Scheduling Recurring Distributed Dataflow Jobs Based on Resource Utilization and Interference

Lauritz Thamsen, Benjamin Rabier, Florian Schmidt, Thomas Renner, Odej Kao
Technische Universität Berlin, Germany
{firstname.lastname}@tu-berlin.de

Abstract—**Resource management systems like YARN or Mesos enable users to share cluster infrastructures by running analytics jobs in temporarily reserved containers. These containers are typically not isolated to achieve high degrees of overall resource utilizations despite the often fluctuating resource usage of single analytic jobs. However, some combinations of jobs utilize the resources better and interfere less with each others when running on the same nodes than others.**

This paper presents an approach for improving the resource utilization and job throughput when scheduling recurring data analysis jobs in shared cluster environments. Using a reinforcement learning algorithm, the scheduler continuously learns which jobs are best executed simultaneously on the cluster. Our evaluation of an implementation built on Hadoop YARN shows that this approach can increase resource utilization and decrease job runtimes. While interference between jobs can be avoided, co-locations of jobs with complementary resource usage are not yet always fully recognized. However, with a better measure of co-location goodness, our solution can be used to automatically adapt the scheduling to workloads with recurring batch jobs.

Index Terms—**Scalable Data Analytics, Distributed Dataflows, Resource Management, Cluster Scheduling, Job Interference**

I. Introduction

Cloud computing has emerged as a paradigm in which providers offer computing capabilities dynamically to an increasing number of users. Data centers have grown up to tens of thousands of nodes. These nodes are the largest fraction of the total cost of ownership for datacenters (50-70%) [1]. Thus, it is important to use these resources efficiently for continued scaling and cost-effectiveness. An important class of applications that run on such clusters and clouds is distributed data analytics, using frameworks like MapReduce [2], Spark [3], or Flink [4], which have become popular tools for scalable processing. However, studies have shown that these workloads often underutilize servers with resource utilization ranging between 10% and 50% [1], [5]–[8].

Different algorithms have been proposed to increase resource utilization through more fine-grained sharing of cluster resources. Some approaches model or profile resource needs of jobs more precisely for better resource allocations [7], [9], [10]. Other approaches schedule multiple jobs on nodes to increase server utilization while containing interference [11], [12]. While those approaches work well as many workloads are overprovisioned and, thus, many resources are unused, they ignore that different jobs can have complementary resource needs. Yet, by favoring co-locations of complementary jobs it is not only possible to avoid interference, but also to improve resource utilization. The execution of workloads could, therefore, be improved by changing the order in which jobs are executed in shared clusters. Furthermore, profiling, which has been used to measure interference in different approaches [7], [11], can be avoided as often more than 40% of jobs are recurring [10]. Therefore, interference and overall resource usage can be measured during the actual execution, improving the scheduling of subsequent job runs.

The approach presented in this paper is a scheduling method for recurring jobs that takes resource utilization and job interference into account. To increase server utilization, our scheduler changes the order of the job queue and selects jobs for execution that stress different resources than the jobs currently running on the nodes with available resources. For this, the scheduler uses a reinforcement learning algorithm to continuously learn which combinations of jobs should be promoted or prevented. In particular, we use the Gradient Bandits method, extended to a matrix of distributions, for estimating the distribution of co-location goodness [13]. Our metric for goodness takes CPU, disk, and network usage as well as I/O wait into account. We implemented our approach on top of Hadoop YARN [14]. The scheduler selects jobs for execution on the cluster based on our reinforcement learning approach. We evaluated our implementation on a cluster with 16 worker nodes and with two different workloads consisting of different Flink jobs.

Contributions. The contributions of this paper are:

- *Approach:* We designed a reinforcement learning solution for scheduling batch analytics workloads in shared cluster setups based on their resource usage and interference between jobs.
- *Implementation:* We implemented our solution practically using Hadoop YARN, supporting distributed dataflow systems that run on YARN such as Spark and Flink.
- *Evaluation:* We evaluated our implementation on a cluster with 16 worker nodes and two different workloads using Flink jobs.

Outline. The remainder of the paper is structured as follows. Section II presents the background. Section III presents our scheduling approach. Section IV presents the implementation of our prototype. Section V presents our evaluation. Section VI presents the related work, while Section VII concludes this paper.

II. Background

This section first describes distributed dataflow systems built to process large datasets. It then illustrates the design of resource management systems for such systems.

A. Distributed Dataflow Systems

Distributed dataflow systems process data through a Directed Acyclic Graph (DAG), where the nodes represent a computation task and edges the dataflow between these tasks. In more detail, tasks are configurable versions of pre-defined operators including Map and Reduce, which both execute user-defined functions (UDFs). Some distributed dataflow systems also provide specific variants of these two operators, like Filter and pre-defined aggregations, such as, sums. Operators like Join or Cross can be used to combine two dataflows. Figure 1 shows an exemplary distributed dataflow program with different data-parallel operator instances.

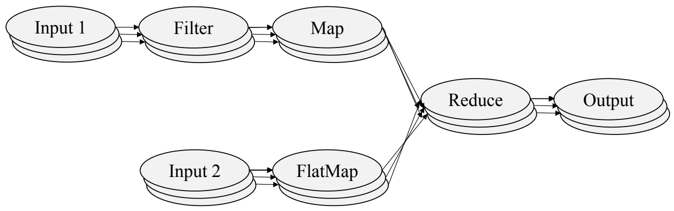

Fig. 1. A distributed dataflow job with different data-parallel operators.

Data-parallel task instances process partitions of the data. A partition can either be created by reading a fraction of the input data in from, for example, an underlying file system or can be received from a predecessor task in the dataflow graph. Sometimes it is necessary that the dataflow needs to shuffled. For example, operators for group-based aggregations or for joining two dataflows require all elements of the same group or identical join keys to be available at the same task instance. In this case, all elements with the same key need to be moved to the same task instance. Such data exchange patterns can yield high network traffic, since the task instances run on networked worker nodes. In addition, each worker provides execution slots, representing compute capabilities. Such a slot can either execute a task or a chain of tasks.

B. Resource Management Systems

Resource management systems regulate access to the resources of a cluster. Figure 2 shows an overview of such a resource-managed cluster. The system itself follows the master-slave pattern. The cluster manager often is a central master unit and arbitrator of all available resources. In addition, the cluster manager is responsible for scheduling workloads in containers on available resources. The slaves provide compute capabilities and, thus, host execution containers, in which the distributed analytics jobs are executed. Distributed dataflow programs in resource management systems are running on a per-job basis, whereby the worker processes run in containers scheduled by the cluster resource manage on to available slave compute nodes.

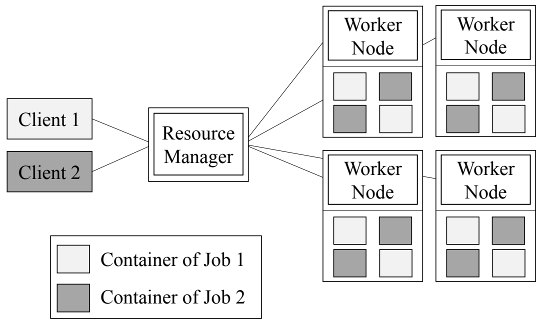

Fig. 2. Overview of a cluster running a resource management system.

Containers can be specified by the number of cores, memory and storage. In addition, they can provide different levels of resource isolation. Without strict resource isolation, the container specification are only used for scheduling purposes. Therefore, a worker process running in a container can use more or less resources than reserved. Jobs that run in containers co-located on the same node can consequently interfere with each other. The container placement and, therefore, which jobs are co-located is decided by the scheduler component of the resource manager. Often resource manager allow to use different schedulers focusing on different goals such as fairness, throughput, or data locality.

III. Approach

This section describes our approach to scheduling recurring jobs based on their resource utilization.

A. Overview

The key idea of our scheduling approach 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 the different resources complementary. Thus, to increase resource utilization in the cluster, good co-locations should be promoted and other ones prevented.

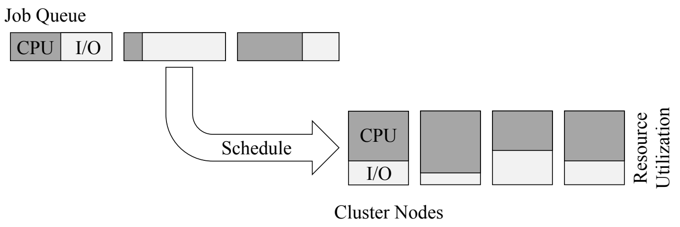

Fig. 3. Reordering of the jobs based on resource utilization.

Information on co-location quality can be used for training a self-learning algorithm. Such an algorithm, which is called reinforcement learning algorithm, is used for our scheduler.

To increase server utilization over time, the algorithm changes the queue of jobs, as presented in Figure 3. Based on currently scheduled jobs and the information learned, the scheduler chooses the most suitable of the n first jobs of the queue. These jobs are regarded as independent, so interactions between jobs is not considered here. Starvation is also not prevented currently but could be by weighting application with respect to the number of times they were skipped.

B. Rating the Goodness of Co-locations

As the goal is to increase server utilization, a good co-location can be defined as one which utilizes the available resources best. However, this overlooks that co-locations which utilize resource 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.

In this work, 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 and are defined as follows: I/O and CPU.

a) 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:

$$h := \tanh\left(\frac{r}{r_{max}} + \frac{w}{w_{max}}\right) \tag{1}$$

The function $\tanh$ is used to increase the robustness to errors on r_{max} and w_{max}.

b) CPU: The CPU usage u_{cpu} simply 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:

$$\begin{aligned} &g := f(u_{cpu} + (h_{disk} + h_{net}) * l(u_{wait})), \\ &where \quad f(x) := \exp(1 + x) \\ &and \quad l(x) := \exp(-5x) \end{aligned} \tag{2}$$

C. Learning the Goodness of Co-locations

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*. Before solving this problem, a simplified version is presented where only one job is scheduled. Furthermore, our solutions makes three simplifying assumptions:

- The servers are considered to be homogeneous.
- The resources of a server are fairly shared among the jobs scheduled on it. Thus, all jobs are considered equal for the goodness of a co-location.
- The current scheduling decision has no impact on future ones.

The first two simplifications could be addressed with weights. The last one is more complex: 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 appropriate job. Multiple reinforcement learning algorithms address this problem, but are more complex and, therefore, left as future work.

1) One Scheduled Application: In the simplified case with only one application scheduled, the problem can be reformulated as: *Select the application with which the co-location is the best, i.e. the one with the highest goodness measure, of the queue.*

One solution to this problem is the gradient bandits [13] where the algorithm learns a preference $H_t(a)$ for each job a in a set of all jobs S. The probability $\pi_t(a)$ of taking a specific application a at the time t is defined as follows:

$$\pi_t(a) = \frac{e^{H_t(a)}}{\sum_{b \in S} e^{H_t(b)}} \tag{3}$$

When a job α is selected at time t, the preference is then updated by:

$$\begin{aligned} H_{t+1}(\alpha_t) &= H_t(\alpha_t) + \gamma(G_t - \bar{G}_t)(1 - \pi_t(\alpha_t)) \\ H_{t+1}(a) &= H_t(a) - \gamma(G_t - \bar{G}_t)\pi_t(a) \quad \forall a \neq \alpha_t, \end{aligned} \tag{4}$$

where γ is the step-size parameter and $\bar{G}_t$ the average of all previous goodness measures, including G_t.

This approach tracks a non-stationary 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.

2) Multiple Scheduled Applications: In order to cope with multiple scheduled jobs, the previous algorithm needs to be extended. The preference is expressed as a matrix H where $H_{t,i}(a)$ is the preference of the job i of being in co-location with the job a at time t. Similarly, the probability $\pi_t(a)$ becomes $\pi_{t,i}(a)$ for the job i. The probability of taking an action a can be constructed as:

$$\begin{aligned} \Pi_t(a) &= \frac{\sum_{i \in C} \Pi_{t,i}(a)}{\sum_{j \in Q} \sum_{i \in C} \Pi_{t,i}(j)} \\ with \quad \Pi_{t,i}(a) &= \frac{\pi_{t,i}(a)}{\sum_{j \in Q} \pi_{t,i}(j)}, \end{aligned} \tag{5}$$

where C is the set of the running jobs on the cluster and Q the set of jobs that can be scheduled. $\Pi_{t,i}(a)$ is normalized with the set of jobs Q as it needs to represent the relative goodness of a compared to other possible choices for the job

i. Thus, $\Pi_t(a)$ is the normalized aggregation of the relative goodness for each scheduled job. To update the preferences, the Equation 6 is used:

$$\begin{aligned} H_{i,j} &:= \alpha(R_n - \bar{R}^i)(1 - \pi_i(j)) \\ &- \sum_{a \in \Omega_n \setminus \{i,j\}} \alpha(R_n - \bar{R}_i)\pi_i(a), \\ &\qquad\qquad \forall i, j \in \Omega_n, \end{aligned} \tag{6}$$

where $i \neq j$ and Ω_n is the set of running jobs of the node n. R_n denotes the reward for the node n and $\bar{R}^i$ is the average of all rewards for which job i was updated.

IV. Implementation

This section describes the implementation of the proposed job co-location scheduler for recurring jobs.

A. Overview

The implementation is designed to work with the cluster resource management system YARN. Thus, it can co-locate jobs of any framework that is supported by YARN, including systems like MapReduce, Spark, and Flink. InfluxDB[1] and Telegraf[2] are used to monitor and store detailed server utilization metrics of all nodes. InfluxDB is used as central database that stores time series monitoring data provided by Telegraf, which runs on each slave node. Our proposed scheduler communicates with YARN and InfluxDB. The monitoring data from InfluxDB is used as input for the reinforcement learning algorithm presented in the Section III-C. The output of the algorithm is used for selecting the next job from the queue of jobs. Afterwards, the selected job is submitted to YARN's ResourceManager for execution.

B. Scheduling Jobs

Our approach for scheduling a new application only takes effect when there is a queue of jobs to be executed. Otherwise, if there are sufficient resources for all jobs, jobs are directly scheduled and executed on the cluster. Before selecting a job from the queue of pending jobs with our reinforcement learning algorithm, presented in the Section III-C, we filter out the jobs that do not fit the available cluster resources. The next job from the queue is selected when a job finishes. Also, when a new job is submitted and this job's reservation fits the remaining available resources, it is selected as well. Afterwards, when a job is selected, it is submitted to YARN's ResourceManager for execution.

C. Updating the Preferences

The goodness of currently running co-located jobs, as presented in Section III-B, is measured periodically in our implementation. For any interval, all system metrics of nodes that run containers of at least two distinct jobs are queried from InfluxDB. Afterwards, the goodness per combination of co-located jobs is calculated with Equation 6 and used to update the matrix of pair-wise job preferences. The result then serves as input for any new job co-location decision.

[1]http://www.influxdata.com/time-series-platform/influxdb, accessed 2016-09-19

[2]http://www.influxdata.com/time-series-platform/telegraf, accessed 2016-09-19

V. Evaluation

We evaluated our approach and implementation with two workloads scheduled with both our algorithm as presented in the section III and a scheduler that does not change the queue order. First the cluster configuration is introduced, followed by a description of the job queues used in our evaluation. Then the results for both scheduling methods are compared and the respective resource usage analyzed.

A. Evaluation Setup

The experiments were done with YARN (2.7.2), HDFS (2.7.2), and Flink (1.0.0).

The cluster used in the experiments is constituted of 17 homogeneous nodes, connected through a single switch. The configuration of each of the nodes is as follows:

- Quadcore Intel Xeon CPU E3-1230 V2 3.30GHz
- 16 GB RAM
- 3TB RAID0 (3x1TB disks, linux software RAID)
- 1 GBit Ethernet NIC
- CentOS 7

The ResourceManager of YARN and the NameNode of HDFS run on a single node, while the remaining 16 nodes are used as worker nodes. Containers have all a fixed size of 1 CPU core and 1.5GB RAM. Leaving space for application master containers, a maximum of six worker containers are placed on each node.

Only pairwise co-locations are permitted as the number of jobs used is reduced. Each job uses a quarter of all containers available, thus four jobs can be scheduled simultaneously. Each job uses a different dataset. As at maximum four applications can run simultaneously, four different datasets are generated and used by the applications.

1) Applications: As basis four different jobs are used to built up the two different job queues.

- Kmeans clustering algorithm with $k = 50$ clusters and 30 iterations. The dataset is generated using the data generator for Kmeans provided with Flink and a standard deviation of 0.1, providing $1.25 * 10^8$ points.
- Connected components (CC) with 12 iterations. For this job, the first quarter of Twitter social graph from [15] is used.
- TPC-H Query 10 and 500 GB of generated data using DBGEN[3].
- TPC-H Query 3 and 250 GB of generated data using DBGEN[4].

Two different job queues are constructed for the experiments. Both queues consists of 48 jobs and alternate I/O-bound

[3]http://www.tpc.org/tpch/spec/tpch2.16.0v1.pdf, accessed 2016-08-25

[4]http://www.tpc.org/tpch/spec/tpch2.16.0v1.pdf, accessed 2016-08-25

applications (TPC-H Query 10 and TPC-H Query 3) and CPU-bound applications (CC and Kmeans). They are constructed as follows:

$$(m * \text{TPC-H}_{10} + m * \text{Kmeans} + m * \text{TPC-H}_3 + m * \text{CC}) * n \quad (7)$$

Based on this construction, the two queues are defined as:

- *Queue A:* with $n = 3$ and $m = 4$.
- *Queue B:* with $n = 4$ and $m = 3$.

2) 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 algorithm described in the section III.
- *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 triplets of containers are scheduled on empty nodes. When containers are placed on each node, nodes that still have available resources are picked randomly. The reason for this strategy is that it is unlikely that an application would be co-located with only one application in a real-world scenario. Furthermore, it increases the algorithms learning speed as it has more different types of co-locations.

B. Evaluation Results

First, execution time and resource usage is shown for both job queues. Then, the learned job preferences are visualized.

1) Execution Time and Resource Usage: Table I provides the median duration of the three tested scheduling methods for the Queue A. It shows that both resource-aware scheduling algorithms improve the execution time by 7-8% in contrast to the baseline FIFO scheduling approach.

TABLE I
DURATION OF THE EXPERIMENTS WITH THE QUEUE A

Scheduling method	Median duration (min)	Improvement (%)
FIFO	148.5	
Resource-aware	138.5	7%
Resource-aware with previous knowledge	136.5	8%

In contrast to the duration improvement for Queue A, there is no improvement detected for Queue B, as shown in Table II.

Figure 6 shows the resource utilization for Queue A and the FIFO as well as the resource-aware scheduler. The resource-aware scheduler creates a more even distribution of the resource usage over time compared to the FIFO scheduler, where the resource usage fluctuates more. From this, we conclude that the improvement of duration time for Queue A is due to the better resource usage. As the applications running have most of the time a similar resource usage, the resources have alternating phases of over- and under-utilization. With the resource-aware algorithm this is avoided.

TABLE II
DURATION OF THE EXPERIMENTS WITH THE QUEUE B

Scheduling method	Median duration (min)	Improvement (%)
FIFO	137	
Resource-aware	137	0%
Resource-aware with previous knowledge	136.5	0%

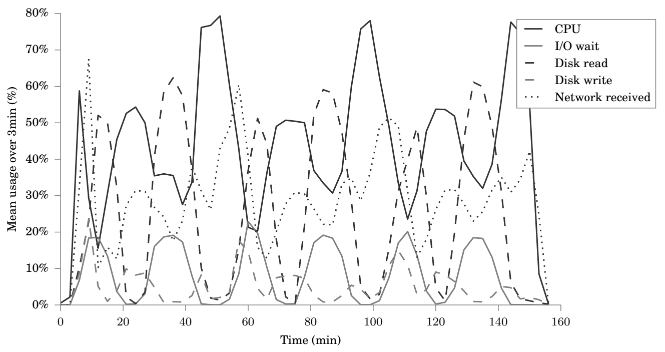

Fig. 4. FIFO

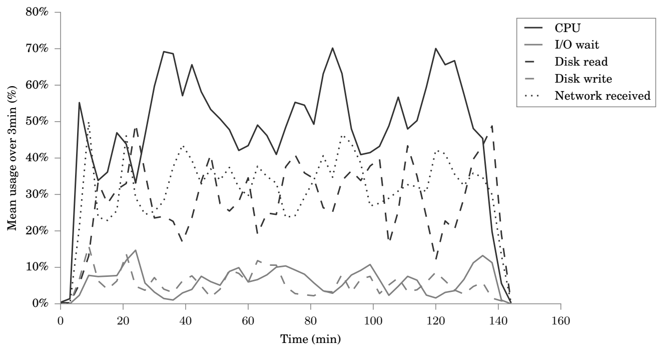

Fig. 5. Resource-aware

Fig. 6. Resource usage of the Queue A.

On the other hand, there is no improvement for Queue B the algorithm for the same reason. As Figure 9 shows for the FIFO scheduler, is the resource usage already quite even distributed. Thus there not much room left for improvements for optimizing the utilization.

Hence 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,

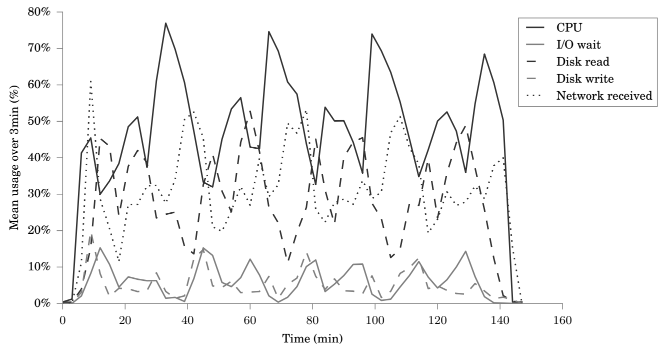

Fig. 7. FIFO

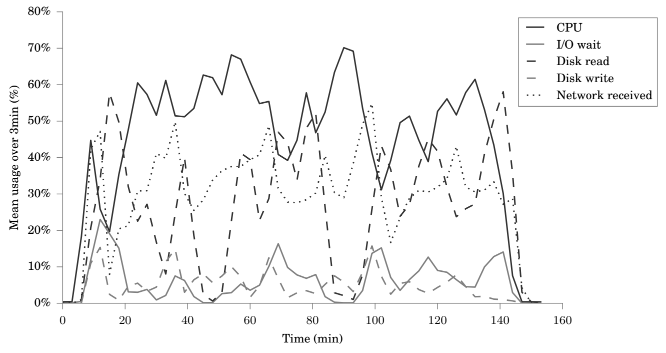

Fig. 8. Resource-aware

Fig. 9. Resource usage of the Queue B.

while cases with a more mixed workload would need a more refined goodness measure.

2) Application Preferences: The data learned from each experiment by the resource-aware scheduler is a preference for an application to be scheduled with another. It is expressed as a probability of one application to be chosen for a co-location (queued applications) when a specific application is scheduled (scheduled applications). Figure 10 illustrates this relationship between the already scheduled applications and the queued applications.

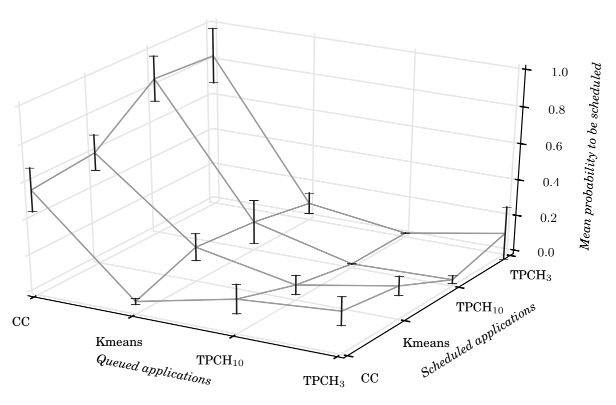

Fig. 10. Probability of queued applications to be selected to run with already scheduled applications.

The connected components application is the favorite application in the evaluation set. Every application would prefer it to be scheduled over the other ones. While it is unsurprising for TPC-H Query 3 or TPC-H Query 10, it is somewhat unexpected for Kmeans as both Kmeans and CC are relatively CPU-intensive. Consequently, one would expect that a co-location with one of the TPC-H applications would have a higher score for Kmeans due to the additional disk and network utilization. A similarly unexpected preference is the one of Kmeans with itself. It has approximately the same chances of being chosen as TPC-H Query 3. TPC-H Query 10 on the other hand is the least appreciated, particularly for both TPC-H applications where its probability of being chosen is less than 0.1%.

However, to achieve these results multiple simplification have been used in the evaluation which are not representative of a real-world case:

- *Small set of applications:* As the algorithms update pairwise co-location goodness measures, $O(n^2)$ values need to be updated. While this could be acceptable in some cases, it might have a too high computation complexity to give real-time updates in a large scale environment.
- *Similar data:* For every version of each type of applications, similar data is used. Yet, data characteristics such as key distributions can have a significant impact on the resource usage of a job.
- *Pairwise co-locations:* As a limited set of applications is used, only pairwise co-locations are made.

VI. Related Work

This section presents three categories of related work: frameworks for general-purpose distributed analytics, resource managers for such systems, and schedulers that take resource usage of analytics applications into account.

A. Distributed Analytics Frameworks

The presented systems are general frameworks for scalable data processing.

1) MapReduce: MapReduce [2] proposes a programming model and an execution model for scalable distributed execution. Programmers provide UDFs for the operations *Map* and *Reduce*, while the framework abstracts many of the difficulties of distributed computing, such as inter-machine communication and failure handling. Map specifies a transformation on each of the input key/value pairs, Reduce then aggregates tuples grouped by key. Between these two steps a shuffle step redistributes the tuples based on their key using a distributed file system like Google File System (GFS) [16], which applies replication for fault tolerance.

2) Spark: Spark [3] builds upon MapReduce, yet provides a more general programming model and in-memory execution. Spark's execution model is based on Resilient Distributed Datasets (RDDs) [17], which are distributed collections annotated with enough linage information to re-compute particular partitions efficiently in case of failures. RDDs can be cached to support interactive and iterative workloads. Keeping the data in memory can improve the performance considerably compared to frameworks such as MapReduce. Moreover, Spark also provides a more comprehensive set of data transformations compared to MapReduce. While Spark uses a batch engine at its core, a stream engine named Spark Streaming [18] runs on top of it by discretizing the input stream into micro-batches.

3) Flink: Flink [4] is another general-purpose dataflow system. Flink offers a similar programming model as Spark does, yet provides true streaming capabilities, effectively using a streaming engine for both batch and stream processing. Coming from the Stratosphere [19] research project, Flink furthermore applies techniques such as automatic query optimizations, managed memory, and native iterations for increased scalability as well as performance. Native support for iterations, for example, speeds up iterative processing by allowing cyclic dataflow graphs, which then only need to be scheduled and deployed once [20].

B. Resource Management Systems

The systems in this section manage clusters and allocate resources to applications with respect to a scheduling policy.

1) YARN: YARN [14] is a centralized system which allocates resources to applications. Upon admission, if an application is accepted, a container is allocated to host the *ApplicationMaster*. This framework-specific entity handles all communications with YARN and negotiates resources. Once a resource request is made, YARN attempts to satisfy the request according to availability and the scheduling policy by launching the requested containers.

2) Mesos: Mesos [21] is comparable to YARN, yet uses an indirect two-level approach for scheduling. Instead of being asked for resources, Mesos makes resource offers to application-specific schedulers. Those can either accept or wait for a better offer, possibly taking into account framework- and job-specific characteristics such as the locations of input files. When accepted Mesos launches the provided application with the offered resources. Fair scheduling and priorities are enforced by controlling the offers. Hence high priority applications will be proposed the most resources. To avoid starvation, a minimum offer can be specified. Concurrency control is pessimistic. That is, resources are only offered to one scheduler at a time until the offer times out.

3) Omega: Omega [22] uses an approach based on optimistic concurrency control. Rather than making resource offers, every scheduler has a copy of the current state of cluster resources. A master copy is held by Omega. Conflicts are handled through atomic commits. Thus, if two schedulers attempt to allocate the same resource, only one will succeed. The other one will have to re-run its scheduling algorithm. As multiple schedulers can work independently, it is possible to obtain better performance and scalability.

C. Resource Usage-aware Schedulers

This section presents different approaches for incorporating the resource usage of applications into scheduling decisions. They differ from our approach as they try either to prevent interference or to confine it, yet do not attempt to find co-locations that provide high overall resource utilization. They also include low-latency user-facing applications in addition to batch analytics. Furthermore, our solution does not use any sort of dedicated profiling and instead learns the behavior of recurring applications over time.

1) Quasar: Quasar [7], which is built on top of Paragon [23], uses fast classification techniques to classify applications with respect to different server configurations and sources of interference. An unknown application is first profiled on a few servers and for a short period of time. Then collaborative filtering techniques are used, in combination with offline characterizations and matching to previously scheduled ones, to classify the new application. The result is a set of estimations of the application's performance with regard to different resource allocations as well as co-locations with other workloads.

2) Bubble-flux: Bubble-flux [11] measures the effect of memory pressure on latency critical applications to predict interference. Upon submission of a known best-effort application, a dynamic "bubble" is generated over a short time to find the limit of admissible memory pressure on each node. As the load varies, batch applications can be periodically switched off for a small period of time to reduce their interference with latency-critical applications, so these have an acceptable mean latency. The same method is used to reduce the impact of the dynamic "bubble". The pressure of new applications is measured by gradually decreasing the period of the off phase.

3) Heracles: Heracles [12] guarantees the resources necessary for latency constraints to user-facing applications, while using surplus resources for co-located batch tasks. Four different isolation mechanisms are used to mitigate interference as necessary: partitioning of the last-level cache as well as the CPU cores, distribution of the power among cores to adapt their frequency, and network bandwidth limits. Heracles needs an offline profile of the applications DRAM bandwidth usage as no accurate enough mechanism has been found to measure it online. Heracles then continuously monitors whether the latency critical application fulfills its objective. If this is the case, best-effort tasks are allowed to grow if there is enough slack. Otherwise they need to release resources.

VII. Conclusion

This paper presented an approach for scheduling distributed dataflow jobs based on their resource usage to improve resource utilization and throughput. Our approach does not profile jobs in isolation, but instead uses a reinforcement learning algorithm to capture how well different combinations of jobs utilize resources when executed co-located. By extending the multi-armed bandit problem to a matrix of distributions, the algorithm effectively learns how good pairwise job co-locations are. The measure of co-location goodness we used takes into account both how well the different resources of a node are utilized and how much jobs interfere with each other. For resource usage, we considered CPU, disk, and network.

For interference, we focussed on I/O wait. We implemented our approach on top of YARN. When a new job is scheduled, the learning algorithm is requested and chooses a job from the scheduling queue based on the currently running jobs. We evaluated our solution on a cluster with 16 worker nodes and with two different workloads, using four different Flink jobs. Our results show a clear improvement for the first workload, in which longer sequences of jobs with similar resource usage are submitted to the YARN cluster. The resulting resource usage is more even over time and the execution time of the entire queue was shortened by around 8%. There was no change in runtime for the second workload, however, in which a more balanced mix of jobs is submitted to begin with. While this suggests that jobs might need to be co-located on a finer granularity, this also shows that in case of already more balanced workloads our approach at least has no negative effect.

In the future, we want to improve learning by taking similarity of jobs into account, so less job combinations have to be run co-located before the scheduler can make effective decisions. Furthermore, we want to improve the goodness measure for co-location by taking more interference sources into account, including, for example, cache metrics.

Acknowledgments

This work has been supported through grants by the German Science Foundation (DFG) as FOR 1306 Stratosphere and by the German Ministry for Education and Research (BMBF) as Berlin Big Data Center BBDC (funding mark 01IS14013A).

References

[1] L. A. Barroso and U. Hölzle, "The Case for Energy-Proportional Computing," *Computer*, vol. 40, no. 12, pp. 33–37, December 2007.

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation*, ser. OSDI'04. USENIX Association, January 2004, pp. 10–10.

[3] 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. USENIX Association, June 2010, pp. 10–10.

[4] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and Batch Processing in a Single Engine," *IEEE Data Engineering Bulletin*, vol. 38, no. 4, pp. 28–38, July 2015.

[5] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis," in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC '12. ACM, October 2012, pp. 7:1–7:13.

[6] M. Carvalho, W. Cirne, F. Brasileiro, and J. Wilkes, "Long-term SLOs for Reclaimed Cloud Computing Resources," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. ACM, October 2014, pp. 20:1–20:13.

[7] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware Cluster Management," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. ACM, March 2014, pp. 127–144.

[8] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale Cluster Management at Google with Borg," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. ACM, April 2015, pp. 18:1–18:17.

[9] 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. ACM, June 2011, pp. 235–244.

[10] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca, "Jockey: Guaranteed Job Latency in Data Parallel Clusters," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12. ACM, April 2012, pp. 99–112.

[11] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. ACM, June 2013, pp. 607–618.

[12] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving Resource Efficiency at Scale," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. ACM, May 2015, pp. 450–462.

[13] M. N. Katehakis and J. Arthur F. Veinott, "The Multi-Armed Bandit Problem: Decomposition and Computation," *Mathematics of Operations Research*, vol. 12, no. 2, pp. 262–268, May 1987.

[14] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet Another Resource Negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. ACM, September 2013, pp. 5:1–5:16.

[15] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a Social Network or a News Media?" in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10. ACM, April 2010, pp. 591–600.

[16] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. ACM, October 2003, pp. 29–43.

[17] 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. USENIX Association, April 2012, pp. 2–2.

[18] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized Streams: Fault-tolerant Streaming Computation at Scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. ACM, October 2013, pp. 423–438.

[19] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke, "The Stratosphere Platform for Big Data Analytics," *The VLDB Journal*, vol. 23, no. 6, pp. 939–964, December 2014.

[20] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl, "Spinning Fast Iterative Data Flows," *Proc. VLDB Endow.*, vol. 5, no. 11, pp. 1268–1279, July 2012.

[21] 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. USENIX Association, March 2011, pp. 295–308.

[22] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, Scalable Schedulers for Large Compute Clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. ACM, April 2013, pp. 351–364.

[23] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware Scheduling for Heterogeneous Datacenters," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. ACM, March 2013, pp. 77–88.