**Spark Performance Tuning**

[**https://umbertogriffo.gitbooks.io/apache-spark-best-practices-and-tuning/content/which\_storage\_level\_to\_choose.html**](https://umbertogriffo.gitbooks.io/apache-spark-best-practices-and-tuning/content/which_storage_level_to_choose.html)

**Use lookup rather than left join**

**Use memoryanddiskserialization when persist.**

## Memory Tuning

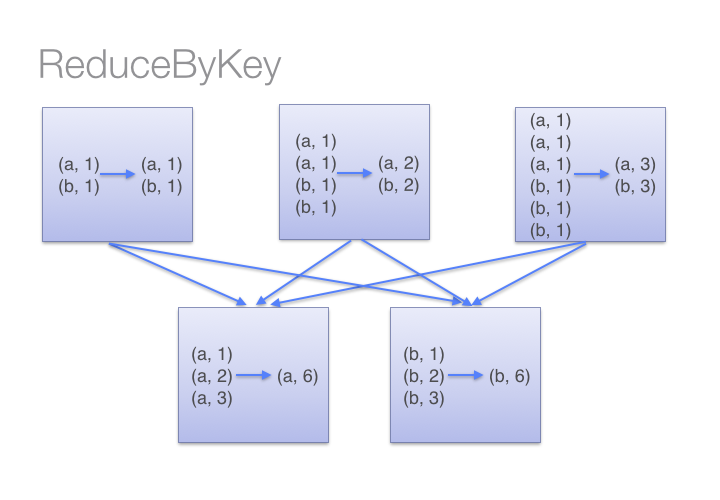
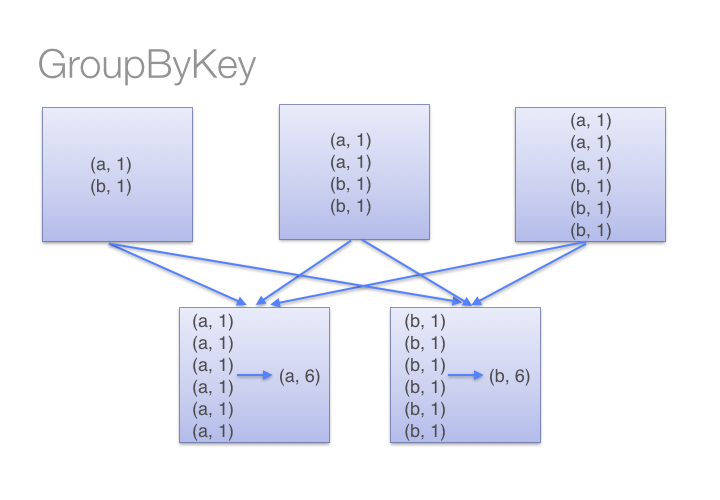
Consider the following three things in tuning memory usage:

* 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. Generally, it considers the tasks that are about 20 Kb for optimization.

[**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".
* 
* 

[**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**

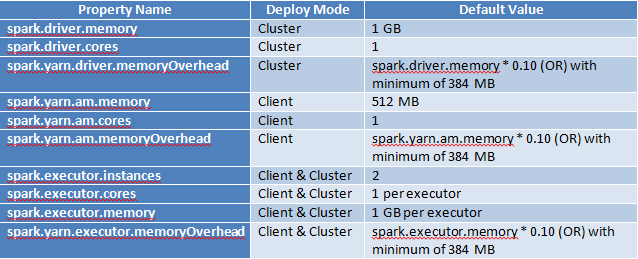
# Spark is a data parallel processing framework, which means it will execute tasks as close to where the data lives as possible (i.e. minimize data transfer).

## 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

You can adjust how long Spark will wait before it times out on each of the phases of data locality (data local --> process local --> node local --> rack local --> Any). For more information on these parameters, see the **spark.locality.\*** configs



#### **Resource Planning (Executors, Core, and Memory)**

A balanced number of executors, core, and memory will significantly improve the performance without any code changes in the Spark application while running on YARN.

## Spark on YARN: Resource Planning

Let's find out the reasonable resources to execute the Spark application in YARN.

|  |  |
| --- | --- |
| **Memory available per node** | 8 GB |
| **Core available per node** | 3 |

To find out the number of executors, cores, and memory and its works for our use case with notable performance improvement, perform the following steps:

### **Step 1**

Allocate 1 GB memory and 1 core for driver per node. The driver can be launched at any one of the nodes at run time. If the output of the action returns more data (for example, more than 1 GB), then driver memory must be adjusted.

|  |  |
| --- | --- |
| **Memory available per node** | 7 GB |
| **Core available per node** | 2 |

### **Step 2**

Assign 1 GB memory and 1 core for OS and Hadoop Daemons overhead per instance.

Let's look at the below instance to launch the cluster.

**Instance details**: m4.xlarge (4 cores, 16 GB RAM)

1 core and 8 GB RAM are freed up for other resources and YARN is configured with 8 GB RAM and 3 cores per node. The freed-up resource will be used on OS and Hadoop Daemons overhead. Memory available and core available per node remains unchanged after Step 2.

### **Step 3**

Find out the number of cores per executor.

As 2 cores per node are available, decide the number of cores as 2 per executor.

**Note**: If you have more cores per instance (for example, 16 – 1(overhead) = 15), then stick with the number of cores per executor as 5 while running in YARN with HDFS due to high HDFS throughput.

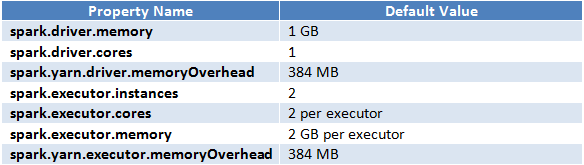
### **Step 4**

Find out the number of executors and the memory per executors.

* **Number of cores per executor**: 2 Total cores = Number of nodes \* Number of cores per node (after taking overhead) => 2 \* 2 = 4
* **Total Executors**: 2
  + Total executors = Total cores / Number of nodes => 4 / 2 = 2
  + Number of executors per node = Total executors / Number of nodes => 2/2 = 1
    - Each node will have one executor.
* **Memory per executor**: 7 GB (this must be adjusted as per the application payload).
  + Memory per node / Number of executors per node => 7 / 1 => 7 GB

This calculation works well with our use case except that the memory per executor as input dataset size is 1.5 GB and using 6 GB per executor to process 1.5 GB is like over using the memory.

Executor memory with 2 GB is applied and increased up to 7 GB per executor to execute the Spark application. 2 GB memory per executor is decided as there are no additional performance improvements while increasing executor memory from 2 GB to 7 GB.

The decided resource allocation derived from the above steps for the use case Spark applications is illustrated in the below table:

**Note**:Different organizations have different workloads and the above steps may not work well for all cases, but you can get an idea of calculating executors, cores, and memory.

* Ensure proper use of all resources in an effective manner.
* Eliminates those jobs that run long.
* Improves the performance time of the system.
* Guarantees that jobs are on correct execution engine.

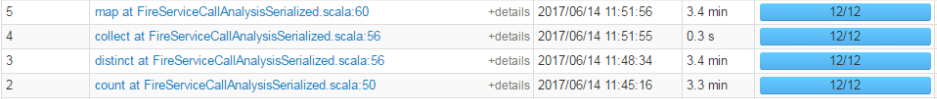
## Data Serialization (Kyro/Tungsten)

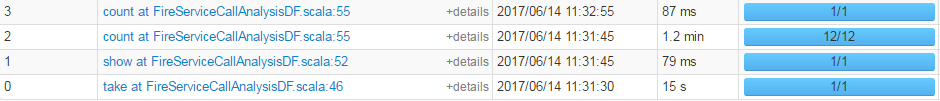
It is the process of converting the in-memory object to another format that can be used to store in a file or send over the network. It plays a distinctive role in the performance of any distributed application. The computation gets slower due to formats that are slow to serialize or consume a large 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.**

This is equal to the Spark default parallelism (spark.default.parallelism) value. The metrics based on default parallelism are shown in the above section.

**spark-submit --name jobname --master yarn --deploy-mode cluster --executor-memory 2g --executor-cores 2 --num-executors 2 --conf spark.sql.shuffle.partitions=23 --conf spark.default.parallelism=23** --class com.inceptez.perftuningclass /home/hduser/perf.jar /user/hduser/usdataset.csv

select