# Adaptive Resource Management for Distributed Data Analytics Based on Container-level Cluster Monitoring

Thomas Renner, Lauritz Thamsen, and Odej Kao
*Technische Universität Berlin, Berlin, Germany*
*Complex and Distributed IT Systems*
{*firstname.lastname*}*@tu-berlin.de*

Keywords: Cluster Resource Management, Distributed Data Analytics, Distributed Dataflow Systems, Adaptive Resource Management

Abstract: Many distributed data analysis jobs are executed repeatedly in production clusters. Examples include daily executed batch jobs and iterative programs. These jobs present an opportunity to learn workload characteristics through continuous fine-grained cluster monitoring. Therefore, based on detailed profiles of resource utilization, data placement, and job runtimes, resource management can in fact adapt to actual workloads. In this paper, we present a system architecture that contains four mechanisms for an adaptive resource management, encompassing data placement, resource allocation, and container as well as job scheduling. In particular, we extended Apache Hadoop's scheduling and data placement to improve resource utilization and job runtimes for recurring analytics jobs. Furthermore, we developed a Hadoop submission tool that allows users to reserve resources for specific target runtimes and which uses historical data available from cluster monitoring for predictions.

## 1 Introduction

Modern distributed data-analytic frameworks such as Spark (Zaharia et al., 2010b) and Flink (Carbone et al., 2015) allow to process large datasets in parallel using a large number of computers. Often, applications of these frameworks (i.e. jobs) run within virtualized containers on top of cluster resource management systems like YARN (Vavilapalli et al., 2013), which provide jobs a specific amount of resources for their execution. In addition, a co-located distributed file system such as HDFS (Shvachko et al., 2010) stores the data that is to be analyzed by the data-analytic frameworks. This design allows to share cluster resources effectively by running data analytic jobs side-by-side on a single cluster infrastructure composed of up to hundreds or more nodes and accessing shared datasets. Thereby, one challenge is to realize an effective resource management of these large cluster infrastructures in order to run data analytics in an economically viable way.

Resource management for data-analytic clusters is a challenging task as analytic jobs often tend to be long running, resource-intensive, and consequently expensive. For instance, in most resource management systems the user needs to specify upfront how many resources a job allocates for its execution in terms of number of containers as well as cores and memory per container. Furthermore, users often tend to allocate too many resources for their jobs. In terms of numbers, a productive cluster at Google achieved aggregate CPU utilization between 25% and 30% and memory utilization of 40%, while resource reservations exceeded 75% and 60% of the available CPU and memory capacities (Verma et al., 2015). Furthermore, another challenge for effective resource management is that data-analytic jobs often have different workload characteristics and, thus, stress different resources. For instance, some frameworks provide libraries for executing scalable machine learning jobs such as Spark MLlib (Meng et al., 2016), which often tend to be more CPU-intensive. On the contrary, jobs containing queries for data aggregation tend to be more I/O-intensive (Jia et al., 2014). Knowing the dominant resource of a job in advance can be used to schedule a job with other jobs for a better node utilization and less interfere between jobs. For data-intensive jobs, achieving a high degree of data locality can reduce the network demand due to the availability of input data on local disks. In this case, data can be ingested faster and the computation can start immediately (Zaharia et al., 2010a). However, the set

of containers, in which a job and its tasks is running, is chosen by the resource manager that often schedules containers without taking the location of data into account. Thus, the task scheduling of data-analytic frameworks is restricted to the set of nodes the containers are running on.

Besides, identifiable workloads characteristics such as containing more I/O- or more CPU-intensive jobs, typically workloads also contain many analytics jobs that are executed repeatedly. This is the case for iterative computations, yet also for recurring batch analytics, for which the same job is executed on a weekly or even daily schedule. Such recurring batch jobs can make up to 40% of all jobs in productive clusters (Agarwal et al., 2012; Chaiken et al., 2008).

In summary, on the one hand cluster workloads can vary significant in terms of which resources are most dominant and thus which optimizations are most effective, while workloads often have defining characteristics as well as consist of repeatedly executed jobs. Therefore, we argue that resource management for data-analytic clusters should automatically adapt to the workloads running on the resources. Furthermore, detailed knowledge of a job's resource usage and scale-out behavior also allows to execute jobs more efficiently as well as with respect to user performance requirements. The basis for this knowledge on workload characteristics and particular jobs is fine-grained cluster monitoring, in case of resource-managed clusters on a container-level. Moreover, user performance requirements can be collected to not only adapt to job characteristics but also make informed decisions based on user needs.

In this paper we report our experience of applying adaptive resource management for distributed data analytics based on such fine-grained container-level cluster monitoring. In particular, we present a system based on Hadoop[1] that allows to manage different aspects of resource management and which we used to improve job execution time, increase resource utilization, and meet performance demands of users. The system measures job runtimes and the resource consumption of every single job based on all its execution containers utilizations. In addition, we record information on the input data to improve data locality of the frameworks and colocate related files on the same set of nodes. We further schedule combinations of jobs to run together on the cluster for which we observed high overall resource utilization and little inference when executed co-located. The system is also capable of taking user performance constraints in terms of runtime targets and automatically allocating resources for these targets, after modeling the scale-out performance of a job based on previous runs. In addition, the system dynamically adapts resource allocations at runtime towards user-defined constraints such as resource utilization targets at barriers between dataflow stages.

*Outline*. The remainder of the paper is structured as follows. Section 2 provides background on systems used for data analytics. Section 3 presents our idea of adaptive resource management. Section 4 explains our system architecture for adaptive resource management. Section 5 presents four different applications of our idea of adaptive resource management. Section 6 discusses related work. Section 7 concludes this paper.

[1] Apache Hadoop, `http://www.hadoop.apache.org/`, accessed 2017-03-20

# 2 Background

This section describes distributed dataflow systems, distributed file systems, and resource management systems. Together these systems often form the setup of shared nothing compute clusters used for data analytics.

## 2.1 Distributed Dataflow Systems

In distributed dataflow systems tasks receive inputs and produce outputs. These tasks are data-parallel, so each parallel task instance receives only parts of the input data and produces only parts of the output data of the entire dataflow. The tasks usually are user-defined versions of a set of defined operators such as Map, Reduce, and Join. A Map, for example, is configured with a user-defined function (UDF). A Join, in contrast, is configured by defining the join criteria, i.e. which attributes have to match for elements to be joined. Therefore, operators like Joins, but also, for example, group-wise reductions require elements with matching keys to be available at the same task instances. If this is not already given from the preceding tasks, the input data is re-partitioned for these operations.

In general the tasks form a directed graph, often supporting joins of multiple separate dataflows or even forks of a single one. Some frameworks also support iterative programs with cyclic directed graphs or features like result caching. Besides data parallelism, task parallelism is often realized in form of pipeline parallelism. Typically, the computation is distributed and orchestrated by a master across many workers. Workers expose their compute capabilities via slots, which host either single task instances or chains of subsequent task instances.

Key features of such distributed dataflow systems include effective task distribution, fault tolerance, scalability, processing speed, and usability. Moreover, program code typically does not contain details on the execution environment, which instead is specified in separate configuration files, allowing the same program to be used in different environments. Consequently, developers can concentrate on application logic, while cluster operation teams focus on system specific such as resource allocation and configuration.

## 2.2 Distributed File Systems

In distributed file systems large files are typically fragmented into blocks, which are then stored across multiple nodes. Each node can store many blocks of multiple different files and the blocks of a large file can be stored across many nodes.

Such systems typically also follow the master/worker pattern. The master stores metadata, such as which block of which file is stored on a particular node. This metadata is usually structured as file directories, allowing users to access files by specifying the path to a file. When a client requests a file from the master, the master returns the exact block locations, which the client consequently uses to read the actual file blocks directly from the workers.

Key features of distributed file systems include fault tolerance, scalability, as well as usability. Fault tolerance is usually realized using replication, so that each block is often replicated three times, each one stored on different nodes or even different racks. The systems then implement fault detection and recovery to continuously re-replicate blocks in case of failures. Regarding usability, from a user's perspective the distributed nature of the file system is abstracted. That is, access to the distributed files is usually just as easy as accessing local files, except for maybe read/write performance. To increase access performance, a commonly applied technique is selecting compute nodes for tasks with a maximum of input data locally available.

## 2.3 Resource Management Systems

Resource management systems allow to share cluster resources among multiple users and applications. Often, these systems also support multiple different analytics solutions such as different distributed dataflow systems. Thus, users can select per application which framework is best for their task. Then, users reserve parts of the cluster for a time through containers. These containers represent compute capabilities. Often, a container is a reservation for a number of cores and an amount of main memory. Nodes can host multiple such containers–as many as do fit their resources–and multiple containers on a number of hosts are typically reserved for a single application.

Resource management systems also usually follow the master/worker pattern, with a central master responsible for scheduling and orchestration, while workers host application containers. More advanced system designs apply master replication for fault tolerance or decentralized scheduling for scalability. In case of multiple schedulers used for scalability, usually optimistic concurrency control schemes are used. There is also a middle-ground between centralized and decentralized resource managers with hybrid solutions with, for example, hierarchical designs.

# 3 Adaptive Resource Management

The goal of adaptive resource management is to continuously improve the resource utilization of a data-analytic cluster. The overall idea is to learn about the cluster workloads and automatically adapt data and container placement as well as job scheduling based on these information. Workloads often have predictable characteristics since up to 40 % of jobs in productive clusters are recurring (Agarwal et al., 2012; Chaiken et al., 2008). For these jobs, the execution logic stays the same for every execution, however the input data may changes. Typical scenarios for recurring jobs include when new data becomes available or at discrete times such as for hourly or nightly batch reports.

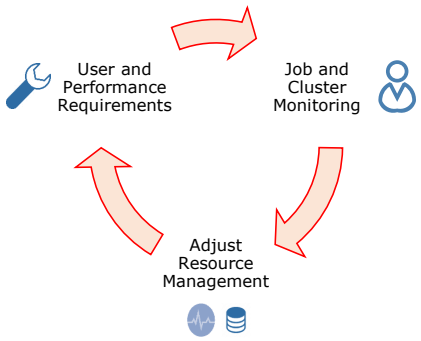


Figure 1: Adaptive resource management cycle.

Figure 1 illustrates the process of our adaptive resource management approach. The distinct steps in this cycle are:

- User and Performance Requirements: User specify performance requirements regarding runtime targets. For instance, a recurring job should be finished in an hour. Based on previous runs of that

job, the necessary set of resources is estimated and allocated.

- Job and Cluster Monitoring: Jobs that are executed on the cluster are continuously monitored on a fine-grained container level. Based on this historical workload data, job repositories are generated. Repository data includes execution time, data placement, data access pattern, resource utilization, and resource allocation. These detailed profiles allow to learn for future executions of jobs and, thereby, improve resource utilization.
- Adjusted Resource Management: Adaptive resource management is achieved by data block placement, container placement, and job scheduling. Data blocks are placed on as many nodes as used by a job for optimal data locality. In addition, related files are colocated on the same set of nodes. Containers of jobs that use these data as input are placed on these nodes. Job scheduling is used to run jobs with complimentary resource usage and low interference together. As in many resource management systems resource utilization of containers is not strictly limited to the number of resources used for scheduling, it is beneficial to schedule an I/O intensive job with an CPU intensive job.

## 4 System Architecture

This section describes our system architecture for implementing adaptive resource management for distributed data analytics. The system consists of a monitoring component called Freamon that provides container-level job monitoring for Hadoop deployments. It monitors resource utilization of single data analytic jobs by monitoring all its execution containers that run on different nodes. A data analytic job can be any application that supports execution on Hadoop YARN. In addition, Freamon measures the total node utilization to analyze interference of different container placements as well as storage utilization and data block placements. Different adaptive resource management application use Freamon's historical job data for adaptive container placement, data block placement, and container scheduling decisions.

Figure 2 shows how Freamon is integrated with existing systems and its components for providing fine-grained monitoring which then enables adaptive resource management applications. Freamon relies on Hadoop YARN and HDFS. The *Compute Layer* is represented by YARNs slave nodes (i.e. *NodeManagers*), which provide data analytic jobs with cluster compute resources through containers. The *Data Layer* includes all HDFS slave nodes (i.e. *DataNodes*), which are responsible for storing datasets in series of blocks distributed across all available DataNodes. Both layers are running on all cluster nodes. An important reason for co-locating both systems is to improve data locality, the attempt to execute tasks on nodes where the input data is stored. Data locality can reduce the network demand due to the availability of the input data on local disks and, thus, ingestion is faster, allowing computation to start earlier.

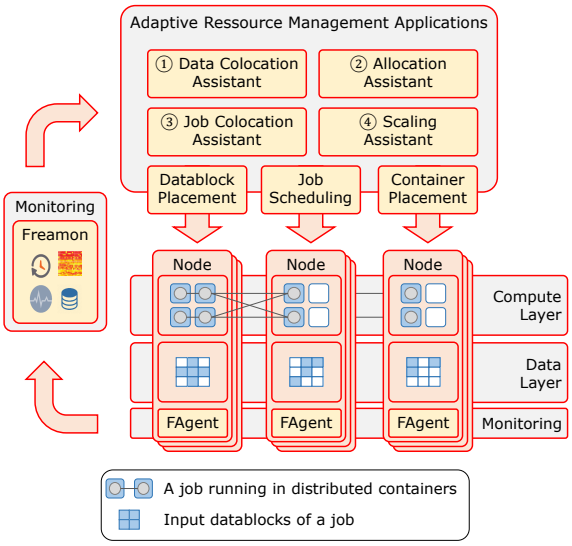


Figure 2: System overview for applying adaptive resource management.

The *FAgent* runs on every node and collects different node-related system metrics. It collects resource utilization of the whole node such as cpu, disk, network, and memory data. In addition, Freamon tracks resource utilization on a more fine-grained container-level, in which the data analytic jobs are executed. Thereby, it is to mentation that resources of containers are not strictly allocated. A container can gain more resources if available. Thus, containers executing jobs are competing for available node resources. The advantage is that a job can use more resources, if available. With Freamon, we know exactly how much a job was utilizing from a node and can use this information later for resource savings for recurring jobs. The agent uses the *proc* filesystem and monitors the *pid* of the container executing the job. It is also a frontend to other unix monitoring tools if available. Records are sent periodically to the Freamon monitoring master.

*Freamon* is the master monitoring component, in which all information about resource utilizations collected by the distributed FAgents come together. In addition, it communicates with the YARN master

node (i.e. *ResourceManager*) to gather job-specific data of finished jobs like: runtime, name, application framework, and scheduling profile including number of containers, cores as well as memory used for scheduling. Freamon also collects information about datasets that are processed by jobs. These information contain the size of input datasets as well as data placements, e.g. on which nodes the data blocks are stored and how often the data is accessed by analyzing the HDFS audit log. All job profiles are stored in a repository of historical workload data. The job and data profiles can be accessed by different adaptive resource management applications.

*Adaptive Resource Management Applications* Adaptive resource management applications use Freamon's fine-grained cluster monitoring and workload repository for different purposes. For example, the Allocation Assistant uses job runtimes for automatic resource allocation according to users' performance targets, whereas the two Colocation Assistants for container as well as data placement aim for improved data locality, local data exchange, resource utilization, and throughput. Different applications of adaptive resource management are discussed in Section 5 in more detail.

# 5 Applications of Adaptive Resource Management

This section summarizes four specific adaptive resource management techniques for distributed data analytics that we implemented. The contribution of this section is to show the applicability of adaptive resource management for distributed data analytics using the envisioned container-level cluster monitoring system in more detail.

## 5.1 Data Colocation Assistent

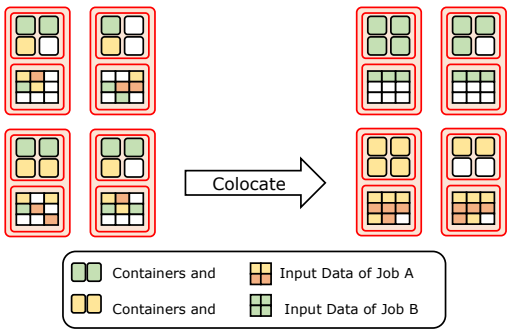


Figure 3: Data Colocation Assistant - data and container colocation for data-intensive applications.

The performance for data-intensive jobs often depends on the time it takes to read input data. Furthermore, many jobs are recurring and, for example, are triggered at a discrete time for daily or nightly execution or when new data becomes available. For this recurring data-intensive jobs, it is possible to decide where to store the input data in the distributed file system and containers before the job execution starts. The Data Colocation Assistant (Renner et al., 2016) uses these characteristics and consists of a data block and container placement phase. First, it allows to mark sets of files as related, which, for instance, often are processed jointly. These data blocks of the related files are automatically placed on the same set or subset of nodes during the data placement phase. Afterwards during the container placement phase, the job and its containers are scheduled on these nodes, where the data was placed before. The main advantage of CoLoc is a reduction of network transfers due to a higher data locality and some locally performed operators like grouping or joining. Thus, especially data-intensive workloads benefit this adaptive resource management application. Figure 3 illustrates the process of the adaptive data and container colocation application.

## 5.2 Resource Allocation Assistant

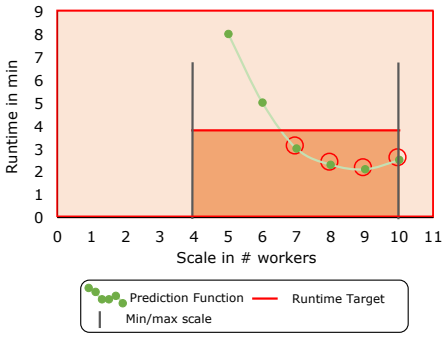


Figure 4: Resource Allocation Assistant - resource selection based on modeled scale-out behavior of a recurring job.

Given a runtime target, the Resource Allocation Assistant (Thamsen et al., 2016b) automatically selects the number of containers to allocate for a job. For this, the Resource Allocation Assistant models the scale-out behavior of job based on previous runs. Therefore, we do not require dedicated isolated profiling of jobs. Instead, the Resource Allocation Assistant retrieves runtime information on the previous executions of a job from Freamon's workload repository and then uses linear regression to find a function for the job's scale-out performance. In particular, we use both a generic parametric model for dis-

tributed computation and nonparametric regression. The parametric model is used for extrapolation and when not enough data is available for effective modeling with the nonparametric approach. The nonparametric approach allows to interpolate arbitrary scaling behavior with high accuracy when enough samples are available. The Resource Allocation Assistant automatically selects between both models using cross-validation. When the scale-out model is determined, this model is used to allocate resources for the user's runtime target. In particular, the Resource Allocation Assistant uses the lowest scale-outs within user-defined bounds that satisfies the target constraint. This process is shown in Figure 4.

## 5.3 Job Colocation Assistant

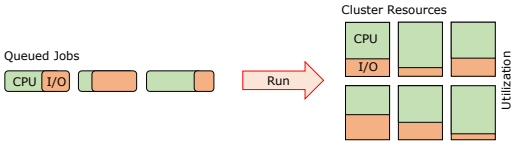


Figure 5: Job Colocation Assistant - job scheduling based on resource usage and interference.

The Job Colocation Assistant (Thamsen et al., 2017) schedules jobs based on their resource usage and interference with other jobs. In particular, it selects jobs to run on the cluster based on the jobs already running on the cluster, as shown in Figure 5. For this, it uses a reinforcement learning algorithm to learn which jobs utilize the different resources best, while interfering with each other least. Consequently, this scheduling scheme can make more informed decisions as time goes by and jobs are executed repeatedly on the cluster, yet does not require dedicated isolated profiling of jobs. To measure the goodness of colocation we take CPU, disk, and network usage as well as I/O wait into account. CPU, disk, and network usage we rate highly when utilized high, while we use I/O wait as indicator of interference. That is, lower I/O wait is better. The computed colocation preferences are then used to select the next job from the queue of scheduled jobs.

## 5.4 Dynamic Scaling Assistant

The Dynamic Scaling Assistant (Thamsen et al., 2016a) adjusts resource allocations at runtime to meet user-defined execution constraints such as resource utilization targets. For these dynamic scalings, we use the barriers between dataflow stages. In particular, there are barriers between the iterations of iterative dataflow programs. Scaling at these barriers allows to adjust the resource allocation without having to migrate task state. Moreover, scaling can be done on the basis of runtime statistics that reflect the entire elements of the dataflow. That is, at a barrier all elements of the previous stage have been processed and, thus, are reflected in, for example, the average resource utilization of the nodes. This is important as data characteristics such as key value distributions can have a significant influence on the performance of dataflows. This is, for example, the case when the partitions are uneven and worker nodes with little work, therefore, have to wait on worker nodes with lots of work at barriers.

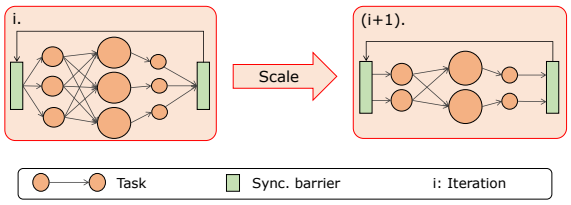


Figure 6: Dynamic Scaling Assistant - resource allocation adjustment for dataflow jobs at synchronization barriers.

# 6 Related Work

This section describes related work in four categories: distributed dataflow systems, distributed file systems, resource management systems, as well as work on using analytical cluster resources adaptively.

## 6.1 Distributed Dataflow Systems

MapReduce (Dean and Ghemawat, 2004) offers a programming and an execution model for scalable data analytics with distributed dataflows on shared nothing clusters. The programming model is based on the two higher order functions *Map* and *Reduce*, which are both supplied with UDFs. The execution models comprises the data-parallel execution of task instances of these two operations, where each Map phase is followed by a Reduce phase. In-between both phases the intermediate results are written to disk and shuffled: elements with the same key are read by the same Reduce task instance. Fault tolerance is given because a distributed file system with replication is used for the intermediate results.

Systems like Dryad (Isard et al., 2007) and Nephele (Warneke and Kao, 2009) added the possibility to develop arbitrary directed acyclic task graphs instead of just combinations of subsequent Map and Reduce tasks. SCOPE (Chaiken et al., 2008), Nephele/PACTs (Battré et al., 2010) and Spark (Za-

haria et al., 2010b) then added more operators such as Joins. Spark also introduced an alternative approach to fault tolerance: instead of saving intermediate results in a fault-tolerant distributed file system, Spark's Resilient Distributed Datasets (RDDs) (Zaharia et al., 2012) store enough linage information to be able to re-compute specific partitions in case of failures. Spark also enables users to explicitly cache intermediate results for future usage. This allows, for example, to speed up iterative computations that require an input repeatedly.

Flink (Carbone et al., 2015; Alexandrov et al., 2014) and Google's Dataflow (Akidau et al., 2015) add many features regarding continuous inputs and scalable stream processing. Flink uses in fact a stream processing engine for both batch and stream processing. Furthermore, they add functionality to define windows for, for example, joins and aggregations. They also provide mechanisms to deal with elements that arrive late. Spark also provides a system for stream processing (Zaharia et al., 2013). However, Spark uses microbatches and no true streaming engine for this.

## 6.2 Distributed File Systems

Distributed File Systems focusing on data analytics typically store a file in series of blocks and stripe the data block across multiple servers. In addition, most add redundancy to the stored data by replicating the data blocks to provide fault tolerance against node failures. HDFS (Shvachko et al., 2010) is part of Hadoop and currently the de-facto storage system for storing large datasets for data analytic tasks. Other famous storage systems are Ceph (Weil et al., 2006), GlusterFS (Davies and Orsaria, 2013), and XtreemFS (Hupfeld et al., 2008).

Alluxio, former known as Tachyon (Li et al., 2014), is a memory-centric distributed storage system enabling data sharing at memory-speed supporting many Hadoop compatible frameworks such as Spark, Flink or MapReduce. It stores data off-heap and can run on top of different storage systems such as HDFS persistant layer.

## 6.3 Resource Management Systems

Different resource management system with a focus on data analytics like YARN (Vavilapalli et al., 2013), Mesos (Hindman et al., 2011), or Omega (Schwarzkopf et al., 2013) have been developed. In these systems, users can allocate cluster resources by usually specifying the number of containers and their size in terms of cores and available memory. After these containers are deployed on the cluster infrastructure, distributed dataflow systems are executed here. YARN is part of Hadoop and emerged of the MapReduce framework, and thus mainly focuses on batch jobs. One significant difference between YARN and Mesos is that YARN is a monolithic and Mesos a non-monolithic two-level scheduler. Other known cluster resource management systems include Microsoft's Apollo (Boutin et al., 2014), and Google's Borg (Verma et al., 2015)

## 6.4 Adaptive Resource Management

The most related architectures are Quasar (Delimitrou and Kozyrakis, 2014) and Jockey (Ferguson et al., 2012).

Quasar (Delimitrou and Kozyrakis, 2014), which is a successor system to Paragon (Delimitrou and Kozyrakis, 2013), automatically performs resource allocation and job placement based on previously observed and dedicated sample runs. Quasar also adjusts allocations at runtime after monitoring. In contrast, our architecture also encompasses data placement and selecting the next job from the queue of scheduled jobs.

Jockey is a resource manager for Scope, which performs initial resource allocation and further adapts the allocation at runtime towards a user-given runtime target. For this, Jockey uses detailed knowledge of the execution model and instrumentation to gain comprehensive insights into the jobs. Jockey, however, does not take care of data placement and also does not schedule jobs.

Further related systems in regards to automatic resource allocation include, for example, Elastisizer (Herodotou et al., 2011a), which is part of the Starfish (Herodotou et al., 2011b) and also selects resources for a user's runtime target.

Related systems in regards to dynamic scaling and other runtime adjustments include many works in the area of large-scale stream processing. Works such a StreamCloud (Gulisano et al., 2012) and QoS-based Dynamic Scaling for Nephele (Lohrmann et al., 2015) have investigated adaptive dynamic scaling for large-scale distributed stream processing. There is also work on adaptively placing and migrating tasks at runtime based on execution statistics. An example is placing tasks that exchange comparably large amounts of the data onto the same hosts in Storm (Aniello et al., 2013)

In regards to adaptive data placement, the closest related works are Scarlett (Ananthanarayanan et al., 2011) and ERMS (Elastic Replica Management system) (Cheng et al., 2012). Both systems mine access

patterns of applications and use these information to increase and decrease data replication factor of files that are accessed most to improve throughput.

## 7 Conclusion

In this paper, we presented an architecture of a set of applications that make resource management for distributed dataflows more adaptive based on cluster monitoring. Our solution monitors job runtimes, container-based resource utilization, and data access. Based on these information, we adapt different aspects of resource management to the actual workload and, in particular, repeatedly executed dataflow jobs such as recurring batch jobs. Specifically, we adapt the following aspects of resource management: data placement, resource allocation, job scheduling, and container placement. The four adaptive resource management applications we implemented are:

1. First, we place data on as many nodes as used by a job for optimal data locality.
2. Second, we allocate as many resources for a job as a user's runtime target requires after modeling the scale-out performance of a job using its previous runs.
3. Third, we schedule jobs to run together that exhibit a high overall resource utilization yet little inference when executed co-located.
4. Fourth, we adjust resource allocations at runtime towards user-defined constraints such as utilization targets at barriers between subsequent dataflow stages.

While we have presented these four different techniques before, this paper presents them as applications in an overall architecture. In the future, we want to evaluate how well these techniques work together and also implement further ideas for adaptive resource management based on fine-grained cluster monitoring.

## ACKNOWLEDGEMENTS

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

Agarwal, S., Kandula, S., Bruno, N., Wu, M.-C., Stoica, I., and Zhou, J. (2012). Reoptimizing data parallel computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 281–294.

Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R. J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., and Whittle, S. (2015). The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing. *Proc. VLDB Endow.*, 8(12):1792–1803.

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., and Warneke, D. (2014). The Stratosphere Platform for Big Data Analytics. *The VLDB Journal*, 23(6):939–964.

Ananthanarayanan, G., Agarwal, S., Kandula, S., Greenberg, A., Stoica, I., Harlan, D., and Harris, E. (2011). Scarlett: coping with skewed content popularity in mapreduce clusters. In *Proceedings of the sixth conference on Computer systems*, pages 287–300. ACM.

Aniello, L., Baldoni, R., and Querzoni, L. (2013). Adaptive Online Scheduling in Storm. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 207–218. ACM.

Battré, D., Ewen, S., Hueske, F., Kao, O., Markl, V., and Warneke, D. (2010). Nephele/PACTs: A Programming Model and Execution Framework for Web-scale Analytical Processing. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 119–130. ACM.

Boutin, E., Ekanayake, J., Lin, W., Shi, B., Zhou, J., Qian, Z., Wu, M., and Zhou, L. (2014). Apollo: Scalable and Coordinated Scheduling for Cloud-scale Computing. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 285–300. USENIX Association.

Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. (2015). Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin*, 38(4):28–38.

Chaiken, R., Jenkins, B., Larson, P.-A., Ramsey, B., Shakib, D., Weaver, S., and Zhou, J. (2008). SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *Proc. VLDB Endow.*, 1(2):1265–1276.

Cheng, Z., Luan, Z., Meng, Y., Xu, Y., Qian, D., Roy, A., Zhang, N., and Guan, G. (2012). ERMS: An Elastic Replication Management System for HDFS. In *2012 IEEE International Conference on Cluster Computing Workshops*, pages 32–40. IEEE.

Davies, A. and Orsaria, A. (2013). Scale out with GlusterFS. *Linux Journal*, 2013(235).

Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Sys-*

*tems Design & Implementation*, OSDI'04, pages 10–10. USENIX Association.

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

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

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

Gulisano, V., Jimenez-Peris, R., Patino-Martinez, M., Soriente, C., and Valduriez, P. (2012). StreamCloud: An Elastic and Scalable Data Streaming System. *IEEE Trans. Parallel Distrib. Syst.*, 23(12):2351–2365.

Herodotou, H., Dong, F., and Babu, S. (2011a). No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 18:1–18:14. ACM.

Herodotou, H., Lim, H., Luo, G., Borisov, N., Dong, L., Cetin, F. B., and Babu, S. (2011b). Starfish: A Self-tuning System for Big Data Analytics. In *Proceedings of the the 5th Conference on Innovative Data Systems Research*, CIDR '11. CIDR 2011.

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., and Stoica, I. (2011). 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*, NSDI'11, pages 295–308. USENIX Association.

Hupfeld, F., Cortes, T., Kolbeck, B., Stender, J., Focht, E., Hess, M., Malo, J., Marti, J., and Cesario, E. (2008). The xtreemfs architecturea case for object-based file systems in grids. *Concurrency and computation: Practice and experience*, 20(17):2049–2060.

Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. (2007). Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72. ACM.

Jia, Z., Zhan, J., Wang, L., Han, R., McKee, S. A., Yang, Q., Luo, C., and Li, J. (2014). Characterizing and subsetting big data workloads. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 191–201. IEEE.

Li, H., Ghodsi, A., Zaharia, M., Shenker, S., and Stoica, I. (2014). Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–15. ACM.

Lohrmann, B., Janacik, P., and Kao, O. (2015). Elastic Stream Processing with Latency Guarantees. In *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems*, ICDCS'15, pages 399–410.

Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., et al. (2016). Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34):1–7.

Renner, T., Thamsen, L., and Kao, O. (2016). Coloc: Distributed data and container colocation for data-intensive applications. In *Big Data (Big Data), 2016 IEEE International Conference on*, pages 1–6. IEEE.

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

Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10.

Thamsen, L., Rabier, B., Schmidt, F., Renner, T., and Kao, O. (2017). Scheduling Recurring Distributed Dataflow Jobs Based on Resource Utilization and Interference. In *2017 IEEE International Congress on Big Data (BigData Congress)*, page to appear. IEEE.

Thamsen, L., Renner, T., and Kao, O. (2016a). Continuously improving the resource utilization of iterative parallel dataflows. In *Distributed Computing Systems Workshops (ICDCSW), 2016 IEEE 36th International Conference on*, pages 1–6. IEEE.

Thamsen, L., Verbitskiy, I., Schmidt, F., Renner, T., and Kao, O. (2016b). Selecting resources for distributed dataflow systems according to runtime targets. In *International Performance Computing and Communications Conference (IPCCC), 2016 IEEE 35th International Conference on*, pages 1–6. 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., and Baldeschwieler, E. (2013). Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16. ACM.

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

Warneke, D. and Kao, O. (2009). Nephele: Efficient Parallel Data Processing in the Cloud. In *Proceedings of the 2Nd Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS '09, pages 8:1–8:10. ACM.

Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. (2006). Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems De-*

*sign and Implementation*, OSDI '06, pages 307–320. USENIX.

Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., and Stoica, I. (2010a). Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278. ACM.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). 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*, NSDI'12, pages 2–2. USENIX Association.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010b). Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10. USENIX Association.

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