Chapter 14. Distributed Shared Variables

the second kind of low-levelAPI in Spark is two types of “**distributed shared variables**”

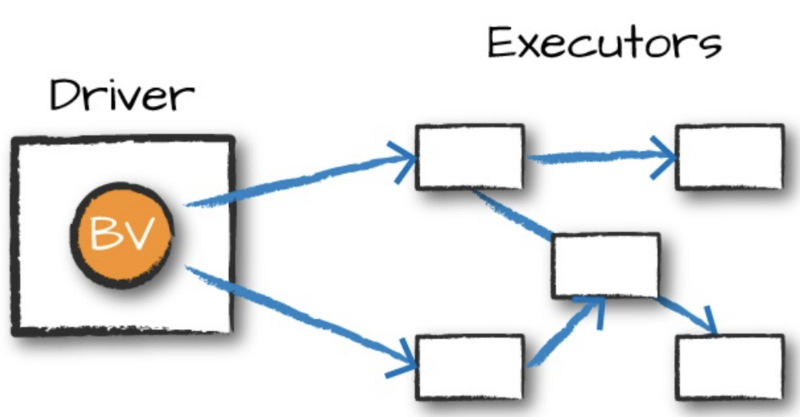
- broadcast variables - save a large value on all the worker nodes and reuse it across Spark actions without re-sending it to the cluster
- accumulators - add together data from all the tasks into a shared result (e.g.,to implement a counter so you can see how many of your job’s input records failed to parse)

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

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)
- this can be inefficient, especially for large variables such as a *lookup table* or a *machine learning model*
- when you use a variable in a closure, it must be **deserialized** on the worker nodes many times (one per task)
- 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**
- Broadcast variables are shared, immutable variables that are cached on **every machine in the cluster instead of serialized with every single task**
- typical use case is to pass around a large lookup table that fits in memory on the executors and use that in a function



Supplement your list of words with other information that you have about the list - as example, the number of kilobytes, megabytes, or gigabytes in size. This is technically a *right join* if we thought about it in terms of SQL:

```
supplementalData = {"Spark":1000, "Definitive":200,
                    "Big":-300, "Simple":100}
```

suppBroadcast

- broadcast this structure across Spark and reference it
- value is immutable and is lazily replicated across all nodes in the cluster when we trigger an action

```
suppBroadcast = spark.sparkContext.broadcast(supplementalData)
```

We reference this variable via the **value** method

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

```
suppBroadcast.value
```

transform our RDD using this value

- 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*

```
words.map(lambda word: (word, suppBroadcast.value.get(word, 0)))\
  .sortBy(lambda wordPair: wordPair[1])\
  .collect()
```

difference between this, and passing it into the closure, is that we have done this in a more efficient manner

- this depends on the amount of data and the number of executors
- with very small data (low KBs) on small clusters, it might not be more efficient
- 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

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

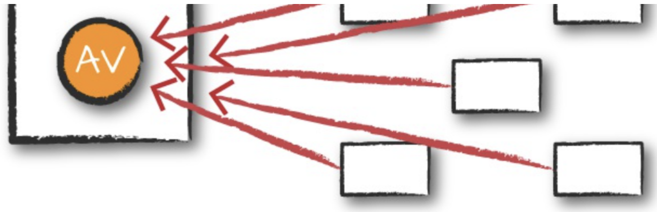


Figure 14-2. Accumulator variable

Accumulators

- provide a mutable variable that a Spark cluster can safely update on a per-row basis
- use these for debugging purposes (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
- with 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 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
    - unnamed ones will not

```
#use the Dataset API as opposed to the RDD API

path= "/FileStore/tables/part_r_00000_1a9822ba_b8fb_4d8e_844a_ea30d0801b9e_gz-1.parquet"

flights = spark.read\
  .parquet(path)
```

create an accumulator that will count the number of flights to or from China

- even though we could do this in a fairly straightforward 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**:

```
accChina = spark.sparkContext.accumulator(0)
```

Our use case fits a named accumulator

- There are two ways to do this: a short-hand method and long-hand one
- The simplest is to use the SparkContext.
- Alternatively can instantiate the accumulator and register it with a name

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 SparkUI, whereas unnamed ones will not
- Next step is to define the way we add to our accumulator

```
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"])
```

iterate over every row in our flights dataset via the **foreach** method

- foreach is an action, and Spark can provide guarantees that perform only inside of actions
- 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:

```
flights.foreach(lambda flight_row: accChinaFunc(flight_row))
```

relevant value, on a per-Executor level, even before querying it programmatically:

Summary Metrics for 1 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.5 s	0.5 s	0.5 s	0.5 s	0.5 s
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms

Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks
driver	10.172.238.229:44026	0.5 s	1	0	1

Accumulators

Accumulable	Value
China	953

Tasks (1)

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Accumulators	Errors
0	210	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2017/01/17 21:33:27	0.5 s		China: 953	

```
# can query it programmatically
accChina.value # 953
```

you might want to build your own **custom accumulator**

- In order to do this you need to subclass the **AccumulatorV2** class
- several abstract methods that you need to implement