TUNING SPARK PERFORMANCE

by

Jie Ni Zhang

A thesis submitted to the School of Computing

In conformity with the requirements for

the degree of Master of Science

Queen's University

Kingston, Ontario, Canada

(August, 2018)

Abstract

In the big data era, big data frameworks play a vital role in storing and processing large amounts of data, providing significant improvements in performance and availability. Spark is one of the most popular big data frameworks, providing high scalability and fault-tolerance with its unique in-memory engine. To hide the complex settings from users, Spark has approximately 200 configurable parameters in the execution engine. Default values assigned to the parameters provide initial ease of use. However, the default values are not the best setting for all workloads. In this work, we propose a general tuning algorithm named QST, Queen's Spark Tuning, to help users with tuning Spark and to improve overall performance. First of all, we study Spark performance for a variety of workloads and identify 9 tunable parameters among more than 200 parameters that have significant impact on performance. Then, we propose QST, a general greedy iterative tuning algorithm for our set of 9 key parameters. By classifying Spark workloads as memory-intensive, shuffle-intensive or all-intensive, QST configures the parameters for each type of workload. We perform an experimental evaluation of QST using benchmark workloads and industry workloads. In our experiments, using QST significantly improves Spark performance. Overall, using QST yields an average speedup of 65% for our benchmark evaluation workloads and 57% for our industry evaluation workloads.

Acknowledgements

First of all, I would like to express my most sincere gratitude to my supervisor, Professor Patrick Martin for his continuous support of my Master study, all his patience, encouragement, and advice provided during my study. I have been extremely lucky to have a supervisor like him who cared so much about my work, and who helped me patiently in writing this thesis.

I have had an amazing time working at Database Systems Laboratory and have the privilege to know so many bright young researchers in big data. I would like extend my gratitude to thank my lab-mates: Shady Samir Mohamed Khalifa, Yahia Elshater and Chris Barnes for all the support and help.

I would like to thank my thesis committee, Professor Mikhail Nediak, and Professor Farhana H. Zulkernine, who have provided me with insightful feedback. I would also like to thank Queen's School of Computing for providing me the opportunity to do my Master study and funding my research.

My sincere thanks also go to Scotiabank DSA team: Karl Wouterloot, Bin Jiang, Jharna Deshmukh, Professor Mikhail Nediak and all members who have provided me an opportunity to have evaluation experiment on Scotiabank's server with their large-scale industrial datasets.

Shout-out to my friends for all the support and fun you brought to me. Special thanks to Ray for his patience and understanding during my study.

And finally, I would like to thank my parents who always encourage me, comfort me and cheer me up from the other side of the world. My study would not be possible without your support.

Table of Contents

Abstract ii
Acknowledgements iii
List of Figures vii
List of Tables viii
Chapter 1 Introduction 1
1.1 Motivation 2
1.2 Contributions 2
1.3 Thesis Outline 3
Chapter 2 Background and Related Work 4
2.1 Big Data Analytics Frameworks 5
2.1.1 Hadoop 5
2.1.2 Spark 9
2.1.3 Spark vs. Hadoop 12
2.2 Previous Studies 13
Chapter 3 Tuning Algorithm: QST 17
3.1 Overview of QST 17
3.2 Workloads 20
3.2.1 Workload Types 20
3.2.2 SparkBench 21
3.2.3 Machine Learning Workloads 22
3.3 Selection of Configuration Parameters 24
3.4 Algorithm 28
3.4.1 Algorithm Input 29
3.4.2 Configuring Environment Parameters 29
3.4.3 Configuring Specific Parameters 32
3.4.4 Configuring Serializers and Compression Codec 35
Chapter 4 Evaluation 37
4.1 Benchmark Workloads 37
4.1.1 Environment 38
4.1.2 Datasets and Workloads 39
4.1.3 Analysis and Results 40
4.1.4 Influence of the Tunable Parameters 43

4.2 Industry Workloads....46
4.2.1 Environment....46
4.2.2 Datasets and Workloads....47
4.2.3 Results and Analysis....48
Chapter 5 Conclusion....52
5.1 Contributions....53
5.2 Limitations....54
5.3 Future Work....54
Bibliography....56
Appendix A Benchmark Workloads....60
Appendix B Industry Workloads....63

List of Figures

Figure 1 Big Data Characteristics Adapted from [9] 5
Figure 2 Architectures of Hadoop 1.x and Hadoop 2.x 6
Figure 3 Components Architecture of Hadoop 1 7
Figure 4 Components Architecture of Hadoop 2 9
Figure 5 Spark Ecosystem 12
Figure 6 Trial-and-error Spark Tuning Methodology Proposed by [6]. 14
Figure 7 Spark Tuning Methodology Proposed by [7] 15
Figure 8 ACS Architecture [8] 16
Figure 9 General Structure of QST 18
Figure 10 Detailed Tuning Algorithm of QST 19
Figure 11 Detailed Transitions in Each Block 20
Figure 12 Configure Environment Parameters 30
Figure 13 Configure Specific Parameters 33
Figure 14 Configure Serializers and Compression Codecs 35
Figure 15 Improvement Percentage for Each Workload 41
Figure 16 Number of Runs for Each Workload 41
Figure 17 Average and Max Speedup Achieved by Environment Configuration Parameters 44
Figure 18 Average and Max Speedup Achieved by Specific Configuration Parameters 45
Figure 19 Average and Max Speedup Influenced by Serializer and Compression codec 46
Figure 20 Improvement Percentage 49
Figure 21 Total Runs During Each Phase 49

List of Tables

Table 1 Configuration Parameters Selected from Previous Studies 25
Table 2 Selected Configuration Parameters 27
Table 3 Initial Input 29
Table 4 Experiment that applied a K-means workload on a 5G dataset 32
Table 5 Cluster Configurations 38
Table 6 Workloads and Datasets 39
Table 7 Default Setting for Benchmark Workloads 40
Table 8 Industry Workloads and Input Dataset 47
Table 9 Default Setting for Industry workloads 48

Chapter 1

Introduction

In the big data era, everyone is generating data through various sources in various formats. As the volume of data grows and the variety of data increases rapidly, traditional techniques used to process data on a single machine are not able to provide an efficient way to store or analyze these massive data. The idea of distributed computing is raised to solve the problem. Big data frameworks based on the concepts of distributed computing play a vital role in storing and processing large amounts of data [1], providing significant improvements in performance and availability.

Hadoop [2] was the leading framework and nearly synonymous with big data analytics for many years. Its MapReduce [3] is a pioneer cluster computing model, implementing large-scale data applications and achieving scalability and fault tolerance. However, as a disk-based model, MapReduce is not suitable for iterative jobs such as common machine learning algorithms that involve multiple iterations using the input data dataset or intermediate results, and interactive analytics that access the same dataset multiple times. MapReduce jobs must read or retrieve the same piece of data from disk repeatedly, incurring a significant performance penalty.

Another more advanced framework, Spark, supports efficient reuse of a dataset in memory and provides similar scalability and fault tolerance properties to MapReduce. Spark [1] was

developed in the AMPLab of the University of California at Berkeley and open-sourced by Apache. The main abstraction in Spark is the resilient distributed dataset (RDD) [4], which is a read-only collection of partitioned records. An RDD is created through transformations on data in stable storage or on other RDDs. All transformations of RDDs are logged in a lineage stored in the Spark program driver. Using lazy evaluation, RDDs are not always materialized in order to save memory space. The lineage provides enough information to compute an RDD from data in stable storage when it is needed for an action. Because of this unique feature, Spark is able to process data in main memory up to 100 times faster than Hadoop [5].

1.1 Motivation

As a unified engine for big data analytics, Spark provides high scalability and fault-tolerance with its unique in-memory engine. There are approximately 200 configurable parameters in the Spark execution engine so default values are assigned to the parameters to hide the complex settings from users and provide initial ease of use. The default values, however, are not the best setting for all workloads. Since Spark is a relatively new framework, studies related to improving performance by tuning configurations are limited. Previous studies [6, 7, 8] propose tuning methodologies for their selected benchmarks. Our research goal is to develop a general tuning algorithm to help users with tuning Spark and to improve overall performance.

1.2 Contributions

This thesis makes 3 contributions:

1) We study Spark performance for a variety of workloads and identify 9 tunable parameters among the more than 200 parameters that have significant impact on

performance. By understanding the meanings of all the parameters and summarizing ideas from previous studies, we select 9 parameters used in our tuning algorithm.

2) We propose a general greedy iterative tuning algorithm for our set of 9 key parameters. We classify Spark workloads as memory-intensive, shuffle-intensive or all-intensive and our algorithm tunes the parameters for each type of workload.
3) We perform an experimental evaluation of our algorithm and analyze the importance of the 9 tunable parameters. We evaluate our algorithm and measure the average speedup percentage achieved for a range of workloads.

1.3 Thesis Outline

The remainder of the thesis is structured as follows. In Chapter 2, we present the background of big data analytics frameworks and related works regarding tuning Spark configurations. Chapter 3 illustrates our tuning algorithm and discusses the tunable parameter selection. Chapter 4 presents the results of our experimental evaluation of our algorithm using both benchmark workloads and an industry workload from the Data Science and Analytics (DSA) lab which is part of the Global Banking and Markets group of Scotiabank. Finally, we conclude this thesis with a summary of our evaluation, and limitations of our work and provide directions for future work in Chapter 5.

Chapter 2

Background and Related Work

The term "Big Data" is commonly characterized by 5 V's: Volume, Velocity, Variety, Veracity and Value [9] as shown in the Figure 1. In brief, big data is the huge **volume** of data, generated at a rapid **velocity,** in different **varieties** and with uncertain **veracity**. Big data analytics is the process of extracting the **value** from the heterogeneous big data such that uncovering and finding insights, hidden patterns, unknown correlations, and other useful information. Currently, big data analytics plays important roles in many organizations from various areas such as health care, finance, and the environment [9]. Organizations collect and store huge amounts of data from different sources and transform the raw data into valuable information by applying big data analytics techniques.

To address the big data challenges such as huge volume, rapid velocity and expanded variety of data, big data frameworks are proposed to store, process and analyze the heterogeneous big data. There are a number of big data frameworks available for different types of analytics. Typical big data analytics frameworks are data-parallel, involving the distributed computing model [10]. The main concept is to distribute data across multiple compute nodes to be operated on concurrently. We introduce and compare two big data frameworks, Spark [5] and its predecessor, Hadoop [2] in Section 2.1. In this thesis, we focus on improving Spark performance. The goal of this thesis is to improve Spark performance by tuning its configuration parameters. In Section 2.2, we survey previous studies related to tuning Spark performance.

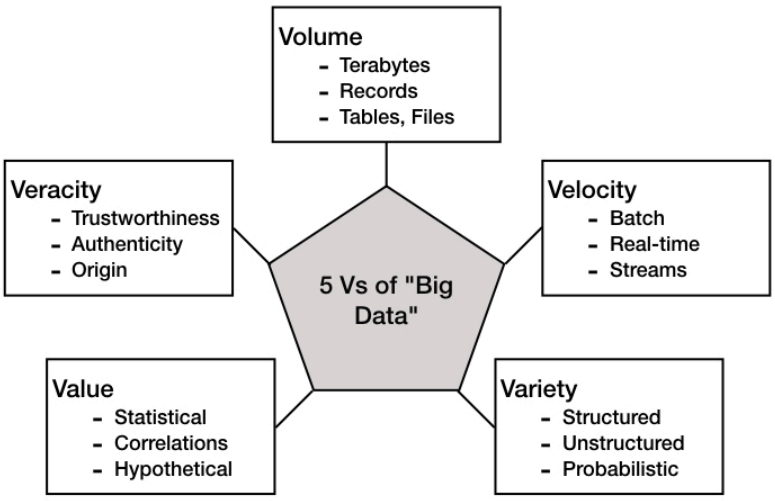

Figure 1 Big Data Characteristics Adapted from [9]

2.1 Big Data Analytics Frameworks

Both Hadoop [2] and Spark [5] are based on the distributed computing model. Hadoop [2], developed in 2005, was the leading framework and almost synonymous with the term "big data" for several years. Spark [5], a newer and more advanced framework, was developed in the AMPLab at the UC Berkeley in 2009 and has become increasingly popular. Hadoop and Spark are introduced in the section 2.1.1 and section 2.1.2 respectively. In section 2.1.3, a comparison between Hadoop and Spark is provided.

2.1.1 Hadoop

Apache Hadoop [2] is an open source big data framework that processes large datasets by distributing data across clusters of instances. Hadoop underwent a huge overhaul in Hadoop 2.0. The differences between the Hadoop 1.x and Hadoop 2.x architectures are shown in Figure 2.

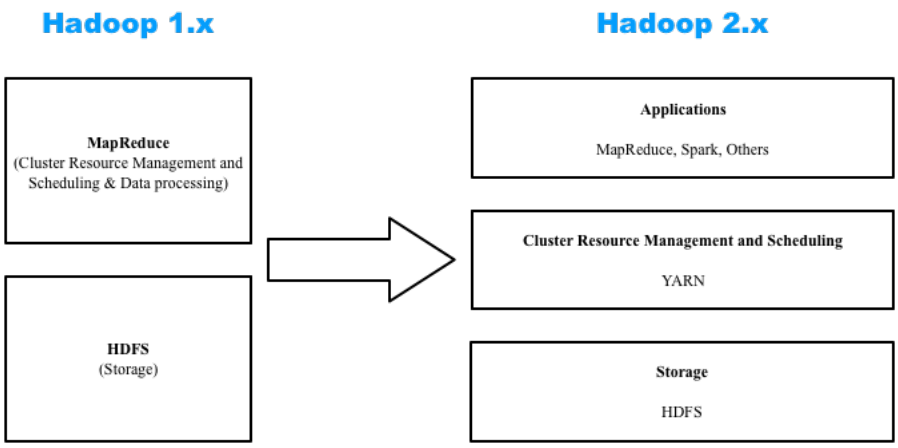

Figure 2 Architectures of Hadoop 1.x and Hadoop 2.x

Originally, Hadoop 1.x and earlier versions had two primary components at the core of Hadoop as shown in the right side of Figure 2: MapReduce [11] and Hadoop Distributed File System (HDFS) [12]. HDFS is Hadoop's default storage system. Data is stored and replicated across a number of data nodes. MapReduce [11] is a foundation batch processing programming model that allows for distributed processing of huge datasets based on the map and reduce operations. Usually, MapReduce is composed of three steps: 1). Map: On each worker node, the map function is applied to the local data and writes the output to a temporary file. The map function takes a set of data records and generates a set of (key, value) pairs; 2). Shuffle: Worker nodes distribute data based on the key value produced by map function, such that all pairs with a certain key are located on the same worker node; 3). Reduce: Each worker node applies the reduce function to each group of output data in parallel. The MapReduce engine parallelizes the computation across clusters of nodes to make efficient use of the network and disks and significantly improve performance.

In Hadoop 1.x, the MapReduce engine not only takes responsibility for processing data, but also for managing resources and scheduling jobs. The architecture components are shown in Figure 3. In a cluster, MapReduce consists of one *JobTracker* placed in a master node and multiple *TaskTrackers* placed in slave nodes. Each *TaskTracker* is configured with a fixed number of map slots and reduce slots. Once a MapReduce job is submitted, the *JobTracker* pushes map or reduce tasks to the *TaskTracker* with available slots that is closest to the data.

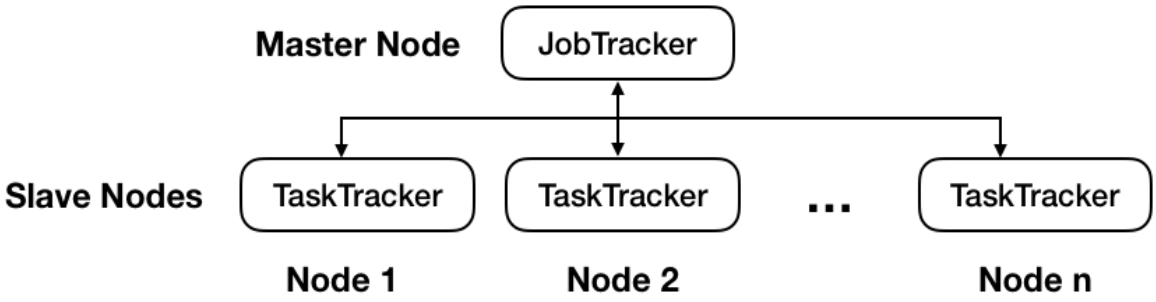

Figure 3 Components Architecture of Hadoop 1

The main limitations of Hadoop 1 include the slot-based resource allocation and limited application, since it only supports MapReduce applications and it cannot efficiently reuse the resources assigned to the idle slots. To resolved these limitations, the Hadoop Community has redesigned the architecture into Hadoop 2.x shown in the left side of Figure 2. The major key component change is that a new feature, Yet Another Resource Negotiator (YARN) is included to take the responsibilities of *JobTracker* and *TaskTrackers* from the MapReduce engine of Hadoop 1.x.

Hadoop 2.x has the following three layers:

a) Storage Layer: contains storage systems such as HDFS [12], providing high-throughput and low-latency access to application data.

b) Resource Management and Scheduling Layer: contains Yet Another Resource Negotiator (YARN) [13] which is the core concept of Hadoop 2.x. It takes responsibility for job scheduling and cluster resources management. In Hadoop 2.x, as shown in Figure 4, three components, a global *ResourceManager*, a per-application *ApplicationMaster*, and a per-node *NodeManager*, are proposed to split up the responsibilities of the *JobTracker* and *TaskTrackers*. The *ResourceManager* placed on the master node is designed to arbitrates resources among all the applications in the system. *ApplicationMaster* takes charge of managing application life-cycle, acquiring required resources for the application from *ResourceManager*, and interacting with the *NodeManagers* to execute tasks. In Hadoop 2.x, containers, which take the place of slots, incorporate resource elements such as memory, CPUs, and disk. Each node contains a set of containers. The size of containers varies by the requests of each application. The *NodeManagers* located on each node are responsible for managing the life-cycle of the containers and monitoring resources utilization.

c) Application Layer: contains data processing engines for parallel processing of huge datasets including Hadoop default engine, MapReduce [11] and other compatible engines

such as Spark [5]. MapReduce in Hadoop 2.x is just a data processing engine. Its *JobTracker* and *TaskTrackers* no longer exist.

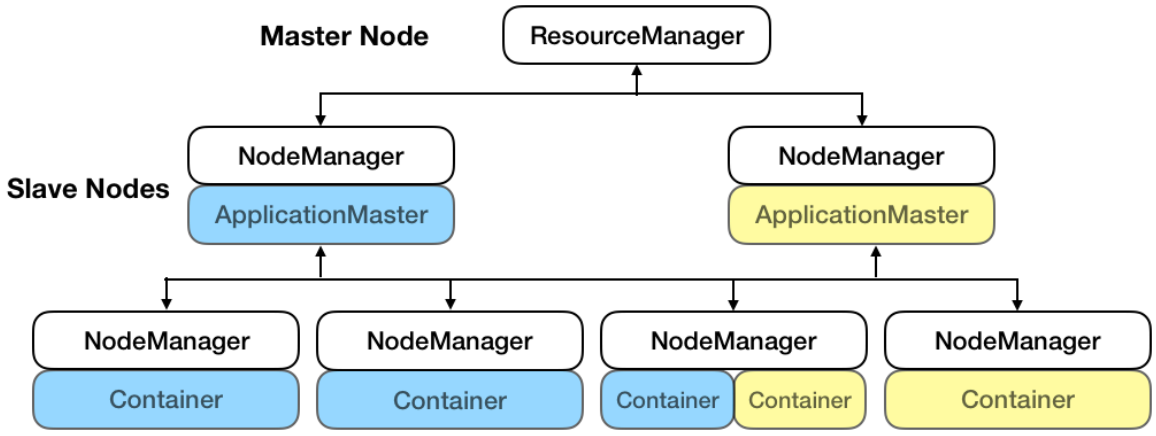

Figure 4 Components Architecture of Hadoop 2

Hadoop 2.x provides better resources utilization and greater usability. It allocates the proper amount of resources to each application instead of having a fixed number of slots for tasks. Since the scheduling and resource management are separated from the data processing component, Hadoop is able to run various types of applications other than MapReduce. Other data processing engines such as Spark can be built on the top YARN, dynamically sharing cluster resources with other frameworks run on YARN and taking advantage of all the features of YARN. Moreover, Hadoop provides built-in fault-tolerance mechanisms to detect and handle failure as well as assuring high-availability at the application layer [13]. Hadoop also provides scalability to handle increased or expanding workload in a capable manner. According to a Yahoo! report, Hadoop can scale to thousands of nodes [13].

2.1.2 Spark

Spark [5] is a distributed computing framework for large-scale datasets, that unifies processing of streaming, batch, and interactive workloads. Its key programming abstraction is resilient distributed datasets (RDDs), which are read-only, partitioned collections of objects that are created through deterministic operations on data either from storage such as HDFS, or on other RDDs. There are two types of RDD operations, transformations and actions. A transformation is a function that takes the existing RDD as input and produces a new RDD, for instance, map, filter and join are examples of transformations. A new RDD is created each time a transformation is applied since RDDs are immutable and not changeable. Applying transformations builds a lineage of an RDD. A lineage, which is kept in stable storage, is a Directed Acyclic Graph (DAG) containing the entire parent RDDs and serves as a log to record all transformations applied to a dataset.

Spark uses lazy evaluations for RDDs, so that transformations are only computed from the lineage when required by an action [4]. Actions are operations that return a value to the application or export data to a storage system, including *count()* which returns the number of elements, *collect()* which returns the elements, and *save()* which outputs the elements to a storage system. Frequently used RDDs can also be persisted in memory cache to keep the elements around for much faster future reuse.

Spark RDDs provide a unique built-in fault-tolerance mechanism. Traditionally, distributed computing systems achieve fault-tolerance by applying data replication or checkpoints. Spark uses the lineage to rebuild lost partitions of RDD. Since it records all transformations of RDDs, lost data can simply be reconstructed without costly replication.

Spark is a data processing engine that typically runs in the application layer on top of a resource management layer like YARN or Mesos [14] and a distributed storage system like HDFS. It also has its own cluster mode called Spark standalone mode. Currently, Spark is in use in variety of organizations such as technology companies, banking, retail, biotechnology, and astronomy [5]. Spark also provides great scalability. As the amount of workload expands, Spark scales out by adding more nodes to improve the processing capability. The known largest size of cluster among these user cases is 8,000 nodes [15].

As a unified data processing engine, Spark powers multiple specialized and integrated processing libraries implemented over the core engine to improve usability. Prior to Spark's release, users had to use several different specialized frameworks to handle different types of jobs. Copying data between these different systems creates complexity and inefficiency. Spark removes the complexity by using higher-level libraries, which target many of the use cases with specialized requirements. Figure 5 illustrates the Spark ecosystem. Spark SQL [16] is used to process structured data and execute SQL queries like a relational database. SparkR [17] is used to execute R language, which is widely used among statisticians for data analysis. MLlib [18] is a machine-learning library, providing several machine learning algorithms such as classification, clustering and regression. GraphX [19] is a library used for graphs and graph-parallel computation. Spark Streaming [20] is used for real time data processing. Although Spark is designed as a batch-processing framework, it can be used to process streaming data using the concept of micro-batches provided by Spark Streaming. All these libraries operate on RDDs as the data abstraction,

so all of them can be combined together in a single application seamlessly to offer significant benefits to users.

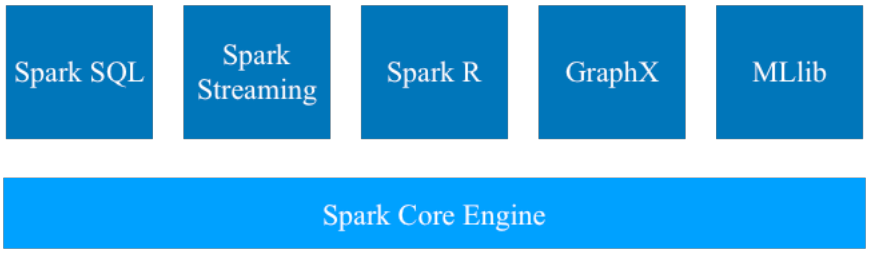

Figure 5 Spark Ecosystem

2.1.3 Spark vs. Hadoop

Both Hadoop, and Spark provide great scalability and fault-tolerance. As the pioneer of big data frameworks, Hadoop has a more complete structure with storage, cluster manager and data processing engine. Spark is designed as a data processing engine which can be part of the Hadoop application layer and run natively with Hadoop components.

The main advantage of Spark is speed. To overcome disk I/O limitations, Spark caches data in memory as much as possible using RDDs to reduce disk I/O. As an in-memory computation application, Spark runs up to 100 times faster than Hadoop [5]. Hadoop is not suitable for iterative machine learning algorithms and interactive data exploration because of its inefficient reuse of intermediate results. On average, Spark outperforms Hadoop by up to 20 times in iterative algorithms.

Generally, Spark is a superset of MapReduce. However, the cost is also an important concern because memory is more expensive than disk. When data is too large to fit in memory, Spark can also use disk to process data, but performance degrades. Without using in-memory cache, it still outperforms Hadoop MapReduce. For example, Spark set the record for the GraySort benchmark in 2014 [21]. Spark sorted 100 TB of data in 23 minutes using a cluster of 207 nodes on disk. The previous world record was 72 minutes using Hadoop MapReduce on a cluster of 2100 nodes in 2013. By contrast, Spark sorted the same amount of data approximately 3 times faster using 10 times fewer nodes [21]. Sorting is a challenge and also a good measurement because there is no reduction of data along the pipeline progress. In other words, unlike operations such as filter and join, sorting has the same amount of input and output data.

2.2 Previous Studies

As Spark has become more widely used, performance problems have been exposed in its use in practical applications. Tuning Spark has therefore become a problem worthy of research. Compared to Hadoop, the research related to performance optimization of Spark is still limited.

Wan and Khan [22] propose a simulation driven prediction model which predicts job performance with high accuracy for Spark. It predicts the execution time and memory usage of Spark applications in the default parameter case but it is not able to predict the execution time with different configuration parameters.

Petridis et al propose a trial-and-error Spark tuning methodology [6] to improve overall performance by focusing on 12 key configurable parameters. The approach shown in Figure 6 illustrates the path of finding the best configuration. As long as performance improves, each configuration is propagated downstream until the final configuration is reached. In an evaluation using three case studies, this methodology is shown to achieve up to more than a 10-fold speedup. However, it has some limitations such that configurations at the lower parts in this methodology can only be configured after configuring upper parts and this methodology is built based on the experimental evidence, in other words, it may not be a general method for all cases.

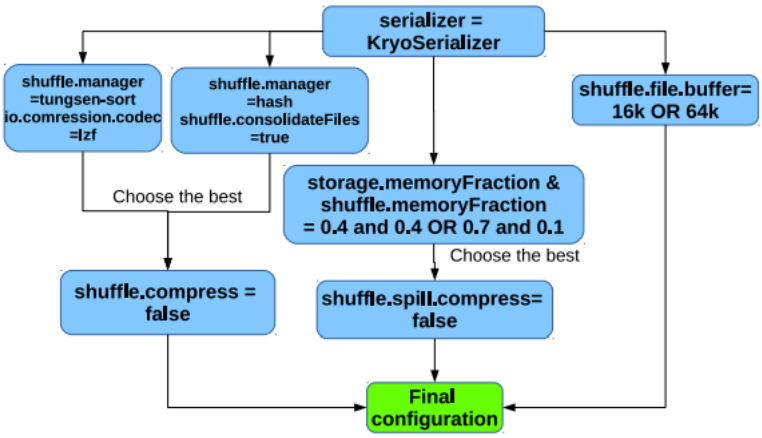

Figure 6 Trial-and-error Spark Tuning Methodology Proposed by [6].

Later work by Gounaris et al [7] proposes an alternative systematic methodology for parameter tuning based on this trial-and-error methodology of Petridis [6]. The methodology illustrated in Figure 7 has 4 phases. It profiles the impact of pairs of parameters, applies a graph algorithm to create complex candidate configurations, tests the set of candidate configurations in parallel and finally chooses the best performing one. This

methodology also focuses on 12 configurable parameters with 15 different values in total. It is evaluated using three real-world case studies and the results show speedups of at least 20% in the running time. Although this methodology is more precise, it still has the limitation of modifying numeric values for each parameter because all parameter values are set during testing in the phase 1. QST is more flexible on modifying the value configuration parameters. It is better to find the best value by adding increment at each step. For example, the configuration parameter *spark.reducer.maxSizeInFlight* can only be set to 72 MB or 24 MB with this new methodology, but with QST, this parameter can be set to more values since the value increases every step until performance degrades. Moreover, both tuning methodologies are run on the Spark 1.x platform and configurable parameters such as spark.shuffle.manager and spark.storage.memoryFraction are deprecated in later releases, but QST is developed based on the Spark 2.x platform. Therefore, QST only includes the configuration parameters which are still in use.

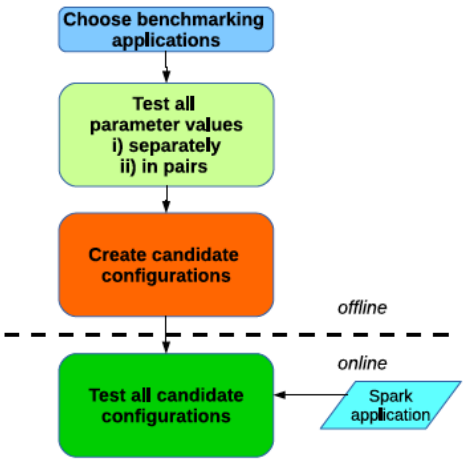

Figure 7 Spark Tuning Methodology Proposed by [7]

A recent work [8] proposes an approach called ACS to automatically configure Spark workloads by applying the random forests algorithm. ACS consists of three stages: collecting, modeling and searching as shown in the Figure 8. ACS first collects performance and configuration data with different values, and then applies the random forest algorithm to construct a performance model for each benchmark. Finally, ACS leverages a genetic algorithm to search the optimum configuration involving 29 configuration parameters. According to the results of the evaluation, ACS can speed up 6 representative Spark workloads by a factor of 2.2x on average and up to a maximum of 8.2x. Since this approach builds a performance model for each workload, it costs a lot of time during the collection phase. Time cost tables show that each workload spends around 22 hours, on average, collecting data for the training model. QST is a general iterative method, it aims at tuning Spark performance by classifying the three types. In other words, QST has three tuning strategies instead of having one unique tuning strategy for each workload. Therefore, QST saves lots of time on collecting data and building separate models for each workload.

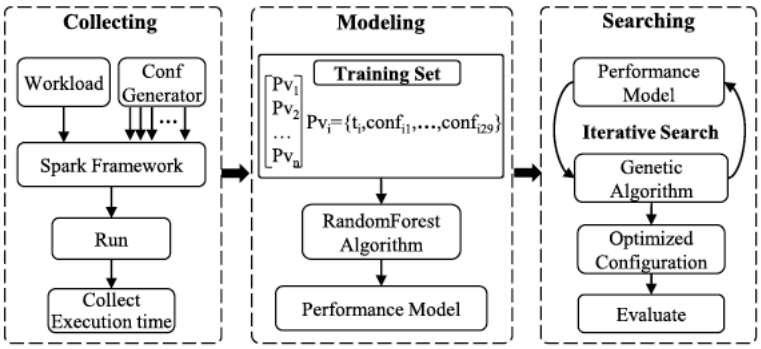

Figure 8 ACS Architecture [8]

Chapter 3

Tuning Algorithm: QST

In this chapter we present our approach, Queen's Spark Tuning (QST), to tuning the configuration of a Spark cluster to achieve optimal performance for different workloads. We discuss the design of the algorithm and the selection of appropriate configuration parameters based on an application's properties relating to shuffle behavior, compression and serialization, and memory management. In Section 3.1, we provide an overview of our algorithm. Section 3.2 describes the three workload types and their characteristics. Section 3.3 explains and discusses the selection of configuration parameters for the algorithm. In Section 3.4, we present QST in detail.

3.1 Overview of QST

The general structure of QST is illustrated in figure 9. First, the initial inputs such as dataset and workload information are accepted for use in further steps. Then, our algorithm iteratively guides the user in setting the configuration parameters until performance can no longer be improved.

QST can be divided into three phases: (1) configuring environment parameters, (2) configuring specific parameters, and (3) configuring serializers and codecs. The detailed steps of the algorithm are elaborated in figure 10. The first phase configures environment parameters, such as those related to resource allocation. The second phase of the algorithm configures specific parameters based on a classification of the workload as memory-

intensive, shuffle-intensive or all-intensive. The third phase configures the Spark serializer and compression codecs.

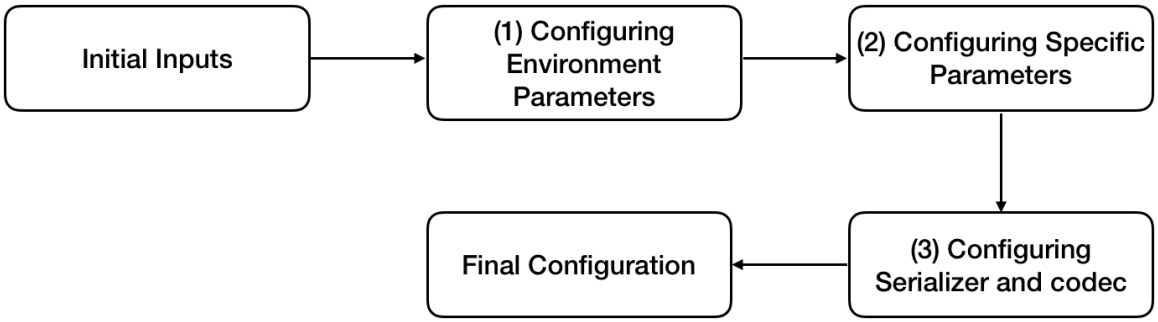

Figure 9 General Structure of QST

Figure 10 illustrates the iterative process of QST in the form of block diagram. The whole tuning process starts from the top box and configures parameters in sequence following the path. In the diagram, each box has two transitions which are elaborated in the figure 11. Each box represents one or more tuning runs altering one configuration parameter. Beside each box, increments and maxima or minima are labeled. In general, if configuring an individual parameter improves the performance, the algorithm keeps configuring the parameter by the increment until the performance degrades or its value reaches the maxima or minima. If the performance doesn't improve at a certain point, the configuration of the current parameter ends and the previous value of the configuration parameter is kept and passed to its children. In other words, the new value replaces the default value of the certain configuration parameter for all the subsequent runs in the path. In QST, the duration time is considered as the measurement of performance. The main concept of QST is to propagate the best configuration of each box downstream to the next box.

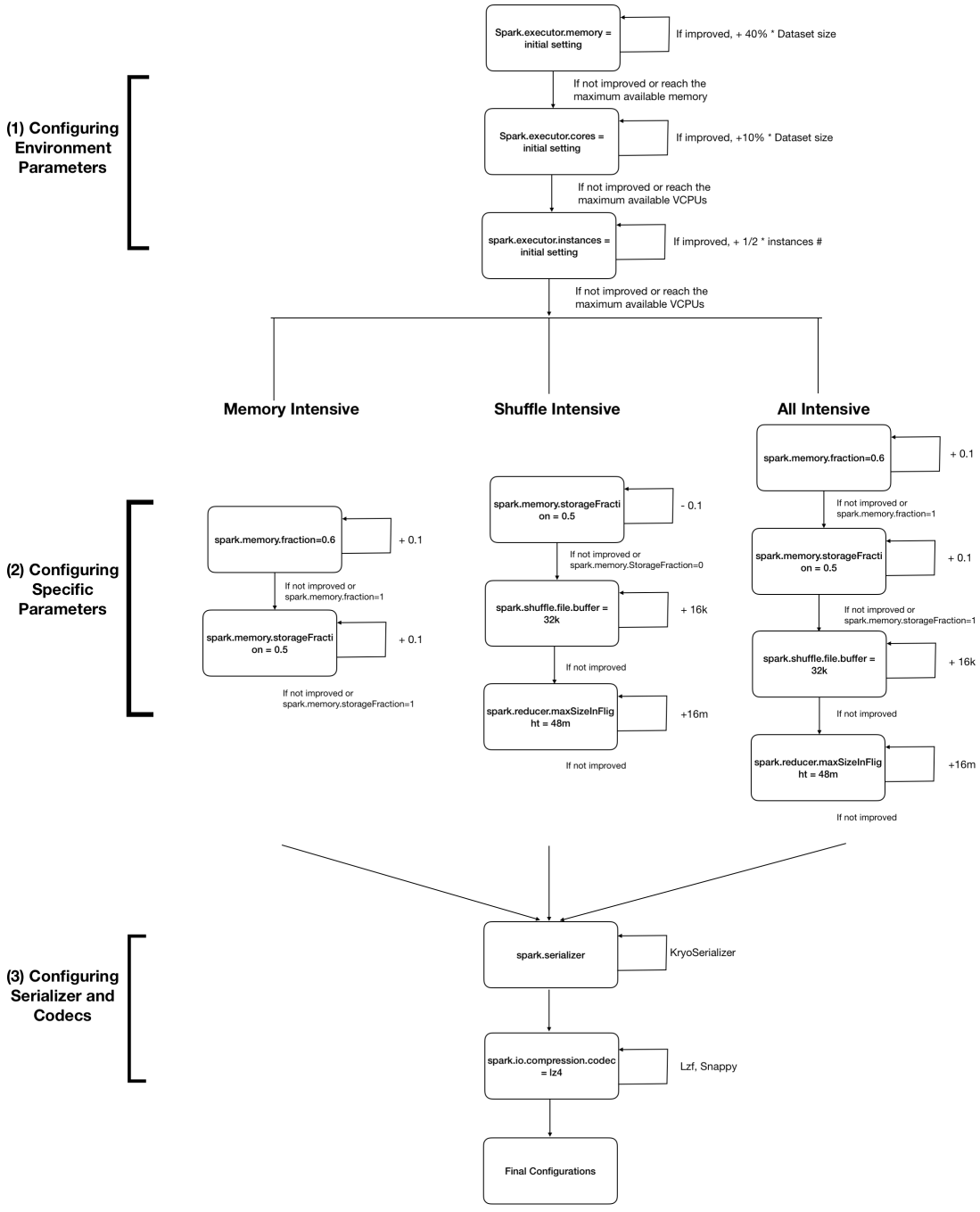

Figure 10 Detailed Tuning Algorithm of QST

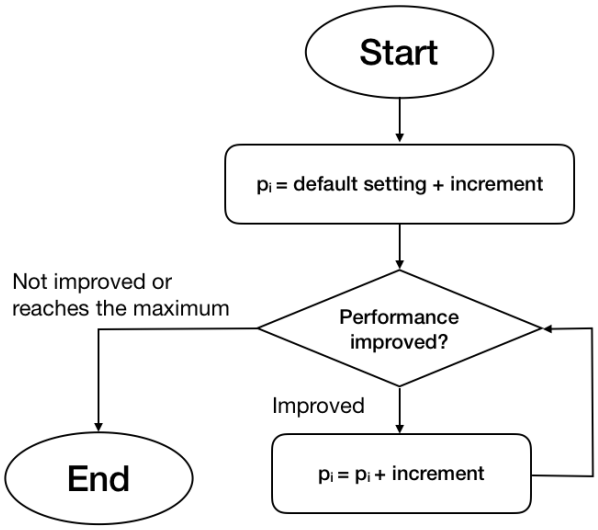

Figure 11 Detailed Transitions in Each Block

3.2 Workloads

In the thesis, we select workloads from SparkBench [23] to evaluate and analyze our tuning algorithm. These workloads are also characterized and classified into three types in the study [24]: memory-intensive, shuffle-intensive, and all-intensive. Section 3.2.1 describes the three workload types and their characteristics. We introduce SparkBench in section 3.2.2. In Section 3.2.3, we describe the machine learning algorithms used in the thesis, namely K-means, logistic regression, matrix factorization and random forest.

3.2.1 Workload Types

The second phase of our algorithm selects the parameters to tune for a job based on the type of the workload. We follow previous work on tuning [24] that classifies a workload based on the resource bottleneck for the workload. The three classes are

(i) Memory-intensive

Memory-intensive workloads consume more memory; therefore, lack of memory is the performance bottleneck. In Spark, RDDs are the main objects that consume memory. For example, in memory-intensive machine learning workloads such as logistic regression, memory consumption increases because the input dataset must be stored as a RDD during the execution of the tasks in the training phase.

(ii) Shuffle-intensive

Shuffle-intensive workloads are bottlenecked by shuffle operations which consume disk I/O and network I/O. Sometimes, a job incurs shuffle traffic because the cache is not large enough to hold all the content of the RDDs so Spark stores some of the data onto disk. For example, during the SQL query workloads in our experiments, a portion of the RDD data is written to disk and, as a result, large amounts of local shuffle writes and I/O traffic occurs.

(iii) All-intensive

The workload is bottlenecked by both memory and shuffle operations. This kind of workload not only consumes memory to store data as RDDs, but also involves lots of shuffle operations. For example, in our experiments the matrix factorization machine learning workload both consumes memory by storing the input data as an RDD and generates shuffle traffic during each stage in the training phase to shuffle data among the nodes.

3.2.2 SparkBench

SparkBench [23] is a Spark-specific benchmarking suite comprising a comprehensive and representative set of workloads including machine learning, graph processing, streaming and SQL query applications currently supported by Spark. SparkBench was first proposed in 2015 and recently went through an extensive rewrite [25]. The legacy version of SparkBench is not user-friendly because of limited documentation and user guide. The latest version, SparkBench 2.3, is not fully completed yet and has less features than the legacy version. Both SparkBench 2.1 and legacy SparkBench play important roles in this thesis. In our experiments, we select three machine learning workloads from SparkBench including K-means, logistic regression and matrix factorization. We also implement random forest workload on the industry dataset as an additional customized benchmark. Moreover, SparkBench provides a data generator for users to create input data of arbitrary size.

3.2.3 Machine Learning Workloads

We use three machine learning workloads from SparkBench in our experiments, namely K-means, logistic regression and matrix factorization. The random forest algorithm, which is commonly used in the Scotiabank DSA Lab, is used in our experiment with industry data.

a) K-means

K-means [26] is one of the most well-known and widely used unsupervised clustering algorithm. The goal of the K-means algorithm is to partition observations in a dataset into *k* clusters according to their similarity. It takes *k* centroids defined by users, one for each cluster. An observation is considered to be in a particular cluster if it is closer to that cluster's centroid than any of the other centroids. This kind of clustering is non-hierarchical,

so they won't overlap. However, it is difficult to define the most appropriate k value and compare the quality of the clusters produced.

b) Logistic Regression

Logistic regression [27] is a regression model which can be used to predict a categorical response. A logistic regression equation, which is shown below, is used to describe data and explain the relationship between variables. In this thesis, the logistic regression workload used is the binomial logistic regression from MLlib [18]. The output response is a binary outcome which only contains two variables, "0" and "1", representing outcomes such as yes or no. The output of logistic regression model has a probabilistic interpretation as shown below, such that, if f(z) > 0.5, the outcome is 1, or 0 otherwise.

$$\mathbf{f(z)} = \frac{\mathbf{1}}{\mathbf{1}+e^{-z}} \textbf{ where } \boldsymbol{z = w^T x}$$

c) Matrix Factorization

Matrix factorization (MF) [28] is a collaborative filtering technique commonly used in recommendation systems to predict user-product associations. It aims to fill in the missing entries of a user-product association matrix. For example, if there are n users (p) and m items (q), an n x m matrix r is used to represent the ratings where r_{ui} is the rating for item i by user u. Matrix r supposes to have many missing entries indicating the unknown ratings. Matrix factorization estimates the unknown ratings. In other words, the goal is to minimize the least squares error of the observed ratings represented as a cost function shown below. In Spark, the "ALS-WR" approach [29] is implemented to solve the cost function for matrix factorization model. Currently, matrix factorization model in Spark MLlib can be configured to use either explicit or implicit feedback from the users.

$$\min_{q^*,p^*} \sum_{(u,i)\in k} (r_{ui} - {q_i}^T p_u)^2 + \lambda(\| q_i \|^2 + \| p_u \|^2)$$

where k is the set of the (*u, i*) pairs for known r_{ui}.

d) Random Forest

Random Forests [30] is one of the most accurate and popular algorithms used for both classification and regression. It is an ensemble of decision trees. It constructs multiple decision trees at training time and takes the average across the trees to develop a prediction. The algorithm can handle thousands of input variables and work for variable selection by computing variable importance. However, the classifications made by random forests can be difficult for humans to interpret.

3.3 Selection of Configuration Parameters

There are over 200 tunable parameters hiding behind the Spark engine related to parallelism, fault tolerance, resource utilization, and so on. In QST, we focus on nine parameters that we selected based on their definitions [31] and on previous tuning studies [6], [7] and [8]. Table 1 summarizes 20 configurable parameters related to shuffle behavior, memory management, application properties and compression and codec from previous tuning studies [6, 7, 8]. In Table 1, each parameter is listed with its related previous studies and our notes indicating if the parameter is selected, removed or deprecated.

We select spark.serializer, spark.shuffle.file.buffer, and spark.io.compress.codec because all three studies claim that these three configuration parameters have a large impact. Spark.reducer.maxSizeInFlight is selected because previous studies [7] and [8] state they

also have a large influence on the overall performance. Spark.memory.fraction and spark.memory.storageFraction are selected to substitute for the deprecated parameter, storage.memoryFraction. Spark.executor.cores and spark.executor.memory are selected since memory and cores are the two main resources in Spark.

Table 1 Configuration Parameters Selected from Previous Studies

No.	Parameter	Previous Studies	Notes
1	spark.serializer	[6], [7], [8]	Selected
2	spark.shuffle.manager	[6], [7]	Removed since Spark 2.0
3	shuffle.memoryFraction	[6], [7]	Deprecated
4	storage.memoryFraction	[6], [7], [8]	Deprecated
5	spark.reducer.maxSizeInFlight	[7], [8]	Selected
6	spark.shuffle.file.buffer	[6], [7], [8]	Selected
7	spark.shuffle.compress	[6], [7]	
8	spark.io.compress.codec	[6], [7], [8]	Selected
9	spark.shuffle.consolidateFiles	[6], [7]	Removed since Spark 2.0
10	spark.rdd.compress	[7]	
11	spark.shuffle.io.preferDirectBufs	[7]	
12	spark.shuffle.spill.compress	[6], [7]	
13	spark.shuffle.sort.bypassMergeThreshold	[8]	
14	spark.storage.memoryMapThreshold	[8]	
15	spark.storage.unrollFraction	[8]	Deprecated
16	spark.io.compression.lz4.blockSize	[8]	
17	spark.kryoserializer.buffer	[8]	
18	spark.kryo.referenceTracking	[8]	
19	spark.executor.cores	[8]	Selected
20	spark.executor.memory	[8]	Selected

Table 2 lists the nine key configuration parameters we selected with simple descriptions and their default values.

Parameters 1 to 3 in the Table 2 are environment parameters which are used to control resource allocation. Memory and CPU are the two main resources allocated for Spark applications. Memory is definitely important since Spark is an in-memory data processing engine, which tries to persist data in memory as much as possible in order to avoid disk I/O. The number of cores determine the number of concurrent tasks that an individual executor can run. For example, if *spark.executor.cores*=2, each executor will run 2 tasks at most simultaneously. The *spark.executor.instances* parameter controls the number of slaves assigned for the application. These three environment parameters are tuned to improve the computation speed and parallelism for Spark applications.

Parameters 4 to 7 are workload specific parameters. For memory-intensive workloads, obviously, memory distribution is important. The value of *spark.memory.fraction* controls the amount memory reserved for storage and for execution. Therefore, the amount of memory available to each application on an individual executor is: (*spark.memory.fraction* * *spark.executor.memory*). The value of *spark.memory.storageFraction* is used to set the fraction of memory used for storage. Therefore, the amount of memory reserved for storage on an individual executor is: (*spark.memory.fraction* * *spark.executor.memory* * *spark.memory.storageFraction*). In a previous study, Li et al [24] observe that caching more data in memory improves the overall performance for memory-intensive workloads.

Table 2 Selected Configuration Parameters

No.	Parameter	Description	Default
1	*Spark.executor.memory*	Amount of RAM allocated on each executor	1 GB
2	*Spark.executor.cores*	Number of cores allocated on each executor	1
3	*Spark.executor.instances*	Number of workers assigned for the application	2
4	*Spark.memory.fraction*	Fraction of memory used for the application	0.6
5	*Spark.memory.StorageFraction*	Fraction of memory used for storage	0.5
6	*Spark.shuffle.file.buffer*	The maximum amount of shuffle buffer	32 KB
7	*Spark.reducer.maxSizeInFlight*	Maximum size of map outputs to fetch simultaneously from each reduce tasks	48 MB
8	*Spark.serializer*	The serializer used for serialize data	JavaSerializer
9	*Spark.io.compression.codec*	Codec used to compress internal data	Lz4

Shuffle-intensive workloads involve more shuffle operations. In order to speedup shuffling, *spark.shuffle.file.buffer* and *spark.reducer.maxSizeInFlight* are the two main parameters selected regarding the shuffle behaviour. Increasing *spark.shuffle.file.buffer* reduces the number of disk seeks and system calls made during the creation of intermediate shuffle

files. Since *spark.reducer.maxSizeInFlight* controls the maximum size of map outputs to fetch simultaneously, increasing its value leads to larger output chunks during shuffling. Although these two tuning parameters can improve the overall performance, the memory requirements may be aggravated.

The last two parameters in Table 2 control the serializer and codec used in a Spark application. Both data serialization and compression play important roles in Spark. Data serialization is the process of converting an in-memory object such as an RDD to another format in order to store or compute faster. Spark gives two serialization libraries, namely Java serializer and Kryo serializer. Kryo serializer is claimed by the Spark team to serialize objects much faster than Java Serializer in most cases since it has a smaller memory footprint when shuffling and caching large amount of data [33].

Spark currently provides three compression codecs, LZ4, LZF and snappy. In Spark applications, compression happens frequently. For instance, serialized RDDs, map output files, data spilled during shuffle, event log, and broadcast are designed to be compressed as default.

3.4 Algorithm

As shown in the figure 9, QST requires several initial inputs. These input variables are listed and described in Section 3.4.1. Then, in Section 3.4.2 to 3.4.4, we present the three phases of our algorithm.

3.4.1 Algorithm Input

QST takes the input parameters shown in Table 3. The type of workload can be determined using a monitoring system, for example Ganglia [32], to track memory, CPU and disk utilization and determine the resource bottleneck. If memory utilization is high and disk I/O is low, then the workload is memory-intensive. If disk utilization is high, then the workload is shuffle-intensive. If both memory and disk utilization are high, then the workload is all-intensive.

Table 3 Initial Input

Inputs	**Explanations**
Dataset Size	During tuning environment parameters, dataset size is an important factor to determine the value of RAM and the number of cores for the next step.
Type of workload	Tuning specific parameters based on the type of workload.
Total Available RAM	The maximum value of *spark.executor.memory*.
Total Available VCPUs	The maximum value of *spark.executor.cores*.
Total Available Number of Instances	The maximum value of *spark.executor.instances*.
Duration Time with Default Configurations	This duration time is recorded as a reference value to determine if the performance is improved.

3.4.2 Configuring Environment Parameters

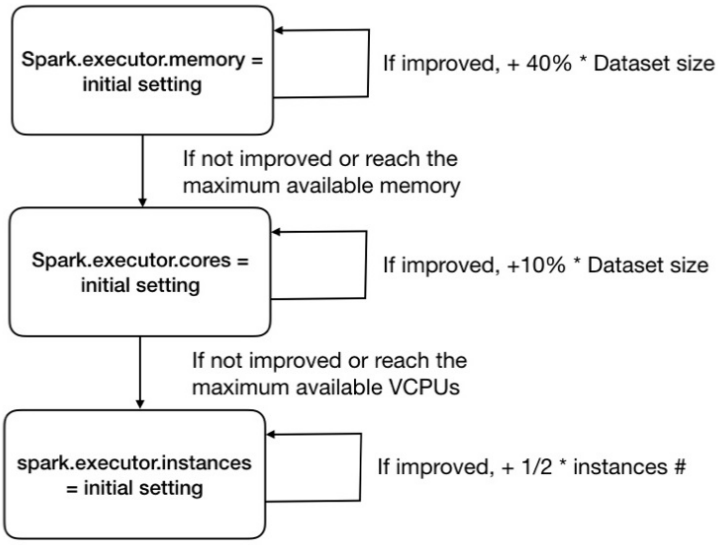

Figure 12 Configure Environment Parameters

The first phase of QST is shown in figure 12 which is extracted from figure 10. The main idea of this phase is to allocate available resources such as memory and cores per executor, and the number of workers. The following three parameters are used to control the resource allocation.

1. Spark.executor.memory

   Spark.executor.memory sets the amount of memory allocated to each executor process. The default value is 1 GB but this can be modified in the env.sh file according to the daily use. Our algorithm increases the value by (40% * dataset size) RAM each step. At first, we set the memory increase for each step to be a constant number, namely 2 GB, but in running our experiments, we found this involved too many iterations when tuning a large cluster if the initial value is small. For example, when the default value is 1 GB and 100 GB memory is available for allocation, it requires 50 steps to tune to reach the maximum memory if increasing 2 GB each step. Since this is not efficient, we decided to use a relative number

instead of an absolute number and set the increase to be 40% of the input dataset size every step.

2. Spark.executor.cores

   Spark.executor.cores is the parameter to set the number of cores allocated to each executor process. The default value set by Spark is 1. This default value can also be modified in the env.sh file according to the daily use. At first, we set this parameter to increase 2 cores for each step, but we found this involved too many iterations when the cluster is large. Therefore, we decided to use a relative number such that the QST increases the value by (10% * *d*) VCPU(s) every step where *d* represents the *d* GB input dataset size.

3. Spark.executor.instances

   Spark.executor.instances is the parameter to set the maximum number of executors assigned to the application. . The default value set by Spark is 2. This default value can also be modified in the env.sh file according to the daily use. This increment was set to 1 at first, but we found a relative number is more efficient than a constant number in a large cluster. So that, the algorithm increases the value by (1/4 * total number of instances) VCPU(s) every step.

The goal of this phase is to assign more resources to worker instances before adding more worker instances since we found that, given a set amount of total recourses, a smaller of number of large executors outperforms a larger number of smaller executors. We

implemented multiple experiments to verify this statement. Table 4 shows the result of one of the experiments that applied a K-means workload on a 5 GB dataset with three different configurations. The total memory (24 GB RAM) and cores (24 VCPUs) allocated for the three applications are the same, but obviously, the first configuration having three larger executors has the best performance among them. Therefore, QST starts with configuring size of worker instances and follows by increasing the number of instances.

Table 4 Experiment that applied a K-means workload on a 5G dataset

	spark.executor.memory (g)	spark.executor.cores	Number of Executors	Duration Time
1	8	8	3	4.3
2	4	4	6	5.9
3	2	2	12	8.6

3.4.3 Configuring Specific Parameters

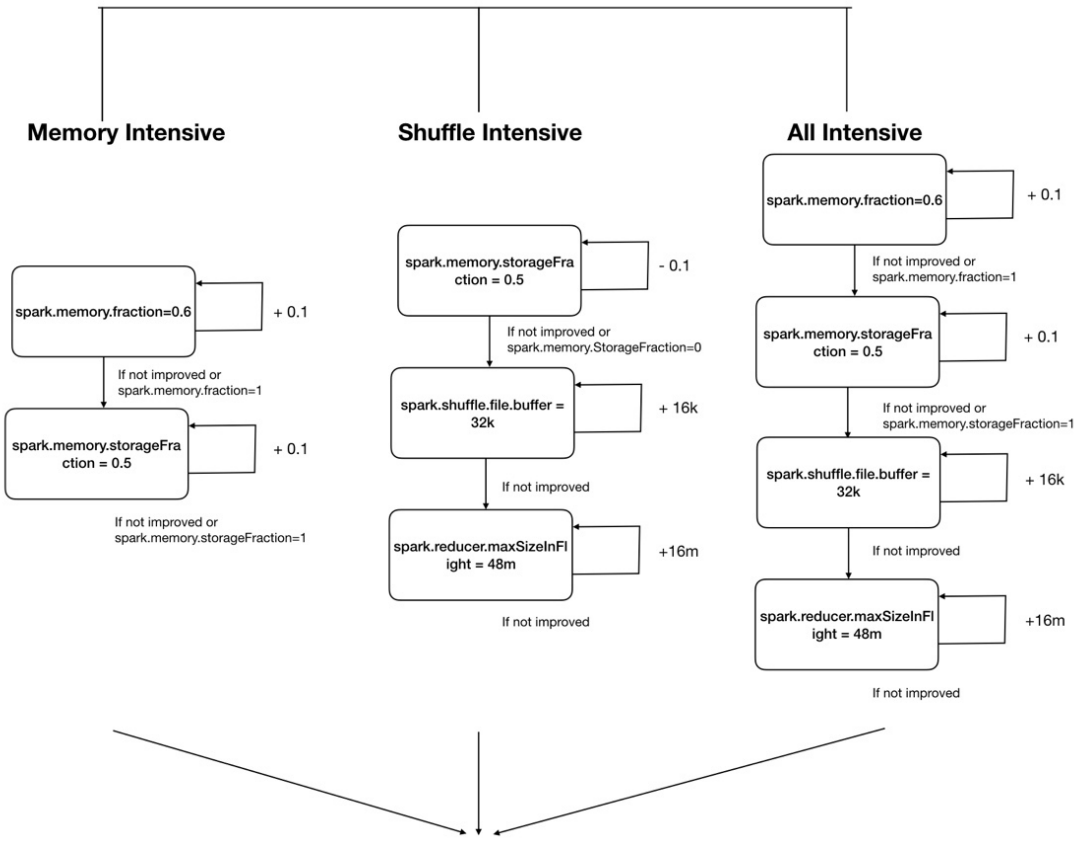

Figure 13 Configure Specific Parameters

The second phase of QST is shown in figure 13 which is extracted from figure 10. This part of the algorithm tunes different sets of parameters depending on the type of the workload.

For memory-intensive workloads, the algorithm adjusts the values of *spark.memory.fraction* and *spark.memory.storageFraction*. Previous studies [24] showed that memory-intensive workloads perform better if more intermediate data are cached into memory. *Spark.memory.fraction* determines the fraction of execution and storage for a Spark application. The default value is 0.6 and our algorithm increases the fraction by 0.1

every step. *Spark.memory.StorageFraction* sets the fraction of memory for storage. The default value is 0.5. The goal is to cache as much of an RDD as possible, so the algorithm increases the fraction by 0.1 every step.

For shuffle-intensive workloads, the algorithm increases the values of *spark.shuffle.file.buffer* and *spark.reducer.maxSizeInFlight* and decreases the value of *spark.memory.storageFraction*. Previous studies [24] showed that the performance of shuffle-intensive workloads is not improved by caching more intermediate data into memory. Therefore, decreasing memory storage fraction and having more memory for execution is better for the overall performance. *Spark.shuffle.file.buffer* is the parameter that controls the maximum size of buffers during shuffling. Buffers reduce the number of disk seeks when creating intermediate shuffle files, therefore, increasing the buffers to their maximum size improves overall performance. The default value is 32k. The algorithm suggests to increase the value by 16k every step. *Spark.reducer.maxSizeInFlight* is the parameter that limits the number of fetch requests. Increasing the value means reducers request bigger output chunks during shuffling in order to improve overall performance. The default value is 48MB, the algorithm suggests to increase the value by 16MB every step.

The all-intensive workloads are intensive with respect to both memory and shuffle, therefore, the algorithm combines the strategies from memory-intensive and shuffle-intensive types and suggests to increase the values of *spark.memory.fraction*,

spark.memory.storageFraction, *spark.shuffle.file.buffer* and *spark.reducer.maxSizeInFlight*.

3.4.4 Configuring Serializers and Compression Codec

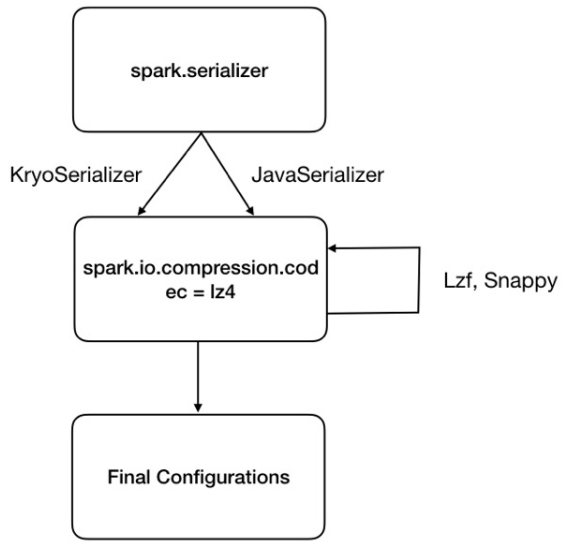

Figure 14 Configure Serializers and Compression Codecs

The third phase of QST is shown in figure 14 which is extracted from figure 10. Serialization is essential in distributed environments [31]. If serializing objects into the required formats are slow, then the whole computation is slowed down. Spark provides two serialization libraries, Java serialization (default) and Kryo serialization [31]. Generally, Kryo serialization is faster and more compact than Java but it does not support all Serializable types. Since the best choice of serializer is application dependent, the algorithm first chooses Kyro and if performance is not improved then it tries Java. Currently, Spark 2.x provides 3 compression algorithms, namely Lz4 (default), Lzf, and Snappy [31]. As a flexible open-sourced engine, Spark also supports customized compression algorithms written by users. No compression is also another option, but the

performance doesn't improve for all the experiment cases. Therefore, in this case, only the three built-in compression codecs are considered. The algorithm suggests to compare all three codecs and choose the best one since the choice of compression algorithm is application dependent.

Chapter 4

Evaluation

In this chapter, we present the results of an experimental evaluation of the effectiveness of QST with both benchmark and industrial workloads. Section 4.1 presents the results of experiments using benchmark workloads implemented on our lab's Openstack cloud. Section 4.2 presents the results of experiments with an industry workload run at the Scotiabank DSA Lab.

In both sets of experiments, we manually tune configuration parameters following our algorithm step by step. At first, we run each of the workloads with default settings and collect the run duration times to serve as a reference value for further comparison. Then, we start by configuring the parameter in the first box of our algorithm and propagate to the next box in the sequence. Each run represents one step. After each run, the run duration time is collected and used to decide if we keep configuring the current parameter or move on to the next configuration parameter according to the condition indicated in our algorithm.

4.1 Benchmark Workloads

In this section, we show the results of experiments using our algorithm to tune a workload involving jobs from a standard benchmark run on a cluster in our lab. The settings for the experimental environment are described in Section 4.1.1. Four workloads are selected from SparkBench. Since data generators are provided for each benchmark in SparkBench, we use generated datasets for the experiments in this section. The four selected workloads and

input datasets are described in Section 4.1.2. In Section 4.1.3, we present the experimental results. Section 4.1.4 discusses the importance of the different configuration parameters according to our results.

4.1.1 Environment

Our experiment is conducted on a 4-node cluster hosted on our lab's Openstack cloud. Each node has 16GM RAM, 80GB disk, and 8 virtual CPUs which are summarized in Table 5. Hadoop and Spark components are built through Apache Ambari [34]. Therefore, Spark is built as a data processing engine on the top of Hadoop YARN and HDFS. One node is configured to be a master node and the other nodes work as slaves for both Spark and Hadoop components. The Ambari built-in dashboard provides real-time resource utilization monitoring but it can only track the resource usage for all the nodes, Ganglia [32] is also installed as a resource monitor to track the resourse utilization for each node.

Table 5 Cluster Configurations

Instance Role	RAM	Disk	CPUs	Number of Instance
Master	16 GB	80 GB	8	1
Worker	16 GB	80 GB	8	3

SparkBench [24] is an open-source Spark benchmarking suite. Currently, there are two versions, SparkBench 2.1 and a legacy version. The legacy SparkBench is more comprehensive, but it is difficult to use because of limited documentation. SparkBench 2.1 is more user-friendly and has more detailed documentation, but it is not yet fully completed. In our experiments, both SparkBench 2.1 and legacy SparkBench play important roles.

4.1.2 Datasets and Workloads

In this set of experiments, 1 or 2 workloads, as shown in Table 6, are selected from SparkBench 2.1 and the legacy version for each of the three types of workloads considered by our algorithm. For the memory-intensive type, logistic regression and K-means are selected. Logistic regression is determined to be a memory-intensive workload in a previous study [24]. K-means is determined to be a memory-intensive workload because we observed in experiments that applying k-means on a 10 GB dataset consumes around 15 GB memory and requires around 1.7 GB shuffle read and write. In other words, it requires more memory for the same data as those workloads classified into memory-intensive type in the previous study [24]. SQL selection is selected for shuffle-intensive type and matrix factorization is selected for all-intensive type [24].

Table 6 Workloads and Datasets

Workload	Type of workload	Dataset Size	SparkBench Version
Logistic Regression	Memory-intensive	3 GB	Legacy
K-means	Memory-intensive	10 GB	2.1
SQL (Selection)	Shuffle-intensive	20 GB	2.1
Matrix Factorization	All-intensive	3.6 GB	Legacy

All the datasets used are created by the data generator provided by SparkBench. A 3 GB file is generated for logistic regression workload which has 5 features and 5,000,000 rows. A 10 GB file is generated for K-means workload which has 12 features and 1,000,000

rows. A 3.6 GB file is generated for matrix factorization workload which has 5 features and 1,000,000 rows. SQL selection is applied on a 20 GB file.

4.1.3 Analysis and Results

In this case, the default setting of *spark.executor.instances* is set to 3, which means the max number of executors assigned to the application is 3 as shown in Table 7. Details of all the runs and the duration time for each run are recorded and attached in Appendix A.

Table 7 Default Setting for Benchmark Workloads

No.	Parameter	Default
1	Spark.executor.memory	1 GB
2	Spark.executor.cores	1
3	Spark.executor.instances	3
4	Spark.memory.fraction	0.6
5	Spark.memory.StorageFraction	0.5
6	Spark.shuffle.file.buffer	32 KB
7	Spark.reducer.maxSizeInFlight	48 MB
8	Spark.serializer	JavaSerializer
9	Spark.io.compression.codec	LZ4

The main results for the four workloads are summarized in Figure 15. Bars in the chart indicates each phase. Y axis represents the improvement percentage which is calculated by the following function: $(d_{i-1} - d_i) / d_0)$ where d_{i-1} is the duration time collected in the previous phase, d_i is the best duration time collected in the phase i, and d_0 is the duration

time collected with default setting at first. For example, to calculate the improvement percentage for phase 1 is $(d_0 - d_1) / d_0$ since it is the first phase, d_0 is the default duration time. Moreover, we illustrate the number of runs for each workload grouped by phase in figure 16.

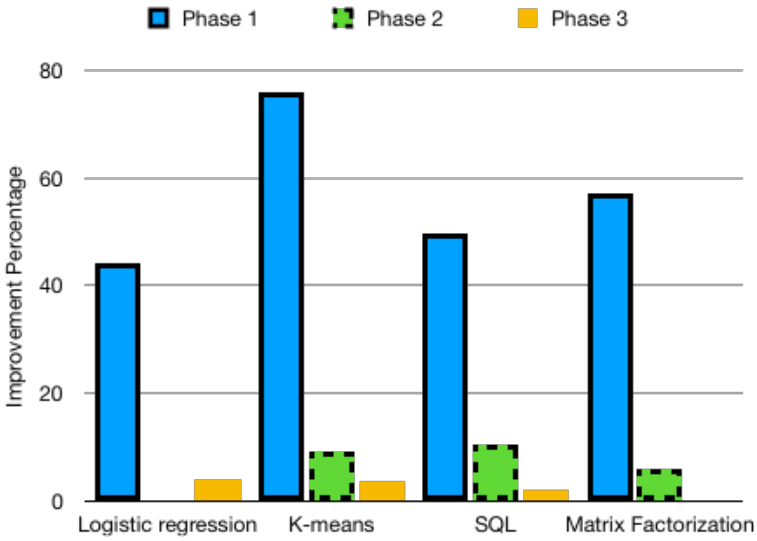

Figure 15 Improvement Percentage for Each Workload

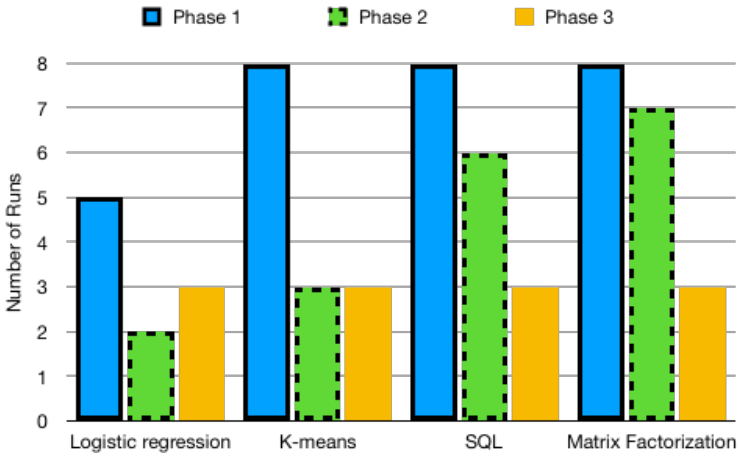

Figure 16 Number of Runs for Each Workload

By running our algorithm, the best configuration of logistic regression workload is [*spark.executor.memory = 4 GB; spark.executor.core = 2; spark.serializer =*

org.apache.spark.serializer.KryoSerializer]. Other parameters remain the default value. The duration time is reduced from 5 min to 2.6 min. This case has a small dataset and requires a small amount of resources to run. Therefore, configuring the memory fraction does not provide benefit in this case. The overall performance has 48% speedup including 44% speedup by configuring environment configurations and 4% speedup by configuring the serializer and codec. The tuning process takes 5 runs in phase 1, 2 runs in phase 2, and 3 runs in phase 3.

K-means' best configuration from our algorithm is [*spark.executor.memory = 8 GB; spark.executor.core = 8; spark.memory.fraction = 0.7; spark.serializer = KryoSerializer*]. In this case, the input dataset size is much larger, so the resources required increase. The overall performance has 88% speedup including 76% speedup by tuning environment configurations, 9.2% speedup by tuning specific parameters, and 3.7% speedup by tuning the serializer and codec. The tuning process takes 8 runs in phase 1, 3 runs in phase 2, and 3 runs in phase 3.

In the SQL case, a selection query is used to retrieve data from a 20G file where the value in the second column is greater than 0.5. The best configuration for SQL is [*spark.executor.memory = 8 GB; spark.executor.core = 8; spark.memory.storageFraction = 0.4; spark.shuffle.file.buffer = 32 KB; spark.reducer.maxSizeInFlight = 64 MB; spark.io.compression.codec = snappy*]. The overall performance has 62% speedup including 49.7% speedup by tuning environment configurations, 10.4% speedup by tuning

specific parameters, and 2% speedup by tuning the serializer and codec. The tuning process takes 8 runs in phase 1, 6 runs in phase 2, and 3 runs in phase 3.

Through the tuning process, the overall performance of matrix factorization has a 63% speedup including 57% speedup by tuning environment configurations, 6% speedup by tuning specific parameters, and 0% speedup by tuning serializer and codec. The best configuration is [*spark.executor.memory = 7 GB; spark.executor.core = 8; spark.memory.fraction = 0.7; spark.shuffle.file.buffer = 32 KB; spark.reducer.maxSizeInFlight = 64 MB; spark.io.compression.codec = LZF*]. The tuning process takes 8 runs in phase 1, 7 runs in phase 2, and 3 runs in phase 3.

From figure 15, we can observe that, in general, first phase improves the performance much more than other phases. Bar chart in figure 16 shows the total number of runs taken place in each phase. Phase 1 needs more runs since the increment number is designed to be an absolute number at first, for instance, the configuration parameter *spark.executor.memory* increases 2 GB and *spark.executor.core* increases 2 every step.

4.1.4 Influence of the Tunable Parameters

From Figure 17, obviously, the environment parameters are observed to be the most influential. Configuring *spark.executor.memory* and *spark.executor.cores* can improve the overall performance by 35.6% and 14% in average respectively. At most, Spark can be speed up by 67% and 22% just by configuring the two environment parameters. Memory and CPU are two main resources in Spark. Since Spark is an in-memory data processing

engine, it tries to cache data into memory as much as possible. The memory parameter influences the amount of data that Spark cache and the maximum sizes of the shuffle data structures. Allocating more memory can avoid disk I/O and speedup the all type of workloads. The number of cores determines the number of concurrent tasks that an individual executor can run. Therefore, increasing the number of cores can improve the speed of computation.

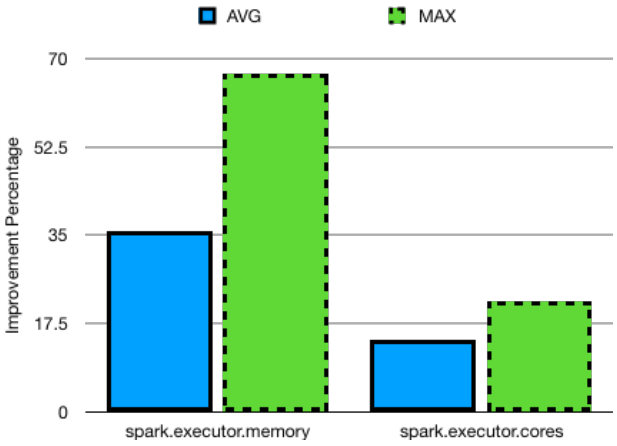

Figure 17 Average and Max Speedup Achieved by Environment Configuration Parameters

Although the specific parameters are not as influential as the environment parameters, they can also impact the performance. As shown in Figure 18, configuring parameters *spark.memory.fraction* and *spark.memory.storageFraction* can improve the overall performance by 3.2% and 2.6% in average, and by 5.9% and 7.7% at most. The value of *spark.memory.fraction* controls the amount memory reserved for storage and for execution and the value of *spark.memory.storageFraction* is used to set the fraction of memory used for storage. For memory intensive-workloads, QST tries to allocate more memory for storage and find the balance point between storage and computation. Configuring

spark.reducer.maxSizeInFlight and *spark.shuffle.file.buffer* can improve the overall performance by 5.9% and 7.7% in average, and by 7.7% and 7.7% at most. Increasing *spark.shuffle.file.buffer* and *spark.reducer.maxSizeInFlight* improves the overall performance because *spark.shuffle.file.buffer* impacts the number of disk seeks and system calls made during the creation of intermediate shuffle files and *spark.reducer.maxSizeInFlight* impacts the maximum size of map outputs to fetch simultaneously.

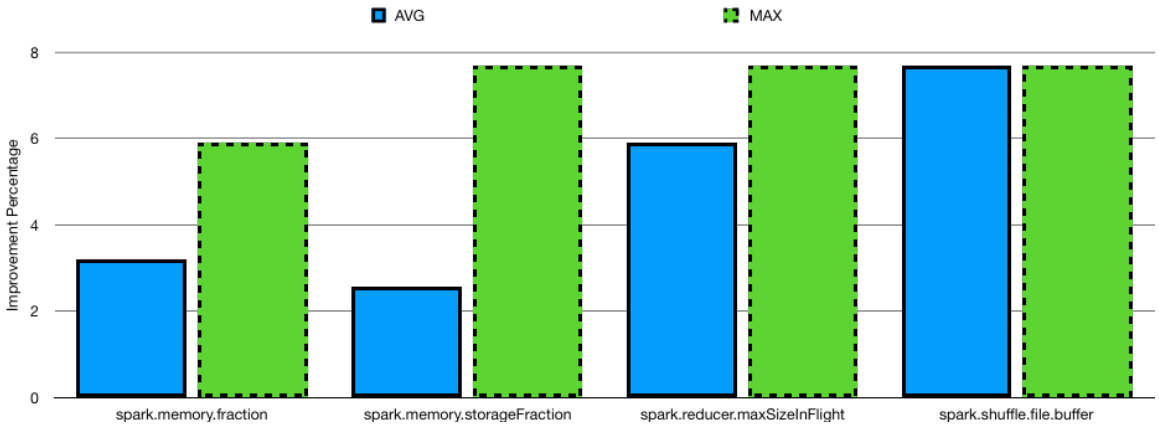

Figure 18 Average and Max Speedup Achieved by Specific Configuration Parameters

Serializer and compression codec have less impact than the other parameters. The average and maximum speedup percentages are illustrated in Figure 19. Configuring serializer and codec can improve the overall performance by an average of 3.3% and 0.7%, respectively, and by 4% and 2% at most, respectively. Data serialization happens during converting an in-memory object such as an RDD to another format in order to store or compute faster. Kryo serializer performs faster than default Java serializer in most cases. Since serialization is just a small part of whole computation and costs very little time, the difference of

duration time is not really obvious. Compression happens during compressing serialized RDDs, map output files, event log and so on. The default compression codec, LZ4, works better in most cases.

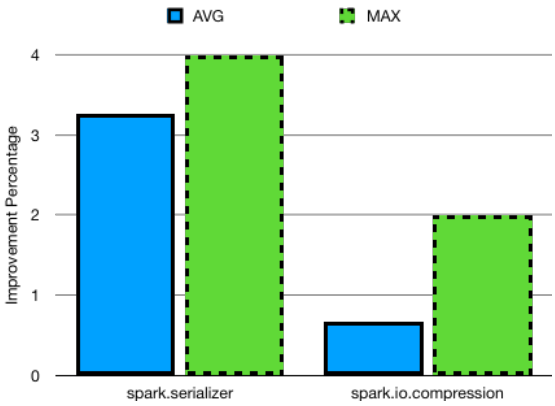

Figure 19 Average and Max Speedup Influenced by Serializer and Compression codec

4.2 Industry Workloads

We also applied the tuning algorithm to a real-world workloads and datasets on the server at Scotiabank's DSA Lab. As part of the Global Banking and Markets group of Scotiabank, DSA (Data Science and Analytics) aims at gaining insights into clients and trading activity by bringing information together from across the organization. Section 4.2.1 describes the experimental environment. Section 4.2.2 describes the workloads describes the three datasets used in the experiments. Section 4.2.3 presents the results of applying QST.

4.2.1 Environment

The Scotiabank's cluster, which is much larger than then the one from our lab in the first set of experiments, is also configured using Apache Ambari. The whole environment is built based on the Hadoop ecosystem. One instance is configured as the master node and the rest six nodes are worker nodes. Seven of the slave instances have Spark slave

components installed. Each instance has 50 cores and 251.5 GB RAM. Since the DSA Lab has production jobs that run continuously every day and night and that consume around 82% of memory and 50% of the CPUs in total, our experiments were limited to a maximum of 20 cores and 32 GB RAM per executor.

4.2.2 Datasets and Workloads

We select logistic regression, k-means and random forest for the memory-intensive type, and SQL query of joining a csv file with a Hive table for the shuffle-intensive. We included random forest since it is one of the most frequently used models in the DSA lab. Random forests creates ensembles of decision trees. All the decision trees are supposed to be trained and stored in memory [35]. Since it consumes more memory, memory issues regarding a random forest implementation are posted on the Spark issue forum, for example issue-3728 [36] and issue-13434 [37]. Therefore, we classify random forest as a memory-intensive workload. There were no suitable examples of all-intensive workloads at the DSA lab.

Table 8 Industry Workloads and Input Dataset

Workloads	Type of workload	Dataset
Logistic Regression	Memory-intensive	Pics
K-means	Memory-intensive	Bess
Random Forest	Memory-intensive	Pics
SQL (join)	Shuffle-intensive	Pics and EFT

The following three datasets are used in this section. Table 8 summarizes the industry workloads and input datasets used for each of them.

1. Bess

The Bess dataset is around 50 GB stored in Hive which collects wired transactions made by Scotiabank customers all over the world. It includes more than 100 million records and more than 300 features

2. EFT

   The EFT dataset is around 373 GB which collects all electronic funds transactions related to Scotiabank, including more than 3 billion rows and hundreds of features

3. Pics

   The Pics dataset is around 97 GB stored in Hive which contains historical electronic funds transactions, including more than 1 billion records and around 90 features

4.2.3 Results and Analysis

Table 9 Default Setting for Industry workloads

No.	Parameter	Default
1	Spark.executor.memory	2 GB
2	Spark.executor.cores	2
3	Spark.executor.instances	2
4	Spark.memory.fraction	0.6
5	Spark.memory.StorageFraction	0.5
6	Spark.shuffle.file.buffer	32 KB
7	Spark.reducer.maxSizeInFlight	48 MB
8	Spark.serializer	JavaSerializer
9	Spark.io.compression.codec	Lz4

DSA lab changed their default setting to [*spark.executor.cores = 2 , spark.executor.memory = 2 GB*]. Other parameters' default values are remained the same as shown in Table 9. Details of all the runs are recorded and attached in the Appendix B.

The main results of four workloads are summarized in Figure 20. Each bar in the chart indicates the improvement percentage for each phase over the run duration with the default settings. We illustrate the number of runs for each workload grouped by phase in figure 21.

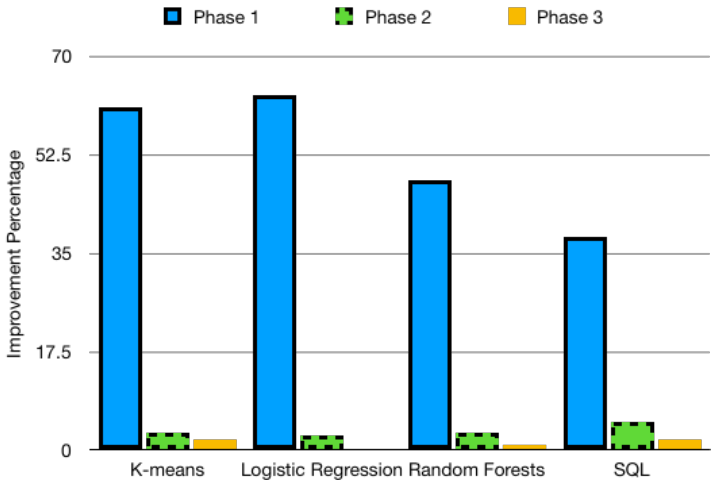

Figure 20 Improvement Percentage

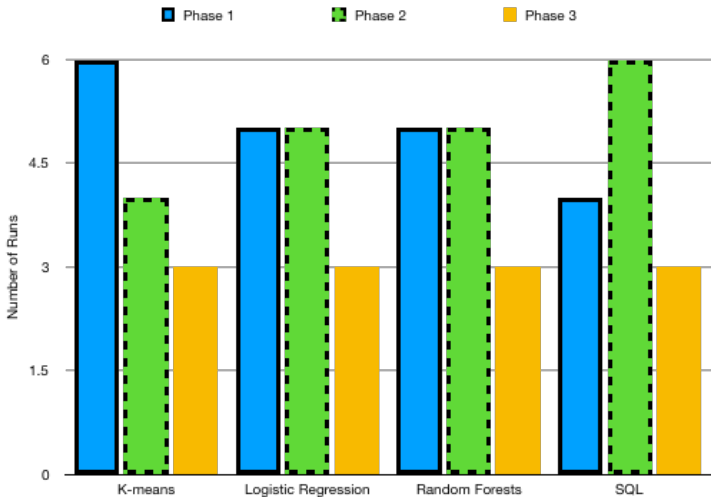

Figure 21 Total Runs During Each Phase

By applying QST, the duration time of the K-means workload was reduced from 54 min to 18 min within 13 runs. The best configuration to K-means is [*spark.executor.memroy=32 GB; spark.executor.cores=12; spark.executor.instances= 7; spark.memory.fraction=0.7; spark.memory.stroageFraction=0.6; spark.serializer=Kryo*]. The overall performance has 66% speedup.

Logistic regression runs out of memory with the default settings, so we allocate 8G RAM and 4 cores to each executor. The duration time collected is 74 min which is recorded as the reference value. The best configuration for logistic regression is [*spark.executor.memroy=32 GB; spark.executor.cores=20; spark.executor.instances= 7; spark.memory.fraction=0.7; spark.memory.stroageFraction=0.7*]. The overall performance has 65% speedup. The tuning process takes 5 runs in phase 1, 5 runs in phase 2, and 3 runs in phase 3.

As with the logistic regression workload, the random forest workload ends with an 'out-of-memory' error with default setting. We also allocate 8G RAM and 4 cores to each executor and record duration time, which is 1hr42min, as the reference value. After tuning, the best configuration to random forest is [*spark.executor.memroy=32 GB; spark.executor.cores=20; spark.executor.instances= 7; spark.memory.fraction=0.8; spark.memory.stroageFraction=0.7, spark.io.compression.codec=LZF*]. The overall performance improves 53%. The tuning process takes 5 runs in phase 1, 5 runs in phase 2, and 3 runs in phase 3.

For the SQL workload, we join two datasets and the larger one is selected to be the input size in our calculations. For example, in this case, pics is larger having 97G, therefore, at the first step, the increment is (40% * input dataset size) = 40% * 97G =38.8G, whereas, the maximum available memory is 32G so *spark.executor.memory* is tuned to 32G. It takes 36 minutes to run the SQL workload with the default settings. The duration time is reduced to 19 minutes 48 seconds after tuning. The best configuration to join is [*spark.executor.memroy = 32 GB; spark.executor.cores = 20; spark.executor.instances = 7; spark.shuffle.buffer = 48 KB; spark.reducer.maxSizeInFlight = 64 MB; spark.serializer = Kryo*]. The overall performance improves 47%. The tuning process takes 4 runs in phase 1, 6 runs in phase 2, and 3 runs in phase 3.

Chapter 5

Conclusion

In this work, we present our Spark tuning methodology, QST, to guide users tuning Spark to achieve better performance for different workloads and evaluate it with benchmark workloads and industry workloads. This thesis starts with the brief introduction in Chapter 1, in addition to motivation, contributions and outline.

Chapter 2 provides background information of Hadoop and Spark, and their comparison. As an in-memory data processing engine, Spark is able to perform up to 100 times faster than Hadoop on some types of workloads. We also survey existed tuning methodologies and compare them with QST in Chapter 2. Most existed tuning methodologies are not general tuning approaches. Our methodology is generally applicable. After taking several initial inputs and identifying the type of the workload, QST iteratively guides the user in configuring until performance can no longer be improved.

Chapter 3 discusses the design of QST and the selection of appropriate configuration parameters based on an application's properties relating to shuffle behavior, compression and serialization, and memory management. QST includes 9 configurable parameters selected among more than 200 parameters by understanding the meanings of all the parameters and summarizing ideas from previous studies. After taking several initial input variables and identifying the type of workload, QST iteratively guides users to tune Spark. Chapter 4 evaluates QST and measure the average speedup percentage, as well as the importance of each configuration parameter. The experimental evaluation shows that our

algorithm brings significant improvement to the overall performance. We measure the average speedup percentage and the importance of each parameter. QST significantly improves Spark performance for the workloads studied. Environment configurations are the most influential, which can improve performance by 49% in average. Configuring specific parameters based on three types of workloads can reduce 3.2% to 7.7% duration time. Choosing the best serializer and compression codec can also speed up the application by around 4%.

5.1 Contributions

We make three contributions in this thesis.

The first contribution is to identify 9 configurable parameters among over 200 parameters. We select them for use in QST by understanding the meanings of all the parameters and summarizing ideas from previous studies.

The second contribution is to propose QST which is a general greedy iterative tuning algorithm for our set of 9 key parameters. We classify Spark workloads as memory-intensive, shuffle-intensive or all-intensive. QST tunes Spark for each type of workload.

The third contribution is to provide an experimental evaluation of QST and measure the speedup percentage achieved for a range of workloads. Overall, using QST yields an average speedup of 65% for our benchmark evaluation workloads and 57% for our industry evaluation workloads.

5.2 Limitations

QST has several limitations.

1. Number of runs are still quite many. Tuning most of our experimental workloads cost over 10 runs. For large-scaled dataset, it costs over 300 minutes to tune the performance.
2. We implement it manually in this thesis. At first, we write a simple bash script to implement the QST. In the script, we get the current time at the beginning, then run the Spark-submit command with certain configuration parameters, and calculate the time difference at the end. However, we find the time difference calculated by the script is a little higher than the amount shown in the Spark history UI maybe due to transmission time.
3. QST only include 9 configuration parameters regarding to application properties, memory management, shuffle behavior, compression and serialization.

5.3 Future Work

To address the limitations discussed in Section 5.2, we listed some possible future work. Since QST is a greedy iterative tuning approach, we found that it takes many steps and consumes lots of time especially when the workload is heavy. Especially, in this thesis, since we implement QST manually, we cost plenty of time to wait for the running application and compare the duration time. In the future, it is possible to write a Bash script to tune Spark automatically by finding a way to get the duration time from Spark API directly. Currently, we classify the workload into only 3 types. In the future, we can use machine learning techniques to cluster workloads into more precise types in order to start at some point of the algorithm instead of starting from the very first step every time.

Combining few steps can reduce the overall tuning time and achieve faster convergence. Another extension of this work can be comparing QST with versions used other search paradigms like simulated annealing. QST follows the algorithm step by step and only accepts the local best configuration for each parameter, the final configuration may not be the overall best configuration. Unlike QST modifies the configuration parameter with the increment and stops once the performance degrades, simulated annealing starts at a random point, searches for a better configuration and probably accepts a configuration which is not the best locally but leads an overall best configuration. Moreover, configuring more parameters may improve the performance even more. Currently, we only include configuration parameters belonging to the following categories: application properties, memory management, shuffle behavior, compression and serialization. In the future, it is possible to add more configuration parameters into QST from other categories such as dynamic allocation, scheduling and networking.

Bibliography

[1]. Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010). Spark: Cluster computing with working sets. *HotCloud*, *10*(10-10), 95.

[2]. Apache hadoop. http://hadoop.apache.org. (retrieved July 10, 2018)

[3]. Dittrich, J., & Quiané-Ruiz, J. A. (2012). Efficient big data processing in Hadoop MapReduce. Proceedings of the VLDB Endowment, 5(12), 2014-2015.

[4]. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S. and Stoica, I., 2012, April. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (pp. 2-2). USENIX Association.

[5]. Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., ... & Ghodsi, A. (2016). Apache spark: a unified engine for big data processing. *Communications of the ACM*, *59*(11), 56-65.

[6]. Petridis, P., Gounaris, A., & Torres, J. (2016, October). Spark parameter tuning via trial-and-error. In *INNS Conference on Big Data* (pp. 226-237). Springer, Cham.

[7]. Gounaris, A., & Torres, J. (2017). A Methodology for Spark Parameter Tuning. *Big Data Research*.

[8]. Bei, Z., Yu, Z., Luo, N., Jiang, C., Xu, C., & Feng, S. (2018). Configuring in-memory cluster computing using random forest. *Future Generation Computer Systems*, *79*, 1-15.

[9]. Chandarana, P., & Vijayalakshmi, M. (2014, April). Big data analytics frameworks. In *Circuits, Systems, Communication and Information Technology Applications (CSCITA), 2014 international conference on* (pp. 430-434). IEEE.

[10]. Kambatla, K., Kollias, G., Kumar, V., & Grama, A. (2014). Trends in big data analytics. *Journal of Parallel and Distributed Computing*, *74*(7), 2561-2573.

[11]. Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, *51*(1), 107-113.

[12]. Shvachko, K., Kuang, H., Radia, S., & Chansler, R. (2010, May). The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on* (pp. 1-10). Ieee.

[13]. Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., ... & Saha, B. (2013, October). Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing* (p. 5). ACM.

[14]. Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. H., ... & Stoica, I. (2011, March). Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI* (Vol. 11, No. 2011, pp. 22-22).

[15]. Apache Spark FAQ. https://spark.apache.org/faq.html. (retrieved July 10, 2018)

[16]. Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., ... & Zaharia, M. (2015, May). Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*(pp. 1383-1394). ACM.

[17]. Venkataraman, S., Yang, Z., Liu, D., Liang, E., Falaki, H., Meng, X., ... & Zaharia, M. (2016, June). Sparkr: Scaling r programs with spark. In *Proceedings of the 2016 International Conference on Management of Data* (pp. 1099-1104). ACM.

[18]. Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., ... & Xin, D. (2016). Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, *17*(1), 1235-1241.

[19]. Xin, R. S., Gonzalez, J. E., Franklin, M. J., & Stoica, I. (2013, June). Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems* (p. 2). ACM.

[20]. Zaharia, M., Das, T., Li, H., Shenker, S., & Stoica, I. (2012). Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. *HotCloud*, *12*, 10-10.

[21]. Xin, R., Deyhim, P., Ghodsi, A., Meng, X., & Zaharia, M. (2014). Graysort on apache spark by databricks. *GraySort Competition*.

[22]. Wang, K., & Khan, M. M. H. (2015, August). Performance prediction for apache spark platform. In *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on* (pp. 166-173). IEEE.

[23]. Li, M., Tan, J., Wang, Y., Zhang, L., & Salapura, V. (2015, May). Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*(p. 53). ACM.

[24]. Li, M., Tan, J., Wang, Y., Zhang, L., & Salapura, V. (2017). SparkBench: a spark benchmarking suite characterizing large-scale in-memory data analytics. *Cluster Computing*, *20*(3), 2575-2589.

[25]. SparkBench. https://github.com/CODAIT/spark-bench. (retrieved July 10, 2018)

[26]. Zhao, W., Ma, H., & He, Q. (2009, December). Parallel k-means clustering based on mapreduce. In *IEEE International Conference on Cloud Computing* (pp. 674-679). Springer, Berlin, Heidelberg.

[27]. James, G., Witten, D., Hastie, T., Tibshirani, R.: An Introduction to Statistical Learning. Springer, New York (2013)

[28]. Koren, Y., Bell, R., & Volinsky, C. (2009). Matrix factorization techniques for recommender systems. *Computer*, *42*(8).

[29]. Zhou, Y., Wilkinson, D., Schreiber, R., & Pan, R. (2008, June). Large-scale parallel collaborative filtering for the netflix prize. In *International Conference on Algorithmic Applications in Management* (pp. 337-348). Springer, Berlin, Heidelberg.

[30]. Breiman, L. (2001). Random forests. *Machine learning*, *45*(1), 5-32.

[31]. Spark Configuration. https://spark.apache.org/docs/latest/configuration.html. (retrieved July 10, 2018)

[32]. Sacerdoti, F. D., Katz, M. J., Massie, M. L., & Culler, D. E. (2003, December). Wide area cluster monitoring with ganglia. In *null* (p. 289). IEEE.

[33]. Tuning Spark. https://spark.apache.org/docs/latest/tuning.html. (retrieved July 10, 2018)

[34]. Apache Ambari. https://ambari.apache.org/. (retrieved July 10, 2018)

[35]. Ensembles - RDD-based API. https://spark.apache.org/docs/2.2.0/mllib-ensembles.html#random-forests. (retrieved July 10, 2018)

[36]. RandomForest: Learn models too large to store in memory. https://issues.apache.org/jira/browse/SPARK-3728. (retrieved July 10, 2018)

[37]. Reduce Spark RandomForest memory footprint. https://issues.apache.org/jira/browse/SPARK-13434?attachmentOrder=asc. (retrieved July 10, 2018)

Appendix A

Benchmark Workloads

1. Logistic Regression

Step	Conf	Duration (min)
	Default	5
1	spark.executor.memory=2 GB	4
2	spark.executor.memory=4 GB	3.9
3	spark.executor.memory=6 GB	4
4	spark.executor.core=2	2.8
5	spark.executor.core=4	2.9
6	spark.memory.fraction=0.7	2.8
7	spark.memory.storageFraction=0.6	2.8
8	spark.serializer=org.apache.spark.serializer.KryoSerializer	2.6
9	spark.io.compression.codec =lzf	2.8
10	spark.io.compression.codec = snappy	2.7

2. K-means

Step	Conf	Duration (min)
	Default	54
1	spark.executor.memory=3 GB	42
2	spark.executor.memory=5 GB	27
3	spark.executor.memory=7 GB	20
4	spark.executor.memory=8 GB	18
5	spark.executor.core=3	16
6	spark.executor.core=5	13
7	spark.executor.core=7	12
8	spark.executor.core=8	10
9	spark.memory.fraction=0.7	8.2
10	spark.memory.fraction=0.8	9

11	spark.memory.storageFraction=0.6	8.5
12	spark.serializer=org.apache.spark.serializer.KryoSerializer	6.3
13	spark.io.compression.codec =lzf	6.8
14	spark.io.compression.codec = snappy	6.5

3. SQL Query: selection

Step	Conf	Duration
	Default	26
1	spark.executor.memory=3 GB	18
2	spark.executor.memory=5 GB	16
3	spark.executor.memory=7 GB	15.5
4	spark.executor.memory=8 GB	15
5	spark.executor.core=3	15
6	spark.executor.core=5	14
7	spark.executor.core=7	14
8	spark.executor.core=8	13
9	spark.memory.storageFraction=0.4	12
10	spark.memory.storageFraction=0.3	13
11	spark.shuffle.file.buffer=48 KB	11
12	spark.shuffle.file.buffer=64 KB	12
13	spark.reducer.maxSizeInFlight=64 MB	10.3
14	spark.reducer.maxSizeInFlight=80 MB	12
15	spark.serializer=org.apache.spark.serializer.KryoSerializer	14
16	spark.io.compression.codec =lzf	12
17	spark.io.compression.codec = snappy	9.8

4. Matrix Factorization

Step	Conf	Duration
	Default	28
1	spark.executor.memory=3 GB	17
2	spark.executor.memory=5 GB	15.8

3	spark.executor.memory=7 GB	15.4
4	spark.executor.memory=8 GB	15.5
5	spark.executor.core=2	15
6	spark.executor.core=4	14
7	spark.executor.core=6	12
8	spark.executor.core=8	13
9	spark.memory.fraction=0.7	11
10	spark.memory.fraction=0.8	13
11	spark.memory.storageFraction=0.6	13
12	spark.shuffle.file.buffer=48k	12
13	spark.reducer.maxSizeInFlight=64 MB	10.3
14	park.reducer.maxSizeInFlight=90 MB	11
15	spark.serializer=org.apache.spark.serializer.KryoSerializer	10.8
16	spark.io.compression.codec =lzf	12
17	spark.io.compression.codec = snappy	11

Appendix B

Industry Workloads

1. K-means

Step	Configuration	Duration	
	Default	54m	
1	spark.executor.memroy= 20G	32m	
2	spark.executor.memroy= 32G	28m	
3	spark.executor.cores= 7	25m	
4	spark.executor.cores= 12	25m	
5	spark.executor.instances= 4	23m	
6	spark.executor.instances= 7	21m	61
7	spark.memory.fraction=0.7	19.5m	
8	spark.memory.fraction=0.8	20m	
9	spark.memory.stroageFraction=0.6	19.2m	3
10	spark.memory.stroageFraction=0.7	21m	
11	Kryo-serializer	18m	2
12	Compression codec= lzf	21m	
13	Compression codec= snappy	22m	

2. Logistic Regression

Step	Configuration	Duration	
	Default: out of memory		
	8G RAM, 4 cores, 2 executors	74	
1	spark.executor.memroy= 32G	55m	
2	spark.executor.cores=12	42m	
3	spark.executor.cores:=20	39m	
4	spark.executor.instances= 4	34m	

5	spark.executor.instances= 7	27m	63
6	spark.memory.fraction=0.7	26m	
7	spark.memory.fraction=0.8	26m	
8	spark.memory.stroageFraction=0.6	25m 48s	
9	spark.memory.stroageFraction=0.7	25m 12s	2.7
10	spark.memory.stroageFraction=0.8	26	
11	Kryo-serialize	26m	
12	Compression codec= lzf	29m	
13	Compression codec= snappy	27m	

3. Random Forests

Step	**Configuration**	**Duration**	
	Default: out of memory		
	8G RAM, 4 cores, 2 executors	102	
1	spark.executor.memroy= 32G	80	
2	spark.executor.cores= 12	67	
3	spark.executor.cores= 20	63m	
4	spark.executor.instances= 4	58m	
5	spark.executor.instances= 7	52m	49
6	spark.memory.fraction=0.7	51m 12s	
7	spark.memory.fraction=0.8	50m 48s	
8	spark.memory.fraction=0.9	51m	
9	spark.memory.stroageFraction=0.7	49m 24s	3
10	spark.memory.stroageFraction=0.8	50m	
11	Kryo-serialize	51m	
12	Compression codec= lzf	49m 12s	1
13	Compression codec= snappy	52m	

4. SQL Query: Join

Step	Configuration	Duration	
	default	36	
1	spark.executor.memroy=32G	29m	
2	spark.executor.cores=20	28m	
3	spark.executor.instances= 4	25m	
4	spark.executor.instances= 7	22m	38
5	spark.memory.fraction=0.4	21m 42s	
6	spark.memory.fraction=0.3	23m	
7	spark.shuffle.buffer=48k	20m 42s	
8	spark.shuffle.buffer=64k	21m 12s	
9	spark.reducer.maxSizeInFlight=64m	20m 12s	6
10	spark.reducer.maxSizeInFlight=80m	21m 6s	
11	Kryo-serializer	19m 48s	3
12	Compression codec= lzf	22m	
13	Compression codec= snappy	20m	