Deadline-Aware Cost Optimization for Spark

Subhajit Sidhanta, Wojciech Golab, and Supratik Mukhopadhyay

Abstract—We present OptEx, a closed-form model of job execution on Apache Spark, a popular parallel processing engine. To the best of our knowledge, OptEx is the first work that analytically models job completion time on Spark. The model can be used to estimate the completion time of a given Spark job on a cloud, with respect to the size of the input dataset, the number of iterations, and the number of nodes comprising the underlying cluster. Experimental results demonstrate that OptEx yields a mean relative error of 6% in estimating the job completion time. Furthermore, the model can be applied for estimating the cost-optimal cluster composition for running a given Spark job on a cloud under a completion deadline specified in the *SLO* (i.e., Service Level Objective). We show experimentally that OptEx is able to correctly estimate the required cluster composition for running a given Spark job under a given SLO deadline with an accuracy of 98%. We also provide a tool which can classify Spark jobs into job categories based on bisimilarity analysis on lineage graphs collected from the given jobs. This tool can further be applied to compare Spark jobs in terms of their lineage and derive interesting insights that can be leveraged for optimizing job execution.

Index Terms—distributed systems, parallel processing, distributed file systems, middleware, performance evaluation, reliability, availability, and serviceability

1 Introduction

Optimizing the cost of usage of cloud resources for running data-intensive jobs on large-scale parallel processing engines is an important, yet relatively less explored problem. Cloud service providers, like Amazon, Rackspace, Microsoft, etc., allow users to outsource the hosting of jobs and services to a cloud using clusters of *virtual machine instances*. The cloud service providers charge a *service usage cost* to the user on the basis of the hourly usage [1] of the virtual machine instances. The cloud service providers present the users with a variety of virtual machine instance types to choose from, such as micro, small, large, etc., for Amazon ec2 [1]. Each virtual machine instance type has a different specification, in terms of CPU, I/O, etc., and different hourly usage cost. The *cost-optimal cluster composition* specifies a number of virtual machine instances (of different virtual machine instance types), that enable execution of the given job under the SLA (i.e., Service Level Agreement) deadline, while minimizing the service usage cost. However most of the current state-of-the-art [2, 3, 4, 5] resource provisioning solutions are designed specifically for Hadoop mapreduce jobs. Other researchers [6, 7] have proposed resource allocation strategies that does not deal with the dual objective of meeting a given SLA deadline for job completion and ensuring that the service usage cost is minimized.

We present OptEx, a closed-form job execution model for Apache Spark [8], the most popular parallel processing engine [9, 10]. OptEx can be used to determine a cost-optimal cluster composition, comprising virtual machine instances provided by cloud service providers for executing a given Spark job under an SLA deadline. As far as we know, OptEx is the first work that analytically models job execution on Spark. We consider Spark applications that are developed for real world industry use cases comprising event-based applications such as click stream, logs, transactional systems, IOT, Twitter, and several others. An example is an event processing system which performs live weather reporting from real-time sensor data [11]. Such applications often have a strict deadline for completion. We consider this deadline as the SLA deadline for OptEx. A real-world data analytics example from the industry is EVAM [12], a real-time streaming analytics, which has a deadline of 50 milliseconds. OptEx decomposes the execution of a target Spark job into smaller phases,and models the completion time of each phase in terms of: 1) the cluster size, the number of iterations, the input dataset size, and 2) certain model parameters estimated using *job profiles*. OptEx categorizes Spark applications into job categories, and generates separate job profiles for each job category by executing specific *representative jobs*. The model parameters for the target job are estimated from the components of the job profile, corresponding to the job category of the target job. Experimental results demonstrate that OptEx yields a mean relative error of 6% in estimating the job completion time.

Using the model of job completion time (OptEx), we derive an objective function for minimizing the service usage cost for running a given Spark job under an SLA deadline. The cost-optimal cluster composition for running the target Spark job under the SLA deadline is obtained using constrained optimization on the above objective function. Experimental results demonstrate that OptEx is able to correctly estimate the cost-optimal cluster composition for running a given Spark job under an SLA deadline with an accuracy of 98%. We also demonstrate experimentally that OptEx can be used to provision a cluster to finish a given Spark job within an SLA deadline under the constraints of a given budget. For lack of space, we have included an extended version of the paper in github [13].

Consider the use case where a web development company needs to run a Spark PageRank job spanning 10 iterations to determine the most important web pages they developed over the years from a dataset comprising 10 million datapoints, using the infrastructure (cluster) provided by a popular cloud provider, like Amazon. Using state-of-the-art provisioning techniques that resort to prior experience for making decisions about provisioning , the

- *S. Sidhanta is currently with Indian Institute of Technology Jodhpur. This work was done while S. Sidhanta was with INESC-ID, Lisbon. E-mail: subhajit@iitj.ac.in*
- *S. Mukhopadhyay is with Louisiana State University, USA. Email: supratik@csc.lsu.edu*
- *W. Golab is with University of Waterloo, Canada. Email: wgolab@uwaterloo.ca*

Manuscript received March 5, 2018; revised May 5, 2018.

company may provision a cluster of 30 m4.xlarge Amazon ec2 instances to run the Spark job, under an SLA deadline of 70 hours. In this case, they may end up actually finishing the job in 40 hours, incurring a service usage cost of $168.45 (at the hourly rate of 0.1403 using the pricing scheme from [1]). However, with OptEx, the job would have completed in 60 hours using only 10 m4.xlarge Amazon Ec2 nodes, incurring just $84.18, while satisfying the deadline. Thus, OptEx helps minimizing the service usage cost without violating the SLA deadline. The technical contributions of this work are summarized as follows.

- We present OptEx, an analytical model for Apache Spark [8] job execution. Following an analytical modelling approach, OptEx OptEx decomposes a Spark job execution into different phases, and expresses the execution time of each phase in terms of the cluster size, the job iteration length, the input dataset size, and certain model parameters.
- We demonstrate that error of estimating completion time of Spark jobs, expressed in terms of the standard deviation of estimated job completion times with respect to the observed job completion time, falls within a range of 0.7 seconds.
- We also provide a tool which can categorize Spark jobs into equivalence classes by performing bisimulation on lineage graphs collected while running the given jobs.

2 Spark Job Execution Phases

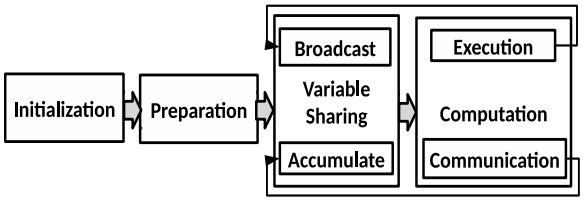


Fig. 1: Phases in a Spark Job Execution Flow

We decompose a typical Spark job execution flow into logically distinct phases illustrated in Figure 1. The first phase in a Spark job is the *initialization* phase, which performs activities like class loading, symbol table creation, object initialization, function loading, and logger initialization. The second phase is the *preparation phase*, which is responsible for job scheduling, resource allocation, and context creation. The initialization and preparation phases are relatively invariant to changes in input variables [8]. The next phase is the *variable sharing phase* that deals with broadcasting or accumulating blocks of data from the Spark master to the workers.

Internally, Spark processes data using a novel in-memory data structure called the *RDD* (i.e., resilient distributed dataset) for fast and fault tolerant computation [8]. Spark provides a wide range of built-in unit RDD operations, packaged within several library modules [14], like MLlib, Spark SQL modules, etc. During the last phase, i.e., the *computation phase* (Figure 1), the given job makes calls to methods from the above library modules, which in turn triggers the respective unit RDD operations on the workers. The computation phase comprises: 1) the *communication phase* that communicates the intermediate variables among the workers, and 2) the *execution phase* that involves the actual execution of the unit RDD operations on the workers. The loops in the rectangles corresponding to the broadcast phase and execution phase represent iterations in iterative jobs. The lengths of the variable sharing phase and the computation phase monotonically increase with the input variables, i.e., the number of iterations, and the dataset size [8]. In particular, the variable sharing phase and the computation phase are repeated under iterations, and the lengths of the above phases increase with respect to number of iterations.

3 Job Categorization and Job Profiles

OptEx categorizes Spark jobs, and following in the footsteps of [3] applies profiling to generate separate job profiles for each job category with representative jobs for each category. As discussed in the extended version, one of the most distinguishing feature of Spark is that it stores the lineage information for the RDDs generated and processed during the execution of a job. We leverage this lineage information to categorize Spark jobs into job categories. Internally, the lineage information for a job is processed in the form of a directed acyclic graph (DAG). Each RDD action invoked from the source code of a given Spark job accepts one or more RDDs, and transforms it into another RDD. Thus, each RDD action corresponds to two nodes in the DAG, a parent node and a child node, connected by a labelled directed edge representing the dependency between the two RDDs. The parent node represents the RDD on which the RDD action is performed. The child node represents the RDD which is created as a result of the action. The parent node, the child node, and the edge between them, are labelled by the name of the parent RDD, name of the child RDD, and the name of the RDD action, respectively. The lineage graphs are of finite size, bounded by the number of RDD actions in the job, which is typically a finite number. The comment about finite size is applicable even in the case of iterative jobs.

In our case the lineage graph is a particularly interesting artefact as it captures the information about how the data is manipulated in different stages of job execution. We can apply well known techniques from graph theory to analyse the structural equivalence of the lineage graphs of different Spark jobs. Particularly, we can check if the lineage graphs of given jobs are bisimilar, i.e., there exist bisimulations[1] among the graphs, to cluster them into different categories. Checking for bisimulations have been used by researchers [15, 16, 17] as an effective technique to determine equivalence between behavioural characteristics of two programs. Bisimulations have been successfully applied on graphical representations of concurrent program behaviours in the past [18, 19, 20]. Checking for strong and weak bisimulations are known to be $co\text{-}NP\text{-}hard$ and $\Pi_2^p\text{-}hard$ problems, respectively [20, 21]. However, Spark DAGs are of finite size, and in most cases the size is not very large. Thus even strong bisimulations can be performed on these DAGs with a considerably low overhead.

We provide a categorization tool (described in Section 7.1) that computes bisimulations on a sample space of spark DAGs collected from execution logs of example Spark jobs. This gives rise to a set of bisimilar equivalence classes of spark DAGs.

1. A bisimulation B is a binary relation $B \subseteq S \times S$ over a labelled transition system $\langle S, \Delta, \rightarrow \rangle$ such that both B and B^{-1} satisfy the following conditions. For each pair of elements $(p, q) \in B$, $\forall \alpha \in \Delta$, and for all $p' \in S$, $p \xrightarrow{\alpha} p'$ implies $\exists q' \in S$ such that $q \xrightarrow{\alpha} q'$ and $(p', q') \in B$. The same condition must be satisfied if we replace the occurrence of B with B^{-1} in the above condition.

TABLE 1: An Example Job Profile: Profile for Data Mining jobs, like movie recommendation, on m1.large instances

App	T_{init}(sec)	T_{prep}(sec)	$T_{vs}^{baseline}$(sec)	$coeff$	$T_{commn}^{baseline}$(sec)	cf_{commn}	T_{exec}	
							RDD task	M_a^k(ms)
ALS	20	13	15	0.004	11	0.070	mean	100
							map	98
							flatmap	72
							first	5
							count	124
							distinct	300

These equivalence classes form the job categories into which we segregate any given Spark job. Each job category is associated with a group of jobs characterised by respective DAGs that are bisimilar to one another. For a given job category, we choose one job from the group of sample jobs associated with it, and label it as the representative job for that category. We can choose any job from the group of sample jobs associated with a given job category as the representative job for that job category. Alterations in this choice would not have caused any changes in our results, since all jobs belonging a given category exhibit the same program behaviour characterised by its corresponding DAG. Then, we can configure the chosen representative job with any standard profiling tool to generate the job profile for the respective job category. We can configure the profiler to execute as a background process on the Java Virtual Machine while running the representative Spark job for a given job category in the foreground (refer to Section 7.3). We can collect the profile information in background from the logging module or the visualizer of the profiler. In the next section, we discuss how we use these job profiles to estimate the OptEx model parameters and coefficients. Before running the bisimulation algorithm, the categorization tool pre-processes the spark DAGs using various techniques which we elaborate in Section 7.1.

TABLE 2: Glossary of symbols and terms

T_{vs}	*Estimated completion time for the variable sharing phase*		T_{Est}	*Estimated job completion time*
n	*The cluster size*		T_{init}	*Estimated completion time for the initialization phase*
M_a^k	*Execution time of the k^{th} RDD operation for job A*		T_{prep}	*Estimated completion time for the preparation phase*
T_{Rec}	*Recorded execution time*		$\mathcal{I}$	*Number of iterations*
mT_{comp}	*Estimated completion time for the computation phase*		T_{commn}	*Estimated completion time for the communication phase in T_{comp}*
$T_{vs}^{baseline}$	*Baseline value of T_{vs}*		$coeff$	*Coefficient of T_{vs} in T_{Est}*
$T_{commn}^{baseline}$	*Baseline value of T_{commn}*		cf_{commn}	*Coefficient of T_{commn} in T_{Est}*

4 OptEx Model Based on Job Profiles

A Job profile consists of values of different parameters and coefficients collected during execution of a given job. Components of the job profile are used as estimates for the model parameters of the target job. The length of the initialization phase (refer to Figure 1)) T_{Init} and the length of the preparation phase T_{prep} remain constant to variations in the input variables [8]. The length of the execution phase (T_{exec}) and the length of the variable sharing phase (T_{vs}) increase monotonically with respect to the input variables [8]. Thus, the length of these phases in the execution of a representative job (contained in the job profile) can act as the point of reference, i.e., *baseline*, for measuring the length of the corresponding phases in the target job [3]. In this section, we elaborate how these baseline values in the job profile can be used for estimating the parameters of the model for each job phase.

During profiling, the representative job a is run on a single node, and the length of the initialization phase, the preparation phase, the variable sharing phase, and the communication phase (Figure 1) is recorded in the job profile. The lengths of the above phases in the job profile act as baseline values for estimating the lengths of the corresponding phases in a given target job. The length of the initialization phase T_{Init} (Table 1) and the length of the preparation phase T_{prep} for a given job are directly estimated from the lengths of the corresponding phases in the job profile (since, as discussed in Section 2, these phases remain constant with respect to the variations in the input variables). As elaborated in Section 2, the length of the variable sharing phase T_{vs} increases monotonically [8] with respect to the cluster size and the number of iterations. Hence, T_{vs} is expressed as a function of:

- The input variable n represents the number of nodes.
- The input variable $\mathcal{I}$ represents the number of iterations.
- The baseline value $T_{vs}^{baseline}$, contained in the job profile, representing the length of the variable sharing phase of the representative job. It is the baseline for estimating the length of the variable sharing phase T_{vs} of a given target job. ble

OptEx computes the total duration of the variable sharing phase across all iterations. The length of the variable sharing phase T_{vs} is a function of the baseline $T_{vs}^{baseline}$, the input variables n, and number of iterations $\mathcal{I}$. Thus T_{vs} is expressed as:

$$T_{vs} = coeff \times \mathcal{I} \times n \times T_{vs}^{baseline}, \quad (1)$$

where $\mathcal{I}$ is the number of iterations, n is the number of nodes, $T_{vs}^{baseline}$ is the baseline value, and $coeff$ is a coefficient term. The coefficient term $coeff$ is empirically estimated during job profiling using curve fitting on the results of repetitive experiments with the representative job. The length of the computation phase T_{comp} (Table 2) is made up of two logical components: the length of the communication phase T_{commn}, and the length of the execution phase T_{exec} (Figure 1).

The communication phase is responsible for fetching the values of the intermediate variables computed by operations in the earlier stages of the given job. While profiling, the length of the communication phase of a representative job a on a single node is recorded in the job profile as $T_{commn}^{baseline}$. It serves as the baseline measure against which the length of the communication phase T_{commn} is estimated. The size s of the input dataset is given in bytes (for example, the size of the input for the wordcount job is given as the combined size of the input files). Since the length of the communication phase T_{commn} increases with respect to the input variable s [8], T_{commn} is expressed as a product of

the input dataset size s, a coefficient cf_{commn}, and the baseline value $T^{baseline}_{commn}$, where cf_{commn} and $T^{baseline}_{commn}$ are obtained from the job profile. Again, the coefficient cf_{commn} is empirically estimated in the profiling stage by applying curve fitting on the outputs of experiments with the representative jobs. Thus,

$$T_{commn} = cf_{commn} \times T^{baseline}_{commn} \times s \tag{2}$$

As discussed in Section 3, the execution phase of a Spark job comprises a permutation of unit RDD operations. The OptEx job profile records the average running time M^k_a (Table 2) of each unit RDD operation component k comprising the representative Spark job a. If there are multiple occurrences of an RDD operation i in a, we consider the average running time for all occurrences of the operation i. By design, the representative job for a job category contains all the unit RDD operations comprising any given job in that category. Hence, the length of the execution phase of the given Spark job is estimated as a function of the average running time M^k_a of each unit RDD operation k in the job profile.

5 Derivation of the OptEx Model

Input Variables: OptEx accepts the following input variables: the size s of the input dataset in bytes, the number of nodes n in the cluster, and the number of iterations $\mathcal{I}$ in the given job [8]. OptEx only considers those iterative jobs for which the number of iterations is either passed as a runtime argument by the developer or can be determined from the source code [8]. Moreover, Spark jobs typically have only few lines of code. Hence if we need to determine $iter$ from the code, we do not require sophisticated techniques involving static analysis [22]. The other input variables, i.e., number of nodes n and input dataset size s, are also directly passed to the model as runtime arguments.

While modelling the estimated total completion time for the target job, the user provides an estimated upper bound for the number of iterations $\mathcal{I}$ for the target job, as an input to the model. During the actual running time of the target job, the user provides the number of iterations $\mathcal{I}_{exec}$ as a runtime argument to the job [8]. The number of iterations $\mathcal{I}_{exec}$ provided in the running time may differ from the number of iterations $\mathcal{I}$ provided in the modelling phase. The difference between $\mathcal{I}_{exec}$ and $\mathcal{I}$ may cause: 1) unpredicted wastage of cluster resources, and 2) the failure to satisfy the SLO. In that case, the estimations need to be redone, with a new input value for the number of iterations. For multiple runs of the target job with different values of the runtime argument $\mathcal{I}_{exec}$ supplied by the user in each run, the maximum of the $\mathcal{I}_{exec}$ values, i.e., $\mathcal{I}^{max}_{exec}$, is supplied as the new input for the estimation. The estimation using the new value $\mathcal{I}^{max}_{exec}$ amounts to computing the value of T_{Est} from the Equation 8 with a time complexity of $\Theta(1)$ (since the degree of T_{Est} is 1 [23]), thus incurring negligible overhead.

Formulation of the Model: OptEx decomposes the job completion time into four phases (Figure 1), and models the total job completion time T_{Est} as the sum of the lengths of the component phases. Thus,

$$T_{Est} = T_{Init} + T_{prep} + T_{vs} + T_{comp}, \tag{3}$$

where T_{Init}, T_{prep}, T_{vs}, and T_{comp} are the lengths of the initialization phase, preparation phase, variable sharing phase, and computation phase, respectively. As discussed in Section 7.3, the execution phase of a given Spark job comprises a permutation of low-level unit RDD operations. The number of unit RDD operations n_{unit} increases monotonically with increasing input dataset size s and number of iterations $\mathcal{I}$ [8]. Hence, the number of unit RDD operations n_{unit} can be expressed as a function of the size of the input dataset denoted by s, the number of iterations in the job denoted by $\mathcal{I}$, and the baseline term for the number of unit RDD operations denoted by $n^{baseline}_{unit}$. The value of the term $n^{baseline}_{unit}$ is obtained from the job profile (Section 4) as follows. Spark enables parallel execution by dividing the input dataset into partitions, and distributing the partitions/slices among the worker nodes [8]. $n^{baseline}_{unit}$ directly corresponds to the number of partitions that the input dataset is comprised of. The number of partitions can be: 1) computed from the size s of the input dataset and the number of iterations $\mathcal{I}$ [8], or 2) programmatically provided as a parameter to the built-in transformation method used to create the RDDs from the input dataset [8].

For example, the Spark ChiSqTest program, working on input files from a HDFS backend, divides the input dataset into as many partitions as the number of HDFS blocks comprising the input files. Consider a portion of the Popular Kids dataset in HDFS [24] consisting of 164 files, where the size of each file is less than the HDFS block size. Hence the number of partitions, and in turn the number of unit RDD operations is 164. Thus, the baseline $n^{baseline}_{unit}$ is 164.

Thus, the increase of n_{unit}, with respect to the above baseline $n^{baseline}_{unit}$, in terms of the parameters s and $\mathcal{I}$, is expressed as

$$n_{unit} = n^{baseline}_{unit} \times s \times \mathcal{I}. \tag{4}$$

As discussed already in Section 4, the length of the initialization phase T_{Init} and the length of the preparation phase T_{prep} are directly estimated from the corresponding components in the job profile (Section 4). As discussed in Section 2, the length of the variable sharing phase T_{vs} and the length of the computation phase T_{comp} vary with the input variables. Hence the length of the variable sharing phase T_{vs} (Equation 1) and the length of the computation phase T_{comp} are estimated as functions of the job profile components, and the input variables (Section 4). The expression for the length of the variable sharing phase T_{vs}, comprising the baseline value $T^{baseline}_{vs}$ and coefficient $coeff$ obtained from the job profile, is given by Equation 1.

The length of the computation phase T_{comp} in Equation 3 can be further decomposed into the following two logical components: **A)** T_{commn}: the length of the communication phase T_{commn} is obtained from the Equation 2, and **B)** T_{exec}: the length of the execution phase T_{exec} in Equation 3 comprises the actual execution of k RDD operations comprising the job on the worker nodes (Section 4). T_{exec} depends on various factors [8]: 1) the running times of the unit RDD operations comprising the given job, 2) the number of iterations $\mathcal{I}$, 3) the number of stages in the job, 4) parallelization of the job across the worker nodes, and 5) sharing of the RDD variables across the cluster. Hence, execution phase length T_{exec} is expressed as the sum over the estimated computation times of all unit RDD operations comprising j, along with coefficients accounting for the above factors. Thus, the length of the execution phase T_{exec} of job a, without taking into account the parallelization factor n, is given as:

$$T_{exec} = \mathcal{I} \times \sum_{k=1}^{n_{unit}} M^k_a, \tag{5}$$

where n_{unit} is the number of unit RDD operations given in Equation 4, M^k_a is the average job execution time of a unit RDD

operation k comprising the job a, and $\mathcal{I}$ is the number of iterations in the job.

Following prior work on modelling execution of parallel operations [3], the overall length of the computation phase T_{comp} is divided by the factor n, taking into account the parallelization of the given job across the n worker nodes. Thus, the computation phase is rewritten as the sum of its two components, divided by n:

$$T_{comp} = \frac{T_{commn} + T_{exec}}{n}. \tag{6}$$

Combining the Equations 2, 5, and 6, we get

$$T_{comp} = \mathcal{I} \times \sum_{k=1}^{n_{unit}} M_a^k / n + \frac{A \times s}{n}, \tag{7}$$

where n_{unit} is the number of unit RDD operations given in Equation 4, and $A = \frac{cf_{commn} \times T_{commn}^{baseline}}{s_{baseline}}$.

Finally, combining the Equations 1 and 7 in Equation 3, the estimated total completion time for the target job is given as

$$T_{Est} = T_{Init} + T_{prep} + n \times \mathcal{I} \times C + \mathcal{I} \times B/n + \frac{A \times s}{n}, \tag{8}$$

where n_{unit} is the number of unit RDD operations given in Equation 4, $A = \frac{cf_{commn} \times T_{commn}^{baseline}}{s_{baseline}}$, $B = \sum_{k=1}^{n_{unit}} M_a^k$, and $C = coeff \times T_{vs}^{baseline}$.

6 Cost Optimal Cluster Composition

OptEx models the completion time T_{Est} (Equation 8) of a Spark job on a cluster comprising virtual machine instances provisioned from a cloud service provider, like Amazon (EC2), RackSpace, Microsoft, etc. The OptEx model is further used to estimate the cost optimal cluster composition for running a given job on virtual machine instances provided by any cloud provider, under the job completion deadline specified in the SLO, while minimizing the service usage cost. Let the optimal cluster size be given as $n = \sum_{t=1}^{m} n_t$, where n_t is the number of virtual machine instances of type t, and m is the total number of possible machine instance types (m depends on the instance offerings of the chosen cloud provider). Let total service usage cost of running the given job on the cloud be denoted by $\mathcal{C}$. Let c_t be the hourly cost of each machine instance of type t (c_t depends on the current rates charged by the chosen cloud provider), and T_{Est} be the estimated completion time of the given job (Equation 8). Our objective is to determine the cost optimal cluster composition for finishing the given Spark job within an SLO deadline with minimum service usage cost. This goal can be mathematically stated as: optimize the objective function

$$\mathcal{C} = \sum_{t=1}^{m} c_t \times n_t \times T_{Est}, \tag{9}$$

and obtain the cost optimal cluster composition, given as N_t =

$$\{n_t \,|\, 1 \leq t \leq m\},$$

under the constraint $T_{Est} < SLO$, where SLO is the given deadline, and T_{Est} is estimated using Equation 8.

We optimize the above objective function (Equation 9) and determine an optimal cluster configuration given by N_t, under the constraint $T_{Est} < SLO$. The above constraint involving T_{Est} is a convex nonlinear function [13] over n and $\mathcal{I}$ (Equation 8). The above optimization problem of minimizing the cost $\mathcal{C}$ under the nonlinear constraint $T_{Est} < SLO$ is solved using the Interior Point algorithm [23] which is one of the most efficient linear programming solvers till date. The solution to the above optimization problem enables: 1) estimating whether a given job will finish under the deadline SLO, 2) optimal job scheduling under the given deadline SLO, while minimizing cost C, and 3) estimating optimal cluster composition, given a cost budget $\mathcal{C}$ and an SLO.

7 Implementation and Evaluation

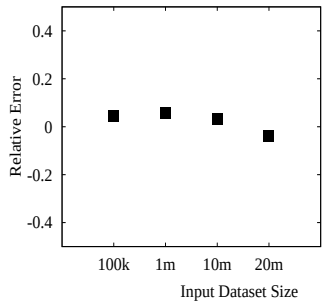
7.1 Implementation: Job Categorization Tool

We provide SparkRDDAnalyze, an open-source tool for determining equivalence relationship between a pair of given Spark applications based on their respective DAGs [25]. This tool is used to categorize Spark applications into job categories, and select representative jobs for each category. The tool uses the Construction and Analysis of Distributed Processes (CADP) toolbox [26]. The BISIMULATOR tool [27] comprised in CADP is used to check equivalence of pairs of DAGs. The tool is widely used by researchers in the programming languages and verification community [28]. We encode the DAGs in Binary Coded Graphs (BCG) file format, which is a popular format for representing Labelled State Transition (LTS) systems such as the Spark DAGs. Then we use the bcg_cmp method of BISIMULATOR to compare the two BCG files and check for bisimilarity between the two respective DAGs. Depending on the boolean output of bcg_cmp, we segregate given sample of Spark jobs into equivalence classes, which we refer to as job categories in this paper.

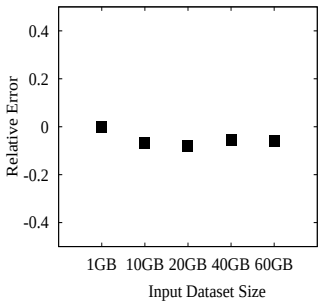
We have implemented two versions of the SparkRDDAnalyze tool, namely SparkRDDAnalyze-unlabelled and SparkRDDAnalyze-labelled. In SparkRDDAnalyze-unlabelled, we omit the edge labels of the DAGs in the pre-processing step before running the bcg_cmp method from the CADP tool. In SparkRDDAnalyze-labelled, we relabel the DAG edges in the pre-processing step according to the following rule. The Spark programming guide lists all available RDD transformations available in a current release of Spark, and includes a brief description of each transformation [29]. If a pair of RDD transformations corresponding to a given pair of DAG edge labels perform an identical task on a given input RDD, and produces as output a particular RDD dataset, according to the programming guide, these RDD transformations are considered equivalent, and thus the two edge labels can be placed interchangeably in the two DAGs. For example, the transformations map, $flatMap$, $mapPartitions$, and $mapPartitionsWithIndex$ are equivalent because inherently they perform the same core task, i.e., they process each element in a given dataset using a function $func$, and returns a new distributed dataset.

7.2 Experimental Setup and Procedure

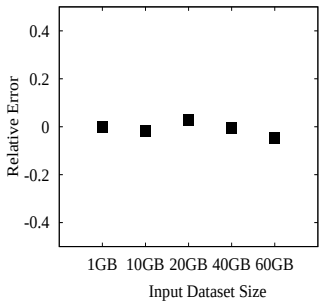
The experimental setup consists of Apache Spark version 2.3.1, built-in within the Cloudera CDH 6.0.1 package, on a cluster of m4.xlarge Amazon EC2 machine instances, each comprising 4 cores, 16 GB of RAM, and 10 GB EBS, and running RedHat Enterprise Linux version 6. We use HDFS as the backend for storing and processing the input dataset in the local persistent storage of the Ec2 instances. We use the Interior point algorithm [23] from the Optimization toolbox of the Matlab version 2013b for solving the given non-linear convex optimization problem (Section 6), and



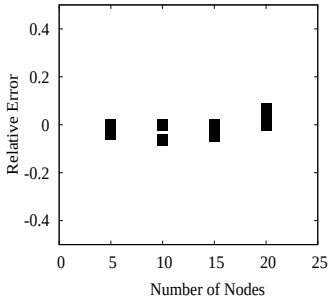
(a) Accuracy of estimation for ALS with varying input dataSet size in stand-alone mode with 20 worker nodes



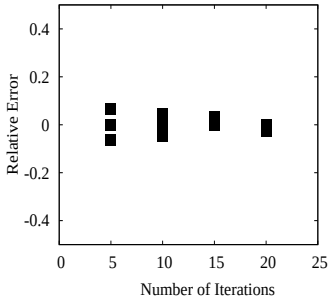
(b) Accuracy of estimation for PageRank with varying input dataset size in stand-alone mode 20 worker nodes and 5 iterations



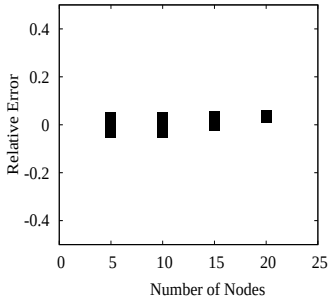
(c) Accuracy of estimation for ChiSqTest with varying input dataset size in stand-alone mode with 20 worker nodes



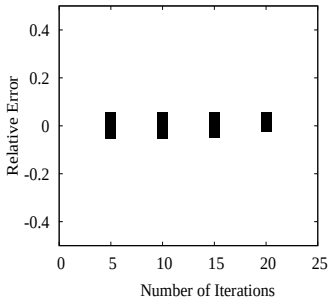
(d) Accuracy of estimation for PageRank with varying number of nodes in stand-alone mode



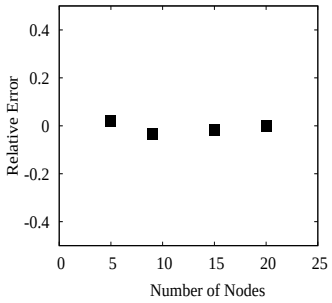
(e) Accuracy of estimation for PageRank with varying number of iterations in stand-alone mode



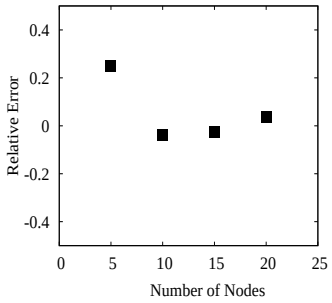
(f) Accuracy of estimation for DeepLearning with varying number of nodes in stand-alone mode



(g) Accuracy of estimation for DeepLearning with varying number of iterations in stand-alone mode



(h) Accuracy of estimation for ChiSqTest with varying number of nodes in stand-alone mode



(i) Accuracy of estimation for ChiSqTest with varying number of nodes in YARN mode

Fig. 2: Accuracy Of estimations against varying input dataset size, number of nodes and iterations

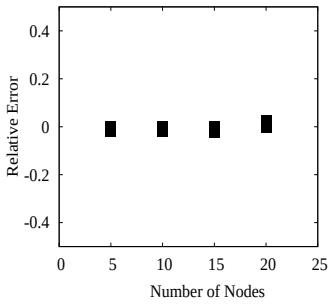
TABLE 3: Relative Error of Estimation of Job Execution Times With the Version of the SparkRDDAnalyze Tool

SparkRDDAnalyze Version	Standalone				YARN			
	Data Mining	Statistical Evaluation	Quantitative Analysis	Miscellaneous	Data Mining	Statistical Evaluation	Quantitative Analysis	Miscellaneous
V1	0.7	0.5	0.5	0.65	0.6	0.55	0.45	0.65
V2	0.3	0.2	0.1	0.25	0.2	0.15	0.2	0.3

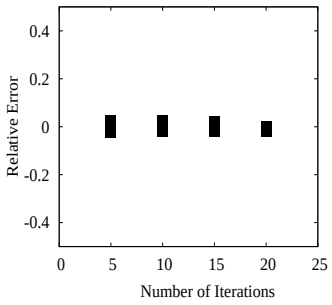
estimating the cost-optimal cluster composition. We use the 10-M MovieLens dataset obtained from grouplens.org [30] as the input workload for the MovieLensALS job. PageRank is evaluated with the social network dataset collected from LiveJournal [31], an online community comprising roughly 10 million members. The LiveJournal dataset has 48,47571 nodes and 6,8,993773 edges. The input workload for the ChiSqTest job is the dataset "Popular Kids" available in the Statlib Data and Story Library (DASL) [24].

7.3 Evaluation of the Job Categorization Scheme

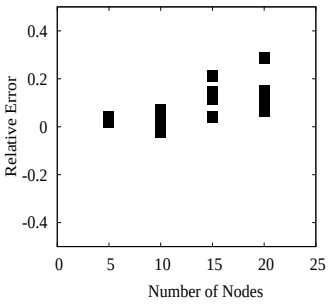
We run the SparkRDDAnalyze tool over a sample of 30 Spark jobs packaged with the Spark v2.2.1 source code to generate different equivalent classes, which correspond to different job categories. Following [3], we obtain job profile for each category by running a representative job for each category. We estimate the components of the job profile from the snapshots of the execution flow of the representative job obtained using YourKit Java Profiler [32]. Table 3 illustrates the accuracy of estimation of job execution times using the two versions of the tool SparkRDDAn-



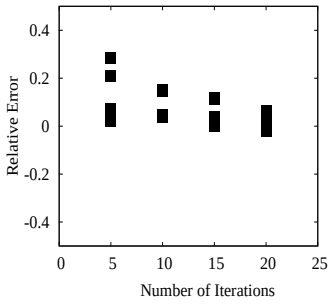
(a) Accuracy of estimation for PageRank with varying number of nodes in YARN mode



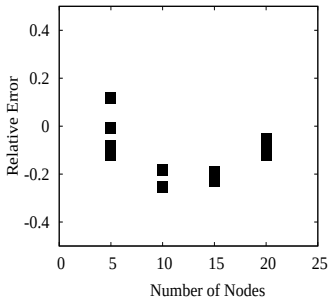
(b) Accuracy of estimation for PageRank with varying number of iterations in YARN mode



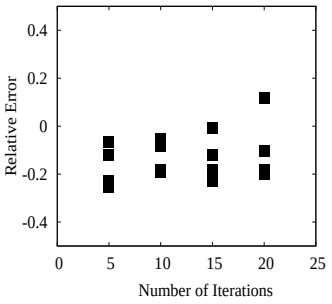
(c) Accuracy of estimation for DeepLearning with varying number of nodes in YARN mode



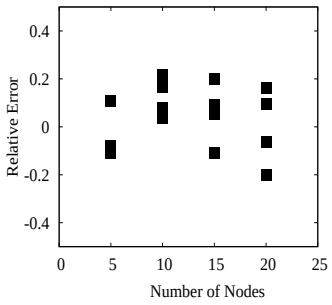
(d) Accuracy of estimation for DeepLearning with varying number of iterations in YARN mode



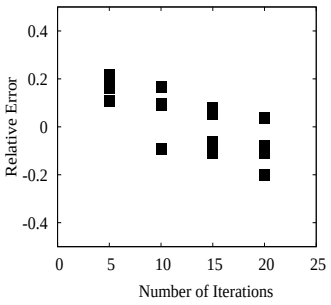
(e) Accuracy of estimation for ALS with varying number of nodes in stand-alone mode



(f) Accuracy of estimation for ALS with varying number of iterations in stand-alone mode



(g) Accuracy of estimation for ALS with varying number of nodes in YARN mode



(h) Accuracy of estimation for ALS with varying number of iterations in YARN mode

Job	Stand-alone	YARN
PageRank	0.019	0.012
ChiSqTest	0.036	0.087
BisectingKMeans	0.020	0.086
MovieLensALS	0.094	0.126

Mean	0.06
Standard Deviation	0.04
Variance	0.0016
Range	0.012 - 0.13
Majority Bound	0.02
Confidence Interval	0.056-0.062

(i) Mean Relative Error of OptEx estimations

Fig. 3: More accuracy results and the observed Mean Relative Error statistics

alyze, namely SparkRDDAnalyze-unlabelled (referred to as V1) and SparkRDDAnalyze-labelled (referred to as V2), respectively. From the estimated completion time T_{Est} and the recorded (i.e., observed) completion time T_{Rec}, we compute the relative error of estimation $RE = (T_{Est} - T_{Rec})/T_{Rec}$ with a particular example jobs chosen as the representative job for that job category. We estimate job completion times of 10 target jobs for each job category with representative jobs chosen according to a scheme discussed later in this section. We record the relative error RE for each target job, and finally compute the mean relative error for all target jobs in a particular job category. Subsequently, we express the accuracy in terms of mean relative error for each respective job category in Table 3. Mean relative error measures overall how close the job completion time estimated with the tool is to the observed job completion time.

Due to the absence of a non-trivial labelling scheme in V1 (i.e, SparkRDDAnalyze-unlabelled), the tool considers edges in a pair of DAGs as dissimilar if and only if their labels are distinct from a lexicographical sense, although essentially those edges might perform an identical set of tasks. According to this scheme, each job in the sample has an unique RDD DAG, and there are as many job categories as there are jobs in the sample. Thus V1 groups Spark jobs into 30 job categories since we have considered a sample size of 30 for the purpose of our experiments. Hence a given Spark job is rarely similar to a given representative job in any job category with SparkRDDAnalyze-unlabelled, which results in considerably high relative error in estimating the execution time of a given job using the job profile of a chosen representative job. On the other hand, SparkRDDAnalyze-labelled produces much more accurate categorization on the same group of DAGs by relabelling the edge labels appropriately as discussed in Section 7.1. V2 (i.e., SparkRDDAnalyze-labelled) produces 4 job categories by relabelling the edge labels in the RDD DAGs according to the scheme described in Section 7.1. Since the number of equivalent

classes is smaller with V2, the chances are higher for a target job being correctly estimated with job profile of a representative job chosen from a particular job category. Due to the above reason, in Table 3 the relative errors with SparkRDDAnalyze-unlabelled are significantly higher than relative errors with SparkRDDAnalyze-labelled.

Specifically, V2 of the tool generates the following equivalence classes: 1) Data Mining, comprising jobs that perform typical data mining tasks like recommendation engines or jobs that run mining queries on logs, 2) Statistical Evaluation, comprising jobs that perform typical statistical hypothesis tests on given input datasets based on different statistical evaluation techniques and standard test metrics, such as Chi Square test, etc., 3) Quantitative Analysis, comprising jobs like pagerank and logistic regression, which perform typical quantitative analysis tasks on a given dataset, and 4) Miscellaneous, comprising any other jobs that can not be grouped into any of the above categories like word count. For the job category Statistical Evaluation we use the job ChiSqTest from the group of example applications packaged with the spark source code. The representative job for the Data Mining group of jobs is the movie rating job MovieLensALS [14]. The Deep Learning Pipelining framework [33] provides a natural construct for applying deep learning models to perform distributed learning on a large-scale dataset. As a representive of heavily compute intensive MLlib applications, we ran an application based on the Deep Learning Pipelining on the MNIST dataset of hand-written digits from yann.lecun.com. The application first trains a InceptionV3 convolutional neural network, and then applies the DeepImageFeaturizer algorithm to filter off the last layer of the InceptionV3 network. The output from the remainning layers of the network as features for trainning a logistic regression model with elastic net regularization. PageRank is the representative job for the Quantitative Analysis group of jobs [14]. We could have chosen any other job from the group of sample jobs associated with the corresponding equivalent class as the representative job for the respective job category without any effect in the mean relative error. In Table 3 the relative error with SparkRDDAnalyze-labelled varies from 0.1 to 0.3, with a mean relative error of 0.21 and a standard deviation of 0.18. The difference in relative error with the two versions V1 and V2 ranges from 0.25 with Quantitative Analysis jobs to the extent of 0.4 with Data Mining and Statistical Evaluation jobs. In the case of V2, mean relative error of 0.21 demonstrates that the estimation with V2 is fairly accurate. The fact that the relative error is smaller with version V2 of the tool than with version V1 signifies that the profile parameters estimated using the representative job chosen with V2 produces a more accurate estimate of the job completion time than the estimates obtained with V1.

7.4 Accuracy of the Estimations Using OptEx

Being the first work in modelling Spark jobs, OptEx has no prior baseline to compare with. However, we demonstrate (see Figures 2 and 3) that OptEx provides accurate (i.e., average relative error 0.06) estimations of the job completion time against variations in all the input parameters of the model (i.e, against increasing size of dataset, number of nodes, and the number of iterations), and on jobs of different categories. The Figures 2(a), 2(b), and 2(c) illustrate the variations in the relative error RE, for the MovieLensALS, PageRank, and ChiSqTest jobs, with increasing size s of the input dataset. The figures 2(d), 2(f), 2(h), and 3(e) illustrate the variations in the relative error RE, for PageRank, DeepLearning, ChiSqTest, and MovieLensALS, against varying cluster size n, in the stand-alone mode. The figures 2(e), 2(g), and 3(f) illustrate the variations in the relative error RE for the same jobs, i.e., PageRank, DeepLearning, and MovieLensALS, respectively, against varying number of iterations $\mathcal{I}$, in the stand-alone mode. Figures 2(i), 3(a), 3(c), and 3(g) illustrate the variations in the relative error RE for ChiSqTest, PageRank, DeepLearning, and MovieLensALS with varying n in the YARN mode. Figures 3(b), 3(d), and 3(h) illustrate the variations in the relative error RE for PageRank, DeepLearning, and MovieLensALS against varying $\mathcal{I}$, in YARN mode. We evaluate the OptEx model with the mean relative error metric [23] $\delta = \frac{\sum_{j=1}^{k}|T_{Est}-T_{Rec}|/T_{Rec}}{k}$, where k is the total number of jobs submitted. The absolute differences between T_{Est} and T_{Rec} eliminate the signs in the error, and gives the magnitudes of the errors. The error values δ are given in the Table 3(i). The average δ score for all the cases is 0.06, i.e., 6%, which is comparable to prior work on performance modelling of state-of-the-art data anytics engines [3].

7.5 Analysis of the Results

The magnitude of the relative error RE for the experiments with ChiSqTest, PageRank, and DeepLearning jobs (Figure 2 is strictly within 0-0.06, bounded by a 95% confidence interval of 0.056-0.062 (refer to Table 3(i)) for the Statistical Evaluation and Data Mining job categories (refer to Section 3), in stand-alone mode. The only outlier is the ChiSqTest run in YARN mode in Figure 2(i) (reasons given later in this section). The experiments with increasing dataset size yield relative error of magnitude between 0.007 to 0.05 (Figures 2(a), 2(b), 2(c)).

The estimated execution time T_{Est} of a Spark job comprises two components X_1 and X_2, where $X_1 = T_{Init} + T_{prep}$ and $X_2 = n \times \mathcal{I} \times C + \mathcal{I} \times B/n + \frac{A \times s}{n}$ (refer to Equation 8). The first component X_1 is independent of variations in the values of the input variables. The second component X_2 comprises the last three phases of the Spark job execution (refer to Section 5), each phase varying differently with respect to the input variables n, $\mathcal{I}$, and s (refer to Section 7.7). Hence, X_2 accounts for the observed random variations in the relative error, with respect to variations in the input variables (Figures 2 and 3). The execution phase in X_2 encompass the execution of the job stages, comprising unit RDD operations, on the worker nodes (Section 5). The execution of the job stages on the workers is inherently non-deterministic (unpredictable) in nature, due to the dependency on various components of the Spark cluster, like the driver, the cluster manager, the workers, etc., and their configuration [8]. The job stages may get unpredictably delayed, i.e., stages can fail and be retried by the master repeatedly, due to various factors like momentary unavailability of required resources, delays in allocation of resources by the master, communication delays among the workers, etc., [8]. The above unpredictable delays in the job stages, however small, can cause the observed values of X_2 to deviate randomly from the estimated values of X_2, while X_1 stays constant (Section 5). This, in turn, causes the overall observed completion time T_{Rec} of the job, to vary unpredictably with respect to the estimated job completion time T_{Est}, estimated from X_1 and X_2 (Section 5). This causes the observed random variations in the values of the relative error (i.e., $RE = T_{Est} - T_{Rec}$), though still bounded by the confidence interval of 0.056-0.062 (Figures 2 and 3). The relative error increases slightly

TABLE 4: Optimal Scheduling With Estimated Optimal Cluster Size Under Varying SLA

SLA(sec)	Mode	App	Iterations											
			5			10			15			20		
			n	T_{Est}(sec)	T_{Rec}(sec)	n	T_{Est}(sec)	T_{Rec}(sec)	n	T_{Est}(sec)	T_{Rec}(sec)	n	T_{Est}(sec)	T_{Rec}(sec)
300	Standalone	ALS	3	203.2	207.0	5	188.0	193.0	6	189.7	92.0	7	188.8	195.0
340	YARN	ALS	2	240.0	430.0	3	328.4	335.0	4	311.5	315.0	4	331.0	337.0
380	Standalone	ChiSqTest	1	332.8	345.0									
800	YARN	ChiSqTest	1	794.8	821.0									
350	Standalone	ALS	3	200.0	297.0	4	299..0	297.0	4	340.5	343.0	6	303.2	113.0
400	YARN	ALS	3	174.2	315.0	4	381.0	385.0	5	378.6	386.0	5	398.0	395.0
430	Standalone	ChiSqTest	2	421.7	425.0									
790	YARN	ChiSqTest	2	783.7	781.0									
200	Standalone	ALS	4	79.5	275.0	6	176.1	178.0	6	189.7	191.0	7	188.8	194.0
260	YARN	ALS	4	250.5	257.0	5	259.0	258.0	6	260.0	255.0	7	2159.8	258.0
425	Standalone	ChiSqTest	2	421.7	423.0									
785	YARN	ChiSqTest	2	783.7	782.0									
175	Standalone	ALS	5	168.5	165.0	7	175.0	167.0	8	171.9	172.0	9	173.1	171.0
240	YARN	ALS	5	239.5	238.0	7	239.9	235.0	9	238.1	233.0	9	239.1	235.0
420	Standalone	ChiSqTest	3	319.6	315.0									
783	YARN	ChiSqTest	3	781.6	754.0									

with increasing number of nodes (Figures 2(a), 2(d), 2(f), 2(h), 2(i), 3(a), 3(c), 3(e), and 3(g)). Increase in number of worker nodes augments the chances of unpredictable failures of the job stages due to dependency on communication between a larger number of nodes, causing unpredictable variations in the component X_2 of the overall job completion time T_{Est}. This, in turn, causes the observed job completion time T_{Rec} to deviate more unpredictably from the estimated completion time T_{Est}, estimated from X_1 and X_2. The result is greater variation in the relative error (i.e., RE = $T_{Est} - T_{Rec}$) with increasing number of nodes. Our goal is to provide correct estimations for SLA-driven user-facing jobs. Few user-facing jobs, that work under an SLA deadline, will require more than 50 nodes [34, 35]. OptEx can provide estimations for such typical SLA-driven user-facing jobs with a relative error close to 0 (Figures 2 and 3). Jobs that do not meet this criteria are batch processing jobs, like bioinformatics, genomics, data analytics jobs, etc., which typically do not work under a deadline [36].

For experiments run in YARN mode (Figures 3(a), 3(b), 3(c), 3(d), 3(g), and 3(h)), the variations in the observed relative error, with respect to the variations in the input variables, are noticeably larger than the experiments run in stand-alone mode. In YARN mode, the submitted jobs are additionally dependent on the YARN resource manager to allocate resources, and to execute the jobs on the workers [37]. Hence, the chances of unpredictable delays in the intermediate stages of a job are greater in YARN mode due to unreliability of the communication channel between the YARN resource manager and the Spark master [37]. Thus, the chances of observing randomness in the relative error is greater for jobs run in the YARN mode, though the magnitude of the average relative error does not exceed 0.04. Further, the relative error, for YARN mode, is even closer to 0 for jobs with number of iterations larger than 10 (Figures 3(a), 3(c), and 3(g)), representing real world use cases reported by various enterprises and implemented in several widely used web services [34, 35]. OptEx cannot account for the non-deterministic delays in communicating the intermediate RDD objects among the worker nodes during the execution of an iterative Spark job on the workers [8]. The above delays result in deviations in the observed length of the later phases of job execution, comprising the component X_2, from the estimated completion time [8]. For experiments with large number of iterations, the job stages in the initial iterations cache the intermediate RDD objects locally in the worker nodes, resulting in a decrease in the time spent in communicating the RDD objects among the workers during the later iterations [8]. This results in a decrease in the deviation in the observed length of the job phases comprising X_2 from their estimated lengths. This, in turn, reduces the deviations in the overall observed completion time T_{Rec} with respect to the estimated overall completion time T_{Est}, estimated from X_1 and X_2. Indeed, with increasing number of iterations, a decreasing trend is observed in the relative error (Figures 2 and 3). Hence we believe that OptEx can provide more accurate estimations with typical production level use cases, which typically involve number of iterations larger than 10 [34, 35].

Interference: Figure 1 in [38] shows that performance in a tightly controlled cloud environment shows a non deterministic fluctuation over time due to varying degrees of interference among VMs co-located in same physical machine with fixed workload and resource allocation. Hence, [38] uses a black box approach for detecting ineterference on the basis of extensive performance statistics collected by actual execution of a given target application. According to [39, 40], the combined performance under inteference with multiple co-located VMs that share the same computing, I/O, and network resources varies with workload type and proportion of workload sharing among the VMs. Further, according to [41], the degree of interference in container-based clouds further varies with the level of mixing of different memory-centric and cpu-centric workloads, which is difficult to quantify. For the sake of keeping the model simpler, we do not explictly consider the above factors that causes interference in a virtualized platform like the public cloud. Despite that, our model generates considerably high accuracy of estimation and relatively unbiased variation in the relative error as shown in Figures 2 and 3. Since all the experiments were performed on an actual public cloud, our results prove that OptEx works reasonably well under varying degrees of interference in a real cloud. Moreover, Spark applications typically are relatively compute-intensive by nature, and require larger VMs. The degree of interference in larger VMs is lesser due to lower chances of them sharing same physical resources with other co-located VMs. Public cloud providers like Amazon also provide options to choose EBS optimized instances which minimize interference and provide performance guarantees. Spark clusters can also be created within a common VPC in the cloud to minimise the interference due to network traffic. Hence in reality interference accounts for only a very small portion of the error in OptEx.

7.6 Optimal Resource Provisioning

The work closest to OptEx is Elastisizer [2], which predicts optimal cluster composition for Hadoop, but does not address Spark. Moreover, Elastisizer over predicts, on an average by 20% and in the worst case by 59% [2]. Since OptEx uses a closed-form to estimate the completion time, it does not suffer from over-prediction. Table 4 demonstrates the effectiveness of the constrained optimization technique of OptEx (Section 6) in designing optimal scheduling strategies. For each job (refer to the column labelled App in Table 4) running in Standalone or YARN mode (refer to the column labelled Mode), Table 4 gives the cost-optimal cluster composition for executing a given job under given SLA deadlines (refer to the SLA column), while minimizing the cost of usage of the virtual machine instances. The optimal cluster size n for 5, 10, 15, and 20 iterations are given in separate group of columns each labelled n. The group of columns labelled T_{Est} in Table 4 gives the completion times T_{Est} estimated using OptEx, for 5, 10, 15, and 20 iterations, respectively. The other group of columns labelled T_{Rec} gives the recorded completion times T_{Rec} with the estimated cluster composition. Following [3], we propose a statistic $\mathcal{S}$ to measure

TABLE 5: Optimal Resource Provisioning With Estimated Optimal Cluster Size Under Given Cost Budget

Budget($)	Mode	App	n	T_{Est} (sec)	T_{Rec} (sec)
0.30	Standalone	ALS	53	49.2	48.0
0.80	YARN	ALS	58	120.2	115.0
1.00	Standalone	ChiSqTest	27	318.0	321.0
1.50	YARN	ChiSqTest	16	780.1	775.0
0.20	Standalone	ALS	35	49.4	50.0
0.50	YARN	ALS	36	120.0	119.0
0.80	Standalone	ChiSqTest	22	318.0	321.0
1.20	YARN	ChiSqTest	13	780.1	780.0
0.20	Standalone	ALS	26	49.7	50.0
0.40	YARN	ALS	29	120.6	117.0
0.60	Standalone	ChiSqTest	16	318.1	311.0
1.00	YARN	ChiSqTest	11	780.1	757.0
0.10	Standalone	ALS	17	50.7	52.0
0.30	YARN	ALS	21	121.1	125.0
0.40	Standalone	ChiSqTest	11	318.1	315.0
0.80	YARN	ChiSqTest	8	780.2	780.0
0.10	Standalone	ALS	13	51.9	50.0
0.20	YARN	ALS	14	122.5	120.0

the effectiveness of OptEx in estimating whether a given job will satisfy the SLA deadline, while minimizing the cost. $\mathcal{S}$ gives the percentage of cases which did not violate the SLA deadline, in the experiments recorded in the Table 4. $\mathcal{S}$ evaluates approximately to 98% on cluster compositions comprising representative virtual machine instances, using job profiles generated by running example Spark applications on benchmark datasets (Section 7). Our evaluation follow closely with benchmark results resported by prior researchers [42]. Table 5 demonstrates that OptEx can be used effectively in project planning, i.e., for optimal cluster provisioning under given budget, while minimizing job execution times. Table 5 records the optimal cluster size (refer to the 4th column) required to run a given job (refer to the 3rd column) in Standalone or YARN mode (the 2nd column) estimated using Equation 8 under different values of the cost budget (refer to the 1st column), while optimizing the completion times. The 5th column of Table 5 gives the completion times T_{Est} estimated using OptEx, and the last column gives the recorded completion time T_{Rec} with the estimated cluster composition.

7.7 Error in Estimation of Execution Times

The OptEx model expresses the execution time of a given Spark job as a function of several input parameters, like number of nodes, size of input dataset, and number of iterations. To ensure that the model is effective in estimating execution time of a given job, the model must accurately reflect the variation of job completion time with the input parameters. Here we demonstrate that the OptEx model exhibits a high degree of accuracy in the estimation of completion times of a given job which processes a given input dataset on a given cluster composition. First, we measure the model's error (which indirectly reflects models's accuracy) in terms of the difference between the standard deviations of the estimated and recorded (observed) job execution times, respectively. The error is computed as follows.

$$error = |\sigma_{T_{Est}} - \sigma_{T_{Rec}}|. \tag{10}$$

According to Equation 3 the total execution time is estimated, i.e., T_{Est} is computed, using the equation $T_{Est} = T_{Init} + T_{prep} + T_{vs} + T_{comp}$. We compute the standard deviation of T_{Est} as follows.

$$\begin{aligned}\sigma^2_{T_{Est}} = \sigma^2_{T_{Init}} + \sigma^2_{T_{prep}} + \sigma^2_{T_{vs}} + \\ \sigma^2_r T_{comp} + 2 \times \mathrm{Cov}(T_{Init}, T_{prep}) + 2 \times \mathrm{Cov}(T_{prep}, T_{vs}) + \\ 2 \times \mathrm{Cov}(T_{vs}, T_{Init}) + 2 \times \mathrm{Cov}(T_{comp}, T_{Init}) + \\ 2 \times \mathrm{Cov}(T_{comp}, T_{prep}) + 2 \times \mathrm{Cov}(T_{comp}, T_{vs}).\end{aligned} \tag{11}$$

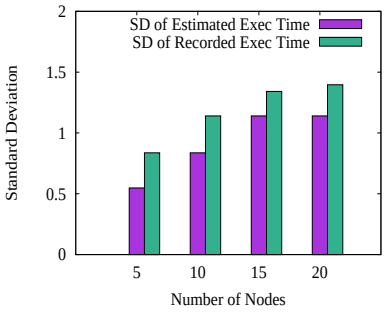
The parameters T_{Init} and T_{prep} are independent of the input variables n, s, and $\mathcal{I}$. Hence the covariances evaluate to 0. Using Equations 12 and 8 in 10 we get the following equation.

$$\begin{aligned} error &= |\sqrt{\sigma^2_{T_{Est}}} - \sqrt{\sigma^2_{T_{Rec}}}| \\ &= |\sqrt{\sigma^2_{T_{Init}+T_{prep}+n\times\mathcal{I}\times C+\mathcal{I}\times B/n+\frac{A\times s}{n}}} \\ &\quad -\sigma_{T_{Rec}}| \\ &= |C^2 * \{\sigma^2_{\mathcal{I}} * \sigma^2_n + \sigma_{\mathcal{I}} * \mathrm{E}^2(n) + \\ &\quad \mathrm{E}^2(\mathcal{I}) * \sigma^2_n\} + B^2 * \{\sigma^2_{\mathcal{I}} * \sigma^2_{1/n} + \\ &\quad \sigma^2_{\mathcal{I}} * \mathrm{E}^2(1/n) + \mathrm{E}^2(s) * \sigma^2_{1/n}\} + \\ &\quad A * \{\sigma^2_s * \sigma^2_{1/n} + \sigma^2_s * \mathrm{E}^2(1/n) + \\ &\quad \mathrm{E}^2(s) * \sigma^2_{1/n}\}|^{1/2} - \sigma_{T_{Rec}}. \end{aligned} \tag{12}$$

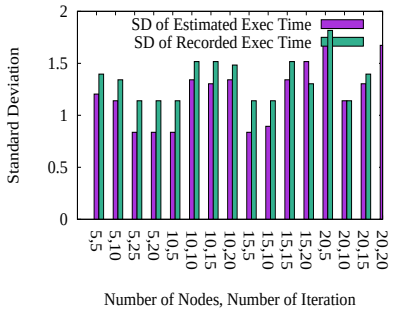
Figures 4(a), 4(b), and 4(c) illustrate standard deviation of estimated job completion times and recorded execution times, for PageRank, ChiSqTest, and MovieLensALS applications, respectively, computed as $\sigma_{T_{Est}}$ and $\sigma_{T_{Rec}}$, respectively. These figures demonstrate that the distribution of estimated job completion time computed from the OptEx model closely echoes the distribution of recorded job completion time, i.e., the OptEx estimation follows the variation pattern of the recorded job completion times. The $error$ between estimated standard deviation for recorded and estimated job completion times, computed using Equation 10, falls within 0.5, i.e., $error \leq \epsilon$ where the error threshold $\epsilon = 0.5$ seconds. This demonstrates the effectiveness of the OptEx model under the error bound of 0.5 seconds.

8 Related Work

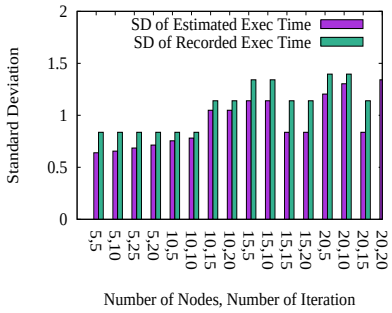
Verma et al. [3] present the design of ARIA, a framework for optimal resource allocation for Hadoop MapReduce. Herodotou



(a) Standard Deviation for ChiSqTest



(b) Standard Deviation for PageRank



(c) Standard Deviation for ALS

et al. [2] present Elastisizer, a system that predicts the optimum cluster configuration for running a given Hadoop MapReduce job, using a model built by exploring a search space consisting of job profiles. Though Spark [8] is fast surpassing Hadoop in popularity and usage, there has not been much work in modelling Spark jobs yet [43, 4, 5]. [44] presents an approach to determine equivalence of Spark programs by converting the user defined functions comprised in the programs to equivalent first order formulaes. But unlike OptEx, the approach described in this paper is restricted to programs that use only map, filter, cartesian product, fold, and foldByKey methods from the Apache Spark library.

Huang et al. [42] presents DtCraft, a high performance system for compute intensive applications using parallel programming model. On the other hand, we are interested in data-intensive comnputing which typically deal with I/O bound data-centric workloads. DtCraft uses abstractions based on StreamGraph which is similar to the lineage graph in Spark. Hence theoretically OptEx can be extended to model DtCraft applications as well but this is out of the scope of this paper. Nguyen et al. [45] presents Galois, a system that provides radical performance improvement while running graph analytics tasks using Domain Specific Languages (DSLs). However, OptEx is designed for cost optimal cluster provisioning for a wider range of tasks, not just graph based DSLs. Ehsan et al. [46] presents AffordHadoop, a system built to process data-intensive computations which is designed specifically for scheduling map-reduce jobs. It deals with optimal resource scheduling for map-reduce jobs already running on a cloud whereas OptEx deals with cloud provisioning, i.e., estimating the composition of a cluster to run a given Spark job. Ousterhout et al. [47] aims at understanding performance bottlenecks in Spark job execution, and the role of various factors like network, disk I/O, stragglers on the execution time. While the findings of the paper are indeed insightful, and beneficial to understanding job execution, they target only the bottlenecks instead of the overall job execution. Since the above technique rely on fine-grained instrumentation specifically based on a small sample of Spark jobs, it can not be applied to wider group of Spark jobs. Taking into account the bottlenecks due to disk I/O, network state, and stragglers in the OptEx model would increase the complexity of the model, which, in turn, complicates the cluster provisioning problem. Considering the above non-deterministic factors would require accurate instrumentation of the jobs against various network and I/O conditions. In that case, the accuracy of the OptEx model would be dependent on a comprehensive profiling technique where all possible network and I/O conditions are considered. However, this is out of the scope of this paper, and we already achieve considerably high accuracy with OptEx even under the simplifying assumptions. Chen et al. [48] presents a performance model of internet services which is based on application level parameters like login rate and number of active connections. They provide power and CPU utilization model for traditional web-based applications and database servers based on login rate and number of active connections. They donot consider large-scale bigdata applications based on modern distributed computing paradigms. Kansal et al. [49] presents a model for power metering and provisioning of virtual machines that can assist in management of power and reduce wastage of power capacity in the virtual machines comprised in the datacenters. Chaisiri et al. [50] provides a technique to reduce custer resource cost by leveraging reservation plans instead of on-demand provisioning plans. Here, the execution time on each virtual machine class is known beforehand, whereas with OptEx the execution time of the target job is unknown. Urgaonkar e al. [51] provides a dynamic resource provisioning model based on the request-response model which is developed from queuing theory. But this model is dependent on the parameters of the queuing model such as arrival rate and request rate which vary with time, hence it is not applicabe in our case. Islam et al. [52] assumes a black box approach that leverages machine learning techniques whereas OptEx provides an empirical model of job completion times.

9 Conclusions

OptEx models Spark job execution using analytical techniques. OptEx provides a mean relative error of 6% in estimating job completion time. OptEx yields a success rate of 98% in completing Spark jobs under a given SLA deadline with cost-optimal cluster compositions. OptEx can be used to estimate whether a given job will finish under a given deadline with the given resources on the cloud. It can be used to devise optimal scheduling strategy for Spark.

Wojciech Golab is partially funded by Natural Sciences and Engineering Research Council (NSERC) of Canada.

References

[1] A. Inc, *Amazon Elastic Compute Cloud (Amazon EC2)*. http://aws.amazon.com/ec2/#pricing: Amazon Inc.
[2] H. Herodotou, F. Dong, and S. Babu, "No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics," in *SOCC '11*.
[3] A. Verma, L. Cherkasova, and R. H. Campbell, "Aria: Automatic resource inference and allocation for mapreduce environments," in *ICAC '11*.
[4] W. Zhang, S. Rajasekaran, T. Wood, and M. Zhu, "Mimp: Deadline and interference aware scheduling of hadoop virtual machines," *CCGrid '14*.
[5] K. Kc and K. Anyanwu, "Scheduling hadoop jobs to meet deadlines," ser. CLOUDCOM '10.

[6] S. Imai, T. Chestna, and C. A. Varela, "Accurate resource prediction for hybrid IAAS clouds using workload-tailored elastic compute units," ser. UCC '13.
[7] G. Singer, I. Livenson, M. Dumas, S. N. Srirama, and U. Norbisrath, "Towards a model for cloud computing cost estimation with reserved instances," *CloudComp 2010*.
[8] 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 *NSDI'12*.
[9] J. G. Shanahan and L. Dai, "Large scale distributed data science using apache spark," in *KDD '15*.
[10] S. Salloum, R. Dautov, X. Chen, P. X. Peng, and J. Z. Huang, "Big data analytics on apache spark," *International Journal of Data Science and Analytics*, 2016.
[11] B. Kandregula, "Real-time weather event processing with hdf, spark streaming, and solr," 2016.
[12] EVAM, "Evam streaming analytics," 2016.
[13] S. Sidhanta, S. Mukhopadhyay, and W. Golab, "OptEx: Deadline-Aware Cost Optimization for Spark," Tech. Rep., 01 2018.
[14] A. S. Foundation, "Apache spark libraries," 2015.
[15] R. Milner, "An algebraic definition of simulation between programs," in *IJCAI'71*.
[16] D. Park, "Concurrency and automata on infinite sequences," in *5th GI-Conference*.
[17] R. Milner, *Communicating and Mobile Systems: The &Pgr;-calculus*. New York, NY, USA: Cambridge University Press, 1999.
[18] D. Latella, M. Massink, and E. P. de Vink, "Bisimulation of labelled state-to-function transition systems coalgebraically," *Logical Methods in Computer Science*, vol. 11, no. 4, 2015.
[19] I. Castellani, "Bisimulations and abstraction homomorphisms," *J. Comput. Syst. Sci.*, vol. 34, no. 2/3, pp. 210–235, 1987.
[20] S. Abriola, P. Barceió, D. Figueira, and S. Figueira, "Bisimulations on data graphs," in *KR'16*.
[21] R. Mayr, "On the complexity of bisimulation problems for basic parallel processes," in *Automata, Languages and Programming*. Springer Berlin Heidelberg, 2000.
[22] F. Zuleger, S. Gulwani, M. Sinn, and H. Veith, "Bound analysis of imperative programs with the size-change abstraction," in *SAS'11*.
[23] J. Leader, *Numerical Analysis and Scientific Computation*. Pearson Addison Wesley, 2004.
[24] M. Chase and G. Dummer, "Statlib data and story library (dasl)," http://www.stat.yale.edu/Courses/1997-98/101/chisq.htm, Jun. 1996.
[25] S. Sidhanta, "Optex job claassifier and model generator," 2018. [Online]. Available: https://github.com/ssidhanta/SparkRDDAnalyze
[26] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "CADP 2011: a toolbox for the construction and analysis of distributed processes," *STTT*, vol. 15, no. 2, pp. 89–107, 2013.
[27] D. Bergamini, N. Descoubes, C. Joubert, and R. Mateescu, "Bisimulator: A modular tool for on-the-fly equivalence checking," in *TACAS'05*.
[28] X. Yang, J. Katoen, H. Lin, and H. Wu, "Proving linearizability via branching bisimulation," *CoRR*, vol. abs/1609.07546, 2016.
[29] A. S. Foundation, "Spark programming guide," 2017.
[30] *MovieLens dataset, http://www.grouplens.org/data/*.
[31] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.
[32] "The YourKit Java Profiler," http://www.yourkit.com, last 2008.
[33] Databricks, "Deep learning pipelines for apache spark," 2017.
[34] A. S. Foundation, "Powered by spark," 2015.
[35] Cloudera, "Category archives: Use case," 2015.
[36] S. Pappas, "9 super-cool uses for supercomputers," 2015.
[37] Apache Software Foundation. (2014) Apache hadoop nextgen mapreduce (yarn).
[38] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini, "Deepdive: Transparently identifying and managing performance interference in virtualized environments," in *USENIXATC'13*.
[39] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, and C. Pu, "Understanding performance interference of i/o workload in virtualized cloud environments," in *2010 IEEE 3rd International Conference on Cloud Computing*.
[40] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, C. Pu, and Y. Cao, "Who is your neighbor: Net i/o performance interference in virtualized clouds," *IEEE Transactions on Services Computing*, vol. 6, no. 3, pp. 314–329, 2013.
[41] M. G. Xavier, I. C. D. Oliveira, F. D. Rossi, R. D. D. Passos, K. J. Matteussi, and C. A. F. D. Rose, "A performance isolation analysis of disk-intensive workloads on container-based clouds," in *PDP'15*.
[42] T. Huang, C. Lin, and M. D. F. Wong, "Dtcraft: A distributed execution engine for compute-intensive applications," in *ICCAD'17*.
[43] N. B. Rizvandi, J. Taheri, R. Moraveji, and A. Y. Zomaya, "On modelling and prediction of total cpu usage for applications in mapreduce environments," in *ICA3PP'12*.
[44] S. Grossman, S. Cohen, S. Itzhaky, N. Rinetzky, and M. Sagiv, "Verifying equivalence of spark programs," in *CAV '17*.
[45] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *SOSP*. ACM, 2013, pp. 456–471.
[46] M. Ehsan, K. Chandrasekaran, Y. Chen, and R. Sion, "Cost-efficient tasks and data co-scheduling with affordhadoop," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2018.
[47] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *NSDI'15*.
[48] G. Chen, W. He, J. Liu, S. Nath, L. Rigas, L. Xiao, and F. Zhao, "Energy-aware server provisioning and load dispatching for connection-intensive internet services," in *NSDI'08*.
[49] A. Kansal, F. Zhao, J. Liu, N. Kothari, and A. A. Bhattacharya, "Virtual machine power metering and provisioning," in *SoCC '10*.
[50] S. Chaisiri, B.-S. Lee, and D. Niyato, "Optimization of resource provisioning cost in cloud computing," *IEEE Trans. Serv. Comput.*
[51] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal, "Dynamic provisioning of multi-tier internet applications," in *ICAC'05*.
[52] S. Islam, J. Keung, K. Lee, and A. Liu, "Empirical prediction models for adaptive resource provisioning in the cloud," *Future Gener. Comput. Syst.*
[53] H. Benaroya, S. Han, and M. Nagurka, *Probability Models in Engineering and Science*, ser. McGraw-Hill professional engineering: Mechanical engineering.
[54] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10, 2010, pp. 10–10.
[55] M. Zaharia, "Apache spark," https://github.com/apache/spark/tree/master/examples/src/main/java/org/apache/spark/examples, 2013.
[56] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska, "Mli: An api for distributed machine learning," *2013 IEEE 13th International Conference on Data Mining*, vol. 0, pp. 1187–1192, 2013.
[57] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Comput. Netw. ISDN Syst.*, vol. 30, no. 1-7, pp. 107–117, Apr. 1998.
[58] E. W. Weisstein, "Convex function. — from mathworld–a wolfram web resource." 2015.

10 Supplementary Material

10.1 Calculation for Analysis of Variations

We observe the variations in the estimated completion time T_{Est} with the number of nodes n and analyse the accuracy of OptEx with respect to the variations in the estimated job completion, compared against the variations in the observed completion times. The function T_{Est} is a discrete nonlinear function over the integer variable n, i.e., $T_{Est} = f(n)$. Also, $f(n)$ is twice differentiable in the interval 1 to N, N being the maximum number of nodes bounded by the cost budget. Let μ and σ be the sample mean and variance of the job completion times for the experiments. The standard deviation and variance of the function represents the stability of the function f with respect to variations in n. The expectation and variance is computed using Taylor expansions [53] and can be expressed as follows: $\mathrm{E}[f(n)] \approx (\mu) + \frac{f''(\mu)}{2}(\sigma)^2$, and $\mathrm{Var}[T_{Est}] \approx (f'(E[n]))^2$. We also compute the confidence interval, which acts as a tolerance bound that limits the estimated values of T_{Est} within an acceptable range. We say that as long as 95% of the estimated T_{Est} values remain within the interval $\mu - \alpha\sigma$ to $\mu + \alpha\sigma$, the estimation is acceptable with 95% confidence level.

10.2 Example Spark Applications

Applications like iterative machine learning and interactive data mining applications, that reuse intermediate results are ideal

candidates to represent jobs that are suitable for running on Apache Spark [8, 54]. We use the following example applications, belonging to the above category, as benchmarks for analyzing performance of Spark jobs.

10.2.1 WordCount

The Wordcount program [55], that counts the number of words occurring in a file or group of files, is a popular example of parallel data processing applications. We chose Wordcount as one of the example Spark applications for our analysis. It splits (i.e. partitions) the file(s) into chunks or blocks of data, in the form of key-value pairs, and distributes the blocks among the worker nodes in the cluster. Each worker thread performs the computations on an individual chunks in an independent concurrent manner, typically known as the map phase. Next the intermediate output files from individual workers are shuffled, i.e., sorted by key pairs, in the shuffle phase. Finally the shuffle outputs are grouped together by the keys, in what is typically regarded as the reduce phase.

Fig. 4: The Pseudo-code for Wordcount

```
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line =>
                 line.split(" ")).map(word => (word,
                      1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

We use the Scala implementation of the WordCount program found in the Spark example jar (see Figure 8). The WordCount program creates the SparkContext and loads the text files from HDFS. The flatMap method is used to split the text content on blank spaces, and create an RDD comprising the text fragments. The map method creates tuples $\langle k, v \rangle$ out of the RDDs, similar to key-value pairs in Hadoop MapReduce. The value v in each tuple contains a number, initialized to 1, denoting the occurrence of the word corresponds to the key k. The reduceByKey method counts the occurrence of the words by adding the values v for all tuples.

10.2.2 Movie Rating using Alternating Least Squares Technique

Recommender systems typically use collaborative filtering techniques to predict what items users might like depending on similarity of users' past behaviour, and preferences based on historic data. MovieLensALS [55] (see Figure 9) is a collaborative filtering based movie rating application, found in Spark's Machine Learning Library (MLlib) [56], a scalable library comprising API's for a host of learning algorithms. MovieLensALS performs training using ALS (i.e., Alternating Least Squares) on a dataset comprising latent factors defining a set of movies and ratings by users, and predict missing movie ratings.

The application runs an iterative machine learning algorithm, updating the features in the training dataset on each iteration, and has been used as an example in papers dealing with Spark [8, 54]. The Movie rating application is particularly suited as an example Spark application representative of the group of applications requiring reuse of immediate output data generated from iterative operations [8].

The textFile method loads the data into the RDD named ratings. ALS trains a matrix factorization model that estimates a user-item association matrix named R, as the product of two lower-rank matrices M and U. Each row in M and U represents a feature vector comprising features of a user and a movie respectively. A user u's rating of a movie m is given as the dot product of the feature vectors corresponding to u and m respectively. The ratings matrix R is a partially filled containing certain already known rating values provided by some users for some movies.

Fig. 5: The Pseudo-code for MovieLensALS

```
val R = sc.textFile(params.input).map { line =>
val fields = line.split("::")}
val Rb = spark.broadcast(R)
for (i <- 1 to ITERATIONS) {
    U = spark.parallelize(0 until u)
    .map(j => updateUser(j, Rb, M))
    .collect()
    M = spark.parallelize(0 until m)
    .map(j => updateUser(j, Rb, U))
    .collect()
}
```

ALS iteratively performs optimizations on U and M, under given M and U respectively, while minimizing the error in R. The final U and M matrices are used to predict the unknown ratings using the $U \times M$ operation. The parallelization of ALS is achieved by using the parallelize method to perform the optimization operations on different rows of U and M on different worker nodes. The ratings matrix R is broadcasted to the nodes, and the collect method is used to aggregate the results from each node into R.

10.2.3 PageRank

Another popular example for iterative computation intensive algorithm is PageRank [57], that ranks documents on the web based on the occurrence and importance of the links from the given web document to other documents. This algorithm particularly suits Spark because of the iterative nature, reuse of the intermediate results, and the scope for parallelising the computations on groups of documents. The program [55] (refer to Figure 10) iteratively updates the rank for each document by adding up contributions from documents that are linked from it.

On each iteration, each document i sends a value r_i denoting it's contribution to the overall rank to its neighbors. It (i.e., the web document) updates its own rank with the weighted sum of the rank contributions from all its neighbours. The lines.map function splits the web document into lines. The mapValues function creates a key-value pair named ranks. Then the links.join method identifies the url links in each line iteratively, and considers the contributions from the linked documents in each iteration. The reduceByKey method aggregates the contributions from the neighbors and updates the rank values by the weighted sum of the contributions. Finally the collect method computes the output ranks.

10.2.4 Logistic Regression

We choose the logistic regression program [55] (refer to Figure 11) from the MLlib library [56] as an example since it is an iterative machine learning algorithm, and used frequently as an example in works dealing with Spark and RDD. Logistic regression is used to classify a given dataset into sets of multi-dimensional data points, where each dimension corresponding to a feature taken from the feature space.

Fig. 6: The Pseudo-code for PageRank

```
val lines = ctx.textFile(args(0), 1)
val links = lines.map{ s =>
              val parts = s.split("\\s+")
              (parts(0), parts(1))
            }.distinct().groupByKey().cache()
var ranks = links.mapValues(v => 1.0)
for (i <- 1 to iters) {
   val contribs = links.join(ranks).values.flatMap
       { case (urls, rank) =>
   val size = urls.size
   urls.map(url => (url, rank / size))
   }
   Ranks = contribs.reduceByKey(_ + _).mapValues
       (0.15 + 0.85 * _)
}

val output = ranks.collect()
```

Fig. 7: The Pseudo-code for Logistic Regression

```
val points = spark.textFile(...).map(parsePoint).
    cache()
var w = Vector.random(D) // current separating plane
for (i <- 1 to ITERATIONS) {
  val gradient = points.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.
        x
  ).reduce(_ + _)
  w -= gradient
}
```

11 APPENDIX

11.1 Example Spark Applications

We estimate components of the job execution model by profiling certain representative Spark applications. Here we discuss a few representative Spark applications. Applications like iterative machine learning and interactive data mining applications, that reuse intermediate results are ideal candidates to represent jobs that are suitable for running on Apache Spark [8, 54]. We use the following example applications, belonging to the above category, as benchmarks for analyzing performance of Spark jobs.

11.1.1 WordCount

The WordCount program [55], that counts the number of words occurring in a file or group of files, is a popular example of parallel data processing applications. We chose WordCount as one of the example Spark applications for our analysis. It splits (i.e. partitions) the file(s) into chunks or blocks of data, in the form of key-value pairs, and distributes the blocks among the worker nodes in the cluster. Each worker thread performs the computations on an individual chunks in an independent concurrent manner, typically known as the map phase. Next the intermediate output files from individual workers are shuffled, i.e., sorted by key pairs, in the shuffle phase. Finally the shuffle outputs are grouped together by the keys, in what is typically regarded as the reduce phase.

We use the Scala implementation of the WordCount program found in the Spark example jar (see Figure 8). The WordCount program creates the SparkContext and loads the text files from HDFS. The flatMap method is used to split the text content on blank spaces, and create an RDD comprising the text fragments. The map method creates tuples $\langle k, v \rangle$ out of the RDDs, similar to

Fig. 8: The Pseudo-code for Wordcount

```
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line =>
                 line.split(" ")).map(word => (word,
                     1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

key-value pairs in Hadoop MapReduce. The value v in each tuple contains a number, initialized to 1, denoting the occurrence of the word corresponds to the key k. The reduceByKey method counts the occurrence of the words by adding the values v for all tuples. The collect method aggregates the occurrence values v from all tuples by the respective keys, and returns the resultant values as the final output.

11.1.2 Movie Rating using Alternating Least Squares Technique

Recommender systems typically use collaborative filtering techniques to predict what items users might like depending on similarity of users' past behavior, and preferences based on historic data. MovieLensALS [55] (see Figure 9) is a collaborative filtering based movie rating application, found in Spark's Machine Learning Library (MLlib) [56], a scalable library comprising API's for a host of learning algorithms. MovieLensALS performs training using ALS (i.e., Alternating Least Squares) on a dataset comprising latent factors defining a set of movies and ratings by users, and predict missing movie ratings.

The application runs an iterative machine learning algorithm, updating the features in the training dataset on each iteration, and has been used as an example in works dealing with Spark [8, 54]. The Movie rating application is particularly suited as an example Spark application representative of the group of applications requiring reuse of immediate output data generated from iterative operations [8].

Fig. 9: The Pseudo-code for MovieLensALS

```
val R = sc.textFile(params.input).map { line =>
val fields = line.split("::")}
val Rb = spark.broadcast(R)
for (i <- 1 to ITERATIONS) {
    U = spark.parallelize(0 until u)
    .map(j => updateUser(j, Rb, M))
    .collect()
    M = spark.parallelize(0 until m)
    .map(j => updateUser(j, Rb, U))
    .collect()
}
```

The ratings data file consists of three columns - a user, a movie, and the rating given by the user to the movie. The textFile method loads the data into the RDD named ratings. The randomSplit method splits the dataset into training and testing datasets. ALS trains a matrix factorization model that estimates a user-item association matrix named R, as the product of two lower-rank matrices M and U. Each row in M and U represents a feature vector comprising features of a user and a movie respectively. A user u's rating of a movie m is given as the dot product of the feature vectors corresponding to u and m respectively. The ratings

matrix R is a partially filled containing certain already known rating values provided by some users for some movies.

ALS iteratively performs optimizations on U and M, under given M and U respectively, while minimizing the error in R. The final U and M matrices are used to predict the unknown ratings using the $U \times M$ operation. The parallelization of ALS is achieved by using the parallelize method to perform the optimization operations on different rows of U and M on different worker nodes. The ratings matrix R is broadcasted to the nodes, and the collect method is used to aggregate the results from each node into R. The model.predict method returns the predicted ratings for unrated movies in form of the RDD named predictions comprising array of ratings.

11.1.3 PageRank

Another popular example for iterative computation intensive algorithm is PageRank [57], that ranks documents on the web based on the occurrence and importance of the links from the given web document to other documents. This algorithm particularly suits Spark because of the iterative nature, reuse of the intermediate results, and the scope for parallelising the computations on groups of documents. The program [55] (refer to Figure 10) iteratively updates the rank for each document by adding up contributions from documents that are linked from it.

On each iteration, each document i sends a value r_i denoting it's contribution to the overall rank to its neighbors. It (i.e., the web document) updates its own rank with the weighted sum of the rank contributions from all its neighbors. The lines.map function splits the web document into lines. The mapValues function creates a key-value pair named ranks. Then the links.join method identifies the url links in each line iteratively, and considers the contributions from the linked documents in each iteration. The reduceByKey method aggregates the contributions from the neighbors and updates the rank values by the weighted sum of the contributions. Finally the collect method computes the output ranks.

Fig. 10: The Pseudo-code for PageRank

```
val lines = ctx.textFile(args(0), 1)
val links = lines.map{ s =>
              val parts = s.split("\\s+")
              (parts(0), parts(1))
            }.distinct().groupByKey().cache()
var ranks = links.mapValues(v => 1.0)
for (i <- 1 to iters) {
   val contribs = links.join(ranks).values.flatMap
       { case (urls, rank) =>
   val size = urls.size
   urls.map(url => (url, rank / size))
   }
   Ranks = contribs.reduceByKey(_ + _).mapValues
       (0.15 + 0.85 * _)
}

val output = ranks.collect()
```

11.1.4 Logistic Regression

We choose the logistic regression program [55] (refer to Figure 11) from the MLlib library [56] as an example since it is an iterative machine learning algorithm, and used frequently as an example in works dealing with Spark and RDD. Logistic regression is used to classify a given dataset into sets of multi-dimensional data points, where each dimension corresponding to a feature taken from the feature space. Logistic Regression searches for a hyperplane that separates the dataset into two sets of points. It initializes the hyperplane w as a random vector, and iteratively improves w. On each iteration, it computes the gradient of each point, and improves the hyperplane w.

Fig. 11: The Pseudo-code for Logistic Regression

```
val points = spark.textFile(...).map(parsePoint).
    cache()
var w = Vector.random(D) // current separating plane
for (i <- 1 to ITERATIONS) {
  val gradient = points.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.
        x
  ).reduce(_ + _)
  w -= gradient
}
```

Subhajit Sidhanta Subhajit Sidhanta received his PhD in Computer Science from Louisiana State University, USA, in 2016. He spent one and a half year as a Postdoctoral Researcher with the Distributed Systems Research Group at INESC-ID research lab, affiliated with Instituto Superior Tecnico at University of Lisbon, Portugal, and then joined the Indian Institute of Technology Jodhpur in 2018 as an Assistant Professor in Computer Science and Engineering. His major research interest encompasses the areas of consistency in distributed systems, performance modelling of distributed storage and parallel computing systems.

Supratik Mukhopadhyay Supratik Mukhopadhyay is a Professor in the Computer Science Department at Louisiana State University, USA. He received his PhD from the Max Planck Institut fuer Informatik and the University of Saarland in 2001, MS from the Indian Statistical Institute, India and his BS from Jadavpur University, India. His areas of specialization include formal verification of software, inference engines, bigdata analytics distributed systems, parallel computing framework and program synthesis.

Wojciech Golab Wojciech Golab received his PhD in Computer Science from the University of Toronto in 2010. After a post-doctoral fellowship at the University of Calgary, he spent two years as a Research Scientist at Hewlett-Packard Labs in Palo Alto. He then joined the University of Waterloo in 2012, where he is presently an Associate Professor in Electrical and Computer Engineering. Dr. Golab's research agenda focuses on algorithmic problems in distributed computing with applications to the design, optimization, and verification of software systems for data storage and analytics. His research publications have won two best papers awards, and his doctoral work on shared memory algorithms was distinguished by the ACM Computing Reviews as one of 91 notable computing items published in 2012".

12 APPENDIX

13 SPARK JOB CONTROL FLOW

Spark jobs are typically launched as jobs by submitting the job code bundled with the dependency library packages [8, 34, 35]. The job can be executed in various modes - local mode, with local worker threads, YARN mode, on the YARN cluster manager, MESOS mode, on a cluster managed by Apache Mesos, or the simple standalone mode. In standalone mode, the submitted job jar is distributed uniformly across all worker nodes.

Spark can be configured to apply either static or dynamic scheduling policy for allocating resources across jobs. Static partitioning assigns the maximum available resource to each job. Dynamic resource allocation enables requesting for additional resources when there are pending unscheduled jobs and the total resources allocated are less than the currently available resources. The scheduler initiates request for workers or executors in rounds, with number of executors increasing exponentially in each round. For scheduling operations comprising a submitted job, Spark uses the FIFO scheduling policy by default. In later Spark versions, it is possible to configure the Fair Scheduler, that uses Round-Robin time-sharing techniques. It is also possible to group operations into pools, thus enabling prioritization of operations belonging to different pools.

Spark uses RDD (i.e., resilient distributed datasets), a read-only distributed data structure for storing and manipulating the data in memory. RDDs facilitate fault tolerance, optimized partitioning scheme, and extremely fast computation using in-memory operations. RDDs are created through coarse-grained transformations like map, join operations, from HDFS files or from other RDDs. Spark job execution flow typically consists of several unit operations on the input, which are internally represented as RDD's. An RDD operation can be: 1) a transformation, that creates new RDD's from HDFS files or from existing RDD's, or 2) an action, that can return a value from the RDD or export the RDD to the HDFS. An RDD transformation happens in the Preparation phase where the input HDFS files are transformed to working RDD's. For each RDD action, the SparkContext initializes an RDD unit operation, and submits the operation to the Spark Scheduler. The Spark DAGScheduler partitions the operation into stages, creates a DAG consisting of the stages in the given operation, and creates markers for the RDD's used in each stage of the DAG. It also handles resubmission of failed operations within a particular stage due to loss of shuffle outputs (i.e. outputs corresponding to various concurrent operations working on different partitions of the given RDD). The DAGScheduler submits the operation stages to the TaskScheduler, which, in turn, sends the operations to the cluster to be executed remotely on the workers. Also, TaskScheduler listens for results, and resubmits the failed operation stages.

Typically, the number of partitions/slices is set automatically based on the input file size, the HDFS block size, and the cluster configuration. It can be configured programmatically by a parameter to the RDD transformation function. Controlling the partitioning on RDD's enable speeding up Spark job throughput by minimizing the communication delay through increased localisation of the data. Key-value RDD's or pair RDD's can be created using transformation operations on existing RDD's or input HDFS collections, like map function. Pair RDD's allow concurrent parallelized operations on each key, and subsequent aggregation of the values for each key by actions like reduceByKey or join.

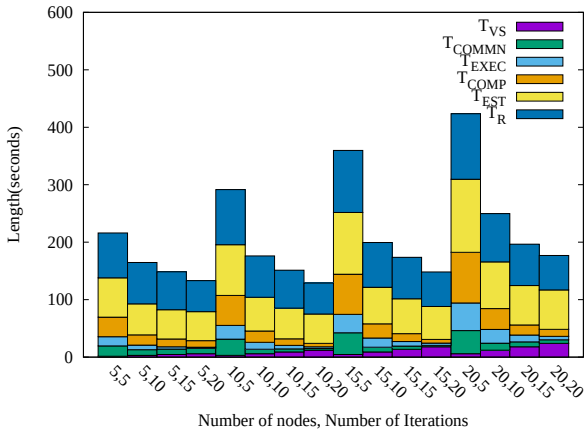


Fig. 12: Estimated and Recorded Values of Various Components of Job Execution Times

13.1 Spark Job Execution Internals

Spark maintains the lineage information for the RDDs created and used during the course of execution of a given job, which enables it to reconstruct RDDs that may be lost during failure of certain job stages. The lost RDDs can then be re-created from their parent RDDs or directly from the input data files. This ensures fault tolerance in Spark in a lightweight manner without resorting to resource expensive techniques like repeating the entire job or checkpointing by storing snapshots of actual data generated in different stages. Spark uses the lineage graph of a given job to generate a logical execution plan for the job. The lineage graph is materialized when a spark job is submitted to the Spark Driver, i.e., when the SparkContext for the job is created from the source code for the given job. The logical execution plan is converted into a physical execution plan for the job by the DAGScheduler, i.e., the scheduling component of Spark. The logical execution plan represented by the lineage graph captures the dependence among RDDs manipulated by the given job, whereas the physical execution plan captures the breakdown of the job in terms of stages.

13.2 Confidence Interval Under Varying Choice of Representative Jobs

The completion times of Spark jobs depend on the components of the job profiles generated using representative jobs for the job categories corresponding to the target jobs. Under the given assumptions regarding the choice of job category (Section 7.2), Table 6 gives the mean, standard deviation, variance, and 95% confidence intervals for the estimated completion time T_{Est}, under varying choice of representative jobs. The function T_{Est} is a nonlinear convex function over the integer variable n, i.e., $T_{Est} = f(n)$ (see Section 13.3). Let μ and σ be the sample mean (or expectation) and variance of the job completion times for the experiments, under varying choice of representative job for each category, given in Table 6. The standard deviation and variance [23] of the function represents the stability of the function f, under given variations in the choice of representative job. We compute the observed standard deviation and variance for the experiments in Section 7.6, and present the results in Table 6. By definition [23], the confidence interval is computed from a given sample of observed values, in terms of the sample expectation

TABLE 6: Confidence Interval of Estimation With Varying Choice of Representative Jobs for Each Job Category in Seconds

Category	Standalone				YARN			
	Standard Deviation	Variance	Mean	Length of 95% Confidence Interval	Standard Deviation	Variance	Mean	Length of 95% Confidence Interval
Data Mining	24	720	66	22	42	1800	138	18
Statistical Evaluation	18	360	342	168	174	29,520	798	84
Quantitative Analysis	120	14,040	1,956	60	198	38,520	2,934	66
Miscellaneous	498	2,50,920	1,602	26.4	354	1,23,840	3,180	174

μ and variance σ, as $[\mu - z^* \frac{\sigma}{\sqrt{N}}, \mu + z^* \frac{\sigma}{\sqrt{N}}]$, where N is the number of observations, z^* is the critical value [23] that depends on the assigned confidence level CI (e.g., here CI=95% and z^* = 1.96). α is the frequency of correct results in an observed sample, computed as $\alpha = 1 - \frac{CI}{100}$. From the sample of observed values in Table 6, we compute the 95% confidence interval (i.e., for $CI = 95$) for each sample (i.e., for each row in the table) using the above formulas, using the final values of μ and σ, given in Table 6. With $CI = 95$, $100(1 - \alpha)$ = 95, hence α = 0.05. The confidence interval for each sample is given in the columns 5 and 9 of Table 6. The confidence interval acts as a tolerance bound that limits the estimated values of T_{Est} within an acceptable range. Here we assume that the random variables in the estimated job execution time follow a normal distribution for the following reason. The exact nature of the distribution is unknown because of the absence of sufficiently large dataset comprising all possible variations of the input parameters. Specifically, because of the dependency of the model on the choice of the job categories and the representative jobs for each category, the distribution can only be approximated. Normal distribution is chosen since the central limit theorem states that most random variables can be approximated by a normal distribution. As per the definition of confidence interval [23], as long as 95% of the estimated T_{Est} values remain within the interval $[\mu - 1.96\frac{\sigma}{\sqrt{N}}, \mu + 1.96\frac{\sigma}{\sqrt{N}}]$, computed using the above approach, OptEx can estimate execution time of any given application with 95% confidence level. In Table 6, for 95% confidence level, the estimated completion time for PageRank job (representative of Quantitative Analysis application category) falls within the range 1920 and 1980 seconds with accuracy of 70% for standalone mode, and within 2938 and 3004 seconds with accuracy of 70% for YARN mode, with a confidence of 95%. Similarly, the 95% confidence interval for MovieLensALS (representative of Data Mining application category) is 55 to 77 seconds with 80% accuracy in standalone mode, and 116 to 134 seconds with 90% accuracy in YARN mode. The 95% confidence interval for ChiSqTest (representative of Statistical Evaluation application category) is 234 to 402 seconds with 100% accuracy in standalone mode, and 736 to 820 seconds with 75% accuracy in YARN mode. The confidence intervals are directly proportional to the respective standard deviations and inversely proportional to the square roots of the means. That is why the values of the confidence interval in Table 6 are considerably smaller than the standard deviation.

13.3 Some Important Mathematical Results

The constrained cost optimization (refer to Section 6) of the cost function $\mathcal{C}$ in Equation 9 is based upon the assumption that the constraint $T_{Est} < SLO$ is a nonlinear convex function over n and $\mathcal{I}$. We rewrite the constraint as $T_{Est} - SLO < 0$. Further, we rewrite the expression for the constraint by replacing the term T_{Est} by the expression in Equation 8 as follows.

$$X_1 + n \times \mathcal{I} \times C + \mathcal{I} \times B/n + \frac{A \times s}{n} - SLO < 0, \quad (13)$$

where $X_1 = T_{Init} + T_{prep}$. By inspection, we observe that the expression in Equation 13 is a function of n and $\mathcal{I}$. We give formal proofs for the convexity of the above function in the following lemmas. The first lemma proves convexity under the assumption that the function is continuous, i.e., variables n and $\mathcal{I}$ are real variables. This proof is relevant, since it allows us to alternatively assume the constraint to be a continuous function, perform convex optimizations, and then round off real valued outputs to nearest integral values. However, in this paper, we consider the function to be discrete, and perform discrete optimization. Later, we prove convexity of the function, assuming that it is discrete.

Lemma 13.1. *Assuming n and $\mathcal{I}$ are real variables, the expression $T_{Est} < SLO$ is a nonlinear convex function with respect to n and $\mathcal{I}$.*

Proof: From inspection, it can be seen that the expression in Equation 13 is a nonlinear function of n and $\mathcal{I}$. Hence, we denote the expression as a function $\mathcal{G}(n, \mathcal{I})$. By assumption, $\mathcal{G}$ is continuous function. Differentiating $\mathcal{G}$ with respect to n, we have

$$\frac{\mathrm{d}\mathcal{G}(n, \mathcal{I})}{\mathrm{d}n} = \mathcal{I} \times C - \mathcal{I} \times B/n^2 - A \times s/n^2. \quad (14)$$

Since $n \geq 1$, the expression of the first derivative in Equation 14 evaluates to real values for all possible values of the input variable n. Hence, $\mathcal{G}(n, \mathcal{I})$ is differentiable with respect to n [23]. Differentiating again with respect to n,

$$\frac{\mathrm{d}^2\mathcal{G}(n, \mathcal{I})}{\mathrm{d}n^2} = \mathcal{I} \times B/n^3 + A \times s/n^3. \quad (15)$$

Inspecting Equation 15, we observe that second derivative exists, and the second derivative ≥ 0. Hence, by definition [23], $\mathcal{G}(n, \mathcal{I})$ is twice differentiable with respect to n. Similarly, differentiating the expression for $\mathcal{G}$ with respect to $\mathcal{I}$,

$$\frac{\mathrm{d}\mathcal{G}(n, \mathcal{I})}{\mathrm{d}\mathcal{I}} = n \times C + B/n. \quad (16)$$

Following the previous line of argument, since $\mathcal{I} \geq 1$, the first derivative with respect to $\mathcal{I}$ exists for all possible values of $\mathcal{I}$. Hence, $\mathcal{G}(n, \mathcal{I})$ is differentiable with respect to $\mathcal{I}$. Differentiating again with respect to $\mathcal{I}$,

$$\frac{\mathrm{d}^2\mathcal{G}(n, \mathcal{I})}{\mathrm{d}\mathcal{I}^2} = 0. \quad (17)$$

In Equation 17, the second derivative is a constant value 0, independent of $\mathcal{I}$. Hence, the second derivative of $\mathcal{G}(n, \mathcal{I})$ exists against all possible values of the input variable $\mathcal{I}$. Also, in Equation 17, the second derivative evaluates to ≥ 0. Hence, by definition, $\mathcal{G}(n, \mathcal{I})$ is twice differentiable with respect to both n and $\mathcal{I}$.

By definition [23], a continuous function is convex, if second derivative exists, and if the second derivative is ≥ 0 for all values of the input variables. Following Equations 15 and 17, Lemma 13.2 holds under the above assumptions. However, in this paper, we consider that the constraint $T_{Est} - SLO < 0$ is a discrete function of n and $\mathcal{I}$. Next, we prove the above constraint is a

convex nonlinear function using the approach described in [58], under the condition that the expression for the constraint is, in fact, a discrete function $\mathcal{G}$ of integer variables n and $\mathcal{I}$.

Lemma 13.2. *If n and $\mathcal{I}$ are discrete integer variables, the expression $T_{Est} < SLO$ is a nonlinear convex function with respect to n and $\mathcal{I}$.*

Proof: Following Lemma 13.1, we express the constraint $T_{Est} - SLO < 0$ as a function $\mathcal{G}(n, \mathcal{I})$. By definition [58, 23], a function $\mathcal{G}$ is convex if the domain dom of $\mathcal{G}$, is a convex set, and $\forall p, q \in dom$,

$$\mathcal{G}(\phi \times p + (1-\phi) \times q) \leq \phi \times \mathcal{G}(p) + (1-\phi) \times \mathcal{G}(q), \tag{18}$$

where $0 \leq \phi \leq 1$. Since n and $\mathcal{I}$ are integer values ≥ 0, the domain of $\mathcal{G}$ is a convex set. Next, to prove that function $\mathcal{G}$ follows the definition of convexity, given by the inequality in Equation 18, we reduce the left hand side (LHS) and right hand side (RHS) of the inequality. We use the expression of $\mathcal{G}$ from Equation 13 in the Equation 18. We rewrite the expression in Equation 18 in two steps; first, we expressing the variables n in terms of variables p, q, and constant term ϕ, respectively. Then, we repeat the above process for the variable $\mathcal{I}$.

First, we reduce the expression $\mathcal{G}(\phi \times p + (1-\phi) \times q)$ in the LHS as follows. We replace the term $\mathcal{G}$ in the LHS with the expression of $\mathcal{G}$ in Equation 13. Since $p, q \in dom$, next we replace the term n in the expression of $\mathcal{G}$ with the expression $\phi \times p + (1-\phi) \times q$ to obtain the expression for $\mathcal{G}(\phi \times p + (1-\phi) \times q)$. Thus, $\mathcal{G}$ can be rewritten in the form of the LHS of Equation 18 as

$$\begin{aligned} \mathcal{G}(\phi \times p + (1-\phi) \times q) = X_1 + \\ (\phi \times p + (1-\phi) \times q) \times \mathcal{I} \times C \\ + \frac{\mathcal{I} \times B}{(\phi \times p + (1-\phi) \times q)} + \frac{A \times s}{(\phi \times p + (1-\phi) \times q)}. \end{aligned} \tag{19}$$

Applying associativity, we rewrite Equation 19 as

$$\begin{aligned} LHS = X_1 + \phi \times p \times \mathcal{I} \times C + (1-\phi) \\ \times q \times \mathcal{I} \times C + \frac{\mathcal{I} \times B}{(\phi \times p + (1-\phi) \times q)} + \\ \frac{A \times s}{(\phi \times p + (1-\phi) \times q)}. \end{aligned} \tag{20}$$

Thus, LHS can be written as

$$LHS = X_1 + T_1 + T_2 + T_3 + T_4, \tag{21}$$

where $T_1 = \phi \times p \times \mathcal{I} \times C$, $T_2 = (1-\phi) \times q \times \mathcal{I} \times C$, $T_3 = \frac{\mathcal{I} \times B}{(\phi \times p + (1-\phi) \times q)}$, and $T_4 = \frac{A \times s}{(\phi \times p + (1-\phi) \times q)}$. Similarly, replacing each occupancy of n in the expressions of $\mathcal{G}$ in the RHS with the expression $\phi \times p + (1-\phi) \times q$, we have

$$\begin{aligned} \mathcal{G}(p) + (1-\phi) \times \mathcal{G}(q) = X_1 + \\ \phi \left(p \times \mathcal{I} \times C + \frac{\mathcal{I} \times B}{p} + \frac{A \times s}{p} \right) + \\ (1-\phi) \left(q \times \mathcal{I} \times C + \frac{\mathcal{I} \times B}{q} + \frac{A \times s}{q} \right). \end{aligned} \tag{22}$$

Applying associativity, we rewrite Equation 22 as

$$RHS = X_1 + T_1 + T_2 + T_5 + T_6, \tag{23}$$

where $T_5 = \phi \times \left(\frac{\mathcal{I} \times B}{p} \right) + (1-\phi) \times \left(\frac{\mathcal{I} \times B}{q} \right)$, $T_6 = \phi \times \left(\frac{A \times s}{p} \right) + (1-\phi) \times \left(\frac{A \times s}{q} \right)$. In Equations 21 and 23, since by inspection, we observe that $T_5 + T_6 \leq T_3 + T_4$, it follows that LHS $\leq$ RHS in Equation 18, with respect to n.

Similarly, we replace the term $\mathcal{I}$ in the expression of $\mathcal{G}$ in Equation 18 with the expression $\phi \times p + (1-\phi) \times q$, and prove that LHS $\leq$ RHS in Equation 18, with respect to the input variable $\mathcal{I}$. It directly follows from the above that $\mathcal{G}$ conforms to the definition in Equation 18; hence, it is a convex nonlinear function over n ad $\mathcal{I}$.