2. Big-data Computation

This chapter describes how big-data platform perform the parallel computation. The chapter gives examples on how the concepts are applied in big-data platforms such as Spark, Hadoop, and AsterixDB. This chapter is not meant to exhaustively cover all aspects of these systems but to describe some of the common and basic concepts of big-data computations.

2.1 Overview of Big-data Computation

Big-data platforms are systems that allow an average developer to write a program that runs in parallel on a shared-nothing cluster. In this part, we start with a simple example that we then build on throughout the chapter. We also give a high-level overview of the main components of big-data systems.

■ **Example 2.1 — Word Count.** In the word count example, we have a large text file and we would like to count the number of occurrences for each word in the file. A sample input and output are given below for illustration. In practice, we use a big-data platform only when the file is extremely large, say, several gigabytes in size.

File 2.1: wordcount-input.txt

```
I often repeat repeat myself
I often repeat repeat
I don't don't know why know why
I simply know that I I I
am am inclined to say to say
a lot a lot this way this way
I often repeat repeat myself
I often repeat repeat
```

File 2.2: wordcount-output.txt

a	2
am	2
don't	2
I	9
inclined	1
know	3
lot	2
myself	2
often	4
repeat	8
say	2
simply	1
that	1
this	2
to	2
way	2
why	2

■

Word Count Walk-through

Figure 2.1 illustrates how the word count example runs on a typical big-data system. The first step is to read the input file. Since the input might be extremely large, the input is partitioned into *input splits* based on the file size. Typically, each split is 128 MB but can be configured. Big-data frameworks deal with records not entire files. Therefore, a record reader takes an input split and produces a set of records; in this case, a set of lines. Each partition is processed independently and they are normally processed on different machines.

After reading files from the input, a line splitter takes each record and produces pairs of $\langle w, 1\rangle$ for each word in the line. Since one partition can contain millions of words, the next step computes a partial word count within each partition by adding up all pairs with the same word into a pair $\langle w, c\rangle$ where c is the partial count of a given word within that partition.

To compute the final count, the big-data framework will *shuffle* the records between partitions to ensure that all pairs with the same word end up in one partition. This is done using a hash partitioner that assigns each word to one of the output partitions, i.e., 4.1 or 4.2. In this example, we use a simple partitioner based on the first letter of the word but typically a hash function is applied for better load balancing.

After that, each machine will independently compute the final word count by adding up the partial counts for the same word. For example, the pairs $\langle I, 4\rangle$, $\langle I, 3\rangle$, and $\langle I, 2\rangle$ in partition 4.1 will be reduced into one record $\langle I, 9\rangle$. Since all pairs with the same key (word) will be assigned to one machine, this final count is guaranteed to be correct. The final step is to write the final counts to the output. To avoid a bottleneck in writing the output, each partition writes a separate file. All files together constitute the output.

Word Count in Hadoop MapReduce

Listing 2.1 shows how the word count program is implemented in Hadoop MapReduce. The program is a little bit longer since Hadoop requires the developer to be more specific about the different aspects of the program. A MapReduce program consists of mainly two functions, map and reduce. The map function is defined in class TokenizerMapper which represents the transformation from partitions 1.x to partitions 2.x. The reduce function is defined in the class

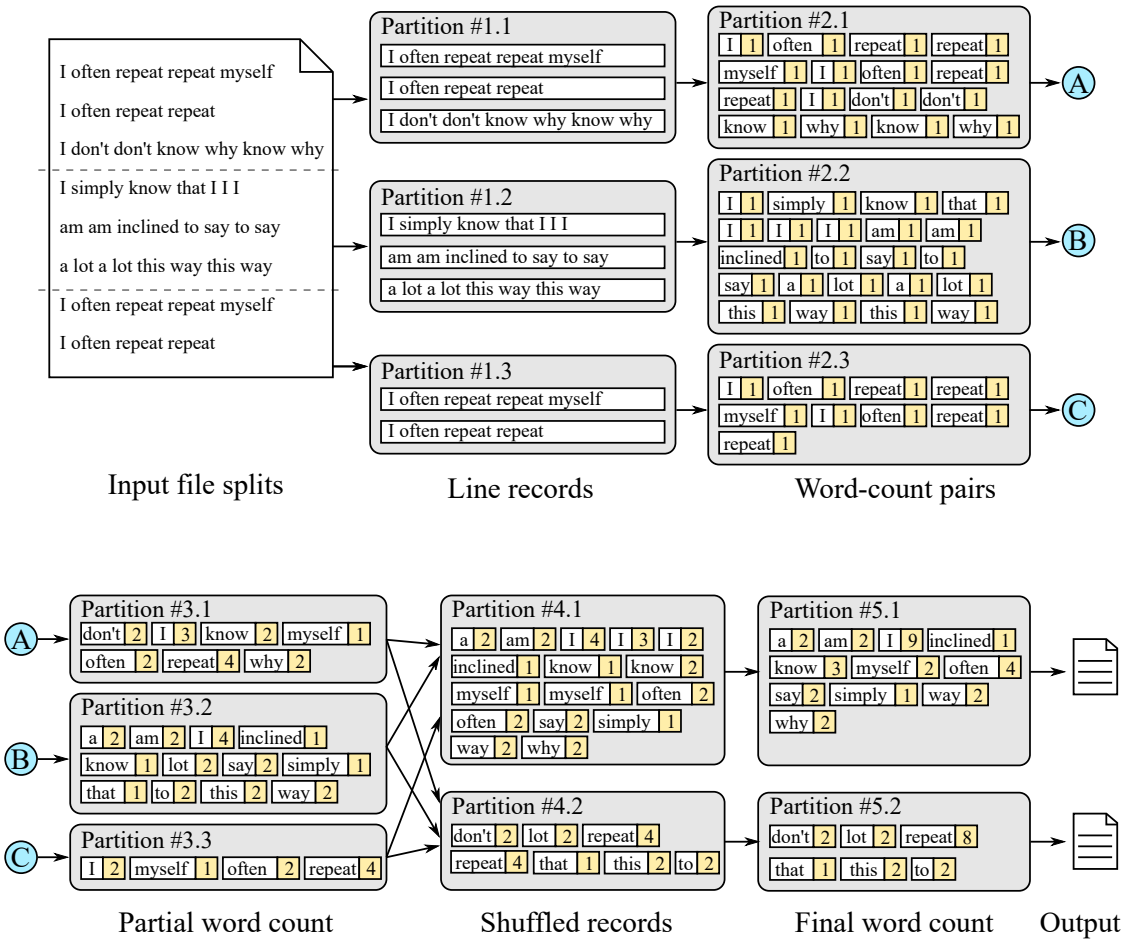


Figure 2.1: Word count execution on big-data

IntSumReducer defines the logic between partitions 2.x to 3.x and 4.x to 5.x. The logic is to simply sum all values with the same key. The rest of the program is defined in the main function. The functions setMapperClass, setCombinerClass, and setReducer class instructs Hadoop to run the given functions on partitions 1.x, 2.x and 4.x. The function addInputPath tells Hadoop to read the input file as a text file. Finally, the function setOutputPath tells Hadoop where to write the output as text.

Listing 2.1: Word count example in Hadoop MapReduce

```
public static class TokenizerMapper
 extends Mapper<Object, Text, Text, IntWritable>{

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

 public void map(Object key, Text value, Context context)
  throws IOException, InterruptedException {
  StringTokenizer itr = new StringTokenizer(value.toString());
  while (itr.hasMoreTokens()) {
   word.set(itr.nextToken());
   context.write(word, one);
  }
 }
}

public static class IntSumReducer
   extends Reducer<Text,IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();

  public void reduce(Text key, Iterable<IntWritable> values,
     Context context) throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values)
      sum += val.get();
    result.set(sum);
    context.write(key, result);
  }
}

public static void main(String[] args) throws Exception {
  Job job = Job.getInstance(new Configuration(), "word count");
  job.setJarByClass(WordCount.class);
  job.setMapperClass(TokenizerMapper.class);
  job.setCombinerClass(IntSumReducer.class);
  job.setReducerClass(IntSumReducer.class);
  FileInputFormat.addInputPath(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));
  System.exit(job.waitForCompletion(true) ? 0 : 1);
 }
}
```

Word count in Spark

Listing 2.2 shows how to implement the word count program in Spark. This program runs in a way similar to the example described above.

Listing 2.2: Word count example in Spark using Scala RDD API

```
1 val lines: RDD[String] = sc.textFile("wordcount-input.txt")
2 val words: RDD[String] = lines.flatMap(l => l.split("\\b").iterator)
3 val wordPairs: RDD[(String, Int)] = words.map(w => (w, 1))
4 val wordCounts: RDD[(String, Int)] = wordPairs.reduceByKey((a,b) => a+b)
5 wordCounts.saveAsTextFile("wordcount-output.txt")
```

Line 1 loads the input data from a text file. The return value is a Resilient Distributed Dataset (RDD) which is a reference that points to a set of text lines distributed over multiple machines. We call this an input RDD since it is a top-level RDD and does not depend on others.

Line 2 splits each text line in the input file to a set of words. The return value is another RDD that contains the set of all words in the file. The function `flatMap` is called a `transformation` that modifies an RDD to another RDD. In this case, we call `input` the *parent* RDD and `words` the *child* RDD. The syntax '`l => l.split("\\b").iterator()`' is called *lambda* expression. Lambda expression is a short-hand for defining a class with a single function like the ones defined in the Hadoop example. In this case, it the function takes as input a string named `l` and returns an iterator of strings for all words in the input string. Similar to Hadoop, Spark relies on *functional programming* as the programming model where the user-defined logic is provided in a set of functions like this one.

Line 3 makes a similar transformation that converts each word w to the pair $\langle w, 1 \rangle$. The result is an RDD where each tuple contains two values. Spark calls that a *pair RDD* and provides an additional set of functions that only applies to pair RDDs.

Line 4 runs another transformation that computes the summation of all values with the same key. In this case, it computes the number of occurrences for each word. The function `reduceByKey` is an example of a transformation that is only applicable to pair RDDs.

Line 5 will store the final result to an output file named 'wordcount-output.txt'. This is called an *action* and it informs Spark on what to do with the final output. Without an action, Spark does not have a complete job specification and will not execute any of the code provided. Only after the action is provided, Spark will run the entire job including loading the file, splitting the records, counting by key, and writing the output.

2.2 Programming Model: Functional Programming

Popular big-data systems adopt a *functional programming* model in which the developer expresses the logic of the program using functions. Listing 2.2 included a few functions such as '`w => (w, 1)`' and '`(a, b) => a+b`'. When writing a big-data program, the developer defines the functions and passes them to the big-data system. It takes the function definitions and broadcasts it to all machines so that it can be applied in parallel. In the MapReduce example in Listing 2.1, the function setJarByClass instructs Hadoop to broadcast that Jar file which contains all the defined classes to all machines in the cluster. Spark does this automatically using reflection. To allow the big-data framework to apply the given functions in parallel and give a correct result, these functions have to be *stateless* and *deterministic*.

Definition 2.2.1 — Stateless. A stateless (memoryless) function does not use or update any internal or global state. In other words, it cannot remember the past records that it processed earlier. It cannot either use a global state that is changed by another function.

Definition 2.2.2 — Deterministic. A deterministic function does not perform any random decisions. It should always return the same output for the same input.

When all functions in the program are stateless and deterministic, the big-data platform has the flexibility to run them out of order, in parallel, and even repeating some of them many times without worrying about messing up the logic of the program.

Another way to think about functions with these constraints is to consider them like mathematical functions. A basic property of a mathematical function is that it has one output for each input. This means it will always produce the same output without keeping memory and without randomization.

While these limitations might seem too restrictive, there is still a lot you can do with big-data systems without violating these requirements.

Hadoop or Spark will not check if your functions are stateless or deterministic. Your program will not immediately fail either if you violate these requirements. At the same time, it is not guaranteed that your program will always run correctly. For example, it might run when you test it in development mode, and fails when it runs on an actual distributed cluster. So, these requirements are more like an agreement between the developer and the big-data platform to ensure that the program will be fine.

■ **Example 2.2 — Word Count Functions.** We revisit the word count example described earlier. In this example, the input tuples consist of strings. The Spark program consists of three functions.

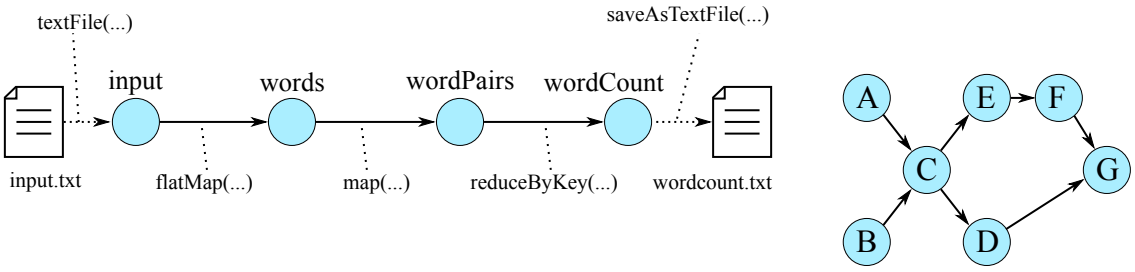
1. `l => l.split("\\b").iterator()`: This function converts a string to a set of words.
2. `w => (w,1)`: This simple function converts a word w to a tuple $(w, 1)$.
3. `(a, b) => a + b`: This function simply adds two numeric values.

As can be seen, the three functions satisfy the two constraints which make the program valid.

Similarly, the MapReduce program consists of two functions defined in classes TokenizerMapper and InSumReducer. While these functions seem to keep some state in the instance variables, `one`, `word`, and `result`, the values of these variables are not carried over from one function call to the next so they do not violate the requirements. ■

2.3 Application Model: Directed Acyclic Graph (DAG)

The functional programming model allows a developer to write a single f unction to customize one of the Spark operations. However, a program is more complicated than a single function. A complete big-data program starts by parsing the input into records, goes through several transformations, and ends by writing the final result to the output. Big-data platforms model a program as a *directed acyclic graph (DAG)* that represents a network of operations. Figure 2.2(a) illustrates the DAG that represents the word count example. Notice that dotted lines are added for annotation and are not part of the DAG. Due to the simplicity of the word count example, the DAG is represented as a simple list. Figure 2.2(b) shows a general example of DAG that can represent a more complex Spark job. Each *vertex* represents a set of records, e.g., RDD, which could be an input, intermediate, or final records. Each *edge* represents an operation that transforms one dataset to another. We adopt Spark terminology of *parent* and *child* to represent the *source* and *destination* of an edge in the graph, respectively. For example, in Figure 2.2, A and B are both parents of C; and C is the parent of both D and E. It follows that a vertex with no parents, i.e., zero in-degree, represents an input and a vertex with with zero out-degree represents an output. The DAG representation is very common in big-data platforms, e.g., AsterixDB, Spark, and Impala, to the point that an open source project, called Tez, was developed just to create and manipulate DAGs. Notice that Hadoop programs are much limited as compared to other systems, like Spark and AsterixDB, but a MapReduce program



(a) DAG of the word count example. Dotted lines are not part of the DAG.

(b) General example of DAG

Figure 2.2: Directed Acyclic Graph (DAG)

can still be represented as a simple graph. Users need to pipeline multiple MapReduce jobs to form a more complex DAG.

The application graph is inherently directed since each operation has an input and output which are asymmetric. Additionally, the graph has to be acyclic to allow big-data frameworks to apply a topological sort algorithm on the vertices and execute the operations in order. If there is a cycle, the system will now know where to start and end the execution of this cycle. To run an iterative algorithm that executes many iterations in a loop, the developer has two options. First, one can unroll the loop to make it linear. The other option is to create a job that represents one iteration of the loop and execute the job many times on Spark. In this case, there must be an external program that controls how many times to run the loop and when to stop.

2.4 Execution Model: Bulk Synchronous Parallel (BSP)

Most big-data frameworks use the *bulk synchronous parallel* (BSP) model [1] to execute large jobs. In other words, it translates the user program written using functional programming, into a lower-level program that uses the BSP model. In this model, the processing is split into separate *stages*[1] of local processing separated by communication steps as shown in Figure 2.3.

In each stage, many processors run concurrently and in isolation, i.e., no communication happens between these processors. We use the term *task* to indicate the work done by each processor. Each processor has its own input and produces its own output that is completely independent of other processors. This gives big-data systems the flexibility to run these tasks in any order and on any machine. If the number of tasks is larger than the number of processors, it can run them in *waves*. If the number of tasks is less than the number of processors, it can run some of them in *speculative* mode.

Definition 2.4.1 — Speculative Execution. Speculative execution is when multiple processors are executing the same task on the same data in parallel to find which processor will finish faster. It can be helpful to avoid the waiting time of a *straggler* machine that is abnormally slow due to system overload.

After all processors are done with their execution, a communication step happens in which each processor can send some information to other processors. During the communication step, all processors are getting ready for the next stage. Once the communication step is done, the next stage starts execution. The communication pattern between processors depends on the logic of the program. For example, the communication pattern can be 1:1 where each processor in stage i sends

[1] We use the term stage which is used by Spark and AsterixDB but it is synonym to the term *superstep* that can be found in literature.

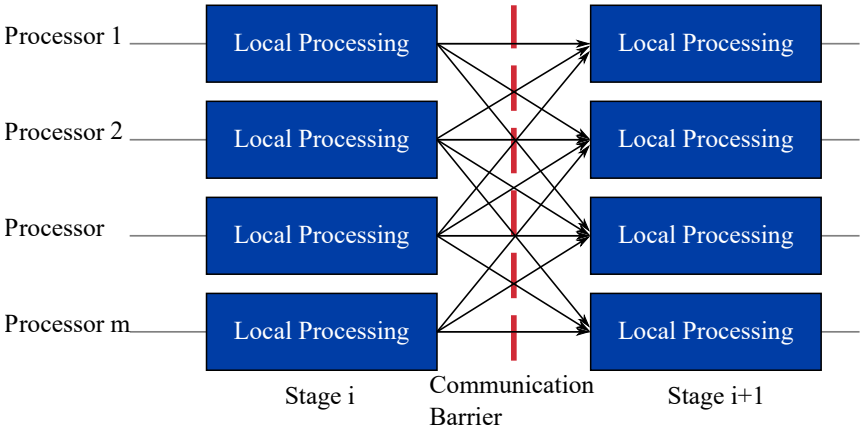


Figure 2.3: Bulk Synchronous Parallel (BSP) Model

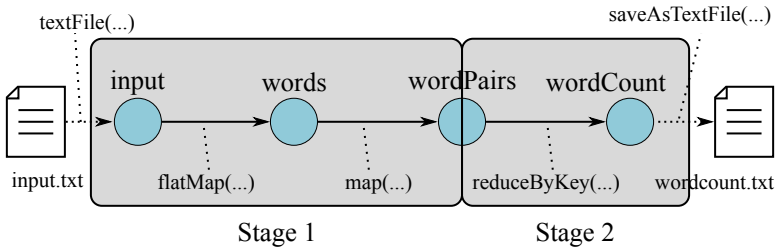


Figure 2.4: Word count stages

its output to one processor in stage $i+1$. On the other extreme, it can be fully connected where each processor broadcasts its output to all processors in the next stage.

Figure 2.4 shows how the word count example is broken into two stages. The first stage consists of reading the input file and computing the partial word-count pairs in each partition. This stage does not require any communication between machines since each partition is processed independently. However, to compute the final word counts, a communication step is needed to put all partial word-count pairs with the same word will be grouped in one partition. No further communication is needed until writing the final output so the program will consists of only two stages.

Due to the limited flexibility of MapReduce, any Hadoop job will consists of either one or two stages. A map-only job with no reduce function is executed in one stage while a map-reduce job, like the word count example, will run in two stages. Spark and AsterixDB are more general and can run a job in more than two stages depending on its flexibility.

The way big-data platforms break down a complex DAG into stages is by inspecting the communication patterns between partitions. Simple patterns like one-to-one patterns do not require communication between partitions, hence, they can be grouped together int one stage. However, a general communication pattern, e.g., m-ot-n, require network communication between partitions, hence, big-data platforms will break the job into two stages along these boundaries.

Spark uses the terms, *narrow dependency* and *wide dependency* to classify the communication patterns. Transformations like `flatMap` and `map` induce a *narrow dependency* so Spark can run them within the same stage. However, the `reduceByKey` transformation has a wide dependency so Spark will break the job into two stages when encountering that operation. AsterixDB uses the

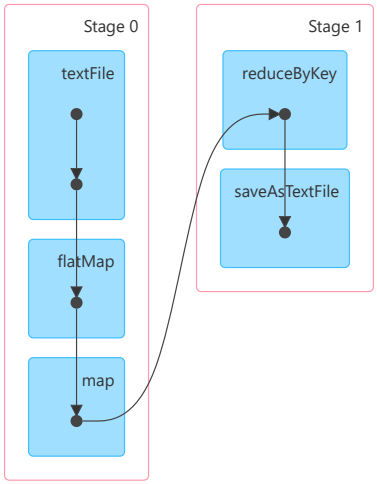


Figure 2.5: The word count job DAG as generated by the Spark web UI

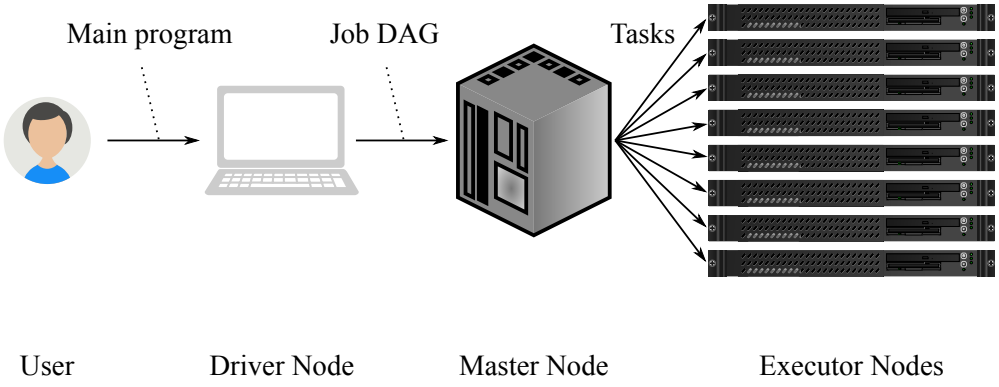


Figure 2.6: Overview of Spark architecture

term *connectors* to indicate the communication patterns in a similar way.

R Big-data platforms achieve parallelization through *horizontal partitioning* in which a set is broken into partitions which are processed in parallel as shown in Figure 2.1. Each partition is processed independently as an individual *task*. Therefore, if the number of partition is equal to one, no parallelization is used. For example, if the number of reducers in Hadoop is set to one, which is unfortunately the default, no parallelization will be done for that stage. Similarly, if any RDD has one partition only, it will be processed in one task regardless of the number of machines in the cluster. The number of partitions in the input is usually driven by the input size. By default one partition is created for each 128 MB file split.

2.5 Big-data Job Execution

This section describes how big-data applications are compiled into a DAG and how they are executed using the BSP model. First, we give a high-level overview of the architecture of big-data systems and then we provide a few implementation-specific details to give a concrete description.

2.5.1 Big-data Architecture

First, we will give a high-level overview of the architecture of a typical big-data cluster. As shown in Figure 2.6, a cluster consists mainly of three types of machines, *driver*, *master*, and *executor*. A driver node is where the developer runs the main program similar to the one given in Listings 2.1 and 2.2. The master node receives the job from the driver node and breaks it down into stages and tasks to run in parallel on the executor nodes as illustrated in Figure 2.1. When the job is done, the driver node is notified with the status of the job, e.g., success or failure.

Definition 2.5.1 — Driver Node. A driver node is where the developer program is first executed. A typical driver program starts by creating a connection to the master node, e.g., SparkContext in Spark or JobContext in Hadoop. This context is then used to submit one or more jobs to the master node to run on the cluster.

Definition 2.5.2 — Master Node. The master node is the main point of contact with the cluster. It is always running and waiting for jobs from driver nodes. The master orchestrates the job by breaking it down into stages and tasks and keeps track of executor status. If the master fails, the entire cluster becomes inaccessible until a secondary master takes over.

Definition 2.5.3 — Executor Node. An executor node carries out individual tasks that comprise the big-data job. Executors get their tasks from the master and periodically report their status to the master. If one or a few executors are inaccessible, the cluster can keep functioning with the remaining ones.

In general, the driver, master, and executors can be on different physical machines. However, it is possible that they partially or fully run on the same physical node. For example, it is common to submit the jobs through the master node in which the master and driver nodes become collocated. Also, during development and testing, it is common to run all the three nodes on the development machine. Interested readers can find more details about this approach for Apache Spark [`https://spark.apache.org/docs/latest/spark-standalone.html`], Hadoop [`https://hadoop.apache.org/docs/r3.2.2/hadoop-project-dist/hadoop-common/SingleCluster.html`], and AsterixDB [`http://asterixdb.apache.org/docs/0.9.6/ncservice.html`].

2.5.2 Input Reading

The first step in any big-data job is reading the initial set of records that will be processed. In the word count example, this means reading the input text file as a set of lines. To read the file in parallel, either the driver or the master node splits the file into multiple splits. Then, the executors process these splits in parallel to extract the records out of them. Therefore, the number of splits defines the level of parallelism for the first stage in processing. For example, if the file is split into four splits, only four processors will actively process it in the first stage. Subsequent stages might have a different level of parallelism. In the following part, we explain how splits are created and processed.

Input Splitting

In this first step, the input is split into smaller pieces that executors can read in parallel. This step is processed by a *single machine*, either the driver or the master node, depending on how the big-data framework is designed. Since it runs on a single machine, this step has to be very efficient and light-weight. If not, it could become a bottleneck in the query pipelines. There are three methods that can be considered when splitting the input.

1. **Node-based splitting** determines the number of partitions based on how many machines are in the cluster. This approach is very popular in High Performance Computing (HPC) systems

where the data is partitioning on the fly based on the number of processors. The disadvantage is that if the input is very large, a single partition might be too big to be processed as one piece. An example of node-based splitting is the `parallelize` method in Spark which takes a range or array and splits it into a fixed number of splits, generally determined based on the cluster size.

2. **Record-based splitting** assigns a fixed number of records per partition. This approach promises a good load balance since the number of records is a strong indicator of the query processing cost. An example of record-based splitting is reading data from a DBMS, e.g., PostgreSQL, where the data is naturally organized in records and the total number of records is already available in the catalog. However, this method is not applicable when processing unstructured or semi-structured data, e.g., CSV or JSON file, since the number of records is not known.
3. **Size-based splitting** creates splits of a fixed size, e.g., 128 MB. This method is preferred when processing unstructured files when only the file size is known. In addition, when the file is stored in HDFS, the split boundaries can be aligned with block boundaries to ensure that each split is contained in one block. This way, the task scheduler can collocate data and processing for increased efficiency.

The result of this step is a set of input splits that *logically* partition the input. It is necessary that the information in these splits should be serializable to be transferred over network from the master node to executors where they will be processed. This can be done in various ways. For example, Hadoop uses a `Writable` interface while Spark supports various options such as the builtin Java serialization framework and the more efficient Kryo serializer.

Record Reading

The second step is when an executor takes one of the input splits and reads the records. Each executor takes one split at a time and extracts all the records in that split. Since each split can be relatively large, e.g., around 128 MB, most big-data frameworks use an iterator interface that passes records one-at-a-time to the processor of the first stage. This way, it avoid materializing all records to the memory. Some systems take this one step further by defining a *mutable* interface in which it creates only one object and modifies the value of this object for each record in the split. This has an advantage of minimizing object allocation and garbage collection which sometimes can be costly.

Hadoop and AsterixDB rely on a mutable interface while Spark uses an immutable interface. This means that Spark will physically allocate a new object for each record in the input or intermediate records. In general, Spark is marketed as a memory-based big-data system which uses as much memory as it can to reduce query processing time. Hadoop and AsterixDB are among memory-conservative systems that try to reduce memory usage.

One of the main issues associated with size-based splitting, is how to deal with records that span multiple splits. For example, when reading a text file, a single line can start in one split and end in the next split. In this case, we need to ensure that the record is read exactly once. Most systems use the *reference point* duplicate avoidance technique. This technique choose a single point in each record, typically the first byte, and associated the entire record to this point. Then, the record is processed only by the split that contain the point. Since the splits are disjoint, the reference point will fall in only one split.

For example, in the case of reading a text file, the line is processed only by the split that contains the first character of that line. To do so, the record reader will process all lines normally in the file. When the end of split is reached, the record reader will continue beyond the end-of-split and continue reading the last line and then stop. In addition, all record readers (except the first one), will read and drop the first few characters until the first end-of-line is reached. These dropped characters

Figure 2.7: MapReduce processing pipeline

correspond to the extra characters read by the record reader processing the previous split. This technique requires that each record reader has access to characters outside its split. HDFS provides this access through its abstract FileSystem interface. Even though the additional characters will be mostly transferred from another machine, the overhead is minimal.

Exercise 2.1 Suppose we have an input text file that is compressed using a block-based compression technique, e.g., B-zip2. This means that the file is compressed into roughly equi-sized blocks. Block boundaries are marked with a special character and each block can be decompressed independently. How would a big-data system efficiently split and read the text lines in that file? ■

2.5.3 Hadoop MapReduce (MR) Processing

This section explains the Hadoop MapReduce query execution engine as an example of big-data processing. In MapReduce, a user program is split into two stages, *map* and *reduce*. The map stage processes each input split independently to produce a set of intermediate pairs $\langle k, v \rangle$. These intermediate pairs are grouped by key and the reduce stage processes the grouped records to produce the final output. Figure 2.7 gives an overview of the MapReduce query processing pipeline.

Map Stage

The map stage consists of multiple processes working parallel to process all input splits. Each process is called a *mapper*. Users customize the mapper using by providing a mandatory *map* function and an optional *combine* function.

The map function takes a single record as input and produces a set of zero or more key-value pairs $\langle k, v \rangle$. Below, is an example of a map function from the word count example.

Listing 2.3: Example of a (simplified) map function in Hadoop

```
1 public void map(Object key, Text value, Context context) {
2   StringTokenizer itr = new StringTokenizer(value.toString());
3   while (itr.hasMoreTokens()) {
4     word.set(itr.nextToken());
5     context.write(word, one);
6   }
7 }
```

The map function receives one input record at a time, represented as the input pair `key` and `value`. It also receives a *context* to which the output pairs will be written. This allows the function return zero or many values through the context since most programming languages, including Java, support only one return value. Notice that the output key and value are generally of different types than the input key and value types. Hadoop is built with a mutable interface which allows all output pairs to use the same Java objects with different values. Hadoop will directly serialize them to avoid losing any information.

The intermediate key-value pairs produced by the map function are internally sorted by key inside each mapper so that pairs with the same key will be consecutive in the sort order. Since the number of pairs can be extremely large, an *external sort* algorithm is used which does not have to keep all records in memory.

An optional *combine* function can be provided by the developer. The combine function takes as input a single key and a set of values and returns another set of key-value pairs. The goal of the combine function is to perform internal aggregation and simplification inside each node to

reduce the number of records before running the more expensive reduce step. Hadoop has two requirements in the combine function to be correct.

- The types of the input key-value pairs must be similar to output key-value pairs.
- The logic of the function should allow it to run zero or more times.

These two requirements give Hadoop the flexibility to skip the combine function, if needed, or to apply it many times. For example, if the output of the map function is already very small, Hadoop might want to skip the combine function. Therefore, the MapReduce program should not rely on the execution of the combine function for correctness.

The output of the combine function is again sorted by key in preparation for the reduce stage.

Reduce Stage

Listing 2.4: Example of a (simplified) reduce function in Hadoop

```
1 public void reduce(Text key, Iterable<IntWritable> values,
2     Context context) {
3   int sum = 0;
4   for (IntWritable val : values)
5     sum += val.get();
6   result.set(sum);
7   context.write(key, result);
8 }
```

The reduce stage runs in parallel on possibly a different set of nodes than the mappers. The reduce stage starts with a *shuffle* phase that sends the output from the mappers to reducers. Assuming we have m mappers and r reducers, Hadoop establishes $m \times r$ connections between every pair of partitions in mappers and reducers. Then, each mapper applies a partition function $p : k \to [0, r)$ that takes a key k and produces an integer in the range $[0, r)$ that indicates the corresponding reducer. Then, each mapper sends every key-value pair to the corresponding reducer. Since all the intermediate key-value lists are already sorted by key, reducers will run a merge step that combines the data it receives from m mappers into a single list of key-value pairs sorted by key.

R Hadoop allows the developer to provide a user-defined partition function but the default is a simple hash function that generally works correctly.

The next phase is the *reduction* phase in which these records are processed using the user-provided reduce function. As shown in Listing 2.4, the reduce function takes as input a single key and a list of values and it produces the final set of records. Similar to the Hadoop and combine functions, Hadoop can push zero or more outputs through the provided context. While the reduce function can be similar to the combine function, it does not have the same constraints imposed by Hadoop on the combine function. First, the output of the reduce function can be of different types than the input. Second, Hadoop will apply the reduce function exactly once so the program can rely on the reduce function being processed for correctness.

R To run the shuffle phase correctly, the mappers needs to know the number of reducers beforehand. Since the BSP model does not allow any data exchange between processors during one stage, Hadoop requires the number of reducers to be known before the entire MapReduce program starts. The default value is one which causes only one reducer to work. A rule of thumb is to set the number of reducers to 0.95 or 1.75 multiplied by number of executors. This allows reducers to run in one or two waves with a little bit of room for a few failing reducers to be rescheduled.

2.5.4 Spark Resilient Distributed Dataset (RDD)

This section focuses on how Spark RDD interface runs jobs in parallel based on concepts described earlier in this chapter. In Spark, RDD represents a set of records. This set has two important properties. First, it is *distributed* meaning that the set is stored on multiple machines which are the executors. Second, it is *resilient* which means that if part of it gets lost, e.g., due to a machine failure, Spark can recover the missing parts. An RDD represents a vertex in the application DAG. There are generally two ways to create an RDD. First, an input RDD is created by loading the objects from an external source or generating them, e.g., loading a text file from disk. Second, an intermediate or output RDD is created by applying a transformation on another RDD (or RDDs).

For example, in the word count example, the operation `sc.textFile("input.txt")` creates an input RDD that loads lines from a text file. This is an input RDD since it does not depend on any other RDDs. Similarly, the transformation `flatMap` creates a second RDD by transforming the input RDD. This is an intermediate RDD since it depends on another (parent) RDD.

2.5.5 Input RDD

The input RDD is a top-level RDD with no parent. It can also be called a source RDD since it is a source of objects to the Spark application. An input RDD needs to create two main functions, `createPartitions` and `compute`.

The `createPartitions` function returns a list of n partitions. A partition is a very small object that identifies a partition of the RDD. It usually contains information on how to compute the objects in the partition and not the objects themselves. Spark calls this function on the *driver node* before the job is executed to know how many partitions the input RDD has. It does not know or care what the partition objects contain. However, these objects need to be serialized over network to the executor nodes where records are created.

Spark needs to know the number of partitions in each RDD in the job before the job is launched. In other words, Spark does not allow the number of partitions to be changed dynamically at run time. This simplifies the execution of the job, especially, the communication step that connects one stage to another.

The `compute` method takes as input a partition identifier and returns an iterator of objects in that partition. For example, it can read the contents from a file or generate them. The compute function is executed on the executor nodes which allows the processing to happen in parallel.

■ **Example 2.3 — Text File RDD.** An example of an input RDD is the text file RDD which reads lines from the input as text[2]. The `createPartitions` function returns a set of file splits[3] where each split is defined by three values, file name, offset, and length. Each split defines a part of a file that starts at the given offset and contains a specific number of bytes. The `compute` function takes a split, opens the file, seeks to the offset, and returns an iterator that will return all lines found until the end of the split.. ■

The input RDD defines additional functions to optimize the processing. One of these functions is `getPreferredLocations` which defines a set of machine names where it is preferable to process a given partition. When reading a partition from HDFS, this method will return the set of datanodes that physically store that block. Spark will *try* to respect these preferred locations but there is no guarantee that the partition will be processed in one of these machines.

[2]Spark reuses Hadoop TextInputFormat to load text files but the design is similar to the explanation given in this section

[3]The term split was originally defined by Hadoop MapReduce and Spark still uses it in its internal code as a synonym to partition

R Spark uses a *pull*-based interface between RDDs. This means that each partition in an RDD provides an iterator that the next phase can pull the records from one-by-one. Other systems, such as AsterixDB, use a *push*-based interface where each partition pushes the record to the next phase through a special method designed for that. Hadoop MapReduce is hybrid as it mixes pull and push interfaces. For example, the map function gets an iterator to pull the input records and an output context that the map function will push the output records to. This requires Hadoop to materialize the records between stages to support this mix of interfaces.

The pull-based interface has an advantage that a record is produced when it is ready to be used by its consumer which can reduce buffering between the different parts. However, it is hard to implement some algorithms using this interface since you cannot simply write a while loop that generates all the records. On the other hand, the push-based interface is natural to use with existing algorithms that produce the output progressively. However, it has a disadvantage that a record might be pushed to a consumer when it is not yet ready to use it. In this case, the record has to be buffered which might result in memory or storage overhead.

2.5.6 Transformation

A transformation defines how one or more parent RDDs are transformed into one child RDD. For example, a *filter* transformation applies a predicate to each record in the parent RDD and only the ones that satisfy the predicate make it to the child RDD. Spark defines a rich set of transformations that developers can use. Furthermore, most of these transformations can be customized with a user-defined function, e.g., the predicate in the filter transformation. The user-defined function has to be stateless and deterministic as defined in Section 2.2.

There are two ways to implement a transformation in Spark for *primitive* and *compound* transformations. A primitive transformation does not depend on any other transformation and is implemented as a class that inherits from the base RDD class. Similar the input RDD, an intermediate RDD needs to implement the `getPartitions` and `compute` functions. In addition, it needs to define a set of *dependencies*. A dependency defines for one child RDD (returned by getPartitions) which partitions in the parent RDDs it depends on. That is why it does not need to be implemented for an input RDD that has no parents. In summary, the dependencies define a set of connections between the partitions in two RDDs. Depending on how these connections look like, Spark define the dependency as either a *narrow* or a *wide* dependency.

Definition 2.5.4 — Narrow Dependency. In a narrow dependency, each partition in the parent RDD is used by at most one partition of the child RDD.

Definition 2.5.5 — Wide Dependency. In a wide dependency, one partition in the parent RDD may be depended on by multiple child partitions.

When Spark detects a series of transformation with narrow dependencies, it will combine of all them in one stage with no synchronization in between. This improves the performance since the records can be pipelined between the compute functions without having to materialize them to memory or disk. When Spark detects a wide dependency, this is when it breaks down the processing into two stages and adds a synchronization barrier in between. Spark will also cache the intermediate records in memory, and spill to disk as needed, to follow the BSP execution model.

2.5.7 Actions

An action tells Spark what to do with the records in an RDD. An action is *required* to have a complete Spark job. In Spark, the action is defined by two functions, *partition processor* and *result handler*. The partition processor function runs on each partition of the output RDD and produces a result object r. When all partitions are processed, the ID of each partition and the correspond result $\langle i, r_i \rangle$ are sent to the result handler to complete the action (and the entire job). For example, in the '`saveAsTextFile`' action, the partition processor will write the contents of each partition to

a separate temporary file and return the path to the file as a result. The result handler will move all the intermediate result files to the output directory. This way, if one of the partitions fail while processing, its temporary file will not appear in the final result file since only the successful ones will be handler by the result handler.

2.5.8 Examples

In this part, we give examples of how common distributed operations are implemented. The goal is to better understand the flexibility of big-data systems regardless of the constraints imposed by the models described earlier.

Filter

The filter transformation takes a user-defined predicate p and applies it to every record in the parent RDD. Only the records that satisfy the predicate will be in the filtered (child) RDD. When the filter transformation is applied on an RDD with records of type T the filter function must take an input of type T and return a Boolean. The filtered RDD will have the same type T with the same or fewer records. Filter is a narrow-dependency transformation that applies on one record at a time. The filtered RDD has the same number of partitions in the parent RDD with a one-to-one dependency between them. The compute function scans the records in one partition, applies the predicate to each record, and adds the matching ones to the corresponding partition in the filtered RDD.

Map

The map transformation applies a function to each record in the parent RDD and adds the result of the function to the mapped (child) RDD. When applied to an RDD of type T, the user-provided map function must take an input of type T and return any output, say U. The mappped RDD will always have the same number of partitions and same number of records as the parent RDD. The dependency is exactly the same as the filter RDD. The compute function scans the records in one partition in the parent RDD, applies the function, and adds the result to the corresponding partition in the output.

Map Partition

The map partition transformation applies a function to *all* records in one partition and produces a *set* of records in the mapped RDD. When applied to an RDD of type T, the user-provided function should take an input of type $Iterator[T]$ and return a value of type $Iterator[U]$ where T and U might be of different classes. Similar to the filter and map transformations, the mapped RDD has the same number of partitions as the parent RDD with one-to-one dependency. The compute function is applied directly on the iterator of records in one partition in the parent RDD and its output is used to add records to the mapped RDD. One common case where the mapPartition transformation is useful is when there is a costly initialization step that you do not want to run for each record individually.

Both the filter and map transformations can be implemented using the map partition transformation but they are easier to use when they are applicable. In fact, they are internally implemented using one type of RDD, called MapPartitionsRDD. The MapPartitionsRDD class is used to implement a wide set of transformations including, filter, map, flatMap, mapPartitions, and mapPartitionsWithIndex. Therefore, there is no significant performance gain in using one of these over the others. You can use whichever is suitable for your program.

Reduce

Reduce is an action that reduces an RDD of type T into a single value T. It takes a single user-defined function that takes as input two values of type T and returns one value of type T. Logically, the function is applied repeatedly on the entire dataset to produce one value. Spark runs it in parallel

by first applying it in each partition in parallel using the local processing function. Each partition produces a single value which is passed to the result handler. The result handler will apply the same function on the partial results to produce a single value that is then returned to the user.