**Understanding closures**

One of the harder things about Spark is understanding the scope and life cycle of variables and methods when executing code across a cluster. RDD operations that modify variables outside of their scope can be a frequent source of confusion. In the example below we’ll look at code that uses foreach() to increment a counter, but similar issues can occur for other operations as well.

**Example**

Consider the naive RDD element sum below, which may behave differently depending on whether execution is happening within the same JVM. A common example of this is when running Spark in local mode (--master = local[n]) versus deploying a Spark application to a cluster (e.g. via spark-submit to YARN):

* [**Scala**](http://spark.apache.org/docs/latest/rdd-programming-guide.html#tab_scala_7)
* [**Java**](http://spark.apache.org/docs/latest/rdd-programming-guide.html#tab_java_7)
* [**Python**](http://spark.apache.org/docs/latest/rdd-programming-guide.html#tab_python_7)

**var** counter **=** 0

**var** rdd **=** sc.parallelize(data)

*// Wrong: Don't do this!!*

rdd.foreach(x **=>** counter += x)

println("Counter value: " + counter)

**Local vs. cluster modes**

The behavior of the above code is undefined, and may not work as intended. To execute jobs, Spark breaks up the processing of RDD operations into tasks, each of which is executed by an executor. Prior to execution, Spark computes the task’s **closure**. The closure is those variables and methods which must be visible for the executor to perform its computations on the RDD (in this case foreach()). This closure is serialized and sent to each executor.

The variables within the closure sent to each executor are now copies and thus, when **counter** is referenced within the foreach function, it’s no longer the **counter** on the driver node. There is still a **counter** in the memory of the driver node but this is no longer visible to the executors! The executors only see the copy from the serialized closure. Thus, the final value of **counter** will still be zero since all operations on **counter** were referencing the value within the serialized closure.

In local mode, in some circumstances, the foreach function will actually execute within the same JVM as the driver and will reference the same original **counter**, and may actually update it.

To ensure well-defined behavior in these sorts of scenarios one should use an [Accumulator](http://spark.apache.org/docs/latest/rdd-programming-guide.html#accumulators). Accumulators in Spark are used specifically to provide a mechanism for safely updating a variable when execution is split up across worker nodes in a cluster. The Accumulators section of this guide discusses these in more detail.

In general, closures - constructs like loops or locally defined methods, should not be used to mutate some global state. Spark does not define or guarantee the behavior of mutations to objects referenced from outside of closures. Some code that does this may work in local mode, but that’s just by accident and such code will not behave as expected in distributed mode. Use an Accumulator instead if some global aggregation is needed.

**Printing elements of an RDD**

Another common idiom is attempting to print out the elements of an RDD using rdd.foreach(println) or rdd.map(println). On a single machine, this will generate the expected output and print all the RDD’s elements. However, in cluster mode, the output to stdout being called by the executors is now writing to the executor’s stdout instead, not the one on the driver, so stdout on the driver won’t show these! To print all elements on the driver, one can use the collect() method to first bring the RDD to the driver node thus: rdd.collect().foreach(println). This can cause the driver to run out of memory, though, because collect() fetches the entire RDD to a single machine; if you only need to print a few elements of the RDD, a safer approach is to use the take(): rdd.take(100).foreach(println).