Al-Sayeh, Hani; Hagedorn, Stefan; Sattler, Kai-Uwe:

A gray-box modeling methodology for runtime prediction of Apache Spark jobs

Original published in: Distributed and parallel databases. - New York, NY [u.a.] : Consultants Bureau. - 38 (2020), 4, p. 819-839.

Original published: 2020-03-10

ISSN: 1573-7578

DOI: 10.1007/s10619-020-07286-y

[*Visited:* 2020-11-17]

Distributed and Parallel Databases (2020) 38:819–839
https://doi.org/10.1007/s10619-020-07286-y

A gray-box modeling methodology for runtime prediction of Apache Spark jobs

Hani Al-Sayeh[1] · Stefan Hagedorn[1] · Kai-Uwe Sattler[1]

Published online: 10 March 2020

Abstract
Apache Spark jobs are often characterized by processing huge data sets and, therefore, require runtimes in the range of minutes to hours. Thus, being able to predict the runtime of such jobs would be useful not only to know when the job will finish, but also for scheduling purposes, to estimate monetary costs for cloud deployment, or to determine an appropriate cluster configuration, such as the number of nodes. However, predicting Spark job runtimes is much more challenging than for standard database queries: cluster configuration and parameters have a significant performance impact and jobs usually contain a lot of user-defined code making it difficult to estimate cardinalities and execution costs. In this paper, we present a gray-box modeling methodology for runtime prediction of Apache Spark jobs. Our approach comprises two steps: first, a white-box model for predicting the cardinalities of the input RDDs of each operator is built based on prior knowledge about the behavior and application parameters such as applied filters data, number of iterations, etc. In the second step, a black-box model for each task constructed by monitoring runtime metrics while varying allocated resources and input RDD cardinalities is used. We further show how to use this gray-box approach not only for predicting the runtime of a given job, but also as part of a decision model for reusing intermediate cached results of Spark jobs. Our methodology is validated with experimental evaluation showing a highly accurate prediction of the actual job runtime and a performance improvement if intermediate results can be reused.

Keywords Big data · Runtime prediction · Modeling

✉ Hani Al-Sayeh
hani-bassam.al-sayeh@tu-ilmenau.de

✉ Stefan Hagedorn
stefan.hagedorn@tu-ilmenau.de

Kai-Uwe Sattler
kus@tu-ilmenau.de

[1] Technische Universität Ilmenau, Ilmenau, Thüringen, Germany

1 Introduction

Big data platforms such as Hadoop, Spark or Flink are mainly used to process and analyze huge volumes of data resulting in runtimes of minutes or even hours. For many users, the prediction of the expected runtime of such jobs would be very helpful. Based on this information, cluster resources could be allocated, scheduling of jobs can be improved, and costs for cloud deployment (e.g., in the form of a what-if analysis) can be estimated.

Though predicting the runtime of arbitrary Spark jobs seems to be nearly impossible simply due to numerous parameters and user-written code, there are scenarios where several opportunities for collecting the information necessary for a good prediction model exist. Often, the development of Spark programs is an explorative task and programs written once are executed multiple times. Consider the following scenarios:

1. A data scientist loads several data files and performs some basic preprocessing and transformation tasks. After looking at the results she chooses the next processing operators to apply. Then, she investigates the results again and continues to modify operators and adds new operators until the original task is solved and the final result is produced.
2. After a (sometimes complex) Spark program has been developed, it is executed again and again with varying parameter values such as thresholds for filters, attributes to extract, or even flags to apply different algorithms/processing steps.
3. Often, different data scientists work with the same input data. Thus, they all have to apply the same preprocessing steps (data cleaning, transformation, etc.) and the different jobs share common sub-tasks which consume computing resources although they all produce the same result. Unfortunately, this sharing (e.g., by materializing intermediate results) is not done automatically and transparently.

From these scenarios we can conclude that (parts of) the same programs are run over and over again and that they often take application parameters that affect the runtime of the resulting Spark jobs. By monitoring the execution of these jobs and collecting information we can try to construct a prediction model. Furthermore, to reduce the runtime and the resource utilization of these jobs, intermediate results of the operations should be shared among different jobs, i.e., an intermediate result that has been computed before and is needed frequently should be materialized in one, and be reused in other jobs. To shift as much responsibilities from the data scientist into the processing engine, the decision for materialization, the actual materialization on persistent storage as well as the reuse must be performed transparently without any user interaction. For this, a decision model based on the costs of operators is needed and to predict these costs, it is imperative to consider also the application parameters beside the allocated resources.

Building a prediction runtime model for a Spark job is non-trivial [27], because it depends on numerous factors such as the input data cardinalities, content and distribution of data, allocated processing resources, current cluster utilization, and the configuration parameters of the Apache Spark platform (more than 210 parameters [2]). Taking application parameters into account adds to the complexity of building the runtime model. To the best of our knowledge, all previous modeling methodologies for runtime prediction of Spark jobs use only the data size, Spark configuration and allocated resources without the application parameters. This results in unacceptable variance between the predicted and the actual runtimes. To address this problem and improve the prediction accuracy, we present in this study a gray-box runtime modeling methodology of Apache Spark jobs that incorporates the application parameters. This methodology consists of the following two steps:

1. White-box modeling: We study the influence of each application parameter on the RDD cardinality while varying the application parameter values and estimate the RDD cardinality for each operator.
2. Black-box modeling: We run experiments with the estimated RDD partition sizes and the allocated resources to obtain the runtime metrics for each task, from which we then make observations to build our prediction model.

The white- and black-box modeling complement each other, resulting in a two-step gray-box modeling methodology. The white-box model predicts the RDD sizes with respect to the application parameters. The predicted RDD sizes are then taken as input to the black-box model for predicting the runtime of tasks and hence the job runtime.

The remainder of the paper is organized as follows: Sect. 2 discusses related work. Section 3 presents background concepts including the Spark job execution model. In Sect. 4 we explain our gray-box modeling methodology for runtime prediction, while we describe in Sect. 5 in detail how to construct a gray-box model for a Spark job. As one of the use cases of this model we describe the materialization and reuse of intermediate job results in Sect. 6. Results of our experimental evaluation are presented in Sect. 7. Finally, Sect. 8 concludes the paper and gives an outlook.

2 Related work

2.1 Runtime prediction modelling

Many works such as [14, 20, 24, 26, 31, 32] have been proposed recently to observe, analyze and predict the runtime performance of large-scale data processing platforms. To predict the runtime of MapReduce jobs, Starfish [14] introduced a self-tuning framework on top of Hadoop that applies an analytical approach to observe and analyze runtime metrics of jobs by running them on a fraction of the data and to optimize system performance by tuning its configuration options. PREDIcT proposed in [20] 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. Its main idea is to predict the number of iterations and build the runtime model of each from sample runs. A bounds-based performance model is presented in [31] to predict the execution time of MapReduce jobs running on heterogeneous clusters. In [26], the authors introduced a simulation-driven model to predict the execution time of Spark jobs by simulating their execution on a fraction of the data and collecting their execution metrics like memory consumption, I/O costs and runtime. Another approach is presented in [16] which models the memory behavior of Spark jobs based on a collection of experiments. Runtime prediction models were presented and a task co-location strategy proposed to improve system throughput. Ernest [24] is a large-scale performance prediction framework that presents a general runtime model for Spark jobs. For each Spark job, Ernest runs it with various configurations on a fraction of the data to build its model. Doppio [32] proposed a runtime prediction model for Spark jobs by studying the I/O impact on in-memory cluster computing frameworks and identified I/O overhead as a dominant bottleneck in such frameworks. Beside these studies, many others focused on constructing runtime models based on statistics [7, 17, 21].

2.2 Reuse and materialization

Reusing partial results has been extensively studied for data warehouses and relational databases, such as [8, 12, 13]. Early works focusing on reusing materialized views (or derived relations) are [15, 29] as well as [4] or [23] investigating the view-matching problem.

In [19] the Hawc architecture is introduced that extends the logical optimizer of an SQL system and considers the query history in order to decide which intermediate result may be worth materializing to speed up further executions – even if this would create a more expensive plan which, however, is executed only once.

For Hadoop MapReduce the MRShare framework [18] merges a batch of jobs into a new batch of jobs so that groups of jobs can share scans over input files and the Map output. Other projects such as ReStore [9], PigReuse [5], or [25] are similar to MRShare in the sense that they all merge a batch of scripts into a single plan or share the intermediate results after a map phase. In PigReuse, the optimization goal is to minimize the number of operators and the number of generated MapReduce jobs - but they do not analyze the total cost of the generated plans.

For Spark several additional frameworks were created to support data analysts with their tasks. KeystoneML [22] is able to identify expensive operations in machine learning pipelines on Big Data platforms like Apache Spark. They employ a cost model using cluster costs (such as network bandwidth, CPU speed, etc.) and operator costs to estimate total execution costs. From this physical operators for a logical plan are chosen and materialization points are determined. RDDShare [6] is also based on Spark and simply identifies common operators in a batch of Spark programs and merges them into a single program.

The presented performance prediction models focused on the impact of data size, platform configuration settings and allocated resources like memory consumption, I/O overhead and network bandwidth with the goal of choosing optimal cluster resource configuration and increasing the cluster throughput. However, none of these considered the application parameters which can affect the runtime performance significantly. Furthermore, we not only use our prediction model to manage cluster resources, but also as input for a cost based decision model to decide which intermediate results should be materialized and to be reused, rather than only merging a batch of programs into a single job.

3 Background

A Spark job consists of one or more stages [26], each consisting of multiple homogeneous tasks that run in parallel and process various RDD partitions of the same data source. The first stage reads data blocks from HDFS and loads them into memory as RDD partitions. Then for each RDD partition, it launches a task to process it. The level of stage execution parallelism is determined by the number of cores allocated for running the Spark job:

$$\#AllocatedCores = \#Executors \times \#CoresPerExecutor \tag{1}$$

The file block size is set using the `dfs.block.size` option when storing the file in HDFS. The number of executors and the number of cores per executor are set, when submitting the Spark job, using the `num-executors` and the `executor-cores` options respectively. Theoretically, the total execution time of a stage is as follows:

$$Runtime(Stage_i) = Runtime(TaskOfStage_i) \times \left\lceil \frac{\#TasksPerStage}{\#AllocatedCores} \right\rceil \tag{2}$$

Practically, however, the program driver makes an unbalanced distribution of tasks among cores:

$$Runtime(job) = Startup + \sum_{i=1}^{\#Stages} Runtime(Stage_i) + Cleanup \tag{3}$$

4 Gray-box modeling methodology

To develop a comprehensive runtime model for Spark jobs we first study the relationship between its application parameters and the RDD sizes (white-box modeling). We then develop the runtime model of each task individually regarding its input RDD partition size and its allocated resources (black-box modeling). Figure 1 shows a general overview on the gray-box modeling concept.

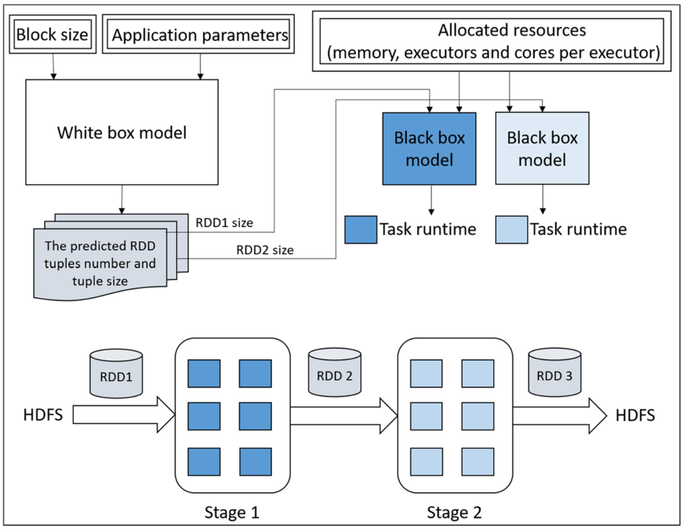


Fig. 1 Overview of gray-box modeling

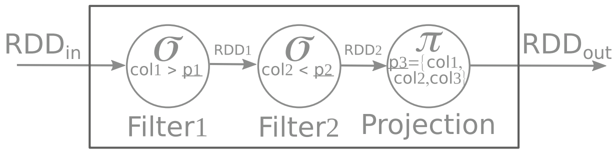


Fig. 2 Example of Apache Spark stage

4.1 White-box model

A Spark job consists of multiple stages running sequentially. Each stage takes an RDD as input (RDDin) and produces another as output (RDDout). The RDDout of a stage will be the RDDin to the one immediately after it. Knowledge of the size of RDDin and the selectivity of all operators that affect the RDD size is required to predict the size of RDDout in terms of the number of tuples and the tuple size.

Figure 2 shows an example of a stage containing three operators: two filters and one projection. $p1$, $p2$, and $p3$ are application parameters that 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 can be predicted by estimating the output RDD size of each operator in turn (RDD1 -> RDD2 -> RDDout).

$$\#RDDPartitionTuples = \frac{\#RDDTuples}{\#TasksPerStage} \tag{4}$$

$$\#RDDPartitionTupleSize = \#RDDTupleSize \tag{5}$$

Application parameters can be categorized by their influence on the RDD sizes and runtime:

Condition parameters Tuning these parameters affects the selectivity of filter operators and the cardinality of joins. In the example shown in Fig. 2:

$$\#RDD1Tuples = \#RDDinTuples \times Filter1Selectivity(p1) \tag{6}$$

In many cases, the selectivity of some operators is indirectly influenced by other parameters:

$$\#RDD2Tuples = \#RDD1Tuples \times Filter2Selectivity(p1, p2) \tag{7}$$

Field selection parameters Tuning these parameters affects RDD tuple sizes. Examples are map and projection (in Spark SQL). Predicting tuple size is important in estimating the required memory size, shuffling runtime, tuple compression and other performance metrics.

Workflow control parameters These parameters, like selecting processing algorithm or the number of iterations in a PageRank application, affect the RDD lineage and therefore the total job runtime.

4.2 Black-box model

The main allocated resources considered during the black-box modeling, in addition to the size of the input RDD partition, are the allocated memory which affects the rate of data spill to disk and the number of allocated executor cores that run concurrently and compete for shared executor resources like disk, memory and network. To build the runtime model of each task, we run the job several times with varying memory allocation, cores per executor and RDD partition sizes. We then collect runtime metrics of tasks using SparkListener [1] and analyze the metrics using regression analysis methods.

A noteworthy phenomenon is that the runtime of tasks may vary significantly on multiple runs on the same resources and with identical input RDD partitions. That is due to presence of straggler tasks [28] and it affects the prediction model. Analyzing the causes of straggler tasks and their impact is out of the scope of this work. However, to deal with this uncertainty, we increase the data size to increase the number of tasks in each job and take their average runtimes.

5 Building the gray-box model

This sections presents how to construct a gray-box model in detail. It then demonstrates the model building with real-world applications: WordCount, PageRank and K-Means.

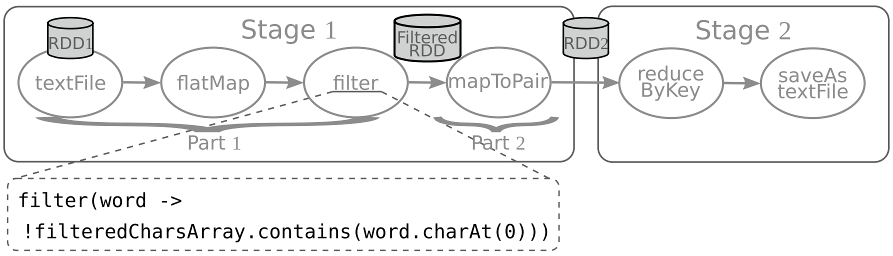
5.1 Building the white-box model

We discuss general rules for constructing the white-box model. The following are three ways to build the model depending on our knowledge of the application:

Knowledge of operators This is the approach to building the model when we have an understanding of the application, knowledge of how the operators work and collection of the data properties. Therefore, for any given application parameters and their values, such prior knowledge allows for building two models with which to predict two cardinalities: one in terms of the number of tuples (i.e., ratio of the number of tuples in output RDD partition to the number of tuples of input RDD partition) and the other in terms of the average tuple size (i.e., ratio of the average size of tuples in output RDD partition to the average size of tuples in input RDD partition). The output of these two models are used for the black-box modelling.

Observing history of runs The second approach is to observe previous application runs with different application parameters and use it to train the two models for predicting the two cardinalities mentioned in the first approach. This approach is useful when we do not know the characteristics of the data or the workings of the operators. It differs from the previous one in building the two models: while the models in the first approach are built from our knowledge of the operators, this approach builds the models from the history. From the history runs, we obtain numerous values of all application parameters and the corresponding cardinalities in terms of the number of tuples and the average tuple size. We start by computing the *standard correlation coefficient* between all application parameters and each of the two cardinalities of each operator individually. Then we select the application parameters as features of the corresponding model based on their high correlation with a cardinality, which is the label. That way, we avoid incorporating features that do not have impact on the label into the model. Next, we develop the two linear regression models for the two cardinalities. The first model takes the (high-correlation) application parameters as features and the cardinality of the number of tuples as labels, while the second model takes the (high-correlation) application parameters as features and the cardinality of average size of tuples as labels. For any given values of application parameters, we use these two models to make predictions of the RDD sizes in terms of the number of tuples and the average tuple size.

Training The third approach is to generate a training data and use it to train the two models discussed previously. This approach is useful when we neither have a history of runs nor know the characteristics of the data or the workings of the operators. We run the application multiple times while tuning the values of application parameters and observe the RDD sizes and study the correlation between them, similar to observing history of runs.



```
filter(word ->
 !filteredCharsArray.contains(word.charAt(0)))
```

Fig. 3 A WordCount lineage with an application purpose filter

5.2 Building the black-box model

We run the job multiple times, varying RDD partition sizes and allocated resources. Then we build the runtime model based on the observed runtime metrics. The difference in run configurations is based on:

- RDD partition size: The simplest way to modify it is to change the HDFS block size by creating multiple versions of the input file with various block sizes.
- Cores allocated per executor: It is set using the `-executor-cores` Spark configuration option.
- Memory allocated per core: It is set using the `-executor-memory` Spark configuration option.

5.3 Use cases

We demonstrate how to apply our approach on three real-world applications and discuss the influence of their application parameters on their runtime.

5.3.1 WordCount

Our first use case is WordCount. Assume that users are allowed to select a group of letters with which to run WordCount jobs via a web portal. The result is the number of words in the source file that do not begin with any of the letters selected. The only application parameter is the list of letters. One filter operator is injected in the first stage's lineage of operators to filter out words that start with the selected letters. Figure 3 depicts the lineage of operators in the WordCount application with their RDDs.

To build the white-box model, RDD sizes are predicted based on the selected letters. The cardinality models of all operators in the lineage are static and can be observed during running sample tests except the cardinality model of the filter, which is dynamic and influenced by the selected letters. The white-box model accurately predicts the number of FilteredRDD (cf. Fig. 3) tuples and their average size for all possible 2^{26} combinations of selected letters. 2^{26} is the *powerset* of the set of all English letters and comprises all possible lists of letters (i.e., application parameters). The relative frequencies of the first letters of words in the English language are known in advance and used to predict the number of tuples in FilteredRDD. Statistics of length of words based on their first letter (obtainable from a 16MB MySQL English dictionary database [3]) are used to predict the average tuple size of FilteredRDD.

Let us assume that a user selects the list {a, f, h, n} as application para-meter. From the statistics, the relative frequencies of words that start with *a, f, h* and *n* are 11.68%, 4.03%, 4.20% and 2.28% respectively. Thus:

– Ratio of number of FilteredRDD Tuples:

$$1 - (11.68\% + 4.03\% + 4.20\% + 2.28\%) = 79.81\%$$

– Average size of FilteredRDD tuple:

$$\frac{\sum_{i='b'}^{NonFilteredLetters} (frequency(i) \times WordAverageSize(i))}{\sum_{i='b'}^{NonFilteredLetters} frequency(i)} = \frac{\sum_{i='b'}^{NonFilteredLetters} (frequency(i) \times WordAverageSize(i))}{79.81\%}$$

Where *frequency* is the relative frequency of words that start with the corresponding letter and *WordAverageSize* is the average length of words that start with the same letter. The cardinality model can be enhanced by including statistics of the most frequently occurring words and other useful distributions of English words. Although this increases the model accuracy, it also increases the modeling efforts and time. In this use case, the selectivity of each operator is as follows:

- flatMap: splits the sentence into a list of words. Assuming that the average number of words per line is known to be T, the selectivity of this operator is T x 100%.
- filter: the selectivity of this operator is dynamic.
- mapToPair: maps each word to a key-value pair. Its selectivity is 100%.
- reduceByKey: the resulting number of RDD tuples equals the number of unique words in the source file, which can be estimated even without knowing the number of tuples input to this operator.

After defining the cardinality model for each operator, we predict the initial RDD size. From the statistics [3], the average word length is 8.12. The average tuple size of the initial RDD is then the summation, in bytes, of the average word length (8.12) and newline delimiter (1). The number of its tuples equals the source file size divided by 9.12.

To build the black-box model, we run the WordCount job multiple times, varying RDD partition sizes and allocated resources. Each experimental dimension is varied to have at least three values like {32 MB, 64 MB and 128 MB block size}, {1, 2 and 4 cores per executor}, {1GB, 2GB and 4GB allocated memory}. To cover all possibilities, we run the job 27 times and the average runtime model of each task is extracted by analyzing the runtime logs provided by SparkListener [1].

Further model enhancements need to be performed to improve the model accuracy. For example, the runtime of Stage 1 is determined by sizes of RDD1 and FilteredRDD, while the runtime of Stage 2 is influenced by the size of RDD2. Also, the runtime of flatMap and filter operators are determined only by RDD1. Although we predict the size of FilteredRDD from the size of RDD1 based on the application parameters, the runtime of reduceByKey operator is influenced only by the size of FilteredRDD. By taking the whole of Stage 1 as a black-box, we would not be able to predict the size of FilteredRDD from that of RDD1. To overcome this, the first stage is divided into two parts, as shown in Fig. 3, and runtime models are developed for both parts separately by injecting a modeling purpose `mapPartitions` operator after the filter. The aim of this injected operator is to profile the task runtime by sending the current timestamp to a central storage unit prepared for modeling purposes [11]. The runtime of the filter is profiled and its runtime cost model is built. The total job runtime decreases when the user selects more letters, while the time required to perform filtering operation increases. In this example, it takes just 2 ms more to filter 25 letters (in 1GB HDFS block) than to filter only one.

5.3.2 PageRank

The second use case is PageRank. We illustrate the construction of its white-box model. We do not discuss its black-box model as it is similar to the black-box model construction described earlier in the WordCount application. Unlike the WordCount use case where the application parameter (list of selected letters) is a condition parameter, the application parameter in the PageRank use case is a workflow control parameter: the number of iterations. The PageRank application consists of only one job. The job is constituted of two static stages at the beginning (i.e., the number of stages is constant), one static stage at the end and dynamic stages in the middle (i.e., the number of stages varies). The number of dynamic stages is determined by the number of iterations: one iteration results in one dynamic stage, two iterations result in two dynamic stages and so on. However, varying the number of iterations does not change the RDD sizes or cardinalities. It only influences the number of the dynamic stages. The runtimes of the dynamic stages are equal because the operation is the same, with the same input data size and allocated resources.

5.3.3 K-Means

Our third use case is K-Means. For this clustering application, the number of clusters (i.e., K) is the application parameter. Similar to PageRank, the application parameter is a workflow control parameter. However, unlike in PageRank where the number of iterations influence the number of stages, in K-Means, it influences the number of jobs. There are eight static jobs at the beginning of the application, one static job at the end of the application, and dynamic jobs in the middle. Each of the dynamic jobs consists of two stages. Secondly, while an iteration in PageRank maps to a stage (x iterations result in x dynamic stages), it is not as simple and straightforward with

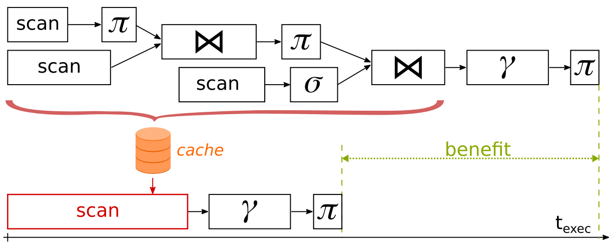


Fig. 4 Runtime difference for loading materialized result

respect to jobs in K-Means. Thirdly, the runtime of dynamic stages in PageRank is not influenced by the application parameter. But in K-Means, the application parameter influences the runtime of dynamic jobs linearly (in addition to its linear correlation with the number of iterations). This results due to the influence of K on the cardinality (and thus the output RDD) of one of the operators in the job. Ultimately, that influences the runtime of the part of the job following the output RDD of the operator.

6 Using the prediction model for cost-based recycling

While the runtime prediction is crucial for scheduling jobs and allocating enough (but not too many) resources, it can be applied in combination with a cost-based decision model to identify operators whose results should be materialized for sharing and reuse.

Figure 4 shows the DAG of a job where the width of a node's box represents its processing time. If, e.g., the result of the second join operator is materialized, subsequent executions of dataflow programs that also contain this part in their respective DAG will benefit by only having to load the already present result from disk.

In [11] we introduced the decision model together with its integration in out Piglet [10] engine. Piglet translates Pig Latin scripts into Spark programs and instrumented the generated code to gather profiling information. The profiling data was used as input for the decision model. With the prediction model of Sect. 5 we are able to apply the decision model, without having to gather profiling information first.

The goal of the decision model is to increase the *benefit* for Spark jobs. The benefit is the execution time a job saves when reusing a materialized result from a previous job, as depicted in Fig.4. Alternatively, one can also regard the benefit as the amount of money saved by needing to rent fewer machines or less resources in a public cloud. To calculate the benefit, the decision model is based on the costs of operators which are taken from the prediction model described in Sects. 4 and 5.

The decision model is built around *materialization points*:

Definition 6.1 A materialization point M is a logical marker in a DAG denoting a position for the decision model to write or load the materialized results and thus, refers to the output of an operator.

To decide the result of which operator should be materialized, we introduce the notion of the *benefit*:

Definition 6.2 The benefit of a materialization point M_i is the execution time saved when loading materialized data instead of recomputing it. It can be expressed as in Eq.(8).

$$t_{\mathsf{benefit}}(M_i) = t_{\mathsf{total}}(M_i) - t_{\mathsf{read}}(M_i) \tag{8}$$

$$t_{\mathsf{total}}(M_i) = \sum_{o \in \mathsf{prefix}(M_i)} t_{\mathsf{exec}}(o) \tag{9}$$

Here $t_{\mathsf{total}}(M_i)$ is the cumulative execution time of all operators in the prefix of o_i from the source operator to M_i, whereas $t_{\mathsf{read}}(M_i)$ denotes the time required to load the materialized data of M_i. We have to distinguish two cases: the prefix of M_i does not contain a join (or similar) operator, or it does. In the first case, $t_{\mathsf{total}}(M_i)$ can be calculated as in Eq.(9). Otherwise, if there is a join operator j, only the longest execution time of the input branches $k_1(j), \dots, k_n(j)$ of j is considered for $t_{\mathsf{total}}(M_i)$:

$$t_{\mathsf{total}}(M_i) = \max\{t_{\mathsf{total}}(k_1(j)), \dots, t_{\mathsf{total}}(k_n(j))\} + \sum_{\substack{o \in \mathsf{prefix}(M_i) \\ \wedge o \notin \mathsf{prefix}(j)}} t_{\mathsf{exec}}(o) \tag{10}$$

The time required to load existing materialized data for M_i from disk depends on the cardinality of M_i as well as the hardware-dependent factor *bps*, denoting the number of bytes that can be read per second. The time to read the materialized result of M_i is calculated as:

$$t_{\mathsf{read}}(M_i) = \frac{\mathsf{card}(o_i) \cdot \mathsf{width}(o_i)}{bps} \tag{11}$$

6.1 Decision model

The decision model has three dimensions to consider when choosing a materialization point to actually write to persistent storage.

(1) Which materialization points should be selected for further investigation?
(2) From the list of materialization points resulting from (1), which of those should be materialized?
(3) If the persistent storage is limited in space, decide which existing materialized result has to be deleted.

Please note that due to space limitations, we only briefly describe one strategy for (2) and refer the reader for more details to [11]. Furthermore, we consider the dimension of cache eviction as out-of-scope of this paper and trivially assume an infinite cache.

(1) *Selection* Obviously, the materialization points belonging to sources and sinks do not need to be considered further as they just materialize the original data or the data that is stored anyway. Therefore, subsequent operations only need to consider materialization points that do not belong to a source or sink operator.

(2) *Ranking materialization points* A ranking strategy is needed to select one or more materialization points from the list of candidates. An obvious strategy is to always materialize the materialization point that yields the highest benefit to the program.

However, using only the benefit might not be enough as it does not consider if that materialization point will be ever reused. Thus, one could also consider the probability for reuse in combination with the achievable benefit, using a e.g., Skyline approach. The probability of one operator following another one can then be modeled using a Markov chain.

(3) *Cache eviction* In practice, the space occupied by materialized results needs to be limited. Thus, when selecting a new M_i^j for a job j, one might have to evict an older M_o^k. To decide which to evict one can employ algorithms like LRU, LFU, etc. well-known from traditional database systems. As these do not account for the benefit these Ms bring, a modified ARC or KNAPSACK may help. However, the cache eviction strategies are not in the scope of this paper and will be investigates in future work.

7 Evaluation

We performed experiments on our Spark cluster to evaluate the proposed modeling methodology. The 16-node cluster has the following specifications per node: Intel Core i5 2.90 GHz, 16 GB DDR3 RAM, 1 TB disk, 1 GBit/s LAN. The cluster runs Hadoop 2.7, Spark 2.0.1, Java 8u102, and Apache YARN on top of a Hadoop Distributed File System (HDFS).

In this evaluation, we first demonstrate the accuracy of the prediction model and then show that the materialization decision significantly improves the overall execution time.

7.1 Prediction model

7.1.1 WordCount

To evaluate the white-box model, we ran the job with various application parameter values 26 times. An empty list was first passed as a parameter to J1. Each of the subsequent jobs added one additional letter to the list in turn, e.g., J2 {'a'}, J3 {'ab'}, J4 {'abc'} and so on.

We built the white-box model to predict the RDD sizes across the operator lineage. Figure 5 shows the experimental results for the white box model.

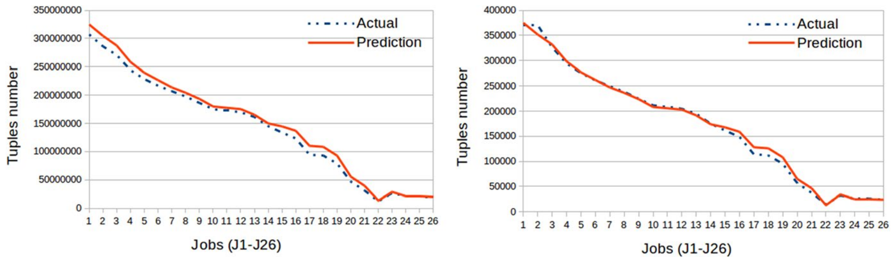


Fig. 5 (Left) Number of RDD tuples resulting from filter and mapToPair operators. (Right) Number of RDD tuples resulting from reduceByKey operator

To evaluate the black-box model, the job was executed 20 times with randomly selected values of data sizes {1–2 GB}, allocated memory {1, 2, 4 GB} and allocated cores {1, 2, 4} per executors. The results are shown in Fig. 6. Finally, we executed 50 WordCount jobs with arbitrary values of application parameters, allocated memory {1, 2, 4 GB}, allocated cores {1, 2, 4} per executor on various source files (3-6 GB) stored in the HDFS with varying block sizes (64 and 128 MB). Figure 7 shows the comparison between the observed actual runtimes and the predicted runtimes (which are calculated using our prediction model) of each job. The average prediction accuracy of the runtime model for this use case was 89.84%.

7.1.2 PageRank

Our approach to constructing the white-box model in PageRank is based on history. We ran the PageRank application on a data set of 5 million pages with three iterations, which resulted in three iterative stages, in addition to the three static stages

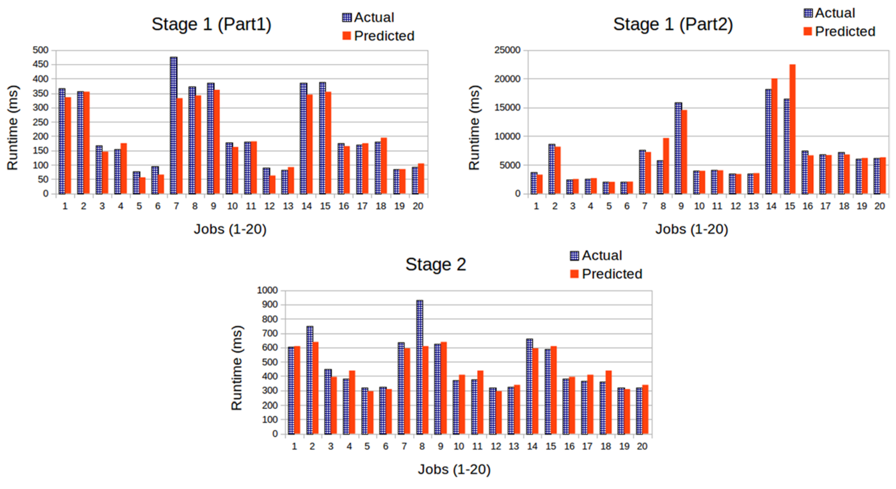


Fig. 6 Black-box model evaluation results

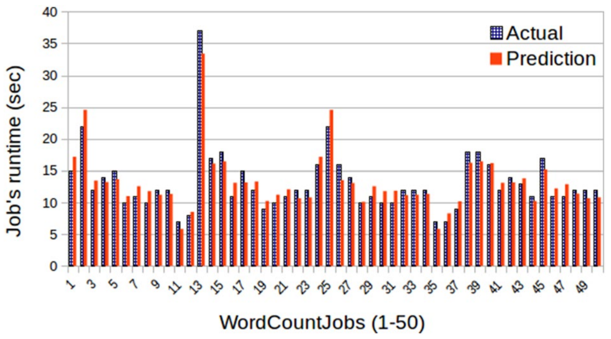


Fig. 7 Gray-box model evaluation results

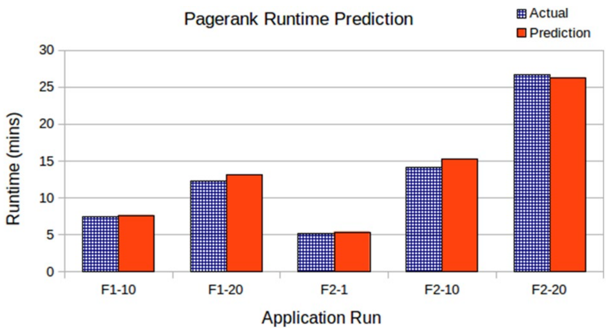


Fig. 8 Application runtime prediction for PageRank

(ref. 5.3.2). The runtime metrics for this job served as our history, from which we predicted the application runtime for different number of iterations. Then we doubled the input data size and made predictions of the application runtime for different number of iterations on the same allocated resources. Figure 8 depicts our prediction accuracy after comparing the actual runtimes with the predicted runtimes using our prediction model. F1 and F2 represent the initial and the doubled data sizes respectively. The figure shows predictions for the initial data size with 10 and 20 iterations, and for the doubled data size with 1, 10 and 20 iterations. Our average prediction accuracy for this use case was 95.72%. The historical run took around 4 min, which is 5.90% of the total time taken by just these few actual runs (67.7 minutes).

7.1.3 K-Means

We carried out evaluation for K-Means using a dataset containing 2 million points with 20 dimensions. We followed the training approach for this use case to construct the white-box model. The number of iterations is influenced by the number of clusters K. Thus, we ran the experiments while varying the values of K and allowed the application to run for as many number of iterations as K results into. As mentioned earlier, the correlation between K and the number of iterations, as well as the correlation between K and the runtime of the dynamic job (i.e., iteration) are both linear. However, for some instances of K, different runs of the application with the same value of K results in slightly different number of iterations. We made runs with K=10 and K=50 in the training phase. Figure 9 shows the accuracy of our prediction of both the number of iterations and the runtime of the iterative job from K after compering the actual number of iterations and runtimes with the predicted number of iterations and runtimes using our prediction model. With regards to runtime of the iterative job, we realized that the operator whose cardinality is influenced by K in the iterative job is the mapPartition operator. The mapPartition is the first operator in the job. That way, K influences the output RDD of the operator, and ultimately, the job runtime. In addition to the iterative job(s), there are eight static jobs at the beginning of the application and one static job at the end of the application. Based on that, we made predictions of the application runtime on the same allocated resources for different instances of K as shown in Fig. 10. We achieved an average prediction accuracy of 85.62%. The training took around 6.5 min, which is 13.97% of the total time taken by just these few actual runs (46.50 min).

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

- 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.
- Although, theoretically, uneven distribution of tasks is expected to decrease the model accuracy, it also reduces the impact of variance between tasks runtime which eliminates the impact of stragglers. We analyzed the affect of this in the presented applications by comparing the actual runtime of stages with the predicted runtime of them (by applying Equation 3) and realized that the accuracy of our prediction model was reduced in the worst case by 7.23%.

7.2 Materialization

As we already demonstrated the quality and applicability of the cost-based decision model for materialization in [11], we want to show here the results based on the prediction model. To do so, we took two exemplary Spark programs, depicted in Fig.11. Program *GDELT* loads a 45GB CSV file *G* from the GDELT[1] project and performs basic preprocessing (projection, filtering), computes average tone of news published per website, and returns only those with an average tone greater than a threshold provided as input parameter. The indicated materialization points are the ones used in Fig. 13. The *Weather* program loads weather observations *O* and sensor

[1] https://www.gdeltproject.org/.

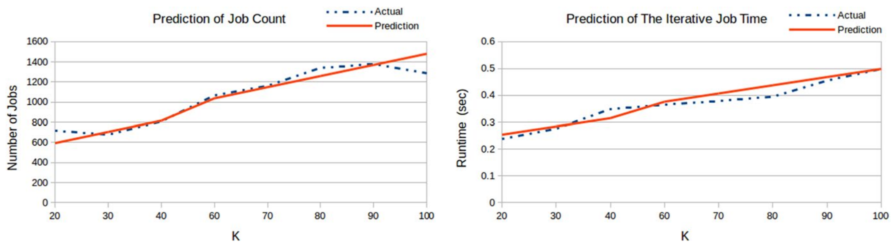


Fig. 9 (Left) Prediction of Number of Job Iterations from Number of Clusters. (Right) Prediction of Output Data Size from Number of Clusters

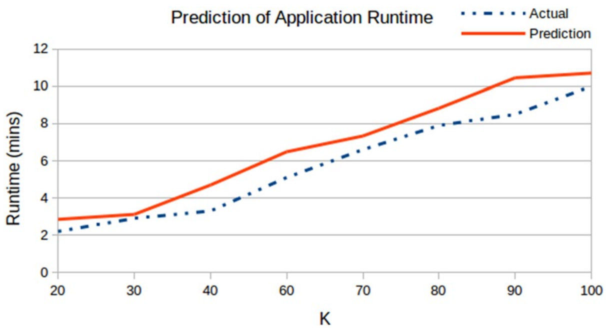


Fig. 10 Application runtime prediction for K-Means

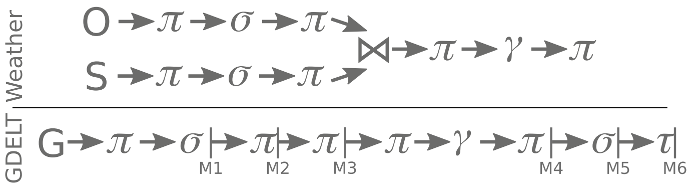


Fig. 11 Depiction of the two Spark jobs used to evaluate the materialization decision

information *S* from the SRBench [30] project. The two input files are filtered according to provided input parameters and joined by sensor. The aggregation counts the number of observations per sensor.

We used the prediction model to obtain operator costs (runtimes) as well as their expected result sizes. Based on these values, we used the *MaxBenefit* strategy to find which operators should be materialized. Figure 12 compares the execution times of the two scripts without any materialization (*plain*), when writing the materialized result to disk (*materialize*), and when reusing the materialized data in the same program (*reuse*). As one can see, the execution time is drastically reduced when we are able to reuse previously computed and materialized intermediate results. Writing the materialized data has a small impact on the execution time, which however is amortized by the time saved when reusing the the materialized data. Figure13 exemplary shows for the GDELT program that

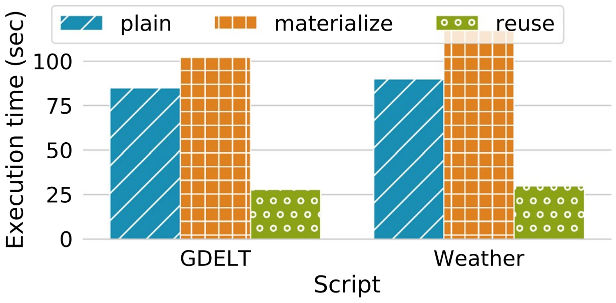


Fig. 12 Impact of materializing (middle) and reusing (right) materialized results compared to normal execution (left)

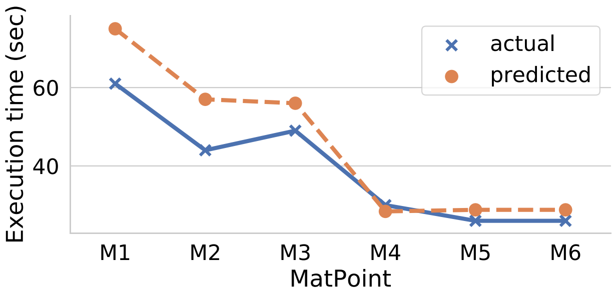


Fig. 13 Predicted versus actual execution time of materialization points of GDELT program

the runtime prediction for the materialization points from Fig. 11 in combination with the cost model does not produce wrong decisions. We rather predict slightly longer execution times for materialization points highlighted in Fig. 4 and therefore a smaller (or no) benefit, that are, however, not too far off. The reason for the higher prediction has several reasons. One is the calibration of the read and write performance from and to HDFS, and of course the measurements of the operator costs, which are pessimistic and orientated on the slowest task. This avoids wrong decisions were the model decides to materialize a result that would actually produce longer execution times when reusing that data.

8 Conclusion and Outlook

In this paper, we presented a gray box modeling methodology for runtime prediction of Apache Spark jobs and its application for reusing intermediate results of such jobs, for which a cost-based decision model is used. The prediction model consists of a white-box model and a black-box model. The white-box model uses prior knowledge to predict the sizes of (intermediate) RDDs in each stage under varying input parameter values, whereas the black-box model is constructed for each task by observing runtime metrics with varying allocated resources and input partition sizes. The decision model is based on the cost information taken from the gray-box model and identifies operators in a job whose result should be materialized in order to share it with other jobs or future executions of the same job.

In the experimental evaluation of the prediction methodology, we showed that the model achieves a high prediction accuracy of up to 83–94%. Furthermore, we showed that the information is very useful for the materialization and a drastic runtime improvement could be achieved when reusing previously computed intermediate results.

In future, we plan to develop a scheduling methodology based on the developed runtime models in this study to improve resource utilization and increase the overall system throughput. Additionally, the developed runtime models in this study will be used in a simulation-based prediction approach to analyze system performance (e.g., throughput, latency average and the utilization of cluster resources) and to answer what-if questions about system configuration. Currently, the materialization decision model can only reuse results that were produced by the exact same lineage of operators with the same parameter values. This will be extended using a query containment approach to be able to decide that a materialized result can be reused by another job if that new job, e.g., was started using different parameter values.

Acknowledgements Open Access funding provided by Projekt DEAL. Funding for Hani Al-Sayeh was provided by Deutsches Zentrum för Luft-und Raumfahrt (DLR). Funding for Stefan Hagedorn Partially funded by DLR.

References

1. Apache spark: Monitoring and instrumentation. https://spark.apache.org/docs/latest/monitoring.html (2019). Accessed 22 Feb 2019
2. Apache spark official website. https://spark.apache.org/docs/latest/configuration.html (2019). Accessed 22 Feb 2019
3. Abiteboul, S., Duschka, O.M.: Complexity of answering queries using materialized views. In: Proceedings of the PODS, pp. 254–263, (1998)
4. Camacho-Rodríguez, J. et al.: PigReuse: a reuse-based optimizer for Pig Latin. Technical Report, Inria Saclay (2016)
5. Chao-Qiang, H. et al.: RDDShare: reusing results of spark RDD. In: Proceedings of the DSC, pp. 370–375, (2016)
6. Chaudhuri, S., Narasayya, V., Ramamurthy, R.: Estimating progress of execution for sql queries. In: Proceedings of the SIGMOD, pp. 803–814, (2004)
7. Chirkova, R., Halevy, A.Y., Suciu, D.: A formal perspective on the view selection problem. In: Proceedings of the VLDB, pp. 59–68, (2001)
8. Elghandour, I., Aboulnaga, A.: Restore: reusing results of mapreduce jobs. VLDB **5**, 586–597 (2012)

9. Hagedorn, S., Sattler, K.: Piglet: interactive and platform transparent analytics for rdf & dynamic data. In: Proceedings of the 25th international conference companion on world wide web, WWW 2016 Companion, pp. 187–190, (2016)
10. Hagedorn, S., Sattler, K.U.: Cost-based sharing and recycling of (intermediate) results in dataflow programs. In: Proceedings of the ADBIS, pp. 185–199. Springer, (2018)
11. Halevy, A.Y.: Answering queries using views: a survey. VLDB J. **10**(4), 270–294 (2001)
12. Harinarayan, V., Rajaraman, A., Ullman, J.D.: Implementing data cubes efficiently. SIGMOD Rec. **25**(2), 205–216 (1996)
13. Herodotou, H., Lim, H., et al.: Starfish: a self-tuning system for big data analytics. Cidr **11**, 261–272 (2011)
14. Larson, P.A., Yang, H.Z.: Computing Queries from Derived Relations: Theoretical Foundation. Department of Computer Science, University of Waterloo, Waterloo (1987)
15. Marco, V.S., Taylor, B. et al.: Improving spark application throughput via memory aware task co-location: a mixture of experts approach. In: Proceedings of the Middleware, pp. 95–108. ACM, (2017)
16. Morton, K., Balazinska, M., Grossman, D.: Paratimer: a progress indicator for mapreduce dags. In: Proceedings of the SIGMOD, pp. 507–518. ACM, (2010)
17. Mysql english dictionary. https://sourceforge.net/projects/mysqlenglishdictionary/ (2019). Accessed 22 Feb 2019
18. Nykiel, T., et al.: MRShare: sharing across multiple queries in MapReduce. PVLDB **3**(1–2), 494–505 (2010)
19. Perez, L.L., Jermaine, C.M.: History-aware query optimization with materialized intermediate views. In: Proceedings of the ICDE, pp. 520–531. IEEE, (2014)
20. Popescu, A.D., Balmin, A., et al.: Predict: towards predicting the runtime of large scale iterative analytics. PVLDB **6**(14), 1678–1689 (2013)
21. Selinger, P., Astrahan, M.M. et al. Access path selection in a relational database management system. In: Proceedings of the SIGMOD, pp. 23–34. ACM, (1979)
22. Sparks, E.R. et al. KeystoneML: optimizing pipelines for large-scale advanced analytics. In: Proceedings of the ICDE, pp. 535–546, (2017)
23. Srivastava, D., Dar, S., Jagadish, H.V., Levy, A.Y.: Answering queries with aggregation using views. VLDB **96**, 318–329 (1996)
24. Venkataraman, S., Yang, Z. et al. Ernest: efficient performance prediction for large-scale advanced analytics. In: Proceedings of the NDIS, pp. 363–378, (2016)
25. Wang, G., Chan, C.Y.: Multi-query optimization in MapReduce framework. In: Proceedings of the PVLDB, pp. 145–156, (2013)
26. Wang, K., Khan, M.M.H.: Performance prediction for apache spark platform. In: Proceedings of the HPCC, pp. 166–173, (2015)
27. Wang, K., Khan, M.M.H., Nguyen, N., Gokhale, S.: Modeling interference for apache spark jobs. In: Proceedings of the CLOUD, pp. 423–431. IEEE, (2016)
28. Xin, R., Deyhim, P., Ghodsi, A., Meng, X., Zaharia, M.: Graysort on apache spark by databricks. In: Proceedings of the GraySort Competition, (2014)
29. Yang, H.Z., Larson, P.A.: Query transformation for PSJ-queries. PVLDB **87**, 245–254 (1987)
30. Zhang, Y. et al. SRBench: a streaming RDF / SPARQL Benchmark. In: Proceedings of the ISWC, pp. 641–657, (2012)
31. Zhang, Z., Cherkasova, L., Loo, B.T.: Performance modeling of MapReduce jobs in heterogeneous cloud environments. In: Proceedings of the CLOUD, pp. 839–846, (2013)
32. Zhou, P., Ruan, Z. et al.: Doppio: I/o-aware performance analysis, modeling and optimization for in-memory computing framework. In: Proceedings of the ISPASS, pp. 22–32. IEEE, (2018)