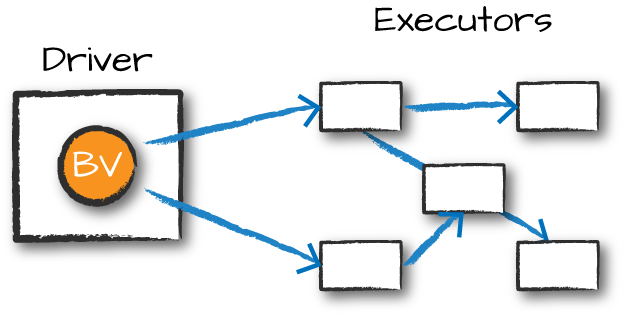
**Chapter 14. Distributed Shared Variables**

In addition to the Resilient Distributed Dataset (RDD) interface, the second kind of low-level API in Spark is two types of “distributed shared variables”: broadcast variables and accumulators. These are variables you can use in your user-defined functions (e.g., in a map function on an RDD or a DataFrame) that have special properties when running on a cluster. Specifically, *accumulators* let you add together data from all the tasks into a shared result (e.g., to implement a counter so you can see how many of your job’s input records failed to parse), while *broadcast variables* let you save a large value on all the worker nodes and reuse it across many Spark actions without re-sending it to the cluster. This chapter discusses some of the motivation for each of these variable types as well as how to use them.

**Broadcast Variables**

Broadcast variables are a way you can share an immutable value efficiently around the cluster without encapsulating that variable in a function closure. The normal way to use a variable in your driver node inside your tasks is to simply reference it in your function closures (e.g., in a map operation), but this can be inefficient, especially for large variables such as a lookup table or a machine learning model. The reason for this is that when you use a variable in a closure, it must be deserialized on the worker nodes many times (one per task). Moreover, if you use the same variable in multiple Spark actions and jobs, it will be re-sent to the workers with every job instead of once.

This is where broadcast variables come in. Broadcast variables are shared, immutable variables that are cached on every machine in the cluster instead of serialized with every single task. The canonical use case is to pass around a large lookup table that fits in memory on the executors and use that in a function, as illustrated in Figure 14-1.



*Figure 14-1. Broadcast variables*

For example, suppose that you have a list of words or values:

*// in Scala*

**val** myCollection **=** "Spark The Definitive Guide : Big Data Processing Made Simple"

.split(" ")

**val** words **=** spark.sparkContext.parallelize(myCollection, 2)

*# in Python*

my\_collection = "Spark The Definitive Guide : Big Data Processing Made Simple"\

.split(" ")

words = spark.sparkContext.parallelize(my\_collection, 2)

You would like to supplement your list of words with other information that you have, which is many kilobytes, megabytes, or potentially even gigabytes in size. This is technically a right join if we thought about it in terms of SQL:

*// in Scala*

**val** supplementalData **=** **Map**("Spark" -> 1000, "Definitive" -> 200,

"Big" -> -300, "Simple" -> 100)

*# in Python*

supplementalData = {"Spark":1000, "Definitive":200,

"Big":-300, "Simple":100}

We can broadcast this structure across Spark and reference it by using suppBroadcast. This value is immutable and is lazily replicated across all nodes in the cluster when we trigger an action:

*// in Scala*

**val** suppBroadcast **=** spark.sparkContext.broadcast(supplementalData)

*# in Python*

suppBroadcast = spark.sparkContext.broadcast(supplementalData)

We reference this variable via the value method, which returns the exact value that we had earlier. This method is accessible within serialized functions without having to serialize the data. This can save you a great deal of serialization and deserialization costs because Spark transfers data more efficiently around the cluster using broadcasts:

*// in Scala*

suppBroadcast.value

*# in Python*

suppBroadcast.value

Now we could transform our RDD using this value. In this instance, we will create a key–value pair according to the value we might have in the map. If we lack the value, we will simply replace it with 0:

*// in Scala*

words.map(word **=>** (word, suppBroadcast.value.getOrElse(word, 0)))

.sortBy(wordPair **=>** wordPair.\_2)

.collect()

*# in Python*

words.map(**lambda** word: (word, suppBroadcast.value.get(word, 0)))\

.sortBy(**lambda** wordPair: wordPair[1])\

.collect()

This returns the following value in Python and the same values in an array type in Scala:

[('Big', -300),

('The', 0),

...

('Definitive', 200),

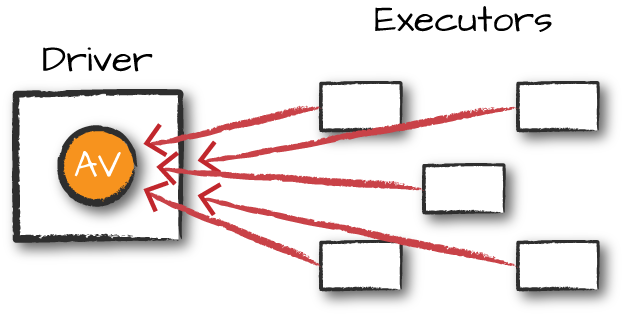
('Spark', 1000)]

The only difference between this and passing it into the closure is that we have done this in a much more efficient manner (Naturally, this depends on the amount of data and the number of executors. For very small data (low KBs) on small clusters, it might not be). Although this small dictionary probably is not too large of a cost, if you have a much larger value, the cost of serializing the data for every task can be quite significant.

One thing to note is that we used this in the context of an RDD; we can also use this in a UDF or in a Dataset and achieve the same result.

**Accumulators**

Accumulators (Figure 14-2), Spark’s second type of shared variable, are a way of updating a value inside of a variety of transformations and propagating that value to the driver node in an efficient and fault-tolerant way.



*Figure 14-2. Accumulator variable*

Accumulators provide a mutable variable that a Spark cluster can safely update on a per-row basis. You can use these for debugging purposes (say to track the values of a certain variable per partition in order to intelligently use it over time) or to create low-level aggregation. Accumulators are variables that are “added” to only through an associative and commutative operation and can therefore be efficiently supported in parallel. You can use them to implement counters (as in MapReduce) or sums. Spark natively supports accumulators of numeric types, and programmers can add support for new types.

For accumulator updates performed inside *actions only*, Spark guarantees that each task’s update to the accumulator will be applied only once, meaning that restarted tasks will not update the value. In transformations, you should be aware that each task’s update can be applied more than once if tasks or job stages are reexecuted.

Accumulators do not change the lazy evaluation model of Spark. If an accumulator is being updated within an operation on an RDD, its value is updated only once that RDD is actually computed (e.g., when you call an action on that RDD or an RDD that depends on it). Consequently, accumulator updates are not guaranteed to be executed when made within a lazy transformation like map().

Accumulators can be both named and unnamed. Named accumulators will display their running results in the Spark UI, whereas unnamed ones will not.

**Basic Example**

Let’s experiment by performing a custom aggregation on the Flight dataset that we created earlier in the book. In this example, we will use the Dataset API as opposed to the RDD API, but the extension is quite similar:

*// in Scala*

**case** **class** **Flight**(**DEST\_COUNTRY\_NAME:** **String**,

**ORIGIN\_COUNTRY\_NAME:** **String**, count**:** **BigInt**)

**val** flights **=** spark.read

.parquet("/data/flight-data/parquet/2010-summary.parquet")

.as[**Flight**]

*# in Python*

flights = spark.read\

.parquet("/data/flight-data/parquet/2010-summary.parquet")

Now let’s create an accumulator that will count the number of flights to or from China. Even though we could do this in a fairly straightfoward manner in SQL, many things might not be so straightfoward. Accumulators provide a programmatic way of allowing for us to do these sorts of counts. The following demonstrates creating an unnamed accumulator:

*// in Scala*

**import** **org.apache.spark.util.LongAccumulator**

**val** accUnnamed **=** **new** **LongAccumulator**

**val** acc **=** spark.sparkContext.register(accUnnamed)

*# in Python*

accChina = spark.sparkContext.accumulator(0)

Our use case fits a named accumulator a bit better. There are two ways to do this: a short-hand method and a long-hand one. The simplest is to use the SparkContext. Alternatively, we can instantiate the accumulator and register it with a name:

*// in Scala*

**val** accChina **=** **new** **LongAccumulator**

**val** accChina2 **=** spark.sparkContext.longAccumulator("China")

spark.sparkContext.register(accChina, "China")

We specify the name of the accumulator in the string value that we pass into the function, or as the second parameter into the register function. Named accumulators will display in the Spark UI, whereas unnamed ones will not.

The next step is to define the way we add to our accumulator. This is a fairly straightforward function:

*// in Scala*

**def** accChinaFunc(flight\_row**:** **Flight**) **=** {

**val** destination **=** flight\_row.**DEST\_COUNTRY\_NAME**

**val** origin **=** flight\_row.**ORIGIN\_COUNTRY\_NAME**

**if** (destination == "China") {

accChina.add(flight\_row.count.toLong)

}

**if** (origin == "China") {

accChina.add(flight\_row.count.toLong)

}

}

*# in Python*

**def** accChinaFunc(flight\_row):

destination = flight\_row["DEST\_COUNTRY\_NAME"]

origin = flight\_row["ORIGIN\_COUNTRY\_NAME"]

**if** destination == "China":

accChina.add(flight\_row["count"])

**if** origin == "China":

accChina.add(flight\_row["count"])

Now, let’s iterate over every row in our flights dataset via the foreach method. The reason for this is because foreach is an action, and Spark can provide guarantees that perform only inside of actions.

The foreach method will run once for each row in the input DataFrame (assuming that we did not filter it) and will run our function against each row, incrementing the accumulator accordingly:

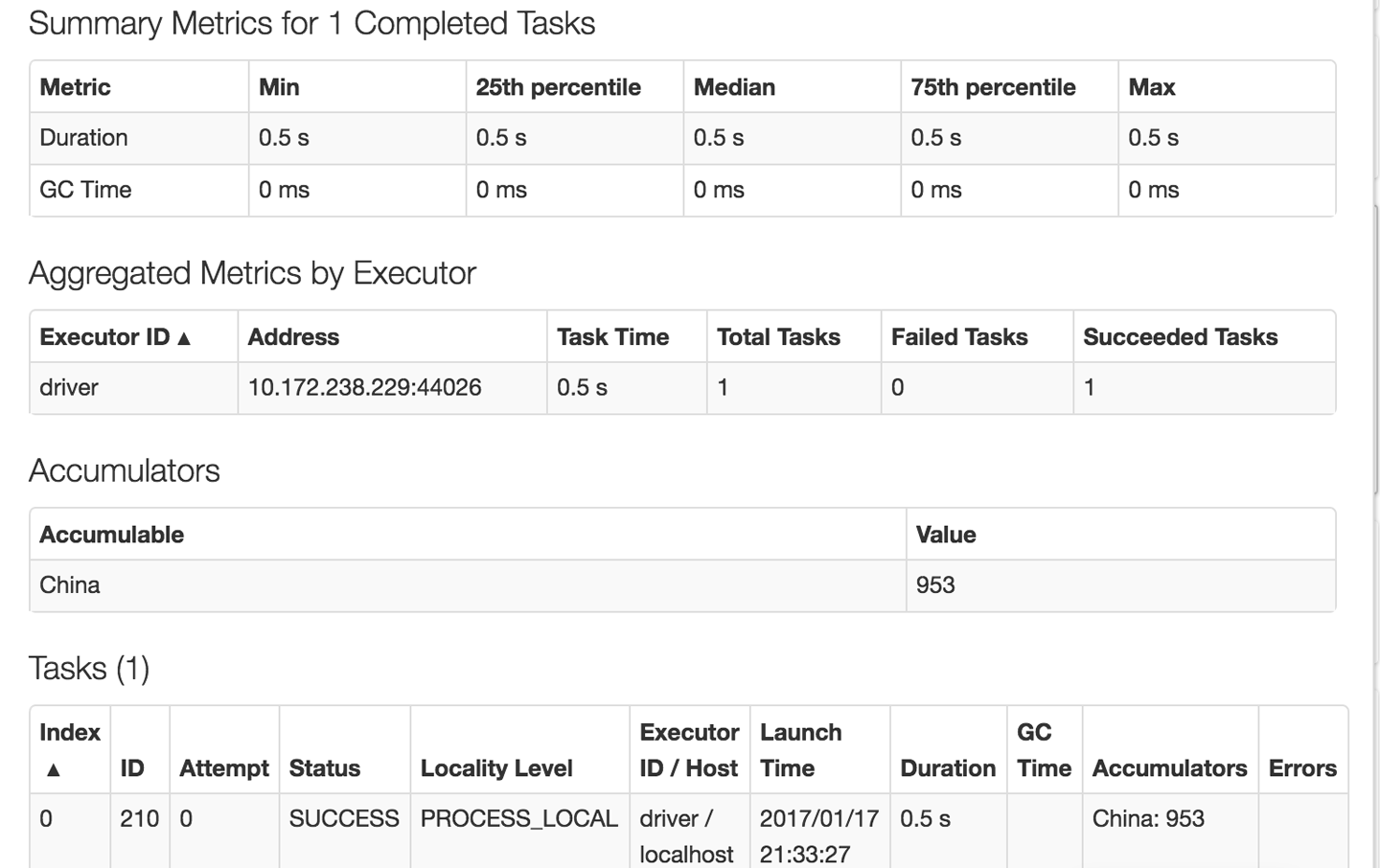
*// in Scala*

flights.foreach(flight\_row **=>** accChinaFunc(flight\_row))

*# in Python*

flights.foreach(**lambda** flight\_row: accChinaFunc(flight\_row))

This will complete fairly quickly, but if you navigate to the Spark UI, you can see the relevant value, on a per-Executor level, even before querying it programmatically, as demonstrated in Figure 14-3.



*Figure 14-3. Executor Spark UI*

Of course, we can query it programmatically, as well. To do this, we use the value property:

*// in Scala*

accChina.value *// 953*

*# in Python*

accChina.value *# 953*

**Custom Accumulators**

Although Spark does provide some default accumulator types, sometimes you might want to build your own custom accumulator. In order to do this you need to subclass the AccumulatorV2 class. There are several abstract methods that you need to implement, as you can see in the example that follows. In this example, you we will add only values that are even to the accumulator. Although this is again simplistic, it should show you how easy it is to build up your own accumulators:

*// in Scala*

**import** **scala.collection.mutable.ArrayBuffer**

**import** **org.apache.spark.util.AccumulatorV2**

**val** arr **=** **ArrayBuffer**[**BigInt**]()

**class** **EvenAccumulator** **extends** **AccumulatorV2**[**BigInt**, **BigInt**] {

**private** **var** num**:BigInt** = 0

**def** reset()**:** **Unit** = {

**this**.num **=** 0

}

**def** add(intValue**:** **BigInt**)**:** **Unit** = {

**if** (intValue % 2 == 0) {

**this**.num += intValue

}

}

**def** merge(other**:** **AccumulatorV2**[**BigInt**,**BigInt**])**:** **Unit** = {

**this**.num += other.value

}

**def** value()**:BigInt** = {

**this**.num

}

**def** copy()**:** **AccumulatorV2**[**BigInt**,**BigInt**] **=** {

**new** **EvenAccumulator**

}

**def** isZero()**:Boolean** = {

**this**.num == 0

}

}

**val** acc **=** **new** **EvenAccumulator**

**val** newAcc **=** sc.register(acc, "evenAcc")

*// in Scala*

acc.value *// 0*

flights.foreach(flight\_row **=>** acc.add(flight\_row.count))

acc.value *// 31390*

If you are predominantly a Python user, you can also create your own custom accumulators by subclassing [AccumulatorParam](https://spark.apache.org/docs/1.1.0/api/python/pyspark.accumulators.AccumulatorParam-class.html) and using it as we saw in the previous example.

**Conclusion**

In this chapter, we covered distributed variables. These can be helpful tools for optimizations or for debugging. In [Chapter 15](https://www.safaribooksonline.com/library/view/spark-the-definitive/9781491912201/ch15.html#s4c0---how-spark-runs-on-a-cluster), we define how Spark runs on a cluster to better understand when these can be helpful.