Tuning Performance of Spark Programs

Hong Zhang
Department of Computer Science
University of Central Florida
hzhang1982@knights.ucf.edu

Zixia Liu
Department of Computer Science
University of Central Florida
zixia@knights.ucf.edu

Liqiang Wang
Department of Computer Science
University of Central Florida
lwang@cs.ucf.edu

Abstract—**Along with the explosive growth of data, there is a great demand to speedup the ability to process them. Although there are several platforms such as Spark that have made analysis easier to developers, the performance tuning for such platforms meanwhile becomes complex. In this paper, we propose an efficient performance optimization engine called Hedgehog to evaluate the performance based on "Law of Diminishing Marginal Utility" and give an optimal configuration setting. The initial experiments show that our optimization can gain 19.6% performance improvement compared to the naive configuration by tuning only 3 parameters.**

I. Introduction

With the exponential growth of data generated by human and machine daily, people increasingly realize the importance in speeding up the performance of data analysis and process. There already exist several excellent commercial platforms such as Hadoop, Spark and Flink. Spark is an open source general-purpose processing framework for big data applications. In-memory storage design makes it run up to 100 times faster than Hadoop. It also supports more convenient operations that make it easy to use, and simple to program.

However, there are over 150 configuration parameters in Spark, which makes tuning performance very complicated, even for Spark experts with rich practical experience. And there exists no default configuration set suitable for every kind of application. Another major problem is how to collect enough profile information during execution for efficient fine-grained Spark tuning. Even Spark already collects some basic knowledge for applications, it is still short of a great amount of fine-grained information such as task and operation details. Because of such information shortage, existing application optimization techniques have a few shortcomings such as:

- Since there is no record for the duration of each operation, we cannot know vital operations to determine how to do code level optimization.
- Due to the lack of data throughput information, it is hard to understand data transferring relationship for each operation, especially for some general-purpose operations like "mapPartitions".
- Spark does not collect the resource usage information such as CPU and memory usage for each task, users cannot tune the resource allocation and parallelism in a correct way.
- Due to the lack of specific profiling for different memory types, it is impossible to know the optimal proportion of every memory type.

The models used to diagnose the performance for Spark can be categorized into two types: white box and black box. A white-box performance model requires necessary information extraction which utilizes instrumentation for application profiling and should understand the internal application structure and process; whereas a black-box model does not require the comprehension of the internal structure but analyzing the performance and behavior transparently. Sometimes, it may be impossible to analyze the performance behavior by a white box, since the application structure is too complex and the relationship among modules is hard to diagnose. The drawback of the black box is also apparent in lacking measure for investigating the root cause of a performance problem. Starfish [1] is a white-box performance model for Hadoop, which instruments the application and builds the relationship between execution performance and configurations by formulas. However, the overhead of the instrumentation method is too expensive, furthermore, its covering for wide-range applications and formalization makes it inaccurate for some types of applications. There are other existing researches to analyze the performance model but typically not focus on the internal structure and comprehensive process analysis [2**?** –6].

In this study, we propose an efficient 3-level sampling performance model, called Hedgehog, and focus on the relationship between resource and performance. This design is a brand new white-box model for Spark, which is more complex and challenging than Hadoop. In our tool, we employ a Java bytecode manipulation and analysis framework called ASM [7] to reduce the profiling overhead dramatically. Our contributions are summarized as follows:

- We design an end-to-end performance model for Spark that contains resource monitor, ASM based profiler, log collector and analyzer, performance evaluator, workflow analyzer, and optimizer, which is easy to use with low overhead.
- Instead of sampling all elements in each RDD, we use 3-level sampling model to sample the test application in input data level, task level, and element level. These 3-level sampling help us reduce the overhead dramatically, and avoid inaccuracy introduced by the profiling process.
- We also introduce a new white box performance model which focuses on the most influential parameters, most of which are related to the resource allocation and parallelism. Our small but rigorous performance model

captures the essential process in Spark, and avoids trivial analysis that contributes less to the performance improvement.

- We employ an important economic law, "Law of Diminishing Marginal Utility", to design our optimizer for evaluating the performance benefit of each unit resource allocation, and determine the optimal memory proposition to make fully use of every type of resource.
- Our performance model not only gives an optimal configuration set, but also helps user determine the cluster size in a public cloud, who can seek suitable cluster size at lower cost.

II. Background

Spark is a distributed computing framework to process big data across cluster. Instead of keeping intermediate data on disk like Hadoop, Spark tries to encourage user to keep them in memory. In addition to "Map" and "Reduce" stages in Hadoop introduced by MapReduce model, which limits the workflow of user's application, Spark supports multiple stages and offers over 80 high-level operators like "filter", "join", and so on. Users can write applications not only in Java, but also in Python and Scala. However, these fancy features make the design and analysis for its performance model more complicated and intricate.

A. Spark 5-Level Architecture

A Spark program includes 5 hierarchical levels: application, job, stage, task, and operation, which is shown in Figure 1. Instead of having job as the highest level in Hadoop, the highest level for Spark is application, which is submitted by a client to the resource manager and is launched in an ApplicationMaster container. An application can contain more than one jobs, the number of which depends on the number of actions, *i.e.*, each action is executed by a job. On the application level, all jobs are merged into one package, which makes it easier to design and more efficient to execute a recursive application such as machine learning application. Each job is triggered by one action, which always returns the result to the driver. In each job, due to the number of shuffling operations, it can be divided into many stages. Spark must shuffle data between two neighboring stages, which is also called wide dependency. A stage may contain multiple tasks, each of which handles one partition of data. Each RDD usually consists of many partitions, thus multiple tasks in each stage may run in parallel on compute nodes. For each task, it still executes several narrow dependent operations, which indicates that operations are executed like pipelining without network shuffling. The lowest level is operation. In general, the lower the level is, the less execution information Spark collects since the overhead is more expensive.

B. Memory Structure

Ever since Apache Spark version 1.6.0, Spark memory is divided into three regions to manage: user memory, storage memory, and execution memory [8], as shown in Figure

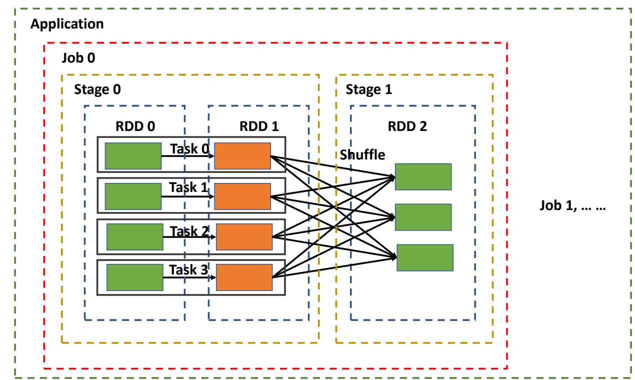

Fig. 1. 5-Level Structure for Spark Application.

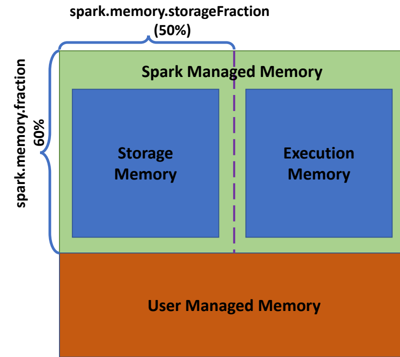

Fig. 2. Memory Structure in Spark 1.6+.

2. User memory completely depends on the user-defined function. The quality and feature of the user code has direct effection on this part of memory usage. Storage Memory is used for both cached data and "broadcast" variables. This type of memory is related to data reusing and broadcasting. Execution Memory stores the intermediate shuffling data on both Map side and Reduce side. The boundary between the storage memory and the execution is not fixed, which means if one kind of memory is insufficient, it can borrow space from the other. How to decide the memory fraction for each memory region is a very challenging problem since the memory usage is totally case by case, and there are so many influencing factors making it hard to analyze.

III. Motivation

Before input dataset is uploaded and processed by Spark, it is usually stored in a distributed file system such as HDFS and divided into blocks across clustered computers. Normally, each task in the first stage selects one block locally to process, consequently the number of tasks in the first stage is determined by the number of blocks in distributed file system. For the subsequent stages, the "repartition" function can be used to increase or decrease the number of partitions. However, there exists a big challenge for users to decide how much memory and CPU resource to be allocated for each task, and

whether repartition is applied when shuffling data between adjacent stages.

When a Spark user submits an application, three major resource allocation parameters need to be considered: the number of cores per executor, memory size per executor, and the number of executors. How to configure these parameters relies on the user's experience. For instance, considering a case where a cluster contains 10 DataNodes, and each DataNode has 16 cores and 128 GB memory together, we first reserve 1 core and 8 GB memory for OS, Hadoop and Spark. Consequently there are 15 cores and 120 GB per node left to allocate. Then the user can determine how many cores be assigned to each task by "spark.task.cpus", which is 1 by default. We assume 1 core for each task is reasonable for the user's application so that the user can execute 15 tasks in parallel in each DataNode. Therefore we can easily calculate how much memory assigned to each task (120 / 15 = 8 GB), however we also need to consider other overheads, so 7 GB per task makes more sense. If 7 GB is not enough for each task, an "out of memory" error will occur, and the user must increase the memory size per task and recalculate the number of tasks per DataNode. Here we assume 7 GB per task is large enough, then decide how many tasks are executed in each executor. Tasks executed in the same executor can share memory and other resource, but too many tasks being processed in the same jvm could lead to poor performance. Here we assume 5 cores per executor is a reasonable configuration. Since we have 15 cores available in each DataNode, 15 / 5 = 3 executors can be allocated in each node, and each executor contains $5 \times 7 = 35$ GB memory.

From analysis above, it is apparent that this configuration depends on too many assumptions we made, which may not make sense or even are wrong in reality, such as 1 core for each task.

IV. Design and Implementation

A. Hedgehog Structure

Figure 3 shows the detailed architecture of our performance optimization engine, called Hedgehog. Hedgehog is a comprehensive performance tuning measure for Spark, which profiles the fine-grained executing information from Spark system, collects necessary information from Spark original logs, analyzes the workflow of Spark application, employs Detective Marginal Utility Model (DMU) to tune ratios among different memory categories, and optimize the performance by reasonable resource allocation. Hedgehog contains several components to coordinate the optimization for Spark.

1) Dynamic Resource Monitor: To monitor the system resource, including CPU usage, memory usage per task dynamically, our resource monitor records CPU usage and Java heap size for each Spark operation, including transformation, and action. The major difference between our monitor and others is that it can monitor the whole life cycle of each operation to exactly tell the resource utilization for each operation and identify the most influential operations in every stage.

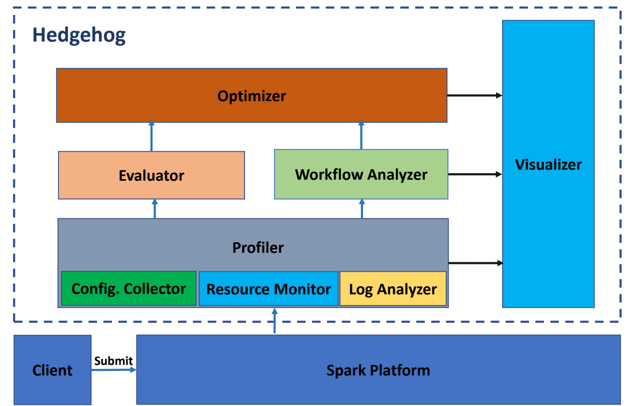

Fig. 3. System Architecture of Hedgehog.

2) Profiler: As aforementioned, Spark application contains 5-level structure; however, the current execution profiling in Spark is far from being sufficient for more advanced performance model. We design a fine-grained profiler to collect more information to help users understand what is happening, provide higher prediction accuracy, and improve job performance, such as real-time data flow, data locality status, container status.

3) Configuration Collector: Even being solely insufficient to our performance model, in reality, Spark already collects lots of useful information about the application, such as job, stage, task duration, and sub-phases like serialization/deserialization time and shuffle read/write time. This collector helps us collect all logs generated by Spark into a user-defined directory. Together with information collected by our profiler, we are able to collect all information from the fine-grained level to highest level.

4) Log Analyzer: We design a log analyzer to extract necessary information, and organize them with the application structure by logs of Spark and our profiler.

5) Evaluator: Based on the information collected, we design a brand-new resource-based performance model to predict application duration. Since Spark utilizes in-memory processing, which keeps data in-memory as much as possible to improve the performance, the influence of the resource especially the memory is very important. A reasonable resource allocation and configuration not only improves the performance of each task, but also optimizes the parallelism to speed up the execution for each stage.

6) Workflow Analyzer: The major functionality of the workflow analyzer is to detect recursive structures in the application. Each stage has a description to describe its major operations, invoked class and code line number. Stages with same description reveal the executing of same task. With scanning all stages in ascending order, we use the stage description as key, the latest stage ID as value, and calculate the stage interval between the neighboring stages with the same key. Then we calculate iterations for this application to simplify the evaluation process.

7) Optimizer: With the knowledge obtained from the evaluator and workflow analyzer, the optimizer employs "Law of Diminishing Marginal Utility" to allocate the memory resource properly. The major configuration problem for user is to configure the executor size (CPU cores, and memory size). Our optimizer can analyze the real CPU usage and memory usage for different phases to improve the resource utilization ratio.

B. Performance Model

Since there are over 150 parameters in Spark. we cannot determine their affection exhaustively. Some parameters have significant impact on performance, some of them depend on the job characteristics. Some of them must be set before running a job, some could be changed after a job launches but before the task runs, and some of them could be reset while the task is running. Hence, figuring out the affection and types of these parameters is very important for our performance model and tuning mechanism. In our performance model, we only consider those general, important, but tricky-tuning parameters, especially related to memory.

$$t_{app} = \sum_{i=1}^{n} t_{job}^{i} = \sum_{i=1}^{n} \sum_{j=1}^{i_m} t_{stage}^{ij} \tag{1}$$

The total execution time t_{app} can be calculated by Equation 1, t_{job}^{i} denotes the duration of job i, and the duration of stage j in job i is symbolized by t_{stage}^{ij}. The execution time of each stage can be calculated by Equation 2. The whole stage is divided into 3 sub-phases: read, run, and write, their durations are t_{read}, t_{run}, and t_{write}, respectively. Spark needs to spill the intermediate data into disk and merge them later if the data are too large. Thus the sub-phases of read and write contain $spill$ and $merge$ processes, which are related to the execution memory. The read sub-phase also needs to load data from the previous stage or from file system, therefore it has $load$ process, which is tied with the storage memory. The user memory affects the duration of the garbage collection process t_{gc}, which belongs to the run sub-phase. Because of the limitation of space, we omit details here.

$$\begin{aligned} t_{stage} &= t_{read} + t_{run} + t_{write} \\ &= t_{rload} + t_{rspill} + t_{rmerge} \\ &+ t_{exact} + t_{gc} + t_{wspill} + t_{wmerge} \end{aligned} \tag{2}$$

V. Initial Results

We evaluated Hedgehog on a cluster containing 10 nodes, 1 NameNode and 9 DataNode. Each node has an Intel(R) Xeon(R) CPU E5-2620 v3 with 6 cores, and 32 GB memory. Our Spark cluster is based on CentOS Linux Server 7, JDK version 1.8, Apache Hadoop version 2.7 and Apache Spark 2.1. We build a very simple performance model which only bound 3 configuration parameters: "executor-memory", "executor-cores", and "num-executors". The results show that our optimizer can gain 19.6% performance improvement compared to the naive configuration.

VI. Conclusions

In this paper, we design an end-to-end performance model for Spark, and use 3-level sampling model to sample the test application, *i.e.*, input data level, task level, and element level. We also introduce a new white box performance model to evaluate the performance based on "Law of Diminishing Marginal Utility". Initial results show that our optimization can gain 19.6% performance improvement compared to the naive configuration, even by tuning only 3 parameters.

VII. ACKNOWLEDGEMENT

This work was supported in part by NSF-1622292.

References

[1] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *CIDR*, volume 11, 2011.

[2] K. Wang and M. Khan. Performance prediction for apache spark platform. In *HPCC*. IEEE, 2015.

[3] H. Zhang, Z. Sun, Z. Liu, C. Xu, and L. Wang. Dart: A geographic information system on Hadoop. In *CLOUD*, pages 90–97. IEEE, 2015.

[4] L. Xu, M. Li, L. Zhang, A. Butt, Y. Wang, and Z. Hu. Memtune: Dynamic memory management for in-memory data analytic platforms. In *IPDPS*. IEEE, 2016.

[5] A. Paul, W. Zhuang, L. Xu, M. Li, M. Rafique, and A. Butt. Chopper: Optimizing data partitioning for in-memory data analytics frameworks. In *CLUSTER*, pages 110–119. IEEE, 2016.

[6] H. Zhang, H. Huang, and L. Wang. Mrapid: An efficient short job optimizer on Hadoop. In *IPDPS*. IEEE, 2017.

[7] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component sys*, 30:19, 2002.

[8] Spark management website. https://0x0fff.com/spark-memory-management/.

[9] S. Diersen, E. Lee, D. Spears, P. Chen, and L. Wang. Classification of seismic windows using artificial neural networks. *Procedia computer science*, 2011.

[10] P. Guo, H. Huang, Q. Chen, L. Wang, E. Lee, and P. Chen. A model-driven partitioning and auto-tuning integrated framework for sparse matrix-vector multiplication on gpus. In *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*. ACM, 2011.

[11] H. Zhang, L. Wang, and H. Huang. Smarth: Enabling multi-pipeline data transfer in HDFS. In *ICPP*, pages 30–39. IEEE, 2014.

[12] H. Huang, L. Wang, E. Lee, and P. Chen. An mpi-cuda implementation and optimization for parallel sparse equations and least squares (lsqr). *Procedia Computer Science*, 9:76–85, 2012.

[13] H. Huang, J. M. Dennis, L. Wang, and P. Chen. A scalable parallel lsqr algorithm for solving large-scale linear system for tomographic problems: a case study in seismic tomography. *Procedia Computer Science*, 18:581–590, 2013.