Hummingbird: A Novel Approach of Finding the Optimal Configuration for Big Data Analytics

Paper ID: 1570388224

***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. Unfortunately, the most beginner of big data analytics like software engineer, computational scientists, systems researchers, and business analysts lack the expertise to configure the big data analytics system setting to get good performance. Due to each of big data analytics platforms has hundreds of parameters for tuning. However, it is challenging to automatically identify the best configuration for a diversity of applications. The Apache Spark platform which consists of an extensible MapReduce execution engine, pluggable resilient distributed storage engines, and a range of procedural to declarative interfaces is a mainstream choice for big data analytics. In generally, Spark 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 platform, like AWS, Azure etc.). We introduce the Hummingbird: a novel approach to find the optimal configuration for big data analytics. It builds on Apache Spark while adapting to user needs and job workloads to provide good performance, help users to understand and manipulate the many tuning knobs in Spark. Furthermore, experimental results show the Hummingbird can get the best configuration considering running time and computational efficiency. Finally, the experimental results also show that the performance can get average 23% gain with the Hummingbird compared with the default configuration of Spark. Certainly, Hummingbird can also be applied to other mainstream big data analytics systems.**

I. INTRODUCTION

With the rapid and continuous development of big data analyze like distributed natural language processing [1], deep learning for image recognition [2], genome analysis [3], astronomy [4] and particle accelerator data processing [5]. 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 efficient big data anaytics platform with large datasets being hosted [6], and the emergence of sophisticated analytics services, such as machine learning, being offered by data anaytics providers[7].

According to IDC(Internet Data Center) predictions by 2020, the size of the digital universe will exceed 44ZB [8]. So that we have entering the era of big data, and big data analytics 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 [9], Spark [10], Storm [11] 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 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 [10]. This is the most simple and effective way to optimize the performance of Spark applications. On the one hand, a lots of parameters enable 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.

New practitoners of big data analytics like computational scientists and systems researchers lack the experitse to tune Spark to get good performance. Currently, there are two ways to tuning the configuration parameters of Spark platforms. Firstly, these parameters are tuned manually by trial and error, which is ineffective and time consuming due to a lots of parameters and the complex interactions among the parameters. In order to overcome the disadvantages of tuning parameters manually, some researchers proposed a cost-based (analytical) performance modeling method to tuning the parameters of Hadoop platform [9]. However, Spark is much different from Hadoop in the underlying implementation mechanism, so that we can not use this method on Spark directly. Meanwhile, it suffers from several weakness when modeling Spark platform based on cost. First, cost-based modeling is a white box approach where a deep and wide 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 hard to capture this complexity with the cost-based model. Secondly, Spark has several customizable components where users can plug in their own policies, such as scheduling and encryption policies, which makes it necessary for the performance model to support these different policies.

These drawbacks motivate us to explore new approach 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 Automatic

tuning using these models are based on observations of the actual system performance with logging information. Second, they are usually more simple to build as there is no need for detailed information about the systems internals. Therefore, our proposed method relies on optimization theory to build 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 and some definition are 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. As a appendix, All of our selected parameters which are most important to the optimal configuration of Spark are listed in the Section VII .

II. Related Work

Performance optimization of big data system has been one of the hot topic in academic research areas recently due to the wide application of big data analytic platforms. But most of research works are still based on the MapReduce computing framework or Hadoop platform. Starfish [12] uses cost-based modeling and simulation to search the desired job configurations for MapReduce workloads. AROMA [13] automates job configuration and resource allocation through leveraging a two phase machine learning and optimization framework for heterogeneous clouds. In reference [14], 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. Reference [15] focused on analyzing the variant effect of resource consumption of different settings for the Map and Reduce slots. In reference [16], 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 [17] 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. Another work [18] proposed a new approach of task scheduling for spark using SDN, but it can not handle the tuning of configration.

The research work about the performance optimization of Apache Spark platform is still relatively few. In reference [19], 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. Ernest [20] introduce a efficient performance prediction method for large-scale analytics, but this method can not tune any of complex parameters. The latest reference Cherrypick [21] proposed a novel architecture for adaptively unearthing the best cloud configuration, However it just can keep a trade-off between cloud cost and Spark configration. Another work [22], 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.

Next we will briefly introduce the Spark platform and its running modal.

III. Apache Spark

In this section we have a briefly insight of Spark architeture and we will know about running processes of Spark.

Apache Spark [10] is the most popular open-source big data platforms in the world, which introduces the concept of resilient distributed datasets (RDDs) [10] to enable fast processing of large volume of data leveraging distributed memory. In memory data operation mechanism makes it well-suited for iterative applications such as iterative machine learning and graph algorithms. It need not swap data between harddisk and memory when it is running.

The Architecture of Spark is shown in Figure 1 as follows.

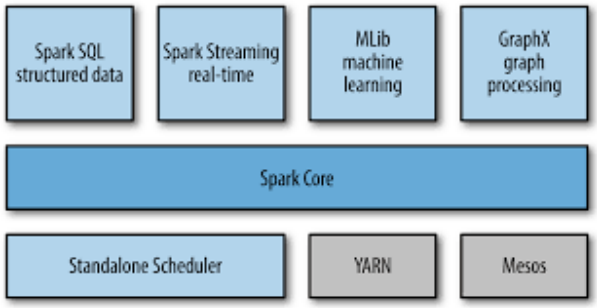


Fig. 1: Architecture of Apache Spark.

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.

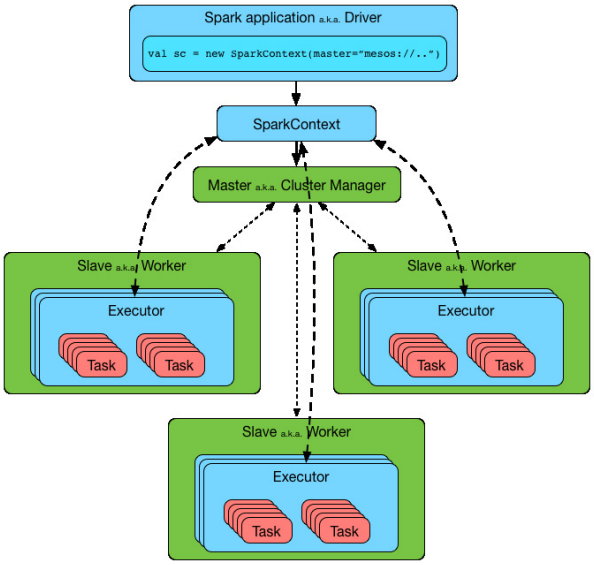


Fig. 2: Running Processes of Spark

Now we will show a typical deployment model of spark clusters (Standalone mode); A Spark cluster typically has multiple processes, each running in its own JVM, and a Spark cluster has a master node and multiple worker nodes, which are equivalent to Hadoop's master and slave nodes. The master node has a master daemon process, which manages all the worker nodes. The master daemon allocates resources across applications. The worker node has a worker daemon process, which is responsible for communicating with the master node and managing local executors. The worker also monitors the liveness and resource consumption of the executors. Each application has one driver and multiple executors. The tasks within the same executor belong to the same application. The driver is the process running the main function of the application and creating the SparkContext.

The following Figure 2 describes the most significant processes.

Because of the in-memory nature of most Spark computations, Spark programs can be bottlenecked by any resource in the cluster: CPU, network bandwidth, or memory. Most often, if the data fits in memory, the bottleneck is network bandwidth, but sometimes, you also need to do some tuning, such as storing RDDs in serialized form, to decrease memory usage.

Fianlly we observe that Apache Spark tunning configuration is very complex, running process is very flexible by users different setting. Moreover we can also sample the parameters by system internal insight. Next we will look at the design of the Hummingbird.

IV. Our Proposed Approach

In this section, a novel approach of adaptively the Spark configurations platform is proposed. And based on the optimizition theroy, we explored a novel algorithm for configuration parameter tuning. And then the specific details of parameter selection, coding method, evaluation function, weighted allocation and Construct the learning examplar are described.

In our problem, the ultimate objective is to find the best configuration. Therefore, the approach does not have enough information to be an accruate performance predictor, but this information is sufficient to find a good configuration within a few steps. As a well known, a greedy algorithm is an algorithmic paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage [23] 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. For example: Particle Swarm Optimization Algorithm(PSO), Ant Colony Optimization Algorithms(ACO), Simulated Annealing Algorithm(SA) etc. PSO is very popular of all, it has a good convergence of the sequence of solutions. However PSO will always get a local optimalization. This means that determining convergence capabilities of different PSO algorithms and parameters therefore still depends on empirical results. So we will attempt at addressing this issue is the development of an "orthogonal learning" strategy for an improved use of the information already existing in the relationship between p and g, so as to form a leading converging exemplar and to be effective with any PSO topology. The aims are to improve the performance of PSO overall, including faster global convergence, higher solution quality, and stronger robustness [24].

In this paper, we explore to select some key parameters of Spark configureation which will speed up its performance, and compose them to different setting soluations as the particle positions based on PSO algorithm. We will also introduce orthogonal learning strategy for improving the PSO performance. Furthermore we will get the global optimalization and improve the performance of Spark.

The flowchart of our proposed approach is shown in Fig. 3.

From Figure 3, our approach is described as the following five steps. These parameters are discussed in Section IV-A. A coding method of particle positions is presented in Section IV-B. A object function f_x for evaluation will be proposed in Section IV-C. The weight factors are given in Section IV-D. Finally, the construction work of learning examplar will detail in Section IV-E.

A. Parameters Selection

In this work, we will select some key parameters from hundreds of Spark parameters [25]. Specially, we investigate the JVM heap, which is key components of Spark [26]. We can see it from Firgure 4 as follows.

Finally, twenty-two parameters are selected, which are shown in Table III of Appendix Section. In Table III, the Default column lists the default value of each parameter. The 'Meaning' column explains the meaning of this parameter. The rules of thumb column lists the values of each parameter that

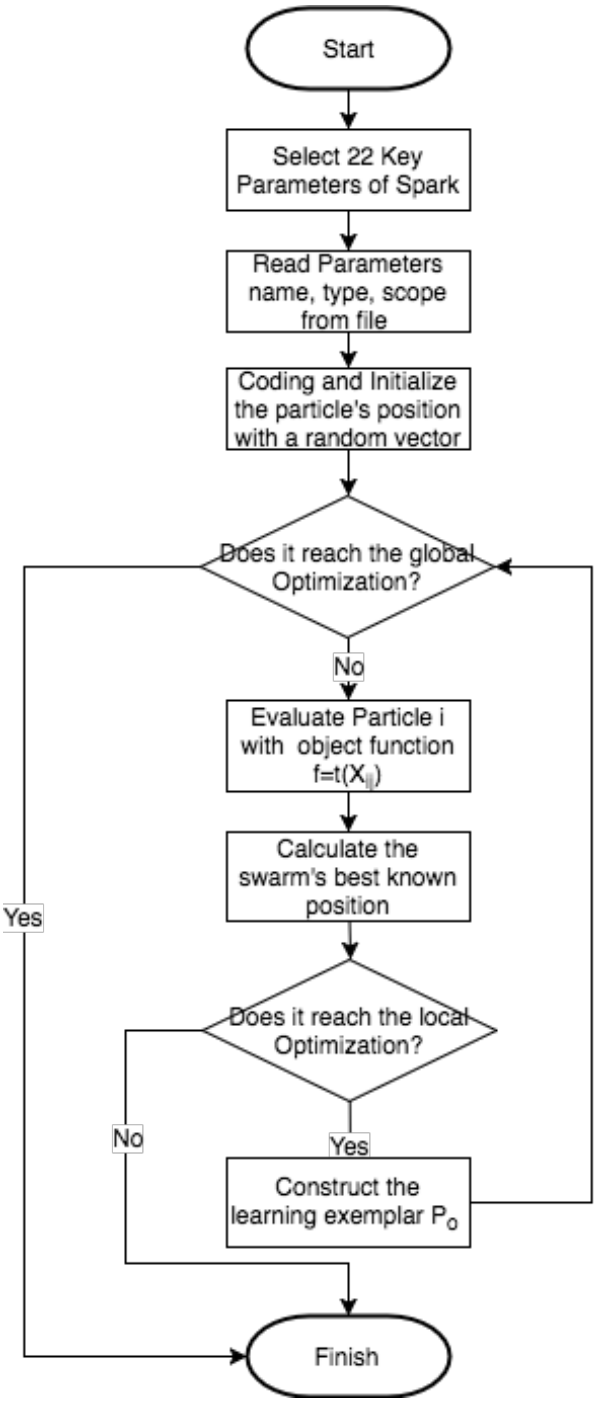


Fig. 3: The flowchart of our proposed approach.

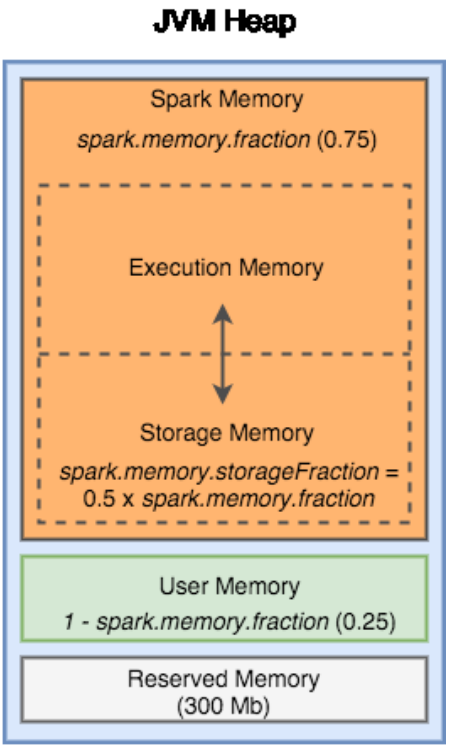


Fig. 4: Parameters of JVM Heap

the industry recommends [27] [28], and the 'Importance' column is brief introduce the reasons of selection. The principle for selecting these parameters are listed as follows.

- The influences of these parameters cover almost all of the available resources in a cluster, such as CPU, Memory, and Disk.
- these parameters have a great impact on different modules of Spark, such as schedule, shuffle and compress modules.
- 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 testing expertise, these parameters that have significant impact on the performance of Spark are selected.

However, parameters selection in a Hummingbird approach is a non-trivial research issue, which cant only depend on the deep insight of system. we will make up on this in the future work.

B. Coding Method

Type of Spark parameters have some difference type, such as: Float, Interger, Enumeration, Boolean. Because binary coding will occur the mapping errors when it discretize in the continuous function, in this paper we will adopt the floating coding method. The length of coding is decided by num of parameters. We assume a set of parameters as $A = p_1, p_2, \cdots, p_N$, the value of floating coding will be as $V = v_1, v_2, \cdots, v_N$. It is shown in Table I as follows.

TABLE I: Coding of Floating Number

Floating Number	a_1	a_2	a_3	$\cdots$	a_N
Coding	v_1	v_2	v_3	$\cdots$	v_N

Other types of parameters values also need to convert to floating type. We assume the range of integer and floating values of parameter are (R^{min}, R^{max}), and the range of enumeration is $(R_1, R_2, \cdots, R_N)$. Therefore we will get the normalization formula as follows:

$$R = \begin{cases} \frac{R-R^{min}}{R^{max}-R^{min}} & Float \\ \frac{R-R^{min}}{R^{max}-R^{min}} & Interger \\ R = true?1.0 : 0.0 & Boolean \\ \frac{i}{N} & Enumeration, R = R_i \end{cases} \quad (1)$$

We now deduce the anti-normalization formula as follows.

$$R = \begin{cases} R' \times (R^{max} - R^{min}) + R^{min} & Float \\ round(R' \times (R^{max} - R^{min}) + R^{min}) & Interger \\ round(R') = 1?true : false & Boolean \\ R_{round(R' \times N)} & Enum \end{cases} \quad (2)$$

Using this coding method will guarantee robustness and completeness of coding, But there are always exist multiple solutions of the coding space. Therefore, It does not satify the uniqueness. In the Hummingbird, we proposed a new method as follows. After a particle movement, the new position is adjusted. The new position is first normalized to obtain the actual attribute value; then the obtained value is normalized. Since the definition domain and the range of the inverse normalization formula are one-to-one correspondence, the solution of the problem space and the location of the coding space are also one-to-one correspondence. Consequently, it will ensure that the floating coding used satisfy non-redundancy in this paper.

C. Evaluation Function

As we known, The key criterion of evaluating configuration solution is job running time. Therefore, we will choose job running time as the main evaluation function. We assume that the position $X_{ij}$of particle i obtained after the jth iteration, and the running time of selected configration is $t(X_{ij})$. So the objective function is expressed as $f = t(X_{ij})$.

D. Weighted Allocation

The particle swarm optimization algorithm always maintains an acceleration in the case of constant inertia, which facilitates global search at the beginning. However, as the number of iterations increasing, the particles should gradually change from global search to local search. Therefore, the inertia weight should be gradual reduced. Paper [29] proposed a inertia weight method to handle it. The inertia weight w is designed as a linearly reduced function, and the formula is as follows.

$$w = w_{max} - \frac{w_{max} - w_{min}}{iter_{max}} \times k \quad (3)$$

where the w_{max} stands for the initial weight, the w_{min} stands for the final weight. The general values are 0.9 and 0.4 respectively. and $iter_{max}$ and k are the max iterations and the current iterations. After modified the inertia weight, particle will transit from global searching to local searching. It guarantees that the alogithm wil have the good convergence. The speed and position of particle will formula as follows.

$$V_{ij}^{k+1} = wV_{ij}^{k} + C_1R_1(pbest_{ij} - X_{ij}^{k}) + C_2R_2(gbest_j - X_{ij}^{k}) \quad (4)$$

$$X_{ij}^{k+1} = X_{ij}^{k} + V_{ij}^{k+1} \quad (5)$$

E. Construct the Learning Examplar

The orthogonal experimental design (OED) offers an ability to discover the best combination levels for different factors with a reasonably small number of experimental samples [30], [31]. We use the OED method to construct a promising learning exemplar. OED is used to discover the best combination of a particles best historical position and its neighborhoods best historical position. The orthogonal experimental factors are the dimensions of the problem and the levels of each dimension (factor) are the two choices of a particles best position value and its neighborhoods best position value on this corresponding dimension.

Let f_m denote the experimental result of the mth ($1 \leq m \leq M$) combination and S_nq denote the effect of the qth ($1 \leq q \leq Q$) level in the nth ($1 \leq n \leq N$) factor. The calculation of S_nq is to add up all the f_m in which the level is q in the nth factor, and then divide the total count of z_mnq, as shown in (6) where z_mnq is 1 if the mth experimental test is with the qth level of the nth factor, otherwise, z_mnq is 0. And P_0 is the new learning examplar.

$$S_nq = \frac{\sum_{m=1} Mf_m \times z_mnq}{\sum_{m=1} Mz_mnq} \quad (6)$$

In this way, the effect of each level on each factor can be calculated and compared. Compare $f(i)$ and $f(j)$ and the level combination of the better solution is used to construct the vector. A method to further reduce the number of the orthogonal combinations is to divide the dimensions into several disjoint groups and regard each group as a factor. This method also may be good for the problems whose dimensions are not independent of each other.

Next we will evaluate the Hummingbird using some popluar benchmarks.

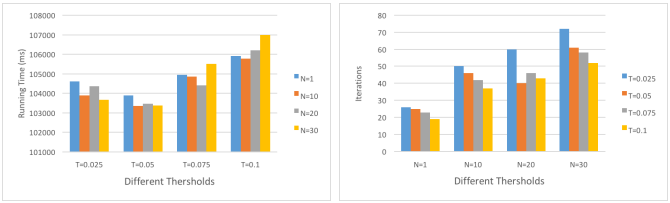
V. Exiperments and Evaluation

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.

We select WordCount and Sort benchmarks respectively. The iterations of algorithm is 100. The population number is 40. In order to get fast running , we set a threshold T, which represents difference degree in the between particles. The value range of T always between 0 and 0.1. If the

threshold value is too big, the particle will not reach the best position. If the threshold value is too small, the particle will enter the local optimization. And we select twenty-two main parameters, which are shown in Table III of Appendix Section. Our motivation for choosing these benchmarks is as follows. First, these workloads are simple and easy to understand. Secondly, these benchmarks are representative of real Spark applications, with a wide range of applications. Finally, these benchmarks cover different application types, including I/O intensive, CPU intensive, memory-intensive, and iterativeintensive.

We adopt the Sort benchmark first, and get some results as follows.



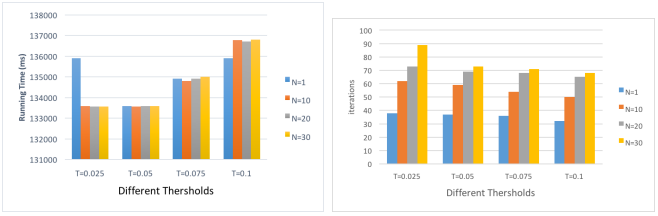
(a) The optimization solution when T and N take different values.

(b) The iterations when T and N take different values.

Fig. 5: Experiment Results in Sort Benchmark.

From above the Figure 5, respectively, for T and N take different values, the input 1.6 GB dataset to obtain the optimal solution and algorithm to obtain the optimal solution required for the number of iterations. When T = 0.025 and N= 20, the optimal solution is 104,371 ms and the number of iterations is 60. When T = 0.05 and N = 10, the suboptimal solution is 103,360 ms and the number of iterations is 46. In contrast, the optimal solution efficiency is lower by 0.02%, but the number of iterations is less than 38.68%. Therefore, when the number of iterations and the optimal solution is taken into account, the algorithm can reach the time of best search status when A = 0.05 and N = 10.

Now we address the Wordcount benchmark. Because the wordcount program has different steps, So we got the following results.



(a) The optimization solution when T and N take different values.

(b) The iterations when T and N take different values.

Fig. 6: Experiment Results in Wordcount Benchmark.

From above the Figure 6, respectively, for T and N take different values, the input 1.6 GB dataset to obtain the optimal solution and algorithm to obtain the optimal solution required for the number of iterations. When T = 0.025 and N= 30, the optimal solution is 133,583 ms and the number of iterations is 73. When T = 0.05 and N = 10, the suboptimal solution is 133,560 ms and the number of iterations is 59. In contrast, the optimal solution efficiency is lower by 0.018%, but the number of iterations is less than 36.63%. Therefore, when the number of iterations and the optimal solution is taken into account, the algorithm can reach the time of best search status when A = 0.05 and N = 30.

Table II for the selection of the same size of 20 different input data experiments, respectively, in the default configuration and our proposed approach to find the optimal configuration run under the average running time. It can be seen that compared with the default configuration scheme, the configuration efficiency of the system can be greatly improved by the our proposed approach, and the operation efficiency is more obvious when the input data of job is larger.

TABLE II: Running Time of Job between Defalut and Optimization.

Configuration	Job 0.8GB	Job 1.6GB	Efficiency(%)
Defalut	10,675ms	21,873ms	28.6
Optimization	7,952ms	14,626ms	18.9

In the other words, we can say the Hummingbird is very suited for big data analysis system. Such as mainstream Spark platform. It has a good robustness and stability.

VI. CONCLUSION

In this paper, Hummingbird: a novel approach of finding the optimal configurations for big data analytics is proposed, and Hummingbird based on PSO is established. Experimental results show that Hummingbird has good accuracy and computational performance across diverse workloads for Wordcount and Sort benchmarks. And experimental results also show that Hummingbird is effective and available to tune the configuration parameters of Spark platform. An average of 23% performance improvement can be got with different size datasets. Moreover, with the increase of input data, the effect of performance improvement is more obvious.

At the same time, our 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 for big data analytics system 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 parameters 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 and diverse application from real world.

REFERENCES

[1] T. Brants, A. C. Popat, P. Xu, F. J. Och, and J. Dean, "Large language models in machine translation," in *In Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*. Citeseer, 2007.

[2] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, A. Senior, P. Tucker, and K. Yang, "Large scale distributed deep networks," in *In Advances in Neural Information Processing Systems*. IEEE, 2012, pp. 1232–1240.
[3] L. D. Stein *et al.*, "The case for cloud computing in genome informatics." *Genome Biology*, vol. 11, no. 2, pp. 207–208, 2010.
[4] G. B. Berriman, G. Juve, E. Deelman, M. Regelson, and P. Plavchan., "The application of cloud computing to astronomy: A study of cost and performance." in *In Sixth IEEE International Conference on e-Science*, 2010.
[5] I. Bird, "Computing for the large hadron collider." *Annual Review of Nuclear and Particle Science*, vol. 61, pp. 99–118, 2011.
[6] "Common crawl." [Online]. Available: http://commoncrawl.org.
[7] "Machine learning, microsoft azure." [Online]. Available: http://azure.microsoft.com/en-us/services/machine-learning/.
[8] J. Gantz and D. Reinsel, "The digital universe in 2020: big data, bigger digital shadows, and biggest growth in the far east," 2012.
[9] "Apache hadoop." [Online]. Available: http://hadoop.apache.org/.
[10] "Apache spark." [Online]. Available: http://spark.apache.org/.
[11] "Apache storm." [Online]. Available: http://storm.apache.org/.
[12] H. Herodotou, H. Lim, and et al., "Starfish: a self-tuning system for big data analytics," in *Proc. of the 5th Biennial Conference on Innovative Data Systems Research (CIDR 2011)*. IEEE, 2011, pp. 261–272.
[13] P. Lama and X. Zhou., "Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud," in *Proc. of the 9th International Conference on Autonomic Computing (ICAC 2012)*. IEEE, 2012, pp. 63–72.
[14] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce rerformance in heterogeneous environments," in *Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI 2008)*. USENIX, 2008, pp. 29–42.
[15] K. Kambatla, A. Pathak, and H. Pucha, "Towards optimizing hadoop provisioning in the cloud," in *Proc. of the First Workshop on Hot Topics in Cloud Computing (HotCloud'09)*. IEEE, 2009, pp. 118–125.
[16] D. Wu and A. S. Gokhale, "A self-tuning system based on application profiling and performance analysis for optimizing hadoop mapreduce cluster configuration," in *Proc. of the 20th Annual International Conference on High Performance Computing (HiPC 2013)*. IEEE, 2013, pp. 89–98.
[17] R. Zhang, M. Li, and D. Hildebrand, "Finding the big data sweet spot: towards automatically recommending configurations for hadoop clusters on docker containers," in *Proc. of the 3rd IEEE International Conference on Cloud Engineering (IC2E 2015)*. IEEE, 2015, pp. 365–368.
[18] X. He and P. J. Shenoy, "Firebird: Network-aware task scheduling for spark using sdns." in *ICCCN*. IEEE, 2016, pp. 1–10.
[19] K. Wang, M. Maifi, and H. Khan, "Performance prediction for apache spark platform," in *Proc. of the 17th IEEE International Conference on High Performance Computing and Communications (HPCC 2015)*. IEEE, 2015, pp. 166–173.
[20] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: USENIX Association, 2016, pp. 363–378. [Online]. Available: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/venkataraman
[21] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 469–482. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/alipourfard
[22] N. Yigitbasi, T. L. Willke, G. Liao, and D. H. Epema, "Towards machine learning-based auto-tuning of mapreduce," in *Proc. of 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2013)*. IEEE, 2013, pp. 11–20.
[23] Cormen, Leiserson, Rivest, and Stein, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 2001.
[24] Z. ZH, Z. J, and S. Y. LiY, "Orthogonal learning particle swarm optimization," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 6, pp. 832–847, 2011.
[25] "Apache spark: Tuning spark." [Online]. Available: https://spark.apache.org/docs/latest/tuning.html.
[26] "Apache spark: Monitoring and instrumentation." [Online]. Available: http://spark.apache.org/docs/latest/monitoring.html.
[27] "Spark configuration." [Online]. Available: http://spark.apache.org/docs/1.6.1/configuration.htm.
[28] "Spark hub." [Online]. Available: https://sparkhub.databricks. com/.
[29] S. Yi and E. R. E. S. of Particle Swarm Optimization, "Towards machine learning-based auto-tuning of mapreduce," in *Proc. of the 1999 Congress on Evolutionary Computation*. IEEE, 1999, pp. 1948–1950.
[30] D. C. Montgomery, *Design and Analysis of Experiments*. New York: Wiley, 2000.
[31] C. A. S. Math. Stat. Res. Group, *Orthogonal Design*. Beijing, China: People Education Pub, 1975.

VII. Appendix

TABLE III: Selected Parameters In the Hummingbird

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 .