Performance Assurance Model for Applications on SPARK Platform

Rekha Singhal(✉) and Praveen Singh

Tata Consultancy Services Research, Mumbai, India
rekha.singhal@tcs.com

Abstract. The wide availability of open source big data processing frameworks, such as Spark, has increased migration of existing applications and deployment of new applications to these cost-effective platforms. One of the challenges is assuring performance of an application with increase in data size in production system. We have addressed this problem in our work for Spark platform using a performance prediction model in development environment. We have proposed a grey box approach to estimate an application execution time on Spark cluster for higher data size using measurements on low volume data in a small size cluster. The proposed model may also be used iteratively to estimate the competent cluster size for desired application performance in production environment. We have discussed both machine learning and analytic based techniques to build the model. The model is also flexible to different configurations of Spark cluster. This flexibility enables the use of the prediction model with optimization techniques to get tuned value of Spark parameters for optimal performance of deployed application on Spark cluster. Our key innovations in building Spark performance prediction model are support for different configurations of Spark platform, and simulator to estimate Spark stage execution time which includes task execution variability due to HDFS, data skew and cluster nodes heterogeneity. We have shown that our proposed approaches are able to predict within 20% error bound for Wordcount, Terasort, K-means and few TPC-H SQL workloads.

1 Introduction

The digitization wave has led to challenge of processing high volume and high velocity data in real time. Apache Spark is one of the commodity cluster platforms available in open source to address this need due to its in-memory processing capability. Application deployment on commodity cluster system has challenge of assuring its performance over time with increase in data size. Conversely, appropriate capacity sizing of production Spark cluster is needed for desired performance irrespective of increase in data size. This raises the need for a performance assurance model, which can estimate an application performance for larger data sizes and variable cluster sizes before deployment. Here, by performance we mean application *execution time.*

One of the popular black box approaches is to use machine learning techniques to build performance prediction model. This requires identification of performance sensitive parameters (or relevant features) and collecting their values for multiple executions of application which may delay deployment. We have discussed this in detail in Sect. 3. An analytic or mathematical model based on few measurements is desirable to reduce cost and time to deploy.

An application deployed on Spark platform is executed as a sequence of Spark jobs. Each Spark job is executed as a directed acyclic graph (DAG) consisting of stages. Each stage has multiple executors running in parallel and each executor has set of concurrent tasks. This complexity cannot be handled by simple mathematics alone. We have proposed a hierarchical model for estimating Spark application execution time. Further, data skew and task execution variability have been handled by building a simulator for Spark jobs. Literature also has similar simulator but for Hadoop MR jobs [12]. We have focused on Spark parameters which can be changed during application execution and hence the proposed performance prediction model may be used with optimization techniques to get tuned value of Spark parameters for auto tuning. This paper has following contributions.

- Analysis of Spark's configurable parameters' sensitivity to application execution time with respect to increase in data size. Use of this analysis to define features to be used by machine learning algorithms for predicting application execution time on larger data sizes. We have compared accuracy of prediction models based on various ML techniques such as Multi Linear Regression (MLR), MLR-Quadratic and Support Vector Machine (SVM).
- Analytic based approach to predict an application execution time, on Spark platform, for larger data and cluster sizes using limited measurements in small size development environment. This has led to innovation in building simulator for estimating Spark job's stages' execution time. We have also built models for estimating task's JVM time, task's scheduler delay and task's shuffle time as function of input data size to support different configurations of Spark cluster. This capability of the model may also be used to build auto tuner.

The paper is organized as follows. Section 2 discusses the related work. The Spark platform performance sensitive parameters analysis and machine learning approach for building performance prediction model are discussed in Sect. 3. The analytic based performance prediction model is presented in Sect. 4. The experimental results for validation of the model are presented in Sect. 5. The extension of the performance prediction models to build auto tuner is formalized in Sect. 6. Finally, the paper is concluded in Sect. 7.

2 Related Work

Lot of work has been done in the area of performance prediction [5,7,14] and auto tuning of applications [4,6,8,9] on big data platforms. Majority of this work addresses Hadoop technology. [10,11] have concentrated on building performance prediction models, using limited measurements in small size development environment, for relational databases and Hive+Hadoop platforms respectively. However, [3,8,13] discuss performance analysis and tuning of Spark cluster. Ahsan [3] has shown that application performance degrades on large data size primarily due to JVM GC overheads. We have also built task's JVM prediction model as function of data size for estimating application execution time on larger data size, as discussed in Sect. 4.3. Machine learning techniques have also been used by big data community to build performance models [7,14]. We categorize the related work into two parts- Machine learning (ML) based approach and Cost based analytic approach.

2.1 ML Based Approach

Machine learning (ML) based approach is a black box method, which has been explored by big data community primarily to model performance of complex big data system. ML models are simple to build and are based on measurements collected during execution of actual workload on actual system. Kay et al. [7] has proposed generic ML approach with design of experiments and feature selection for analytic workload on big data platforms. [14] talks about tuning and performance prediction of Hadoop jobs using machine learning approach. They have focused on four performance sensitive parameters of Hadoop platform along with data size to build model. They have compared the accuracy of models built using different algorithms such as MLR, MLR-quadratic, SVM etc. We have customized this approach for Spark platform as discussed in Sect. 3.

2.2 Cost Based Approach

Cost based approach employs white box technique, which builds model based on deeper understanding of a system. However, it uses finite resources to build model unlike ML approach. Starfish [5] conducts instrumentation of Hadoop to collect performance measurements and build performance model to estimate a job execution time as function of various Hadoop platform parameters and data sizes. This method makes it adaptable for auto tuning by optimizing the model for different parameter settings. We have used similar methodology for Spark platform, but without instrumentation, by including Spark platform performance sensitive parameters as inputs to the prediction model as discussed in Sect. 4.

Panagiotis [8] proposes to tune a large number of Spark parameters using trial and error rule base created with few measurements. Their focus has been more on serialization and memory related parameters, however, we are interested in parallelism and memory related parameters in this paper. Shi et al. [9] has proposed Produce-Transfer-Consume (PTC) approach to model Hadoop job execution cost and used this to get optimal setting for Hadoop platforms. They have identified only few key parameters which are used to tune Hadoop system for a given job. We have also chosen only few performance sensitive parameters to build performance prediction model for Spark applications. The proposed model is formulated along the steps involved in an application execution on Spark platform. However, we do not perform white box instrumentation rather conduct our own experiments to collect the desired performance data for each such step. Wang et al. [13] has proposed an analytic based model for predicting Spark job performance and is closely related to our work discussed in Sect. 4. However, their model is restricted to same values for Spark parameters both in the sample and actual execution of an application. We could overcome this limitation by including sub models for estimating task JVM time, Shuffle time and Scheduler delay time. The heterogeneity at data, HDFS and hardware level for task execution has been handled by a simulator for estimating Spark stage execution time unlike the mathematical approach proposed in [13].

3 Machine Learning Based Model

Building a machine learning based model requires correct identification of features and choice of right machine learning algorithm. Spark platform has more than 100 parameters to configure [8]. The first challenge is to identify right set of parameters which impact an application execution time for varying data and cluster sizes and this set constitutes our feature set. We targeted only those parameters which could be changed during an application execution.

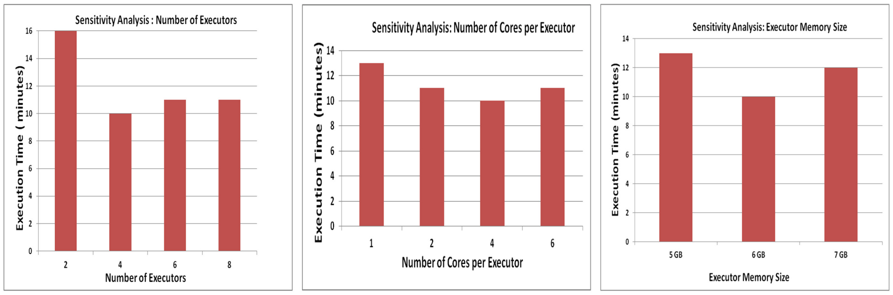

Fig. 1. Performance sensitivity analysis on 20 GB data size and 2 node cluster

We have conducted performance sensitivity analysis for number of parameters which are potential candidates for feature selection. The most sensitive parameters identified are the ones, whose changes led to variations in performance of the application. Our observed feature set in Spark 2.0 constitutes number of executors, number of cores per executor, executor memory size (this controls both shuffle memory and JVM heap size) and data size as shown in Fig. 1.

3.1 Experimental Set Up and Results

Our experimental setup consists of 5 nodes, each of Intel(R) Xeon(R) CPU X5365 @ 3.00 GHz, 8 cores and 16 GB RAM. The platform stack consists of Yarn, Apache Spark 2.01 and HDFS 2.6. We have one master and maximum four slaves in these experiments. We have formulated set of experiments based on the hardware constraints of the system. For example, the product of 'number of cores per executor' and 'number of executors' can vary from 1 to maximum cores in the cluster. The experimental configurations to collect training data is given in Table 1. We have built and tested ML prediction models for three types of workloads – Wordcount, Terasort and K-means [1] for data sizes varying from 5 GB to 15 GB. A linux bash script executes each of the application for all combinations of the parameters settings given in Table 1. Few of the combinations are invalid due to resource mismatch and are skipped. In total we could collect around 400 data points, as training set, for each application to build ML model.

Table 1. Experimental setup configuration for machine learning model

Configuration parameter	Minimum value	Maximum value
Number of executors (–num-executor)	2	10
Number of cores per executor (–executor-cores)	1	8
Executor memory (–executor-memory)	1	12
Data size	1 GB	15 GB

Table 2. Accuracy of ML models for different algorithms

Wordcount		Terasort		K-Means	
Model	MAPE	Model	MAPE	Model	MAPE
Linear	0.2345	Linear	0.3049	Linear	0.2500
MLR-I	0.2445	MLR-I	0.21985	MLR-I	0.2900
MLR-Q	0.2310	MLR-Q	0.2998	MLR-Q	0.2508
SVM	0.2356	SVM	0.1701	SVM	0.2009
SVM tuning	0.2234	SVM tuning	0.0876	SVM Tuning	0.2152

Table 3. Performance tuning results for Applications on 20 GB data size with default settings on $(4 + 1)$ node cluster where, Ne: Number of executors, Nc: Number of cores per executor, Nm: Executor memory size in GB

Application	Default values	Execution time on default values	Parameter optimal values	Optimal execution time (Gain%)
Wordcount	Ne = 2, Nc = 1, Nm = 1	1165.1450	Ne = 10, Nc = 2, Nm = 4	377.02 (67%)
Terasort	Ne = 2, Nc = 1, Nm = 1	884.395	Ne = 4, Nc = 4, Nm = 1	664.81 (24%)
K-means	Ne = 2, Nc = 1, Nm = 1	14778.06	Ne = 8, Nc = 6, Nm = 12	875.479 (94%)

A statistical tool *R* is used to build performance prediction model using various algorithms such as Multiple Linear Regression (MLR), MLR with quadratic effect and SVM with and without tuning [14]. These ML model are used for predicting application execution time on 20 GB data size. As shown in Table 2, these algorithms are able to predict with Mean Absolute Percentage Error (MAPE) 22% on average. These performance prediction models are integrated with optimization algorithm in *R* and could yield up to 94% improvement in application performance as shown in Table 3. Machine Learning based prediction models requires lot of resources and time for data collection. Agile development framework does not allow time delay incurred in collecting training data for building performance assurance model. Therefore, we have built analytic model using one time measurements as discussed in the next section.

4 Measurement Based Analytic Performance Prediction Model

We assume a small size Spark cluster with application and its representative data sets available in development environment. The cluster is assumed to have atleast one instance of each type of heterogeneous nodes deployed in production system. The application is executed in this small cluster on small data size (*DevSize*). The application logs created by Spark platform are parsed to collect granular level performance data as given in Table 4. The problem statement is to estimate the application execution time for production environment, having larger data size (say *ProdSize*) and larger cluster size (say CS_{prod}) with different Spark parameter configurations, using the collected measurements. We will use notations given in Table 4 for further explanation of the model. An application is executed as a serial execution of a number of Spark jobs as shown in Fig. 2. Therefore, the application's predicted execution time is summation of the estimated execution time of its jobs launched one after another i.e.

$$ApplnExecutionTime = \sum_{i=0}^{i=N} pJobT_i \tag{1}$$

A Spark job is executed in a form of directed acyclic graph (DAG), where each node in the graph represents a stage. A new stage is created whenever next operation requires data to be shuffled. A job's execution time is predicted as summation of the estimated execution time of all its stages i.e.

$$pJobT_i = JobSt_i + \sum_{k=0}^{k=SN_i} pStageT_i^k + JobCln_i \tag{2}$$

Each stage is executed as set of concurrent executors with parallel tasks in each executor, depending on values of *number of executors* and *number of cores per executor* parameters. If executors allocated per node (i.e. $\frac{NE_p}{CS_p}$) can not be scheduled concurrently, due to non-availability of cores on the node (i.e. $\frac{NE_p}{CS_p} * NC_p <$ node's available cores), then the executors are serialized and executed one after another, and the stage's estimated execution time is increased by factor of the number of serialized executors. For simplification, we assume that Spark parameter configuration is such that executors are running concurrently at each node. Each executor spawns multiple threads, one for each task. All tasks launched in an executor share the same JVM memory. Each task processes a defined size of data set (i.e.block size). For a given data size, an executor may have multiple waves of such parallel tasks executions on each core. Since all tasks in a stage are identical and read same data size, therefore, execution time of stage 'j' of job 'i' may be estimated as:

$$pStageT_i^j = StgSt_i^j + Avg(pTskT_i^j) * \left\lceil \frac{pDS_i^j}{(BS_p * NE_p * NC_p)} \right\rceil + StgCln_i^j \quad (3)$$

However, variation in tasks' execution time may break the symmetry and number of tasks assigned per core (or wave count) may not be same at all cores. Variation in tasks' execution time could be due to data skew, heterogeneous nodes and/or variability in location of HDFS block(s) read by a task-local, same rack or remote. We have built a stage task execution simulator, using performance summary created in development environment (in Sect. 4.1), to capture this variability as discussed in Sect. 4.5. A task execution time constitutes scheduler delay, serialization time, de-serialization time, JVM overheads, compute time including IO read/write time in HDFS and shuffle IO time. Note that each task reads either shuffled data or input data and writes shuffled data or output data. Therefore execution time of a task in stage 'j' of job 'i' is estimated as:

$$pTskT_i^j = pTskSd_i^j + pTskSer_i^j + pTskCt_i^j + pTskJvm_i^j + pTskSf_i^j \quad (4)$$

A task's serialization and de-serialization time depends on amount of data processed by a task, which depends on the block size. Since this is a compute operation, it can be assumed to increase linearly with block size. For same block size in both the environments, $pTskSer_i^j = dTskSer_i^j$. A task's JVM time represents the overhead in garbage collection while managing multiple threads. The JVM time estimation depends on type of computation, hardware system and number of threads, which is discussed in detail in Sect. 4.3. For a given Spark cluster, an increase in input data size may increase a task's shuffle data such that it may not fit in the allocated memory. This results in spill over to disk and may increase shuffle time non-linearly because of additional disk read and write operations. We need a model to estimate shuffle read and write time as function of input data size, cluster size and shuffle memory as discussed in Sect. 4.4.

4.1 Performance Summary

We have observed that variation in task execution time also relates to its launch time on a core. To capture this variation, we divided a stage tasks into two types

Table 4. Notations used in the analytic model discussed in Sect. 4

Parameter	Development	Production
Block size	BS_{dev}	BS_{prod}
Number of executors	NE_d	NE_p
Number of cores per executor	NC_d	NC_p
Number of jobs in the application	N	N
Number of stages in job ‘i’	SN_i	SN_i
Job ‘i’ execution time	$dJobT_i$	$pJobT_i$
Job ‘i’ start up time	$JobSt_i$	$JobSt_i$
Job ‘i’ clean up time	$JobCln_i$	$JobCln_i$
Job ‘i’ stage ‘j’ execution time	$dStageT_i^j$	$pStageT_i^j$
Job ‘i’ stage ‘j’ startup time	$StgSt_i^j$	$StgSt_i^j$
Job ‘i’ stage ‘j’ cleanup time	$StgCln_i^j$	$StgCln_i^j$
Job ‘i’, stage ‘j’, number of tasks	dNT_i^j	pNT_i^j
Job ‘i’, stage ‘j’, size of shuffled data	dDS_i^j	pDS_i^j
Job ‘i’, stage ‘j’, task Execution Time	$dTskT_i^j$	$pTskT_i^j$
Job ‘i’, stage ‘j’, task execution time in ‘k’th wave	$dTskT_i^{jk}$	$pTskT_i^{jk}$
Job ‘i’, stage ‘j’, task serialization + de-serialization time	$dTskSer_i^j$	$pTskSer_i^j$
Job ‘i’, stage ‘j’, task JVM time	$dTskJvm_i^j$	$pTskJvm_i^j$
Job ‘i’, stage ‘j’, task shuffle IO time	$dTskSf_i^j$	$pTskSf_i^j$
Job ‘i’, stage ‘j’, task scheduler delay	$dTskSd_i^j$	$pTskSd_i^j$
Job ‘i’, stage ‘j’, first wave task compute time	$dFstTskCt_i^j$	$pFstTskCt_i^j$
Job ‘i’, stage ‘j’, first wave task scheduler delay	$dFstTskSd_i^j$	$pFstTskSd_i^j$
Job ‘i’, stage ‘j’, rest wave ‘k’th bucket duration	$dRstTkBktDur_i^{jk}$	$pRstTkBktDur_i^{jk}$
Job ‘i’, stage ‘j’, number of rest wave tasks in ‘k’th bucket	$dRstTkBktN_i^{jk}$	$pRstTskBktN_i^{jk}$
Job ‘i’, stage ‘j’, rest wave task compute time	$dRstTskCt_i^j$	$pRstTskCt_i^j$
Job ‘i’, stage ‘j’, rest wave task scheduler delay	$dRstTskSd_i^j$	$pRstTskSd_i^j$
Job ‘i’, stage ‘j’, rest wave task maximum compute time	$dRstTkMaxCT_i^j$	–
Job ‘i’, stage ‘j’, rest wave task minimum compute time	$dRstTkMinCT_i^j$	–

of tasks - first wave tasks and rest wave tasks as shown in Fig. 2 by emulating the task scheduling behaviour of Spark platform across $NE_d * NC_d$ cores. An application log is parsed to collect list of all $dTskT_i^j$ sorted in the order of their launch time. An array of data structure of size $NE_d * NC_d$ is allocated with each ‘k'th element storing the ‘k'th core current finish time. Initially all elements are initialized to zero. dNT_i^j tasks are scheduled on $NE_d * NC_d$ cores such that the next task in the list is scheduled on the core having minimum finish time, leading to a task allocation structure as shown in Fig. 2. Using the measurements collected from the Spark application log, $dFstTskCT_i^j$ and $dFstTskSd_i^j$ are computed as

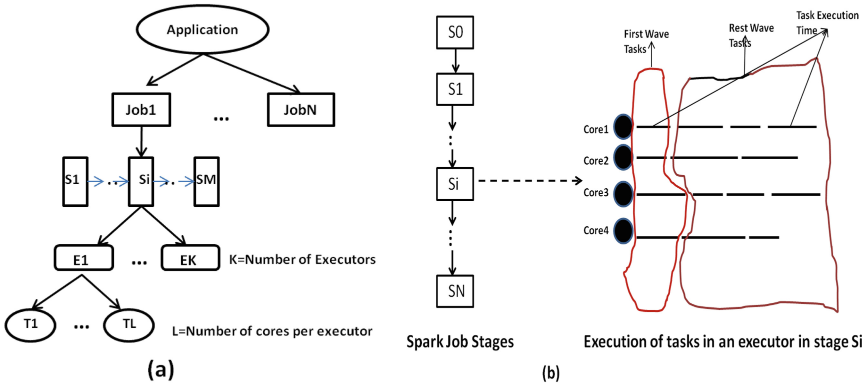

Fig. 2. (a) Application execution on Spark (b) Tasks execution in an executor for a stage Si with 4 cores

average of $(dTskT_i^j - dTskJvm_i^j - dTskSf_i^j - dTskSd_i^j)$ and average of $dTskSd_i^j$ respectively of all the tasks in the first wave. Similarly, $dRstTskSd_i^j$ is calculated as the average of scheduler delay of all tasks in the rest wave. $dRstTskCT_i^j$ is also computed as histogram of $(dTskT_i^j - dTskJvm_i^j - dTskSf_i^j - dTskSd_i^j)$ for all rest wave tasks, to capture variability in task execution time. The histogram has 'm' buckets each of size $BkSize_i^j$ such that

$$BkSize_i^j = \frac{(dRstTskMaxCT_i^j - dRstTskMinCT_i^j)}{m} \tag{5}$$

The 'k'th bucket duration is from $(k-1) * BkSize_i^j$ to $k * BkSize_i^j$. Each of the rest wave tasks is categorized into one of 'm' buckets such that $dRstTskCT_i^j$ falls into the duration of the bucket. $dRstTskBktDur_i^{jk}$ is computed as average of $dRstTskCT_i^j$ for all tasks in 'k'th bucket. Performance summary of stage 'j' of job 'i' consists of $dFstTskCT_i^j$ and 'm' buckets each with its average duration $dRstTskBktDur_i^{jl}$ and $dRstTskBktN_i^{jl}$ number of tasks in 'l'th bucket. Higher the value of 'm', more variation in task execution time can be captured. However, it may also increase the time taken to mimic scheduler for rest wave tasks, whose time complexity is O(n + m) for 'n' tasks.

4.2 Task Scheduler Delay Prediction Model

Scheduler delay is the delay incurred while scheduling a task. We have observed larger scheduler delay for first wave tasks due to task scheduling preparation overheads. Therefore,

$$\begin{aligned} pFstTskSd_i^j &= dFstTskSd_i^j * \frac{pNT_i^j}{dNT_i^j} \\ pRstTskSd_i^j &= dRstTskSd_i^j \end{aligned} \tag{6}$$

4.3 Task JVM Time Prediction Model

On Spark platform, each executor has single JVM and all tasks scheduled in it share the same JVM, therefore JVM overheads increases with increase in the number of concurrent tasks (threads) accessing the same JVM which is controlled by number of cores per executor parameter. Also, we have observed in our experiments that it increases linearly with number of executors scheduled concurrently on the same machine. This may be because a JVM manager has more JVM instances to manage and overheads are assumed to increase linearly for the model. These overheads are system and application dependent, so we model the JVM overheads as function of number of cores per executor by taking average of measured $dTskJvmT_i^j$. The measurements are taken by varying number of cores per executor and keeping only one executor per machine. We use regression to estimate JVM overheads for NC_p cores per executor in the production environment. For example, Fig. 3 shows the JVM model used in our experimental setup for Wordcount and Terasort applications for one executor per machine. $pTskJVM_i^j$ is further extrapolated linearly to the number of concurrent executors per node.

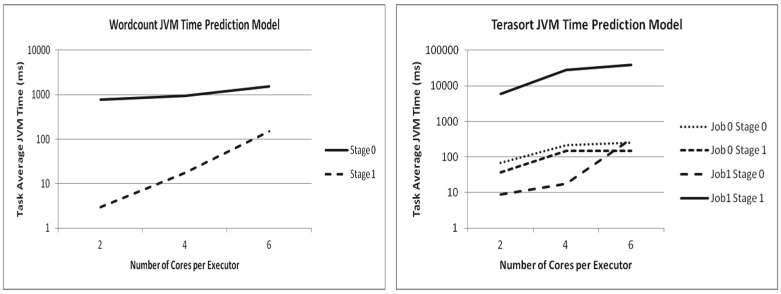

Fig. 3. JVM time estimation model for Wordcount and Terasort applications built on experimental set up given in Sect. 5

4.4 Task Shuffle Time Prediction Model

A naive approach to estimate a task shuffle time is linear extrapolation i.e.

$$pTskSf_i^j = \frac{\sum_{\forall_{tasks}} dTskSf_i^j}{dNT_i^j} * \frac{pDS_i^j}{dDS_i^j} \tag{7}$$

However, it may hold true only for those configurations of production system where a task's shuffle data size is small enough to fit in the allocated memory. Otherwise, shuffle operation leads to spill over to disk and incurs extra disk IO read/write operations for a task. We model this by estimating shuffle data size per task and predict if this will lead to spill over. If it does, we estimate the overheads of spill over and add that to a tasks shuffle time. Spill overheads are calculated at small data size by constraining the development environment to generate spurious disk spills. For simplicity, we assume that network is not a bottleneck here, so communication overhead increases linear to shuffle data size. Assuming block size unchanged, the size of shuffle data generated per task remains same.

Shuffle data size per executor is estimated as $(\frac{pDS_i^j}{NE_p})$. We have observed that shuffle data size in memory increases due to de-serialization. For an executor, if this increased size is more than allocated shuffle memory (i.e. storage memory fraction * executor memory size), the shuffle operation will spill over to disk for the executor tasks. Let say $OptS_i^j$ is the largest shuffle data size per executor which fits into allocated shuffle memory after serialization and does not spill over to disk, then for measured spill overheads as $Spill$ (in MB) per task in the development environment and $\frac{pDS_i^j}{NE_p} > OptS_i^j$,

$$pTskSf_i^j = \frac{\sum_{\forall_{tasks}} dTskSf_i^j}{dNT_i^j} * \frac{pDS_i^j}{dDS_i^j} + \left(\frac{pDS_i^j}{NE_p} - OptS_i^j\right) * Spill \\ \text{where}, pDS_i^j = dDS_i^j * \frac{Prodsize}{DevSize} \tag{8}$$

4.5 Stage Task Execution Simulation

To estimate execution time of a stage, we need to estimate number of tasks, pNT_i^j, and their estimated execution time i.e. $pTskT_i^j$. pNT_i^j is estimated as $\frac{pDS_i^j}{BS_p}$ where, pDS_i^j is given in Eq. 8. As mentioned in Sect. 4.1, a stage tasks are divided into first wave and rest wave tasks, therefore we estimate average execution time for both the waves' task separately using the performance summary (Sect. 4.1) created in the development environment and prediction models discussed in Sects. 4.2, 4.3 and 4.4. Using Eq. 4, for first wave tasks,

$$pTskT_i^j = dFstTskCT_i^j + pFstTskSd_i^j + dTskSer_i^j + pTskJvm_i^j + pTskSf_i^j \tag{9}$$

Similarly, for rest wave tasks,

$$\begin{aligned} \forall_{l=(1,m)}, pRstTskBktDur_i^{jl} &= dRstTskBktDur_i^{jl} + pRstTskSd_i^j + \\ &\quad dTskSer_i^j + pTskJvm_i^j + pTskSf_i^j \\ \forall_{l=(1,m)}, pRstTskBktN_i^{jl} &= \frac{(pNT_i^j - NE_p * NC_p)}{(dNT_i^j - NE_d * NC_d)} * dRstTskBktN_i^{jl} \end{aligned} \tag{10}$$

Stage execution is simulated by scheduling pNT_i^j tasks across $NE_p * NC_p$ number of cores. The simulator maintains an array of data structure of size $NE_p * NC_p$ with each 'k'th element storing the 'k'th core current finish time. $NE_p * NC_p$ tasks are allocated as the first wave tasks of duration given in Eq. 9 to each of the cores. Then, all the rest wave tasks are scheduled from each of 'm' buckets of duration given in Eq. 10 in round robin fashion such that a task is scheduled on the core having minimum finish time so far. Therefore,

$$\begin{aligned} pStageT_i^j &= StgSt_i^j + \text{Max on } T \text{ cores} \sum_{\text{`}k\text{'}thCoreTasks} pTskT_i^{jk} + StgCln_i^j \\ where, \quad T &= NE_p * NC_p \end{aligned} \tag{11}$$

5 Experimental Results and Analysis

Our experimental setup consists of 5 nodes, each of Intel(R) Xeon(R) CPU X5365 @ 3.00 GHz, 8 cores and 16 GB RAM. Each node has disk capacity of 30 GB. The platform stack consists of Yarn, Apache Spark 2.01, Hive 1.2.1 and HDFS 2.6. We have one master and maximum four slaves in these experiments. We have kept executor memory as 4 GB across all experiments in both the development and production environments. However, model supports different executor memory size as well. The different experimental configuration are shown in Table 5. We have tested the prediction model for four types of workloads- Wordcount, Terasort, K-means [1], two SQL queries from TPC-H [2] benchmarks, for data sizes varying from 5 GB to 20 GB. The development environment consists of 1 + 2 node cluster with 5 GB data size. We have executed each application on 1 node cluster by varying *–executor-cores* parameter to build JVM model for each application as shown in Fig. 3. Each workload listed in Table 5 is executed in the development environment to build the model as discussed in Sect. 4. The analytic model is built in Java. It has two components - Parser for parsing the Spark application log and Prediction module for building the prediction models which takes input from the parser to build the model. Equation 1 is used to predict each application execution time for different production environments created by possible combinations of parameters listed in Table 5.

Table 5. Production system configuration for model validation

Configuration parameter	Values
Number of executors (–num-executor)	2, 4, 6
Number of cores per executor (–executor-cores)	2, 4, 6
Executor memory size	4 GB
Cluster size	2, 4
Data size	10 GB, 20 GB
Workload	Wordcount, Terasort, Kmeans
SQL1	select sum(l_extendedprice * (1 – l_discount)) as revenue from Lineitem
SQL2	select sum(l_extendedprice * (1 – l_discount)) as revenue from Lineitem, Order where l_orderkey = o_orderkey

5.1 Discussions

We have validated the model for around 15 production configurations for each of the workloads. Prediction error is calculated as the ratio of the absolute difference in the actual execution time and the model's estimated execution time, to the actual execution time. We have observed an average 15% prediction error for each application as shown in Fig. 4 with maximum 30% error. We have observed that large prediction errors are due to gaps in capturing variations in tasks execution time i.e. $pTskCt_i^j$ in Eq. 4. The analytic model accuracy has been compared with that of the machine learning model proposed in Sect. 3 for 20 GB data size on 4 nodes cluster for four different configurations of Spark platform parameters. Figure 5 shows that prediction accuracy of the analytic model is better than that of the machine learning model. This is because the ML model uses black box techniques while the analytic model is based on Spark internal job processing details. Figure 6 shows the actual execution time vs. predicted execution time for different production environments for Wordcount, Terasort and K-means applications.

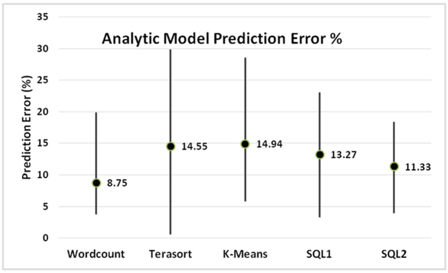

Fig. 4. Analytic model average prediction error(%)

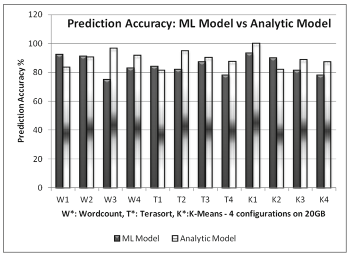

Fig. 5. ML vs analytic model accuracy

Wordcount application has only one job with two stages. It is a simple map-reduce application, where the proposed model's estimations are very close to the actual execution time. We have observed an average accuracy of 91%. Terasort is a sorting application with two jobs and two stages in each job. For most of the test cases we observed an accuracy of atleast 80%, however there is one outlier on 4 node cluster with 4 executors and 4 cores per executor, where the estimated execution time is 30% more than the actual. This is because for stage 4, where partially sorted data sets are merged and written back to disk, the model estimates more number of tasks with larger execution time than the actual. This is due to uniform extrapolation of number of tasks in each bucket which may need to be refined using data distribution. K-means application has around 20 jobs, each job with 2 stages. Here, we observed accuracy of 85% percent. Few outliers with at most error of 23% are due to the variation in task execution time which may not be captured in the histogram for few jobs. The proposed

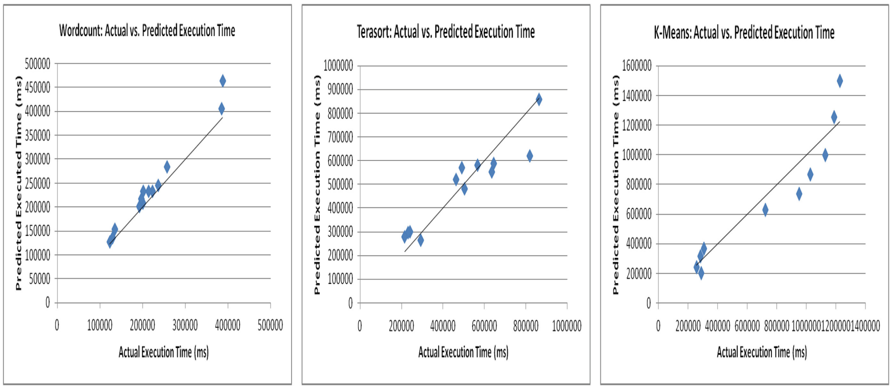

Fig. 6. Model validation for Wordcount, Terasort and K-Means applications: Better accuracy for points closer to the line

simulator uses only four buckets irrespective of type of job or stage- this may need to be tuned for better accuracy.

We have also validated the model for two simple SQL queries based on TPC-H benchmarks as shown in Fig. 7. The model may not work for complex SQL queries having multiple joins. The optimization in Spark 2.0 may lead to execution of multiple steps of a complex SQL query in a single stage and difficult to get performance data for each step of SQL query. Whereas, a SQL query execution time is sensitive to each join operator's input data sizes, which is not being considered in our model. SQL1 query is more like an aggregation which has one job with two stages and SQL2 query has one aggregate and one join operation which is executed as one job with 4 stages. As shown in Fig. 7 the estimated values for both SQL queries are closer to the actual value with accuracy of 90%.

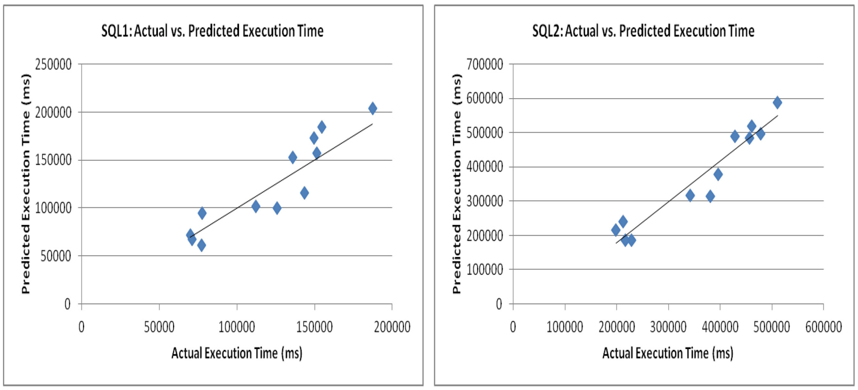

Fig. 7. Model validation for TPC-H SQL queries: Better accuracy for points closer to the line

6 Auto Tuning of Application Execution on Spark

The performance prediction model presented in Sect. 4 can be used iteratively in an optimization algorithm to get Spark parameters values for an optimal performance (i.e. minimum execution time) on a given cluster and data size as given in Fig. 8. The parameters we have considered for tuning are number of executors, number of cores per executors and executor memory size. Note that, there is a scope to include more performance parameters as discussed in [8], however, we have restricted the model for these three parameters only in this paper.

```
OptimizationModule(Input: DataSize, ClusterSize)
{
Optimal_time = 9999;

For Numexecutor = 1 to max cores in the Cluster do
For Numcore_Executor = 1 to max  cores on node do
For Executormemory =  Min Size to RAM size on node do

If ValidConfiguration(cluster size, numexecutor, numcoreExecutor, Executormemory)
{
 Time = PredictTimeModel(DataSize, ClusterSize, Numexecutor, NumcoreExecutor, Executormemory)
 if  Time < optimal_time
 {
    optimal_Numexecutor = Numexecutor
    optimal_NumcoreExecutor = NumcoreExecutor
    optimal_Executormemory = Executormemory
    optimal_time = Time;
 }
}
Done
Done
Done
Return (optimal_Numexecutor, optimal_NumcoreExecutor, optimal_Executormemory)
}
```

Fig. 8. Auto tuner: Optimization of application execution using prediction model

7 Conclusions and Future Work

Spark is a widely deployed commodity based parallel processing framework. The challenge is to assure performance of applications on Spark cluster for larger data size before deployment. In this paper, we have presented a model to predict application execution time for larger data size using finite measurements in small size development environment. We have presented both machine learning based approach and analytic model. The analytic model handles data skew and node heterogeneity by building a simulator for estimating Spark's stage execution time. The analytic model is flexible to different Spark configurations since it also estimates execution time of all components of Spark's task as function of the Spark production cluster's configuration. This capability of the model may be harnessed to build auto tuner for applications deployed on Spark platform. The proposed model shows prediction accuracy of atleast 80% for different workloads. There is scope to extend the model to support more parameters as discussed in [8]. Further work is needed for extensive validation of the model for different applications, more combinations of Spark parameters, larger data size, larger cluster size and also for cloud deployments. We also plan to create synthetic benchmarks which can be matched to a given applications to enhance the model's prediction capability for an unknown application.

References

1. SparkBench: Spark performance tests. https://github.com/databricks/spark-perf
2. TPC-H benchmarks. https://www.tpc.org/tpch
3. Awan, A.J., Brorsson, M., Vlassov, V., Ayguade, E.: How data volume affects spark based data analytics on a scale-up server. arXiv:1507.08340 (2015)
4. Awan, A.J., Brorsson, M., Vlassov, V., Ayguade, E.: Architectural impact on performance of in-memory data analytics: apache spark case study. arXiv:1604.08484 (2016)
5. Herodotou, H., Babu, S.: Profiling, what-if, analysis, and cost-based optimization of mapreduce programs. In: The 37th International Conference on Very Large Data Bases (2011)
6. Jia, Z., Xue, C., Chen, G., Zhan, J., Zhang, L., Lin, Y., Hofstee, P.: Auto-tuning spark big data workloads on POWER8: prediction-based dynamic SMT threading. In: Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (2016)
7. Ousterhout, K., Rasti, R., Ratnasamy, S., Shenker, S., Chun, B.: Making sense of performance in data analytics frameworks. In: Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2015) (2015)
8. Petridis, P., Gounaris, A., Torres, J.: Spark parameter tuning via trial-and-error. arXiv:1607.07348 (2016)
9. Shi, J., Zou, J., Lu, J., Cao, Z., Li, S., Wang, C.: MRTuner: a toolkit to enable holistic optimization for mapreduce jobs. PVLDB **7**(13), 1319–1330 (2014)
10. Singhal, R., Nambiar, M.: Predicting SQL query execution time for large data volume. In: ACM Proceedings of IDEAS (2016)
11. Singhal, R., Sangroya, A.: Performance assurance model for HiveQL on large data volume. In: International Workshop on Foundations of Big Data Computing in conjunction with 22nd IEEE International Conference on High Performance Computing (2015)
12. Singhal, R., Verma, A.: Predicting job completion time in heterogeneous mapreduce environments. In: Proceedings of IPDPS: Heterogeneous Computing Workshop, IPDPS (2016)
13. Wang, K., Khan, M.M.H.: Performance prediction for apache spark platform. In: IEEE 17th International Conference on High Performance Computing and Communications (HPCC) (2015)
14. Yigitbasi, N., Willke, T., Liao, G., Epema, D.: Towards machine learning-based auto-tuning of mapreduce. In: IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (2013)