Parameter Testing in Spark

By
Panagiotis PETRIDIS
Department of Informatics

February 2016
Abstract

In this work attempts are made to identify the impact that certain parameters of Apache Spark’s 1.5.2 configuration have on the overall performance of applications. To our knowledge this is the first, or one of the first ventures on this subject. Since the number of the parameters is extensive, we identify those that theoretically have the greatest impact based on their documentation, while taking into consideration possible scenarios that would affect the performance. We conduct a series of experiments with known benchmarks on the MareNostrum petascale supercomputer in order to test the performance sensitivity and impact of the selected parameters and present our results and conclusions. Finally, we offer a methodology for tuning Apaches Spark parameters that when applied to an application may yield better performance.
# Contents

1 Introduction  
   1.1 Problem Setting .............................................. 2  
   1.2 Objectives and Contribution ................................. 3  
   1.3 Background .................................................. 3  
      1.3.1 MapReduce ............................................ 3  
      1.3.2 Apache Spark ........................................ 4  
   1.4 Thesis layout ................................................ 4  

2 Spark Parameter Description  
   2.1 Setting parameter values ..................................... 7  
   2.2 Spark Parameters Categories ................................. 8  

3 Related Work  
   3.1 Apache Spark optimization .................................. 11  
   3.2 Parameter and Workflow Optimization related work ...... 19  

4 Parameter Sensitivity and Spark Tuning Methodology  
   4.1 Choosing the Parameters ..................................... 23  
   4.2 Experimentation and Parameter Sensitivity ................ 28  
      4.2.1 Benchmarks ........................................... 29  
      4.2.2 MareNostrum and Spark4mn Framework ............... 29  
      4.2.3 Experiments .......................................... 30  
         Sort-by-Key experiments ................................. 31  
         TeraGenandShuffle experiments ......................... 33  
         K-Means experiments .................................. 36  
         Overall Statistics ..................................... 38  
   4.3 Tuning Methodology .......................................... 38  

5 Conclusions  
   5.1 Discussion .................................................. 43  
   5.2 Future Work ................................................ 43  

Appendices  

A Spark Parameters .............................................. 47
List of Figures

3.1 System Information, Awan et al., 2015 ...................................................... 14
3.2 Spark Configuration for each Benchmark, Awan et al., 2015 .............. 14
3.3 Speed-up for different number of maps (left), Speed-up for different compression options (right), Poggi et al., 2014 ..................... 18
3.4 Speed-up of using SSDs and InfiniBand (left), Speed-up of different Cloud deployment options for disks (right), Poggi et al., 2014 ................................................... 18
3.5 Cost vs Performance for Terasort (left) and WordCount (right) for different Software and Hardware Poggi et al., 2014 ......................... 19
3.6 The traditional workflow cycle and the new phase introduced, Holl et al., 2014 .......................................................... 20
4.1 Spark memory Distribution, Spark Architecture Article .................. 28
4.2 Performance impact of spark.serializer for Sort-by-key ................ 31
4.3 Performance impact for Sort-by-key ...................................................... 33
4.4 Performance impact of spark.serializer for TeraGenAndShuffle .......................... 34
4.5 Performance impact for TeraGenAndShuffle ......................................... 34
4.6 Performance impact of spark.serializer for K-Means (100.000.000 points) ......................................................... 36
4.7 Performance impact of spark.serializer for K-Means (200.000.000 points) ................................................................. 37
4.8 Performance impact for K-Means (100.000.000 points) .................... 37
4.9 Performance impact for K-Means (200.000.000 points) .................... 38
4.10 Spark Parameter tuning methodology ................................................. 41
List of Tables

2.1 spark-defaults.conf file example ............................. 7
4.1 Cluster Information ............................................. 31
4.2 Experiment Information ........................................ 31
4.3 TeraGenAndShuffle Information ............................... 33
4.4 K-Means experiment information ............................. 36
4.5 Average Parameter Impact ..................................... 39
4.6 Maximum Parameter Impact ................................. 39
Chapter 1

Introduction

It is common knowledge that the past decade is characterized by a great growth in data and information demand in the web, services and computer science as a whole. While advances in hardware technology are also great, they are not enough to counterbalance the demands and in some cases, they are unable to meet the demand. To accommodate such services, great effort is being put into developing new methods and optimizing existing ones in order to optimize and get the most out of the existing technology. Main goal of such methods is to use efficiently the already existing platforms. This is done either by micro-managing or by fine-tuning, thus finding the configuration that fits best on the application in question and yields the best results. Such results could be considered better either because they reduce the cost or because they increase performance.

Accommodating the demands mentioned above can be done in a variety of ways. One could say that we can always use bigger and more powerful machines when algorithms and optimization fails. Although this is the easiest solution to think of, it is the hardest to deploy. Such machines are usually costly and hard to come by. The next step and compromise to the above, is to use more than one machines with average to low capabilities and deploy them in a cluster. Of course adjustments need to be made in order for the application to run on more than one machines. This applies to both the algorithm that is being used and on the way the machines interact. One of the most common programming models that does all of the above is the Map-Reduce which is briefly explained later on.

Although the Map-Reduce programming model offers a way to deal with big data more efficiently across a cluster, optimization can still take place. As in most technologies, optimization here can lead to great performance, so effort is being put into exploring such new grounds. In this work we take one of the leading platforms in cluster computing, Apache Spark, and conduct research on ways that such a platform could be optimized and tuned, always having performance in mind.
Chapter 1. Introduction

1.1 Problem Setting

Apache Spark along with its predecessor, Apache Hadoop are frameworks for fast cluster computing that revolve around performance and speed. While such technologies are getting better and better (for instance Spark could be considered an improvement over Hadoop), there is always room for improvement. Spark makes this easy by providing over 100 parameters that can be tuned and changed according to the needs of the user, the application in question and the type of the computer cluster it runs on. However, this field remains, to our knowledge, still unexplored. Apart from a few tips and common strategies, mostly from Apache Spark’s configuration manual, there is little to be found.

Apache Spark’s parameter configuration is rich and its documentation proves a great asset in understanding the role of every single parameter. However, one cannot just start changing the configuration with hopes of improving performance, or rather, one cannot always achieve a better result by doing so. It would be easier if certain guidelines existed on what to change on Spark’s parameters and how would it affect the overall performance in most cases.

The guidelines described above are hard to come by. For starters, one would have to put a great amount of effort into identifying the impact and role of every single parameter. In addition, there is a cross-parameter issue to be considered. That is, when a parameter changes it may affect the whole configuration and result in another parameter behaving differently than anticipated. However, even if such an extensive research could be conducted and even if all parameter and cross-parameter impacts could be identified there would still be two major problems. The impact of said parameters may vary from application to application and it most certainly will vary from cluster to cluster. While the first issue may be dealt with by expanding the research to more than one type of applications by using certain benchmarks, the later is harder to deal with since none usually has the resources for a variety of clusters. So with this in mind compromises need to be made.
1.2 Objectives and Contribution

The centermost point of this research is to explore Apache Spark’s configuration parameters with hopes of understanding the impact of individual parameters to the performance of applications and possibly finding associations between individual parameters. Additionally, with this knowledge we can tune the configuration easier and faster since less test runs are required.

In addition, our main goal is to also provide insight on the effect and impact of most Apache Spark’s configuration parameters via experimentation. Of course some parameters are left out of our experimentation since it is either obvious that they cannot have any effect on the performance, or because they cannot be properly tested on the cluster we conduct the experiments due to hardware limitations.

While the above research is ripe and can be considered extensive to some degree, we take it one step further and provide a methodology for tuning Spark for running a user’s application. In other words, given an application and performing a few proposed test runs with different parameter configurations, the performance of said application can be improved. In this methodology all application are treated as black-box applications and effort is being put into grouping parameters and reducing the number of total test runs required to find a configuration on which the application runs faster.

1.3 Background

1.3.1 MapReduce

At a time before Map-Reduce, when analysts and computer scientists would face the problem on dealing with huge amounts of data (that is several GB at that time) they would have to resolve to writing additional code that splits the data in question and sends them to each computer node that is available. This was done in order for the whole procedure to be parallelized and be able to finish in an acceptable amount of time. Additional code would be required in order to check the state of the nodes, errors, possible data transfers and getting the result.

The Map-Reduce framework covers all of the above and shields the user from such code implementations. Generally all data are Mapped into key-value pairs. This mapping is important since values with the same key are grouped by the framework and end up on the same node of the cluster.
A simplified Map-Reduce workflow would require two scripts from the user. A Map script and a Reduce script. The Map step would split all the available data into chunks and distribute them over the machines of the cluster. The Map script, provided by the user, will emit key-value pairs that favor the application at hand. Next, each pair with the same key is shuffled, meaning that it will switch machines and go to a machine with other pairs with the same key as itself. Finally the Reduced script is used in order to provide the desired processing result.

Commonly, when we refer to the term Map-Reduce we do not necessarily refer to a specific framework, but instead we refer to the programming model mentioned above. There are several frameworks that utilize this model, with the most prevalent of them being Apache Spark. Apache Spark’s predecessor is Apache Hadoop and since Spark is based on it one can safely say that it is an improvement over it.

1.3.2 Apache Spark

Apache Spark is an open source cluster computing framework. It provides an interface enabling users to produce applications while managing entire clusters without having to worry about data parallelism issues. Spark achieves that by introducing and utilizing the Resilient Distributed Dataset (RDD). An RDD is, as the name implies, a read-only dataset with its elements distributed over the cluster that can be operated on in parallel.

While one can loosely say that Apache Spark is based on Map-Reduce, he would not be entirely correct. Spark does not follow the traditional Map-Reduce workflow. It offers the user, via its RDD functions, a distributed shared memory which is something that is outside of the traditional Map-Reduce.

As far as technology is concerned, Apache Spark could possibly be considered one of the best open source cluster computing frameworks with great potential since its community remains active and produces very often new versions and implementations.

1.4 Thesis layout

On the chapter that follows, an extensive explanation of Apache Spark’s configuration parameters is conducted. Information around setting Spark’s configuration and information regarding each individual parameter is presented. In Chapter 3, the related work on the field of Apache Spark optimization is presented along with existing work in parameter tuning in
general. In Chapter 4, the parameters that were included in the experiments are listed along with the experiments that were conducted in order to understand their impact. At the end of that chapter a methodology is proposed for tuning Spark in order to possibly improve performance. In the last chapter the conclusions of this work are presented.
Chapter 2

Spark Parameter Description

The Apache Spark framework, being open source, offers a great amount of tuning capabilities and customization. Its configuration has a total of 155 parameters that can be changed easily by the user and customized according to the application and cluster he uses. This number may seem overwhelming but a closer look to each individual parameter would make oneself think otherwise. While the sheer number of parameters is big, the parameters that affect performance-related issues are far more less. In this work we identified such parameters based on logic since some of them can be surely eliminated from the optimization process. The selection process is explained later on.

2.1 Setting parameter values

Spark interface provides a variety of ways of setting parameter values. Namely, the user can do it with three different ways.

Through the spark-defaults.conf file.

A user can change parameter values by modifying the conf/spark-defaults.conf file in the spark directory. An example file can be seen below.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spark.master</td>
<td>spark://5.6.7.8:7077</td>
</tr>
<tr>
<td>spark.executor.memory</td>
<td>4g</td>
</tr>
<tr>
<td>spark.eventLog.enabled</td>
<td>true</td>
</tr>
<tr>
<td>spark.serializer</td>
<td>org.apache.spark.serializer.KryoSerializer</td>
</tr>
</tbody>
</table>

At runtime as arguments in the spark-submit command.

A user can change the parameter values though the spark-submit command by adding the -conf argument followed by the parameter name and its desired value, for instance:
Chapter 2. Spark Parameter Description

```bash
./bin/spark-submit --name "My_app" --master local[4]
--conf spark.shuffle.spill=false myApp
```

Though the set configuration function from inside the application.

Parameter’s values can be changed from inside the application by utilizing the SparkConf’s set function. For instance:

```scala
val conf = new SparkConf().setAppName("TestRunner: "+testName).set("spark.serializer",
"org.apache.spark.serializer.KryoSerializer");
```

In our work, due to cluster restrictions we set the parameter values by using the third method.

2.2 Spark Parameters Categories

In this section all categories of Apache Spark’s Parameters are listed and a short description is provided. Note that each individual parameter is presented and explained in appendix A.

Properties Categories

In Spark’s configuration, properties are categorized and grouped according to their relation with the framework and their purpose. The groups that emerge from this categorization are the following:

- **Application Properties**: Parameters in this group have a general role regarding the application name, the memory that will be given in executors and drivers etc.

- **Runtime Environment**: Parameters that belong to this group involve environment settings like class paths, java options and logging.

- **Shuffle Behavior**: Here parameters have to do with the shuffling mechanism of Spark. They involve buffer settings, sizes, shuffling methods etc.

- **Spark UI**: Here we find UI related parameters that have to do mostly with UI event-logging and enabling option for Spark’s UI.

- **Compression and Serialization**: Parameters of this group target compression and serialization issues. They mostly have to do with whether compression will take place or not, what compression codec will be
used, the codec’s parameters, what serializer will be used and its buffer options.

- **Execution Behavior**: parameter of this group have a more general role and serve different purposes. While there is a detailed description in appendix A, here we will just say that the most important parameters of this group refer to memory fractions, number of execution cores and parallelism.

- **Networking**: This group contains parameters that provide options for network issues. In this work we do not experiment with this group due to the fact that any results would have to be unique to our network and would have no real value that could be applied to most networks. That is because most parameters refer to timeout options, heartbeat pauses, ports and network retries and optimal values are not only hard to be identified but also have not cross-cluster value.

- **Scheduling**: Most parameters in the group cover scheduling options. Some noteworthy parameters have to do with scheduling mode, and whether to use the speculation optimization and the maximum number of CPU cores that will be used.

- **Dynamic Allocation**: By using Dynamic Allocation Spark can increase or decrease the number of executors based on its workload. This is only available in Yarn which we do not use in this work. However, even if we did, such an optimization is optional and theoretically it would not have a huge impact on performance.

- **Security**: The purpose of parameters in this group is self explanatory. Again we did not experiment on this group for obvious reasons.

- **Encryption**: Same as Security. Properties here refer to encryption algorithms passwords and keys, as expected.

- **Spark Streaming and SparkR**: Here parameters involve Spark Streaming options and SparkR option respectively. Again these have nothing to do with the performance issues we aim to improve so they are left out of our experiments.

For more information for Spark’s parameter the reader should refer to appendix A.
Chapter 3

Related Work

As it was mentioned on the first chapter, there is a great need for optimization and improvement in order to accommodate big data needs. Such attempts can be categorized into two categories when related to our problem, in attempts to improve Apache Spark and provide insight on its behavior and in attempts that have to do with optimizing parameters. So, in this chapter we discuss the related work regarding Apache Spark and parameter optimization in general.

3.1 Apache Spark optimization

In this section related work regarding spark optimization is presented. At Tous et al., 2015 the Apache Spark Dataflow Framework is chosen in order to provide insight on the shift between compute-centric paradigm and the more recent approach, data-centric paradigm, that was developed in response to the growth of data-sensitive applications. Authors aim to the full integration of Spark to the Aloja Project and provide assistance in deploying Apache Spark and also in improving Spark applications.

They introduce the SPark4MN Framework and conduct their work on the Spanish Tier-0 supercomputer, MareNostrum. This is the same framework and supercomputer that is used in this work and there will be a detailed reference to it on the following chapters.

In order to conduct their experiments, authors use the spark-perf Spark performance testing framework and the experiments are run over generated synthetic data. They select two benchmarking applications, sort-by-key and k-means. These applications are selected by having in mind the different challenges for their parallelization. Sort-by-key needs heavy shuffling and great communication between the nodes while k-means requires great CPU resources and networking. Authors consider these applications as representatives of two types of real applications. The first type, which is represented by k-means, has to do with applications that are parallelized...
straight away and have little to none shuffling data and communication between the nodes. The second type, represented by sort-by-key, involves application that heavily re-partition the data in order to perform e.g. sorting.

The above experiments are conducted over clusters of several hundreds of nodes and the memory allocated is just enough so that the input RDD can fit. Also authors’ main focus is parallelism configurations so scenarios where RDDs cannot fit into the main memory are considered out of scope. The results of their research are presented here.

The first issue that is targeted is performance and scalability with aims in testing speed-up, scale-up and size-up properties.

Three sets of experiments were conducted for the k-means speed up test, with 10M vectors of 1000 dimensions, 100M vectors of 100 dimensions, 1B vectors of 10 dimensions and in each experiment K-means runs for 10 iterations. All datasets are run over 8, 16 and 32 nodes to test speeding-up. In short, the conclusions that are derived are the fact that processing costs for each set differ significantly. In the 1000 dimensions set, speed-up is smaller and it costs more to process large datasets with small records that small datasets with big records due to the Java objects being larger in size.

Regarding the K-means scaling-up tests the same procedure is followed with the difference that now both the number of records and machines is modified. The most linear behavior is exhibited by the 100 dimension dataset but there is no more than 15% offset by the rest.

Finally for the K-means size-up experiments the number of nodes is kept the same but the dataset size is being increased. Results show a near linear size-up with the sole exception of the 10 dimensional data set where the size-up is more than double when doubling up the dataset size.

The respective experiments for the sort-by-key, speed-up, scale-up and size-up yield the following results. For the same dataset when increasing from 1024 to 2048 nodes, there is a 1.38 x speed-up while when increasing from 2048 to 4096 there is a 10% slowdown. The results for size-up and scale-up are nearly linear.

Following the above experimentations the impact of configurations is examined. Specifically the degree of parallelism, executor size and processor affinity. For K-means the experiments yield results which indicate that the best practice is to have as many partitions as cores. Anything more than that most often may have up to 31% of performance degradation and quite rarely an increase in performance.

In the matter of affinity for k-means the results suggest that there is no
3.1. Apache Spark optimization

real benefit in forcing all cores of a worker to be on the same physical machine and that forcing the joint use of physical cores degrades performance. Also forcing the nodes to be as physically close as possible yielded no increase in performance and there was an extra cost in scheduling the job based on those restrictions.

For the impact of the degree of parallelism for sort-by-key applications the authors test the speed-ups for 1, 2 or 4 partitions per core. At 2 partitions per core speed-ups are observed but for 4 there is a significant degradation.

Further tests are conducted in order to determine the impact of the networking. Sort-by-key and k-means are run on Infiniband and Etherent networks with data either of the GPFS or generated on the main storage. Results show a 22.5% benefit for the Infiniband over the Ethernet for the 1000 dimension dataset for K-means (due to the fact that there is increased shuffling here). For sort-by-key there is a 27% difference for small numbers of partitions.

In Awan et al., 2015 the issue of data volume is examined. To be more specific they explore issues regarding Apache spark data on large scale-up servers, the impact of garbage collection, file I/O and data size. Towards these ends, they experiment over the effect of data volume on performance of Spark, try to find limitations on Spark with large volumes on data and try to determine variations on the performance of different applications.

In order to achieve the above goals they introduce a set of application benchmarks for Spark. These are:

- **Word Count**: application that counts the number of occurrences of every word in a text file.
- **Grep**: searches for a keyword (in this case the keyword is "The") in a text file and gives as output the lines with this keyword.
- **Sort**: sorts a number of records based on their key.
- **Naive Bayes**: uses semi-structured Amazon Movie Reviews data-sets for sentiment classification. Only the classification part of the benchmark is used.
- **K-means**: Usual k-means algorithm, clusters data points into a pre-defined number of clusters.

In figure 3.1 we can see the details about the machine used for the experiments. Also in 3.2 we see the Spark Configuration used for each Benchmark application. In our work we used some of these parameter choices
Chapter 3. Related Work

Figure 3.1: System Information, Awan et al., 2015

<table>
<thead>
<tr>
<th>Component</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel Xeon E5-2697 V2, Ivy Bridge micro-architecture</td>
</tr>
<tr>
<td>Cores</td>
<td>12 @ 2.7 GHz (Turbo upto 3.5 GHz)</td>
</tr>
<tr>
<td>Threads</td>
<td>2 per core</td>
</tr>
<tr>
<td>Sockets</td>
<td>2</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>32 KB for instructions and 32 KB for data per core</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>256 KB per core</td>
</tr>
<tr>
<td>L3 Cache (LLC)</td>
<td>30 MB per socket</td>
</tr>
<tr>
<td>Memory</td>
<td>2 x 32 GB, 4 DDR3 channels, Max BW 60 GB/s</td>
</tr>
<tr>
<td>OS</td>
<td>Linux kernel version 2.6.32</td>
</tr>
<tr>
<td>JVM</td>
<td>Oracle Hotspot JDK version 7u71</td>
</tr>
<tr>
<td>Spark</td>
<td>Version 1.3.0</td>
</tr>
</tbody>
</table>

Figure 3.2: Spark Configuration for each Benchmark, Awan et al., 2015

<table>
<thead>
<tr>
<th>JVM</th>
<th>Heap Size (GB)</th>
<th>Wc</th>
<th>Gp</th>
<th>So</th>
<th>Km</th>
<th>Nb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old Generation Garbage Collector</td>
<td>PS MarkSweep</td>
<td>50</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Young Generation Garbage Collector</td>
<td>PS Scavange</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spark</td>
<td>spark.storage.memoryFraction</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.6</td>
<td>0.1</td>
</tr>
<tr>
<td></td>
<td>spark.shuffle.memoryFraction</td>
<td>0.7</td>
<td>0.7</td>
<td>0.7</td>
<td>0.4</td>
<td>0.7</td>
</tr>
<tr>
<td></td>
<td>spark.shuffle.consolidateFiles</td>
<td>true</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>spark.shuffle.compress</td>
<td>true</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>spark.shuffle.spill</td>
<td>true</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>spark.shuffle.spill.compress</td>
<td>true</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>spark.rdd.compress</td>
<td>true</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>spark.broadcast.compress</td>
<td>true</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

when we conducted our experiments. Most of the parameters used here are some of the one that we select for experimentation, something that is promising.

When conducting the experiments, authors conduct the same experiments for each benchmark application and the experiments can be split into two categories, scalability and Scale-up limitations.

Regarding the issue of scalability authors run experiments for 1, 6, 12, 18 and 24 executor pool threads for each benchmark. Results show that there is no benefit in using more than 12 executor threads since there is a sub-linear speed up in the results.

In addition to the above experiment set Spark’s performance consistency is tested for various data sizes. Namely 6, 12, 24 GB of input are used. K-Means has up to 92.94% performance degradation from 6 to 24GB, while Grep has only 11.66%. Furthermore, when the data size is increased from 6 to 12GB there is an overall performance drop of 49.12% and an additional
3.1. **Apache Spark optimization**

8.51% when it is increased to 24GB.

On the matter of implications of Scaling up, the Garbage collection time is examined first. The GC times of the previous experiments is measured and the results show that GC time is increased as the cores increase across all benchmarks and it acts as a bottleneck. For K-means, Word Count and Naive Bayes there can be an increase of 48% for 24 cores. On the issue of data processing capabilities, for the same benchmarks there is a 39.8x increase in GC time when the input data is quadrupled. As a result there is a 14x performance decrease for K-means and a 3x decrease for Naive Bayes. This shows that because GC time does not scale linearly with the data size, performance for Spark applications degrades.

Authors also try to explain whether file I/O becomes a bottleneck under large volumes of data. In order to achieve that they analyze CPU time for each thread and identify I/O wait time and idle time as the top waiting functions. With the exception of the grep applications, it is shown that if the input data increases, so does the wait time. CPU time decreases by 54.15%, 74.98% and 82.45% in Word Count, Naive Bayes and Sort respectively but for Grep it increases by 21.73% when comparing 6 to 24GB input. In other words I/O wait time increases by 5.8x, 17.5x and 25.4x for Word Count, Naive Bayes and Sort, while is increases by 1.2x for Grep.

Finally the issue of micro-architecture performance regarding the input data size is addressed. While using 24 threads all benchmarks are run for 6, 12 and 24GB of data. On average across the workloads, retiring category accounts for 28.9% of pipeline slots in 6 GB case and it increases to 31.64% in the 24 GB case. Back-end bound fraction decreases from 54.2% to 50.4% on average across the workloads. K-Means sees the highest increase of 10% in retiring fraction in 24 GB case in comparison to 6 GB case.

Continuing related work regarding the Apache Spark’s optimization, on Ousterhout et al., 2015 there is an extensive work in order to identify bottlenecks in a large scale data analytics framework, such as Spark, and find out what the impact of each bottleneck is on the overall performance. Most types of bottlenecks are taken into account, network, disk and tasks taking too long to complete are some of them.

Two contributions are made. They provide a methodology, blocked time analysis, for identifying bottlenecks on a data analytics framework. This provides users with the ability to predict how long will a certain job have taken in order to complete if there was no bottleneck at all, thus finding if any task is worth optimizing by setting an upper bound. The second contribution has to do with applying blocked time analysis on two industry
benchmarks that are run on Apache Spark.

The benchmarks that are run are the big data benchmarks (BDBench) and a variation of the Transaction Processing Performance Council’s decision-support benchmark (TPC-DS). All make use of Apache SQL. The first benchmark consists of four queries. These are two exploratory SQL queries, one join query and one page-rank-like query. The second benchmark supports multiple users that run in parallel a plethora queries that can be decision-support, reporting, data mining queries.

The main contribution, which is blocked-time analysis takes into consideration a tasks inactivity. Such inactivity is usually due to being blocked (hence blocked-time analysis) by waiting for resources such as disk I/Os, CPU time and the network. In their work authors also measure resource utilization in order to cross validate their measurements. All this was achieved by implementing Apache Spark’s logging interface.

The findings regarding the Network bottlenecks yield that there can be no noteworthy benefit from network optimization. No noteworthy is on an average of 2%. That is, according to the authors mainly because the data transferred over the network are far more less than the actual input. So, even in optimal conditions there would be no actual benefit in optimizing the network.

Regarding the Disk I/O bottlenecks, there can be at most a 19% improvement based on blocked time analysis. That means that if there was no time waiting for I/O all processes would finish just short of 19% faster. Finally stragglers processes only reduce performance for up to 10%.

Related to optimization contributions is the ALOJA Project, introduced at Poggi et al., 2014. While it does not revolve around Spark but Apache Hadoop, the motivation is the same. This motivation was the fact that as Hadoop became more and more popular among companies, more applications where being deployed and need to tuning and optimizing was needed. Works that increased Hadoop performance trifold compared to the default configuration assured the above. This, combined with the rarity of configuration changed to Hadoop’s distribution, along with it performing poorly on scale-up hardware gave enough motivation for the ALOJA project to take place.

The ALOJA project is an initiative of the Barcelona Supercomputing Center. Its main goal is to provide a public Hadoop benchmarking repository in which not only software configuration parameters are going to be compared but also hardware. The project has three phases. The initial
one, Phase 1, where authors perform runs over a range of hardware components, parameter configurations, deployment patterns and study the results. The hardware components are tested with regard to storage, network connections and On-premise vs Cloud servers while parameters with regard to number of mappers/reducers/, data compression and IO buffer/block sizes and data replication factors. The deployment patterns have to do with on-premise scale-up physical servers and commodity hardware and a varied number of Virtual Machines.

The second phase of the project introduces models that runs over the performance data gathered from phase one. These models can provide predictions on the expected performance based on the above characteristics. It also expands phase one with new configurations and parameters that are put into play. All of the above are included on an online application for ease of access and comparison with visualization tools. The third and final planned phase revolves around automatic calculation of the most cost effective setup, ranging from the hardware down to deployment patterns that must be used, for a specific set of workload characteristics. In other words, it answers questions like "I want to do this work, what do I need?".

The benchmarks that were selected for the experiments had to cover a variety of applications due to the large number of configuration options. The following are the ones that are used:

- Terasort
- WordCount
- Sort
- Pagerank
- Bayes
- K-Means
- DFSIOE

The findings of the many runs that were conducted with the above benchmarks are presented into three categories. The first results, regarding the impact of Software Configurations can be seen on 3.3. It is obvious that the best performing setting differs for each benchmark. Regarding the maximum number of mappers option for pagerank, sort and wordcount the best performance is achieved with 8 mappers, but for terasort there is a 1.7x speed-up for 6 mappers. It should be noted that the default configuration for the mappers is 2. On compression speed-up, we see that no
Chapter 3. Related Work

**Figure 3.3**: Speed-up for different number of maps (left), Speed-up for different compression options (right), Poggi et al., 2014

Compression slows down the sort benchmark and the optimal solution for the rest is different.

**Figure 3.4**: Speed-up of using SSDs and InfiniBand (left), Speed-up of different Cloud deployment options for disks (right), Poggi et al., 2014

The second category results exhibit the impact of hardware Configurations and can be seen briefly on 3.4 for some of the benchmarks. As the figures dictate there is usually minimal speed-up for some benchmarks, which can never be bad, but for some there is even a 2x speed-up.

The third category of results address a Performance vs Cost analysis. In figure 3.5 the results for WordCount and Terasort can be seen. The closest a point is to the (0,0) point the more fast and economic its setup and configuration is for the benchmark in question.

Finally the work of Davidson and Or, 2013 focuses on Spark’s shuffling
optimization with two approaches. One is addressing the issue of columnar compression which, while providing some information regarding the compression codecs, does not yield any significant optimization. The second approach is addressing the file consolidation issue during shuffling. The results show certain speedups when consolidation is used and provide insight regarding the scaling of this approach. Lastly file consolidation on Spark is compared with Hadoop’s performance along with several tests that prove that the filesystem’s type affect the performance of file consolidation.

3.2 Parameter and Workflow Optimization related work

In this section related work regarding parameter optimization is presented briefly.

In Chung, Hollingsworth, et al., 2004 authors present Active Harmony. Because of the importance of parameter tuning and the performance improvements that it can offer an automated runtime performance tuning system was developed. Given a range of parameters, Active Harmony can tune and improve performance during a single execution. Due to the fact that during parameter tuning the search space can become extremely large and time consuming, parameters that are considered most important are prioritized. Splitting parameters based on the effect they have on performance is a common strategy, something that we also use in our work. Each parameter can be considered as an extra dimension on the search space that needs to be tuned, so eliminating the ones that have or may not have any significant impact is important. To identify such parameters authors test the sensitivity of the application by changing each parameters values. If
the application is sensitive to such changes then tuning these parameters may change the performance overall. However, it needs to be noted that this approach is effective only when there is minimal to no interaction between parameters.

The version of Active Harmony presented follows the basic principal of the tool described but it applies it with a bit more refined way. The previous versions of Active Harmony tested each parameters for sensitivity just by selecting their extreme values on their experiments. It is stated that for obvious reasons this is not always a great approach. The refinement that is suggested and that is adapted in our work is to select values that are near the default parameter value.

Among the above authors present the Data Analyzer used in their work, along with some synthetic Data experiments and the performance estimation algorithm that they use with Historical data or no. These are followed by an analysis regarding the parameter sensitivity, tuning with experience and search refinement but these move out of the scope of our work.

At Holl et al., 2014 the issue of parameter optimization on workflows is addressed. The main motivation behind this work is the fact that while for small workflows a trial and error optimization method can be applied, when more complex workflows come into play the interrelationship impact between the parameters cannot be easily identified. The traditional workflow cycle can be seen in 3.6 along the the new automatic optimization phase that is introduced.

FIGURE 3.6: The traditional workflow cycle and the new phase introduced, Holl et al., 2014

![Diagram of workflow cycle and new phase](image-url)
Generally a variety of different aspects of workflow optimization are defined. Parameter optimization, extended parameter optimization, composition optimization, component optimization and provenance based optimization. While all of the above can contribute greatly in optimizing an application the part that interests us regarding this work is the parameter optimization part. On this issue authors use Genetic algorithms, in order to refrain from extended parameter sweep methods and reduce the search space of the parameters. In short Genetic algorithms use a combination of exploration and hill climbing methods that provides them with the ability to find a global optimum without getting stuck. In their work authors make use of the Java Genetic Algorithms Package and go into detail about the plugin and setup of the optimization package, something that is again, out of our scope of work.

At Chirigati et al., 2012 an attempt is conducted in order to evaluate Parameter Sweep Workflows (PS) on High Performance Computing (HPC). The intense data processing is identified as an important characteristic of such workflows since the space of parameter values can sometimes be extremely large. This is the main reason why a HPC environment is needed in order to easily produce results. Authors developed a study to show the impact of different execution models for parameter sweep workflows. They observe five PS workflow patterns and identify four basic activities of PS workflows (Map, Split Map, Reduce and Join) in order to describe them. At the end of their work they conduct an experimental evaluation for four distinct PS workflow models on top of the Chiron workflow engine.

Finally the issue of parameter optimization is also addressed at Kumar et al., 2009. Motivated by two applications, Pixel Intensity Quantification and Neuroblastoma Classification, the parameters that impact the performance of spatial data analysis applications are presented and classified. This classification includes two categories, Quality-Preserving parameters and Quality-Trading parameters. Authors also introduce an integrated framework for performance optimization in multiple dimensions of the parameter space and conduct and experimental evaluation on it. The results of tuning parameters of both categories yield improvements for both of the applications that were tested. It is worth mentioning that improvements were up to a 50% in runtime for the Pixel Intesity Quantification application.
Chapter 4

Parameter Sensitivity and Spark Tuning Methodology

In this section a set of parameters that are theoretically believed to have an impact on Spark’s performance are selected and an attempt is conducted to test the framework’s sensitivity on them via experimentation. The selection of this set is mostly done based on logic since the majority of Spark’s parameters can be safely left out of the process based on their purpose and documentation. For the rest, we identify scenarios in which they can possibly have an impact on performance. Along with it, a basic methodology of Spark’s parameter tuning is described and experiments are conducted to test it. The main goal is the following: given a specific cluster and Apache Spark setup to which one cannot make changes along with an application that is treated as a black box without knowing its true purpose or its implementation, to be able to make a small number of test runs with different configurations that will eventually provide a better performance. Those test runs should be given in the form of a methodology and are the same for each and every application. This methodology is explained in the sections that follow, along with the theory behind selecting the parameters and the experiments conducted.

4.1 Choosing the Parameters

If one was to make a survey on the issue of parameter tuning he would find that one of the biggest challenges on this field is search space. If the number of parameters is large, then the number of different combinations and values increases nearly exponentially. So finding the best set of parameters is a lot harder just because there are so many places to look for it. However in our case, what we are trying to achieve offers a slight but
adequate advantage over other most automated parameter tuning methodologies. In Chung, Hollingsworth, et al., 2004, where an automated parameter tuning system based on prior runs is introduced, the tuning problem is general. The user provides the set of parameters and the system tries to find what contributes to the system performance in an as smart way as it can. Testing different values for each parameter comes into play in order to test sensitivity and then the optimal setup is selected based on performance, but each and every parameter has to be tested at some point since the automated parameter tuning system does not know anything about the application it is trying to tune. However, in the case of Apache Spark we know the set of parameters before we even begin tuning and we are able to decide whether a certain parameter might benefit the performance at all. This is mainly done by referring to the Apache Spark Configuration. One can easily conclude from there that the main bulk of parameters have nothing to the with the performance. Those for example are parameters that set a path or a custom class. Moreover, for some of the parameters we can identify certain scenarios in which they could have a significant impact on the performance. This way we do not have to deal with all 155 configuration parameters and we can reduce the search space down to a few of them.

In this section the main parameters that were selected from the previously described process are presented along with an explanation of why those should have an impact on Spark’s performance. These parameters are the ones that should be first taken into consideration if one wishes to improve the framework’s performance.

- `spark.reducer.maxSizeInFlight`: Each reducer maintains a buffer of size dictated by this parameter for fetching map outputs. If the number of reducers is high enough the amount of memory that is put into these buffers can become several gigabytes. Something like this is not that uncommon since usually there can be several hundreds of reducers in an application. However, if this value is increased, reducers would request bigger output chunks, something that would increase the overall performance but it would be at the expense of the memory that is used by the reducers, since more of it will be used as buffers. On clusters that there is adequate memory, increasing, let us say doubling, this value could yield better results with not apparent cost. On the other hand, if a cluster does not have adequate memory available, freeing some of the 48mb in order to provide more memory for the task should yield a better result.
4.1. Choosing the Parameters

- `spark.shuffle.compress`: This parameter provides the option to compress the map output files or not and reduces the size of transferred data. Generally compressing data before they are transferred over the network is a good idea. It reduces the data that will be transferred on a relatively slow speed so there will be a speed-up on the overall performance. However, this applies only to the scenario where the time it takes to compress the data and transfer them is less than the time it takes to transfer them uncompressed. That is, if the processing capabilities of the system are faster than the network available. On the other hand, on the scenario that the network transfer times are faster than the CPU processing times, the main bottleneck of the application is shifted. Now the process is not stalled by the amount of data that are transferred over the network but from the time it takes for the system to compress them. Both scenarios are equally viable since they depend on the cluster setup that is available. It is worth mentioning that although it is a good idea to compress the data in this case, on the terasorting benchmarking performed by Apache Spark the data compressions were removed and yielded better results.

- `spark.shuffle.file.buffer`: The size of memory buffers for the intermediate shuffle files on the output stream. This parameter could be considered similar for the `spark.shuffle.maxSizeInFlight` parameter and the reasons it is worth considering are the same. Again if a cluster has adequate memory, then this value could be increased in order to get the maximum performance. If not, there would be performance issues since too much memory would be put into buffers alone. It is worth noting that in previous versions of Spark, this parameter’s default value was 100kb instead of 32kb that it is now and that is was reduced for the same reason. [optimizing shuffle performance in Spark](https://issues.apache.org/jira/browse/SPARK-2503)

- `spark.shuffle.manager`: The implementation that Spark is going to use for data shuffling. The available implementations are three: sort, hash, and tungsten-sort. On the earlier versions of Apache Spark the default shuffle manager was the Hash manager until it was changed at 1.2 with the Sort-based manager. The main issue was the fact that it created too many open files for certain inputs as seen in [Optimizing Shuffle Performance in Spark](https://github.com/apache/spark/pull/1781). However as an improvement upon it there is an implementation regarding consolidating files that can be selected by the `spark.shuffle.consolidateFiles` parameter that when used
with the *Hash* it should have an impact on performance. On Apache Spark 1.5.2 *tungsten-sort* is introduced. It is stated in the *Tungsten-sort description* that for this manager to work perfectly and yield performance improvements the following must be satisfied:

- The shuffle dependency specifies no aggregation. Applying aggregation means the need to store deserialized value to be able to aggregate new incoming values to it. This way you lose the main advantage of this shuffle with its operations on serialized data
- The shuffle serializer supports relocation of serialized values (this is currently supported by KryoSerializer and Spark SQL’s custom serializer)
- The shuffle produces less than 16777216 output partitions
- No individual record is larger than 128 MB in serialized form

Since there are so many factors that can contribute to the performance of the shuffling managers this parameter might have an impact on the performance.

- `spark.io.compression.codec` : The codec that is going to be used for compression purposes, such as compressing RDDs, shuffling data, and broadcast variables. Three options are available, snappy, lz4, lzf. Generally, it is easy to test and see what is the faster and most cost-efficient on a certain dataset. There are many tests conducted by various authors (one of which being Davidson and Or, 2013 that test the performance of these codecs against each other. However, we could not safely say that this would always be the case. There is always the possibility for a certain type and size of a dataset that one of those codecs would perform different than expected. This is the main reason that this parameter is considered to have an impact on performance. Moreover, as it was stated previously, *tungsten-sort* configuration states that using lzf is the best way of improving performance with this manager.

- `spark.shuffle.io.preferDirectBufs` : Only used in Netty. Whether to use off-heap buffers in order to reduce garbage collection in shuffling process and caching. It is stated in Spark’s documentation that this option is for environments where off-heap memory is not tightly limited. If the off-heap memory is indeed limited this is option is suggested to
4.1. Choosing the Parameters

be turned off. Again in our methodology we do not know what of the above is the case, so this parameter is included as well.

- **spark.rdd.compress**: Whether to compress serialized RDD partitions. Again here we have the same trade-off issue that we encountered on **spark.shuffle.compress**. In this case though the trade-off lies between CPU time and memory and the two scenarios that benefit from this parameter are a fast CPU but small memory and the opposite.

- **spark.serializer**: What Serializer is going to be used for any serialization purpose. It is stated in Spark’s configuration documentation that KryoSerializer is thought to perform much better than the default Java Serializer when speed is the main goal. However, the Java serializer is still the default choice so this parameter is worth being considered.

- **spark.shuffle.memoryFraction**: This value indicates the fraction of the Java Heap that will be used for aggregation and cogroups during a shuffle. When this value is surpassed then the data will be spilled into the disk. If spills are often then this value (0.2) should be increased to avoid unnecessary spills. Since this parameter is directly linked to the amount of memory that is going to be utilized, a high performance impact is expected. Since one cannot just request more memory from the system, when increasing or decreasing this value it should be done on the expense or advantage of the **spark.storage.memoryFraction** parameter, which will be explained next.

- **spark.storage.memoryFraction**: This parameter indicates the fraction of JVM memory that is going to be used for caching purposes. The default value is 0.6 and 20% of this is allocated for unrolling blocks in memory. Again since this parameter is directly linked to the amount of memory that is going to be utilized, a high sensitivity is expected. In figure 4.1 we can see the JVM Heap model that is used in Spark. To avoid out of memory errors spark allows only 90% of the JVM Heap to be utilized. This fraction is shown as "Safe" (from safe use) on the figure. Fractions of this safe part are distributed based on the **spark.storage/shuffle.memoryFraction** parameters.

- **spark.shuffle.consolidateFiles**: This parameter provides the option of consolidating intermediate files created during a shuffle. This way fewer files are created and performance is increased. It is stated however in Spaks configuration that it is recommended to be used on ext4 and xfs filesystems but for filesystems that use ext3 with more than 8
CPU cores there might be a performance degradation. Also as it was stated earlier using this option along with the hash manager was the default option before sort manager was introduced and is expected to have a modest impact on performance. The impact that shuffle consolidation has on the performance is greatly explored in Davidson and Or (2013).

- `spark.shuffle.spill.compress`: Whether to compress the data that are spilled into the disk. Again this is the same issue with the previous compression-related parameters regarding where the application bottleneck lies. In this case if transferring the uncompressed data on an I/O operation is faster than compression and transferring compressed then this option should be set to false. Provided that there is a high amount of spills, this parameter may have an impact on performance.

### 4.2 Experimentation and Parameter Sensitivity

In this section the experiments regarding parameter sensitivity are presented. The parameters that were included are the ones described in the previous section and the main goal is to determine the impact that each of them has on the performance of certain applications.
4.2.1 Benchmarks

For continuation purposes with previous related work [ref] the following 3 benchmark applications were selected for testing.

- K-means
- Sort-by-key
- Shuffling test

The first two, *K-means* and *Sort-by-key* are part of the HiBench and where selected mostly because each of them could be considered as representative of a variety of applications and are aimed to test Spark’s performance in different ways. *K-means* is a well known, unsupervised learning algorithm that can be considered heavily reliant on CPU resources and network communication. On the other hand *Sort-by-key* usually requires a great amount of shuffling. Since each of them relies on different resources the impact of parameters that relates to those resources can easily be identified.

The third application that is used for testing is [Rxin’s] *TeraGenAndShuffle*. *TeraGenAndShuffle* generates data according to the terasort spec on the fly and then conducts a shuffling test by shuffling them. This application was selected mostly due to the fact that it presented the possibility of testing the shuffling mechanism even further over seemingly big data inputs.

It should be noted here that for our runs we used the same code that was used in Tous et al., 2015. This was done mainly because there where implementations regarding data generation prior to runs which are very useful in our case. Also both K-means and Sort-by-key are implemented already on Spark’s framework. Although the data over which these benchmarks run is generated at the beginning of each test, the generation process is not taken into account on the overall performance.

4.2.2 MareNostrum and Spark4mn Framework

MareNostrum is the Spanish Tier-0 supercomputer provided by BSC. It is an IBM System X iDataplex based on Intel Sandy Bridge EP processors at 2.6 GHz (two 8-core Intel Xeon processors E5-2670 per machine), 2 GB/core (32 GB/node) and around 500 GB of local disk (IBM 500 GB 7.2K 6Gbps NL SATA 3.5). Currently the supercomputer consists of 48896 Intel Sandy Bridge cores in 3056 JS21 nodes, and 84 Xeon Phi 5110P in 42 nodes (not used in this work), with more than 104.6 TB of main memory and 2 PB of GPFS (General Parallel File System) disk storage. More specifically, GPFS provides 1.9 PB for user data storage, 33.5 TB for metadata storage (inodes
and internal filesystem data) and total aggregated performance of 15GB/s. The GPFS filesystems are configured and optimized to be mounted on 3000 nodes. All compute nodes are interconnected through an Infiniband FDR10 network, with a non-blocking fat tree network topology. In addition to the 40 Gb/s Infiniband, 1 Gb/s full duplex Ethernet is in place. With the last upgrade, MareNostrum has a peak performance of 1.1 Petaflops.

On this supercomputer we conducted the experiments on the following section. In order to do that we used the Spark4mn framework that was used in Tous et al., 2015 on the same supercomputer.

### 4.2.3 Experiments

The sensitivity to each parameter is tested similarly with [Active Harmony]. For each of them a separate run (or runs, depending on the cardinality of values it has) is conducted with a different value. Then the performance is compared with the performance of the default value. If there is a big offset among the results then the parameter can be considered to have an impact on the overall performance.

The parameter values are selected as follows. If the parameter takes a true or false value, for instance those that refer to whether to use an optimization implemented, then the non-default value is tested. For parameters that have a variety of different values that are distinct, for instance the compression codec that will be used (snappy, lz4, lz4), all the different values are tested. Finally for parameters that take non-distinct, continuous values (e.g. spark.io.file.buffer), values that are near the default are tested.

In order to get more accurate results each experiment is conducted five times and the median value of all five is considered the result.

Regarding the cluster over which we run the experiments, MareNostrum offers the capability of requesting virtual machines based on our linking. However, the overall number of nodes and their capacity in matters of memory is kept the same thought all of the experiments. This is done not only for the sake of comparison but also for the simple fact that any performance improvement or impact should be accounted only to parameter sensitivity. Thus, keeping the cluster identical for each experiment and dataset helps us cut off the clustering optimization factor. The specifics of the setup can be seen on table 4.1
4.2. Experimentation and Parameter Sensitivity

<table>
<thead>
<tr>
<th>Number of Nodes</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Workers per Node</td>
<td>2</td>
</tr>
<tr>
<td>Number of Cores per Worker</td>
<td>8</td>
</tr>
<tr>
<td>Worker memory size</td>
<td>12GB</td>
</tr>
</tbody>
</table>

**Sort-by-Key experiments**

Sort-by-Key experiments where conducted in the fashion explained above. Both the input data size and the cluster were kept the same on each run. On table 4.2 the experiment’s details can be seen. 1000000000 Key-Value pairs are used, and each key and value have a length of 10 and 90 respectively. Also there are both 1000000 unique values for both keys and values.

<table>
<thead>
<tr>
<th>Number of Key-Value Pairs</th>
<th>1000000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key length</td>
<td>10</td>
</tr>
<tr>
<td>Value length</td>
<td>90</td>
</tr>
<tr>
<td>Unique Keys/Values</td>
<td>1000000</td>
</tr>
<tr>
<td>Number of Partitions</td>
<td>640</td>
</tr>
<tr>
<td>Number of Reduce Tasks</td>
<td>640</td>
</tr>
</tbody>
</table>

On Figure 4.2 we can see the impact of `spark.serializer`. It is obvious that KryoSerializer performs better than the default Java Serializer giving a nearly 25% speed-up. Since this gap is so big and in order to be able to extract insights regarding the performance of the rest of the parameters the experiments that followed where conducted with the KryoSerializer.

**Figure 4.2:** Performance impact of `spark.serializer` for Sort-by-key

In figure 4.3 we can see the impact of all the parameters. Different values of the same parameter are presented with the same color and as default
reference we use the `spark.serializer = KryoSerializer` which performs a bit over the 150 second mark. Depending how much higher or lower than that each parameter performs dictates its impact.

In particular from bottom to top, we see that both non-default shuffle managers perform better than the default. `Hash` performs at 127 seconds and `Tungsten-sort` at 131 seconds, nearly 30 seconds faster than the default 160.

Regarding the memory fraction parameters, the values for `shuffle.memoryFraction=0.4` and `storage.memoryFraction=0.4` does not provide any significant speed-up (139 seconds), although it is expected since this is a shuffle-heavy application so providing more memory in shuffling related issues should have yielded better results. The second test for this parameter, `shuffle.memoryFraction=0.1` and `storage.memoryFraction=0.7`, cannot be conducted of course, because due to not enough memory for the shuffling the application crashed.

`spark.reducer.maxSizeInFlight` does not appear to have any significant impact on performance either. Increasing this parameter’s value at 96mb yields the same performance with the default (167 seconds) but decreasing it to 24mb gives a bit of performance improvement (149 seconds) which is not noteworthy.

The exact same thing can be said for `spark.shuffle.file.buffer` but here increasing the value yields slightly better results at 140 seconds.

The biggest impact on performance however can be seen on the `shuffle.compression` test runs. Disabling the compression degrades the performance for more than 100%. It is worth noting that this will not always be the case. As it was stated earlier in this chapter this parameter is greatly affected by the hardware and network, so this radical performance degradation might only take place on a cluster such as the one the experiments are conducted.

Regarding the compression codecs, there is not any noteworthy impact since both `lz4` and `lz4` seem to be performing nearly exactly as the default codec, `snappy`. Also, the file consolidation implementation does not appear to provide any significant improvement either. This could be attributed to a variety of things, one being the fact that the `sort-by-key` application does not generate that many files during shuffling for this data size to make a difference.

The case for the last three parameters `spark.shuffle.spill.compress`, `spark.shuffle.io.preferDirectBufs` and `spark.rdd.compress` is the same. For the first, performing nearly the same with the default value can be attributed to
4.2. Experimentation and Parameter Sensitivity

Figure 4.3: Performance impact for Sort-by-key

The fact that the spills conducted are not that many to make any significant change to the overall performance. Spark.rdd.compress does not appear to have any particular impact also. A small performance degradation was expected since more time will be spent on the CPU for compression but it is not apparent in this application.

TeraGenandShuffle experiments

As was the case on the sort-by-key experiments in this set of runs we keep the cluster setup the same constantly along with the input size. All in all the cluster has 20 nodes of 12GB of memory, which could be considered about 10GB of clean memory available. So there is a total of about 200GB of memory. In the experiments that where conducted the total input size was set to 400GB, twice the available memory. The experiment details can be seen on table 4.3.

Table 4.3: TeraGenAndShuffle Information

<table>
<thead>
<tr>
<th>Input Size</th>
<th>400GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduce tasks</td>
<td>640</td>
</tr>
<tr>
<td>Partitions</td>
<td>640</td>
</tr>
</tbody>
</table>

As before, we first test the impact of spark.serializer and compare it with the default performance. Figure 4.4 shows the result. It is safe to say that KryoSerializer performs nearly the same with the default Java serializer. In the experiments that follow, we use this performance of the Kryoserilizer
in the default configuration and compare against it the rest of the parameter performances.

**Figure 4.4:** Performance impact of spark.serializer for TeraGenAndShuffle

![Graph showing performance impact of spark.serializer](image)

In figure 4.5 the results for the performance impact of the parameters are presented. As before values of the same parameter are grouped have a unique color.

**Figure 4.5:** Performance impact for TeraGenAndShuffle

![Graph showing performance impact for TeraGenAndShuffle](image)

Again beginning from bottom to top one can see that on this application there is little to none variation between the performance of different parameter values. For comparison purposes the completion time of the default configuration with *KryoSerializer* is 815 seconds for 400GB of input.

Contrary with the previous experiments here the *Hash* shuffle manager performs worse with nearly 200 seconds degradation. This could probably
be attributed to the fact that because the input is so much larger than the available memory the files that are created during the shuffles are indeed too many. This was one of the main reasons that Hash manager was changed from default with the Sort manager. On the other hand Tungsten-sort manager offers a nearly 90s second speedup which is about a 9% improvement.

Similarly with sort-by-key increasing the memory available for shuffling at the expense of the storage.memoryFraction offers no real change to the performance. Of course as before, doing the opposite does not leave enough memory for shuffling and the application crashes.

Regarding the spark.reducer.maxSizeInFlight parameter, there seems to be no impact to the application. This can be contributed to the fact that changing this parameter should have effect of systems that have very small memory. This should happen because even if its value is reduced to half the system gains only a few extra megabytes. In our experiments the input size is too large for those megabytes to make a real difference. The issue is considered to be the same for the spark.shuffle.file.buffer parameter. The only difference however is that when reducing the buffer size from 32k to 15k the performance degrades for about 135 seconds, which is more than 10%. This might happen for the following reason. As it states in Spark’s configuration, these buffers reduce the number of disk seeks and system calls made in creating intermediate shuffle files. When the input is big enough and the buffers are overall small the number of system calls and disk seeks increases enough to degrade the performance.

It is apparent that disabling the shuffle compression offers no improvement and greatly reduces completion time. The reason is attributed to the same bottleneck issue with the sort-by-key experiments.

Regarding the compression codec parameter, the lzf codec does not seem to have any impact on the overall performance. The lz4 codec however, increases the application completion time by about 200 seconds which is about 25% increased runtime. Since the implementation of these codecs is outside the scope of this work, this performance cannot be attributed to any specific reason.

Finally, the last three parameters configuration produce runtimes near the default value and as it was the case with the sort-by-key experiments they do not appear to have any significant impact of the performance. As stated before thought, this is only the case for this cluster setup with these applications and it does not exclude the fact that they might impact any other setup.
K-Means experiments

The final series of experiments, as it was stated before, tests the impact of the parameters on the performance of the K-Means benchmark application. The same cluster with the previous experiments is used but the only difference here is that the series of experiments is conducted two times for two different input sizes. This was done mainly because on the first series of experimentation the impact of most of the parameters was very small and it was considered best if more tests were conducted to make sure that it was not because of the input size. Table 4.4 presents the information regarding each experiment.

<table>
<thead>
<tr>
<th></th>
<th>1st experiment cycle</th>
<th>2nd experiment cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of centers</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Number of iterations</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Number of partitions</td>
<td>640</td>
<td>640</td>
</tr>
<tr>
<td>Number of points</td>
<td>100.000.000</td>
<td>200.000.000</td>
</tr>
<tr>
<td>Number of columns</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

As it was conducted in the previous experiments with tests the performance of KryoSerializer first. We can see in Figures 4.6 and 4.7 that there is no apparent impact for this parameter in both data sizes. The KryoSerializer is used on the following experiments in the default configuration.

Figures 4.8 and 4.9 present the results for each experiment cycle.

It is apparent in both figures that the impact of most of the parameters is minimal. In figure 4.8 we see that there is an at most 2 second variance in runtimes which at this amount of execution time cannot be considered
The exact same thing can be seen in figure 4.9 for the second experiment cycle. Here the variance is only 3 seconds at most.

However, while concluding that the parameters selected do not affect the performance of the K-Means benchmark one thing that should be taken into consideration. The parameter regarding the compression during shuffling `spark.shuffle.compress` that degraded dramatically the performance of the previous experiments has no overall impact on this benchmark. This should be expected since it mostly affects the data shuffling part of the application and K-Means does not shuffle that much. This must be the case...
must be for the rest of the parameters, since most of them offer implementations to the shuffling mechanism or are connected somehow to it for the most part.

**Figure 4.9: Performance impact for K-Means (200,000,000 points)**

---

### Overall Statistics

In table 4.5 the total and average impact of each parameter can be seen for each benchmark. All percentages take into account any deviation from the default runtime and the mean of the impact of each value is presented. Also in table 4.6 the maximum impact of each parameter is presents. It is worth noting for K-Means that the difference between most runtimes was just a few seconds. This, combined with a low runtime may make the certain impact of a parameter appear bigger than it is. Other than that the results care self explanatory.

### 4.3 Tuning Methodology

In this section a methodology for tuning Spark’s parameters is suggested based on the overall experience with the framework. Generally it takes into consideration the tips that are given to each parameter’s documentation and arranges them in the form of steps that when conducted, a better performance might occur. The following parameter configurations are included on the methodology.
4.3. Tuning Methodology

TABLE 4.5: Average Parameter Impact

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Sort-by-key</th>
<th>TeraGenAndShuffle</th>
<th>K-Means</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>spark.serializer</td>
<td>26.6%</td>
<td>9.2%</td>
<td>&lt;5%</td>
<td>12.6%</td>
</tr>
<tr>
<td>spark.shuffle.manager</td>
<td>19.1%</td>
<td>15.5%</td>
<td>&lt;5%</td>
<td>13.1%</td>
</tr>
<tr>
<td>shuffle/storage.memoryFraction</td>
<td>13.1%</td>
<td>11.9%</td>
<td>8.3%</td>
<td>11.3%</td>
</tr>
<tr>
<td>spark.reducer.maxSizeInFlight</td>
<td>5.5%</td>
<td>5.7%</td>
<td>11.5%</td>
<td>7.5%</td>
</tr>
<tr>
<td>spark.shuffle.file.buffer</td>
<td>6.3%</td>
<td>11.6%</td>
<td>6.9%</td>
<td>8.2%</td>
</tr>
<tr>
<td>spark.shuffle.compress</td>
<td>137.5%</td>
<td>182%</td>
<td>&lt;5%</td>
<td>107.2%</td>
</tr>
<tr>
<td>spark.io.compress.codec</td>
<td>&lt;5%</td>
<td>18%</td>
<td>6.1%</td>
<td>8.9%</td>
</tr>
<tr>
<td>spark.shuffle.consolidateFiles</td>
<td>13%</td>
<td>11%</td>
<td>7.7%</td>
<td>10.5%</td>
</tr>
<tr>
<td>spark.rdd.compress</td>
<td>&lt;5%</td>
<td>&lt;5%</td>
<td>5%</td>
<td>&lt;5%</td>
</tr>
<tr>
<td>spark.shuffle.io.preferDirectBufs</td>
<td>5.6%</td>
<td>9.9%</td>
<td>&lt;5%</td>
<td>5.9%</td>
</tr>
<tr>
<td>spark.shuffle.spill.compress</td>
<td>&lt;5%</td>
<td>6.1%</td>
<td>&lt;5%</td>
<td>&lt;5%</td>
</tr>
</tbody>
</table>

TABLE 4.6: Maximum Parameter Impact

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Sort-by-key</th>
<th>TeraGenAndShuffle</th>
<th>K-Means</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>spark.serializer</td>
<td>26.6%</td>
<td>9.2%</td>
<td>&lt;5%</td>
<td>13.2%</td>
</tr>
<tr>
<td>spark.shuffle.manager</td>
<td>20.6%</td>
<td>21.9%</td>
<td>8.3%</td>
<td>16.9%</td>
</tr>
<tr>
<td>shuffle/storage.memoryFraction</td>
<td>13.1%</td>
<td>11.9%</td>
<td>16%</td>
<td>13.6%</td>
</tr>
<tr>
<td>spark.reducer.maxSizeInFlight</td>
<td>6.8%</td>
<td>7.9%</td>
<td>22.2%</td>
<td>12.3%</td>
</tr>
<tr>
<td>spark.shuffle.file.buffer</td>
<td>12.5%</td>
<td>16.5%</td>
<td>16%</td>
<td>15%</td>
</tr>
<tr>
<td>spark.shuffle.compress</td>
<td>137.5%</td>
<td>182%</td>
<td>&lt;5%</td>
<td>108.5%</td>
</tr>
<tr>
<td>spark.io.compress.codec</td>
<td>5%</td>
<td>18%</td>
<td>6.1%</td>
<td>9.7%</td>
</tr>
<tr>
<td>spark.shuffle.consolidateFiles</td>
<td>13%</td>
<td>11%</td>
<td>7.7%</td>
<td>10.5%</td>
</tr>
<tr>
<td>spark.rdd.compress</td>
<td>&lt;5%</td>
<td>&lt;5%</td>
<td>5.5%</td>
<td>&lt;5%</td>
</tr>
<tr>
<td>spark.shuffle.io.preferDirectBufs</td>
<td>5.6%</td>
<td>9.9%</td>
<td>&lt;5%</td>
<td>6.6%</td>
</tr>
<tr>
<td>spark.shuffle.spill.compress</td>
<td>&lt;5%</td>
<td>6.1%</td>
<td>5.5%</td>
<td>5.1%</td>
</tr>
</tbody>
</table>

• spark.shuffle.manager: as seen in the results of the previous section, the shuffle manager has a high impact on performance, so it should be included in the methodology. Since tungsten-sort works better with the lzf compression codec the test run for this shuffle manager is conducted with this codec. Also, the test run with the hash shuffling manager is conducted with the implementation of consolidating files during a shuffle, since it is known that is tends to create too many files during it.

• storage/shuffle.memoryFraction: memory fraction should theoretically have the biggest impact in performance. However, in our work we see that the impact is about 13% which may not be extreme but is still enough to add them in the methodology.

• spark.shuffle.spill.compress: while this parameter appears not to have any significant impact on the performance in the experiments that
where conducted, it is highly linked with the shuffling memory fraction, and since in this methodology the later is taken into consideration, so should this one.

- `spark.shuffle.compression`: the case with this parameter is that although in our work it degrades radically the performance, there is a precedence in Apache’s spark terasort benchmarking where it yielded better results.

- `spark.shuffle.file.buffer`: while this parameter does not appear to have more than a 10% impact, it was apparent from the TeraGenAndShuffle experiments that when set to low values for big inputs the performance degraded. Since the opposite is a possibility it is included in the methodology.

In figure 4.10 the methodology for parameter tuning is presented. Each box represents a test run and the process begins from the top. Test runs are conducted with the configurations in each box beginning from the top to the bottom and the results are compared with the default. For each run if the performance with the selected configuration is improved then the parameter configuration is kept for the next test runs as the process moves downwards. If not, then the configuration is not added. When two nodes merge to one node with different values for the same parameter, then the configuration that has the best performance out of the two is selected. In other words, each parameter configuration moves downwards to the final configuration as long as it performs better. If it does not it stops being used.

Next the methodology is applied to `sort-by-key` benchmark in order to test its effectiveness. In order to avoid unnecessary runs, a threshold of about 20 seconds is set. Any performance difference over this threshold would be considered an improvement. The default performance is 218 seconds as seen on our experiments.

The first step checks if the KryoSerializer should be included on the configuration.

**Step 1**

```
spark.serializer = KryoSerializer : 160 seconds
```

*default configuration*: 218 seconds

Since the performance improvement is more than 20 seconds KryoSerializer should be selected and kept on the configuration.
4.3. Tuning Methodology

Figure 4.10: Spark Parameter tuning methodology

Step 2

**previous performance**: 160 seconds

- `spark.shuffle.manager=tungsten-sort + spark.io compression.code=lzf`: 132 sec.
- `spark.shuffle.manager=hash + spark.shuffle.consolidateFiles=true`: 128 seconds

**shuffle/storage.memoryFraction = 0.4/0.4**: 139 seconds
**shuffle/storage.memoryFraction = 0.1/0.7**: N/A seconds

- `spark.shuffle.file.buffer = 64k`: 140 seconds
- `spark.shuffle.file.buffer = 16k`: 160 seconds

In this step the configuration that pass the optimization threshold can be seen in green. For the shuffle.manager branch since both configurations perform better the fastest one is selected which is the `spark.shuffle.manager=hash + spark.shuffle.consolidateFiles=true`. For the memoryFraction branch the `shuffle/storage.memoryFraction = 0.4/0.4` configuration is selected.

Step 3
previous performance: 128 seconds
spark.shuffle.manager=hash + spark.shuffle.consolidateFiles=true +
spark.shuffle.compress=false : 398 seconds

previous performance: 139 seconds
shuffle/storage.memoryFraction = 0.4/0.4 + spark.shuffle.spill.compress=false : 140 seconds

Since no significant improvement is seen on this step the configurations are kept the same for each branch and those that were selected on the previous steps are merged.

Step 4
previous performance: 128 and 139 seconds
spark.shuffle.manager=hash + spark.shuffle.consolidateFiles=true +
shuffle/storage.memoryFraction = 0.4/0.4 : 120 seconds

It is worth noting that the overall speedup is from 218 seconds down to 120 seconds. This is an about 44% performance improvement. However most of it can be attributed to the KryoSerializer but the independent improvement for each parameter is apparent in all the steps of the methodology.
Chapter 5

Conclusions

5.1 Discussion

In this work an attempt was made to identify the impact of Apache Spark parameters on the performance of certain applications. We selected a set of parameters based on Spark’s documentation and on scenarios that we identified in which they would possibly impact the overall performance. For these parameters we conducted several experiments over three different benchmarks in order to test the framework’s sensitivity to them. The results show a nearly 30% impact on overall performance for some parameters and an average of about 10-15% for most. However, this is not the case for all of the set since some of them appear to have less than 5% overall impact. It is worth noting that for the input sizes we selected the K-Means was relatively unresponsive to any parameter value changes, contrary to the other, more shuffle reliant application we used. This could be mainly because most of the parameters that were selected refer to the shuffling mechanism in some way so applications like K-means are not affected by them. Finally a methodology for spark tuning was proposed based on our experience with Spark. When applied to the Sort-by-key application it yielded a 44% performance improvement.

5.2 Future Work

This work can be expanded in a variety of ways, the most important of which is the expansion of the parameter set that was tested. Also conducting sensitivity tests on more application types can yield interesting findings as well. In this work we did not conduct any tests on YARN which seems to be also quite popular among the community, so implementing this work to that direction could be beneficial.
Appendices
Appendix A

Spark Parameters

This section contains full list on Spark’s v1.5.2 configuration properties. Categorized based on Spark’s configuration groups each property is followed by its default value and a short description.

Application Properties

- `spark.app.name` (none) : The name given to the application. This name will be visible to the UI and logging data.

- `spark.driver.cores` (1) : The total number of cores that will be used on the driver. Available in cluster mode.

- `spark.driver.maxResultSize` (1g) : As the name implies, sets the max size of the serialized result. 0 value sets the max size to unlimited. Any use of this parameter should be based on available memory in order to avoid out of memory errors.

- `spark.driver.memory` (1g) : How much memory each executor process will have.

- `spark.extraListeners` (none) : A list of provided classes that Implement SparkListner. At SparkContext’s initialization these classes will be created.

- `spark.local.dir` (/tmp) : Directory for space usage in Spark, for output and RDD that are passed on disk.

- `spark.logConf` (false) : Logs the effective SparkConf as INFO when a SparkContext is started.

- `spark.master` (none) : Cluster manager URL.

Runtime Environment

- `spark.driver.extraClassPath` (none) : Classpath entries to prepend to the classpath of the driver.
Appendix A. Spark Parameters

- \textit{spark.driver.extraJavaOptions} (none) : JVM options for the driver. E.g. garbage collection settings or other logging.

- \textit{spark.driver.extraLibraryPath} (none) : Special library path to use when launching the JVM of the driver.

- \textit{spark.driver.userClassPathFirst} (false) : Whether to prioritize jars provided by the user over Spark's jars when loading classes.

- \textit{spark.executor.extraClassPath} (none) : Extra classpath entries to add to the classpath of executors.

- \textit{spark.executor.extraLibraryPath} (none) : Library path for when JVM of the executor is launched.

- \textit{spark.executor.logs.rolling.maxRetainedFiles} (none) : The max number of the latest rolling log files that are going to be kept in the system.

- \textit{spark.executor.logs.rolling.maxSize} (none) : The maximum file size on which the executor logs will be rolled on.

- \textit{spark.executor.logs.rolling.strategy} (none) : The strategy that is going to be used for rolling the executor logs. The available choices are bases on "time" and based on "size".

- \textit{spark.executor.logs.rolling.time.interval} (daily) : The time interval on which the logs will be rolled over. The available choices are "daily", "hourly", "minutely", or time in seconds.

- \textit{spark.executor.userClassPathFirst} (false) : Same as \textit{spark.driver.userClassPathFirst} but for executors.

- \textit{spark.executorEnv.[EnvironmentVariableName]} (none) : Adds the EnvironmentVariableName to the executor process.

- \textit{spark.python.profile} (false) : Enable profiling in Python worker, the profile result will show up by \texttt{sc.show_profiles}, or it will be displayed before the driver exiting. It also can be dumped into disk by \texttt{sc.dump_profiles(path)}.

- \textit{spark.python.profile.dump} (none) : Directory in which the results will be dumped when the driver exits.

- \textit{spark.python.worker.memory} (512m) : How much memory will each python worker process use during the aggregation phase.

- \textit{spark.python.worker.reuse} (true) : Whether to reuse a Python worker or not.
Appendix A. Spark Parameters

Shuffle Behavior

- `spark.reducer.maxSizeInFlight` (48m) : The size of the buffer that will receive each output. Since the buffer and the output need to be the same size it sets an upper bound to the output size it will be transferred.

- `spark.shuffle.blackTransferService` (netty) : What implementation will be used for transferring blocks in cache or during shuffle. There are two options, netty, and nio.

- `spark.shuffle.compress` (true) : Parameter that provides the option to compress the map output files or not. Reduces size of transferred data.

- `spark.shuffle.consolidateFiles` (false) : Whether to consolidate intermediate files that are created during shuffling.

- `spark.shuffle.file.buffer` (32k) : The size of memory buffers for the intermediate shuffle files on the output stream.

- `spark.shuffle.io.maxRetries` (3) : Only used in Netty implementation. Number of retries for IO-related exceptions.

- `spark.shuffle.io.numConnectionsPerPeer` (1) : Only used in Netty implementation. In order to reduce the number of connections that need to be initiated between the hosts, existing connections are reused. In order to avoid concurrency issues on certain machines this value can be increased.

- `spark.shuffle.io.preferDirectBufs` (true) : Only used in Netty. Whether to use off-heap buffers in order to reduce garbage collection in shuffling process and caching.

- `spark.shuffle.io.retryWait` (5s) : Only used in Netty. How long to wait between retries for data fetches.

- `spark.shuffle.manager` (sort) : What implementation is spark going to use for data shuffling. The available implementations are three: sort, hash, and tungsten-sort.

- `spark.shuffle.memoryFraction` (0.2) : What fraction of the java heap will be used for aggregation and grouping during shuffling if spilling during shuffle is activated.

- `spark.shuffle.service.enabled` (false) : Whether to preserve shuffle files that are written by executors so that they can be safely removed.
- `spark.shuffle.service.port` (7337) : Port on which the external shuffle service will run.

- `spark.shuffle.sort.bypassMergeThreshold` (200) : Whether to avoid merge-sorting data if there is no map-side aggregation and there are at most this many reduce partitions, when using the sort method in shuffle manager.

- `spark.shuffle.spill` (true) : Whether to spill data on the disk or not.

- `spark.shuffle.spill.compress` (true) : Whether to compress the data that are spilled on the disk from `spark.shuffle.spill`

**Spark UI parameters**

- `spark.evenLog.compress` (false) : whether to compress event logs.

- `spark.eventLog.dir` (file:///tmp/spark-events) : The directory in which Spark event logs will be stored.

- `spark.eventLog.enabled` (false) : Whether to use Spark’s event logging service.

- `spark.ui.killEnabled` (true) : Whether to allow killing jobs from the web UI.

- `spark.ui.port` (4040) : Port for the application’s dashboard, which shows memory and workload data.

- `spark.ui.retainedJobs` (1000) : The number of jobs that are being retained before garbage collecting.

- `spark.ui.retainedStages` (1000) : Same with `spark.ui.retainedJobs` but for stages.

- `spark.ui.retainedExecutors` (1000) : Same with `spark.ui.retainedJobs` but for executors.

- `spark.ui.retainedDrivers` (1000) : Same with `spark.ui.retainedJobs` but for Drivers.

- `spark.ui.retainedExecutions` (1000) : Same with Same with `spark.ui.retainedJobs` but for finished executions.

- `spark.streaming.ui.retainedBatches` (1000) : Same with Same with `spark.ui.retainedJobs` but for finished batches.
Compression and Serialization parameters

- `spark.broadcast.compress` (true) : Whether to compress broadcast variables before sending them to nodes.

- `spark.closure.serializer` (org.apache.spark.serializer.JavaSerializer) : What serializer class is going to be used for closures. Only JavaSerializer is supported.

- `spark.io.compression.codec` (snappy) : What codec is going to be used for compression purposes, such as compressing RDDs, shuffling data, and broadcast variables. Three options are available, snappy, lz4, lzf.

- `spark.compression.lz4.blockSize` (32k) : What block size is going to be used when the lz4 compression codec is enabled.

- `spark.compression.snappy.blockSize` (23k) : What block size is going to be used when the snappy compression codec is enabled.

- `spark.kryo.classesToRegister` (none) : A list of custom class names to register when using Kryo Serializer.

- `spark.kryo.referenceTracking` (true) : Whether to track references of the same object during serialization when using Kryo Serializer.

- `spark.kryo.registrationRequired` (false) : when using Kryo serializer , whether to require class registration or not.

- `spark.kryo.registrator` (none) : A class to register use defined classes when using Kryo Serializer.

- `spark.kryoserilalizer.buffer.max` (64m) : Maximum size of the serialization buffer. The buffer will inflate up to this size.

- `spark.kryoserilalizer.buffer` (64k) : The initial size of the of the serialization buffer.

- `spark.dd.compress` (false) : Option to compress serialized RDD partitions.

- `spark.serializer` (org.apache.spark.serializer.JavaSerializer) : The class that is going to be used in order to be serialize object over the network.

- `spark.serializer.objectStreamReset` (100) :

  When serializing using org.apache.spark.serializer.JavaSerializer, the serializer caches objects to prevent writing redundant data, however
that stops garbage collection of those objects. By calling ‘reset’ you
flush that info from the serializer, and allow old objects to be collected.
To turn off this periodic reset set it to -1. By default it will reset the
serializer every 100 objects.

**Execution Behavior parameters**

- `spark.broadcast.blockSize` (4m) : Size of the block that is going to be
  transferred during broadcast.

- `spark.broadcast.factory` (org.apache.spark.broadcast.TorrentBroadcastFactory)
  : Which broadcast implementation to use.

- `spark.cleaner.ttl` (infinite) : Option that controls how long will Spark
  remember metadata.

- `spark.executor.cores` (1 in YARN mode , all the available cores on the
  worker in standalone mode) : Number of cores that are going to be
  used on each executor.

- `spark.default.parallelism` (Local mode: number of cores on the local ma-
  chine,Mesos fine grained mode: 8 ,Others: total number of cores on
  all executor nodes or 2, whichever is larger) : Number of partitions
  returned by transformations like join etc.

- `spark.executor.heartbeatInterval` (10s) : The interval between heartbeats
  from the executor to the driver, used in order to confirm that the first
  is still alive.

- `spark.files.fetchTimeout` (60s) : Timeout time when fetching files from
  spark driver.

- `spark.files.useFetchCache` (true) : Option to use local cache when fetch-
  ing files.

- `spark.files.overwrite` (false) : Option to overwrite files added from Spark-
  Context.addFile() when file already exists.

- `spark.hadoop.cloneConf` (false) : Option to use a new Hadoop configu-
  ration for each task.

- `spark.hadoop.validateOutputSpecs` (true) : Whether to validate output
  specification used in saveAsHadoopFile and other variants.

- `spark.storage.memoryFraction` (0.6) : The fraction of Java heap that is
  going to be used for Spark’s memory cache.
Appendix A. Spark Parameters

- spark.storage.memoryMapThreshold (2m) : Size above which spark memory will map when it reads data from the disk.

- spark.storage.unrollFraction (0.2) : Fraction of spark.storage.memoryFraction to use for unrolling blocks in memory.

- spark.externalBlockStore.blockManager (org.apache.spark.storage.TachyonBlockManager) : An implementation of external block manager that stores RDD.

- spark.externalBlockStore.baseDir (System.getProperty("java.io.tmpdir") : Directories of the external block store that store RDDs.

- spark.externalBlockStore.url (tachyon://localhost:199998 for Tachyon) : The URL of the underlying external blocker file system in the external block store.

Spark Networking Parameters

- spark akka.frameSize (128) : Option that sets the maximum message size between executors.

- spark akka.heartbeat.interval (1000s) : Interval on which the failure detector will be activated.

- spark akka.heartbeab.pause (6000s) : Margin between heartbeats that is acceptable by Akka.

- spark akka.threads (4) : Number of threads that are going to be used for communication.

- spark akka.timeout (100s) : When will a communication timeout between Spark nodes occur.

- spark blockManager.port (random) : Port for all block managers to listen on. These exist on both the driver and the executors.

- spark broadcast.port (random) : Port for the driver’s HTTP broadcast server to listen on.

- spark driver.host (local hostname) : Hostname or IP address for the driver to listen on.

- spark driver.port (random) : Port for the driver to listen on.

- spark executor.port (random) : Port for the executor to listen on.
Appendix A. Spark Parameters

- `spark.fileserver.port` (random): port for the driver’s HTTP file server to listen on.

- `spark.network.timeout` (120s): Default timeout for all network interactions.

- `spark.port.maxRetries` (16): Maximum number of retries when binding to a port before stopping.

- `spark.replClassServer.port` (random): Port for the driver’s HTTP class server to listen on.

- `spark.rpc.numRetries` (3): Number of times to retry before an RPC task gives up.

- `spark.rpc.retry.wait` (3s): Duration for an RPC ask operation to wait before trying again.

- `spark.rpc.askTimeout` (120s): Duration for an RPC ask operation to wait before timing out.

- `spark.rpc.lookupTimeout` (120s): Duration for an RPC remote endpoint lookup operation to wait before timing up.

Scheduling Parameters

- `spark.core.max` (not set): The maximum number of CPU cores to be requested across the entire cluster.

- `spark.locality.wait` (3s): How much time to wait before giving up in launching a data-local task locally and launching it on another non local node.

- `spark.locality.wait.node / spark.locality.wait.process / spark.locality wait.rack` (spark.locality.wait): Option that customizes the locality wait for node/process/rack locality.

- `spark.scheduler.maxRegisteredResourceWaitingTime` (30s): Maximum amount of time to wait for resources to register before scheduling begins.

- `spark.scheduler.minRegisteredResourcesRation` (0.8 for YARN, 0.0 otherwise): The minimum resource registration ratio before scheduling will begin.

- `spark.scheduler.mode` (FIFO): Option for the scheduling mode.
Appendix A. Spark Parameters

- spark.scheduler.revive.interval (1s) : The interval length for the scheduler to revive the worker resource offers to run tasks.

- spark.speculation (false) : Option that enables task speculation. When an task runs slowly, it will be relaunched.

- spark.speculation.interval (100ms) : Time interval between speculation checks.

- spark.speculation.multiplier (1.5) : The minimum ratio between the median execution time of all the tasks and the task at hand for it to be considered for speculation.

- spark.speculation.quantile (0.75) : Percentage of tasks that must be completed before speculation is enabled.

- spark.task.cpus (1) Number of cores to allocate for each task.

- spark.task.maxFailures (4) Number of individual task failures before giving up on a job.

Dynamic Allocation Parameters

- Spark.dynamicAllocation.enabled (false) : Option on enabling dynamic Allocation. Dynamic allocation scales the number of executors based on workload.

- spark.dynamicAllocation.executorIdleTimeout (60s) : The maximum amount of time an executor can be idle before it is removed when dynamic allocation is enabled.

- spark.dynamicAllocation.cachedExecutorIdleTimeout (infinity) : The maximum amount of time an executor that has cached data can be idle before it is removed when dynamic allocation is enabled.

- spark.dynamicAllocation.maxExecutors (infinity) : The maximum number of executors that can be allocated.

- spark.dynamicAllocation.minExecutors (infinity) : The minimum number of executors that can be allocated.

- spark.dynamicAllocation.initialExecutors (spark.dynamicAllocation.minExecutors) : Initial number of executors when dynamic allocation is enabled.

- spark.dynamicAllocation.schedulerBacklogTimeout (1s) : The maximum time a task can be pending before new executors will be requested.
Appendix A. Spark Parameters

- `spark.dynamicAllocation.sustainedSchedulerBacklogTimeout` (schedulerBacklogTimeout): Same as `spark.dynamicAllocation.schedulerBacklogTimeout`, but used only for subsequent executor requests.

Spark Security Parameters

- `spark.acls.enable` (false): Option for whether spark acls should be enabled. Spark acls requires users to have permission in viewing or modifying a job.

- `spark.admin.acls` (Empty): A list of all admins and users that have access to all Spark jobs.

- `spark.authenticate` (false): Option for whether Spark will require authentication for internal connections.

- `spark.authenticate.secret` (None): The secret key that is going to be used for Spark internal authentication.

- `spark.authenticate.enableSaslEncryption` (false): Option to enable encrypted authentication.

- `spark.network.sasl.serverAlwaysEncrypt` (false): Disable all unencrypted connections if a service supports SASL authentication.

- `spark.core.connection.ack.wait.timeout` (60s): Maximum time to wait for ack to occur during a connection.

- `spark.core.connection.auth.wait.timeout` (30s): Maximum time to wait for a connection authentication before timing out.

- `spark.modify.acls` (Empty): List of all users that can modify a spark job.

- `spark.ui.filters` (none): list of filter class names to apply to the Spark web UI.

- `spark.ui.view.acls` (Empty): List of users that have view access to Spark web UI.

Encryption

- `spark.ssl.enabled` (false): Option to enable SSL connections on all supported protocols.

- `spark.ssl.enabledAlgorithms` (Empty): List of ciphers, must be supported by JVM.
Appendix A. Spark Parameters

- `spark.ssl.keyPassword` (None) : The password to the private key in key-store.

- `spark.ssl.keyStore` (None) : A path to a key-store file.

- `spark.ssl.keyStorePassword` (None) : The password to the key-store.

- `spark.ssl.protocol` (None) : A protocol name, must be supported by JVM.

- `spark.ssl.trustStore` (None) : A path to a trust-store file.

- `spark.ssl.trustStorePassword` (None) : The password to the trust-store

Spark Streaming Parameters

- `spark.streaming.backpressure.enabled` (false) : Option for enabling Spark Streaming’s internal backpressure mechanism. The internal backpressure mechanism sets a limit to the rate in which the system can receive data so it does not receive data faster than it can process.

- `spark.streaming.blockInterval` (200ms) : Interval at which data received by Spark Streaming receivers is chunked into blocks of data before storing them in Spark.

- `spark.streaming.receiver.maxRate` (not set) : The maximum rate at which receivers will receive data.

- `spark.streaming.receiver.writeAheadLog.enable` (false) : Option to enable the write ahead logs for receivers. These logs have input data that are received so that they can be recovered after possible driver failures.

- `spark.streaming.unpersist` (true) : Option to automatically unpersist RDDs that are generated by Spark Streaming.

- `spark.streaming.kafka.maxRatePerPartition` (not set) : Maximum rate (number of records per second) at which data will be read from each Kafka partition when using the new Kafka direct stream API.

- `spark.streaming.kafka.maxRetries` (1) : Maximum number of consecutive retries the driver will make in order to find the latest offsets on the leader of each partition (a default value of 1 means that the driver will make a maximum of 2 attempts). Only applies to the new Kafka direct stream API.

- `spark.streaming.ui.retainedBatches` (1000) : How many batches the Spark Streaming UI and status APIs remember before garbage collecting.
Appendix A. Spark Parameters

SparkR Parameters

- `spark.r.numRBackendThreads (2)` : Number of threads used by RBackend to handle RPC calls from SparkR package.

- `spark.r.command (Rscript)` : Executable for executing R scripts in cluster modes for both driver and workers.

- `spark.r.driver.command (spark.r.command)` : Executable for executing R scripts in client modes for driver. Ignored in cluster modes.


