Received: 14 December 2019 | Revised: 14 March 2020 | Accepted: 21 April 2020

DOI: 10.1002/cpe.5823

WILEY

SPECIAL ISSUE PAPER

Mary, Hugo, and Hugo*: Learning to schedule distributed data-parallel processing jobs on shared clusters

Lauritz Thamsen[1] | Jossekin Beilharz[2] | Vinh Thuy Tran[1,3] | Sasho Nedelkoski[1] | Odej Kao[1]

[1]Complex and Distributed IT Systems, Technische Universität Berlin, Berlin, Germany

[2]Operating Systems and Middleware Group, Hasso Plattner Institute, University of Potsdam, Potsdam, Germany

[3]Thryve (mHealth Pioneers GmbH), Berlin, Germany

Correspondence
Lauritz Thamsen, Complex and Distributed IT Systems, Technische Universität Berlin, Germany.
Email: lauritz.thamsen@tu-berlin.de

Funding information
Bundesministerium für Bildung und Forschung (BMBF), Grant/Award Numbers: 01IS14013A and 01IS18025A (BBDC).

Summary

Distributed data-parallel processing systems like MapReduce, Spark, and Flink are popular for analyzing large datasets using cluster resources. Resource management systems like YARN or Mesos in turn allow multiple data-parallel processing jobs to share cluster resources in temporary containers. Often, the containers do not isolate resource usage to achieve high degrees of overall resource utilization despite overprovisioning and the often fluctuating utilization of specific jobs. However, some combinations of jobs utilize resources better and interfere less with each other when running on the same shared nodes than others. This article presents an approach for improving the resource utilization and job throughput when scheduling recurring distributed data-parallel processing jobs in shared clusters. The approach is based on reinforcement learning and a measure of co-location goodness to have cluster schedulers learn over time which jobs are best executed together on shared resources. We evaluated this approach over the last years with three prototype schedulers that build on each other: Mary, Hugo, and Hugo*. For the evaluation we used exemplary Flink and Spark jobs from different application domains and clusters of commodity nodes managed by YARN. The results of these experiments show that our approach can increase resource utilization and job throughput significantly.

KEYWORDS

cluster resource management, distributed data-parallel processing, job co-location, reinforcement learning, self-learning scheduler

This is an extended discussion of the works we published at the Big Data Congress 2017[1] (https://doi.org/10.1109/BigDataCongress.2017.28, © IEEE, 2017), in the STBD journal 4(1)[2] (https://doi.org/10.29268/stbd.2017.4.1.3), and the ParaMo workshop at Euro-Par 2019[3] (to appear, © Springer, 2019), presenting all our scheduler variants together in one article for the first time. In comparison to our previous publications, we also added a new motivation and a comparison of the related work. We further thoroughly revised the material we used for all sections of this article.

1 | INTRODUCTION

Distributed data-parallel processing systems such as MapReduce,[4] Spark,[5] and Flink[6] enable users to take advantage of clusters of machines for the analysis of large datasets. These systems have become popular tools for workloads that range from data aggregation and search to relational

queries, graph processing, and machine learning.[7-9] Jobs from these diverse domains stress different resources, while the resource demands typically also fluctuate significantly over the runtime of jobs.[2,10-12] Therefore, multiple jobs usually share cluster resources without isolation, so they can benefit from statistical multiplexing.[2,13,14] This is implemented by using resource management systems like YARN[15] and Mesos,[16] which allow users to reserve fractions of cluster resources via the notion of containers, in which users then run one or multiple jobs using the frameworks of their choice.

By default the resource management systems use simple scheduling methods such as round-robin, FIFO, greedy approaches, and other reservation-based methods such as dominant-resource fairness,[5,15-17] while low resource utilization remains a major problem in industry.[14,18,19] One reason for this is that it is hard to estimate the resource requirements of long-running distributed data processing jobs and users tend to overprovision significantly.[20-22] This has been addressed by many systems that allocate sets of resources automatically using models for the runtimes and horizontal scaling of jobs.[22-25] Yet, even with accurate estimates of resource requirements, the resource utilization of the jobs still fluctuates over their often considerable runtimes. That is, few jobs utilize resources well for the entirety of their runtimes, making it beneficial to co-locate multiple jobs onto shared resources, since the combined resource utilization of multiple co-located jobs often fluctuates less. Moreover, to further increase resource utilization, one often-applied strategy is to oversubscribe resources to a certain extent when scheduling jobs onto cluster resources.[26] However, since jobs differ considerably in which resources they stress and how much their resource utilization fluctuates, it can make a significant difference in terms of overall resource utilization and makespan which specific combinations of jobs share resources. Therefore, schedulers should actively co-locate those jobs that share resources efficiently. The benefits of such approaches have been demonstrated before with multiple schedulers that explicitly take combined resource utilization and interference among co-located workloads into account[11,18,27] or learn the impact of this indirectly,[28,29] taking advantage of the recurrence of a majority of jobs in industry workloads.[21,26]

In this article we present our approach to scheduling recurring distributed data-parallel processing jobs on shared cluster resources. To increase server utilization, we select jobs for execution from the queue of submitted jobs that stress different resources than the jobs already running on the nodes with available resources. For this, we take the resource utilization of and interference among co-located jobs into account. Furthermore, we use reinforcement learning to continuously learn which combinations of jobs should be promoted or prevented. In particular, we use the Gradient Bandits method for estimating the distribution of job co-location goodness. For this measure of co-location goodness we incorporate the CPU, disk, and network utilization as well as I/O wait. We evaluated this approach over the last years with three prototype implementations that build on each other. The first scheduler, which we call *Mary*, implements the reinforcement learning algorithm and measure of co-location goodness. The second scheduler, which we call *Hugo*, builds on the first scheduler and adds offline grouping of jobs to provide a scheduling mechanism that efficiently generalizes from specific monitored job combinations. The third scheduler, which we call *Hugo**, further builds on the second variant and implements bounded waiting, showing how additional scheduling requirements can be integrated. We implemented all our scheduler variants for YARN. We used exemplary Flink as well as Spark jobs from different application domains to test their performances in terms of cluster resource utilization, makespan, and waiting times.

Outline. The remainder of this article is structured as follows. Section 2 motivates our work with a problem analysis. Section 3 describes our scheduling approach. Section 4 presents the three scheduler variants implementing our approach and our experiments. Section 5 summarizes the related work. Section 6 concludes this paper.

2 | MOTIVATION

In this section we motivate co-locating specific combinations of distributed data-parallel processing jobs onto shared resources. For this, we first describe our experimental setup, before we present two experiments.

2.1 | Cluster setup

All the experiments presented in this article were performed on a commodity cluster of 40 homogeneous servers with CPU Intel Quad-Core 3.30 GHz (8 logical cores), 16 GB RAM, 3 TB hard drive (RAID0 - 1TB x 3), and 1 GB Ethernet. All servers run CentOS 7 (kernel version 3.10.0).

We used the distributed dataflow frameworks Apache Spark and Apache Flink and a set of exemplary jobs from different application domains. The applications and datasets have been selected to show utilization of different resources with runtimes of around 10 minutes. We used variants of the jobs we also used to evaluate our scheduler prototypes. The details of the Spark jobs are shown in Table 6. Of these, we used the jobs Logistic Regression (LR), SVM, PageRank (PR), and TPC-H Query 3 (TPC-3) for our initial experiments. The Flink jobs are described in Table 4 and we used K-Means, Connected Components (CC), and TPC-H Query 10 (TPC-10), yet altered configurations for the initial motivating experiments: $\text{K-Means}_{50}^{500}$ and $\text{K-Means}_{70}^{500}$ run, respectively, 50 and 70 iterations to cluster $1.25 \cdot 10^8$ points using 500 centers, while $\text{K-Means}_{4}^{1000}$ executes just four iterations but analyzes 10^9 points with 1000 centers. CC_3 and CC_8 run respectively three and eight iterations of the algorithm on the entire Twitter social graph dataset. TPC-10 processes 1 TB of generated data.

We conducted the experiments on clusters of resources managed by Hadoop YARN, reading input data from and writing output data to a co-located instance of HDFS. We used 1 core per container and allowed up to 8 containers per node. For the experiments with Apache Spark, we used Hadoop version 2.8.5 and Spark version 2.4.0. For the experiments with Apache Flink, we used Hadoop version 2.7.2 and Flink version 1.0.0. We report median runtimes out of five runs.

2.2 | Resource usage of exemplary jobs

The resource usage of distributed data-parallel processing jobs is diverse. Some jobs are CPU-bound in a particular cluster environment, while others are I/O-bound. Moreover, jobs often stress different resources over their often considerable runtimes.

To demonstrate the problem of underutilization of hardware resources for distributed dataflow jobs we ran Spark jobs on a Hadoop YARN cluster consisting of 10 servers. We monitored the resource usage of each worker node separately in addition to the accumulated average resource usage. Table 1 shows the averaged results, making apparent that the different jobs utilize different resources. For example, LR is high in CPU usage, while PR is low. At the same time, LR does not utilize the disk a lot, while TPC-3 on the other hand is a disk-intensive job. Furthermore, from the SD across nodes, we observe that the resource consumption also varies considerably between worker nodes. This already suggests that co-locating workloads onto shared nodes based on their resource usage can improve the performance.

At the same time, the resource usage also fluctuates considerably over the runtime of jobs, which can be seen in Table 2. Over the runtime of the jobs the resource utilization varies depending on the particular job. For example, we can see that the LR and the SVM jobs are negatively correlated. That is, both jobs might make a good combination for co-location in terms of their CPU usage, as the CPU might then be utilized more constantly over the runtime of the jobs.

For these reasons, jobs should be co-located, balancing resource usage over the different types of resources and also fluctuations across workers and time.

2.3 | Combined resource usage by job combinations

Since jobs differ in their resource consumption across multiple dimensions, the decision which jobs should be co-located onto shared resources is important. To demonstrate this, we ran a set of Flink jobs on a cluster of 21 YARN worker nodes. First, we ran the jobs in isolation, placing all the containers of each job onto 10 of the worker nodes. Then we ran the jobs again, but co-located the job with a second job, using the same number of containers, yet placing the containers of both jobs to share the resources of 20 worker nodes. The results of this experiment are shown in Table 3.

Not all co-locations reduce job runtimes equally, suggesting that some combinations are more effective in terms of balancing resource usage across the resource types, worker nodes, and job runtimes. For example, K-Means$^{500}_{50}$ profits from the co-location with K-Means$^{1000}_{4}$, resulting in a

TABLE 1 Average resource usage by node for multiple jobs (%)

	Logistic Regression		SVM		Page Rank		TPC-H Query 3	
Job	Mean	SD	Mean	SD	Mean	SD	Mean	SD
CPU	24.87	16.93	15.96	4.29	4.95	5.15	26.79	1.26
Memory	46.01	6.53	51.05	4.19	28.23	7.8	49.93	0.81
Disk	3.45	8.60	4.14	7.29	0.37	0.13	33.25	2.19
Net	4.05	4.44	5.86	4.36	1.16	1.31	5.84	6.57

TABLE 2 Average CPU resource usage by node for multiple jobs over time (%)

Job	0-200 (seconds)	200-400 (seconds)	400-600 (seconds)
LR	17.16	26.19	30.91
SVM	25.24	17.01	6.51
PR	9.34	2.82	2.52
TPC-3	36.11	20.38	25.17

TABLE 3 Co-location results when scheduling jobs with more or less similar resource usage onto shared cluster resources

Job Combinations	Duration (seconds) with placement		Duration Comparison (%)
	Isolated	Co-located	
$K\text{-Means}^{500}_{50}$	676	600	−11%
$K\text{-Means}^{1000}_{4}$	783	691	−12%
TPC-10	625	608	−3%
TPC-10	625	621	−1%
$K\text{-Means}^{500}_{70}$	919	893	−3%
CC_8	858	830	−3%
$K\text{-Means}^{1000}_{4}$	786	688	−12%
CC_3	617	585	−5%

reduced runtime of the jobs. Similar behavior is visible for $K\text{-Means}^{1000}_{4}$, when co-located with CC_3, yet surprisingly this effect is a lot less pronounced when co-locating $K\text{-Means}^{500}_{70}$ and CC_8. Co-locating disk-intensive jobs like the TPC-10 also seems to provide little benefit.

From these preliminary experiments, we conclude that co-locations matter significantly. They can increase the utilization of the resources and decrease the overall duration for the execution of the jobs. However, the benefit depends significantly on the specific combinations.

3 | APPROACH

This section describes our scheduling approach. The central idea is that it is often beneficial for resource utilization and throughput to co-locate jobs that stress different resources. While some jobs interfere with each other as they compete for the same resource, others use different resources and, therefore, complement each other when scheduled onto shared cluster resources. Therefore, to improve the utilization of cluster resources and, thereby, reduce job runtimes, beneficial co-locations should be promoted and others prevented.

Since it is generally difficult to anticipate resource requirements of distributed data-parallel processing jobs, recurring jobs provide an opportunity to learn how to schedule jobs efficiently onto shared resources. Information on the monitored co-location quality can then be used for training a self-learning algorithm. Such an algorithm, which is called reinforcement learning algorithm, is used for our approach. There is no need to tell the scheduler if a recurring job has changed or not. Changes in the behavior of recurring jobs, for example, due to modified input files or changed job parameters, are automatically taken into account over time as the co-location goodness measure changes.

The problem of scheduling jobs based on possible co-locations can be expressed as follows: *Given a set of running jobs, select which job should be scheduled next from the queue.* Our approach makes four simplifying assumptions: First, the servers of the cluster are homogeneous. Second, the resources of a server are shared fairly among the jobs scheduled to it. Third, the current scheduling decision is assumed to have an impact on future decisions. Fourth, individual jobs in the job queue are expected to be independent. Of these assumptions, the first two could be addressed by introducing weights. The third one is more difficult. Scheduling a job removes it from the queue. This can lead to less optimal co-locations later on, which could have been partially or entirely avoided by previously scheduling a less optimal job. Taking this into account would introduce significantly more complexity to the scheduling. The same goes for the fourth assumption.

Figure 1 depicts our approach of using reinforcement learning and monitoring of a measure of co-location goodness to learn over time which jobs best share cluster resources. Whenever resources become available, the scheduler selects a new job to run on the cluster from a queue of submitted jobs. For this, it uses a co-location preferences matrix that indicates which combinations of jobs have previously shared resources well. These co-location preferences are updated periodically based on monitoring data on the jobs that currently share resources on the cluster.

As the goal is to increase server utilization and thereby improve job throughput, a good co-location can be defined as one which utilizes the available resources best. A naive approach would therefore be to sum the usage of the different resources. However, this overlooks that co-locations which utilize resources well can also have negative effects, in particular interference. Consequently, the goodness measure needs to take this into account as well and be a trade-off between the combined resource usage of and interference between jobs. The particular algorithm and measure for co-location goodness we developed is presented in Section 4.1.

If we train a model for the goodness of job co-locations using monitoring data for job combinations that were actually executed on the cluster, the scheduler has no insights into jobs that have not been executed yet. Reinforcement learning is a method that trains itself over time by computing rewards for its decisions using a cost function. However, it is possible to make use of the similarity between a new job and jobs already seen on the cluster to improve the learning efficiency and scalability of the scheduler. For example, a new job can be profiled and matched to a group of similar

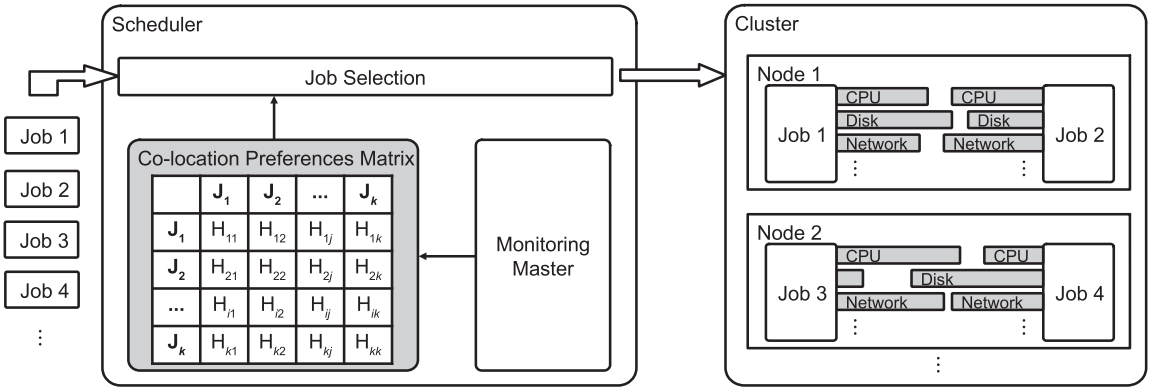

FIGURE 1 Scheduling cluster jobs based on learned co-location preferences, resulting in the co-location of complementing jobs

jobs based on its resource usage, while the reinforcement learning-based scheduling algorithm then makes decisions based on job groups instead of individual jobs. We explore this strategy in Section 4.2.

Typically, when scheduling cluster jobs in practice, there are more objectives and constraints than just improving the makespan by utilizing resources better. In practice, jobs should, for instance, not be starved of resources indefinitely, but scheduled eventually. More generally, resources should usually be assigned to users in a manner that is fair. We show how bounded waiting can be integrated into our scheduling approach in Section 4.3.

4 | CLUSTER SCHEDULING METHODS AND EXPERIMENTS

This section describes the three scheduler variants, in which we implemented our approach and different methods, each building upon and refining the previous variant. The first scheduler, called *Mary*, incorporates gradient bandits and a specific cost function. This scheduler was extended to base the scheduling decisions on groups of similar jobs in *Hugo*. A further extension *Hugo** uses bounded waiting. The prototype implementations are designed to work with the cluster resource manager YARN and can co-locate jobs of any framework that is supported by it, including MapReduce, Spark, and Flink.

4.1 | Mary: learning co-location decisions

Mary learns co-location decisions based on a measure of co-location goodness. The following sections describe this goodness measure, how to learn on it, and the evaluation of this approach.

4.1.1 | Rating the goodness of co-locations

As described in Section 3 the goal is to increase server utilization. Server utilization is defined as the utilization of the CPU, the disks, and the network interfaces, whereas interference between jobs is represented by the I/O wait metric, which indicates how long the CPU has to wait for I/O operations to complete. These metrics are grouped into two categories, I/O and CPU, and defined as follows.

I/O (disk and network): The disk and network usage are defined by the number of bytes read r (respectively received) and written w (respectively sent). The given values are normalized to the relative value with respect to previously defined maxima r_{max} and w_{max}, fixed by the physical limits of the hardware. Those two metrics are aggregated in a non-linear way with the function h, defined as:

$$h(r, w) := \tanh\left(\frac{r}{r_{max}} + \frac{w}{w_{max}}\right) .$$

CPU: The CPU usage u_{cpu} represents the percentage of used CPU. The CPU I/O wait metric u_{wait} is used as indicator that computation power is lost. So, it is used to weigh down the I/O utilization indicators h_{disk} and network h_{net} as they are only saturated. As a better co-location can certainly be found, this I/O weight function is exponentially decreasing.

Finally, the function f is used to favor high goodness. Put together, the goodness measure G is defined as:

$$G := f\left(u_{\text{cpu}} + (h(r_{\text{disk}}, w_{\text{disk}}) + h(r_{\text{net}}, w_{\text{net}})) \cdot l(u_{\text{wait}})\right), \tag{1}$$

where $f(x) := \exp(1 + x)$ and $l(x) := \exp(-5x)$.

4.1.2 | Learning the goodness of co-locations

To learn the goodness of co-locations a reinforcement learning algorithm is used. In the simplified case with one application running and one application to schedule, the problem can be reformulated as: *Given the running application, select the application with which the co-location is the best, that is, the one with the highest goodness measure, of the queue.*

To achieve this, we learn probability distributions $P(A_t = a|A = a')$ that encode the probability of scheduling application a at time t given that a' is already running. To obtain the distribution, we learn the preferences $H_t(a|a')$ and transform them into a probability distribution using the softmax function

$$P(A_t = a|A = a') := \pi_t(a|a') = \frac{e^{H_t(a|a')}}{\sum_{b \in S} e^{H_t(b|a')}},$$

where S is the set of all applications. We use the *Gradient Bandits method* to learn the preferences $H_t(a|a')$ for all applications $a, a' \in S$. When a job a is selected at time t while job a' is already running, the preference is updated by

$$H_{t+1}(a|a') = H_t(a|a') + \gamma(G_t - \overline{G}_t)(1 - \pi_t(a|a')), \tag{2}$$

while the preferences of the not selected jobs $\bar{a} \neq a$ are updated by

$$H_{t+1}(\bar{a}|a') = H_t(\bar{a}|a') - \gamma(G_t - \overline{G}_t)\pi_t(\bar{a}|a'), \tag{3}$$

where γ is the step-size parameter and $\overline{G}_t$ the average of all previous goodness measures, including the goodness measure G_t at time t as defined in Equation (1). The preferences are initialized with $H_0(a|a') := 0$ for all $a, a' \in S$ resulting in a uniform distribution. The preference updates occur periodically. At every update, the goodness measure is computed using the mean resource usage of the preceding time period.

This approach tracks a nonstationary problem: As the job can have different data or parameters for each run, it is possible that its resource usage changes and, therefore, the goodness of particular co-locations. The other advantage is that the choice of the scheduled application is done based on probabilities, so compared to a greedy algorithm, a extremely bad co-location would have considerably less chances to happen again.

4.1.3 | Evaluation

We evaluated the approach and implementation of Mary with two workloads scheduled with both our algorithms and a FIFO scheduler that does not change the queue order. The evaluation was done on 17 nodes of the cluster described in Section 2.1. Four different applications are used for the evaluation of the co-location decisions, as shown in Table 4. Only pairwise co-locations are permitted and each job uses a quarter of all containers, thus the four jobs can be scheduled simultaneously.

TABLE 4 Jobs used to evaluate Mary

Job	Data	Arguments	Abbreviation
K-Means (A)	$1.25{\cdot}10^8$ points with 500 centers	30 iterations	K-Means
Connected Components (B)	First quarter of the Twitter social graph	12 iterations	CC
TPC-H Query 10 (C)	500 GB of data generated with DBGEN	—	TPC-H^{10}_{500}
TPC-H Query 3 (D)	250 GB of data generated with DBGEN	—	TPC-H^{3}_{250}

Two different job queues are constructed for the experiments. Both queues consists of 48 jobs and alternate I/O-bound applications (TPC-H Query 10 and TPC-H Query 3) and CPU-bound applications (CC and K-Means). They are constructed as follows: $(m\times\text{TPC-H}_{10}\oplus m\times\text{K-Means}\oplus m\times\text{TPC-H}_{3}\oplus m\times\text{CC})\times n$. Based on this construction, the two queues are defined as:

Queue 1 with n=3 and m=4: C C C C A A A A D D D D B B B B C C C C A A A A D D D D B B B B C C C C A A A A D D D D B B B B

Queue 2 with n=4 and m=3: C C C A A A D D D B B B C C C A A A D D D B B B C C C A A A D D D B B B C C C A A A D D D B B B

Scheduling Strategies

Three different scheduling algorithms are used three times with both queues. They differ in their management of the Queue as well as initialization, and are defined as follows:

FIFO	The queue is unchanged for comparison purposes.
Resource-aware	The queue is modified according to the ideas described before to co-locate jobs based on their resource usage patterns.
Resource-aware with previous knowledge	Extension of the resource-aware scheduling approach by reusing the preferences learned from a previous run to limit exploration and favor the exploitation phase.

The same placement strategy is used for all scheduling algorithms: At first, three containers of a job are scheduled to empty nodes. Then, after containers have been placed on each node, nodes that still have available resources are picked at random to schedule another three containers of a job. This way, we used six of the available eight slots of each node, assigning the same number of containers that host the applications' worker processes to each node, while leaving space for the applications' master containers. The reason for this random selection strategy is that it is unlikely that an application would be co-located with only one other application in a real-world scenario. Furthermore, it speeds up the algorithm's learning as there are more different co-locations.

Results

Table 5 summarizes the performance of the three tested scheduling methods for both queues. It compares the median runtime of the baseline FIFO algorithms against the resource-aware scheduling (RA) and resource-aware scheduling with previous knowledge (RA*) methods. It shows that both resource-aware scheduling algorithms improve the execution time by 7% to 8% compared to the baseline for Queue 1. For Queue 2 there is no improvement of the execution time.

Monitoring data from this evaluation show that the resource-aware scheduler creates a more even distribution of the resource usage over time compared to the FIFO scheduler, where the resource usage alternates mainly between high CPU usage and high network usage. For Queue 1 this effect is more pronounced and results in better overall resource usage and thus shorter runtime. For Queue 2 the resource usage is already distributed quite evenly by the FIFO scheduler, as different jobs are scheduled together due to the job queue's characteristics. While the resource-aware scheduler shows a more even resource usage distribution also for Queue 1, there is not enough room left for improvement of the overall runtime. Details about this effect, including resource usage graphs for the different resources can be found in the original publication about Mary.[2]

In conclusion, the resource-aware algorithms avoid bad co-locations, which are indicated by fully saturated disks and, therefore, lost computation power. I/O-intensive applications are in such cases co-located with CPU-intensive applications. Therefore, the algorithm seems especially useful in cases where multiple similar applications are submitted in batches, while cases with a more mixed workload would benefit less from this approach.

4.2 | Hugo: Learning scheduling decisions on groups of jobs

Hugo extends Mary by aggregating jobs with similar resource usage patterns into groups and co-locating combinations of these groups. It uses offline clustering and online reinforcement learning for efficient learning, scalability, and adaptation to changes in workloads. Using groups provides increased scalability and induces less computation overhead for calculating the preference matrix, especially when the number of distinct jobs in a

TABLE 5 Effect of the scheduler on the duration of the experiments with the Queues 1 and 2

	Queue 1		Queue 2	
Scheduling	**Duration (minutes)**	**Change**	**Duration (minutes)**	**Change**
FIFO	148.5	—	137.0	—
RA	138.5	7%	137.0	0%
RA*	136.5	8%	136.5	0%

workload is high. In the following we describe the method of grouping jobs, learning good co-location of groups, and the evaluation of the scheduling decisions by Hugo. More details about the calculation of the scheduling probabilities as well as the prototype implementation can be found in the previous publication about Hugo.[3]

4.2.1 | Grouping jobs based on resource metrics

Grouping of the jobs is the key idea of Hugo to ensure scalability and improve the learning efficiency of the scheduler, since it reduces the size of the preference matrix to *k* groups. Using the existing job resource usage statistics for all previously executed jobs we group them into *k* groups. Depending on the clustering method, and the job profiling information, the groups can have different meaning. For example, there could be groups for jobs that predominantly stress the CPU, memory, disks, or network, while jobs of other groups could also exhibit mixed high usage of multiple resources.

4.2.2 | Learning to schedule job combinations

We learn the co-location goodness on the level of job groups. Therefore, our preference matrix contains a goodness measure for pairs of job groups. The co-location goodness measure assesses how specific combinations of job groups utilize resources, using metrics that capture the resource utilization and interference among co-located jobs. We use the same measure of co-location goodness and reinforcement learning algorithm as we used for Mary.

4.2.3 | Evaluation

We tested the Hugo scheduler with three experiments on a commodity cluster with various exemplary jobs. For this evaluation, 34 nodes of the cluster discussed in Section 2.1 were used. In the following we describe the test workload and the experiments with the respective results.

Test workload

For simulating a mixed data processing workload, nine Spark analytic jobs are used throughout the experiment. The jobs and their datasets used for benchmarking are shown in Table 6. For further reference each job is annotated by its own letter, while a number the annotated number denotes the job's group. We grouped the jobs into six distinct groups by their utilization of CPU, disk, and memory, including groups of mixed utilization and overall low resource utilization. The sizes of the input data are chosen so that the runtime of all jobs is similar and lasts approximately 10 minutes. The jobs were chosen such that they cover different application domains like machine learning (A, D, E, F), graph processing (B, C), relational queries (G), and text processing (H, I).

TABLE 6 Jobs used to evaluate Hugo

ob (job, group)	Data source	Data parameters
K-Means (A, 1)	KMeansDataGenerator[a]	100 000 000 points, 80 clusters
PageRank (B, 3)	Graph Challenge datasets[b]	46 656 000 edges, 2 174 640 vertices
Connected Components (C, 6)	Graph Challenge datasets[b]	38 880 000 edges, 1 812 200 vertices
Linear Regression (D, 1)	LinearDataGenerator[a]	90 000 000 samples, 20 features per sample
Logistic Regression (E, 2)	LogisticRegressionDataGenerator[a]	11 000 000 samples, 10 features per sample
SVM (F, 2)	SVMDataGenerator[a]	70 000 000 samples, 10 features per sample
TPC-H Query 3 (G, 4)	DBGEN[c]	100 GB generated DB
Sort (H, 5)	DBGEN[c]	143 999 787 records
Word Count (I, 1)	Wikipedia backup data[d]	53 GB text document

[a]From the org.apache.spark.mllib.util package
[b]Graph Challenge datasets provided by Amazon, https://graphchallenge.mit.edu/data-sets
[c]TPC-H benchmark suite, http://www.tpc.org/tpch/
[d]Wikipedia database dump, https://dumps.wikimedia.org

Results

We conducted three experiments. Each experiment shows how our scheduler performs in a different scenario in comparison to the baseline round-robin scheduler in terms of makespan and resource utilization. In the following we describe and motivate each of the experiments and present the results.

Learning Phase. The aim of this experiment is to gain insights into how Hugo compares to the baseline round-robin scheduler when there is no preference data available when the scheduler starts. That is, we start with an empty preference matrix. The algorithm then populates and updates the preference matrix, continuously evaluating the resource usage of pairs of jobs. To speed up the learning process, the job queue contains a job from each job group. The jobs are placed in a repeating pattern as follows: C B G A F H $\times$ 10.

Using the Hugo scheduler all queued jobs took 169 minutes 13 seconds to finish as opposed to the round-robin scheduler with 180 minutes 40 seconds, an improvement by 6.3%. This result indicates that using the Hugo scheduler is beneficial in comparison to the round-robin scheduler, even without any prior preference data and therefore while training, when the workload contains periodically recurring jobs.

Prior Preference Data. In the follow-up experiment, the preference matrix output from the first experiment is used as the input for the Hugo scheduler. However, in this experiment we exchange some of the jobs in the queue with jobs that did not appear in the queue of the previous experiment. This way, we want to evaluate how well the grouping of our scheduler generalizes to unseen jobs. The jobs are placed in a repeating pattern as follows: D E B C H G I $\times$ 5.

The Hugo scheduler again yields a faster running time: 114 minutes 49 seconds compared to the running time of the round-robin scheduler of 128 minutes 17 seconds. It thus produces a schedule that needs 10.5% less time to finish all jobs. Considering the job queue in this experiment has the same diversity of job groups as in the first experiment, the result suggests that the improvement is due to the prior knowledge of preferences between job groups. Also, the result indicates that it is possible to have beneficial co-location of new jobs on the basis of the calculated co-location goodness of previously executed similar jobs.

Randomized Queues. In this experiment, we included all nine dataflow jobs. The output preference matrix from the previous experiment is used as input preference data for Hugo in this experiment. With this experiment we want to gain insights into how our scheduler behaves with a more realistic queue as opposed to the manually created queues of the previous experiments. For this, we generated the following two randomized job queues:

Queue 3: C B B E A E E B I H H C B I H C E G F F A F C I G D A G I C G A F F D E G D A I D B H D H

Queue 4: E I A B C H G C A H E G C B F F G D B A C G D D H F I G C D B A F I F E I E E A H H B D I

Hugo reduces the overall processing time by 12.42% (Queue 3) and 11.14% (Queue 4) compared to the baseline round-robin scheduler. The resource utilization is increased for both queues and for all monitored resources between 0.5 and 6.2 percentage points.

4.3 | Hugo*: Integrating bounded waiting as additional scheduling constraint

In practice, there are other requirements for scheduling beyond resource utilization and throughput. Examples of these are fairness among users and priorities with particularly critical jobs. Hugo* is an extension of Hugo that integrates a mechanism to balance waiting times and prevent job starvation. As explained, when a job group is chosen as the next one to be scheduled, Hugo randomly chooses among the currently waiting jobs of that group with equal probabilities. To balance waiting times the choosing probabilities can be modified by the waiting times, in Hugo*, the probabilities of choosing a job a within the chosen job group G is subsequently calculated by

$$\pi_a = \frac{w_a}{\sum_{i \in G} w_i},$$

where w_i is the waiting time of job i.

The above usage of job waiting time only takes effect if the job group is eventually selected to schedule next. However, if the co-location goodness preference of the job group itself is low compared to most of the other job groups, the jobs in that group are still at risk of not getting selected. We therefore further define the global parameter *waiting limit* for the scheduler. When a job's waiting time reaches this limit, it is scheduled regardless of the co-location preferences. In case of multiple jobs with waiting times above the limit, one of them is chosen with probabilities according to the waiting time using the formula as above.

4.3.1 | Evaluation

We evaluate how effective Hugo* deals with job queues where jobs have different waiting times. The preference input data for this evaluation is the output of the second experiment in Section 4.2.3. The job queue used is the same randomized job queue 3 from the third experiment of the evaluation of Hugo. However, in contrast to these experiments, the whole queue is not known to the scheduler right away. Instead, jobs join the queue after every scheduling round. We look at two arrival patterns. With constant arrival rate (CAR), a single job is added to the queue after every scheduling round. With arbitrary arrival rate (AAR), 1 to 3 new jobs are added to the queue after every scheduling round. The exact amount of jobs added to the queue follows a probability distribution where the probability of adding 1, 2, or 3 jobs equals 60%, 20%, and 20%, respectively.

Results

Hugo is faster than the baseline (round-robin) for every constellation. With CAR the total duration is 145 minutes 44 seconds, which is a reduction by 10.73%. With AAR, Hugo* performs worse with a total duration of 154 minutes 29 seconds, reducing the duration by 5.37%. The reason being that a trade-off has to be made between job starvation and efficient job order.

Figure 2 shows how the waiting time is distributed with Hugo and Hugo*. For Hugo there is a significantly higher number of jobs with or exceeding the waiting time limit of 20. Hugo*, on the other hand, is able to successfully reduce the amount of jobs exceeding the waiting time limit. However, since Hugo* gives jobs that are waiting longer than the global limit the highest preference, the schedules exhibit less optimal co-locations, reflected in longer total running time.

5 | RELATED WORK

In this section we present related work on scheduling distributed data-parallel workloads based on their resource utilization and interference. Table 7 provides an overview, classifying the related work in terms of cluster infrastructure, target workload, main objective, and the used method.

Paragon[27] profiles incoming jobs and matches them with jobs that are similar with regard to the impact of different hardware and interference with co-located workloads. Paragon then assigns jobs to available resources using its classes of similar jobs and collaborative filtering, aiming to minimize interference and maximize resource utilization. In comparison, our approach specifically targets distributed data-parallel jobs and employs more resource utilization metrics for its co-location goodness.

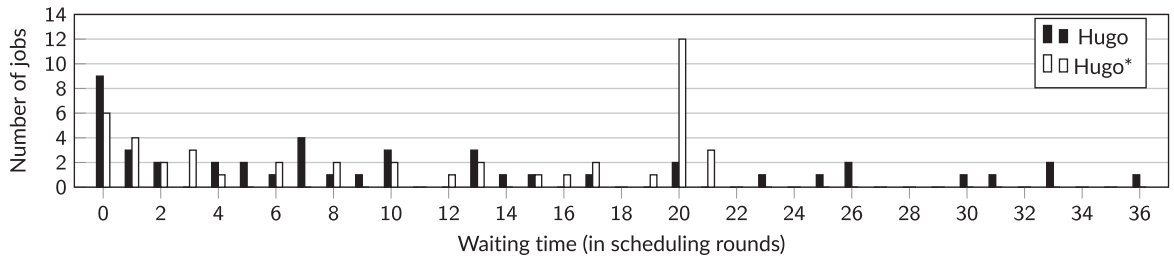

FIGURE 2 Comparison of job waiting times between the Hugo and the Hugo* scheduler

TABLE 7 Overview over the related work on scheduling cluster workloads based on resource utilization and interference

	Cluster	Workload	Objective	Method
Paragon[27]	Heterogeneous	Mixed with QoS targets	Meeting QoS targets	Collaborative filtering
Quasar[18]	Heterogeneous	Mixed with performance constraints	Resource utilization	Collaborative filtering
Gemini[11]	Homogeneous	Data-parallel	Makespan and fairness	Regression
DeepRM[28]	Homogeneous	Multi-resource	Average job speedup	Deep reinforcement learning
Decima[29]	Homogeneous	Multi-stage data-parallel	Makespan	Deep reinforcement learning
OASiS[17]	Homogeneous	Data-parallel ML	Overall job utility	Primal-dual optimization
SCARL[30]	Heterogeneous	Mixed	Average job speedup	Deep reinforcement learning

Quasar[18] uses classification to assess the impact of resources and interference with co-located workloads when scheduling jobs. It takes performance requirements of users into account, monitors job performance at runtime, and adjusts models and allocations dynamically. Quasar does assume full control over both resource allocation and assignment, while our scope is only scheduling of distributed data-parallel jobs.

Gemini[11] uses a model that captures the tradeoff between performance improvement and fairness loss for jobs scheduled in shared clusters. The model quantifies the complementarity in the resource demands of jobs and is trained on historic workload data. Gemini then decides automatically whether the fairness loss of a computed schedule is valid under a user's setting of required fairness in relation to Dominant Resource Fairness. In comparison, we use a reinforcement learning algorithm, groups of jobs, and only provide bounded waiting times as an option related to fairness.

DeepRM[28] is a scheduler that relies on deep reinforcement learning. The scheduler models the state of a cluster system, taking into account the already allocated resources along with resource profiles for the queued jobs. It then uses a neural network to obtain a probability distribution over all possible scheduling actions, using the rewards obtained after every action to update the parameters of the neural network. In comparison, DeepRM is a generic reinforcement learning framework for job scheduling, whereas our schedulers particularly target job co-location effects. Consequently, DeepRM might require more training effort.

Similar to DeepRM, Decima[29] also uses reinforcement learning along with a neural network for job scheduling. The system focuses on dataflow jobs that are described as directed acyclic graphs. Decima does not only perform scheduling, but also learns job parameters such as task parallelism. In comparison, we do not make assumptions about a job's structure beyond it being a data-parallel distributed job and also do not assume control over configuration options such as task parallelism.

OASiS[17] is an algorithm for scheduling asynchronous data-parallel machine learning workloads that follow the parameter server architecture. It computes an optimized job schedule upon arrival of each job, incorporating the projected resource availability and also performing admission control based on a job's utility compared to its resource consumption, using a primal-dual framework for the optimization problem. OASiS dynamically adjusts the level of data-parallelism both for workers and parameter servers with its schedules. In comparison to our schedulers, OASiS focuses explicitly on distributed model training, assumes control over job admission, and does not learn interference patterns.

SCARL[30] is a scheduler that assigns cluster jobs with diverse resource requirements onto heterogeneous cluster nodes. SCARL first selects jobs and then assigns the selected jobs to cluster resources. For this, it uses that uses deep reinforcement learning and attentive embedding to be able to deal with dynamically changing cluster environments. Aside of the differences in methods, our scheduler target data-parallel distributed processing workloads in homogeneous cluster environments.

6 | CONCLUSION

In this article we presented our approach for scheduling complementary distributed data-parallel processing jobs on shared cluster resources and our evaluation of this idea with the scheduler prototypes Mary, Hugo, and Hugo*. All our schedulers use a measure of co-location goodness and a reinforcement learning algorithm to learn over time which combinations of recurring jobs best utilize shared resources. They select among the queued jobs using learned job preferences, aiming to choose jobs to schedule next that complement the jobs already running on the shared cluster resources. While Mary learns which combinations of specific jobs yield high resource utilization and low interference, Hugo and Hugo* learn preferences not for single jobs but for groups of jobs with similar resource usage. This way, Hugo and Hugo* efficiently generalize from specific monitored job combinations and co-locate even new jobs effectively. In comparison to Hugo, Hugo* takes waiting times into account, giving priority to jobs that waited exceedingly long. We implemented the scheduler prototypes as job submission tools for YARN and used workloads consisting of exemplary Flink and Spark jobs to demonstrate that our scheduling approach can be used to reduce makespan by up to 12.5%, while resource utilization is increased and job waiting times can be bounded.

In the future, we want to investigate the effects of more fine-grained resource metrics when using an otherwise high-level approach that relies on job groups and reinforcement learning. We also plan to further explore the tradeoffs between the efficiency and fairness.

ACKNOWLEDGEMENTS

We acknowledge the co-authors of our previous publications on this topic,[1-3] especially Benjamin Rabier, Ilya Verbitskiy, and Florian Schmidt. This study was funded by German Ministry for Education and Research (BMBF) as BBDC (01IS14013A and 01IS18025A).

ORCID

Lauritz Thamsen https://orcid.org/0000-0003-3755-1503

REFERENCES

1. Thamsen L, Rabier B, Schmidt F, Renner T, Kao O. scheduling recurring distributed dataflow jobs based on resource utilization and interference. Big Data Congress'17; 2017; IEEE. https://doi.org/10.1109/BigDataCongress.2017.28
2. Thamsen L, Verbitskiy I, Rabier B, Kao O. Learning efficient co-locations for scheduling distributed dataflows in shared clusters. *Serv Trans Big Data*. 2018;4(1). https://doi.org/10.29268/stbd.2017.4.1.3.

3. Thamsen L, Verbitskiy I, Nedelkoski S, et al. Hugo: a cluster scheduler that efficiently learns to select complementary data-parallel jobs. ParaMo'19; 2019; Springer.
4. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. OSDI'04; 2004; USENIX.
5. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: cluster computing with working sets. the Hot Cloud'10; 2010; USENIX.
6. Carbone P, Katsifodimos A, Ewen S, Markl V, Haridi S, Tzoumas K. Apache Flink™ stream and batch processing in a single engine. *IEEE Data Eng Bull*. 2015;38(4).28–38.
7. Olston C, Reed B, Srivastava U, Kumar R, Tomkins A. Pig Latin: a not-so-foreign language for data processing. SIGMOD'08; 2008; ACM.
8. Gonzalez JE, Xin RS, Dave A, Crankshaw D, Franklin MJ, Stoica I. GraphX: graph processing in a distributed dataflow framework. OSDI'14; 2014; USENIX.
9. Meng X, Bradley J, Yavuz B, et al. MLlib: machine learning in apache spark. *J Mach Learn Res*. 2016;17(1).1235–1241.
10. Ousterhout K, Rasti R, Ratnasamy S, Shenker S, Chun BG. Making sense of performance in data analytics frameworks. NSDI'15; 2015; USENIX.
11. Niu Z, Tang S, He B. Gemini: an adaptive performance-fairness scheduler for data-intensive cluster computing. CloudCom'15; 2015; IEEE.
12. Renner T, Thamsen L, Kao O. Network-aware resource management for scalable data analytics frameworks. Big Data'15; 2015; IEEE.
13. Zaharia M, Borthakur D, Sen Sarma J, Elmeleegy K, Shenker S, Stoica I. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. EuroSys'10; 2010; ACM.
14. Reiss C, Tumanov A, Ganger GR, Katz RH, Kozuch MA. Heterogeneity and dynamicity of clouds at scale: google trace analysis. SoCC'12; 2012; ACM.
15. Vavilapalli VK, Murthy AC, Douglas C, et al. Apache hadoop YARN: yet another resource negotiator. SoCC'13; 2013; ACM.
16. Hindman B, Konwinski A, Zaharia M, et al. Mesos: a platform for fine-grained resource sharing in the data center. NSDI'11; 2011; USENIX.
17. Bao Y, Peng Y, Wu C, Li Z. Online job scheduling in distributed machine learning clusters. INFOCOM'18; 2018; IEEE.
18. Delimitrou C, Kozyrakis C. Quasar: resource-efficient and QoS-aware cluster management. ASPLOS'14; 2014; ACM.
19. Rasley J, Karanasos K, Kandula S, Fonseca R, Vojnovic M, Rao S. Efficient queue management for cluster scheduling. EuroSys'16; 2016; ACM.
20. Lama P, Zhou X. AROMA: automated resource allocation and configuration of mapreduce environment in the cloud. ICAC'12; 2012; ACM.
21. Jyothi SA, Curino C, Menache I, et al. Morpheus: towards automated SLOs for enterprise clusters. OSDI'16; 2016; USENIX.
22. Rajan K, Kakadia D, Curino C, Krishnan S. PerfOrator: eloquent performance models for resource optimization. SoCC'16; 2016; ACM.
23. Verma A, Cherkasova L, Campbell RH. ARIA: automatic resource inference and allocation for mapreduce environments. ICAC'11; 2011; ACM.
24. Venkataraman S, Yang Z, Franklin M, Recht B, Stoica I. Ernest: efficient performance prediction for large-scale advanced analytics. NSDI'16; 2016; USENIX.
25. Thamsen L, Verbitskiy I, Schmidt F, Renner T, Kao O. Selecting resources for distributed dataflow systems according to runtime targets. IPCCC'16; 2016; IEEE.
26. Ferguson AD, Bodik P, Kandula S, Boutin E, Fonseca R. Jockey: guaranteed job latency in data parallel clusters. EuroSys'12; 2012; ACM.
27. Delimitrou C, Kozyrakis C. Paragon: QoS-aware scheduling for heterogeneous datacenters. ASPLOS'13. 2013ACM.
28. Mao H, Alizadeh M, Menache I, Kandula S. Resource management with deep reinforcement learning. HotNets'16; 2016; ACM.
29. Mao H, Schwarzkopf M, Venkatakrishnan SB, Meng Z, Alizadeh M. Learning scheduling algorithms for data processing clusters. SIGCOMM'19; 2019; ACM.
30. Cheong M, Lee H, Yeom I, Woo H. SCARL: attentive reinforcement learning-based scheduling in a multi-resource heterogeneous cluster. *IEEE Access*. 2019;7.153432–153444.

How to cite this article: Thamsen L, Beilharz J, Tran VT, Nedelkoski S, Kao O. Mary, Hugo, and Hugo*: Learning to schedule distributed data-parallel processing jobs on shared clusters. *Concurrency Computat Pract Exper*. 2020;e5823. https://doi.org/10.1002/cpe.5823