# Spark Job Configuration

## 1. Data Serialization

Spark is a distributed computation engine, which requires send and receive data(objects) through network. Serialization and De-serialization plays an important role in the performance. Formats and data structures that are slow to serialize objects into, or consume a large number of bytes, will greatly slow down the computation.

As a result, for a well optimized spark application, data serialization is the first thing you need to take into account. Spark supports two serialization libraries:

- **Java serialization**: By default, Spark serializes objects using Java's ObjectOutputStream framework, and can work with any class you create that implements java.io.Serializable. You can also control the performance of your serialization more closely by extending java.io.Externalizable. **Java serialization is flexible but often quite slow, and leads to large serialized formats for many classes**, so not recommended for large spark jobs
- **Kryo serialization**: Spark can also use the Kryo library (version 4) to serialize objects more quickly. **Kryo is significantly faster and more compact than Java serialization (often as much as 10x), but does not support all Serializable types and requires you to register the classes you'll use in the program in advance for best performance**.

### 1.1 Configure Spark job to use Kryo as serialization

Old school config by using spark context, may be deprecated in new version. So not recommended for new applications.

```
val conf = new SparkConf().setMaster("My spark cluster resource manager
url").setAppName("My spark application name")
conf.registerKryoClasses(Array(classOf[java.lang.Class],
classOf[org.apache.spark.sql.catalyst.InternalRow],...))
val sc = new SparkContext(conf)
```

```
//create a spark session who works with Kryo.
object SparkSessionKryo {
    def getSparkSession: SparkSession = {
        val spark = SparkSession
            .builder
            .master("My spark cluster resource manager url")
            .appName("My spark application name")
            .config(getConfig)
            .config("spark.serializer",
"org.apache.spark.serializer.KryoSerializer")
            // use this if you need to increment Kryo buffer size. Default
```

```
64k
            .config("spark.kryoserializer.buffer", "1024k")
            // use this if you need to increment Kryo buffer max size.
Default 64m
            .config("spark.kryoserializer.buffer.max", "1024m")
            /*
            * Use this if you need to register all Kryo required classes.
            * If it is false, you do not need register any class for Kryo,
but it will increase your data size when the data is serializing.
            */
            .config("spark.kryo.registrationRequired", "true")
            .getOrCreate
    }

    private def getConfig = {
        val conf = new SparkConf()
        conf.registerKryoClasses(
          Array(
            classOf[scala.collection.mutable.WrappedArray.ofRef[_]],
            classOf[org.apache.spark.sql.types.StructType],
            classOf[Array[org.apache.spark.sql.types.StructType]],
            classOf[org.apache.spark.sql.types.StructField],
            classOf[Array[org.apache.spark.sql.types.StructField]],
            Class.forName("org.apache.spark.sql.types.StringType$"),
            Class.forName("org.apache.spark.sql.types.LongType$"),
            Class.forName("org.apache.spark.sql.types.BooleanType$"),
            Class.forName("org.apache.spark.sql.types.DoubleType$"),
            Class.forName("[[B"),
            classOf[org.apache.spark.sql.types.Metadata],
            classOf[org.apache.spark.sql.types.ArrayType],
Class.forName("org.apache.spark.sql.execution.joins.UnsafeHashedRelation"),
            classOf[org.apache.spark.sql.catalyst.InternalRow],
            classOf[Array[org.apache.spark.sql.catalyst.InternalRow]],
            classOf[org.apache.spark.sql.catalyst.expressions.UnsafeRow],
Class.forName("org.apache.spark.sql.execution.joins.LongHashedRelation"),
Class.forName("org.apache.spark.sql.execution.joins.LongToUnsafeRowMap"),
            classOf[org.apache.spark.util.collection.BitSet],
            classOf[org.apache.spark.sql.types.DataType],
            classOf[Array[org.apache.spark.sql.types.DataType]],
            Class.forName("org.apache.spark.sql.types.NullType$"),
            Class.forName("org.apache.spark.sql.types.IntegerType$"),
            Class.forName("org.apache.spark.sql.types.TimestampType$"),
Class.forName("org.apache.spark.sql.execution.datasources.FileFormatWriter$W
riteTaskResult"),
Class.forName("org.apache.spark.internal.io.FileCommitProtocol$TaskCommitMes
sage"),
            Class.forName("scala.collection.immutable.Set$EmptySet$"),
            Class.forName("scala.reflect.ClassTag$$anon$1"),
            Class.forName("java.lang.Class")
          )
          )
      }
}
```

In the above config, we set spark.kryoserializer.buffer = 1024K. This value needs to be large enough to hold the largest object you will serialize. We also set spark.kryoserializer.buffer.max=1024M to avoid kryo takes all memory of spark worker

For more details about Kryo configuration, please visit the Kryo documentation (https://github.com/EsotericSoftware/kryo), which describes more advanced registration options.

Finally, if you don’t register your custom classes, Kryo will still work, but it will have to store the full class name with each object, which is wasteful.

## 2. Memory management inside spark application

Spark largely divide memory into two categories:

- Execution: It refers to memory that is used for computation in shuffles, joins, sorts, aggregations, etc.
- Storage: It refers to memory that is used for caching and propagating internal data across the cluster.

Execution and storage memory share a **unified region (M)**. When no execution memory is used, **storage can acquire all the available memory and vice versa. Execution may evict storage if necessary, but only until total storage memory usage falls under a certain threshold (R).** In other words, R describes a subregion within M where cached blocks are never evicted. **Storage may not evict execution due to complexities in implementation**.

As a result Execution has priority over Storage memory, and storage memory has a minimun reservation for data caching.

### 2.1 Reset default memory region size

There are two relevant configurations for **unified region (M)** and **cached subregion (R)**, the typical user should not need to adjust them as the default values are applicable to most workloads:

- spark.memory.fraction expresses the size of M as a fraction of the JVM heap space (default value is 0.6, means 60% of the memory). The rest of the space (40%) is reserved for user data structures, internal metadata in Spark, and safeguarding against OOM errors in the case of sparse and unusually large records.
- spark.memory.storageFraction expresses the size of R as a fraction of M (default 0.5). R is the storage space within M where cached blocks immune to being evicted by execution.

The value of spark.memory.fraction should be set in order to fit this amount of heap space comfortably within the JVM’s old or “tenured” generation. See the discussion of advanced GC tuning below for details.

## 2.2 Determining Memory Consumption

### 2.2.1 Dataset memory consumption

The best way to size the amount of memory consumption a dataset will require is to create an RDD, put it into cache, and look at the “Storage” page in the web UI. The page will tell you how much memory the RDD is occupying.

### 2.2.1 Application Object consumption

To estimate the memory consumption of a particular object, use **SizeEstimator’s estimate** method. This is useful for experimenting with different data layouts to trim memory usage, as well as determining the amount of space a broadcast variable will occupy on each executor heap.

```
import org.apache.spark.util.SizeEstimator

val someObj

//estimate is a static method, so no need to create an instance
SizeEstimator.estimate(someObj)

// estimate will return a Long in byte, for example if it returns
115'715'808 (bytes), it means omeObj will consume 116MB in cached object
region of spark application
```

## 2.3 Reduce memory consumption by choosing right data structures

To reduce memory consumption in spark jobs, we need to avoid the Java features that add overhead, such as pointer-based data structures and wrapper objects. There are several ways to do this:

### 2.3.1 Use primitive type

**Use primitive type** as much as you can, don't use their wrapper classes.(Use int instead of Integer). Each distinct Java object has an “object header”, which is about 16 bytes and contains information such as a pointer to its class.

For example, a int of 32bits(4 Bytes), the Integer object header(16 Bytes) is greater the actual data. Java Strings have about 40 bytes of overhead over the raw string data (since they store it in an array of Chars and keep extra data such as the length), and store each character as two bytes due to String’s internal usage of UTF-16 encoding. Thus a 10-character string can easily consume 60 bytes.

### 2.3.2 Don't use string for keys

Consider using numeric IDs or enumeration objects instead of strings for keys.

### 2.3.3 Use array

Don't use standard Java or Scala collection classes(e.g. HashMap, LinkedList, etc.). Use array in preference. If you have to use collection classes in your code, check **fastutil** (http://fastutil.di.unimi.it/#big) lib, which provides convenient collection classes with a small memory footprint and fast access and insertion.

They are too main reasons for this.

- Common collection classes, such as HashMap and LinkedList, use linked data structures, where there is a “wrapper” object for each entry (e.g. Map.Entry). This object not only has a header, but also pointers (typically 8 bytes each) to the next object in the list.
- Java common Collection classes does not support primitive types, it store them as “boxed” objects such as java.lang.Integer which add unnecessary overhead.

## 2.4 Reduce object overhead by changing JVM config

In spark-env.sh, you can set the JVM flag -XX:+UseCompressedOops to make pointers be four bytes instead of eight.

For example, in spark-env.sh, you can set the jvm flag for both driver and executor.

```
# reduce pointer size for driver and executor
spark.driver.extraJavaOptions=-XX:+UseCompressedOops
spark.executor.extraJavaOptions=-XX:+UseCompressedOops

#If you have many jvm options to add, just separate them by space, for
example we want to print GC details
spark.executor.extraJavaOptions=-XX:+UseCompressedOops -XX:+PrintGCDetails -
XX:+PrintGCTimeStamps
```

If you don't want to do this in your spark-env.sh. You can change it also in your sparkSession

```
object SparkSessionReducePointerSize {
    def getSparkSession: SparkSession = {
        val spark = SparkSession
            .builder
            .master("My spark cluster resource manager url")
            .appName("My spark application name")
            .config("spark.executor.extraJavaOptions", "-
XX:+UseCompressedOops")
            .config("spark.driver.extraJavaOptions", "-
XX:+UseCompressedOops")
            .getOrCreate
    }
```

In case of submit, you can change it in your submit command

```
./bin/spark-submit \
  --name "My app" \
  --master local[4] \
  --conf spark.eventLog.enabled=false \
  --conf "spark.driver.extraJavaOptions=-XX:+UseCompressedOops -
XX:+PrintGCDetails -XX:+PrintGCTimeStamps" \
  --conf "spark.executor.extraJavaOptions=-XX:+UseCompressedOops -
XX:+PrintGCDetails -XX:+PrintGCTimeStamps" \
  myApp.jar
```

### 2.5 Reduce memory consumption by caching data(RDD) in serialize form

You can mark an RDD to be persisted using the persist() or cache() methods on it. The first time it is computed in an action, it will be kept in memory on the nodes. Spark’s cache is fault-tolerant – if any partition of an RDD is lost, it will automatically be recomputed using the transformations that originally created it.

The cache() method is a shorthand for using the default storage level, which is StorageLevel.MEMORY_ONLY (store deserialized objects in memory).

With persist() method, you have the following storage level to choose

- MEMORY_ONLY : It stores RDD as **deserialized Java objects** in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
- MEMORY_AND_DISK : It stores RDD as **deserialized Java objects** in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
- MEMORY_ONLY_SER : (Java and Scala) It stores RDD as **serialized Java objects** (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
- MEMORY_AND_DISK_SER : (Java and Scala) Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
- DISK_ONLY : It stores the RDD partitions only on disk.
- MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc : Same as the levels above, but replicate each partition on two cluster nodes.
- OFF_HEAP (experimental) : Similar to MEMORY_ONLY_SER, but store the data in off-heap memory. This requires off-heap memory to be enabled.

#### The recommendation

If your data (RDD,df) requires heavy CPU consumption, or will be reused by many other jobs, its better to persist it. And if its too large to persist in deserialize form, you can persist it in serialize form. W**e**

**highly recommend using Kryo if you want to cache data in serialized form**, as it leads to much smaller sizes than Java serialization.

The only downside of storing data in serialized form is slower access times, due to having to deserialize each object on the fly

# 3. Spark partitions

## 3.1 Input data partitions

Use spark default partition size(i.e. 128MB), unless:

- You have much more cores than your partition, and you want to increase the parallelism
- Your data is heavily nested/repetitive
- You generate data by using **explode**
- Your data source structure is not optimal(upstream)
- You use UDFs in your spark application

```
// We can directly modify the config of spark session by using the following code
// The default size is 128MB
spark.conf.set("spark.sql.files.maxPartitionBytes",1024*1024*128)

// We can set it to 16MB, now the partition size is decreased, and the partition number is increased. We have more tasks, so the parallelism is increased.
spark.conf.set("spark.sql.files.maxPartitionBytes",1024*1024*16)
```

## 3.2 Shuffle data partitions

You have no longer control on the shuffle data partitions size. You can only control the number of partitions.The default number of shuffle partitions is 200. You need to change it.

There are some rules to determine the shuffle partition numbers:

1. Master equation to determine the shuffle partition count base line: **Partition_Count = Stage_Input_Data_Size/Shuffle_partition_size**
2. The Shuffle_partition_size should always less than 200MB(The default max size is 2GB). Because larger partition size will create shuffle spill in memory and disk.
3. The Shuffle_partition_number should always = total_core_number*N where N is 1,2,..,n. This can avoid extra useless shuffle task cycle

### 3.2.1 Shuffle partition count calculation example

For example, we have Stage_Input_Data_Size = 210GB, and we set Shuffle_partition_size =200 MB

As a result, Partition_Count=210000MB/100MB = 2100.

**case 1, our cluster has more core than the partition**

If our spark cluster has more than 2100 cores, for example 4000 core, if we only have 2100 partitions, it means we have 1900 cores that will not be used. To use all the cores in our cluster, we should set shuffle partition count to 4000. Note in some case, you may not want to use all the cores in your cluster. But we don't discuss these cases in this article.

```
// You can change the shuffle partition number with the following command
spark.conf.set("spark.sql.shuffle.partitions",4000)
```

**case 2, our cluster has less core than the partition**

If our spark cluster has less than 2100 cores, we should use rule number 3. Suppose we have 256 cores in our cluster. 2100/256=8.2. So we need 8.2 cycle to finish all the shuffle task. But the last cycle uses only 20 percent of the cores which is a huge waster. So we want to finish all the tasks in 8 cycles. 256*8= 2048. As a result the optimal shuffle partition number is 2048, and the partition size is 103MB.

```
// You can change the shuffle partition number with the following command
spark.conf.set("spark.sql.shuffle.partitions",2048)
```

**Important note**

**In your spark ui, you can check the data shuffled between each stage. If you see shuffle spill (memory) and shuffle spill (disk) happens a lot(e.g. more than half of the tasks) and it takes a lot of space(e.g. 1 or 2GB). This means your partition count number is wrong. You need to change it.**

## 3.3 Output data partitions

We can control the number and the size.

```
// To shrink the partition number
Coalesce(n)

// For example, myDf has 100 partitions, we want to shrink it to 10
val shrinkedDf=myDf.Coalesce(10)


// Increase partition number to n
Repartition(n)

// note Repartition can only increase the number of partition, for example
if myDf has 10 partitions
// The following command will not change the number of partition. newDf
```

```
still has 10 partitions
val newDf= myDf.repartition(1)

// We can also control how many lines in one partition
df.write.option("maxRecordsPerFile",N)
```

## 4. JVM Garbage Collection Tuning

All JVM based languages (e.g. scala/spark) need GC to clear unused object and free memory. **And the cost of garbage collection is proportional to the number of Java objects inside JVM heap memory. Because it needs to scan all objects in JVM to check if they are still in use or not. So if your GC take too much time to complete, try to reduce the number of Java objects.**

For example, If we have a RDD or DF, each record contains a LinkList of Integer of 100 elements. If we have 100 records, the GC will need to work with 100*100 object. If we persist it in serialize form, this RDD will become one object(a byte array) for each RDD partition.

Before trying other techniques, the first thing to try if GC is a problem is to use serialized caching.

### 4.1 Measuring the Impact of GC for all your spark jobs

The first step in GC tuning is to collect statistics on how frequently garbage collection occurs and the amount of time spent GC. This can be done by adding -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps to the Java options. Depends on how you run your spark jobs, you may want to set this config for both executor and driver.

```
#Deploy client mode, the driver runs on your local machine, the executors
runs on spark cluster

val spark = SparkSession
            .builder
            .master("My spark cluster resource manager url")
            .appName("My spark application name")
            .config("spark.executor.extraJavaOptions", "-verbose:gc -
XX:+PrintGCDetails -XX:+PrintGCTimeStamps")
            .config("spark.driver.extraJavaOptions", "-verbose:gc -
XX:+PrintGCDetails -XX:+PrintGCTimeStamps")
            .getOrCreate
```

```
# Deploy server mode, the driver and executors are all running on spark
cluster
./bin/spark-submit \
  --name "My app" \
  --master local[4] \
  --conf spark.eventLog.enabled=false \
  --conf "spark.driver.extraJavaOptions=-verbose:gc -XX:+PrintGCDetails -
XX:+PrintGCTimeStamps" \
  --conf "spark.executor.extraJavaOptions=-verbose:gc -XX:+PrintGCDetails -
```

```
XX:+PrintGCTimeStamps" \
  myApp.jar
```

When you run your Spark job with above config, you will see messages printed in the logs (each time a garbage collection occurs. Note executors' logs will be on your cluster’s worker nodes (in the stdout files in their work directories). The driver's log can be found in your local machine or workers who runs the driver, depends on your deploy mode)

### 4.2 Basic JVM GC tunning tips

The goal of GC tuning in Spark is to ensure that **only long-lived RDDs are stored in the Old generation and that the Young generation is sufficiently sized to store short-lived objects**. If you don't know what is old/young generation, please go read Understanding Java Garbage Collection before.

Check in GC stats, there are some signs which indicates your jvm memory config is wrong:

- If a full GC is invoked multiple times for before a task completes, it means that there isn’t enough memory available for executing tasks.
- If there are too many minor collections but not many major GCs, allocating more memory for Eden would help. You can set the size of the Eden to be an over-estimate of how much memory each task will need. If the size of Eden is determined to be E, then you can set the size of the Young generation using the option -Xmn=4/3*E. (The scaling up by 4/3 is to account for space used by survivor regions as well.). To be more accurate, you can estimate the size of Eden by using the size of the input data. For example, if you read data from HDFS, a block of 128MB will need 4*3*128MB. Because the size of a decompressed block is often 2 or 3 times the size of the block, and if we want to have 3 or 4 tasks’ worth of working space.
- if the OldGen is close to being full, reduce the amount of memory used for caching by lowering spark.memory.fraction; it is better to cache fewer objects than to slow down task execution. Alternatively, consider decreasing the size of the Young generation. This means lowering -Xmn if you’ve set it as above. If not, try changing the value of the JVM’s NewRatio parameter. Many JVMs default this to 2, meaning that the Old generation occupies 2/3 of the heap. It should be large enough such that this fraction exceeds spark.memory.fraction.

### 4.3 Deal with large memory worker node

If your worker node has more than 200GB memory, the default jvm GC may not be able to handle it efficiently. So use more advanced GC such as **G1 garbage collector** may improve performance in some situations where garbage collection is a bottleneck. You can find a detailed doc on how to tune G1 performance here(https://www.oracle.com/technical-resources/articles/java/g1gc.html).

### 4.4 GC tuning Summery

Here we don't give any specific numbers in the GC tuning, because it depends on your spark application and the amount of memory available of your spark cluster. So a specific configuration does not make sense. You need to use above principals to estimate the correct configuration and test it. Then monitor how the frequency and time taken by garbage collection changes with the new settings.

GC tuning flags for executors can be specified by setting spark.executor.defaultJavaOptions or spark.executor.extraJavaOptions in a job’s configuration.

There are many more tuning options(https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/index.html), if you want to know more about GC tunning, please go read it.

## 5. Increase Parallelism on reading input paths

If your spark job needs to read data from a large number of directories, you may also need to increase directory listing parallelism, otherwise the process could take a very long time, especially when against object store like S3 or Minio.

If your job works on **RDD with Hadoop input formats** (e.g., via SparkContext.sequenceFile), the parallelism is controlled via spark.hadoop.mapreduce.input.fileinputformat.list-status.num-threads (currently default is 1).

```
config("spark.hadoop.mapreduce.input.fileinputformat.list-status.num-
threads", 1)
```

For **Spark SQL with file-based data sources**, you can tune spark.sql.sources.parallelPartitionDiscovery.threshold and spark.sql.sources.parallelPartitionDiscovery.parallelism to improve listing parallelism. Please refer to Spark SQL performance tuning guide for more details.

```
# the default value for sql threshold is 32, for sql parallelism is 10000
val spark = SparkSession
            .builder
            .master("My spark cluster resource manager url")
            .appName("My spark application name")
.config("spark.sql.sources.parallelPartitionDiscovery.threshold", 32)
.config("spark.sql.sources.parallelPartitionDiscovery.parallelism", 10000)
            .getOrCreate
```

The spark source code which uses sql threshold config https://github.com/apache/spark/blob/v2.3.0/sql/core/src/main/scala/org/apache/spark/sql/execution/datasources/InMemoryFileIndex.scala#L171-L176

The spark source code which uses sql parallelism config https://github.com/apache/spark/blob/v2.3.0/sql/core/src/main/scala/org/apache/spark/sql/execution/datasources/InMemoryFileIndex.scala#L187-L189

The default setting code for sql threshold and parallelism : https://github.com/apache/spark/blob/v2.3.0/sql/catalyst/src/main/scala/org/apache/spark/sql/internal/SQLConf.scala#L584-L601

## 6. Data locality

Data locality can have a major impact on the performance of Spark jobs. If data and the code that operates on it are together then computation tends to be fast. But if code and data are separated, one must move to the other. Typically it is faster to ship serialized code from place to place than a chunk of data because code size is much smaller than data. Spark builds its scheduling around this general principle of data locality.

**Data locality is how close data is to the code processing it**. There are several levels of locality based on the data's current location. In order from closest to farthest:

- **PROCESS_LOCAL** : data is in the same JVM as the running code. This is the best locality possible
- **NODE_LOCAL** : data is on the same node. Examples might be in HDFS on the same node, or in another executor on the same node. This is a little slower than PROCESS_LOCAL because the data has to travel between processes
- **NO_PREF** : data is accessed equally quickly from anywhere and has no locality preference
- **RACK_LOCAL** : data is on the same rack of servers. Data is on a different server on the same rack so needs to be sent over the network, typically through a single switch
- **ANY** : data is elsewhere on the network and not in the same rack

Spark prefers to schedule all tasks at the best locality level. But its common to be in a situation that many data reside in a worker and all executors are busy, and another worker is IDLE because no data is available on the worker.

To avoid this kind of situation, Spark typically waits a bit in the hopes that a busy CPU frees up. Once that timeout expires, it starts moving the data from far away to the free CPU. The wait timeout for fallback between each level can be configured individually or all together with the following configuration parameters:

- **spark.locality.wait**: How long to wait to launch a data-local task before giving up and launching it on a less-local node. The same wait will be used to step through multiple locality levels (process-local, node-local, rack-local and then any).
- **spark.locality.wait.process**: Customize the locality wait for process locality.
- **spark.locality.wait.node**: Customize the locality wait for node locality.
- **spark.locality.wait.rack**: Customize the locality wait for rack locality.

The default value for all the wait time is 3 sec.

```
//Spark session config
.config("spark.locality.wait", 3)
.config("spark.locality.wait.node", 2)

//Spark submit config
./bin/spark-submit --name "My app" --master yarn-client --conf
spark.locality.wait=3
  --conf spark.locality.wait.node=2 myApp.jar
```

The default config usually works well. But you may encounter two possible scenarios:

- Scenario 1 (Large data sets): Jobs are taking too long to complete and on investigation, it was determined that tasks are being run on non-local nodes. Large amounts of data are being transferred to remote nodes (shuffling) resulting in delays in processing.
- Scenario 2 (Small data sets): Despite jobs processing small amounts of data, it is still taking too long since only 1 or 2 tasks are running on data-local nodes with the rest of the cluster mostly idle.

In scenario 1, shuffling data between workers costs more expensive than wait the worker to finish. So we should increase the spark.locality.wait time from 3 to 30s. This can avoid data shuffling.

In scenario 2, the data sets are small, so the cost of shuffling is negligible. We can reduce the spark.locality.wait time from 3 to 0. So spark will not wait and do shuffling directly.

As a general rule, long running jobs with big datasets would benefit from a higher wait time since the cost of waiting will have a less impact on the job's overall completion time. short time running Jobs with small datasets(e.g. 0.5-2 seconds) will be better off with a small wait time. In some circumstances, really short jobs should use a wait time of zero since they get executed immediately on the next available node.

Ultimately, determine the best wait time by achieving a balance between the cost of shuffling data around the cluster against the parallelisation of tasks around the cluster.