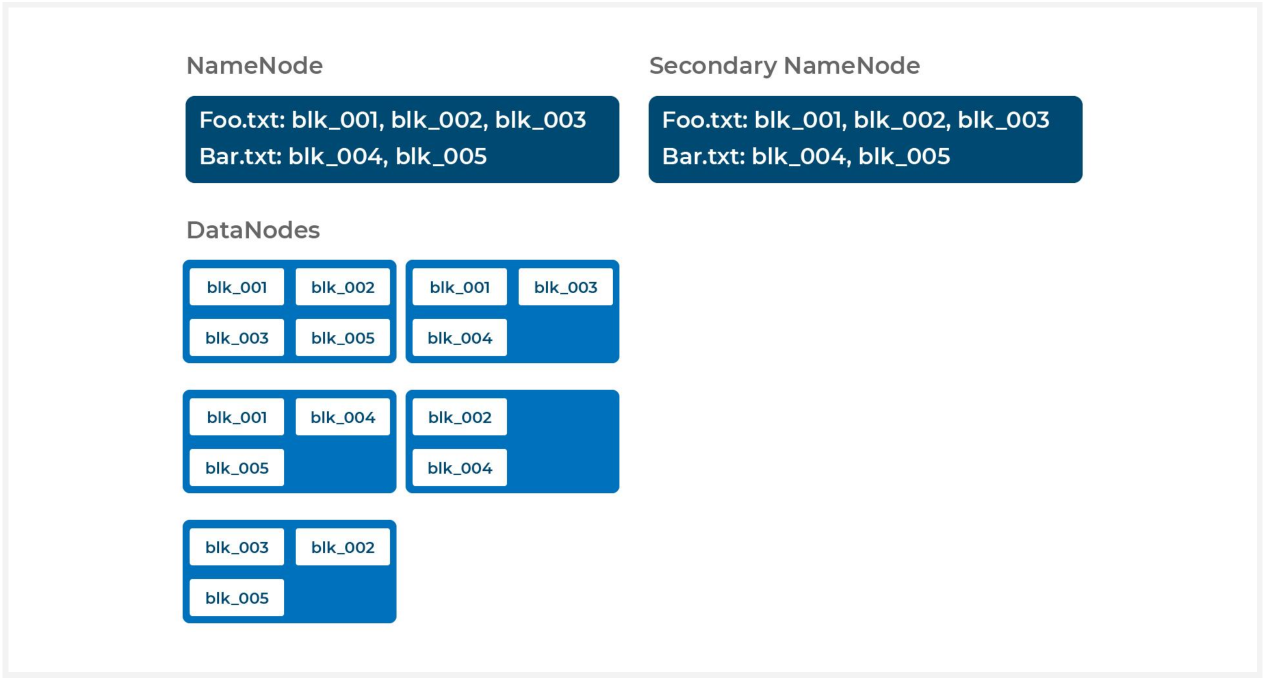


Figure 4: Constructors of the NameNode, DataNodes, and Secondary NameNode

The SNN differs from the NameNode in that the SNN does not receive or record any updates to HDFS. Instead, the SNN captures the information of the HDFS metadata at intervals defined in the cluster configuration. The information captured by the SNN helps minimize downtime and data loss in case the NameNode malfunctions.

JobTracker and TaskTracker

In addition to the NameNode, SecondaryNode and DataNode, there are two additional constructs called *JobTracker* and *TaskTracker* that orchestrate the processing and execution of jobs and associated tasks within a Hadoop cluster. The JobTracker is the master overseeing the overall execution of a MapReduce job, and the TaskTracker handles the execution of individual tasks on each slave node (Figure 5).

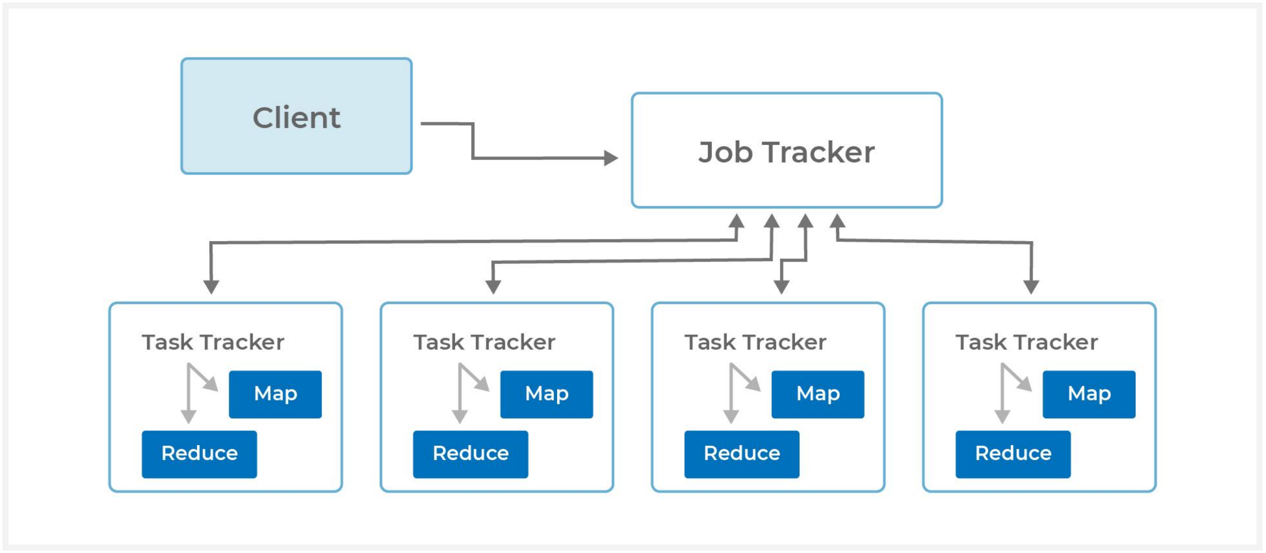


Figure 5: JobTracker and TaskTracker organization

Once an application client submits code to the cluster, the JobTracker determines the execution plan in this way:

1. Determines which file(s) need to be processed
2. Assigns nodes to the different tasks
3. Monitors all tasks that are running

If a specific task fails, the JobTracker will automatically try relaunching the failed task up to a predefined limit of retries.

The TaskTracker executes the individual tasks that the JobTracker assigns. After a client application calls the JobTracker to commence a data processing job, the JobTracker assigns different map-and-reduce tasks to each TaskTracker in the cluster. Note that there is a single TaskTracker per node. However, every TaskTracker can spawn multiple Java Virtual Machines (JVM) to support many map-and-reduce tasks in parallel.

Interacting with HDFS

When interfacing with the HDFS, there is a consistent command structure that needs to be adhered to:

```
hadoop fs -cmd <args>
```

- *`hadoop fs`* - refers to the Hadoop filesystem.
- *`cmd`* - is the specific file command that needs to be executed.
- *`<args>`* - is the optional set of arguments that provides greater specificity to *`cmd`*.

In addition, please observe the following commands.

List Files in the Current Directory

The following command will list the files in the current directory.

```
hadoop fs -ls
```

The common file management tasks in Hadoop revolve around adding files/directories, accessing files, and removing files. Before a program can be executed within the Hadoop framework, data needs to be deposited into HDFS first. The file containing the data will be moved into a working directory in HDFS, but first, a working directory will need to be created.

Create a Working Directory

Use the following command to create the */user/leo* working directory.

```
hadoop fs -mkdir /user/leo
```

Move a File into HDFS

Let's say the data that needs to be ingested into HDFS is in a file called *shakespeare.txt*. Use the following command to move the file containing the data into HDFS.

```
hadoop fs -put shakespeare.txt /user/leo
```

Examine Entire File Contents

Use the following command to examine the entire contents of the file once it is in HDFS.

```
hadoop fs -cat /user/leo/shakespeare.txt
```

Display the First Few Lines of a File

To display only the first few lines from a file, use the following command.

```
hadoop fs -cat /user/leo/shakespeare.txt| head
```

Retrieve Files

Once the file is in HDFS, use the following command to retrieve it into your local file system.

```
hadoop fs -get /user/leo/shakespeare.txt
```

Remove/Delete Files

Use the following command to remove/delete files from HDFS.

```
hadoop fs -rm /user/leo/shakespeare.txt
```

MapReduce

MapReduce is a data processing model. The biggest advantage it affords is the easy scaling of data processing jobs across a cluster of nodes. Under the MapReduce model, the data processing constructs are referred to as “mappers” and “reducers.” While deconstructing a data processing job into mappers and reducers requires careful code design, once the code is in MapReduce form, scaling for execution over thousands of machines is as easy as making a configuration change. This advantageous feature has attracted many developers to the MapReduce model.

Basics of a MapReduce Flow

MapReduce programs are designed to compute large quantities of data in a parallel fashion. As the name suggests, MapReduce consists of two phases: 1) *map*, and 2) *reduce*.

The Mapper Function

A list of input data elements is submitted to the *mapper* function which in turn outputs a list of data elements. Essentially, data is fed to the mapper function and the mapper function outputs data. When the mapper function is given a list of key→value pairs, it returns a list of other key→value pairs.

Let’s say there is a need to convert all text in a document/file to uppercase. The input to the mapper function would be the data elements that constitute the file, and the output from the mapper function would be the data elements in upper case.

The mapper function creates a new output list by applying a function to the individual elements of an input list (Figure 6).

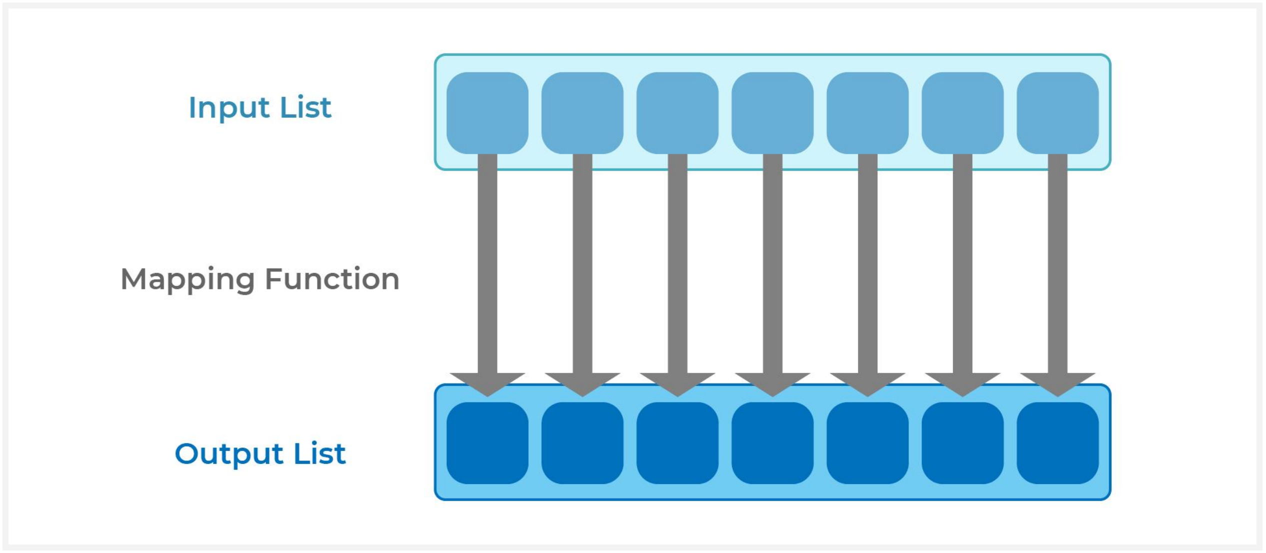


Figure 6: Model of mapping function

The Reducer Function

The *reducer* function helps aggregate values together. A reducer function receives an iterator of input values from the mapper function, and it combines the values together to generate a single output value (Figure 7).

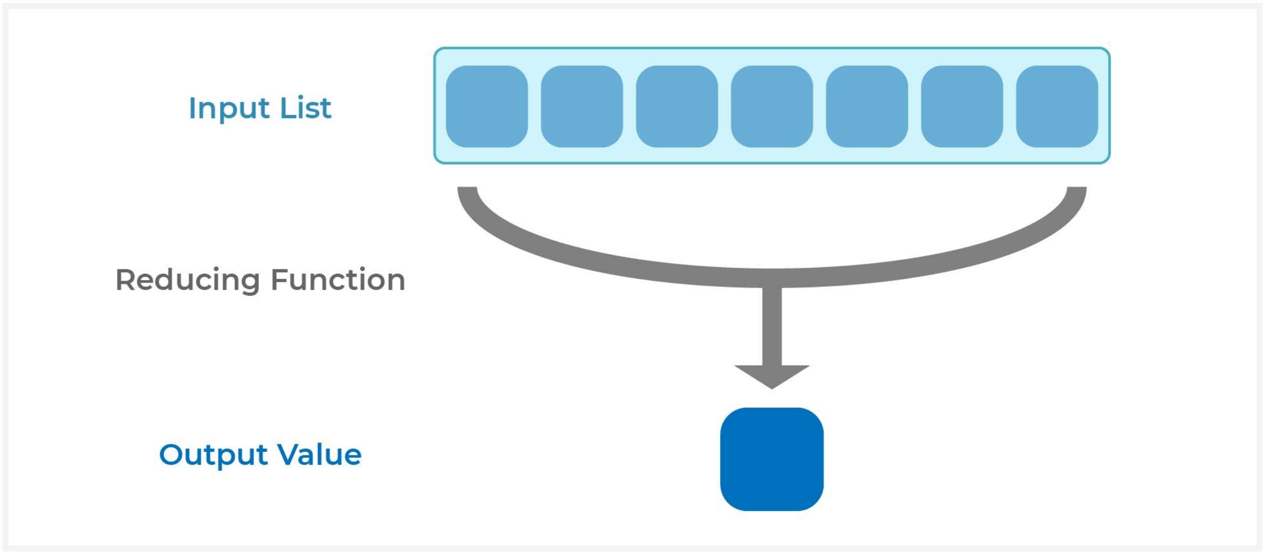


Figure 7: Model of reducing function

The reducer function is often used to produce summary/aggregate data. For example, a reducer function can be utilized to generate the sum of a list of input values, i.e., for all different key→value pairs output by the mapper function, and the reducer function will return an aggregate output value for every single key that the mapper function has output.

The chart (Figure 8) below illustrates the MapReduce data flow at a high-level. The details of the flow will be covered in more detail in the subsequent module.

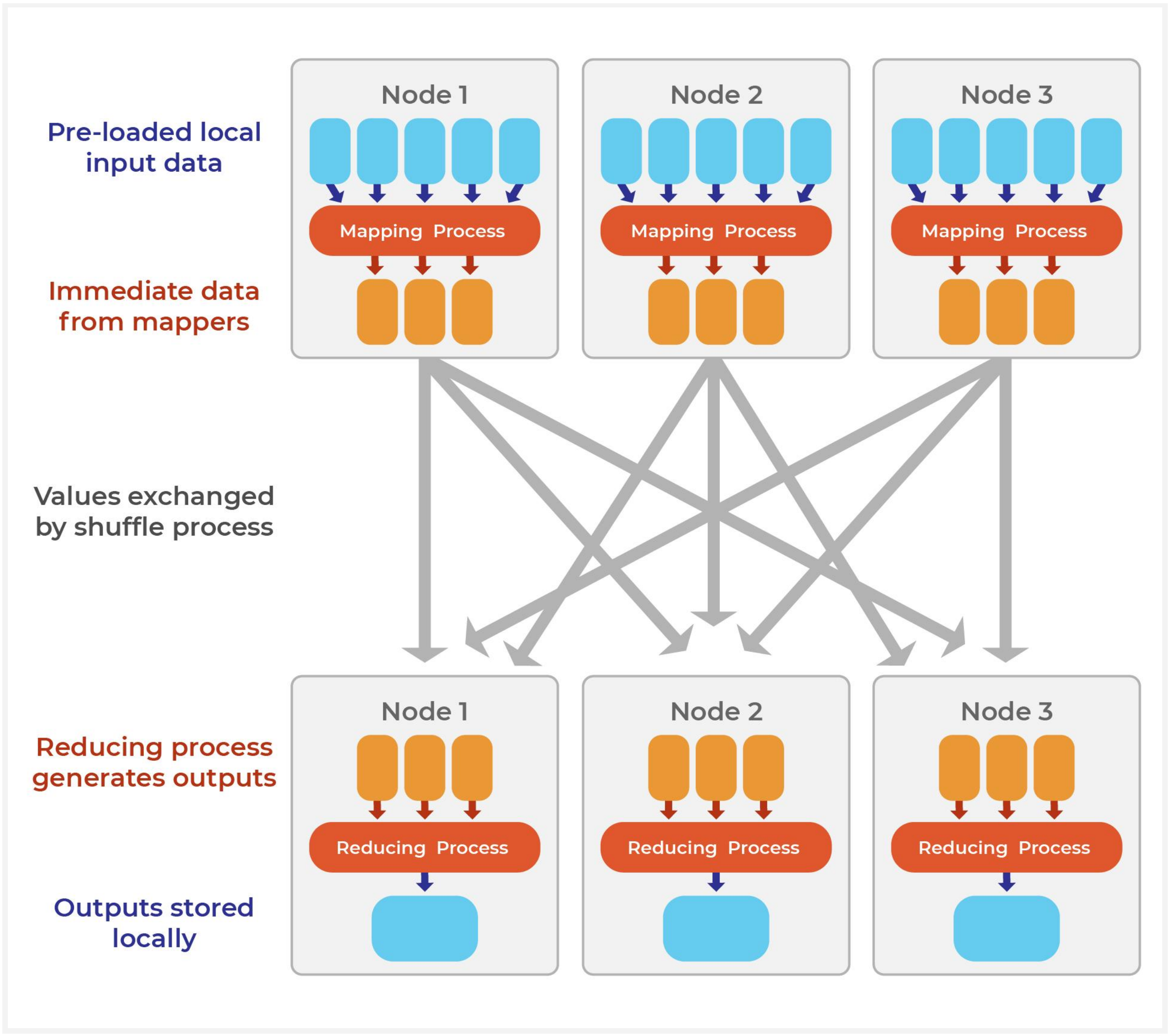


Figure 8: MapReduce data flow at a high-level

Steps of Flow

The overall flow can be distilled into the following steps:

- **Step 1**: The input data is distributed to nodes across the cluster.
- **Step 2**: Each map task operates on a split of the data within its own node.
- **Step 3**: The mapper function outputs intermediate values, i.e., given a set of key→value pairs, it outputs a set of other key→value pairs.

- **Step 4**: The data is exchanged between nodes in a "shuffle" process. Note that after distributing input data to different nodes, the only other occasion that nodes communicate with each other during the MapReduce flow is at the "shuffle" step.
- **Step 5**: All values with the same key go to the same reducer.
- **Step 6**: The reducer function iterates over the list of input values to produce a single aggregate value as output.

Overview of MapReduce Framework

The MapReduce data flow can be decomposed into distinct phases, and within each phase, certain constructs are utilized to orchestrate the flow of data through the MapReduce framework (Figure 9).

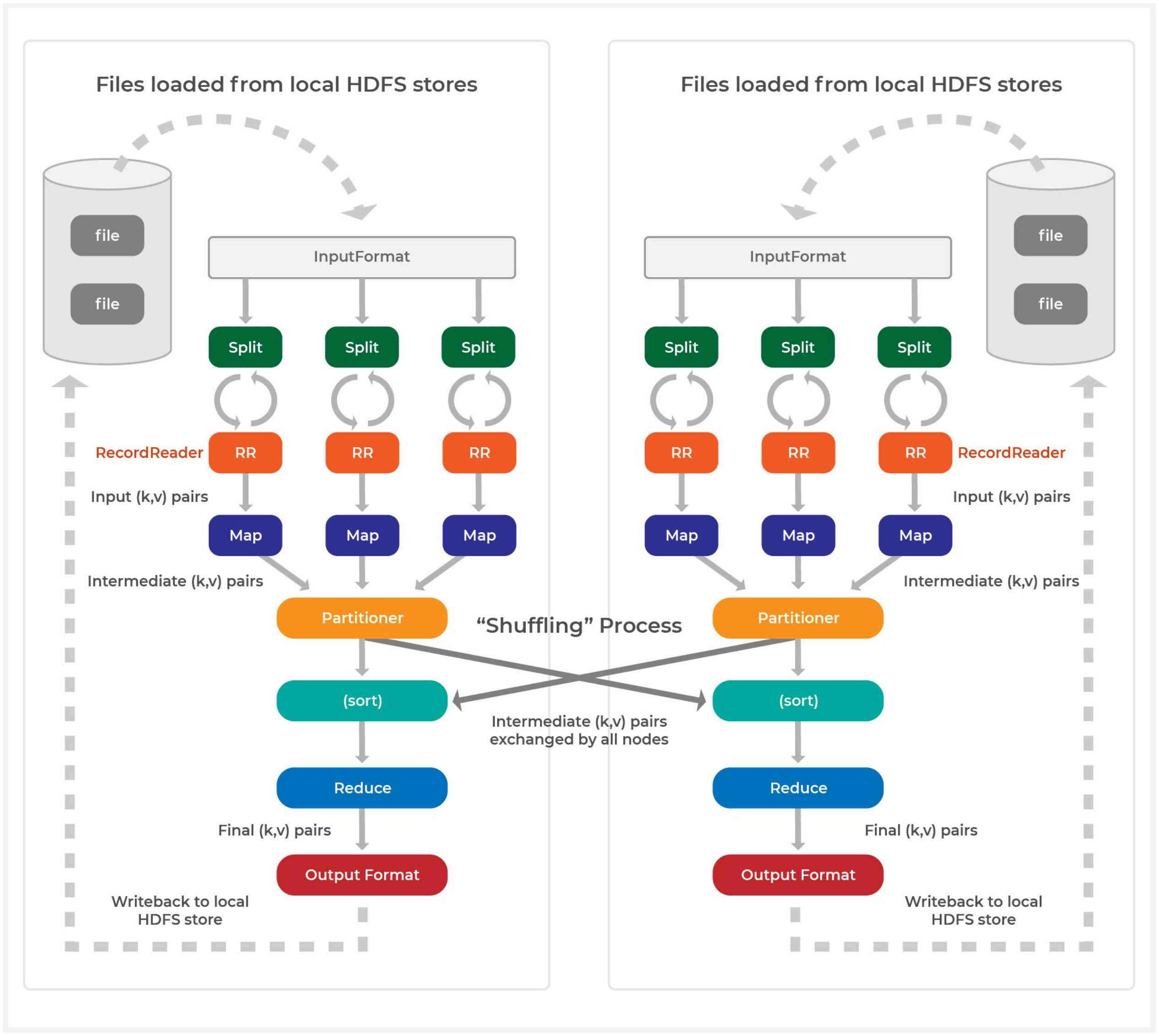


Figure 9: MapReduce data flow framework

The input data resides in files, usually hundreds or thousands of gigabytes in size. The MapReduce data processing framework splits the input data into chunks (Figure 10). In Hadoop parlance, these chunks are called *input splits*. The input splits (or chunks) can be processed in parallel using the nodes within a Hadoop cluster. The size of each input split should be small enough to facilitate a more granular parallelization.

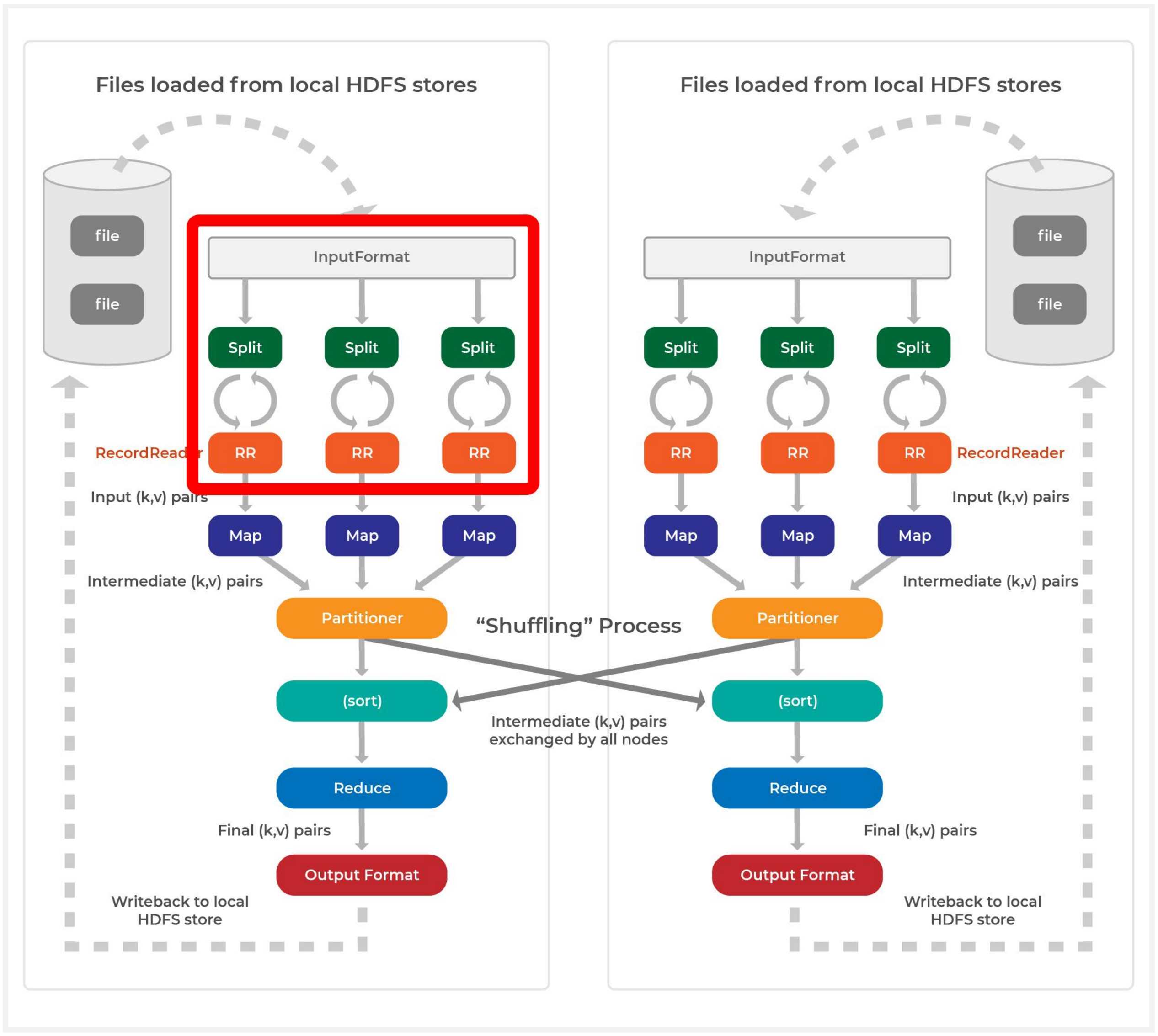


Figure 10: InputFormat function of MapReduce data flow

InputFormat (Text & KeyValueText)

The way an input file is split up and read is defined by the *InputFormat* construct. There are a few different implementations of the InputFormat construct, but the *TextInputFormat* is the default implementation. The TextInputFormat implementation accepts two parameters, *key* and *value*. The key is the byte offset of the line in the file, and the value is the contents of the associated line in the file.

Another often used implementation of the InputFormat construct is the *KeyValueTextInputFormat*. Similar to the TextInputFormat implementation, the KeyValueTextInputFormat accepts two parameters, *key* and *value*. The first delimiter (or separator) is the key, and everything following the separator is the value. The tab (*t*) is the default separator character.

If the KeyValueTextInputFormat implementation is used to read the following data file, then the key is the content before the tab (*t*) character, and the value is the content after the tab (*t*) character. Each line depicted below is a record with key and value separated by a tab (*t*) character.

Key	**Value**
19:18:20	Apache Spark™ - Unified Analytics Engine for Big Data
19:18:21	Spark SQL & DataFrames | Apache Spark
19:18:22	Overview - Spark 3.0.1 Documentation (apache.org)
19:18:23	Cluster Mode Overview - Spark 3.0.1 Documentation (apache.org)
19:18:27	Quick Start - Spark 3.0.1 Documentation (apache.org)

The two main tasks of the InputFormat construct are to:

a) Identify all the files that are utilized as input data and decompose the input data into input splits.

b) Iterate through the records in a given input split using the *RecordReader* construct to parse each record into associate *key* and *value* pairs of predefined types.

Mapper

The *Mapper* interface is responsible for the data processing step. It's single method '*map()*' accepts data in the form of key→value pairs and outputs data in the form of key→value pairs (Figure 11).

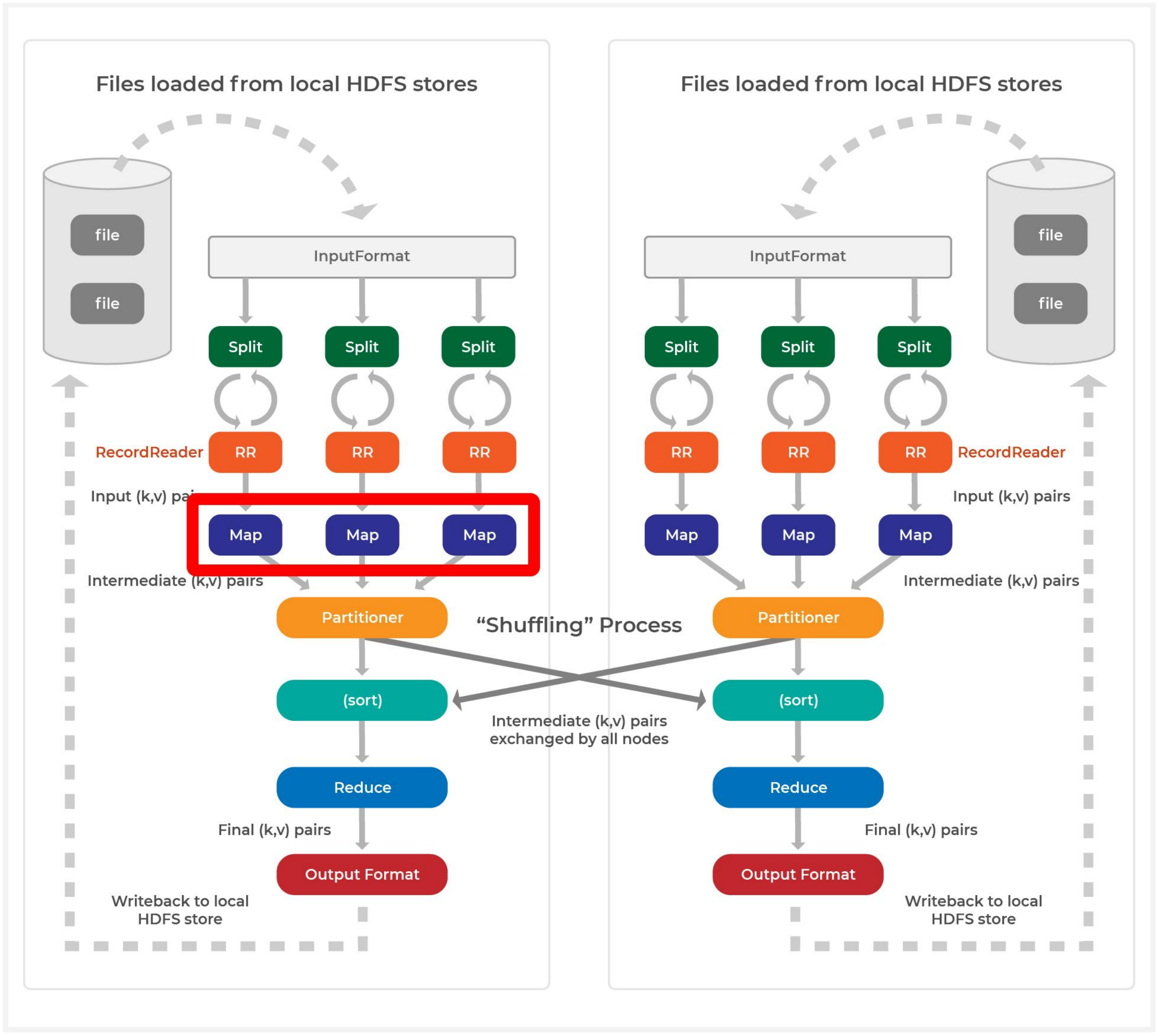
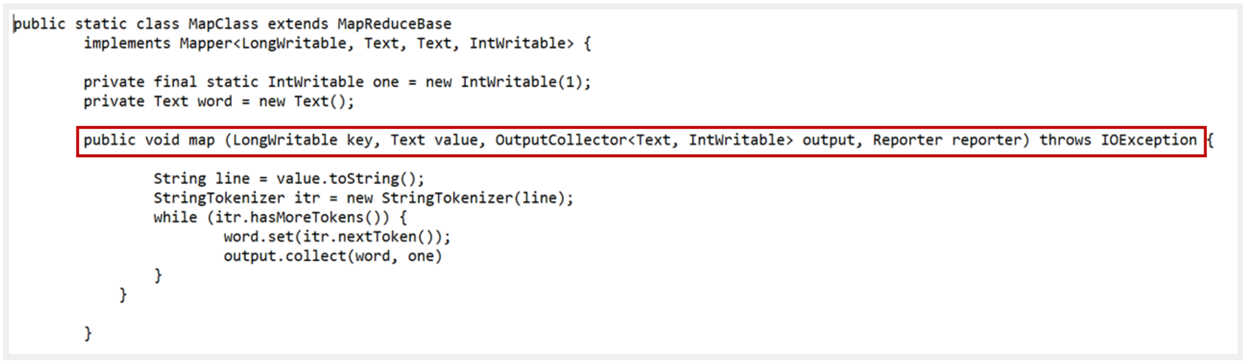


Figure 11: Mapping function of a MapReduce data flow

The map function outlined below (Figure 12) generates a list of key→value pairs as output for a given input set of key→value pairs. The *OutputCollector* receives the output of the mapping process, and the *Reporter* records additional information about the mapping process as data processing is performed.



```
public static class MapClass extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, IntWritable> {

        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map (LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {

                String line = value.toString();
                StringTokenizer itr = new StringTokenizer(line);
                while (itr.hasMoreTokens()) {
                        word.set(itr.nextToken());
                        output.collect(word, one)
                }
            }

        }
```

Figure 12: Code snippet for map function

In the code snippet above, the map function parses each key→value pair that it receives from the *RecordReader*. It decomposes the *value* (which is the contents of the line into the respective word tokens), and then it outputs each word as a key and a value of *one*.

Partitioner and Sort

The Mapper emits data as key→value pairs. Values associated with the same key need to be forwarded to the same reducer. The *Partitioner* ensures that all values for the same key are always reduced together, i.e., all values associated with the same key are sent to the same reducer (Figure 13).

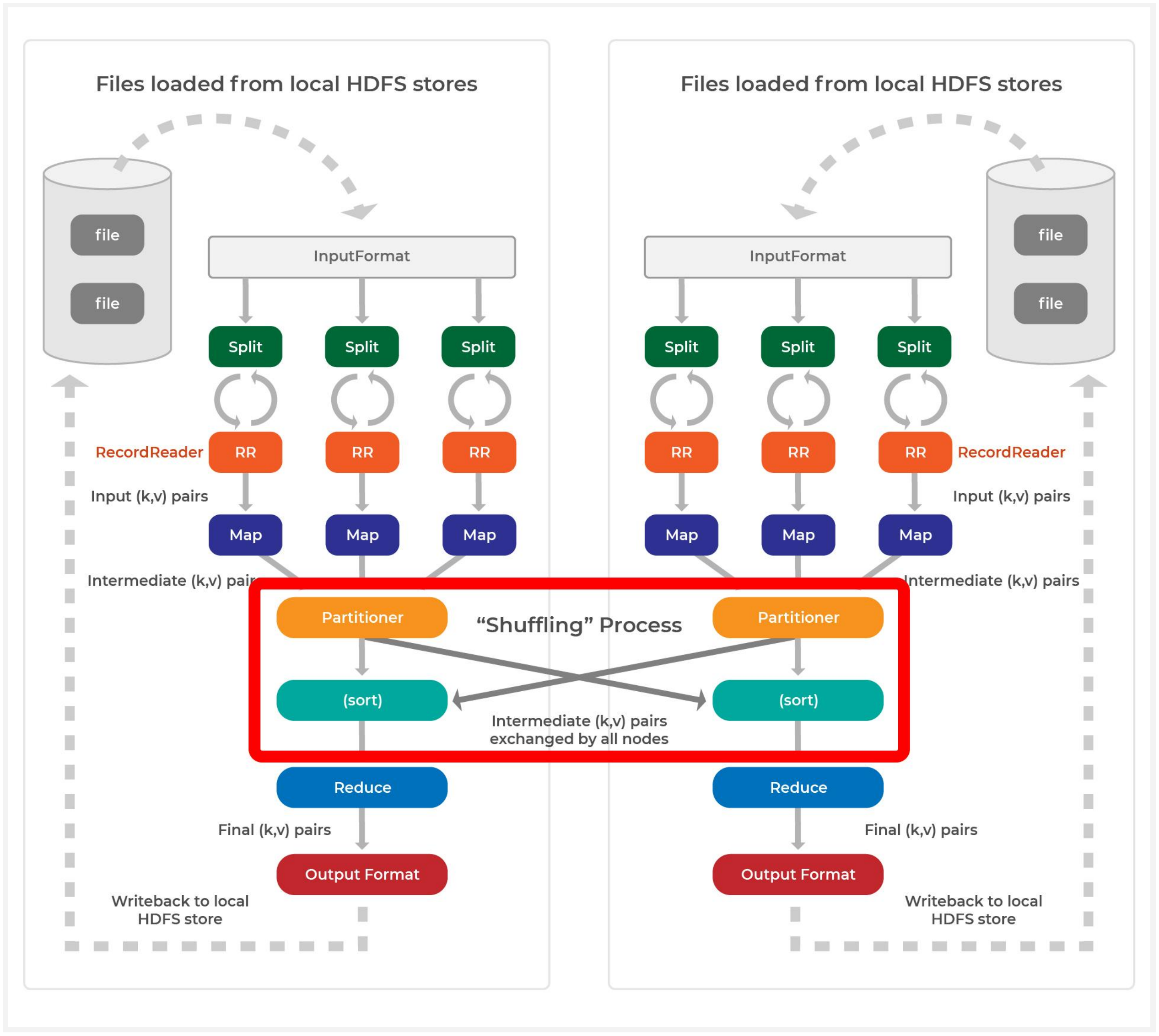


Figure 13: Partitioning and sorting functions of a MapReduce data flow

Sort is an automatic operation handled by Hadoop which ensures that keys forwarded to the same reducer are in sorted order.

Reducer

For each key assigned to a Reducer, the Reducer's "reduce" method is called once. The reduce method receives a key and an iterator over all the values associated with the key. The reduce method then generates a list of key→value pairs by iterating over the values associated with a key (Figure 14).

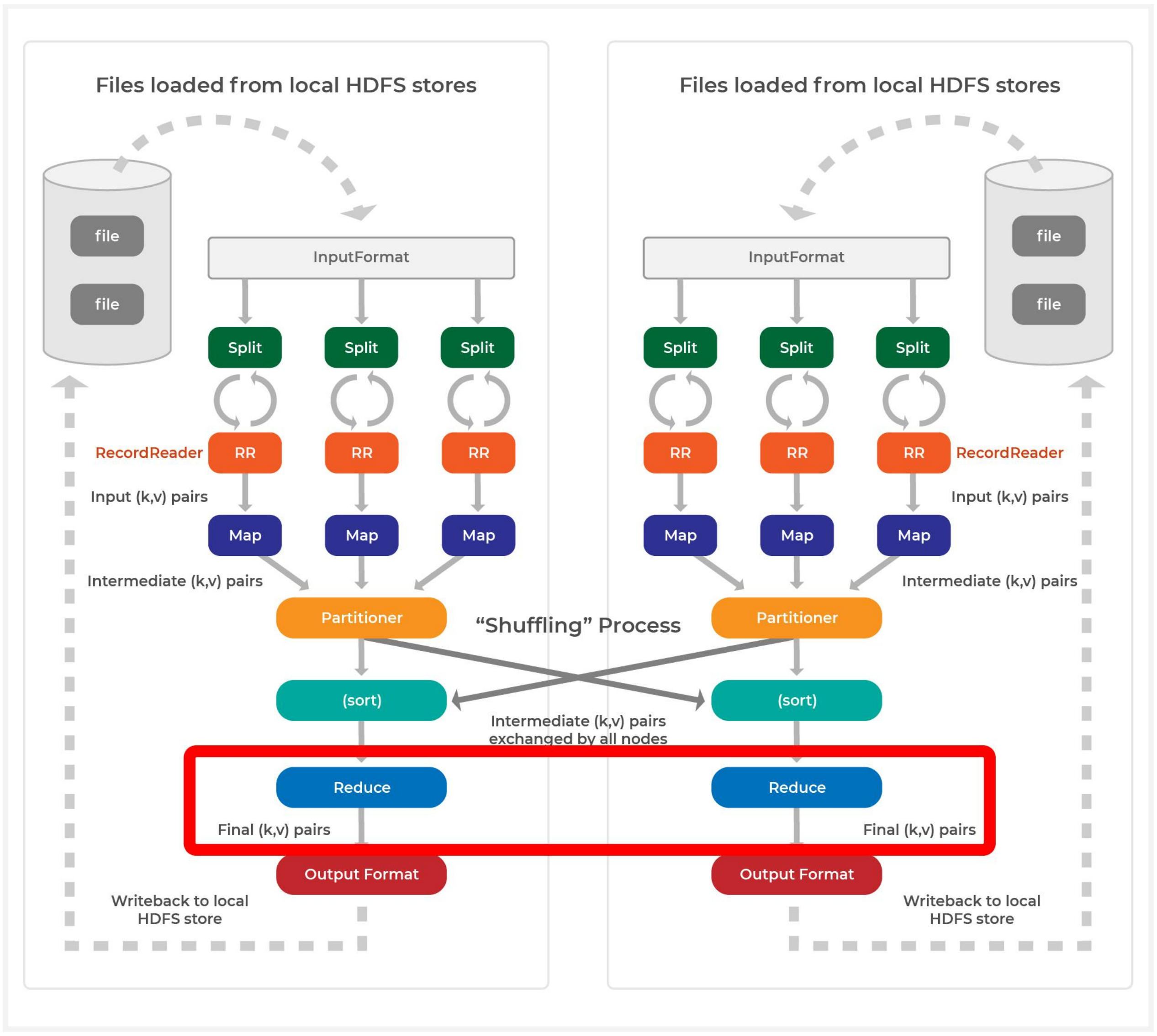
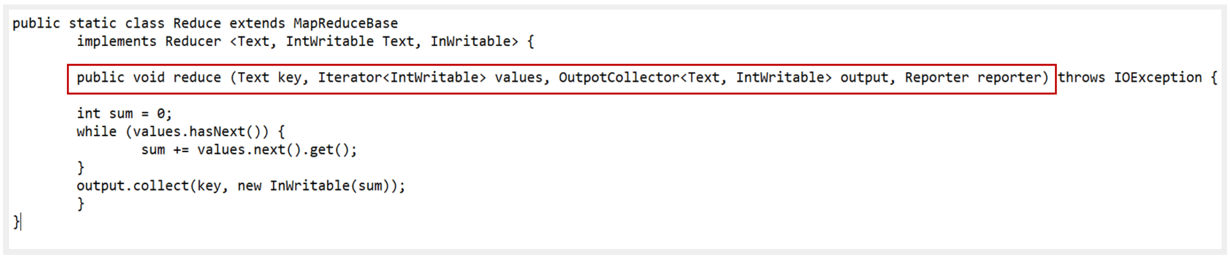


Figure 14: Reducing function of a MapReduce data flow



```
public static class Reduce extends MapReduceBase
        implements Reducer <Text, IntWritable Text, InWritable> {

        public void reduce (Text key, Iterator<IntWritable> values, OutpotCollector<Text, IntWritable> output, Reporter reporter) throws IOException {

        int sum = 0;
        while (values.hasNext()) {
                sum += values.next().get();
        }
        output.collect(key, new InWritable(sum));
        }
}
```

Figure 15: Code snippet showing reducing method return

The reduce method, which is user-defined, usually performs some sort of aggregation. In the code snippet displayed above (Figure 15), the reduce method returns the sum of the values associated with the key.

The *OutputCollector* receives the output of the reduce process and writes it to an output file. The *Reporter* provides the option to record extra information about the reducer as the task progresses.

OutputFormat

The MapReduce flow outputs data into files utilizing the *OutputFormat* class (see Figure 16). The output files reside in a common directory and are usually named "*part-nnnnn"* where "*nnnnn"* is the partition ID of the reducer.

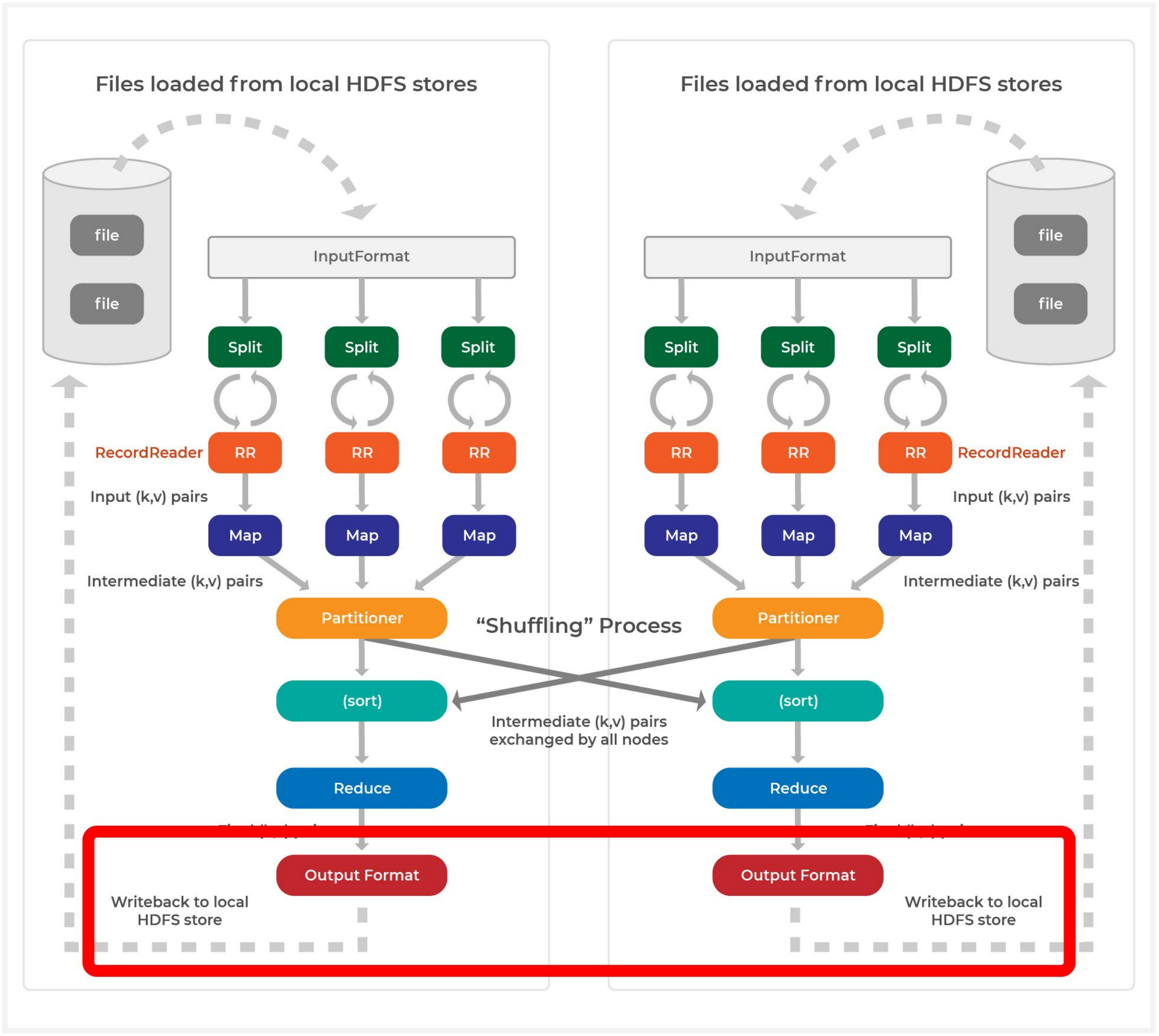


Figure 16: OutputFormat function of a MapReduce data flow

There are a few different implementations of the OutputFormat construct, but the TextOutputFormat is the default implementation. The TextOutputFormat implementation writes each record as a line of text. The keys and values are written as strings and separated by a (*t*) character.

The other implementations of the OutputFormat class are *SequenceFileOutputFormat* and *NullOutputFormat*, respectively. The SequenceFileOutputFormat writes the output in a sequence file format, which is Hadoop's proprietary file format. The NullOutputFormat is used when the output needs to be suppressed completely.

Examples of MapReduce Program Flow

Example #1: Pseudo-code

Objective: Compute (i.e., count) the number of times a word appears in a file.

```
mapper (filename, file-contents) :
      for each word in file-contents :
            emit(word, 1)
```

The mapper receives data as key→value pairs and emits data as key→value pairs.

```
reducer (word, values) :
      sum = 0
      for each value in values :
            sum = sum + value
      emit(word,sum)
```

The Reducer receives a key and an iterator of values associated with the key. It typically performs some sort of aggregation. In this case, it sums the values associated with a key, and emits the key and the sum of the values for the specific key:

File 1 → Sweet, this is the foo file.
File 2 → This is the bar file.

The MapReduce flow will produce the following output:

sweet	1
this	2
is	2
the	2

foo	1
bar	1
file	2

Example #2

- Input to the Map function:

  (3424, 'the dog sat on the mat')
  (3447, 'the parrot sat on the sofa')

- Map function (see original source):

```
public static class MapClass extends MapReduceBase
      implements Mapper <LongWritable, Text, Text, IntWritable> {

      private final static IntWritable one = new IntWritable (1);
      private Text word = new Text ();
      public void map (LongWritable key, Text value, OutputCollector <Text,
IntWritable> output, Reporter reporter) throws IOException {
            String line = value.toString();
            StringTokenizer itr = new StringTokenizer(line);
            while (itr.hasMoreTokens()) {
                  word.set(itr.nextToken());
                  output.collect(word, one)
            }
      }
}
```

- Output from the Map function:

  ('the', 1), ('dog', 1), ('sat', 1), ('on', 1), ('the', 1), ('mat', 1)

  ('the', 1), ('parrot', 1), ('sat', 1), ('on', 1), ('the', 1), ('sofa', 1)

- Data sent to the Reducer (*Note: the keys are in sorted order*):

  ('dog', [1])

  ('mat', [1])

('on', [1,1])

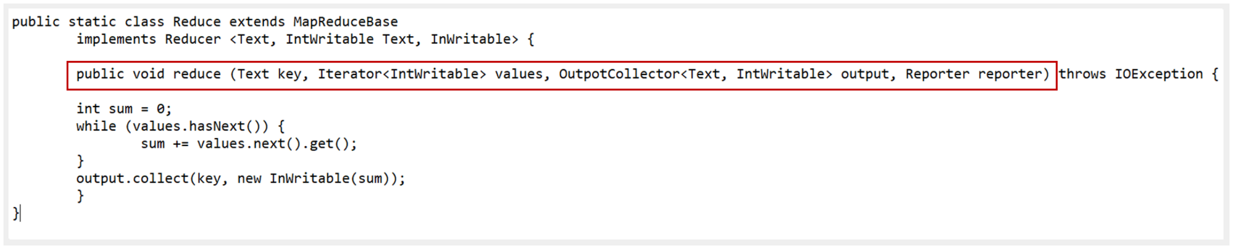
('parrot', [1])

('sat', [1, 1])

('sofa', [1])

('the', [1, 1, 1, 1])

- Reduce function:



```
public static class Reduce extends MapReduceBase
        implements Reducer <Text, IntWritable Text, InWritable> {

        public void reduce (Text key, Iterator<IntWritable> values, OutpotCollector<Text, IntWritable> output, Reporter reporter) throws IOException {

        int sum = 0;
        while (values.hasNext()) {
                sum += values.next().get();
        }
        output.collect(key, new InWritable(sum));
        }
}
```

- Output from the reducer:

('dog', 1)

('mat', 1)

('on', 2)

('parrot', 1)

('sat', 2)

('sofa', 1)

('the', 4)

References

tutorialspoint. (n.d.). Hadoop - MapReduce.

Hurwitz, J., Nugent, A. Halper, F. & Kaufman, M. (2013). Chapter 12: Defining Big Data Analytics. *Big Data for Dummies.*