OptEx: A Deadline-Aware Cost Optimization Model for Spark

Subhajit Sidhanta*, Wojciech Golab†, and Supratik Mukhopadhyay*
*Louisiana State University, Baton Rouge, Louisiana, USA, Email: {ssidha1, supratik}@csc.lsu.edu
†University of Waterloo, Waterloo, Ontario, Canada, Email: wgolab@uwaterloo.ca

***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, 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 cost optimal cluster composition for running a given Spark job under an SLO deadline with an accuracy of 98%.**

I. 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 applications 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, the current state-of-the-art [2, 3, 4] cluster provisioning solutions do not ensure that a given SLA deadline for job execution is satisfied, while at the same time above service usage cost is minimized.

We present OptEx[1], a closed-form job execution model for Apache Spark [5], a popular parallel processing engine. OptEx can be used to determine the cost optimal cluster composition, comprising virtual machine instances provided by cloud service providers, like Amazon, RackSpace, Microsoft, etc., 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. OptEx analytically models the *job completion time* of Spark jobs on a cluster of virtual machine instances. It 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 application categories, and generates separate job profiles for each application 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 application 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 the 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 design an optimal schedule for running a given job on a given cluster composition under an SLA deadline.

Consider the use case where a web development company needs to run a Spark PageRank application to determine the most important web pages they developed over the years, using the infrastructure (cluster) provided by a popular cloud provider, like Amazon. Using state-of-the-art [2, 3, 4] prior experience-based provisioning techniques, the company may provision a cluster of 30 m2.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 m2.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 Spark [5] job execution.
- We provide a technique for estimating the cost optimal cluster composition for running a given Spark job

[1]The project is partially supported by Army Research Office (ARO) under Grant W911NF1010495. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the ARO or the United States Government.

under an SLA deadline, using the above model.

A. Motivation

Scaling out (i.e., adding nodes to the cluster) [6] is the common way of increasing the performance of parallel processing on the cloud. Cost is an important factor in scaling out, with the cost of cluster usage increasing linearly with the number of virtual machine instances in a cluster, evident from analysis of the Amazon Ec2 pricing policy [1]. Hence for minimizing the service usage cost, determining the optimal cluster size for executing a given job is of utmost importance. In the current state-of-the-art [2, 3, 4], a cloud service consumer can choose the required cluster configuration in one of the following manners: 1) arbitrarily, or 2) make an informed decision using previous experience of running similar jobs on the cloud. However none of the above strategies ensure that the SLA deadline is satisfied, and at the same time the service usage cost is minimized. Elastisizer [6] is the only successful work in this direction, but it addresses Hadoop MapReduce and does not address Spark.

B. The OptEx Approach

OptEx decomposes a Spark job execution into different phases, namely the initialization phase, the preparation phase, the variable sharing phase, and the computation phase. Following an analytical modelling approach, OptEx expresses the execution time of each phase in terms of the cluster size, number of iterations, the input dataset size, and certain model parameters. Similar to the ARIA framework [7], which applies profiling for scheduling Hadoop MapReduce jobs, OptEx estimates the model parameters with the components of the specific job profile corresponding to the application category of the target job. ARIA uses Hadoop-specific parameters for profiling [7], and hence is unsuitable for application to Spark. OptEx considers job completion deadline as an SLA parameter [8, 9], that acts as the constraint for minimizing the service usage cost. An objective function for the service usage cost is obtained based on the job execution model. Constrained optimization techniques [10] are applied on the objective function to estimate the cost optimal cluster composition for finishing a Spark job within a given SLA deadline.

II. Spark Application Control Flow

Spark applications are typically launched as jobs by submitting the application code bundled with the dependency library packages. 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 [11], or the simple standalone mode. In standalone mode, the submitted application jar is distributed uniformly across all worker nodes.

Spark can be configured to apply either static or dynamic scheduling policy for allocating resources across applications. Static partitioning assigns the maximum available resource to each application. 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 tasks comprising a submitted job, Spark uses the FIFO scheduling policy [12, 13] by default. In later Spark versions, it is possible to configure the Fair Scheduler, that uses Round-Robin time-sharing techniques [14, 15, 16]). It is also possible to group tasks into pools, thus enabling prioritization of tasks belonging to different pools.

A Spark job execution flow typically consists of several unit operations on the input, which is 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 task, and submits the task to the Spark Scheduler. The Spark DAGScheduler partitions the task into stages, creates a DAG consisting of the stages in the given task, and creates markers for the RDD's used in each stage of the DAG. It also handles resubmission of failed tasks within a particular stage due to loss of shuffle outputs (i.e. outputs corresponding to various concurrent tasks working on different partitions of the given RDD). The DAGScheduler submits the task stages to the TaskScheduler, which, in turn, sends the tasks to the cluster to be executed remotely on the workers. Also, TaskScheduler listens for results, and resubmits the failed task 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.

III. Spark Job Execution Phases

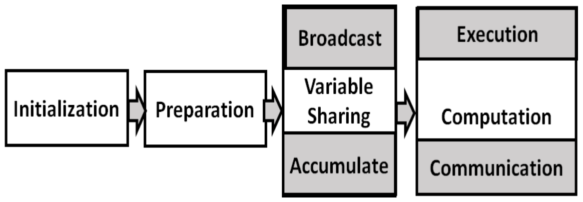


Fig. 1: Phases in a Spark Job Execution Flow

We decompose a typical Spark job execution flow into logically distinct phases illustrated in Figure 1. Each of these phases behave differently with respect to variations in the number of iterations, the cluster size, and the dataset size. 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 [5]. The next phase is the *variable sharing phase* that deals with broadcasting or accumulating blocks of data from the Spark master to the workers.

Spark uses a novel in-memory data structure called the *RDD* (i.e., resilient distributed dataset) for fast and fault tolerant computation [5]. Internally, each Spark job is processed as a permutation of several *unit RDD tasks* (operations like flatmap, reduce, etc), that are executed in parallel on the worker nodes. Spark provides a wide range of built-in unit RDD tasks, packaged within several library modules [17], like MLlib, Spark SQL modules, etc. During the last phase, i.e., the *computation phase* (Figure 1), the given application makes calls to methods from the above library modules, which in turn triggers the respective unit RDD tasks 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 tasks on the workers. The lengths of the variable sharing phase and the computation phase monotonically increase with the input variables, i.e., the number of iterations, the cluster size, and the dataset size [5]. 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.

IV. Application of Profiling For Estimating the Model Parameters

The most common technique for estimating the performance of a given job [7, 18] is using a standard profiling tool, that measures real-time performance statistics, to generate job profile for the target job using a representative job. OptEx categorizes Spark applications, and uses profiling to generate separate job profiles for each application category with representative jobs for each category. Components of the job profile are used as estimates for the model parameters of the target job.

A. Application Categorization

As discussed earlier, OptEx categorizes Spark applications into application categories, chooses a representative job for each category, and creates the job profile for that category using the respective representative job. Application categorization is a difficult open problem, dependent on the application domain, and beyond the scope of this work. OptEx allows the developers to choose their own categorization scheme.

In this work, we categorize Spark applications according to the category of library modules that an application uses. Each Spark application uses specific libraries depending on the business logic [5]. The Apache Spark distribution [17] currently organizes the library modules into following four categories: 1) Spark SQL, which supports Apache Hive or JDBC queries, 2) Spark Streaming, which comprises streaming applications, 3) MLlib, which supports machine learning, and 4) GraphX, which facilitates working with graphs and collections. Thus, OptEx uses four application categories, and a specific job profile for each category is obtained using a representative job for each category.

B. Choice of Representative Jobs For Each Category

The execution phase (Figure 1) of a Spark job comprises a permutation of low-level unit RDD tasks. We call an application a to be a representative job for a given job j, if: 1) the job a contains all the unit RDD tasks comprising the job j, and 2) if job j is iterative, a is also iterative, and vice versa. According to the given categorization scheme (Section IV-A), OptEx categorizes applications on the basis of the Spark library modules they use. The Apache Spark distribution web page [17] describes an example application for each group of library modules. For a given application category, the respective example application is chosen as the representative job, under the given categorization scheme. By the design of the Spark libraries, these chosen jobs trivially satisfy the above two conditions for being a representative job (see Section VIII).

The custom Spark application, mentioned as an example in the web page of Spark Streaming library [17], is used as the representative job for the applications using the Spark Streaming. It runs on the Twitter dataset [19] , and lists the current tweets on a sliding window. Similarly, the representative application for the MLlib group of applications is the movie rating application MovieLensALS [17]. The input workload for the MovieLensALS applications is the MovieLens dataset made available by Netflix at grouplens.org [20]. PageRank is the representative application for the GraphX group of applications [17]. In the absence of an example application for the Spark SQL category, the widely used Big Data Benchmark [21] developed by AMPLab is used as the representative job for this category. It has been widely accepted as a benchmark for Spark SQL jobs [17].

C. Estimation of Model Parameters from the Job Profile

TABLE I: 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	$iter$	Number of iterations
t	number of possible instance types	s	size of input dataset
T_{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}

TABLE II: An Example Job Profile: Profile for MLlib jobs 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	0.07	mean	100
							map	98
							flatmap	72
							first	5
							count	124
							distinct	300

A Spark job is typically written in a high-level language (like Scala, Python, etc.) internally executed in different phases (Figure 1). The length of the initialization phase T_{Init} and the length of the preparation phase T_{prep} remain constant to variations in the input variables [5]. 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 [5]. 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. 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 application 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 length 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 II) 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 III, these phases remain constant with respect to the variations in the input variables). As elaborated in Section III, the length of the variable sharing phase T_{vs} increases monotonically [5] 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 $iter$ 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 application. It is the baseline for estimating the length of the variable sharing phase T_{vs} of a given target job.

The increase in the length of the variable sharing phase T_{vs}, estimated relative to the baseline $T_{vs}^{baseline}$, in terms of the given values of the input variables n and $iter$ (we compute the total duration of the variable sharing phase across all iterations), i.e., T_{vs} is expressed as:

$$T_{vs} = coeff \times iter \times n \times T_{vs}^{baseline}, \tag{1}$$

where $iter$ 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 I) 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 tasks in the earlier stages of the given job. While profiling, the length of the communication phase of a representative application 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 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 application is given as the size of the input files). Since the length of the communication phase T_{commn} increases with respect to the input variable s [5], T_{commn} is expressed as a product of the input dataset size s, a coefficient cf_{commn}, and the baseline value $T_{commn}^{baseline}$, where cf_{commn} and $T_{commn}^{baseline}$ are obtained from the job profile. Again, the coefficient cf_{commn} is empirically estimated in the profiling stage applying curve fitting on the outputs of experiments with the representative job. Thus,

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

As discussed in Section IV-B, the execution phase of a Spark job comprises a permutation of unit RDD tasks. The OptEx job profile records the average running time M_a^k (Table I) of each unit RDD task component k comprising the representative Spark application a. If there are multiple occurrences of an RDD task i in a, we consider the average running time for all occurrences of the task i. By design the representative job for an application category contains all the unit RDD tasks 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_a^k of each unit RDD task k in the job profile.

V. Derivation of the Spark Job Execution Model

A. 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 $iter$ in the given job [5]. The number of iterations for an iterative Spark application is typically passed as a runtime argument by the developer [5]. Moreover, Spark applications 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 $iter$ 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 $iter_{exec}$ as a runtime argument to the job [5]. The number of iterations $iter_{exec}$ provided in the running time may differ from the number of iterations $iter$ provided in the modelling phase. The difference between $iter_{exec}$ and $iter$ 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 $iter_{exec}$ supplied by the user in each run, the maximum of the $iter_{exec}$ values, i.e., $iter_{exec}^{max}$, is supplied as the new input for the estimation. The estimation using the new value $iter_{exec}^{max}$ 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 [10]), thus incurring negligible overhead.

B. 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} is the length of the initialization phase, T_{prep} is the length of the preparation phase, T_{vs} is the length of the variable sharing phase, and T_{comp} is the length of the computation phase. As discussed in Section IV-B, the execution phase of a given Spark job comprises a permutation of low-level unit RDD tasks. The number of unit RDD tasks n_{unit} increases monotonically with increasing the input dataset size s and the number of iterations $iter$ [5]. Hence, the number of unit RDD tasks n_{unit} can be expressed as a function comprising the following terms:

- The size of the input dataset is denoted as s.
- The number of iterations in the job is given as $iter$.
- The baseline term for the number of unit RDD tasks is given as $n_{unit}^{baseline}$. It is obtained from the job profile (Section IV-C). Spark enables parallel execution by dividing the input dataset into partitions, and distributing the partitions/slices among the worker nodes [5]. $n_{unit}^{baseline}$ 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 $iter$ [5], or 2) programmatically provided as a parameter to the built-in transformation method used to create the RDDs from the input dataset [5].
  For example, the Spark Wordcount 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 Wikipedia dump [23] 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 tasks is 164. Thus, the baseline $n_{unit}^{baseline}$ is 164.

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

$$n_{unit} = n_{unit}^{baseline} \times s \times iter. \tag{4}$$

As discussed already in Section IV-C, 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 IV-C). As discussed in Section III, the length of the variable sharing phase T_{vs} and the length of the computation phase T_{commn} vary with respect to 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 IV-C). The expression for the length of the variable sharing phase T_{vs}, comprising the baseline value $T_{vs}^{baseline}$ 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)** $\boldsymbol{T_{commn}}$: The length of the communication phase T_{commn} is obtained from the Equation 2. **B)** $\boldsymbol{T_{exec}}$: 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 IV-C). T_{exec} depends on various factors [5]: 1) the running times of the unit RDD tasks comprising the given job, 2) the number of iterations $iter$, 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 tasks 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:

$$T_{exec} = iter \times \sum_{k=1}^{n_{unit}} M_a^k, \tag{5}$$

where n_{unit} is the number of unit RDD tasks given in Equation 4, M_a^k is the average job execution time of a unit RDD task k comprising the job a, and $iter$ is the number of iterations in the job.

Following prior work on modelling execution of parallel tasks [7], the overall length of the computation phase T_{comp} is divided by the factor n, taking into account the parallelization of the 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

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

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

TABLE III: Stepwise Accuracy of Estimations

$iter$	n	T_{vs}(sec)	T_{commn}(sec)	T_{exec}(sec)	T_{comp}(sec)	T_{Est}(sec)	T_{Rec}(sec)
5	5	1.5	18	16	34	68.52	78
5	10	3	9.88	8	17.88	53.88	72
5	15	4.5	9.5	4	13.5	51	66
5	20	6	9.3	2	11.4	50.4	54
10	5	3	28.2	24	52.2	88.2	96
10	10	6	7.74	12	19.74	58.74	72
10	15	9	5.4	6	11.4	53.4	66
10	20	12	3	3	6	51	54
15	5	4.5	37.9	32	69.9	107.4	108
15	10	9	8.3	16	24.6	63.6	78
15	15	13.5	5.7	8	13.7	60.7	72
15	20	18	2.4	4	6.4	57.4	60
20	5	6	40.2	48	88.2	127.2	114
20	10	12	12.2	24	36.2	81.4	84
20	15	18	8.5	12	17.5	68.5	72
20	20	24	6.2	6	12.2	68.52	60

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

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

where n_{unit} is the number of unit RDD tasks given in Equation 4, $A = \frac{cf_{commn} \times T_{commn}^{baseline}}{s_{baseline}}$, $B = \sum_{k=1}^{n_{unit}} M_a^k$, and $C = coeff \times T_{vs}^{baseline}$. Table III shows the stepwise calculations for the length of the various phases in the estimated total completion time T_{Est} for the MovieLensALS application, in standalone mode, with varying number of nodes n, and the number of iterations $iter$, on m1.large Ec2 instance.

VI. Estimation of 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 (depends on the instance offerings of the chosen cloud provider), and m is the total number of possible machine instance types. 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 (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 described as a convex nonlinear function (see Section VIII) over n (Equation 8), and is twice differentiable with respect to n, i.e., both first and second derivatives of T_{Est} exist with respect to n (see Section VIII). The above optimization problem, for minimizing the cost $\mathcal{C}$, under the nonlinear constraint $T_{Est} < SLO$ is solved using the Interior Point algorithm [10]. 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.

VII. Implementation and Evaluation

A. Experimental Setup

The experimental setup consists of Apache Spark version 1.2.1, built-in within the Cloudera Express 5.3.1 package, on a cluster of m1.xlarge Amazon EC2 machine instances, each comprising 8 cores, 15 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. The underlying Hadoop cluster has a namenode, a secondary namenode, and 5 datanodes under a replication factor of 3.

We have dedicated machines for the Cloudera manager services, the HDFS namenode, the Spark master, and the Spark workers. The Spark driver or gateway node is collocated with one of the HDFS datanodes, and the YARN services are located in a separate set of nodes.

As explained in Section IV-A, we allow the developers to categorize jobs based on the respective application domain, use case, and user demands. For the purpose of our experiments, as discussed in Section IV-A and Section IV-B, we have followed the categorization scheme of library modules in the Apache Spark distribution web page. We have chosen representative jobs for each category based on the choice of example jobs in the Apache web page.

B. Experimental Procedure

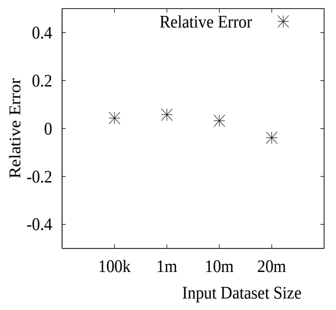
Applications like iterative machine learning and interactive data mining applications, that reuse intermediate results are ideal candidates to represent jobs that are suitable for running on Apache Spark [5, 24]. We use example applications described in Section XIII, as benchmarks for analyzing performance of Spark jobs. We use the Interior point algorithm [10] from the Optimization toolbox of the Matlab version 2013b for solving the given non-linear convex optimization problem (Section VI), and estimating the cost optimal cluster composition. We use the default FIFO scheduler. The average scheduler delay is 4 ms, and can be neglected relative to the other components of the execution time. The input workload for the MovieLensALS application is the 10-M MovieLens dataset obtained from grouplens.org [20]. PageRank is evaluated with the social network dataset for LiveJournal [23], an online community comprising roughly 10 million members. The LiveJournal dataset has 4847571 nodes and 68993773 edges. The input workload for the Wordcount application are the Wikipedia dumps obtained from SNAP [23].

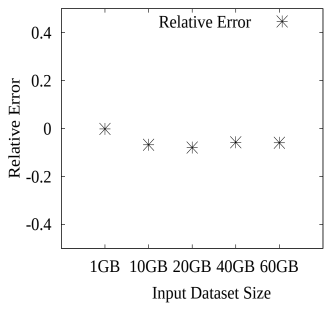
C. Technique for Generating Job Profiles

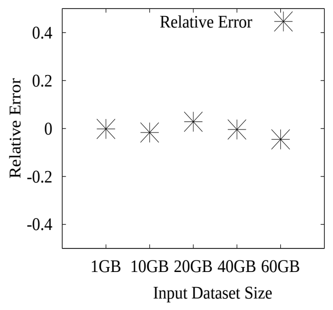
For computing the job profile of a target given job j with respect to a virtual machine instance type t, we run the representative job a (Section IV-B) for the application category corresponding to the target job j (Section IV-A), on a cloud instance of type t in stand-alone mode with a benchmark workload [25]. We estimate the components of the job profile from the snapshots of the execution flow of the representative job obtained using YourKit Java Profiler [26]. To minimize overhead, YourKit is run in the sampling mode.

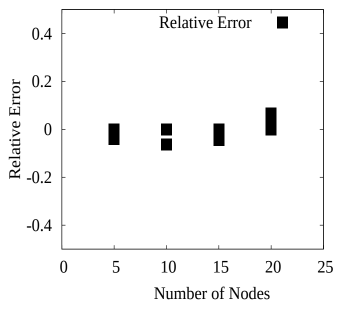
D. Accuracy of the Estimations Using OptEx

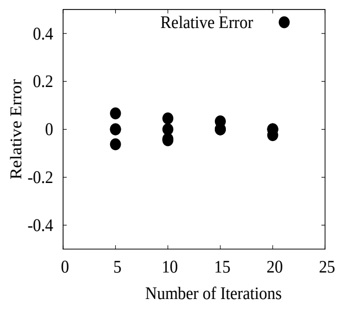
Being the first work in modelling Spark jobs, OptEx has no prior baseline to compare with. However, we demonstrate (see Figures 2 and 3) that OptEx provides accurate (i.e., average relative error 0.06) estimations of the job completion time against variations in all the input parameters of the model (i.e, against increasing size of dataset, number of nodes, and

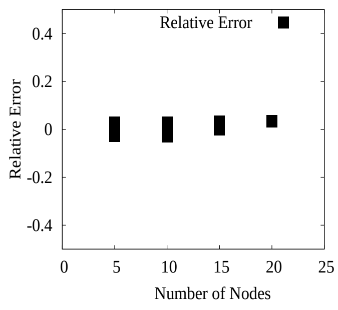

(a) Accuracy of Estimation for ALS With Varying Input DataSet Size in stand-alone mode with 20 worker nodes

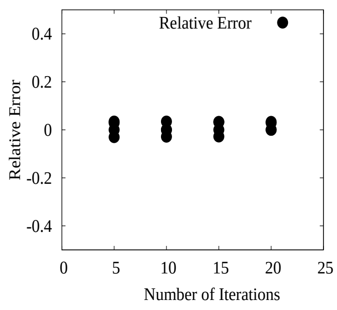

(b) Accuracy of Estimation for PageRank With Varying Input DataSet Size in stand-alone mode 20 worker nodes and 5 iterations

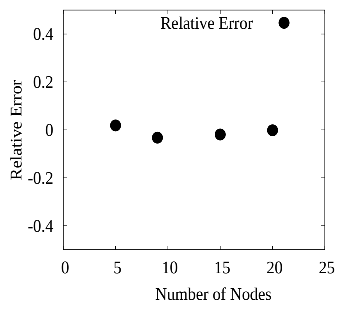

(c) Accuracy of Estimation for WordCount With Varying Input DataSet Size in stand-alone mode with 20 worker nodes


(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 Logistic Regression With Varying Number of Nodes in stand-alone mode


(g) Accuracy of Estimation for Logistic Regression With Varying Number of Iterations in stand-alone mode


(h) Accuracy of Estimation for WordCount With Varying Number of Nodes in stand-alone mode

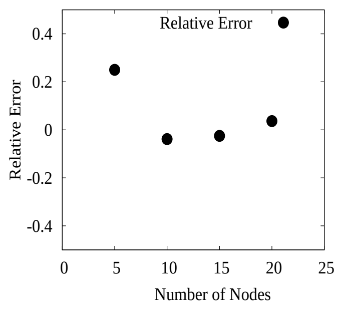
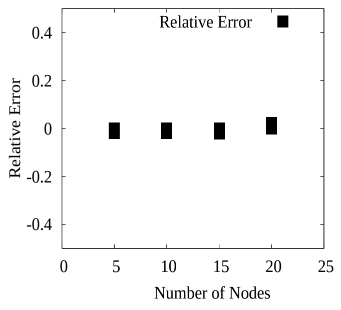

(i) Accuracy of Estimation for WordCount With Varying Number of Nodes in YARN mode

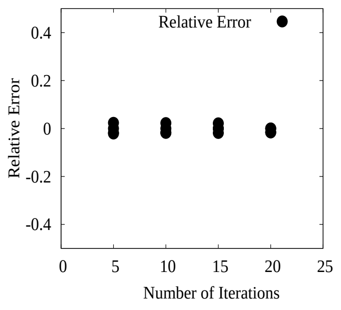
Fig. 2: Accuracy Of Estimations against varying input dataset size, number of nodes and iterations

the number of iterations), and on applications of different categories. From the estimated completion time T_{Est} and the recorded (i.e., observed) completion time T_{Rec}, we compute the relative error $RE = (T_{Est} - T_{Rec})/T_{Rec}$. The Figures 2(a), 2(b), and 2(c) illustrate the variations in the relative error RE, for the MovieLensALS, Wordcount, and PageRank applications, with increasing size s of the input dataset.

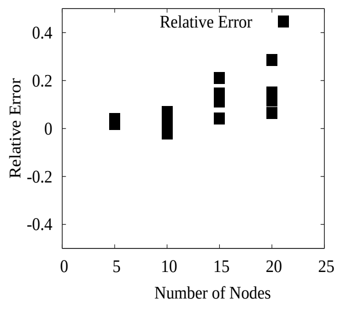
The figures 2(h), 2(d), 2(f), and 3(e) illustrate the variations in the relative error RE, for MovieLensALS, Wordcount, PageRank, and Logistic Regression, against varying cluster size n, in the stand-alone mode. The figures 2(e), 2(g), and 3(f) illustrate the variations in the relative error RE for the same applications, against varying number of iterations $iter$, in the stand-alone mode. Figures 2(i), 3(a), 3(c), and 3(g) illustrate the variations in the relative error RE for the same applications with varying n in the YARN mode. Figures 3(b), 3(d), and 3(h) illustrate the variations in the relative error RE for the same applications against varying $iter$, in YARN mode. We evaluate the OptEx model with the mean relative error metric [10] $\delta = \frac{\sum_{j=1}^{k} |T_{Est} - T_{Rec}|/T_{Rec}}{k}$, 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%.



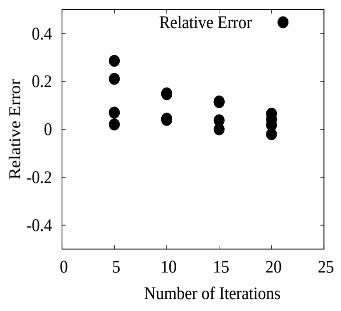
(a) Accuracy of Estimation for PageRank With Varying Number of Nodes in YARN mode



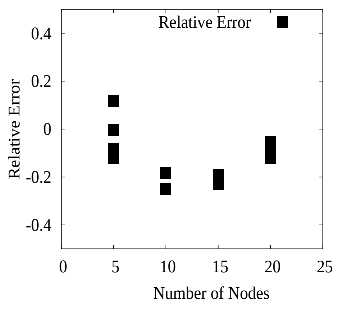
(b) Accuracy of Estimation for PageRank With Varying Number of Iterations in YARN mode



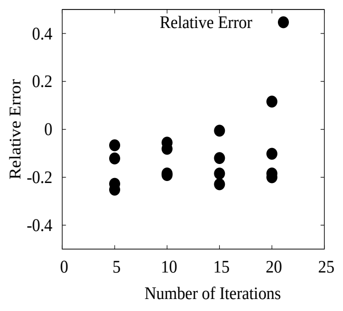
(c) Accuracy of Estimation for Logistic Regression With Varying Number of Nodes in YARN mode



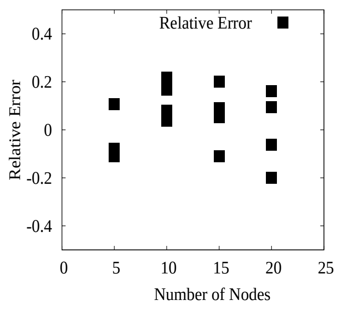
(d) Accuracy of Estimation for Logistic Regression With Varying Number of Iterations in YARN mode



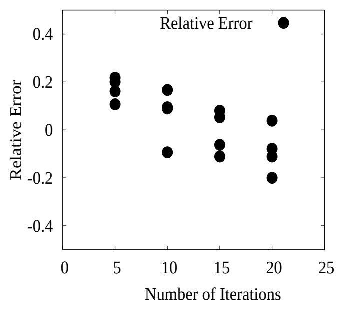
(e) Accuracy of Estimation for ALS With Varying Number of Nodes in stand-alone mode



(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

Application	Stand-alone	YARN
PageRank	0.019	0.012
Wordcount	0.036	0.087
LR	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 Sattistics

E. Analysis of the Results

The magnitude of the relative error RE for the experiments with Wordcount, PageRank, and Logistic Regression applications, representing the Streaming and GraphX categories (Section IV-A), in stand-alone mode, is strictly within 0-0.06, (Figures 2(a) through 3(b)), bounded by a 95% confidence interval of 0.056-0.062 (Table 3(i)), except for one observation in Figure 2(i). 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 Spark job execution time T_{Est} comprises two components X_1 and X_2, where $X_1 = T_{Init} + T_{prep}$ and $X_2 = n \times iter \times C + iter \times B/n + \frac{A \times s}{n}$ (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 (Section V), each phase varying differently with respect to the input variables n, $iter$, and s (Section V-A). 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 phases in X_2 encompass the execution of the job stages, comprising unit RDD tasks, on the worker nodes (Section V-B). 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., [5]. The job stages may get unpredictably delayed, i.e., can fail and get retried by the master repeatedly, due to various factors like momentary unavailability of required resources, delays in

TABLE IV: 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)
200	Standalone	ALS	3	103.2	107	5	88	93	6	89.67	92	7	88.84	95
240	YARN	ALS	2	240	250	3	228.44	235	4	211.5	215	4	240	237
350	Standalone	Wordcount	1	332.8	345									
800	YARN	Wordcount	1	794.8	821									
150	Standalone	ALS	3	100	97	4	99	97	4	140.5	143	6	103.22	113
200	YARN	ALS	3	174.22	215	4	181	185	5	178.56	186	5	198	195
330	Standalone	Wordcount	2	321.69	325									
790	YARN	Wordcount	2	783.69	781									
100	Standalone	ALS	4	79.5	75	6	76.11	78	6	89.67	91	7	88.84	94
160	YARN	ALS	4	150.5	157	5	159	158	6	160	155	7	159.84	158
325	Standalone	Wordcount	2	321.69	323									
785	YARN	Wordcount	2	783.69	782									
75	Standalone	ALS	5	68.52	65	7	75	67	8	71.88	72	9	73.1	71
140	YARN	ALS	5	139.52	138	7	139.92	135	9	138.07	133	9	139.1	135
320	Standalone	Wordcount	3	319.64	315									
783	YARN	Wordcount	3	781.64	754									
60	Standalone	ALS	7	58.96	56	10	58.76	57	11	61.1	55	12	62.56	58
121	YARN	ALS	22	121.05	118	31	121.02	115	39	120.97	112	42	121.1	113
319	Standalone	Wordcount	4	318.93	332									
781	YARN	Wordcount	4	780.93	772									

Category	Standalone				YARN			
	Standard Deviation	Variance	Mean	Confidence	Standard Deviation	Variance	Mean	Confidence
MLlib	0.373	0.139	1.106	0.183	0.702	0.493	2.2790	0.344
Spark Streaming	0.265	0.07	5.7	2.794	2.85	8.15	13.275	1.4
Spark SQL	1.96	3.85	32.63	0.96	3.263	10.65	48.88	0.087
GraphX	8.348	69.696	26.687	4.38	5.86	34.4	53	2.874

TABLE V: Confidence Interval of Estimation With Varying Choice of Representative Jobs for Each Job Category

allocation of resources by the master, communication delays among the workers, etc., [5]. 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 V-B). 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 V). 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 with increasing number of nodes (Figures 2(h), 2(d), 2(f), 2(i), 3(a), 3(c), 2(a), 3(e), and 3(g)). Worker nodes increasing in number 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 applications. Few user-facing applications, that work under an SLA deadline, will require more than 50 nodes [27, 28]. OptEx can provide estimations for typical SLA-driven user-facing applications with a relative error close to 0 (Figures 2 and 3). Applications that do not meet this criteria are batch processing applications, like bioinformatics, genomics, data analytics applications, etc., which typically do not work under a deadline [29].

For experiments run in YARN mode, 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 (Figures 3(a), and 3(c)). In YARN mode, the submitted jobs are additionally dependent on the YARN resource manager to allocate resources, and to execute the jobs on the worker [30]. Hence, the chances of unpredictable delays in the intermediate stages of a job are greater in YARN mode due to additional communication between the YARN resource manager and the Spark master [30]. Thus, the chances of observing randomness in the relative error is greater for applications run in the YARN mode, though the magnitude of the average error is 0.04. Further, the relative error, for YARN mode, is even closer to 0 for applications with number of iterations larger than 10, representing production level use cases [27, 28] (Figures 3(a), 3(c), and 3(g)).

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 [5]. The above delays result in deviations in the observed length of the job stages, comprising the component X_2, from the estimated completion time [5]. 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 [5]. This results in a decrease in the deviation in the observed length of the job stages comprising X_2 from the estimated lengths of the stages. 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). So, we believe that OptEx can provide more accurate estimations with typical production level use cases, which typically involve number of iterations larger than 10 [27, 28].

F. Optimal Scheduling and Project Planning using OptEx

TABLE VI: Optimal Scheduling With Estimated Optimal Cluster Size Under Given Cost Budget

Budget($)	Mode	App	n	T_{Est} (sec)	T_{Rec} (sec)
0.3	Standalone	ALS	53	49.17	48
0.8	YARN	ALS	58	120.15	115
1	Standalone	Wordcount	27	318.02	321
1.5	YARN	Wordcount	16	780.05	775
0.2	Standalone	ALS	35	49.4	50
0.5	YARN	ALS	36	120.01	119
0.8	Standalone	Wordcount	22	318.03	321
1.2	YARN	Wordcount	13	780.09	780
0.15	Standalone	ALS	26	49.72	50
0.4	YARN	ALS	29	120.58	117
0.6	Standalone	Wordcount	16	318.06	311
1	YARN	Wordcount	11	780.12	757
0.1	Standalone	ALS	17	50.69	52
0.3	YARN	ALS	21	121.1	125
0.4	Standalone	Wordcount	11	318.12	315
0.8	YARN	Wordcount	8	780.24	780
0.08	Standalone	ALS	13	51.89	50
0.2	YARN	ALS	14	122.49	120
0.1	Standalone	Wordcount	5	318.6	310
0.5	YARN	Wordcount	4	780.94	780

Again, being the first work in modeling the execution time of Spark jobs, OptEx has no prior results to compare directly with. The closest work is Elastisizer [6], which predicts optimal cluster composition for Hadoop, but does not address Spark. Moreover, Elastisizer over predicts, on an average by 20.1% and worst case 58.6% [6]. Since OptEx uses a closed-form to estimate the completion time, it does not suffer from over-prediction. Table IV demonstrates the effectiveness of the constrained optimization techniques of OptEx (Section VI) in designing optimal scheduling strategies. For each application (refer to the 3rd column of Table IV) running in Standalone or YARN mode (the 2nd column), Table IV gives the cost optimal cluster composition (the optimal cluster size n is given in the 4th, 7th, 10th and 13th columns against 5, 10, 15, and 20 iterations, respectively) for executing a given job under given SLA deadlines (the 1st column), while minimizing the cost of usage of the virtual machine instances. The 5th, 8th, 11th, and 14th columns of Table IV give the completion times T_{Est} estimated using OptEx, for 5, 10, 15, and 20 iterations, respectively. The 6th, 9th, 12th, and 15th columns of Table IV give the recorded completion times T_{Rec} with the estimated cluster composition.

Following [7], we propose a statistic $\mathcal{S}$ to measure 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 IV. $\mathcal{S}$ evaluates to approximately 98%, which proves that OptEx is, in fact, very effective for scheduling Spark jobs on the cloud, while minimizing the service usage cost.

Table VI demonstrates that OptEx can be used in project planning for optimal cluster provisioning under given budget, while optimizing job execution times. Table VI records the optimal cluster size (the 4th column) required to run a given application (the 3rd column) in Standalone or YARN mode (the 2nd column) estimated using Equation 8 under different values of the cost budget (the 1st column), while optimizing the completion times. The 5th column of Table VI 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.

G. Confidence Under Varying Choice of Representative Jobs

The job completion times depend on the job profile generated using representative jobs. With the assumptions regarding the choice of application category (Section VII-A), Table V 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 function over the integer variable n, i.e., $T_{Est} = f(n)$ (Section VI). Let μ and σ be the sample mean and variance of the job completion times for the experiments, under varying choice of representative job for each category, given in Table V. The standard deviation and variance (Table V) of the function represents the stability of the function f, under given variations in the choice of representative job. The expectation and variance (Table V) is computed using Taylor expansions [10], and can be expressed as follows: E$[f(n)] \approx (\mu) + \frac{f''(\mu)}{2}(\sigma)^2$, and Var$[T_{Est}] \approx (f'(E[n]))^2$. The confidence interval 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 $\mu + \alpha\sigma$ (Table V), the estimation is acceptable with 95% confidence level.

For 95% confidence level, $100\{1 - \alpha\}$ = 95, and α = 0.05. The estimated completion time for PageRank algorithm falls within the range 22.3 and 31 minutes with accuracy of 70% for standalone mode, and within 50 and 55.874 minutes with accuracy of 70% for YARN mode, with a confidence of 95%. The 95% confidence interval for MovieLensALS is 0.923 to 1.289 minutes with 80 % accuracy in standalone mode, and 1.935 to 2.623 minutes with 90 % accuracy in YARN mode. Similarly, the 95% confidence interval for Wordcount is 2.9 to 8.494 minutes with 100 % accuracy in standalone mode, and 11.875 to 14.675 minutes with 75 % accuracy in YARN mode. The 95% confidence interval for Logistic Regression, i.e., LR, is 31.67 to 33.59 seconds with accuracy of 70% in standalone mode, and 48.793 to 48.967 seconds with accuracy of 70% in YARN mode.

VIII. Theorems and Postulates

Let a_i denotes an application i, c is the total number of application categories, rep_j is the representative job for the application category C_j, n^i_{unit} is the number of unit RDD operations comprising the representative job rep_i, rdd_j denotes a unit RDD operation j, and $\mathcal{U}_{unit}$ is the universe of the unit RDD operations.

Theorem VIII.1. $\bigcup_{i=1}^{c} n_{unit}^{i} = \mathcal{U}_{unit}$

Proof: We categorize Spark applications to directly reflect the categorization of Apache Spark API library modules given in the Apache Spark website. The above categorization scheme implies that the union of above application categories comprise the union of all possible Spark library modules available openly. Each Spark library module is responsible for spawning a specific group of unit RDD operations. Hence, the union of above application categories comprises the universe of all possible unit RDD operations rdd_j that a Spark application can comprise. Further, we choose representative job rep_i for each category C_i in such a way that each representative job comprises all possible unit RDD operations rdd_j that applications in the respective application category C_i can possibly spawn. Hence, the union of all possible unit RDD operations spawned by representative jobs for the above application categories, i.e., $\bigcup_{i=1}^{c} n_{unit}^{i}$, comprises the universe of possible unit RDD operations, i.e., $\mathcal{U}_{unit}$.

Theorem VIII.2. $\bigcap_{i=1}^{c} n_{unit}^{i} \neq \emptyset$

Proof: It directly follows from Theorem VIII.1 that union of the representative jobs for the above application categories comprises the universe of all possible unit RDD operations. However, by design of the Spark API libraries, functions from a library module can call functions from library modules belonging to other libraries. Hence, an application from an application category can call a function from a library module that may belong to a different library. Thus, a representative job for a given application category may spawn a unit RDD operation belonging to a library that is used by applications in a different application category. Hence, the intersection of unit RDD operations spawned by different representative jobs, i.e., $\bigcap_{i=1}^{c} n_{unit}^{i}$, is a non empty set.

Theorem VIII.3. $\nexists rdd_j \in a_i \wedge a_i \in C_j \wedge rdd_j \notin rep_j$

Proof: The above theorem directly follows from Theorem VIII.1.

Theorem VIII.4. *$C = \sum_{t=1}^{m} c_t \times n_t \times T_{Est}$ is twice differential with respect to n.*

Proof: We rewrite the expression for $\mathcal{C}$ by replacing the term T_{Est} by the expression for Equation 8 as follows.

$$\mathcal{C} = \sum_{t=1}^{m} c_t \times n_t \times (T_{Init} + T_{prep} + n \times iter \times C + iter \times B/n + \frac{A \times s}{n}) \tag{10}$$

Differentiating the above expression for $\mathcal{C}$ with respect to n, we have

$$\frac{\mathrm{d}\mathcal{C}}{\mathrm{d}n} = iter \times C - iter \times B/n^2 - A \times s/n^2. \tag{11}$$

Since $n \geq 1$, $\mathcal{C}$ is differentiable with respect to n. Differentiating again with respect to n,

$$\frac{\mathrm{d}^2\mathcal{C}}{\mathrm{d}n^2} = iter \times B/n^3 + A \times s/n^3. \tag{12}$$

Since $n \geq 1$, $\frac{\mathrm{d}^2\mathcal{C}}{\mathrm{d}n^2} \geq 0$. Hence, $\mathcal{C}$ is twice differentiable with respect to n.

Theorem VIII.5. *$C = \sum_{t=1}^{m} c_t \times n_t \times T_{Est}$ is twice differential with respect to $iter$.*

Proof: Differentiating the above expression for $\mathcal{C}$ with respect to $iter$,

$$\frac{\mathrm{d}\mathcal{C}}{\mathrm{d}iter} = n \times C + B/n. \tag{13}$$

Since $iter \geq 1$, $\mathcal{C}$ is differentiable with respect to $iter$. Differentiating again with respect to $iter$,

$$\frac{\mathrm{d}^2\mathcal{C}}{\mathrm{d}iter^2} = 0. \tag{14}$$

Hence, $\mathcal{C}$ is twice differentiable with respect to $iter$.

Theorem VIII.6. *Minimize $\mathcal{C} = \sum_{t=1}^{m} c_t \times n_t \times T_{Est}$ is a convex function.*

Proof: It follows from Theorems VIII.4 and VIII.5 that Theorem VIII.6 is valid.

Theorem VIII.7. *Minimize $\mathcal{C} = \sum_{t=1}^{m} c_t \times n_t \times T_{Est}$, under $T_{Est} < SLO$ is a convex optimization problem.*

Proof: It follows from Theorem VIII.6 that Theorem VIII.7 is valid.

IX. Possible Extensions

In this section, we discuss how the OptEx model, which expresses completion time of Spark applications in terms of the underlying number of nodes, number of iterations, and size of the input dataset, can be applied to a larger group of parallel processing frameworks. In particular, we show how OptEx can be applied to model the execution of Hadoop MapReduce jobs, by appropriately modifying the expression for estimated total job completion time T_{Est}. In general, the estimated total job completion time for an application running on a parallel processing framework can be represented by the expression for estimated total job completion time $T_{Est} = T_{Init} + T_{brd} + T_{comp}$, replacing Spark-specific expressions for the terms T_{vs} and T_{comp} in Section V with expressions specific to the parallel processing framework under consideration. The terms T_{Init} and T_{brd} can be estimated from the job profile directly, by running representative jobs for each application category. Similar to our approach with Spark (refer to Section IV-A), the application categorization scheme is kept open to the discretion of the user. The term T_{comp} can be rewritten in the appropriate form as per the operating methodology of the framework under consideration. Essentially, T_{comp} is a function of the average execution time of unit task components comprising the given application. In case of Spark, the unit RDD operations comprise the unit tasks. For applying the OptEx model to a particular framework, the user needs to determine the unit task components that an application gets disintegrated into while running on the framework under consideration. The running times of the unit tasks comprising an application running on the framework can be obtained from the job profiles for the respective framework. Finally, the expression for T_{comp} can be rewritten in terms of the framework under the consideration, replacing the Spark-specific parameters with parameters specific to the framework under consideration.

A. *Extension of OptEx to Hadoop MapReduce Version 1*

In case of Hadoop MapReduce, the unit tasks are map, reduce, and shuffle tasks, that a given application gets partitioned into. The average execution time of a map, reduce, or shuffle task can be estimated from the job profile obtained using the same profiling techniques used in OptEx. In fact, Verma et al. [7] estimate the average, minimum, and maximum running times for the map, reduce, and shuffle tasks applying a technique that is identical to the job profiling approach in OptEx. OptEx can be extended to estimate execution time of a Hadoop MapReduce application by making changes to the expression for T_{Est} as follows. OptEx estimates the total job completion time, expressed using Equation 8. Unlike Verma et al., who only model job execution at the worker nodes, i.e., they estimate only T_{comp}, the job profiles in OptEx must estimate the lengths of the phases T_{init}, T_{prep}, and T_{vs}, as well. The values of T_{init} and T_{prep} are directly estimated from the job profile obtained by running benchmark Hadoop jobs, identical to the ones used by Verma et al. The term T_{comp} can be expressed using the equations for the average job completion time at the worker nodes, T_j^{avj}, given by Verma et al. [7], for Hadoop MapReduce version 1, as follows. Following [7], we write

$$T_{comp} = T_j^{avg} = (T_M^{up} + T_j^{low})/2, \tag{15}$$

where T_M^{up} is the upper bound for the running time of a unit map task, and T_j^{low} is the lower bound for the execution time at the worker node (i.e., the lower bound for T_{comp}). In [7],

$$T_M^{up} = (N_M^j - 1) * M_{avg}/S_M^j + M_{max}, \tag{16}$$

and

$$T_j^{low} = T_M^{low} + Sh_{avg}^1 + T_{Sh}^{low} + T_R^{low}, \tag{17}$$

where T_M^{low} is the lower bound for the running time of a unit map task, Sh_{avg}^1 is the average running time of a *first* shuffle task (i.e, “shuffle phase of the first reduce wave" obtained from the job profiles), T_{Sh}^{low} is the lower bound of the running time of a unit shuffle task, and T_R^{low} is the lower bound of running time of a unit reduce task. [7] further computes

$$T_M^{low} = N_M^j * M_{avg}/S_M^j \tag{18}$$

and

$$T_{Sh}^{low} = (N_R^j/S_R^j - 1) * Sh_{avg}^{typ}, \tag{19}$$

where N_M^j and N_R^j are the number of unit map and reduce tasks, M_{avg} is the average running time of unit map task (obtained from the job profiles), S_M^j and S_R^j are the number of map and reduce slots, Sh_{avg}^{typ} is average running time of a *typical* shuffle task (obtained from the job profiles).

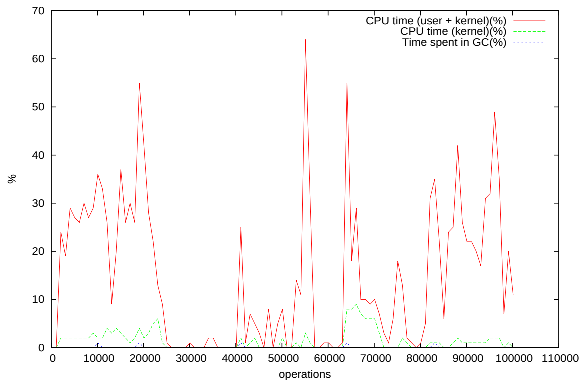


Fig. 4: CPU Time for ALS in stand-alone mode

B. *Extension To Hadoop MapReduce Version 2 or YARN*

However for Hadoop MapReduce version 2 or YARN, the equations in [7] need to be modified, because of the absence of the concept of map and reduce slots in MapReduce version 2. In YARN [30], the concurrent number of map and reduce tasks being spawned, i.e., , N_M^j and N_R^j, depend on the memory and cores being allocated to the tasks. Let $P, M, p,$ and m be the total number of cores per worker node, total memory per worker, number of cores allocated per unit task, and memory allocated per unit task, respectively. For YARN and Hadoop MapReduce version 2, N_M^j and N_R^j are computed as follows.

$$\begin{aligned} If(p \leq m) \\ N_M^j = P/p \\ N_R^j = P/p \\ Else \\ N_M^j = M/m \\ N_R^j = M/M \end{aligned} \tag{20}$$

Replacing these values in the expression for T_{comp} obtained above, we can apply the expression for T_{Est} to model completion time of applications using YARN and MapReduce version 2. Using the above expressions, we can rewrite the expression for the usage cost C in Section VI to represent the total usage cost for running an application j on Hadoop MapReduce version 2 or YARN. Given a completion time deadline for a job j, we can apply constraint minimization (same as the approach in Section VI) on the above expression for C to estimate the cost optimal cluster composition to complete the execution of j within an SLA deadline using virtual machine instances provide by cloud providers like Amazon, RackSpace, etc.

X. Performance Trade-offs in Scaling Out

Anderson et al. [31] have pointed out the fact that existing parallel computing systems need to maximize the resource usage at each node consisting the underlying cluster, in order to obtain performance improvement proportional to the applied scale out factor. While running the job profiles during our modelling process, we have analyzed the performance of the machine instances comprising the Spark cluster. In the process,

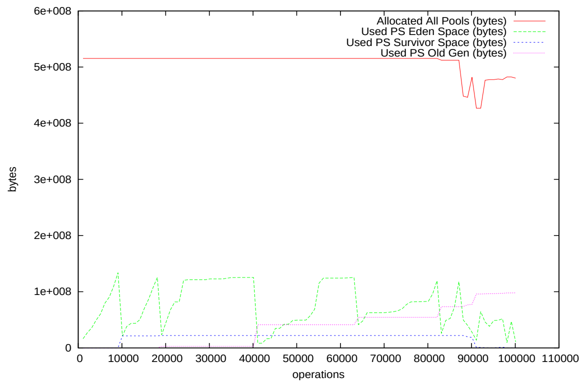


Fig. 5: The Heap Memory for ALS in stand-alone mode

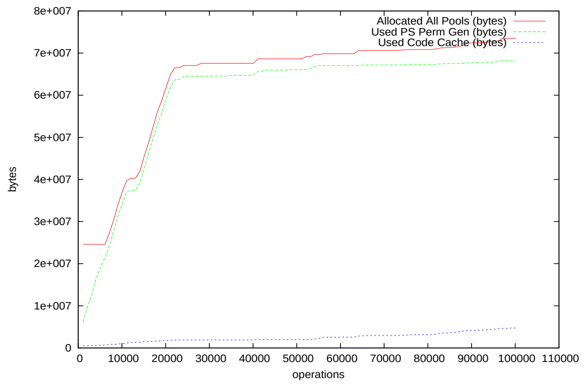


Fig. 6: The Non Heap Memory for ALS in stand-alone mode

we have identified certain issues regarding the performance utilization of the individual machines running Spark. We observed that correct configuration is essential to properly utilize the available capability of each machine, in terms of memory or CPU usage. We can see from the Figures 5, and 6 that the available memory is not utilized 100 % while running Spark jobs on a distributed cluster. Similarly Figure 4 show that the CPU of individual machines is not utilized properly throughout the running time of the Spark jobs. Hence, while scaling out a Spark cluster, we have to consider the utilization of individual machine instances in the cluster, to make proper usage of our resource. Before adding nodes to an existing cluster, we should attempt in making proper utilization of the resources available in the existing cluster, in order to optimize the cost and resources simultaneously.

XI. Related Work

The use of job profiling, performance modeling [37, 18, 34], and benchmarking techniques [32, 36, 32] for efficient load balancing [2, 3, 35], cost [4] and power optimization [33], have been attempted in quite a few cloud based systems. Singer et al. [4] presents a preliminary model that can be used to estimate virtual cloud instances required for replacing in-house hosting environments, while minimizing the cost. All of the above are standard resource allocation solutions rather than the minimal cluster composition prediction problem which we target. None of them attempt to provision a cost optimal cluster for cloud applications under given SLA deadlines.

There has been considerable amount of work [6, 7, 38, 39] on job scheduling and resource allocation on Hadoop, but none on Spark. Krevat et al. [40] presents empirical models for optimizing parallel execution of Hadoop MapReduce jobs, fully utilizing the given hardware resources, under a given input data size, filtering level, disk and network throughput. Verma et al. [7] presents the design of ARIA, a framework for optimal resource allocation for Hadoop MapReduce. Herodotou et al. [6] presents 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. ARIA [7] models Hadoop MapReduce job execution, and cannot be readily applied to other parallel processing frameworks because: 1) it uses Hadoop-specific configuration parameters like map-reduce slots, and 2) it constructs job profile with Hadoop-specific statistics, like running time of map, reduce, and shuffle phases. Elastisizer [6] uses expensive search based or black box based techniques, that require a huge database, for provisioning Hadoop clusters, and has inherent problems like over predicting. Though Spark [5] is fast surpassing Hadoop in popularity and usage, there has not been much work in modelling Spark jobs yet.

XII. 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 composition estimated using OptEx. 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.

References

[1] A. Inc, *Amazon Elastic Compute Cloud (Amazon EC2)*. http://aws.amazon.com/ec2/#pricing: Amazon Inc., 2008. [Online]. Available: http://aws.amazon.com/ec2/\#pricing

[2] S. Daniel and M. Kwon, "Prediction-based virtual instance migration for balanced workload in the cloud datacenters," 2011. [Online]. Available: http://scholarworks.rit.edu/article/985

[3] S. Imai, T. Chestna, and C. A. Varela, "Accurate resource prediction for hybrid iaas clouds using workload-tailored elastic compute units," in *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*, ser. UCC '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 171–178. [Online]. Available: http://dx.doi.org/10.1109/UCC.2013.40

[4] G. Singer, I. Livenson, M. Dumas, S. N. Srirama, and U. Norbisrath, "Towards a model for cloud computing cost estimation with reserved instances," *Proc. of 2nd Int. ICST Conf. on Cloud Computing, CloudComp 2010.*

[5] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228301

[6] H. Herodotou, F. Dong, and S. Babu, "No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics," in *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, ser. SOCC '11. New York, NY, USA: ACM, 2011, pp. 18:1–18:14. [Online]. Available: http://doi.acm.org/10.1145/2038916.2038934

[7] A. Verma, L. Cherkasova, and R. H. Campbell, "Aria: Automatic resource inference and allocation for mapreduce environments," in

Proceedings of the 8th ACM International Conference on Autonomic Computing, ser. ICAC '11. New York, NY, USA: ACM, 2011, pp. 235–244. [Online]. Available: http://doi.acm.org/10.1145/1998582.1998637

[8] Y. Chi, H. J. Moon, and H. Hacigümüş, "icbs: Incremental cost-based scheduling under piecewise linear slas," *Proc. VLDB Endow.*, vol. 4, no. 9, pp. 563–574, Jun. 2011. [Online]. Available: http://dx.doi.org/10.14778/2002938.2002942

[9] N. Zhang, J. Tatemura, J. M. Patel, and H. Hacigümüş, "Towards cost-effective storage provisioning for dbmss," *Proc. VLDB Endow.*, vol. 5, no. 4, pp. 274–285, Dec. 2011. [Online]. Available: http://dx.doi.org/10.14778/2095686.2095687

[10] J. Leader, *Numerical Analysis and Scientific Computation*. Pearson Addison Wesley, 2004. [Online]. Available: http://books.google.com/books?id=y-XEGAAACAAJ

[11] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 295–308. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972488

[12] D. G. Kendall, "Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain," *Ann. Math. Statist.*, vol. 24, no. 3, pp. 338–354, 09 1953. [Online]. Available: http://dx.doi.org/10.1214/aoms/1177728975

[13] L. Kleinrock, *Theory, Volume 1, Queueing Systems*. Wiley-Interscience, 1975.

[14] ——, "Time-shared systems: A theoretical treatment," *J. ACM*, vol. 14, no. 2, pp. 242–261, Apr. 1967. [Online]. Available: http://doi.acm.org/10.1145/321386.321388

[15] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10. New York, NY, USA: ACM, 2010, pp. 265–278. [Online]. Available: http://doi.acm.org/10.1145/1755913.1755940

[16] ——, "Job scheduling for multi-user mapreduce clusters," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-55, Apr 2009. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-55.html

[17] A. S. Foundation, "Apache spark libraries," 2015, [Online; accessed 25-June-2015]. [Online]. Available: https://spark.apache.org/

[18] P. Brebner and A. Liu, "Performance and cost assessment of cloud services," in *Service-Oriented Computing*, ser. Lecture Notes in Computer Science, E. Maximilien, G. Rossi, S.-T. Yuan, H. Ludwig, and M. Fantinato, Eds. Springer Berlin Heidelberg, 2011, vol. 6568, pp. 39–50. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-19394-1_5

[19] R. Zafarani and H. Liu, "Social computing data repository at ASU," 2009. [Online]. Available: http://socialcomputing.asu.edu

[20] *MovieLens dataset, http://www.grouplens.org/data/*, as of 2003. [Online]. Available: http://www.grouplens.org/data/

[21] AMPLAB, "Big data benchmark by amplab of uc berkeley," 2013, [Online; accessed 25-June-2015]. [Online]. Available: https://amplab.cs.berkeley.edu/benchmark/

[22] F. Zuleger, S. Gulwani, M. Sinn, and H. Veith, "Bound analysis of imperative programs with the size-change abstraction," in *Proceedings of the 18th International Conference on Static Analysis*, ser. SAS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 280–297. [Online]. Available: http://dl.acm.org/citation.cfm?id=2041552.2041574

[23] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[24] 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. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[25] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '09. New York, NY, USA: ACM, 2009, pp. 165–178. [Online]. Available: http://doi.acm.org/10.1145/1559845.1559865

[26] "The YourKit Java Profiler," http://www.yourkit.com, last 2008.

[27] A. S. Foundation, "Powered by spark," 2015, [Online; accessed 02-November-2015]. [Online]. Available: https://cwiki.apache.org/confluence/display/SPARK/Powered+By+Spark

[28] Cloudera, "Category archives: Use case," 2015, [Online; accessed 02-November-2015]. [Online]. Available: http://blog.cloudera.com/blog/category/use-case/

[29] S. Pappas, "9 super-cool uses for supercomputers," 2015, [Online; accessed 02-November-2015]. [Online]. Available: http://www.livescience.com/6392-9-super-cool-supercomputers.html

[30] A. S. Foundation. (2014) Apache hadoop nextgen mapreduce (yarn). [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[31] E. Anderson and J. Tucek, "Efficiency matters!" *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 1, pp. 40–45, Mar. 2010. [Online]. Available: http://doi.acm.org/10.1145/1740390.1740400

[32] J. Dejun, G. Pierre, and C.-H. Chi, "Resource provisioning of web applications in heterogeneous clouds," in *Proceedings of the 2Nd USENIX Conference on Web Application Development*, ser. WebApps'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 5–5. [Online]. Available: http://dl.acm.org/citation.cfm?id=2002168.2002173

[33] L. Ganesh, H. Weatherspoon, M. Balakrishnan, and K. Birman, "Optimizing power consumption in large scale storage systems," in *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, ser. HOTOS'07. Berkeley, CA, USA: USENIX Association, 2007, pp. 9:1–9:6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1361397.1361406

[34] G. Wang, A. R. Butt, H. Monti, and K. Gupta, "Towards synthesizing realistic workload traces for studying the hadoop ecosystem," in *Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, ser. MASCOTS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 400–408. [Online]. Available: http://dx.doi.org/10.1109/MASCOTS.2011.59

[35] S. K. Garg, S. K. Gopalaiyengar, and R. Buyya, "Sla-based resource provisioning for heterogeneous workloads in a virtualized cloud datacenter."

[36] S. Imai, T. Chestna, and C. A. Varela, "Accurate resource prediction for hybrid iaas clouds using workload-tailored elastic compute units."

[37] 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 *Proceedings of the 12th International Conference on Algorithms and Architectures for Parallel Processing - Volume Part I*, ser. ICA3PP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 414–427. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33078-0_30

[38] W. Zhang, S. Rajasekaran, T. Wood, and M. Zhu, "Mimp: Deadline and interference aware scheduling of hadoop virtual machines," *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, vol. 0, pp. 394–403, 2014.

[39] K. Kc and K. Anyanwu, "Scheduling hadoop jobs to meet deadlines," in *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, ser. CLOUDCOM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 388–392. [Online]. Available: http://dx.doi.org/10.1109/CloudCom.2010.97

[40] E. Krevat, T. Shiran, E. Anderson, J. Tucek, J. J. Wylie, and G. R. Ganger, "Applying performance models to understand data-intensive computing efficiency," DTIC Document, Tech. Rep., 2010.

[41] M. Zaharia, "Apache spark," https://github.com/apache/spark/tree/master/examples/src/main/java/org/apache/spark/examples, 2013.

[42] 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.

[43] 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. [Online]. Available: http://dx.doi.org/10.1016/S0169-7552(98)00110-X

XIII. Appendix

A. 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 [5, 24]. We use the following example applications, belonging to the above category, as benchmarks for analyzing performance of Spark jobs.

1) WordCount: The WordCount program [41], 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. 7: 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 7). 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. The collect method aggregates the occurrence values v from all tuples by the respective keys, and returns the resultant values as the final output.

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 [41] (see Figure 8) is a collaborative filtering based movie rating application, found in Spark's Machine Learning Library (MLlib) [42], 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 [5, 24]. 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 [5].

Fig. 8: 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.

3) PageRank: Another popular example for iterative computation intensive algorithm is PageRank [43], 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 [41] (refer to Figure 9) 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. 9: 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()
```

4) Logistic Regression: We choose the logistic regression program [41] (refer to Figure 10) from the MLlib library [42] 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. 10: 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
}
```