Hummingbird: A Novel Approach of Adaptively the Spark Configurations for Big Data Analytics

***Abstract*—Real time and cost-effective analytics for Big Data is now a key ingredient for success in many businesses, scientific research and engineering disciplines, and government endeavors. The Spark platformwhich consists of an extensible MapReduce execution engine, pluggable distributed storage engines, and a range of procedural to declarative interfacesis a mainstream choice for big data analytics. Most practitioners of big data analyticslike computational scientists, systems researchers, and business analystslack the expertise to configure the system setting to get good performance. However, it is challenging to automatically identify the best configuration for a diversity of applications and cloud configurations. Unfortunately, Sparks performance out of the box leaves much to be desired, leading to suboptimal use of resources, time, and money (in pay as-you-go clouds). We introduce a new approach of adaptively the spark configurations for big data analytics. It builds on Spark while adapting to user needs and system workloads to provide good performance automatically, without any need for users to understand and manipulate the many tuning knobs in Spark.**

I. INTRODUCTION

With the rapid and continuous development of "Big Data" analyze like distributed natural language processing [1], deep learning for image recognition [1], genome analysis [1], astronomy [1] and particle accelerator data processing [1]. These applications differ from traditional analytics workloads (e.g., SQL queries) in that they are not only data-intensive but also computation-intensive, and typically run for a long time (and hence are expensive). Along with new workloads, we have seen widespread adoption of spark platform with large data sets being hosted [1], and the emergence of sophisticated analytics services, such as machine learning, being offered by spark providers[1].

With the era of big data coming, big data technologies have attracted more and more attention from the academia, industry and governments in recent years. The traditional computing model in the era of big data cannot meet the demands of performance and efficiency, Hadoop [2], Spark [3], Storm [4] and other distributed frameworks emerge as time goes on. Among several big data frameworks, Apache Spark has become the most popular and universal big data processing platform because of its excellent performance and rich application scenarios support.

With the wide application of Spark, it also exposes some problems in practical applications. One of the most important aspects is the performance problem. Due to unreasonable parameter configuration of Spark, the performance of the Spark applications is difficult to achieve theoretical peak speed of computer, so how to optimize the performance of Spark applications is one problem that is worthy of research.

Apache Spark has more than 180 configuration parameters, which can be adjusted by users according to their own specific application so as to optimize the performance [3]. This is the most simple and effective way to optimize the performance of Spark applications. On the one hand, the large parameter space enables a lot of opportunities for significant performance improvement by careful parameter tuning, but on the other hand, the large parameter space and the complex interactions among the parameters make the appropriate parameter tuning a challenging task.

Currently, there are two ways to tuning the configuration parameters of big data platforms. Firstly, these parameters are tuned manually by trial and error, which is ineffective and time consuming due to the large parameter space and the complex interactions among the parameters. In order to overcome the disadvantages of tuning parameters manually by trial and error, some researchers proposed a cost-based (analytical) performance modeling method to tuning the parameters of Hadoop platform [5].However, Spark is much different from Hadoop in the underlying implementation mechanism, so that we cant use this method on Apache Spark directly. At the same time, it suffers from several drawbacks when modeling Spark platform based on cost. First, cost-based modeling is a white box approach where a deep knowledge about a systems internals is required, and since the system that comprises the software stack (operating system, JVM, Spark, and workloads) and the hardware stack is very complex, it becomes very difficult to capture this complexity with the cost-based model. Second, Spark has several customizable components where users can plug in their own policies, such as scheduling and shuffling policies, which makes it necessary for the performance model to support these different policies.

These drawbacks motivate us to explore machine learning based performance models, which are black box models. Compared with the existing approaches, black box models have two main advantages. First, the recommendations of an auto-tuner using these kinds of models are based on observations of the actual system performance under a particular workload and cluster. Second, they are usually simpler to build than white box models as there is no need for detailed information about the systems internals. Finally, our proposed method relies on training data to learn the performance model, which makes it more robust and flexible.

The rest of the paper is organized as follows. The related work is discussed in Section II. A brief overview of Apache Spark is presented in Section III. A performance model for multiple model fusion based on machine learning and

optimization method of spark platform will be proposed in Section IV. Experimental methodology and experimental result analysis are given in Section V. Finally, the conclusions and future work are summarized in Section VI. [1]

II. Related Work

Performance optimization of big data System has been one of the hot research areas in recent years due to the wide adoption of big data analytic platforms. But most of research works are based on the MapReduce computing framework or Hadoop platform. Starfish [5] uses cost-based modeling and simulation to search the desired job configurations for MapReduce workloads. AROMA [6] automates job configuration and resource allocation through leveraging a two phase machine learning and optimization framework for heterogeneous clouds. In reference [7], the authors point out that Hadoop's scheduler can cause severe performance degradation in heterogeneous environments and present a new scheduler called Longest Approximate Time to End. Another work [8] focused on analyzing the variant effect of resource consumption of different settings for the Map and Reduce slots. In reference [9], the authors address these problems by presenting the Profiling and Performance Analysis-based System (PPABS) framework, which can automate the tuning of Hadoop configuration settings based on deduced application performance requirements. The key contributions includes the modifications to the well-known KMeans++ clustering and Simulated Annealing algorithms, which were required to adapt them to the MapReduce paradigm. Reference [10] proposes to alleviate this issue with an engine that recommends configurations for a newly submitted analytics job in an intelligent and timely manner. The engine is rooted in a modified k-nearest neighbor algorithm, which finds desirable configurations from similar past jobs that have performed well.

The research work about the performance optimization of Apache Spark platform is still relatively few. In reference [11], the authors present simulation driven prediction model that can predict job performance with high accuracy for Apache Spark platform. Their model is used to predict the execution time and memory usage of Spark applications in the default parameter case. However, their model is too simple to predict the execution time of different configuration parameters. In reference [12], the authors show that support vector regression model (SVR) has good accuracy and is also computationally efficient. Their findings reveal that their autotuning approach can provide comparable or in some cases better performance improvements than Starfish with a smaller number of parameters.

III. Definitions

A. Apache Spark

Apache Spark [3] is one of the popular open-source big data platforms, which introduces the concept of resilient distributed datasets (RDDs) [3] to enable fast processing of large volume of data leveraging distributed memory. Inmemory data operation mechanism makes it well-suited for iterative applications such as iterative machine learning and graph algorithms.

The main abstraction in Spark is resilient distributed dataset (RDD), which represents a read-only collection of objects distributed across a set of machines. Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like parallel operations. RDDs achieve fault tolerance through a notion of lineage: if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition. Although RDDs are not a general shared memory abstraction, they represent a sweet-spot between expressivity on the one hand and scalability and reliability on the other hand, and we have found them wellsuited for a variety of applications. The framework of Spark cluster is shown in Figure 1.

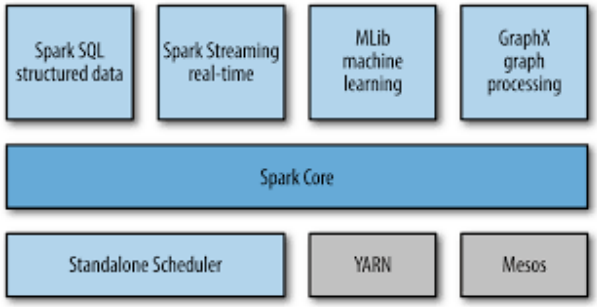

Fig. 1: Architecture of Apache Spark.

B. Definitions

IV. Our Proposed Approach

In this section, a novel approach of adaptively the Spark configurations platform is proposed, and based on the model; several common machine learning algorithms for configuration

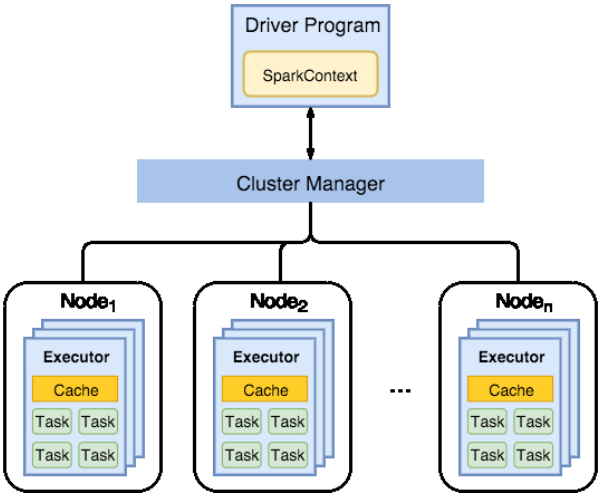

Fig. 2: Workflow of Spark

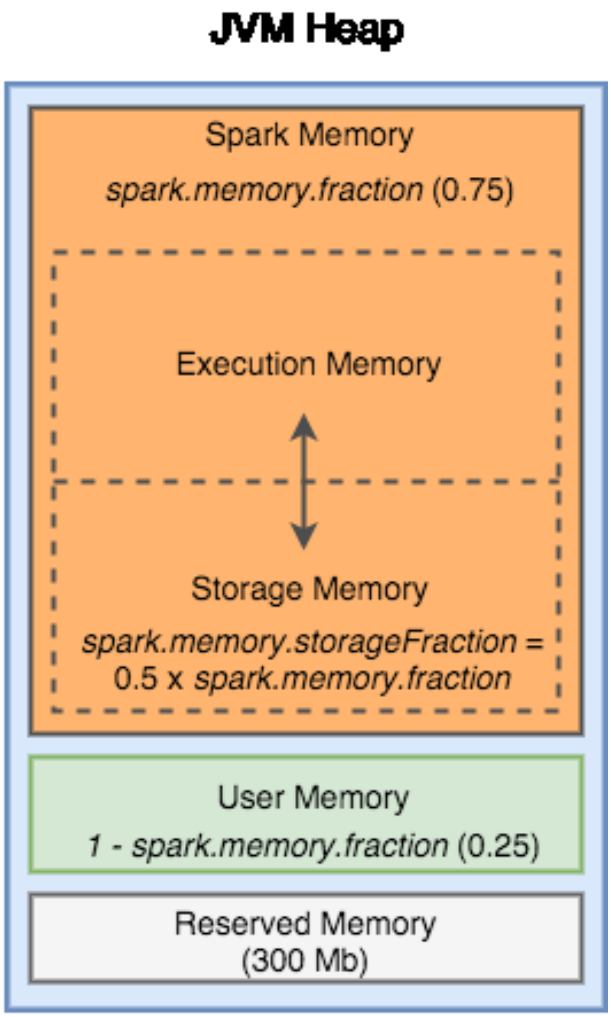

Fig. 3: Parameters of JVM Heap

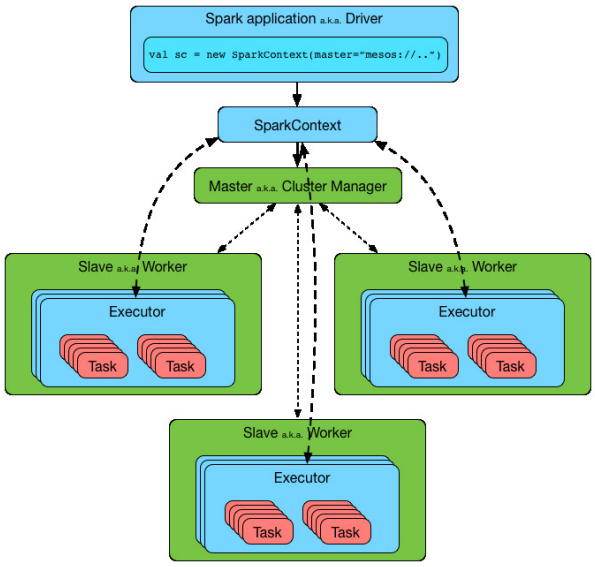

Fig. 4: Running of Spark

parameter tuning are explored. And then the specific details of parameter selection, coding method, evaluation function, weighted allocation, chaos handling, data collection, and training, testing and parameter space searching are described. Finally, the models are evaluated in their accuracy, computational performance, and their sensitivity to various variables such as size of training data, application characteristics.

A greedy algorithm is an algorithmic paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage[2] with the hope of finding a global optimum. In many problems, a greedy strategy does not in general produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a global optimal solution in a reasonable time.

Our approach will be based on Chaos Particle Swarm Optimization(CPSO) algorithms. We will adopt some Spark parameters as ?????

Markov decision processes (MDPs) provide a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker. MDPs are useful for studying a wide range of optimization problems solved via dynamic programming and reinforcement learning.

A. *Parameters Selection*

In this work, 22 parameters are selected, which are shown in Table I. The first thirteen parameters are configuration parameters of Spark and the last one is the size of input data. In Table I, the default column lists the default value of each parameter, and the rules of thumb column lists the values of each parameter that the industry recommends [13][14]. The reasons for selecting these parameters are listed as follows. First, the influences of these parameters cover almost all of the available resources in a cluster, such as CPU, Memory, and Disk. Secondly, these parameters have a great impact on different modules of Spark, such as schedule, shuffle and compress modules. Finally, these parameters have impact in different levels of a cluster, such as machine-level, cluster-level. Generally speaking, various factors that affect the performance of the cluster are comprehensively considered. Based on our domain expertise, these parameters that have significant impact on the performance of Spark are selected.

However, parameter selection in a machine learning-based performance model is a non-trivial research issue, which cant only depend on the domain experts; we will concentrate on this in the future work.

B. *Coding Method*

C. *Evaluation Function*

D. *Weighted Allocation*

E. *Chaos Handling*

V. Exiperments and Evaluation

A. *Exiperments*

To evaluate the effectiveness of our method, a Spark cluster with 4 nodes (one master node and three slave node) is

deployed. The master node has Xeon processor E3-1200 v3 and 128GB memory. All slave nodes were with eight 3.40 GHz In-tel Core i7 and 8GB memory. Each node has also the same software stack: Ubuntu 16.04 LT and Spark 2.2.0 with Hadoop 2.7.3. The detailed configuration of the Spark cluster is listed in Table II.

B. Evaluation

In order to model the performance of Spark applications, we select twenty-two main parameters, which are shown in Table I. The first fourteen parameters are configuration parameters of Spark and the last one is the size of input data. In Table I, the default' column lists the default value of each parameter, and the rules of thumb column lists the values of each parameter that the industry recommends [13][14].

VI. CONCLUSION

In this paper, a novel approach of adaptively the spark parameter configurations for big data analytics is proposed, and a model based on binary classification and multi classification is established. Furthermore, several common machine learning algorithms for configuration parameter tuning are explored based on the model. Experimental results show that CPSO has good accuracy and computational performance across diverse workloads for binary classification and multi-classification. And experimental results also show that our approach is effective and available to tune the configuration parameters of Spark platform. An average of 36 % performance improvement can be got with the proposed method. Moreover, with the increase of input data, the effect of performance improvement is more obvious.

At the same time, the proposed approach is more robust and flexible as it is easier to adapt to changes. Experimental results demonstrate that it is possible to build an effective performance auto-tuner for Spark in a black box manner, that is, in manner of only using observations from the system and without getting its internals.

In the future, we will conduct further research on parameter selection in order to select more suitable parameters. And, we will also consider more parameters affecting on the performance, try to cover all of spark parameters.

ACKNOWLEDGMENT

The authors would like to thank all people in PDLab, especially Junjie Wu, Yiqiang Qiu and Haining Huang for their help during the preparation of this paper. This work is also supported by the National Natural Science Foundation of China (No. 61272437, 71203137 and 61672385).

REFERENCES

[1] R. Wang, D. Butnariu, J. Rexford *et al.*, "Openflow-based server load balancing gone wild." *Hot-ICE*, vol. 11, pp. 12–12, 2011.

[2] R. Cormen, Leiserson and Stein, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 2001.

TABLE I: Selected Parameters In Our Approach

Parameter Name	Default	Meaning	Rules of Thumb	Importance
spark.driver.memory	512m	Amount of memory to use for the driver process.	1g-4g	Directly affect computing performace of spark memory.
spark.driver.maxResultSize	1g	Each Spark action in the total size of the serialization results for all partitions.	1g-8g	It represents a fixed memory overhead per reduce task.
spark.executor.memory	1g	Amount of memory to use per executor process.	2g-8g	Directly affect computing performace of spark memory.
spark.driver.cores	1	Number of cores to use for the driver process.	1-8.	Directly affect the parallelism of the program.
spark.executor.cores	1	Number of cores to use per executor process.	10-40	Directly affect the parallelism of the program.
spark.shuffle.file.buffer	32k	Size of the in-memory buffer for each shuffle file output stream.	10-40	These buffers reduce the number of disk seeks and system calls made in creating intermediate shuffle files.
spark.reducer.maxSizeInFlight	48m	Maximum size of map outputs to fetch simultaneously from each reduce task.	24k-96k	This represents a fixed memory overhead per reduce task.
spark.shuffle.compress	true	Whether to compress map output files.	true/false	It will speed up the shuffle action.
spark.shuffle.spill.compress	true	Whether to compress data spilled during shuffles.	true/false	It will speed up the shuffle action.
spark.broadcast.compress	true	Whether to compress broadcast variables before sending them.	true/false	It will speed up the broadcast action.
spark.io.compression.codec	lz4	The codec used to compress internal data such as RDD partitions, event log, broadcast variables and shuffle outputs.	lz4/lzf/snappy.	It will speed up all of I/O action.
spark.rdd.compress	false	Whether to compress serialized RDD partitions.	true/fasle	It can save substantial space at the cost of some extra CPU time.
spark.default.parallelism	0.6	Default number of partitions in RDDs returned by transformations.	0.4-0.8	It will add RDD parallelism when user operate join, reduceByKey, and parallelize.
spark.memory.fraction	0.6	Fraction of (heap space - 300MB) used for execution and storage	0.4-0.8	The purpose of this config is to set aside memory for internal metadata, user data structures, and imprecise size estimation in the case of sparse, unusually large records.
spark.memory.storageFraction	0.5	Amount of storage memory immune to eviction, expressed as a fraction of the size of the region set aside by it.	0.3-0.8	The higher this is, the less working memory may be available to execution and tasks may spill to disk more often.
spark.broadcast.blockSize	4m	Size of each piece of a block for TorrentBroadcastFactory.	2m-8m	It will affect parallelism during broadcast.
spark.files.useFetchCache	true	File fetching will use a local cache that is shared by executors that belong to the same application, which can improve task launching performance when running many executors on the same host.	true/false	This optimization may be disabled in order to use Spark local directories that reside on NFS filesystems.
spark.files.openCostInBytes	4M	The estimated cost to open a file, measured by the number of bytes could be scanned in the same time.	true/false	the partitions with small files will be faster than partitions with bigger files.
spark.storage.memoryMapThreshold	2m	Size of a block above which Spark memory maps when reading a block from disk.	1m-4m	It prevents Spark from memory mapping very small blocks.
spark.rpc.message.maxSize	true	Maximum message size (in MB) to allow in ”control plane” communication.	true/false	It will speed up the network action when you are running jobs with many thousands of map and reduce tasks.
spark.speculation.quantile	0.75	Fraction of tasks which must be complete before speculation is enabled for a particular stage.	0.45-0.9	It will speed up the task.
spark.dynamicAllocation.enabled	true	Whether to use dynamic resource allocation, which scales the number of executors registered with this application up and down based on the workload.	true/false	It will increase the resource when we are handling a workload .