# Ellis: Dynamically Scaling Distributed Dataflows To Meet Runtime Targets

Lauritz Thamsen*, Ilya Verbitskiy*, Jossekin Beilharz†, Thomas Renner*, Andreas Polze†, and Odej Kao*
*Technische Universität Berlin, Germany, {firstname.lastname}@tu-berlin.de
†Hasso Plattner Institute, University of Potsdam, Germany, {firstname.lastname}@hpi.de

***Abstract*—Distributed dataflow systems like MapReduce, Spark, and Flink help users in analyzing large datasets with a set of cluster resources. Performance modeling and runtime prediction is then used for automatically allocating resources for specific performance goals. However, the actual performance of distributed dataflow jobs can vary significantly due to factors like interference with co-located workloads, varying degrees of data locality, and failures.**

**We address this problem with *Ellis*, a system that allocates an initial set of resources for a specific runtime target, yet also continuously monitors a job's progress towards the target and if necessary dynamically adjusts the allocation. For this, Ellis models the scale-out behavior of individual stages of distributed dataflow jobs based on previous executions. Our evaluation of Ellis with iterative Spark jobs shows that dynamic adjustments can reduce the number of constraint violations by 30.7-75.0% and the magnitude of constraint violations by 70.6-94.5%.**

***Index Terms*—Scalable Data Analytics, Distributed Dataflows, Dynamic Scaling, Runtime Prediction, Resource Management**

## I. Introduction

Distributed dataflow systems like Spark [1] and Flink [2] are widely used tools for processing large datasets in cloud or cluster environments. Such environments are usually shared among multiple users, running multiple jobs using different analytics frameworks. Individual users reserve a number of machines or containers for each of their dataflow jobs from a resource management system. A container in this context is an abstraction for a resource reservation, representing for example an amount of virtual cores and memory. Estimating what performance a specific reservation of entire machines or containers provides is difficult for users [3]–[5]. Consequently, users have a hard time in reserving resources for specific performance goals. Yet, they often have defined goals such as target runtimes, especially with recurring batch jobs, which make up a significant part of jobs in production clusters [4], [6]. Therefore, to meet their runtime targets, even though estimating the scale-out behavior is hard, users tend to overprovision heavily in production environments, leading to reduced cluster utilizations [7], [8].

Many systems model the performance of distributed dataflow jobs based on dedicated profiling runs [8]–[11], previous executions of recurring jobs [12], or combinations of dedicated profiling and historical information [3], [4], [13]. These models then allow to allocate resources automatically for users' performance goals. Yet, the runtime performance of distributed dataflow jobs does not only depend on the resource allocation. Other important factors include data locality [14] and competing access to data [15] as well as interference between jobs through resource contention and adjacent usage of spare resources [9]. Furthermore, there is fluctuation in runtimes due to stragglers and failures [16]–[19]. Therefore, even when running the same program on the same dataset with an equal set of containers, the runtime of jobs can vary significantly.

This paper presents *Ellis*, a system addressing this problem by continuously monitoring and dynamically adjusting resource allocations to meet runtime targets despite varying job performance. Based on our previous work on modeling the performance of distributed dataflows [12], Ellis learns the scale-out behavior of jobs from previous executions. However, in contrast to our previous work and other black-box modeling approaches for predicting the runtime of distributed dataflows [4], [11], Ellis models the scale-out behavior of individual dataflow job stages. This way, Ellis is able to effectively predict the runtime of the remaining stages while a job is running. Moreover, when the predicted runtime for the remaining stages significantly deviates from the runtime target, Ellis uses the stage-wise models to search for a scale-out that is predicted to meet the target, using more or less resources. This way, Ellis reviews the current job's progress and also submits changes to the current resource allocation at synchronization barriers between stages, effectively scaling distributed jobs dynamically. Ellis also makes an initial allocation that is predicted to meet the runtime target, so that users only need to provide a desired runtime, yet no specific resource reservations. Ellis is integrated with resource management system Hadoop YARN [20] and the distributed dataflow system Spark.

In contrast to related work on dynamic adjustments to meet targeted runtimes such as Jockey [9], Ellis can predict runtimes for multiple different distributed dataflow frameworks by using black-box modeling. Morpheus [4] is similar to Ellis in that regard, yet does not model the scale-out behavior. Instead Morpheus assumes overprovisioning and attempts to allocate as many resources as usable by distributed analytics jobs. Moreover, Morpheus also assumes control over job scheduling in addition to resource allocation.

A particularly interesting class of distributed dataflows for applying Ellis are iterative workloads. Iterative programs include not just many machine learning and graph algorithms, but also execute the same stages repeatedly, resulting in a high number of overall stages and, thus, points for adaptation.

*Contributions*. The contributions of this paper are:

- An approach for estimating the remaining runtime of distributed dataflow jobs and dynamically adjusting resource allocations to meet users' runtime targets.
- A practical implementation of our approach, which we call Ellis and which is integrated with YARN and Spark.
- An evaluation with four iterative distributed dataflow jobs, showing that our solution yields job runtimes that better meet runtime targets.

*Outline*. The remainder of the paper is structured as follows. Section II provides some background. Section III explains our approach. Section IV describes our implementation. Section V presents our evaluation. Section VI summarizes related work, while Section VII concludes this paper.

## II. Background

This section reviews the design of distributed dataflow systems. It then describes resource management and distributed file systems, which both are typically used besides distributed dataflow systems in analytical clusters.

### A. Distributed Dataflow Systems

Jobs in distributed dataflow systems consists of tasks, usually organized as a Directed Acyclic Graph (DAG). Tasks are executed by connected workers that run on a set of physical or virtual machines. These tasks are data transformation operations such as Map, Reduce, Filter, and Join. Tasks can be executed data-parallelly, so that each task instance operates on a partition of the data. A partition of the data can either be read from the input data or be received from predecessor tasks in the dataflow DAG. The number of parallel instances per task, called Degree of Parallelism (DoP), is usually configured by the user and often set to match the total number of cores on all available workers. However, if partitions are not equally distributed but skewed, which for example occurs when partitions are key-based and one key includes considerably more values, simply using a higher DoP does not necessarily decrease the execution time of jobs. This is because a dataflow DAG may contain pipeline breaking operators, such as a group-by or join operator. These operators typically need all elements of a specific group or key of the dataflow and need to wait until all predecessor tasks are finished, if the predecessor tasks did not already work on the right groups. In that case, data is re-partitioned between subsequent tasks, which is called shuffling, and which finishes when the slowest task instance finishes. That is, the slowest task instance such as one processing a considerably larger group, dictates the overall runtime of this step in the DAG. Besides data skew, tasks running on slow nodes, for instance due to misconfiguration or hardware failures, can be a reason for stragglers and similarly increase the overall job execution time. Moreover, there is also a fraction of the runtime due to steps that need to be executed sequentially, including the overhead for scheduling and starting distributed tasks.

### B. Resource Management and Distributed File Systems

Resource management systems for distributed data analysis coordinate the use of cluster resources shared by multiple jobs, often submitted from different users and possibly using different application frameworks. Users reserve a number of containers, which in this context is an abstraction for an amount of resources such as a number of cores and an amount of memory. That is, containers can vary in their size and are scheduled onto worker machines according to available capacities. Containers scheduled onto the same node may be strictly isolated, yet often are not to more efficiently share resources between distributed applications with their fluctuating resource usage over the execution time.

Distributed file systems store large files by splitting them up into blocks, which are then stored on multiple connected machines. Blocks are usually replicated for fault tolerance. Therefore, the same data is usually available on multiple machines, yet only on a few nodes in possibly large clusters. Since reading data locally is typically significantly faster than requesting data from another node, especially when the network is also used for shuffling data, scheduling containers of an application onto nodes storing parts of the input files becomes important. This is known as data locality. Consequently, distributed file systems and resource management systems usually run co-located on clusters used for data analytics.

## III. Approach

This section explains our approach, gives a formal description, and highlights the application to iterative jobs.

### A. General Idea

The scale-out behavior of distributed dataflows can be learned from example runs. Learned scale-out models then allow to predict job runtimes for specific sets of resources and, thus, automatically allocate resources for runtime targets. However, the runtime of distributed dataflow jobs varies inherently due to factors like resource congestion, different degrees of data locality, and failures. Therefore, a job's progress should be monitored continuously and resource allocations adapted dynamically to meet runtime targets.

Our approach, which we implemented as *Ellis*, is to learn the scale-out behavior of jobs from previous executions with a black-box modeling approach, using regression. The key idea is to model not entire jobs, but individual job stages, allowing to predict the runtime of distinct parts of a job for particular resource allocations. This way, Ellis evaluates the progress of a job towards a given runtime target at the synchronization barriers between stages: the system sums up the predicted runtimes of the remaining stages for the current resource allocation and compares the result with the runtime target. If the predicted remaining runtime deviates considerably from the runtime target, Ellis searches for the smallest scale-out for which the predicted runtime of the remaining stages is within the bounds of the runtime target. In addition to the runtime target, users also submit jobs with a minimal and a maximal

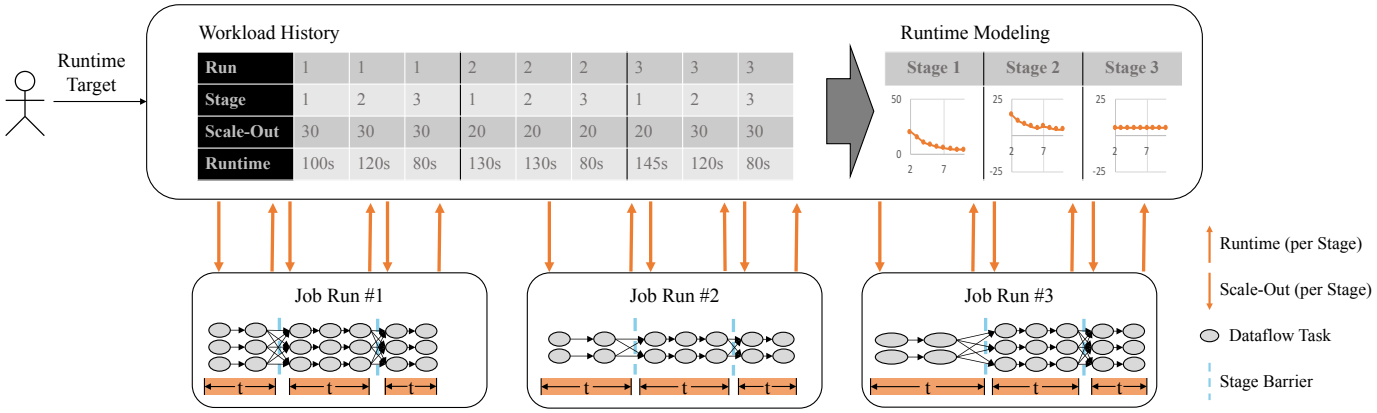


Fig. 1. Modeling the scale-out behavior of job stages to dynamically scale each stage of a recurring distributed dataflow job.

scale-out, so Ellis only searches for scale-outs within these discrete bounds.

Ellis scales resource allocations dynamically at synchronization barriers between job stages as no operator state exists at these points, which otherwise would have to be migrated between workers [21]. Still, dynamic scaling involves costs such as overheads for starting up new workers and possibly additional shuffling of the data to the new set of workers. For this reason, Ellis searches for a single scale-out for all the remaining stages, only scales resource allocations dynamically if the job's progress is significantly off track, and includes additional slack for the overhead of dynamic scaling when selecting a new scale-out.

Figure 1 depicts the central ideas of our approach. A user submits a job with a runtime target. Based on the available data on previous runs, Ellis models the scale-out behavior the job's stages and uses these models to allocate resources for each stage individually. The job has three stages and is subsequently executed three times. When the job is executed for the first time, Ellis runs the job with a scale-out that yields a runtime significantly below the user's runtime target. The next time the job is run, Ellis chooses a lower scale-out and the job more closely meets the runtime target. For the third run, Ellis again initially allocates the lower scale-out, yet the job takes longer than expected for the first stage. Therefore, Ellis selects again the higher scale-out for the two remaining stages at the first synchronization barrier, so the job still meets its runtime target.

### B. Formal Description

Let $\mathbf{x}^{(\alpha)}$ be the scale-outs previously used for stage $\alpha$ and $\mathbf{y}^{(\alpha)}$ the corresponding runtimes. We use regression to find a function $f_\alpha$ that fits this data. The procedure OnJobStart takes as input the runtime constraint $C$ as well as the scale-out constraints $x_{\min}$ and $x_{\max}$ and selects the initial scale-out.

**procedure** OnJobStart($C$, $x_{\min}$, $x_{\max}$)
  $f \leftarrow \sum_\alpha f_\alpha$
  $X \leftarrow \{x_{\min}, \ldots, x_{\max}\}$
  $D \leftarrow \{x \in X \mid f(x) < C\}$
  **if** $D \neq \emptyset$ **then**
    $x^* \leftarrow \min D$
  **else**
    $x^* \leftarrow \arg\min_{x \in X} f(x)$
  SetScaleOut($x^*$)

After each stage $\alpha$, the OnStageEnd procedure is executed. It predicts the runtime $r$ of future stages given the current scale-out $x$ and compares it to the remaining runtime $C - t$, where $t$ is the time the job has already ran. The parameters $s_{\text{tr}}$, $s_{\text{ta}}$ are the relative and absolute slack for triggering a scaling, respectively. The value $s_{\text{o}}$ compensates for any overheads that may arise due to the scaling.

**procedure** OnStageEnd($C$, $x_{\min}$, $x_{\max}$, $\alpha$, $x$, $t$, $s_{\text{tr}}$, $s_{\text{ta}}$, $s_{\text{o}}$)
  $f_{>\alpha} \leftarrow \sum_{\alpha' > \alpha} f_{\alpha'}$
  $r \leftarrow f_{>\alpha}(x)$ ▷ remaining runtime prediction
  **if** $r > (C - t) \cdot (1 + s_{\text{tr}}) + s_{\text{ta}}$ **then**
    $X \leftarrow \{x_{\min}, \ldots, x_{\max}\}$
    $D \leftarrow \{x' \in X \mid f_{>\alpha}(x') < s_{\text{o}} \cdot (C - t)\}$
    **if** $D \neq \emptyset$ **then**
      $x^* \leftarrow \min D$
    **else**
      $x^* \leftarrow \arg\min_{x' \in X} f_{>\alpha}(x')$
    SetScaleOut($x^*$)
  **else if** $r < (C - t) \cdot (1 - s_{\text{tr}}) - s_{\text{ta}}$ **then**
    $X \leftarrow \{x_{\min}, \ldots, x\}$
    $D \leftarrow \{x' \in X \mid f_{>\alpha}(x') < s_{\text{o}} \cdot (C - t)\}$
    **if** $D \neq \emptyset$ **then**
      $x^* \leftarrow \min D$
      SetScaleOut($x^*$)

### C. Application to Iterative Jobs

A particular class of distributed dataflow jobs are iterative jobs, including many widely used machine learning and graph analysis algorithms. Iterative distributed dataflow jobs often consist of many stages, since certain stages are executed repeatedly, either for a fixed number of iterations or until a convergence criteria is met. When our approach is applied to iterative jobs, the scale-out behavior of the stages of each

iteration is modeled, enabling Ellis to predict the runtime of the remaining iterations and, if necessary, adapt resource allocations in-between iterations.

Some iterative programs allow for incremental processing towards convergence. For example, many iterative algorithms allow to consider only those parts of the data that have changed in the last iteration. This fact can be used to speed up processing [22], but also yields considerably different runtimes for iterations, even though the same dataflow stages are executed. For this reason, we do not create a single model for repeatedly executed stages, but model all stages separately. This is an approximation as the convergence, including how many iterations are executed until a given convergence criteria is met, can vary considerably and depends on multiple factors such as dataset characteristics, program parameters, and the DoP.

## IV. Implementation

The main components of Ellis in context of a YARN cluster setup are depicted in Figure 2. When a client submits an application, YARN starts the first container for this application, which runs the framework-specific *App Master*. The App Master is responsible for negotiating resources with YARN's *Resource Manager* as well as for starting and monitoring the distributed application in all allocated containers. The App Master runs the submitted application's driver program, which contains Ellis' core logic for selecting scale-outs and the *Bell* system for runtime prediction [12].

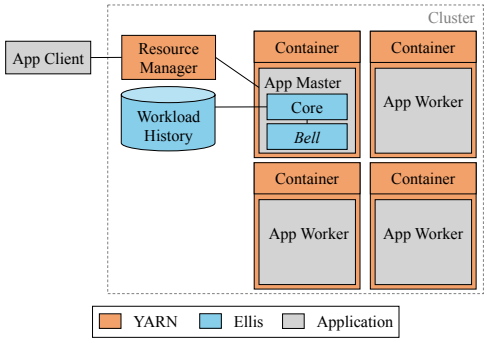


Fig. 2. System components when using Ellis with Hadoop YARN.

Ellis uses Bell to model the individual stages of a job. Bell provides black-box modeling of the scale-out behavior of distributed dataflows based on previous runs. For this, it uses regression. Furthermore, to provide predictions for varying amounts of available training data, Bell automatically chooses between two different regression models. For dense training data Bell uses nonparametric regression, which can interpolate arbitrary scale-out behavior, while for sparser training data Bell applies a simple parametric model of distributed computation, which can extrapolate the scale-out behavior from a few training samples. Bell chooses among these two models automatically using cross-validation, aiming to select the optimal model for the available training data for a specific job.

Ellis is first employed to compute an initial allocation of resources predicted to meet the given runtime target. Then, Ellis is used at all stage barriers, including the synchronization barriers between subsequent iterations. With Spark, we use a listener that is notified whenever a stage begins or finishes. This listener triggers Ellis. Ellis then predicts the runtime of the remaining stages and, if necessary, computes a new scale-out. The scale-out for the next iteration is given to a function of Spark, which requests more containers from YARN's Resource Manager if the scale-out increases and releases containers if the scale-out decreases.

Ellis models each iteration for itself as each iteration might actually behave differently even though the same stages and operators are executed, since the data may change from one iteration to the next. Iterations that are not executed due to incremental processing are stored with a runtime of zero, which is taken into account when making predictions for these iterations.

## V. Evaluation

This section presents the cluster setup, the jobs and datasets used, and our results.

### A. Cluster Setup

All experiments were done with a cluster of 50 machines. Each of the nodes is equipped with a quad-core Intel Xeon CPU 3.30 GHz (8 hardware contexts) and 16 GB RAM. All nodes are connected by a single 1 Gb switch. We used Linux (Kernel 3.10.0), Java 1.8.0, Hadoop 2.7.3, and Spark 2.1.0.

### B. Experiments

To evaluate the effectiveness of dynamic adjustments with Ellis, we compared using the system only for the initial resource allocation (non-adaptive mode) to also using it at all synchronization barriers between stages (adaptive mode). For this, we used four iterative Spark jobs and three different generated datasets. Each job used a scale-out range from 4 to 50 nodes. We started Ellis with an empty history database. Then, an initial ten runs were performed without dynamic adjustments. The runtimes of these ten runs was used as initial training data for both experiments with each job. For comparison, we did 50 non-adaptive and 50 adaptive runs.

*1) Jobs:* We used four Spark jobs as benchmarks, namely Multilayer Perceptron (MLP), Gradient Boosted Trees (GBT), Stochastic Gradient Descent (SGD), and K-Means. Table I shows the jobs and the respective input parameters. The implementation are based on Spark MLlib[1], a library for implementing distributed machine learning algorithms. All jobs were taken from the examples bundled with the framework.

*2) Datasets:* We used three different datasets for the benchmark jobs. All datasets were generated synthetically.

- *Multiclass*: A classification dataset with three classes, and 200 features. The dataset was generated using scikit-learns' classification generator[2].

[1]http://spark.apache.org/mllib/, accessed 2017-08-03
[2]http://scikit-learn.org/, accessed 2017-08-03

TABLE I
OVERVIEW OF BENCHMARK JOBS

| Job | Dataset | Input Size | Parameters |
|---|---|---|---|
| MLP | Multiclass | 29 GB | 20 iterations, 4 layers with 200-100-50-3 perceptrons |
| GBT | Vandermonde | 111 GB | 10 iterations, "Regression" configuration |
| SGD | Vandermonde | 37 GB | 20 iterations |
| K-Means | Points | 50 GB | 8 clusters, 10 iterations |

- *Vandermonde*: A regression dataset with 20 features. The dataset was generated using our own generator by explicitly computing the Vandermonde matrix. For this, data points were randomly generated following a polynomial of degree 19 with added Gaussian noise.
- *Points*: A two-dimensional dataset with points sampled from a Gaussian Mixture Model (GMM) of eight normal distributions with random cluster centers and equal variances.

### C. Results

The runtime of the 50 adaptive and non-adaptive runs of each job is summarized in Figure 3. All jobs show less spikes above the target runtime when using the adaptive approach. In addition, if there are constraint violations, their magnitude is lower.

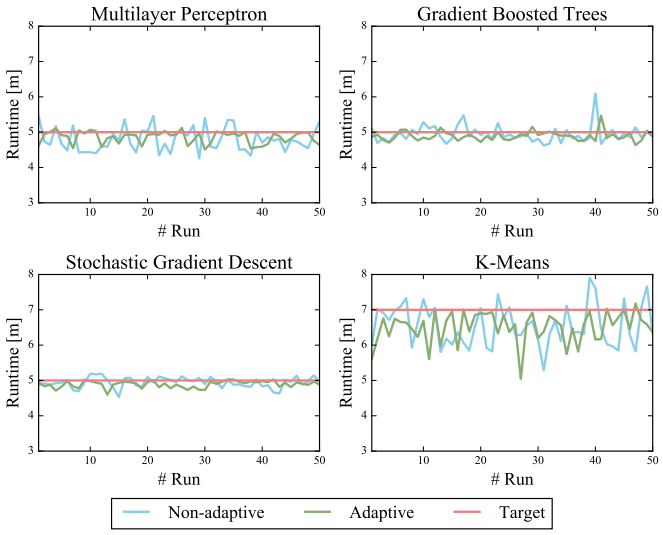


Fig. 3. Comparison of the runtimes of 50 runs of using Ellis only to allocate resources initially (non-adaptive) to also using Ellis for dynamic adjustments in-between stages (adaptive) and to the runtime target (red line).

To quantitatively assess the performance of the implementation we used three metrics. First, we capture the resource usage of a run by multiplying a stage's runtime with its scale-out and summing over all stages of the run. That is, the resource usage of a job run is defined as $\mathrm{R} = \sum_i y_i \cdot x_i$ where $x_i$ is the scale-out of stage $i$ and $y_i$ the corresponding runtime. Second, to compare how well the implementation adheres to the constraint we introduce two metrics. The constraint violation count $\mathrm{CVC} = \sum_{j \in \{j \mid Y_j > C\}} 1$ captures how often the constraint $C$ is violated by the job runtimes where $Y_j$ is the duration of the $j$-th job. Third, the constraint violation sum $\mathrm{CVS} = \sum_{j \in \{j \mid Y_j > C\}} Y_j - C$ summarizes the magnitude of the violations.

For the evaluation we calculated the ratios of the metrics for the adaptive and non-adaptive runs. The ratios are calculated by dividing the metric for the adaptive runs by the same metric for the non-adaptive runs.

TABLE II
CONSTRAINT VIOLATIONS AND RESOURCE USAGE RATIOS

| Name | R Ratio | CVC Ratio | CVS Ratio |
|---|---|---|---|
| Multilayer Perceptron | 1.2704 | 0.6923 | 0.1367 |
| Gradient Boosted Trees | 0.9985 | 0.4667 | 0.2942 |
| Stochastic Gradient Descent | 0.9475 | 0.2500 | 0.0673 |
| K-Means | 0.8003 | 0.3571 | 0.0546 |

Table II summarizes the ratios for our four benchmarks jobs. For every job the amount as well as the intensity of constraint violations is reduced. With K-Means the implementation was also able to reduce the resource usage significantly. On the other hand, for MLP the reduced constraint violations came at the cost of a higher resource usage. However, compared to the other three jobs, MLP used smaller scale-outs for the non-adaptive runs with an average of 11 nodes per run. A 27% higher resource usage then translates to an average of just three more nodes per run.

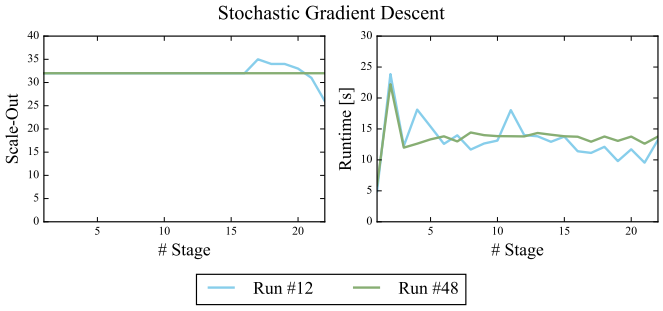


Fig. 4. Two exemplary runs of the SGD job, showing the scale-out selected for each iteration in the left chart and the corresponding runtime in the right chart.

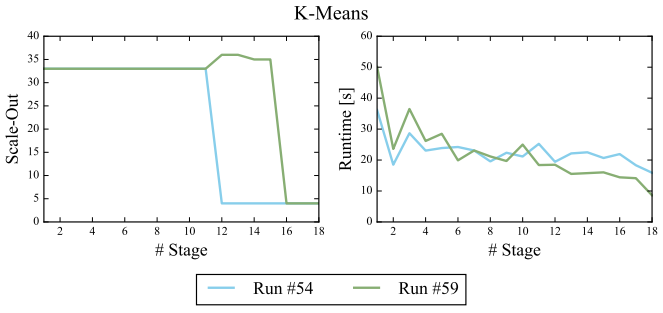


Fig. 5. Two exemplary runs of the K-Means job, showing the scale-out selected for each iteration on the left chart and the corresponding runtime on the right chart.

To give an intuition of the dynamic scalings triggered by Ellis to meet a given runtime target, Figure 4 shows two exemplary runs of SGD and Figure 5 two of K-Means. During *Run #12* of SGD Ellis increased the scale-out after the 16th stage, yet released some of the allocated resources after the 20th stage. In contrast, Ellis did not make any adjustments to the resource allocation during *Run #48*. In comparison it is visible that the dynamic scaling in *Run #12* clearly had an effect on the runtime. The adjustments were probably triggered to compensate for the spikes in the 4th and 11th stage and the run did finish with an overall runtime below the runtime target. The two exemplary runs of K-Means shown in Figure 5 both release a majority of the resources towards the end of the runs. The graph shows that later stages run faster than earlier stages even with considerably less resources, displaying the converging behavior of the algorithm and explaining the low R ratio reported for K-Means.

## VI. Related Work

This section first presents related work on distributed dataflow systems. It then describes work on runtime prediction and automatic resource allocation comparable to Ellis.

### A. Distributed Dataflow Systems

MapReduce [23] introduced a programming model based on the second-order functions Map and Reduce. Users provide user-defined functions (UDFs) to these operators, which are then executed in parallel on connected distributed workers. The execution model is based on the alternating execution of the two operations. First, the Map operation is executed data-parallelly, then Reduce is executed data-parallelly. The intermediate results between these two phases are stored to disk for fault-tolerance. The Map tasks sort the results by key, so that the Reduce tasks can efficiently read and reduce defined groups of data.

Dryad [24] and Nephele [25] also execute data-parallel tasks, yet allow to connect these in a general DAG. These tasks can be connected by different channels such as, for example, through network channels or disk-based. Both systems do not provide predefined operators like Map and Reduce.

SCOPE [26] allows to use a general DAG for dataflow programs as well, yet furthermore provides a rich set of pre-defined operators, including for example also joins. Moreover, SCOPE incorporates optimizer techniques from parallel databases to generate optimized query plans [27]. Consequently, developers can use a high-level scripting language similar to SQL to write SCOPE programs.

Similar to SCOPE Stratosphere [28] also allows to connect operations from a pre-defined set of operators to a general dataflow graph [29] and uses optimization techniques to generate query plans from a high-level scripting language [30]–[32]. Stratosphere furthermore supports incremental processing of converging iterative dataflow programs natively [22], effectively allowing cyclic dataflow graphs.

Naiad [33] is a similar system in that it too supports incremental processing of converging iterative programs natively, yet Naiad can also incrementally compute results when inputs change.

Spark [1] is similar to SCOPE and Stratosphere, also allowing to create dataflow programs using operators like Map, Reduce, and Join as well as connect these to general acyclic job graphs. Spark's key feature is its abstraction for distributed datasets called Resilient Distributed Datasets (RDDs) [34]. RDDs use linage for fault tolerance and, thus, execute at in-memory speed in the absence of faults. Furthermore, RDDs can be cached for faster interactive and iterative processing. Spark Streaming [35] is a distributed streaming system based on the batch processing engine of Spark, using micro-batches.

Flink [2] build upon the Stratosphere stack, but uses a streaming engine for both stream and batch processing. It provides low latency stream processing, including exactly-ones-guarantees, but the same engine and interfaces can be also be used for distributed batch processing.

Google's Dataflow [36] is a distributed stream processing system, which too can be used for batch processing. Dataflow is unique in that in provides mechanisms to explicitly handle elements arriving late.

Given this considerable number of similar data analytics systems, Tez [37] was developed as a framework for building distributed dataflow systems.

### B. Runtime Prediction and Automatic Resource Allocation

Systems for purely estimating the progress and the runtime of distributed dataflow jobs include Parallax [38], ParaTimer [39], and PREDIcT [40]. Parallax and ParaTimer, for example, estimate the runtime of MapReduce jobs. Parallax predicts the runtime of sequences of MapReduce jobs compiled from Pig programs [41]. The runtime prediction of Parallax uses a simple model of parallelism, cardinality estimates based on standard optimizer techniques, and profiling runs on user-defined samples of the input data. ParaTimer builds upon Parallax. ParaTimer allows a general DAG of MapReduce jobs, not just sequences. For this, ParaTimer identifies the critical path in a DAG and estimates the runtime of that, effectively ignoring all other paths. ParaTimer also handles skew and failures by providing a set of estimates for different possible scenarios. PREDIcT [40] provides runtime prediction not just for MapReduce, but for iterative distributed dataflow jobs that operate on homogeneous graph structures and have a global convergence condition. It uses a profiling run on a sample of the input data and a custom function to extrapolate to the entire input data. In comparison, Ellis also predicts runtimes, yet specifically for automatically allocating resources to meet given runtime targets.

There are multiple systems that use runtime prediction to allocate resources according to a given runtime target. Examples of systems that use a MapReduce-specific model include Aria [13], [42], Elastisizer [43], [44], Bazaar [10], and AROMA [3]. More generic solutions that support multiple different distributed dataflow systems include Ernest [11], Bell [12], and PerfOrator [5]. Ernest uses profiling runs to train a simple parametric model of distributed computing and

aims at automatically choosing optimal sample configurations. Bell [12] automatically chooses between parametric regression and nonparametric regression to provide optimal runtime prediction for recurring jobs from the available workload history. PerfOrator [5] models the performance of jobs using non-linear regression on profile runs for automatic resource allocation. Yet, PerfOrator requires framework models as input to accurately capture parallelism and, therefore, is not directly applicable to distributed dataflow systems in general, supporting at the time of writing MapReduce and Tez. These differ from Ellis in that Ellis, if necessary, adjusts allocations at runtime.

Systems that automatically allocate resources for user's performance goals, but then like Ellis also continuously monitor progress at runtime and adjust allocations include Jockey [9], Morpheus [4], and Quasar [8]. Jockey automatically adapts resource allocations at runtime based on the predicted remaining runtime, but in contrast to Ellis uses a framework-specific model for SCOPE and requires detailed job statistics. Morpheus estimates the resources required to optimally run a recurring jobs from resource utilization data of previous runs, yet unlike Ellis effectively assumes overprovisioning and also control over job scheduling. Quasar incorporates scale-out, scale-up, and interference between jobs into performance models using both previous runs and dedicated sample runs. Quasar then jointly performs resource allocation and assignment, before continuously monitoring progress and, if necessary, adjusting resource allocations dynamically. Quasar does, however, assume full control over resource allocation for this and does not scale dynamically based on predicted runtimes, yet based on the results of online reclassification, in contrast to Ellis.

## VII. Conclusion

This paper presented Ellis. Ellis models the scale-out behavior of the stages of distributed dataflows based on previous executions of a job, allowing to predict the runtime of the remaining stages of a running job with different sets of cluster resources. This way, Ellis assesses whether a running job is predicted to meet its runtime target with its current scale-out and, if not, selects the smallest scale-out, for which the runtime of the remaining stages is predicted to be within the bounds of the given target. Thereby, Ellis addresses the inherent variance in job runtimes and compensates for factors like competing access to shared resources, varying degrees of local access to input data, as well as stragglers and failures. Furthermore, Ellis also allocates resources initially. That is, users are released from the task of essentially guessing a set of resources for their performance goals and dynamic adjustments are used as an alternative to overprovisioning significantly to meet runtime targets despite runtime variance.

Since Ellis uses a black-box approach for modeling the behavior of job stages and is integrated with the resource management system YARN, it can be used with different distributed dataflow frameworks.

Ellis is best applied for iterative jobs, which repeatedly execute the same stages, resulting in a high overall number of stages and, consequently, opportunities to assess a job's progress towards its runtime target as well as, if necessary, dynamically adjust resource allocations.

In the future, we want to improve the support for converging iterative programs by matching the currently executing job at runtime to previous executions based on similarity. Besides stage runtime, similarity criteria could, for example, include the number of active records and resource utilization to match previous runs with a similar convergence. Moreover, since dynamic scaling introduces overheads due to, for example, worker startup, but also possibly necessary additional repartitioning of intermediate data across the new set of workers, we want to investigate using a more detailed cost model when deciding to adapt resource allocations at runtime. Nevertheless, our evaluation with four iterative Spark jobs shows that Ellis is already effective in reducing runtime constraint violations and does not use a lot more resources for this in the average case.

## Acknowledgments

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

## References

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. USENIX Association, June 2010, pp. 10–10.

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

[3] P. Lama and X. Zhou, "AROMA: Automated Resource Allocation and Configuration of Mapreduce Environment in the Cloud," in *Proceedings of the 9th International Conference on Autonomic Computing*, ser. ICAC '12. ACM, September 2012, pp. 63–72.

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

[5] K. Rajan, D. Kakadia, C. Curino, and S. Krishnan, "PerfOrator: Eloquent Performance Models for Resource Optimization," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, ser. SoCC '16. ACM, 2016, pp. 415–427.

[6] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou, "Reoptimizing Data Parallel Computing," in *In Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, April 2012, pp. 281–294.

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

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

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

[10] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Bridging the Tenant-provider Gap in Cloud Services," in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC '12. ACM, October 2012, pp. 10:1–10:14.

[11] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient Performance Prediction for Large-scale Advanced Analytics," in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, ser. NSDI'16. USENIX Association, March 2016, pp. 363–378.

[12] L. Thamsen, I. Verbitskiy, F. Schmidt, T. Renner, and O. Kao, "Selecting resources for distributed dataflow systems according to runtime targets," in *2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)*. IEEE, December 2016, pp. 1–8.

[13] A. Verma, L. Cherkasova, and R. H. Campbell, "ARIA: Automatic Resource Inference and Allocation for Mapreduce Environments," in *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ser. ICAC '11. ACM, June 2011, pp. 235–244.

[14] T. Renner and L. Thamsen and O. Kao, "CoLoc: Distributed Data and Container Colocation for Data-intensive Applications," in *2016 IEEE International Conference on Big Data (Big Data)*, ser. BigData 2016. IEEE, Dec 2016, pp. 3008–3015.

[15] Ananthanarayanan, Ganesh and Agarwal, Sameer and Kandula, Srikanth and Greenberg, Albert and Stoica, Ion and Harlan, Duke and Harris, Ed, "Scarlett: Coping with Skewed Content Popularity in Mapreduce Clusters," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11. ACM, April 2011, pp. 287–300.

[16] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta, "On Availability of Intermediate Data in Cloud Computations," in *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, ser. HotOS'09. USENIX Association, May 2009, pp. 6–6.

[17] G. Wang, A. R. Butt, P. Pandey, and K. Gupta, "A Simulation Approach to Evaluating Design Decisions in MapReduce Setups," in *2009 IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems*. IEEE, Sept 2009, pp. 1–11.

[18] K. V. Vishwanath and N. Nagappan, "Characterizing Cloud Computing Hardware Reliability," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. ACM, June 2010, pp. 193–204.

[19] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective Straggler Mitigation: Attack of the Clones," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'13. USENIX Association, April 2013, pp. 185–198.

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

[21] L. Thamsen, T. Renner, and O. Kao, "Continuously Improving the Resource Utilization of Iterative Parallel Dataflows," in *Proceedings of the 6th International Workshop on Big Data and Cloud Performance*, ser. DCPerf 2016. IEEE, June 2016, pp. 1–6.

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

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

[24] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-parallel Programs from Sequential Building Blocks," in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. ACM, March 2007, pp. 59–72.

[25] D. Warneke and O. Kao, "Nephele: Efficient Parallel Data Processing in the Cloud," in *Proceedings of the 2Nd Workshop on Many-Task Computing on Grids and Supercomputers*, ser. MTAGS '09. ACM, November 2009, pp. 8:1–8:10.

[26] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1265–1276, August 2008.

[27] J. Zhou, N. Bruno, M.-C. Wu, P.-A. Larson, R. Chaiken, and D. Shakib, "SCOPE: Parallel Databases Meet MapReduce," *The VLDB Journal*, vol. 21, no. 5, pp. 611–636, October 2012.

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

[29] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke, "Nephele/PACTs: A Programming Model and Execution Framework for Web-scale Analytical Processing," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. ACM, June 2010, pp. 119–130.

[30] F. Hueske, M. Peters, M. J. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas, "Opening the black boxes in data flow optimization," *Proc. VLDB Endow.*, vol. 5, no. 11, pp. 1256–1267, July 2012.

[31] A. Heise, A. Rheinländer, M. Leich, U. Leser, and F. Naumann, "Meteor/Sopremo: An Extensible Query Language and Operator Model," in *Proceedings of the International Workshop on End-to-end Management of Big Data (BigData) in conjunction with VLDB 2012*, ser. BigData '2012. VLDB Endowmen, August 2012, pp. 1–10.

[32] A. Rheinländer, A. Heise, F. Hueske, U. Leser, and F. Naumann, "SOFA," *Information Systems*, vol. 52, no. C, pp. 96–125, August 2015.

[33] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A Timely Dataflow System," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, November 2013, pp. 439–455.

[34] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. USENIX Association, April 2012, pp. 2–2.

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

[36] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1792–1803, August 2015.

[37] Saha, Bikas and Shah, Hitesh and Seth, Siddharth and Vijayaraghavan, Gopal and Murthy, Arun and Curino, Carlo, "Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. ACM, June 2015, pp. 1357–1369.

[38] K. Morton, A. Friesen, M. Balazinska, and D. Grossman, "Estimating the Progress of MapReduce Pipelines," in *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, ser. ICDE 2010. IEEE, March 2010, pp. 681–684.

[39] K. Morton, M. Balazinska, and D. Grossman, "ParaTimer: A Progress Indicator for MapReduce DAGs," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. ACM, June 2010, pp. 507–518.

[40] A. D. Popescu, A. Balmin, V. Ercegovac, and A. Ailamaki, "PREDIcT: Towards Predicting the Runtime of Large Scale Iterative Analytics," *Proc. VLDB Endow.*, vol. 6, no. 14, pp. 1678–1689, September 2013.

[41] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A Not-so-foreign Language for Data Processing," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08. ACM, June 2008, pp. 1099–1110.

[42] A. Verma, L. Cherkasova, and R. H. Campbell, "Resource Provisioning Framework for Mapreduce Jobs with Performance Goals," in *Proceedings of the 12th ACM/IFIP/USENIX International Conference on Middleware*, ser. Middleware'11. Springer, December 2011, pp. 165–186.

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

[44] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: A Self-tuning System for Big Data Analytics," in *Proceedings of the the 5th Conference on Innovative Data Systems Research*, ser. CIDR '11. CIDR 2011, January 2011, pp. 261–272.