# Learning Efficient Co-locations for Scheduling Distributed Dataflows in Shared Clusters

Lauritz Thamsen, Ilya Verbitskiy, Benjamin Rabier*, and Odej Kao
Technische Universität Berlin
{firstname.lastname}@tu-berlin.de

## Abstract

Resource management systems like YARN or Mesos allow sharing cluster resources by running data-parallel processing jobs in temporarily reserved containers. Containers, in this context, are logical leases of resources as, for instance, a number of cores and main memory, allocated on a particular node. Typically, containers are used without resource isolation to achieve high degrees of overall resource utilization despite the often fluctuating resource usage of single analytic jobs. However, some combinations of jobs utilize the resources better and interfere less with each other when running on the same nodes than others.

This paper presents an approach for improving the resource utilization and job throughput when scheduling recurring distributed data-parallel processing jobs in shared cluster environments. Using a reinforcement learning algorithm, the scheduler continuously learns which jobs are best executed simultaneously on the cluster. We evaluated a prototype implementation of our approach with Hadoop YARN, exemplary Flink jobs from different application domains, and a cluster of commodity nodes. Even though the measure we use to assess the goodness of schedules can still be improved, the results of our evaluation show that our approach increases resource utilization and job throughput.

This is an extended work of Thamsen, Rabier, Schmidt, Renner, & Kao, © 2017 IEEE, published in the *Proceedings of the 6th 2017 IEEE International Congress on Big Data (BigData Congress 2017)*.

**Keywords:** Scalable Data Analytics, Distributed Dataflows, Resource Management, Cluster Scheduling

## 1. Introduction

Data centers have grown to tens of thousands of nodes. These nodes are the largest fraction of the total cost of ownership for datacenters (Barroso & Hölzle, 2007). Therefore, it is important to use these resources efficiently for cost-effectiveness and continued scaling. An important class of applications that runs on such clusters and clouds is distributed data-parallel processing, using distributed dataflow frameworks like MapReduce (Dean & Ghemawat, 2004), Spark (Zaharia et al., 2010), and Flink (Carbone et al., 2015). However, studies have shown that these workloads often underutilize servers with resource utilizations ranging between 10% and 50% (Barroso & Hölzle, 2007; Reiss et al., 2012; Carvalho et al., 2014; Delimitrou & Kozyrakis, 2014; Verma et al., 2015).

Different approaches have been proposed to increase resource utilization through more fine-grained sharing of cluster resources. Some techniques model or profile resource needs of jobs more precisely for better resource allocations (Verma et al., 2011; Ferguson et al., 2012; Delimitrou & Kozyrakis, 2014). Other approaches attempt to contain interference when scheduling multiple jobs on nodes to increase server utilization (Yang et al., 2013; Lo et al., 2015). While those approaches work well as many workloads are in fact overprovisioned and, thus, resources are unused, they ignore that different jobs can have complementary resource needs. Yet, by scheduling jobs with such complementary resource needs together, it is not only possible to reduce interference, but also to improve resource utilization. The execu-

*Work done while at Technische Universität Berlin, now at Nokia Digital Health in Paris, France.

tion of workloads could, therefore, be improved by changing the order in which jobs are executed in shared clusters. Since as many as over 60% of the jobs running on production clusters are reported to be periodically running batch jobs, dedicated profiling of jobs is not necessary (Jyothi et al., 2016). Instead, interference and overall resource usage can be measured during the actual execution for many important production jobs, 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 for estimating the distribution of job combination goodness (Sutton & Barto, 1998). 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 (Vavilapalli et al., 2013). The scheduler selects jobs for execution on the cluster based on our continuous modeling of the rewards of scheduling specific job combinations. 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:

- We motivated the need for adaptive scheduling approaches that take the interference between co-located workloads into account with a set of cluster experiments.
- We designed a reinforcement learning solution for scheduling recurring distributed dataflow jobs in shared clusters based on their resource usage and interference between jobs.
- We implemented our solution practically for Hadoop YARN, supporting distributed dataflow systems that run on YARN such as Spark and Flink, and evaluated this implementation using a cluster of 16 commodity nodes and two workloads of Flink jobs.

*Outline.* The remainder of the paper is structured as follows. Section 2 presents the background. Section 3 presents a detailed problem analysis. Section 4 presents our scheduling approach. Section 5 presents the implementation of our prototype. Section 6 presents our evaluation. Section 7 presents the related work, while Section 8 concludes this paper.

## 2. 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.

### 2.1 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.

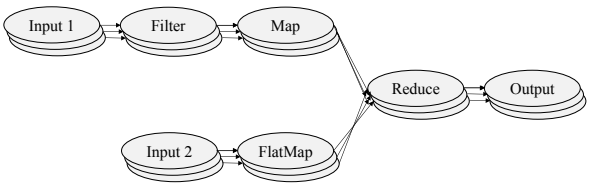



**Figure 1:** *A distributed dataflow job with different data-parallel operators.*

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.

### 2.2 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.

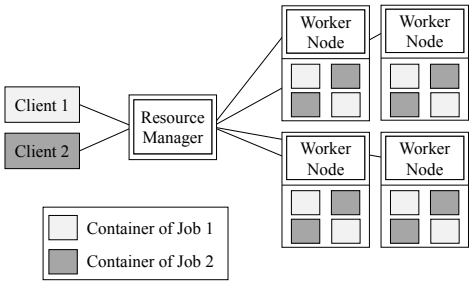



**Figure 2:** *Overview of a cluster running a resource management system.*

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.

## 3. Problem Analysis

To design a scheduler which takes into account that some co-locations are better than others, it is necessary to first identify those. Thus different co-locations of applications are presented and analyzed in this chapter.

### 3.1 Experimental Setup

In this section the setup of the co-location experiments is presented. First the cluster with its hardware and software setup is described, then the applications used for the co-locations and lastly the placement of the applications is explained.

**Cluster Setup.** The experiments are based on a forked version of the framework Flink (1.0.0) which allows to specify the nodes on which containers should be placed. The resource manager is YARN (2.7.2) and the data is stored with HDFS (2.7.2) with a data replication factor of three.

The cluster used in the following co-location experiments is constituted of 21 homogeneous nodes. Those are connected through a complete graph topology. The configuration of the nodes is as follows:

- Quadcore Intel Xeon CPU E3-1230 V2 @ 3.30GHz
- 16 GB RAM
- 3 TB RAID0 (3x1 TB HDD disks, Linux software RAID)
- 1 GBit Ethernet NIC
- CentOS 7

One node is used to run YARN's ResourceManager and HDFS' NameNode. The remaining containers are used to store data and run the applications. All containers have a fixed size of 1 CPU core and 1.5 GB of RAM.

**Container Placement.** The application execution times with two different placements are compared. The first one consists of separating the applications by avoiding any co-location on the nodes. In

**Table 1:** *Applications used in the co-location experiments.*

| Application | Data | Arguments | Abbreviation |
|---|---|---|---|
| K-Means | $10^9$ points with 1000 centers | 4 iterations | $\text{K-Means}_{1000}$ |
| | $1.25 \cdot 10^8$ points with 500 centers | 70 iterations | $\text{K-Means}_{500}$ |
| Connected Components | Twitter social graph | 3 iterations | $\text{CC}_3$ |
| | | 8 iterations | $\text{CC}_8$ |
| TPC-H Query 10 | 1 TB of data generated with DBGEN | – | $\text{TPCH}_{1000}^{10}$ |

the second configuration the applications are co-located on every node with each half of the nodes containers. In both settings, the applications have the same number of containers. What is more, individual runs were performed for each application to gain insight into the resource usage without any interference.

## 3.2 Test Workload

Three types of applications are used in the co-location experiments. The data for the K-Means job is generated using the k-means data generator bundled with Flink. The generator is configured with a standard deviation of 0.1. For Connected Components (CC), the Twitter social graph (Kwak et al., 2010) is used as the input dataset. Finally, the TPC-H[1] query jobs use data generated using DBGEN. The precise job configurations are summarized in Table 1.

To avoid that an application finishes early, leaving the co-located application with the access to the whole cluster resources, which would skew the experiment, the data used and the number of iterations are chosen such as the execution time of each co-located application is similar. The applications are always started together in all experiments.

The applications have been selected for their different resource usage. The different workloads used can be categorized as:

*CPU-intensive.* Two different version are used: $\text{K-Means}_{1000}$, which shows distinct iterations and thus having phases where CPU is almost unused, and $\text{K-Means}_{500}$ with shorter iterations leading to higher mean CPU usage of 80% instead of 60%.

*I/O-intensive.* $\text{TPCH}_{1000}^{10}$ is limited by either the disk or the network as the CPU almost never exceeds a mean usage of 50%.

*Diverse.* Connected Components is selected for its more diverse resource usage. While it is mostly CPU-intensive, the first stage is constrained by the disk and the network is used considerably all over the execution. Two versions are used to match the different execution times of $\text{K-Means}_{500}$ and $\text{K-Means}_{1000}$.

## 3.3 Results

In this section the results of the different co-location are presented. First a comparison between the co-located and the separated placement is presented, then the different co-locations are compared.

**Advantage of co-locating.** As the applications execution do not overlap exactly, it is necessary to compare more than the individual duration of each application. Hence two additional metrics are used in the following tables, which are defined as follows:

*All.* Difference between the duration of both experiments. This is the actual gain if the two applications are the only ones to be scheduled.

*First.* Difference between the duration of the first application to finish in both placements. As in the reality, more application would need to be scheduled, it indicates when the next application could be scheduled

Table 2 presents the experiments with co-location of applications with a similar resource

[1]The TPC-H specification can be found at `http://www.tpc.org/tpch/spec/tpch2.16.0v1.pdf` [accessed 2016-08-25].

**Table 2:** *Co-location results when scheduling applications with a similar resource usage.*

| Application | # Nodes | Duration [s] | | Duration Comparison | | |
|---|---|---|---|---|---|---|
| | | Separated | Co-located | Individually | All | First |
| $K\text{-}Means_{500}$[2] | 20 | 676 | 600 | -11% | -12% | -11% |
| $K\text{-}Means_{1000}$ | | 783 | 691 | -12% | | |
| $TPCH^{10}_{1000}$ | 20 | 625 | 608 | -3% | -1% | -3% |
| $TPCH^{10}_{1000}$ | | 625 | 621 | -1% | | |
| $K\text{-}Means_{500}$ | 20 | 919 | 893 | -3% | -3% | -3% |
| $CC_8$ | | 858 | 830 | -3% | | |
| $K\text{-}Means_{500}$ | 10 | 1500 | 1494 | 0% | -4% | -14% |
| $CC_8$ | | 1560 | 1290 | -17% | | |
| $K\text{-}Means_{1000}$ | 20 | 786 | 688 | -12% | -12% | -5% |
| $CC_3$ | | 617 | 585 | -5% | | |
| $K\text{-}Means_{1000}$ | 10 | 1214 | 1062 | -13% | -13% | -12% |
| $CC_3$ | | 1028 | 904 | -12% | | |

**Table 3:** *Co-location results when scheduling applications with a different resource usage.*

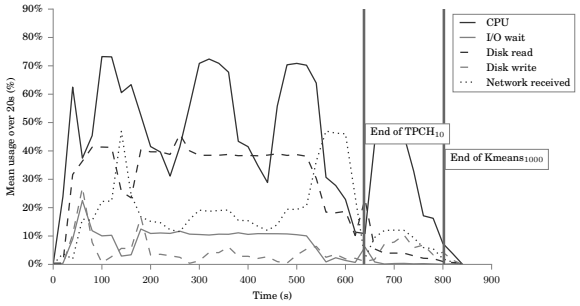
| Application | # Nodes | Duration [s] | | Duration Comparison | | |
|---|---|---|---|---|---|---|
| | | Separated | Co-located | Individually | All | First |
| $K\text{-}Means_{1000}$ | 20 | 805 | 764 | -5% | -5% | -29% |
| $TPCH^{10}_{1000}$ | | 644 | 457 | -29% | | |
| $K\text{-}Means_{1000}$ | 10 | 1244 | 1042 | -16% | -31% | -23% |
| $TPCH^{10}_{1000}$ | | 1521 | 954 | -37% | | |
| $K\text{-}Means_{1000}$ | | 1019 | 921 | -10% | | |
| $TPCH^{10}_{1000}$ | 20 | 1300 | 790 | -39% | -29% | -18% |
| $CC_3$ | | 931 | 763 | -18% | | |

usage. In Table 3 applications with different resource usage pattern are co-located. The results are clearly better for the applications which are different. Nonetheless, in both cases there is an improvement for most co-locations. Only the co-locations of $TPCH^{10}_{1000}$ with itself and $K\text{-}Means_{500}$ with $CC_8$ show only little improvement.

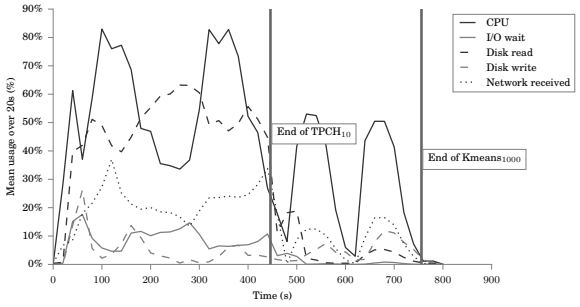
Co-locating applications seems to be the best choice from the present results. Applications have access to more resources and seem to profit from it. This can be seen by comparing the execution of $K\text{-}Means_{1000}$ with $TPCH^{10}_{1000}$ in a co-location in Figure 3b or separately in Figure 3a. However, some are better than others, and the best configuration is not always the one expected as will be seen in the next section.

**Co-location comparison.** While co-locating applications is in general profitable, not all co-locations are equal, some are better than others. For example, $K\text{-}Means_{500}$ profits from the co-location with $K\text{-}Means_{1000}$ but not from the one with $CC_8$. This difference can be explained by the pauses between the iterations of $K\text{-}Means_{1000}$, where the CPU is almost unused. $K\text{-}Means_{500}$ seems to take advantage of these phases with low CPU utilization. On the opposite, as $CC_8$ is CPU-intensive most of the execution, so there is no real improvement.

[2]Only 50 iterations are used for this experiment to have a closer execution time as in $K\text{-}Means_{1000}$

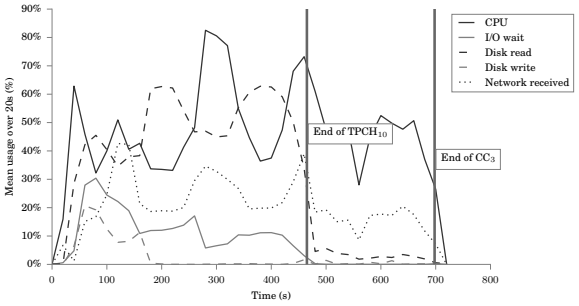



**(a)** *Separated execution.*

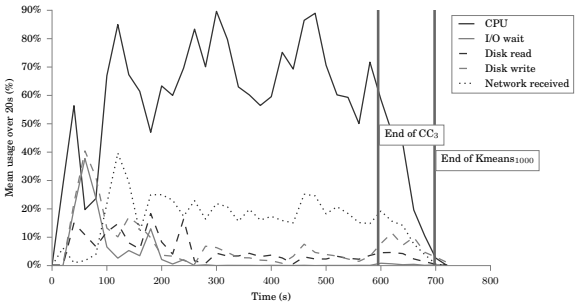



**(b)** *Co-located execution.*

**Figure 3:** *Resource usage of the execution of K-Means$_{1000}$ and TPCH$^{10}_{1000}$ on 20 nodes.*




**(a)** *Co-location with TPCH$^{10}_{1000}$.*




**(b)** *Co-location with K-Means$_{1000}$.*

**Figure 4:** *Resource usage of the execution of CC$_3$ co-located with another application.*

On the other hand, K-Means$_{1000}$ profits from every co-location with K-Means$_{500}$ or CC$_3$ by around 10%. But the reason is unclear. It may be the stage between the CPU spikes which becomes shorter, but it was not measured.

TPCH$^{10}_{1000}$ is faster in almost all case by at least 30% when co-located with CC$_3$ or K-Means$_{1000}$ as there is not much competition for the disk usage. When co-located with itself, there is almost no difference.

On first impression, one would suspect that CC$_3$ would have a better co-location with TPCH$^{10}_{1000}$ than K-Means$_{1000}$ as it uses mostly the CPU. Yet, it is the opposite as shown in Figure 4 where CC$_3$ finishes 15% faster with K-Means$_{1000}$, because of the first phase in CC$_3$ which is limited by the Disk. The negative influence of TPCH$^{10}_{1000}$ can be seen by comparing Figure 4a and Figure 3b. There is an I/O wait spike around the first 100 seconds. This interference would be hard to predict, as CC$_3$ is mostly CPU-bound.

# 4. Approach

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

## 4.1 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.

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

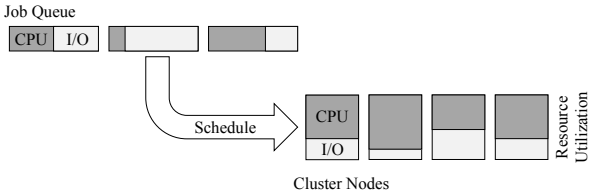



**Figure 5:** *Reordering of the jobs based on resource utilization.*

Figure 5. 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.

## 4.2 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.

*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:

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

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

*CPU.* The CPU usage $u_{\text{cpu}}$ simply represents the percentage of used CPU. The CPU I/O wait metric $u_{\text{wait}}$ is used as indicator that computation power is lost. So, it is used to weigh down the I/O utilization indicators $h_{\text{disk}}$ and network $h_{\text{net}}$ 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:

$$G \coloneqq f\Big(u_{\text{cpu}} + \big(h(r_{\text{disk}}, w_{\text{disk}}) + h(r_{\text{net}}, w_{\text{net}})\big) \cdot l(u_{\text{wait}})\Big) , \tag{1}$$

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

## 4.3 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*. In this paper, we cover a simplified version of the problem 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.

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, i.e. the one with the highest goodness measure, of the queue.*

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

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

where $S$ is the set of all applications. We use the Gradient Bandits algorithm (Sutton & Barto, 1998) to learn the preferences $H_t(a \mid a')$ 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

$$H_{t+1}(a \mid a') = H_t(a \mid a') + \gamma(G_t - \bar{G}_t)(1 - \pi_t(a \mid a')) \quad (2)$$

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

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

where $\gamma$ is the step-size parameter and $\bar{G}_t$ 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 $H_0(a \mid a') := 0$ 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 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. The other advantage is that the choice of the scheduled application is done with probabilities, so compared to a greedy algorithm, a extremely bad co-location would be have considerably less chances to happen again.

Starvation is not prevented currently but could be by weighting application with respect to the number of times they were skipped. For example, it is possible to weight $P(A_t = a \mid A' = a')$ with some positive weight $w_a$ that encodes the wait time of application $a$. Renormalizing the weighted probabilities would then result in a probability distribution that can be used for further schedulings.

# 5. Implementation

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

## 5.1 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[3] and Telegraf[4] 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 Section 3. 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.

## 5.2 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 3, 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

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

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

**Table 4:** *Applications used in the evaluation and their configuration.*

| Application | Data | Arguments | Abbreviation |
|---|---|---|---|
| K-Means | $1.25 \cdot 10^8$ points with 50 centers | 30 iterations | K-Means |
| Connected Components | First quarter of the Twitter social graph | 12 iterations | CC |
| TPCH Query 10 | 500 GB of data generated with DBGEN | – | $\text{TPCH}_{500}^{10}$ |
| TPCH Query 3 | 250 GB of data generated with DBGEN | – | $\text{TPCH}_{250}^{3}$ |

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.

### 5.3 Updating the Preferences

The goodness of currently running co-located jobs, as presented in Section 2, is measured periodically in our implementation. For any interval, all system metrics of nodes that run containers of two distinct jobs are queried from InfluxDB. Afterwards, the goodness per combination of co-located jobs is calculated with Equations (2) and (3). The result then serves as input for any new job co-location decision.

## 6. Evaluation

We evaluated our approach and implementation with two workloads scheduled with both our algorithm as presented in the Section 4 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.

### 6.1 Evaluation Setup

The cluster setup is the same as presented in Section 1.1 except that only 16 nodes were used to run the jobs. The applications used have different settings and TPCH Query 3 is added to have another I/O-bound application. The different parameters are shown in Table 4. Only pairwise co-locations are permitted. Each job uses a quarter of all containers available, thus four jobs can be scheduled simultaneously.

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:

$$(m \cdot \text{TPC-H}_{10} + m \cdot \text{K-Means} + m \cdot \text{TPC-H}_3 + m \cdot \text{CC}) \cdot n$$

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

1. *Queue A* with $n = 3$ and $m = 4$.
2. *Queue B* with $n = 4$ and $m = 3$.

**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 in the Section 4.

*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.

## 6.2 Evaluation Results

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

**Execution Time and Resource Usage.** Table 5 summarizes the performance of the three tested scheduling methods for the Queue A. 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 7–8% compared to the baseline.

**Table 5:** *Duration of the experiments with the Queue A.*

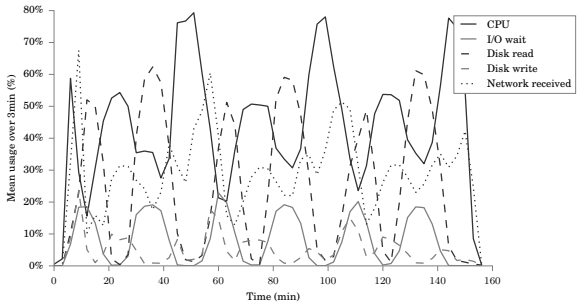
| Scheduling | Duration [min] | Change |
|---|---|---|
| FIFO | 148.5 | – |
| RA | 138.5 | 7% |
| RA* | 136.5 | 8% |

In contrast to the duration improvement for Queue A, there is no improvement detected for Queue B, as shown in Table 6.

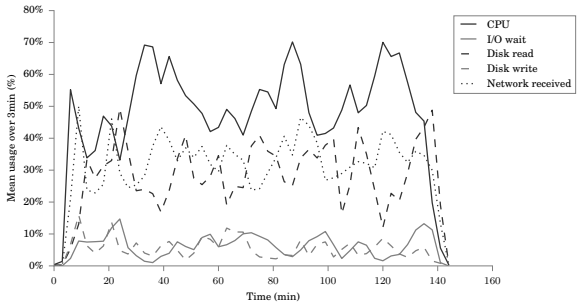
**Table 6:** *Duration of the experiments with the Queue B.*

| Scheduling | Duration [min] | Change |
|---|---|---|
| FIFO | 137.0 | – |
| RA | 137.0 | 0% |
| RA* | 136.5 | 0% |

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.




**(a)** *FIFO scheduling.*



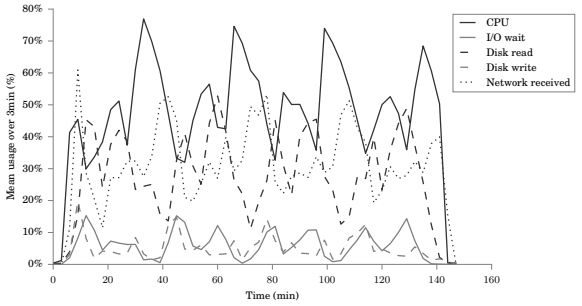

**(b)** *Resource-aware scheduling.*

**Figure 6:** *Resource usage of the Queue A.*

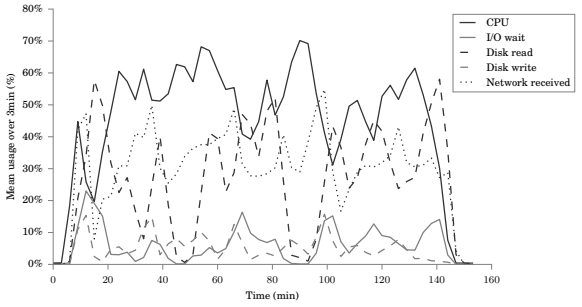
On the other hand, there is no improvement for Queue B the algorithm for the same reason. As Figure 7 shows for the FIFO scheduler, is the resource usage already distributed quite evenly. Thus there not much room left for improvements for optimizing the utilization.

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

**Application Preferences.** The data learned from each experiment by the resource-aware scheduler




**(a)** *FIFO.*




**(b)** *Resource-aware.*

**Figure 7:** *Resource usage of the Queue B.*

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 8 illustrates this relationship between the already scheduled applications and the queued applications.

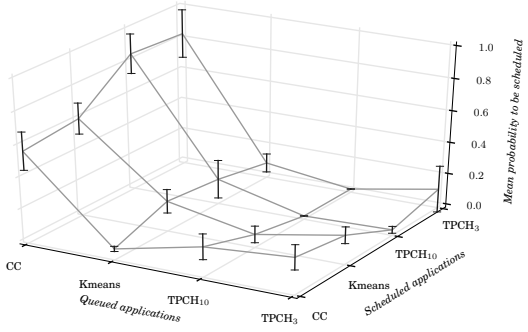



**Figure 8:** *Probability of queued applications to be selected to run with already scheduled applications.*

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 K-Means as both K-Means 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 K-Means due to the additional disk and network utilization. A similarly unexpected preference is the one of K-Means 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%.

# 7. 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.

## 7.1 Distributed Analytics Frameworks

MapReduce (Dean & Ghemawat, 2004) 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) (Ghemawat et al., 2003), which applies replication for fault tolerance.

Spark (Zaharia et al., 2010) 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) (Zaharia et al., 2012), 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 (Zaharia et al., 2013) runs on top of it by discretizing the input stream into micro-batches.

Flink (Carbone et al., 2015) 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 (Alexandrov et al., 2014) 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 (Ewen et al., 2012).

### 7.2 Resource Management Systems

YARN (Vavilapalli et al., 2013) 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.

Mesos (Hindman et al., 2011) 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.

Omega (Schwarzkopf et al., 2013) 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.

### 7.3 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.

Quasar (Delimitrou & Kozyrakis, 2014), which is built on top of Paragon (Delimitrou & Kozyrakis, 2013), 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.

Bubble-flux (Yang et al., 2013) 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 appli-

cations, 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.

Heracles (Lo et al., 2015) 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.

## 8. Conclusion

This paper presented an approach for scheduling distributed dataflow jobs based on their resource usage and interference between co-located workloads to efficiently use the resources of shared cluster infrastructures. Using a reinforcement learning algorithm to capture how well different combinations of jobs utilize shared resources, the approach does require dedicated isolated profiling of jobs, yet continuously learns scheduling which jobs jointly on to the cluster infrastructure is most advantageous for overall resource utilization and job throughput. 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 for this takes into account both how well the resources of a node are utilized and how much two jobs interfere with each other when sharing resources. For resource usage, we considered CPU, disk, and network. For interference, we incorporated I/O wait.

We implemented our approach on top of YARN. When a new job is scheduled, the learning algorithm 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 sequences of jobs with similar resource usage are submitted to the YARN cluster. The resulting resource usage fluctuates less and the execution time of the entire queue was shortened by around 8%, when our algorithm is used for scheduling jobs to the cluster resources. There was no change in runtime for the second workload, however, in which a more balanced mix of jobs was 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 balanced workloads our approach at least has no negative effect.

In the future, we want to improve learning by taking similarity between 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

Alexandrov, A., Bergmann, R., Ewen, S., Freytag, J.-C., Hueske, F., Heise, A., Kao, O., Leich, M., Leser, U., Markl, V., Naumann, F., Peters, M., Rheinländer, A., Sax, M. J., Schelter, S., Höger, M., Tzoumas, K., & Warneke, D. (2014). The Stratosphere Platform for Big Data Analytics. *The VLDB Journal*, *23*(6), 939–964.

Barroso, L. A., & Hölzle, U. (2007). The Case for Energy-Proportional Computing. *Computer*, *40*(12), 33–37.

Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., & Tzoumas, K. (2015). Apache Flink$^{TM}$: Stream and Batch Processing in a Sin-

gle Engine. *IEEE Data Engineering Bulletin*, *38*(4), 28–38.

Carvalho, M., Cirne, W., Brasileiro, F., & Wilkes, J. (2014). Long-term SLOs for Reclaimed Cloud Computing Resources. In *ACM Symposium on Cloud Computing*, SOCC '14, (pp. 20:1–20:13). ACM.

Dean, J., & Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters. In *6th Conference on Symposium on Operating Systems Design & Implementation*, OSDI'04, (pp. 10–10). USENIX Association.

Delimitrou, C., & Kozyrakis, C. (2013). Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, (pp. 77–88). ACM.

Delimitrou, C., & Kozyrakis, C. (2014). Quasar: Resource-efficient and QoS-aware Cluster Management. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (pp. 127–144). ACM.

Ewen, S., Tzoumas, K., Kaufmann, M., & Markl, V. (2012). Spinning Fast Iterative Data Flows. *Proc. VLDB Endow.*, *5*(11), 1268–1279.

Ferguson, A. D., Bodik, P., Kandula, S., Boutin, E., & Fonseca, R. (2012). Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *7th ACM European Conference on Computer Systems*, EuroSys '12, (pp. 99–112). ACM.

Ghemawat, S., Gobioff, H., & Leung, S.-T. (2003). The Google File System. In *Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, (pp. 29–43). ACM.

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., & Stoica, I. (2011). Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, (pp. 295–308). USENIX Association.

Jyothi, S. A., Curino, C., Menache, I., Narayanamurthy, S. M., Tumanov, A., Yaniv, J., Mavlyutov, R., Goiri, I. n., Krishnan, S., Kulkarni, J., & Rao, S. (2016). Morpheus: Towards Automated SLOs for Enterprise Clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, (pp. 117–134). USENIX Association.

Kwak, H., Lee, C., Park, H., & Moon, S. (2010). What is Twitter, a Social Network or a News Media? In *19th International Conference on World Wide Web*, WWW '10, (pp. 591–600). ACM.

Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P., & Kozyrakis, C. (2015). Heracles: Improving Resource Efficiency at Scale. In *42nd Annual International Symposium on Computer Architecture*, ISCA '15, (pp. 450–462). ACM.

Reiss, C., Tumanov, A., Ganger, G. R., Katz, R. H., & Kozuch, M. A. (2012). Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *Third ACM Symposium on Cloud Computing*, SoCC '12, (pp. 7:1–7:13). ACM.

Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., & Wilkes, J. (2013). Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *8th ACM European Conference on Computer Systems*, EuroSys '13, (pp. 351–364). ACM.

Sutton, R. S., & Barto, A. G. (1998). *Introduction to Reinforcement Learning*. Cambridge, MA, USA: MIT Press, 1st ed.

Thamsen, L., Rabier, B., Schmidt, F., Renner, T., & Kao, O. (2017). Scheduling Recurring Distributed Dataflow Jobs Based on Resource Utilization and Interference. In *6th 2017 IEEE International Congress on Big Data (BigData Congress 2017)*, BigData Congress, (pp. 145–152). IEEE.

Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., & Baldeschwieler, E. (2013). Apache Hadoop YARN: Yet Another Resource Negotiator. In *4th Annual Symposium on Cloud Computing*, SOCC '13, (pp. 5:1–5:16). ACM.

Verma, A., Cherkasova, L., & Campbell, R. H. (2011). ARIA: Automatic Resource Inference

and Allocation for Mapreduce Environments. In *8th ACM International Conference on Autonomic Computing*, ICAC '11, (pp. 235–244). ACM.

Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., & Wilkes, J. (2015). Large-scale Cluster Management at Google with Borg. In *Tenth European Conference on Computer Systems*, EuroSys '15, (pp. 18:1–18:17). ACM.

Yang, H., Breslow, A., Mars, J., & Tang, L. (2013). Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *40th Annual International Symposium on Computer Architecture*, ISCA '13, (pp. 607–618). ACM.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., & Stoica, I. (2012). Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, (pp. 2–2). USENIX Association.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010). Spark: Cluster Computing with Working Sets. In *2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, (pp. 10–10). USENIX Association.

Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., & Stoica, I. (2013). Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (pp. 423–438). ACM.

## Authors

**Lauritz Thamsen** is a PhD candidate at the Technische Universität Berlin, working in the Complex and Distributed Systems group. Prior to that, Lauritz earned his Bachelor's and Master's degree in Software Engineering from Hasso Plattner Institute, University of Potsdam, where he was part of the Software Architecture Group of Robert Hirschfeld. His research interests include distributed data processing and programming tools.

**Ilya Verbitskiy** is a PhD candidate at the Technische Universität Berlin, working in the Complex and Distributed Systems group. He received his Bachelor's and Master's degree in Computer Science from the same university. His research interests include distributed data processing and machine learning.

**Benjamin Rabier** is working as a Data Scientist at Nokia Digital Health in Paris, France. He received his engineering degree from the École Centrale de Lyon and his M.Sc. in Software Engineering from the Technische Universität Berlin, where he was part of the Complex and Distributed Systems group. His research interest include distributed data processing and machine learning.

**Dr. Odej Kao** is a Full Professor at the Technische Universität Berlin and head of the research group on Complex and Distributed IT systems. He is also the chairman of the Einstein Center Digital Future, which is hosting 50 interdisciplinary professors. Dr. Kao is a graduate from the TU Clausthal, where he earned a Master's degree in Computer Science in 1995, a PhD in 1997, and an advanced PhD in 2002. In 2002, Dr. Kao joined the University of Paderborn, Germany as Associated Professor for Distributed and Operating systems. One year later, he became managing director of the Paderborn Center for Parallel Computing (PC2), where he has conducted research and many industry-driven projects on high-performance computing, resource management, and Grid/Cloud computing. In 2006, he moved to the TU Berlin and focused his research on cloud computing, resource management, QoS, anomaly detection, and Big Data analytics. Since 1998, he has published over 300 papers in peer-reviewed scientific conferences and journals.