Data Model Optimization for Reducing Computational Cost at Apache Spark *

Rohyoung Myung, Han-Yee Kim, Sukyong Choi, Taeweon Suh, Heonchang Yu
Department of Computer Science and Engineering,
Korea University
145 Anam-ro, Seongbuk-gu, Seoul, Korea
{mry1811, hanyeemy, csukyong, suhtw, yuhc}@korea.ac.kr

ABSTRACT

As the performance of distributed parallel processing on big data is considered as a main concern, Apache Spark the most prevalent open source based distributed processing engine has triggering much interest for performance optimization. According to the user's purpose, Apache Spark provides diverse scope of system customization via system parameters. In general, main configurations of job optimization are: degree of parallelization, resource utilization (# of available cores, available memory, and etc), and caching. The researches which focus on tuning parameters related to distributed execution and which take fully advantage of memory caching have been proceeded. However researching data structures for optimizing the original distributed processing model of Spark and analyzing execution cost when applying them have not been considered. In this paper, we propose a data model that reduces the execution cost of parallel processing model on Apache Spark. In addition, by drilling down a specific example of distributed processing API, we prove effectiveness of the proposed data model compared to the previous one. Finally, we verify our scheme by adapting the proposed model on general big data application and prove the validity of it by scaling up its input data size and the number of executors.

KEYWORDS

Performance Optimization, Data Model, Apache Spark

1 INTRODUCTION

The dramatic advance of IoT and sensor system result in influx of big data these days. The flowed data which can be used for simple data analysis, even for complex deep learning, is so big that distributed parallel processing is generally conducted. In master-slave cluster environment, there are many things to consider: distributed processing model in cluster level, parallel processing in worker node level, the computing power of cluster system, methods for fault tolerance and so on.

The Apache Spark[1] open source distributed processing engine provides high-level API for analyzing big data so that it can support implementing applications based on Hadoop Map Reduce programming model[2] as well as various data analyzing tasks such as machine learning[3] and streaming data processing[4]. Also, Spark improves executing performance compared to the previous system[5] by adapting in-memory computing, lazy execution. Moreover, it supports fault-tolerance by utilizing its own distributed data structure called RDD[6]. Since Spark provides the libraries necessary for implementing data analysis and default system configurations for those applications, its performance fully depends on how users and developers tune system parameters, and utilize characteristics of the applications.

In this paper, we propose data model for reducing execution cost of distributed processing model on Spark. For proving that our scheme actually reduces execution cost, we analyze API generally used for data analysis to deduct a cost model and demonstrate theoretically that our scheme reduces amount of execution time significantly compared to the previous one. Finally we apply our scheme to real world application on spark cluster system and present the results that our scheme not only decrease execution time of the applications but also is independent to the size of input data and the number of working nodes.

2 BACKGROUND

Spark has a specific architecture for distributed parallel processing which is shown at Fig. 1. At first, users submit their application and its required input data to Spark Driver. For executing the application, Spark Driver transform submitted data to RDD and express executing procedure of the RDD into Directed Acyclic Graph(DAG).

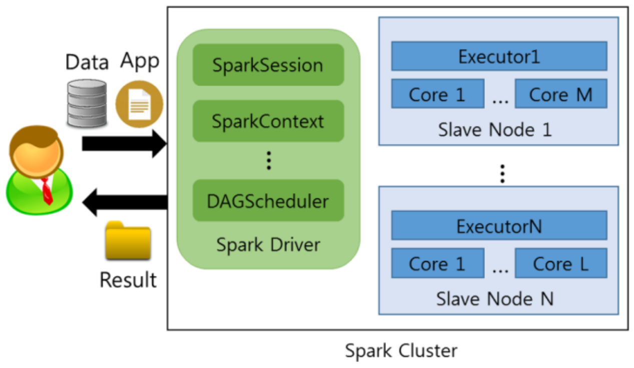

Figure 1: Spark cluster system architecture and interaction between user and Spark cluster

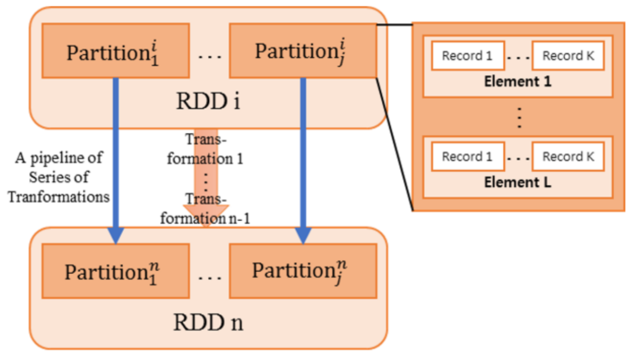

Figure 2: Pipelining procedure and inner structure of RDD

A RDD consists of at least one partition and a partition consists of at least one element which is depicted at Fig. 2. There are two relationships among RDDs: wide dependency and narrow dependency according to relationship between parent and child partitions.

Partitions which have narrow dependency can be pipelined as each of them doesn’t have any dependency with one another. However it is impossible to pipelining partitions which have wide dependency since a parent partition has correlation with multiple child partitions. Moreover, wide dependency causes shuffle whose execution time is deeply related to the computing power of executors, network transmission speed at that time, and occurrence of faults. Thus, it is hard to express wide dependency as deterministic model. So in this research, we only focus on narrow dependency.

3 DATA SUBMISSION MODEL

3.1 General API Processing Procedure

In the case of partitions which have narrow dependency, when a user submit application, Spark Driver divides the application into the multiple tasks according to RDD DAG and pipelines them. Then, it allocates the tasks evenly to executors and they serially execute whole records of partitions. Also, the executors create multiple threads according to the cores available and allocate assigned tasks to the cores evenly so that they execute the partitions in parallel.

At this time, each task is defined as transformation or action. The transformations or actions which are own API of Spark[7] are divided into two types. The first type such as ‘groupByKey’ and ‘HashPartitioning’ which has predefined functions execute task manually and the second type such as ‘map’ and ‘flatMap’ has inner functions which should be defined by Spark user. For example, when ‘map’ whose inner function is ‘split’ is processed, whole sequence of executing single RDD which is processed at each core is described at Fig. 3.

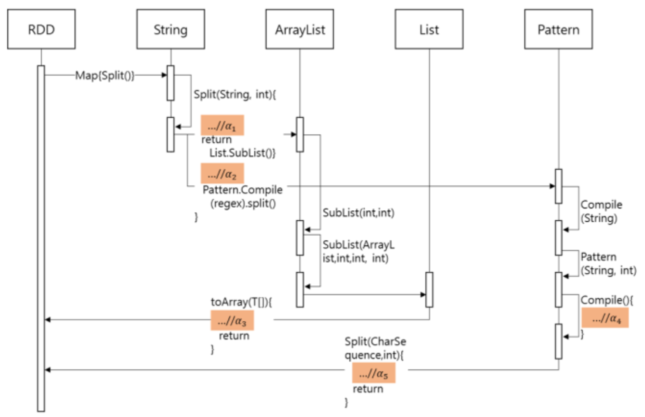

Figure 3: Sequence diagram when Spark API ‘map’ and its inner function ‘split’ is processed. The shaded parts(α_1-α_5) are the additional cost for each element.

3.2 Computational Cost Model

When an executor process single partition which is known as ‘task’ at Spark, total execution time is divided into two parts. The first part includes the cost that when an executor traverse whole records of single partition and apply its inner functions into each records iteratively. It is essential execution cost so that it can be described as deterministic model: execution time for applying inner function to single record multiplied by the number of whole records in a partition.

The second part is additional cost that not correlated to the number of records but correlated to the number of elements which is caused when an executor processes single element. For example, there are class generation, object initialization, and other additional methods. They are invoked when the executor processes an element and the cost(shaded parts in Fig. 3.: α_1 - α_5) can be modeled deterministically: total execution time of additional overheads multiplied by the number of whole elements in a partition. The definitions of all notations required for calculating the two costs are described at Table 1.

Table 1: Definition of the notations for computational cost model

Notation	*Meaning*
e	Total number of elements in a partition
r	The number of records in an element
l	The iterative overhead for executing single record
α	The additional overhead for executing single element
K	The number of records per element
C	Original total cost for executing single task
C’	Optimized total cost for executing single task

Equation 1. models the summation of the first and the second part. The first part is described as $r * l$ and the second part is described as $e * \alpha$

$$C = r * l + e * \alpha \tag{1}$$

Since the number of records in a partition is static, the execution cost of the first part is also static. However, when the number of records in an element is multiplied by K total cost can be described as Equation 2.

$$C' = r * l + \frac{1}{K} * e * \alpha \quad (2)$$

The first half of Equation 2 is same as the previous equation but the cost for the last half of the equation is reduced as the number of additional overheads is divided by K. However, notice that when K is increased, task size for executing an element is also increased which causes additional 'garbage collection overhead' which is proportional to K. Therefore, users should find optimal value for K by profiling their cluster system environment before applying our model.

4 EXPERIMENTAL RESULT

In the first experiment, we prove that the performance enhancement of our model is independent to input data size. So we fix the number of executor as 1 and increase the input data size from 1GB to 5GB. We selected application 'WordCount' since it is widely used benchmark for performance evaluation. Fig. 4. describes that when applying our model with proper value 'K', in our system: 100, the execution time of the application is reduced at most 3.5x more than the vanilla data model. Also, when the size of input data is linearly increased the execution time is increased linearly.

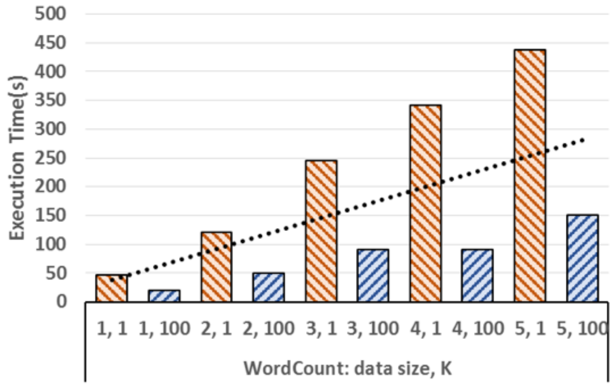

Figure 4: Execution time(s) of 'WordCount' when data size is linearly increased(1-5GB) with single executor and 'K': 1(vanilla model), 100(our scheme applied).

In the second experiment, we demonstrate that the performance enhancement of our model is scalable to the number of executors. Thus, we fix the size of input data as 5GB and increase the number of executors from 1 to 3. We used 'WordCount' as application for the same reason. Fig. 5. describes that when the number of executors are increased linearly, the execution time of the application is decreased linearly which is inversely proportional to the number of executors. It also describes that our model reduces the execution time at least 2x. Note that the value 'K' is 100 but should be carefully determined with regard to user's cluster system specifications.

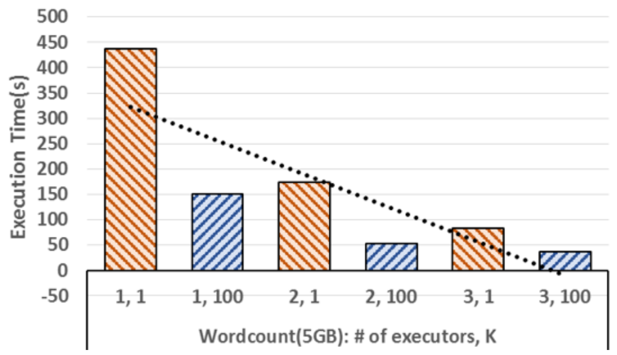

Figure 5: **Execution time(s) of 'WordCount' when the number of executors is linearly increased(1-3) with data size:5GB and 'K': 1(vanilla model), 100(our scheme applied).**

5 CONCLUSION

Our research analyzes structure details of Spark, its distributed data structure, and its details of distributed processing procedure. We proposed the data model which significantly reduces execution time of distributed parallel model. We demonstrate how our data model reduces execution cost by deterministically modeling the cost of processing multiple series of data in parallel. Finally, we provide experimental results for validating our model on real-world Spark cluster. Also, we prove that the data model reduces total execution time of application whether the size of data and the number of executors are increased or not. However, the main idea of our research: tuning the number of records per element requires another model for calculating overhead of 'garbage collection'. Currently, we model correlation among garbage collection overhead, cluster system specifications, task and partition size configuration, real size of single task for building integrated model which includes not only the data model of this paper but also supplement model considering garbage collection overhead and customizable system configurations.

ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIP) (No. 2017M3C4A7081955).

REFERENCES

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In Proc. USENIX Hot- Cloud, 2010
[2] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In Proc. USENIX OSDI, 2004
[3] "Spark MLlib" https://spark.apache.org/MLlib/
[4] "Spark Streaming." https://spark.apache.org/streaming/
[5] Shvachko, Konstantin, et al. "The hadoop distributed file system." Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on. IEEE, 2010.

[6] M. Zaharia et al., "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in Proceedings of the 9th USENIX Conference on NSDI. USENIX Association, 2012.
[7] "Spark Docs" https://spark.apache.org/docs/latest