Gray Box Modeling Methodology for Runtime Prediction of Apache Spark Jobs

Hani Al-Sayeh
Technische Universität Ilmenau
Ilmenau, Germany
hani-bassam.al-sayeh@tu-ilmenau.de

Kai-Uwe Sattler
Technische Universität Ilmenau
Ilmenau, Germany
kus@tu-ilmenau.de

Abstract—Nowadays, many data centers facilitate data processing and acquisition by developing multiple Apache Spark jobs which can be executed in private clouds with various parameters. Each job might take various application parameters which influence its execution time. Some examples of application parameters can be a selected area of interest in spatiotemporal data processing application or a time range of events in a complex event stream processing application. To predict its runtime accurately, these application parameters shall be considered during constructing its runtime model. Runtime prediction of Spark jobs allows us to schedule them efficiently in order to utilize cloud resources, increase system throughput, reduce job latency and meet customers requirements, e.g. deadlines and QoS. Also, the prediction is considered as important advantage when using a pay-as-you-go pricing model.

In this paper, we present a gray box modeling methodology for runtime prediction of each individual Apache Spark job in two steps. The first one is building a white box model for predicting the input RDD size of each stage relying on prior knowledge about its behaviour and taking the application parameters into consideration. The second one is extracting a black box runtime model of each task by observing its runtime metrics according to various allocated resources and variant input RDD sizes. The modeling methodology is validated with experimental evaluation on a real-world application, and the results show a high matching accuracy which reached 83-94% of the actual runtime of the tested application.

I. INTRODUCTION

Currently, many data centers provide services for customers to run big data processing jobs. These jobs collect required files from long-term archiving systems, load them into local HDFS [1], and process them using a distributed processing framework such as Apache Spark [2]. To overcome many challenges such as data confidentiality and system safety, data centers develop and test their own big data jobs, provide web access for users, e.g. CODE-DE web platform [3], and allow them to run these jobs after selecting various application parameters. Some general purpose Spark jobs serve a lot of application domains and different customers. That is, the inputs of these jobs include a large number of application parameters.

Some data centers collect raw data from multiple sources, like sensors and systems logs, and perform data cleaning and early stage processing operations before storing it into long-term achieving systems. Meanwhile, important meta data that describes the data content, such as the number of interesting patterns, data skew factors and many others, can be extracted and stored.

The runtime of Spark jobs is influenced by many types of application parameters. Some of these parameters affect the RDD [4] sizes, e.g. conditions that affect the selectivity of filter operators and the cardinality of join operators. Some others affect the RDD lineage or determine the used processing algorithms. Thus, it is important to consider these parameters, in addition to the allocated resources, during the construction of the runtime prediction model for such Spark jobs.

Runtime prediction of Spark jobs is a hot research topic for several reasons. It allows online schedulers to utilize resources efficiently with minimal operational costs. Also, it facilitates applying new scheduling strategies. For example, first in first out (FIFO) and the earliest deadline first scheduling approaches are always applicable without the need for runtime prediction. However, this will not be the case with the shortest job first strategy. Moreover, runtime prediction is an essential advantage when using a pay-as-you-go pricing model [5]. With an accurate runtime prediction, the cost for running Spark jobs in public cloud can be estimated and, thus, selecting suitable cloud computing resources or even deciding whether to run the job in public cloud or not can be taken in advance.

Extracting a prediction runtime model for a Spark job is nontrivial [6] and hard to achieve, because it depends on multiple complex factors such as the input data size, data content, allocated processing resources (like CPU speed, memory capacity, disk I/O speed and network bandwidth), cluster usage, and the configuration parameters of the Apache Spark platform (more than 210 parameters [7]). Considering tens of application parameters and including them in the runtime model makes constructing it much more complicated.

In some use cases, data centers run for multiple years the same generic Spark job with multiple application parameters. Therefore, the time required to construct its runtime model is not critical. On the contrary, when a user runs a job, predicting its runtime should be taken in a glance, especially, for short-running jobs when sampling increases job latency tremendously and wastes cluster resources.

To the best of our knowledge, all presented models and modeling methodologies for Spark jobs runtime prediction just consider data size, Spark configuration and allocated resources.
without taking the application parameters into account which cause a huge runtime variance of the same Spark jobs. Without considering these parameters in some cases, the runtime prediction model of Spark jobs is not realistic.

Extracting this model, that takes all application parameters into account, is nearly impossible by just relying on experimental observations. On the other hand, without experimental observations, the runtime model will be not realistic. To overcome this challenge, we present in this study a gray box modeling methodology that consists of the following two steps:

1) White box modeling: In this step, we study the impact of each individual application parameter on the RDD size, depending on prior knowledge of how the Spark job exactly behaves. Based on this model, we can predict the influence of modified application parameter values on the RDD size during each stage.

2) Black box modeling: In this step, we extract the runtime model of each task, which is the smallest execution unit in the stage, depending on observing its runtime metrics, that are extracted during performing modeling purpose experiments. This model takes the RDD partition size extracted from the white box model and the allocated resources as parameters and returns the tasks’ predicted runtime.

The remainder of the paper is organized as follows. Related work is discussed in Sect. 2. Sect. 3 discusses background concepts including Spark job execution model. Sect. 4 explains the gray box modeling methodology which is presented in this work for Spark jobs runtime prediction. Sect. 5 describes in details how to construct a gray box model for a Spark job. Experimental results are presented in Sect. 6. Finally, a conclusion and an outlook are presented in Sect. 7.

II. RELATED WORK

Recently, many works have been proposed to observe, analyze, and predict the runtime performance of large-scale data processing platforms such as [8]–[13]. To predict the runtime of MapReduce jobs, Starfish [10] introduced a self-tuning framework on top of Hadoop that applies an analytical approach to observe and analyze jobs runtime metrics by running them on data fraction and optimize system performance by tuning its configuration options. PREDicT proposed in [8] is an experimental methodology to predict the runtime of a class of iterative algorithms like graph processing, semi-clustering, and ranking implemented on the Hadoop MapReduce platform [14]. Its main idea is to predict the number of iterations and the runtime model of each one depending on sample runs. A bounds-based performance model is presented in [9] to predict the execution time of MapReduce jobs running on heterogeneous clusters. In [11], the authors introduced a simulation driven model to predict the execution time of Spark jobs by simulating their execution on data fraction and collecting their detailed execution metrics like memory consumption, I/O costs, and runtime. Another approach is presented in [15] which models the memory behavior of Spark job based on a mixture of experiments. Based on the extracted models, runtime prediction models are presented and a task co-location strategy is proposed to improve system throughput. Ernest [13] is a large-scale performance prediction framework that presents a general runtime model for Spark jobs. For each individual Spark job, Ernest runs it with various configurations and data fractions to extract its runtime model with adequate coefficients. Doppio [12] proposed runtime prediction model for Spark jobs by studying the I/O impact on the in-memory cluster computing frameworks and identified I/O overhead as a dominant bottleneck in such frameworks.

Beside these studies, many others focus on constructing runtime models based on statistics (prior knowledge about the selectivity and cardinality of query operators) [16]–[18]. These statistics can be obtained by scanning the tables and refining them over the time. Also, Catalyst query optimizer for Spark SQL jobs [19] is used to analyze and optimize query logical plan by transforming its tree into an equivalent optimized one using rule-based and cost-based optimization.

We found from the previous works, that the presented models focus on the impact of data size, platform configuration settings, and allocated resources like memory consumption, I/O overhead, and network bandwidth. However, non of them considered the application parameters which may affect the runtime performance significantly. In our work, we build a modeling methodology to predict the runtime by taking into consideration not only the previous factors but also the application parameters as well. Also, even though query optimization techniques are useful for predicting traditional data-intensive jobs performance, advanced datacomputing-intensive Apache Spark applications [20]–[22] differ and require additional modeling efforts, especially, when dealing with arbitrary black box operators.

III. BACKGROUND

In the following, we will briefly discuss Big Data storage and processing platforms, HDFS and Spark respectively.

A. HDFS

HDFS [23], [24] stores huge files by fragmenting them into blocks which are replicated and stored on multiple machines. HDFS contains a single NameNode and multiple DataNodes. Each DataNode stores data blocks. The NameNode stores data allocation catalog and is responsible for handling users operations on files such as create, delete and rename.

B. Spark

Apache Spark is a general purpose large-scale data processing engine aimed to perform fast distributed data processing [25]. Relying on its powerful in-memory computing model, which is built upon the Resilient Distributed Datasets (RDDs) abstraction [4], Spark outperforms disk-based engines like Hadoop up to 100x.

It provides a general execution model that optimizes data processing through a directed acyclic graph (DAG) of arbitrary operators. Each group of operators that do not require data
shuffling is performed in a stage. Thus, each Spark job consists of one or more sequential stage(s). Each stage consists of multiple homogeneous tasks that run in parallel and process various RDD partitions of the same data source. The number of tasks per stage equals the source file blocks number, by default. Also, it can be explicitly defined by programmers. Fig. 1 shows an example of Spark jobs life cycle.

![Fig. 1: A Spark job that contains n stages and runs in four cores. Its input file is composed of eight blocks.](image)

Fig. 1: A Spark job that contains n stages and runs in four cores. Its input file is composed of eight blocks.

As presented in [11], a Spark job consists of multiple sequential stages. The first stage reads data blocks from HDFS and creates an HadoopRDD by loading the source file blocks into memory (as RDD partitions), then it launches a task to process each RDD partition. The number of tasks in the first stage is determined by the number of input file blocks which is computed as follows:

\[
\text{#TasksPerStage} = \left\lceil \frac{\text{FileSize}}{\text{BlockSize}} \right\rceil
\]

The file block size is determined during storing the file in HDFS by setting the `dfs.block.size` option. By default, the number of tasks per stage remains the same in all stages. The level of stage execution parallelism is determined by the number of cores allocated for running the Spark job, which is determined as follows:

\[
\text{#AllocatedCores} = \text{#Executors} \times \text{#CoresPerExecutor}
\]

During the submission of the Spark job, the number of executors is determined by the `num-executors` option, while the `executor-cores` option determines the number of cores per executor. Theoretically, the total execution time of a stage is as follows:

\[
\text{Runtime(Stage}_i\text{)} = \text{Runtime(TaskOfStage}_i\text{)} \times \left\lceil \frac{\text{#TasksPerStage}}{\text{#AllocatedCores}} \right\rceil
\]

Practically, the program driver distributes tasks among cores in a non-balanced way. The total run time of a Spark job is:

\[
\text{Runtime(job)} = \text{Startup} + \sum_{i=1}^{\text{#Stages}} \text{Runtime((Stage}_i\text{)} + \text{Cleanup}
\]

IV. GRAY BOX MODELING METHODOLOGY

To build a comprehensive runtime model for Spark jobs, firstly, we have to study the relationship between its application parameters and the RDD sizes to extract the white box model. Then, we develop the runtime model of each task individually regarding its input RDD partition size and its allocated resources to extract its black box runtime model. Fig. 2 shows a general overview on the gray box modeling concept.

Building the white box model relies on prior knowledge of how the Spark job behaves in details while processing input data. Therefore, the prior knowledge is considered as follows:

- Knowledge of processing workflow (DAG) that Spark job developers know in advance. With this knowledge, the number of stages and shuffling operations are known in advance with clear understanding on stages internals.
- Knowledge of data which can be collected from meta data files constructed during early stage processing phase. This knowledge is essential to predict the selectivity of job operators.

A. White Box Model

As mentioned previously, a Spark job consists of multiple stages running sequentially. Each stage takes an RDD as input (RDDin) and produces another one as output (RDDout). In each individual stage, the RDDout size is influenced by the RDDin size and the behaviour of the stage’s operators lineage. In order to predict the RDDout size, in terms of the number of tuples and the tuple size, prior knowledge of the RDDin size and the selectivity of each operator are essential. The operators that do not affect the RDD size, are not considered in the white box model. On the other hand, all operators that affect the RDD size shall be considered even if their behaviour is not affected by any application parameter.
Tuning these parameters affect the RDDout size. Assuming that the selectivity of each filter operator and the size of the projected columns are known, the resulting RDDout size would be predictable. Hence, the impact of any change in RDDin size and the values of $p1$, $p2$, and $p3$ on the RDDout size is predictable by estimating the output RDD size of each operator one by one sequentially (RDD1 and RDD2 and RDDout). While the resulting RDDout will be an input for later stages, any change in $p1$ value will be propagated to all following stages, and its influence on their RDDin sizes will be predictable. To predict the partition size of any RDD, the following two equations are used:

$$\#\text{RDD Partition Tuples} = \frac{\#\text{RDD Tuples}}{\#\text{TasksPerStage}}$$

$$\#\text{RDD Partition TupleSize} = \#\text{RDD TupleSize}$$

While application parameters vary according to their influence on RDD sizes, they are categorized as follows:

1) **Condition Parameters:** Tuning these parameters affects the selectivity of filter operators and the cardinality of joins. As a result, the number of RDD tuples will vary regarding various selected condition values. Taking the filter operator as an example, predicting its resulting RDD tuple number relies on its selectivity, which is known in advance and takes relevant application parameters into account like $p1$. In the example shown in Fig. 3, RDD1 tuples number, resulting from Filter1 operator, is predicted as follows:

$$\#\text{RDD1 Tuples} = \#\text{RDDin Tuples} \times \text{Filter1 Selectivity}(p1)$$

In many cases, the selectivity of some operators is influenced by multiple parameters indirectly. Let us assume that the selectivity of Filter2 is influenced by param1 and param2, thus, its selectivity model shall consider the combination of both parameters as follows:

$$\#\text{RDD2 Tuples} = \#\text{RDD1 Tuples} \times \text{Filter2 Selectivity}(p1, p2)$$

2) **Field Selection Parameters:** Tuning these parameters affects RDD tuple sizes. Some Spark operators like map and projection (in Spark SQL) change tuple sizes. Predicting tuple size is important as long as it plays an important role in estimating the required memory size, shuffling runtime, tuple compression metrics, and other performance aspects.

3) **Parallelism Parameters:** Some Spark shuffling operators like repartition, coalesce, join and reduceByKey modify RDD partitions number, which influences tuples number of each RDD partition in later stages, and their tasks runtime as well.

4) **Workflow Control Parameters:** Some application parameters, like selecting processing algorithm or the number of iterations in the PageRank application, affect the RDD lineage and therefore the total job runtime.

### B. Black Box Model

While a task is the smallest processing unit in each Spark job, building a Spark job runtime model relies on observing the runtime metrics of each unique task and extracting its runtime model. The extracted runtime model of each task includes task computing, serialization and deserialization times. Also, it includes data shuffle read and write, and disk I/O operations as well. The aimed model shall consider the allocated resources for the task and the size of its input RDD partition. The main allocated resources considered during the black box modeling are the allocated memory size which affects data spilling rate into disk and the number of allocated executor cores that run concurrently and compete for shared executor resources like disk, memory and network. Therefore, the considered input of each task’s runtime model contains the executor-memory and the executor-cores Spark options besides its input RDD partition size.

To extract the runtime model of each task, we run the job multiple times with variant allocated memory, cores per executor and RDD partition sizes and analyze its runtime metrics using regression analysis statistical methods [26]. The runtime metrics of tasks can be collected using SparkListener [27].

Despite of running same tasks on same allocated resources with identical input RDD partitions (the same size and content), tasks runtimes vary significantly. This is caused by straggler tasks [28] that affect the prediction model. However, analyzing the causes of straggler tasks and their impact is out of the scope at this stage of our research. To deal with this uncertainty behaviour, we increase the tasks number in each job, by increasing data size, and take their average runtimes.

### V. BUILDING A GRAY BOX MODEL

As mentioned previously, to build a gray box model for any Spark job, two main modeling steps shall be performed. The first one is building the white box model that is aimed to predict the RDD sizes in each stage based on the selected application parameters and prior knowledge of how the target job behaves. The second one is building a black box model for each unique task considering its input RDD partition size and the allocated resources. In the following, a WordCount Spark job is taken as an example to illustrate the gray box model construction steps.

The aim of this section is to show in details how to construct gray box models for Spark jobs. Therefore, we select a simple real-world use case with one application parameter for illustration purposes. However, this concept is applicable for complex applications as well.
A. WordCount

Assuming the following use case. Users are allowed to run WordCount Spark jobs via a web portal after selecting a specific group of letters that the result words shall not start with. For simplicity, the average number of words per line in the source file is known in advance.

1) The Application Parameter: In this use case, there is one application parameter which is a list of letters that the WordCount Spark job takes as a parameter. As shown in Fig. 4, to exclude the words that do not start with the selected letters, one filter operator is injected in the first stage operators lineage.

Fig. 4: A WordCount lineage that includes an application purpose filter.

2) The White Box Model: To build the white box model, RDD sizes have to be predicted regarding the selected letters. The selectivity models of all other RDD sizes in this lineage are static and can be observed during running sample tests. As an exception, the selectivity model of the filter is dynamic and influenced by the selected letters. The white box model should predict accurately the number of FilteredRDD tuples and their average size according to all possible \(26\) letter combinations.

To predict the number of tuples in FilteredRDD, the frequencies of the first letters of a word in English language have to be known in advance. Fig. 5 shows relative frequencies of the first letters of a word in English language. On the other hand, to predict the average tuple size in FilteredRDD, another useful statistics can be collected to predict word length according to its starting letter. Fig. 6 shows extracted statistics from a 16MB MySQL English dictionary database [29].

For example, if a user selects \{'a, f, h, n\}\) letters as an application parameter, the FilteredRDD size will be as follows:

- Tuples number ratio:
  \[
  1 - (11.68\% + 4.03\% + 4.20\% + 2.28\%) = 79.81\%
  \]

- Tuple average size:
  \[
  \sum_{i=1}^{26} frequency(i) \times WordAverageSize(i) \quad 79.81\%
  \]

Which equals 7.73 bytes.

The selectivity model can be enhanced by enriching it with the most frequently used words statistics and many other useful English words distribution studies. Even though this increases the model accuracy, modeling efforts and time are also increased. In this use case, the selectivity model of each operator is as follows:

- filter: is dynamic as described previously.
- flatMap: This operator splits the sentence into a list of words. While we assumed that the average number of words per line \(T\) is known, the selectivity of this operator equals \(T \times 100\%\).
- mapToPair: This operator maps each word to a key value pair. Therefore, its selectivity \(100\%\).
- reduceByKey: The resulted RDD tuples number equals the number of unique words in the source file. It can be estimated even without knowing the number of tuples entered this operator. Therefore, a rough estimation of the unique words number in the source file would be enough.

After defining the selectivity model of each operator, the only missing part that we need to predict is the initial RDD size. Its average tuple size is known in our case which equals the word length average \(8.12 + 1\) (new line delimiter) bytes. The number of its tuples equals the source file size divided by 9.12.

As mentioned previously, the RDD tuple size prediction is important because it influences memory consumption and shuffling operation. The data structure used to store tuple content called DataFrame. So, the actual DataFrame size shall be predicted instead of the data length. In this case, a prediction model shall be extracted, by experiments, to predict
the DataFrame size regarding the actual data size (word length in our example).

In this use case, the prior knowledge is considered as knowing the number of stages, the static/dynamic selectivity model of each operator regarding the selected application parameters, operators lineage and data content and distribution. Based on this, the RDD partition size during each stage would be predictable when the HDFS data block size and the application parameters are known.

3) The Black Box Model: As mentioned previously, building a runtime model of each task is done by observing its runtime metrics with variant RDD partition sizes and allocated resources. Therefore, we shall run the WordCount job multiple times with several configurations. Then, based on the observed tasks’ runtime, their model can be extracted. These jobs shall be identical in terms of inserted application parameters, thus the same group of letters shall be passed as an input during each run. Running these jobs is performed based on the following three dimensions:

- **RDD partition size:** The simplest way to modify it is to change the HDFS block size. In some cases, especially, when the RDD tuple average size varies significantly, this dimension shall be divided into two dimensions. One is for the tuples number and the other is for the average tuple size.
- **Allocated cores per executor:** It is defined by the `--executor-cores` Spark configuration option.
- **Allocated memory per core:** It is defined by the `--executor-memory` Spark configuration option.

Each dimension shall at least contain three values like $\{32$ MB, $64$ MB and $128$ MB block size$\} \times \{1$, 2 and 4 cores per executor$\} \times \{1$GB, 2GB and 4GB allocated memory$\}$. To cover all probabilities, we shall run the job 27 times. Then, the average runtime model of each task can be extracted by analyzing the runtime logs provided by the SparkListener. As mentioned previously, the RDD resulted by the reduceByKey operator is static, because its tuples represent unique words. In this case, changing the HDFS block sizes will not influence its size. To overcome this challenge, we can edit the data content or tune the application parameter value and perform a second modeling round to extract the runtime model of the second stage.

Sometimes predicting the RDD partition size that each task processes is not enough and further model enhancements shall be performed to improve the model accuracy. For example, it is clear that RDD2 size influences the runtime of stage 2. But it does not in case of stage 1, because both RDD1 and filtered-RDD sizes affect its runtime. To overcome this challenge, the first stage can be divided into two parts, as shown in Fig. 4. In addition, a runtime model shall be extracted for each part individually during building the black box model. This can be achieved by injecting a modeling purpose `mapPartitions` operator after the filter. The aim of this injected operator is profiling the task runtime by sending the current timestamp to a central storage unit prepared for modeling purposes [31]. With this approach, it is possible to profile the runtime of the filter itself and thus extract its runtime cost model. Even though the total job runtime is decreased when the user selects more letters, the time required to perform filtering operation is increased. In this example, to filter 25 letters (in 1GB HDFS block) it takes just 2 milliseconds more than the time required to filter only one. Therefore, it is neglected.

VI. EVALUATION

To evaluate the proposed modeling methodology, we performed experiments on our Spark cluster which consists of 16 nodes with the following specifications: Intel Core i5 2.90 GHz, 16 GB DDR3 RAM, 1 TB disk, 1 GBit/s LAN. Also, the cluster runs Hadoop 2.7, Spark 2.0.1, Java 8u102, and Apache Yarn on top of Hadoop Distributed File System (HDFS).

We evaluated the gray box modeling methodology on the WordCount job use case described in the previous section. This job is tested with an assumption that each line in the source file contains up to twenty words (10 per line on average).

To evaluate the white box model, we ran the job with various application parameter values 26 times. Starting by passing $J1$ an empty list as a parameter, each later job takes an additional letter to the list, e.g. $J2$ {'a'}, $J3$ {'ab'}, $J4$ {'abc'} and so.

First of all, we built the white box model to predict the RDD sizes over the operators lineage. Fig. 7 shows the white box model experimental results.
The initial RDD size is not influenced by the application parameter. That’s why, in all experiments it shows the same accuracy average which reaches 99.45%. Even though the reduceByKey resulting RDD is not an input for an individual task, we included it to evaluate the model accuracy in the final RDD.

To evaluate the black box model, the job executed 20 times with randomly selected variant data sizes, allocated memory and allocated cores per executors. The results are shown in Fig. 8.

To evaluate the black box model, the job executed 20 times with randomly selected variant data sizes, allocated memory and allocated cores per executors. The results are shown in Fig. 8.

Finally, we executed 50 WordCount jobs with arbitrary application parameter values, allocated memory, allocated cores per executor on various source files (3-6 GB) stored in the HDFS with various block sizes (64 and 128 MB). Fig. 9 shows the comparison between the actual and the predicted runtimes of each job. The matching accuracy of the average runtime model reached 83-94% of the actual runtime for the tested jobs.

To study the impact of the unbalanced tasks distribution among cores on our modeling methodology, we ran additional experiments that shows the following outcomes:

- Tasks distribution among cores is uniform, and the variance between the minimum and the maximum allocated tasks per core did not exceed 12% even when the average number of tasks per core was 40.
- There is no negative impact on our model accuracy. Theoretically, non-equal tasks distribution shall reduce the model accuracy. But on the other hand, this behaviour reduces the impact of tasks runtime variance and eliminates the impact of stragglers.

In general, the high matching accuracy of the presented gray box modeling methodology shows that applying it for many applications will improve Spark jobs prediction accuracy, especially, that all of them behave similar to the selected WordCount use case, e.g. DAG, shuffling, sequential run of stages, tasks parallelism, etc.

VII. CONCLUSION & OUTLOOK

In this paper, we presented a gray box modeling methodology for runtime prediction of Apache Spark jobs. The methodology is based on prior knowledge of their behaviours and takes into consideration the application parameters to predict changes in RDD sizes in each stage (white box model). In addition, we extract a runtime model for each task by observing runtime metrics with various allocated resources and variant input RDD partitions sizes (black box model). The modeling methodology is validated with experimental evaluation on a real-world application. The results showed a high matching accuracy which reached 83-94% between the average runtime model and the actual runtime of the tested application.
In future, we plan to develop a scheduling methodology based on the extracted runtime models in this study to improve resource utilization and increase the overall system throughput.

Additionally, the extracted runtime models in this study will be used in a simulation based prediction approach that we will propose to analyze the current system performance (e.g., throughput, latency average and the utilization of cluster resources) and answer what-if questions about system configuration.

ACKNOWLEDGMENTS

This work was partially funded by the German Aerospace Center (DLR).

REFERENCES

[1] D. Borthakur et al., “HDFS’s architecture guide,” Hadoop Apache Project, vol. 53, pp. 1–13, 2008.
[2] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. I. Franklin et al., “Apache spark: a unified engine for big data processing,” Communications of the ACM, vol. 59, no. 11, pp. 56–65, 2016.
[3] T. Storch, C. Reck, S. Holzwarth, and V. Keuck, “Code-de-the german operational environment for accessing and processing copernicus sentinel products,” in IGARSS 2019-2018 IEEE International Geoscience and Remote Sensing Symposium. IEEE, 2018, pp. 6520–6523.
[4] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012, pp. 2–2.
[5] A. Gohad, N. C. Narendra, and P. Ramachandran, “Cloud pricing models: A survey and position paper,” in 2013 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM). IEEE, 2013, pp. 1–8.
[6] K. Wang, M. M. H. Khan, N. Nguyen, and S. Gokhale, “Modeling interference for apache spark jobs,” in Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on. IEEE, 2016, pp. 423–431.
[7] “Apache spark official website,” https://spark.apache.org/docs/latest/configuration.html, [Online; accessed 22-February-2019].
[8] A. D. Popescu, A. Balmin, V. Ercegovac, and A. Ailamaki, “Predicting time to write in a large-size sequence of analytics,” Proceedings of the VLDB Endowment, vol. 6, no. 14, pp. 1678–1689, 2013.
[9] Z. Zhang, L. Cherkasova, and B. T. Loo, “Performance modeling of mapreduce jobs in heterogeneous cloud environments,” in Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on. IEEE, 2013, pp. 839–846.
[10] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, “Starfish: a self-tuning system for big data analytics.” in Cidr, vol. 11, no. 2011, 2011, pp. 261–272.
[11] K. Wang and M. M. H. Khan, “Performance prediction for apache spark platform,” in High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cybersecurity and Safety (CSS), 2015 IEEE 12th International Conference on Embedded Software and Systems (ICCESS), 2015 IEEE 17th International Conference on. IEEE, 2015, pp. 166–173.
[12] P. Zhou, Z. Ruan, Z. Fang, M. Shand, D. Roazen, and J. Cong, “Doppio: Io-aware performance analysis, modeling and optimization for in-memory computing framework,” in Performance Analysis of Systems and Software (ISPASS), 2018 IEEE International Symposium on. IEEE, 2018, pp. 22–32.
[13] 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), 2016, pp. 363–379.
[14] “Apache hadoop official website,” https://hadoop.apache.org/, [Online; accessed 22-February-2019].
[15] V. S. Marco, B. Taylor, B. Porter, and Z. Wang, “Improving spark application throughput via memory aware task co-location: A mixture of experts approach,” in Proceedings of the 18th ACM/SIGAPP Middleware Conference. ACM, 2017, pp. 95–108.
[16] S. Chaudhuri, V. Narasayya, and R. Ramamurthy, “Estimating progress of execution for sql queries,” in Proceedings of the 2004 ACM SIGMOD international conference on Management of data. ACM, 2004, pp. 803–814.
[17] K. Morton, M. Balazinska, and D. Grossman, “Paratimer: a progress indicator for mapreduce jobs,” in Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010, pp. 507–518.
[18] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, “Access path selection in a relational database management system,” in Proceedings of the 1979 ACM SIGMOD international conference on Management of data. ACM, 1979, pp. 23–34.
[19] M. Armbrust, R. Xin, X. Meng, T. Kafan, M. J. Franklin, A. Ghodsi et al., “Spark sql: Relational data processing in spark,” in Proceedings of the 2015 ACM SIGMOD international conference on Management of data. ACM, 2015, pp. 1383–1394.
[20] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen et al., “Mlib: Machine learning in apache spark,” The Journal of Machine Learning Research, vol. 17, no. 1, pp. 1235–1241, 2016.
[21] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, S. Senior, P. Tucker, K. Yang, Q. V. Le et al., “Large scale distributed deep networks,” in Advances in neural information processing systems, 2012, pp. 1223–1231.
[22] F. A. Nothaft, M. Massie, T. Danford, Z. Zhang, U. Laserson, C. Yeksiyan, J. Kottalam, A. Ahuja, J. Hammerbacher, M. Linderman et al., “Rethinking data-intensive science using scalable analytics systems,” in Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, 2015, pp. 631–646.
[23] K. Shivachko, H. Kuang, S. Radia, and R. Chansler, “The hadoop distributed file system,” in Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on. Ieee, 2010, pp. 1–10.
[24] T. White, Hadoop: The definitive guide. “ O’Reilly Media, Inc.”, 2012.
[25] D. García-Gil, S. Ramírez-Gállego, S. García, and F. Herrera, “A comparison on scalability for batch big data processing on apache spark and apache flink,” Big Data Analytics, vol. 2, no. 1, p. 1, 2017.
[26] Wikipedia contributors, “Regression analysis — Wikipedia, the free encyclopedia,” 2019, [Online; accessed 22-February-2019]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Regression_analysis&oldid=882057523
[27] “Apache spark: Monitoring and instrumentation,” https://spark.apache.org/docs/latest/monitoring.html, [Online; accessed 22-February-2019].
[28] R. Xin, P. Deyhim, A. Ghodsi, X. Meng, and M. Zaharia, “GraySort on apache spark by databricks,” GraySort Competition, 2014.
[29] “Mysql english dictionary,” https://sourceforge.net/projects/mysqlenglishdictionary/, [Online; accessed 22-February-2019].
[30] Wikipedia contributors, “Letter frequency — Wikipedia, the free encyclopedia,” 2019, [Online; accessed 22-February-2019]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Letter_frequency&oldid=870957806
[31] S. Hagedorn and K.-U. Ritter, “Cost-based sharing and recycling of (intermediate) results in dataflow programs,” in European Conference on Advances in Databases and Information Systems. Springer, 2018, pp. 185–199.