**Spark Performance Tuning**

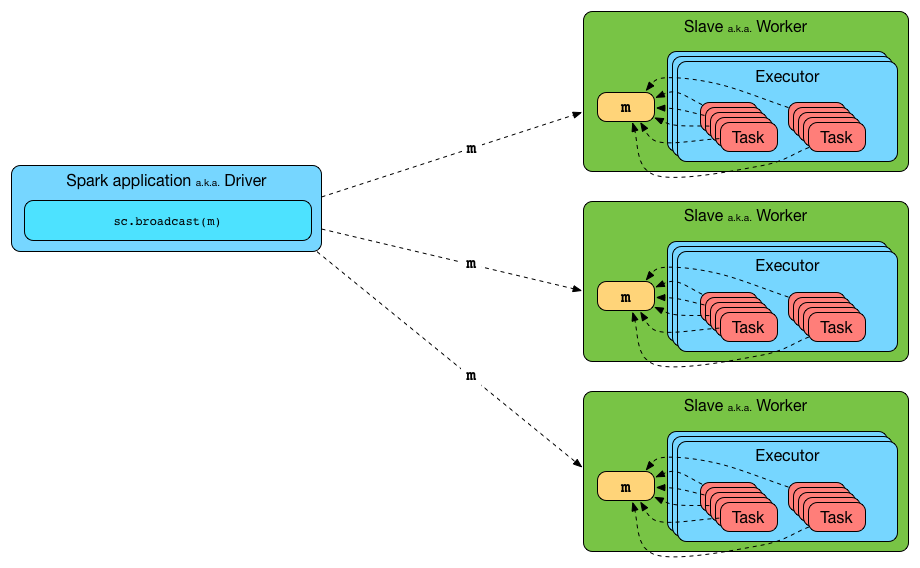
## Memory Tuning

Consider the following three things in tuning memory usage:

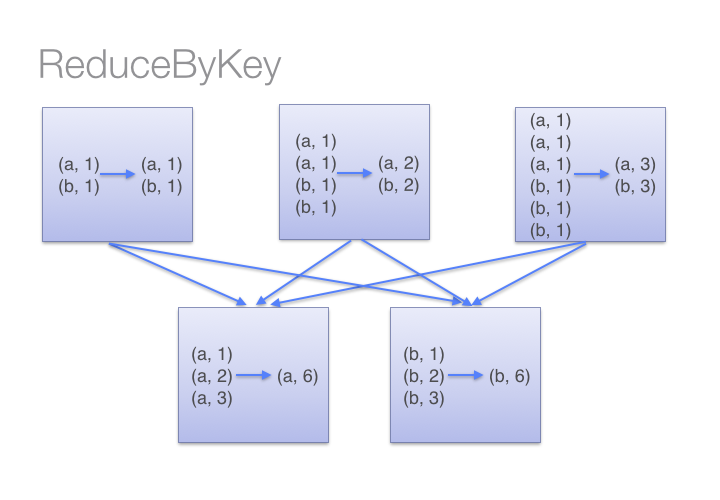
* Amount of memory used by objects (the entire dataset should fit in-memory)
* The cost of accessing those objects
* Overhead of garbage collection.

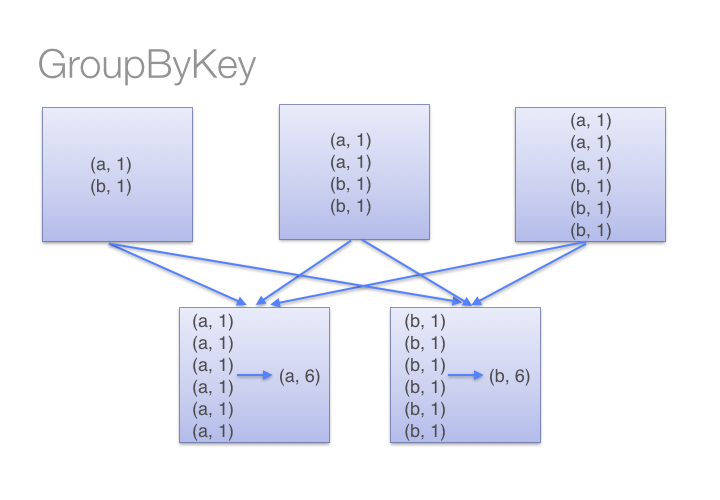
### **Broadcasting Large Variables**

The size of each serialized task reduces by using broadcast functionality in **[SparkContext](http://data-flair.training/blogs/sparkcontext-in-apache-spark-tutorial/).** If a task uses a large object from driver program inside of them, turn it into the broadcast variable.



[**Avoid GroupByKey**](https://databricks.gitbooks.io/databricks-spark-knowledge-base/content/best_practices/prefer_reducebykey_over_groupbykey.html) **use reduceByKey**

* Here are more functions to prefer over groupByKey:
* combineByKey can be used when you are combining elements but your return type differs from your input value type.
* foldByKey merges the values for each key using an associative function and a neutral "zero value".
* 



**Use lookup function rather than left join**

**Use memoryanddiskserialization when persist.**

[**Don't copy all elements of a large RDD to the driver**](https://databricks.gitbooks.io/databricks-spark-knowledge-base/content/best_practices/dont_call_collect_on_a_very_large_rdd.html)

* If your RDD is so large that all of it's elements won't fit in memory on the drive machine, don't do this:
* val values = myVeryLargeRDD.collect()
* Collect will attempt to copy every single element in the RDD onto the single driver program, and then run out of memory and crash.
* Instead, you can make sure the number of elements you return is capped by calling take or takeSample, or perhaps filtering or sampling your RDD.

[**Gracefully Dealing with Bad Input Data**](https://databricks.gitbooks.io/databricks-spark-knowledge-base/content/best_practices/best_practices/dealing_with_bad_data.html)

* Handle exception handling, rdd.isempty check, count the number of elements in the array, use dataset for typesafe.

# **Data Locality**

# Spark is a data parallel processing framework, which means it will execute tasks as close to where the data lives as possible (i.e. minimize data transfer).

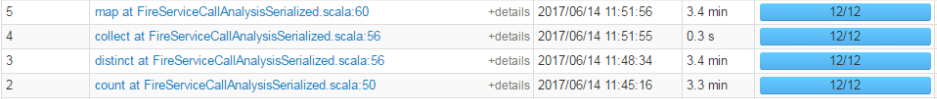
## Data Serialization (Kyro/Tungsten)

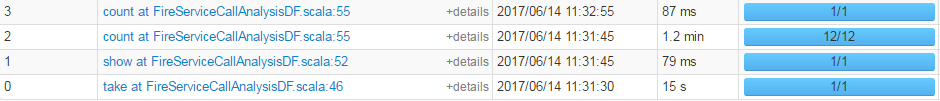
It is the process of converting the in-memory object to another format that can be used to store in a file or send over the network. It plays a distinctive role in the performance of any distributed application. The computation gets slower due to formats that are slow to serialize or consume a large number of files.[Apache Spark](http://data-flair.training/blogs/apache-spark-introduction-tutorial/)gives serialization libraries:

***conf.set(“spark.serializer”, “org.apache.spark.serializer.KyroSerializer****”)*

#### **Straggler Tasks (Long Running Tasks)**

The straggler tasks can be identified in the Stages view and take a long time to complete. In this use case, the following are the straggler tasks that took longer time.

**RDD Implementation Straggler Task**

**DataFrame Implementation Straggler Task**

**Use Dataframes/Datasets as much as possible**

Apache Spark 2.x version ships with the second-generation Tungsten engine. This engine is built upon ideas from modern compilers to emit optimized code at runtime that collapses the entire query into a single function by using “whole-stage code generation” technique. Thereby, eliminating virtual function calls and leveraging CPU registers for intermediate data. This optimization is applied only to Spark high-level APIs such as DataFrame and Dataset and not to low-level RDD API.

* Dataset API gains the advantage of Spark’s optimizers such as Catalyst and Tungsten.
* Datasets acquire two discrete APIs characteristics such as strongly typed and untyped.

Let us first decide the number of partitions based on the input dataset size. The rule of thumb to decide the partition size while working with HDFS is **128 MB**. As our input dataset size is about 1.5 GB (1500 MB) and going with 128 MB per partition, the number of partitions will be:

Total input dataset size / partition size => 1500 / 128 = 11.71 =**~12 partitions.**

## Checking Locality

## The best means of checking whether a task ran locally is to inspect a given stage in the Spark UI. Notice from the screenshot below that the "Locality Level" column displays which locality a given task ran with.

## Locality