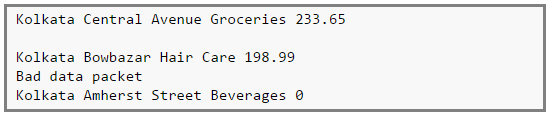
What are accumulators?

Accumulators are variables that are used for aggregating information across the executors. For example, this information can pertain to data or API diagnosis like how many records are corrupted or how many times a particular library API was called.

To understand why we need accumulators, let’s see a small example.

Here’s an imaginary log of transactions of a chain of stores around the central Kolkata region.



There are 4 fields,

*Field 1 -> City*

*Field 2 -> Locality*

*Field 3 -> Category of item sold*

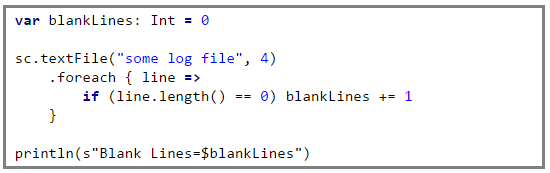
*Field 4 -> Value of item sold*

However, the logs can be corrupted. For example, the second line is a blank line, the fourth line reports some network issues and finally the last line shows a sales value of zero (which cannot happen!).

We can use accumulators to analyse the transaction log to find out the number of blank logs (blank lines), number of times the network failed, any product that does not have a category or even number of times zero sales were recorded. The full sample log can be found [here](https://github.com/prithvirajbose/spark-dev/blob/master/data/purchases.log).  
Accumulators are applicable to any operation which are,  
1. Commutative -> *f(x, y) = f(y, x)*, and  
2. Associative -> *f(f(x, y), z) = f(f(x, z), y) = f(f(y, z), x)*  
For example, *sum* and *max* functions satisfy the above conditions whereas *average* does not.

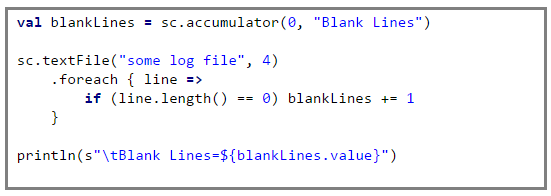
Why use Spark Accumulators?

Now why do we need accumulators and why not just use variables as shown in the code below.



The problem with the above code is that when the driver prints the variable *blankLines* its value will be zero. This is because when Spark ships this code to every executor the variables become local to that executor and its updated value is not relayed back to the driver. To avoid this problem we need to make *blankLines* an accumulator such that all the updates to this variable in every executor is relayed back to the driver.

So the above code should be written as,



This guarantees that the accumulator *blankLines*is updated across every executor and the updates are relayed back to the driver.

We can implement other counters for network errors or zero sales value, etc. The full source code along with the implementation of the other counters can be found [here](https://github.com/prithvirajbose/spark-dev/blob/master/src/main/scala/examples/PurchaseLogAnalysis.scala).

People familiar with Hadoop Map-Reduce will notice that Spark’s accumulators are similar to Hadoop’s Map-Reduce counters.

Caveats

When using accumulators there are some caveats that we as programmers need to be aware of,

1. Computations inside *transformations* are evaluated lazily, so unless an*action* happens on an RDD the*transformations*are not executed. As a result of this, accumulators used inside functions like *map()*or *filter()* wont get executed unless some *action* happen on the RDD.
2. Spark guarantees to update accumulators **inside *actions*only once**. So even if a task is restarted and the lineage is recomputed, the accumulators will be updated only once.
3. Spark does not guarantee this for *transformations*. So if a task is restarted and the lineage is recomputed, there are chances of undesirable side effects when the accumulators will be updated more than once.

**To be on the safe side, always use accumulators inside actions ONLY.**  
The code [here](https://github.com/prithvirajbose/spark-dev/blob/master/src/main/scala/examples/PurchaseLogAnalysis.scala) shows a simple yet effective example on how to achieve this.  
For more information on accumulators, read [this](http://spark.apache.org/docs/latest/rdd-programming-guide.html#accumulators).