Selecting Resources for Distributed Dataflow Systems According to Runtime Targets

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

***Abstract*—Distributed dataflow systems like Spark or Flink enable users to analyze large datasets. Users create programs by providing sequential user-defined functions for a set of well-defined operations, select a set of resources, and the systems automatically distribute the jobs across these resources. However, selecting resources for specific performance needs is inherently difficult and users consequently tend to overprovision, which results in poor cluster utilization. At the same time, many important jobs are executed recurringly in production clusters.**

This paper presents *Bell*, a practical system that monitors job execution, models the scale-out behavior of jobs based on previous runs, and selects resources according to user-provided runtime targets. Bell automatically chooses between different runtime prediction models to optimally support different distributed dataflow systems. Bell is implemented as a job submission tool for YARN and, thus, works with existing cluster setups. We evaluated Bell's runtime prediction with six exemplary data analytics jobs using both Spark and Flink. We present the learned scale-out models for these jobs and evaluate the relative prediction error using cross-validation, showing that our model selection approach provides better overall performance than the individual prediction models.

***Index Terms*—Scalable Data Analytics, Distributed Dataflows, Runtime Prediction, Resource Allocation, Cluster Management**

I. Introduction

Distributed data-parallel processing systems like MapReduce [1], Spark [2], and Flink [3] allow users to analyze large datasets in parallel using clusters of computers. Users often use multiple of these systems: different ones for different tasks and use cases. Multiple jobs from different analysis frameworks consequently run side-by-side in shared clusters, managed by resource management systems like YARN [4] and Mesos [5].

With these frameworks users essentially create programs from sequential building blocks, which are then automatically parallelized and distributed. Thus, the frameworks handle both task parallelization and distribution. Yet, users still need to specify the amount of resources the jobs should be executed on. Predicting the runtime behavior that a specific resource allocation yields for a given analysis job is, however, inherently difficult. This is due to the many factors the performance depends on such as user-defined functions (UDFs), task dependencies, data characteristics, system configurations, and the execution environment. For example, without detailed knowledge of data characteristics such as key value distributions, it is unclear how well the data partitioning will work for a specific level of parallelism. For this reason, studies of production clusters often show that users tend to overprovision significantly to ensure minimal performance goals, leading to underutilized clusters. For example, a utilization analysis for a data analysis cluster at Twitter [6] shows that the aggregate CPU utilization was consistently below 20%, even though the reservations reached close to 80% of the total capacity. Memory utilization for the cluster was between 40–50% but still differed significantly from the reserved capacity. Similarly, a production cluster at Google managed by the company's Borg system [7] achieved aggregate CPU utilization of 25–35% and aggregate memory utilization of 40%, while resource reservations exceeded 75% and 60% of the available CPU and memory capacities [8].

At the same time, many batch jobs are executed periodically on updated or similar datasets [9]–[11]. For example, it was reported that recurring jobs make up 40.32% of the jobs as well as 39.71% of the cluster hours for a production cluster used at Microsoft [10]. These recurring jobs were also noted as the ones for which users have particular performance requirements [11]. Moreover, data-parallel processing jobs often exhibit relatively predictable scaling behavior—often close to linear speedup for many of the use cases the systems were built for [12]–[16]. Therefore, previous job executions can be used to model the performance of jobs depending on provided resources, which in turn allows to select resources for specific performance targets based on these models. Then, users only need to specify their target runtimes instead of having to select specific sets of resources themselves.

Previous efforts based on this idea make at least one of three assumptions. First, a lot of these systems were designed for specific processing frameworks [9], [11], [17]–[20]. Second, multiple approaches require dedicated isolated training runs [6], [19], [21]. Third, some solutions go beyond resource allocation and also assume control over, for instance, job execution order [9] or container placement [6]. Yet, even for a single framework, there is some inherent variance in job runtime in shared clusters due to factors like data locality, caching, hardware failures, and interference between jobs [11], [22]–[25]. These factors render even detailed models built from isolated profiling runs imprecise. Moreover, incorporating lots of statistics like detailed data statistics into estimation models requires extensive instrumentation. Such instrumentation can impose additional overheads on job execution. Switching to entirely different schedulers or resource managers on the other hand is often just impractical for organizations.

This paper presents *Bell*, a practical system that models

the scale-out behavior of jobs using previously observed runs and performs automatic resource allocation based on these models. Bell is a general solution for multiple analytics frameworks, acknowledging the fact that users need to choose the best tool for the task at hand. Bell is also not based on dedicated isolated training runs, but aims to make the most of available historic data. For this, Bell uses regression to learn how the runtime of a job depends on the scale-out. To be able to effectively learn the scale-out behavior of different distributed dataflow frameworks and jobs, Bell automatically selects between different scale-out models. In particular, Bell uses either parametric or nonparametric regression depending on the available workload data and the prediction task. For parametric regression, Bell utilizes a simple model of distributed computation, which is applicable to distributed dataflows and can provide reasonable predictions from a few data points. Nonparametric regression on the other hand allows to interpolate more arbitrary scale-out behaviors from many data points. Since we implemented Bell as a job submission tool for YARN, it can be used with existing Hadoop clusters, including different schedulers.

Contributions. The contributions of this paper are:

- A broadly applicable solution to predicting the runtime of distributed dataflow jobs from available workload data, which selects a specific prediction model automatically.
- A practical application of the runtime prediction in a job submission tool for YARN, which we call Bell, which selects resources according to user-defined performance requirements, and which can be used with existing setups.
- An evaluation of our approach to runtime prediction using six exemplary data analysis jobs—three in Spark and three in Flink—showing how different models perform for different sets of available previous runtimes.

Outline. The remainder of the paper is structured as follows. Section II provides background on distributed dataflow systems and resource management for these. Section III presents our approach to runtime prediction and resource selection for different distributed dataflow systems. Section IV shows our evaluation. Section V discusses related work. Section VI concludes this paper.

II. Background

This section describes how distributed dataflow systems process large datasets and how resource managers enable users to utilize different dataflow systems in single shared clusters.

A. Distributed Dataflow Systems

Distributed dataflow systems process data through graphs of tasks. Figure 1 shows such a graph. The tasks are configured versions of pre-defined operators. These operators include Map and Reduce, which both execute UDFs. Specific variants of these two operators are operators like Filter and pre-defined aggregations for computing, for example, sums. Two dataflows can be combined using operators like Join or Cross.

Each of the tasks is executed data-parallelly. That is, each task instance processes a partition of the data. In the dataflow graph in Figure 1, each task has two data-parallel instances each. A partition can be created by reading parts of the input from disk, for example from a distributed file system. It can also be received from a predecessor task in the dataflow graph. Operators for group-based aggregations or joining two dataflows require all elements of the same group or with the identical join key to be available at the same task instance. Therefore, if the data is not already partitioned by these keys, the dataflow needs to be shuffled: all elements with the same key need to be moved to the same task instance, leading to all-to-all communication. This pattern of communication is visible before the Join and Reduce tasks in Figure 1. Other data exchange patterns include all-to-one and one-to-one communication.

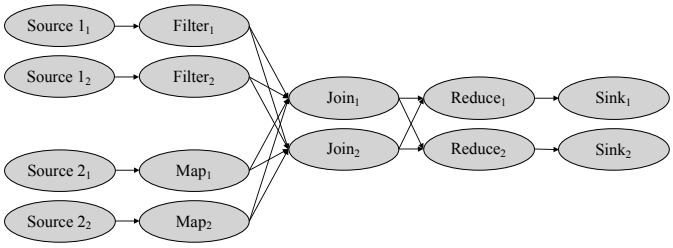

Fig. 1. A dataflow graph with two data-parallel instances for each task.

Task instances run on networked worker nodes. Each worker provides execution slots, representing the compute capabilities. Such a slot can execute a task or a chain of tasks. Except for certain operators, chained tasks can in principle be executed in parallel, adding pipeline parallelism. Pipelining breaking operators are those that require all elements with certain keys to be available.

How many parallel instances of each task are executed is usually decided by the user as the Degree of Parallelism (DoP). However, all instances need to receive data. Especially if partitions are key-based, the parallelism is thus limited by the number of different values. Furthermore, if value distributions are not equal but skewed, using a higher DoP also does not necessarily lead to decreased runtimes. This is because pipeline breaking operators basically synchronize all parallel threads of the dataflow. As these operators need to wait for all elements of a group or key, they need to wait for all predecessor tasks to finish. Consequently, the slowest task instances determine the overall runtime.

For some operators there are multiple implementation strategies, which require different synchronization and communication. For example, if two dataflows should be joined, one side can be broadcast to all parallel task instances, allowing the other side to be pipelined without any shuffling. This strategy is, however, only feasible when one side is relatively small. Another strategy involves shuffling both sides, so all elements with the same key are received by the same task instances.

B. Resource Management Systems

Resource management systems allow fine-grained sharing of cluster resources. Instead of permanently running a single data-processing framework on cluster nodes, a resource management system runs permanently while individual frameworks run on a per-job basis. Users make reservations for their jobs in terms of containers. Containers are leases of resources, bound to a specific node and allocated to particular jobs. Containers bundle a number of cores and an amount of main memory. Depending on the size of containers and node capabilities multiple containers can run on a single node. Figure 2 shows managed cluster nodes that host containers of two different applications.

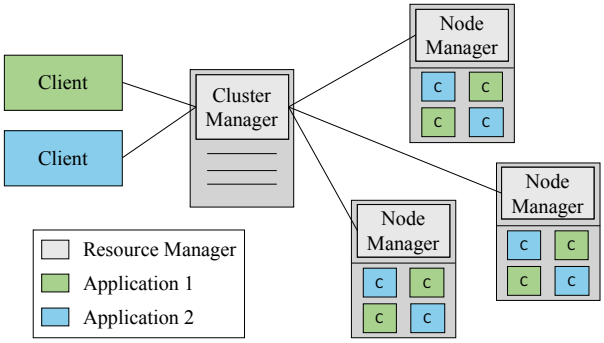

Fig. 2. Managed cluster nodes hosting containers of two applications.

Containers can also provide resource isolation. Without resource isolation though, they only represent computing capabilities. Therefore, a worker running in a container can use more resources than reserved with the container. Different workers that run in containers on the same node can consequently interfere with each other. At the same time, this allows statistical multiplexing as jobs often do not stress resources continuously.

Which jobs are co-located in this manner is decided by the scheduler component of the resource manager. Often resource manager allow to use multiple different schedulers that focus on different scheduling goals like fairness, throughput, or data locality.

III. Runtime Prediction and Resource Allocation

This section presents how Bell is integrated with existing systems and how Bell predicts job runtimes to choose resources for user-provided runtime targets.

A. System Overview

Bell automatically allocates resources according to user-defined performance targets. In particular, users submit their jobs along with a runtime target and minimal and maximal scale-out constraints to Bell. Bell then translates these arguments into a scale-out to be used for the job. For this, Bell models the scale-out behavior of a job based on previously observed runtimes. Using these models, Bell selects the minimal scale-out with a shorter predicted runtime than the target.

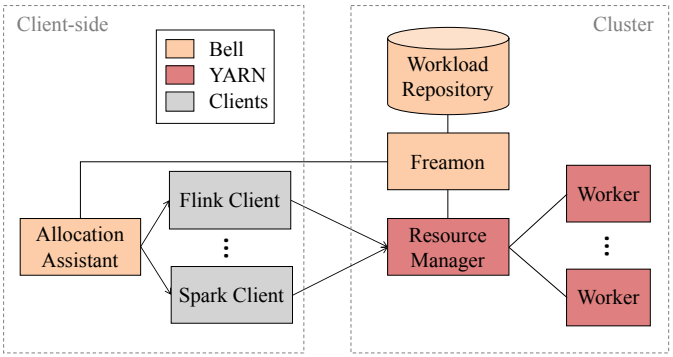

Fig. 3. Bell's main components and integration with YARN, Spark, and Flink.

Figure 3 shows how Bell is integrated with existing systems. Bell is essentially a tool for submitting jobs to YARN. It wraps existing clients and converts arguments expressing user-level performance requirements to arguments expressing a concrete reservation. In particular, Bell's *Allocation Assistant* component translates a target runtime into a reservation of containers while respecting scale-out limits. Users can provide a fallback reservation to be used when not enough previous runs are available to effectively model the scale-out behavior of a job. For previous runs, Bell takes those executions into account that ran the same program on input datasets of a similar size. This requires basic information on previously executed jobs to be available. For this, Bell uses a component we call *Freamon*. Freamon monitors the execution of YARN applications. For each job, Freamon stores basic information on both the program and the inputs, including the hash of the JAR file and the size of input datasets. As previous executions of a job, Bell uses all jobs with the same JAR that processed datasets within 10% of the sizes of the current inputs.

Implementing Bell as a wrapper around YARN clients allows the system to be used for different processing systems that run on YARN. Currently Bell supports Spark and Flink, but adding more clients is straightforward. Furthermore, this design decision allows Bell to be added to existing YARN setups, including different YARN schedulers.

B. Runtime Prediction Models

To create a runtime behavior model, Bell uses parametric and nonparametric regression. Formally, regression is the task to predict some target $\hat{y}$, given the input x such that $\hat{y}$ is close to the true value y given some error measure. Such a regression model is represented by a function $f : X \to Y$, learned by the applied regression algorithm. Parametric regression requires a parameterized model to be fitted by choosing the optimal weights, while nonparametric regression constructs the model from the data. Bell stores data for each executed job, including the job's scale-out and runtime. This data is used as training data for the regression models.

1) Parametric Regression: Given training data and a parameterized model, parametric regression learns the model's parameters such that the model optimally fits the training

data regarding some error measure. As model for parametric regression, Bell uses Equation 1, which is based on the model for distributed processing presented in [21].

$$f = \theta_0 + \theta_1 \cdot \frac{1}{\mathit{containers}} + \theta_2 \cdot \log(\mathit{containers}) + \theta_3 \cdot \mathit{containers} \quad (1)$$

Equation 1 consists of four additive terms, each representing aspects of parallel computation and communication of a certain number of *containers*. In particular and in the order shown in the equation, the four terms represent: serial computation, parallel computation, communication patterns for step-wise aggregation, and overheads that scale with the number of containers such as from all-to-one communication. Since all four terms represent costs, the parameters are chosen such that they are non-negative. Therefore, the model is estimated using non-negative least square (NNLS).

2) Nonparametric Regression: To enhance the capabilities of Bell, we also included nonparametric regression, since parametric regression based on the given model might not fit all possible scale-out behaviors well. Nonparametric regression infers the model automatically by assuming defined local behavior in the dataset. Bell estimates the regression function through local linear kernel regression (LLKR) with Gaussian kernel [26]. The kernel width is selected via cross-validation. In order to apply LLKR, a dense training dataset is needed due to the locally optimized fitted curves.

C. Resource Allocation

For resource allocation, Bell selects one of the two regression models described above. Since nonparametric regression requires a dense set of training samples, using parametric regression might provide better results when only few historical data points are available. Therefore, Bell first performs a model selection strategy to decide which prediction model to use.

1) Model Selection: Bell selects one of the two described regression learners using cross-validation. By repeating a fixed number of steps, first the models are learned and then evaluated. The best performing model is selected for the process of choosing the number of resources.

This is done by first selecting n data points where failed runs and extreme outliers are removed from the historical dataset. Assuming $k+2$ unique scale-outs in the data set, Bell performs k-fold cross-validation where each test fold contains all points of a single scale-out. Since the aim is to assess the interpolation performance, the 2 test folds for the smallest and largest scale-out are omitted. As a last step, the model with the smallest cross-validation error is selected for interpolation.

Note that nonparametric regression is only usable for interpolation and not for extrapolation, due to the local optimization criterion. Therefore, Bell only performs parametric regression for estimating the runtime for lower or higher number of resources than previously used.

2) Resource Allocation Approach: As Bell selects the number of resources to be allocated, it must conform to the user-provided constraints. These constraints are the maximum running time for the job as well as a minimum and a maximum for the scale-out. The scale-out limits can be provided by the user or can be inferred by the system. Natural limits are the maximal resources the system can provide and at least a single resource. Given these constraints and regression model, Bell uses a straightforward greedy approach by choosing the smallest number of resources satisfying the constraints.

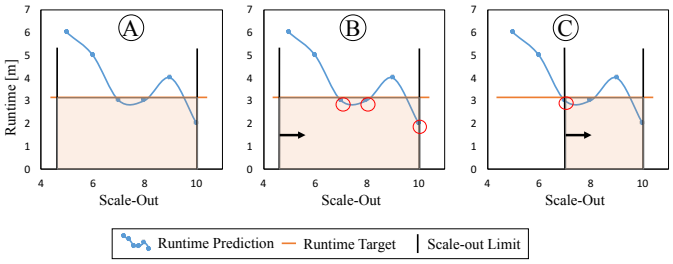

Fig. 4. Process of selecting resources in Bell.

Figure 4 depicts the process of selecting resources. The initial conditions are shown in Ⓐ. As shown in Ⓑ, we intercept the constraints with the runtime prediction function to create a set of potential recommendations. From the resulting set of potential recommendations the smallest set of resources is selected, assuming resources have high costs. This solution can be computed by moving the minimum resources constraint in the direction of the maximum as shown in Ⓒ. For each of the discrete scale-outs we compute the predicted runtime, selecting the first that is in the set of potential recommendations.

IV. Evaluation

This section presents our experimental setup, the test workload, and benchmark results.

A. Experimental Setup

All experiments were done using a cluster of 60 machines. Each of the nodes is equipped with a quad-core Intel Xeon CPU 3.30 GHz (4 physical cores, 8 hardware contexts), 16 GB RAM, and three 1 TB disks (RAID 0). All nodes are connected through a single switch and 1 Gigabit Ethernet. Each node runs Linux (Kernel 3.10.0) and Java 1.8.0. We used Spark 2.0.0 (GraphX 1.6.0 and MLLib 1.1.0), Flink 1.0.3, and Hadoop 2.7.1.

B. Test Workload

We evaluated Bell using six jobs and five datasets. The jobs cover different domains: search, relational queries, graph processing, and machine learning. The datasets cover different types of data, various data sizes, and range from simple to realistic data characteristics.

1) Jobs: We used six different benchmark jobs, three Spark jobs and three Flink jobs, as shown by Table I. For Stochastic Gradient Descent (SGD) we used the implementation of Spark's MLib[1], a library for scalable machine learning. For PageRank we used GraphX [14], a graph processing system built on top of Spark. All other implementations are taken from the examples provided with the frameworks.

TABLE I
OVERVIEW OF BENCHMARK JOBS

Job	System	Dataset	Input Size	Parameters
Grep	Spark	Wiki	250 GB	filtering for word "Berlin"
WordCount	Flink	Wiki	250 GB	—
TPC-H Query 10	Flink	Tables	200 GB	—
K-Means	Flink	Points	50 GB	5 clusters, 10 iterations
SGD	Spark	Features	10 GB	100 iterations, step size = 1.0
PageRank	GraphX	Graph	3.4 GB	5 iterations

2) Datasets: We used five different data generators. For graph and text data, we used generators from the Big Data Generator Suite (BDGS) [27], which effectively scale real datasets while preserving key characteristics of the data. For relational data we used the data generator of the TPC-H benchmark suite[2]. In addition, we implemented two generators, one for three-dimensional points and one for multi-dimensional feature vectors. We generated the following five datasets:

- *Wiki*: The Wiki dataset was generated using the text generator of BDGS, which applies latent dirichlet allocation (LDA) [28] to create large datasets based on articles from the English Wikipedia, while preserving topic and word distributions. We created 250 GB of text data.
- *Graph*: The Graph dataset was generated with the graph generator of BDGS, which uses the Kronecker graph model [29] and a real graph of linked Web pages. Using 25 Kronecker iterations we created a 3.4 GB large graph with 33,554,432 nodes and 213,614,240 directed edges.
- *Tables*: The Tables dataset was generated using the TPC-H data generator, which generates tables representing customers, nations, orders, and line items. We set the Scale Factor to 200, resulting in around 200 GB of table data.
- *Features*: The Features dataset was generated using our own generator, explicitly creating a Vandermonde matrix to generate multi-dimensional feature vectors that fit a polynomial model of a certain degree with added Gaussian noise. We generated 20,000,000 points, each with 20 features, yielding 10 GB.
- *Points*: The Points dataset was generated using our own generator to produce 4,216,562,650 three-dimensional points following a Gaussian mixture model (GMM) of five normal distributions with random cluster centers and equal variances, resulting in 50 GB.

[1]http://spark.apache.org/mllib/, accessed 2016-08-26
[2]http://www.tpc.org/tpch/spec/tpch2.16.0v1.pdf, accessed 2016-08-25

C. Benchmark Results

To assess the prediction performance, we acquired runtime data for the six benchmark jobs. In particular, each job was executed using 15 different and equally spaced scale-outs ranging from 4 to 60 nodes. For every scale-out the job was run 7 times, out of which we dropped the fastest and the slowest runs, resulting in a total of 75 data points per job.

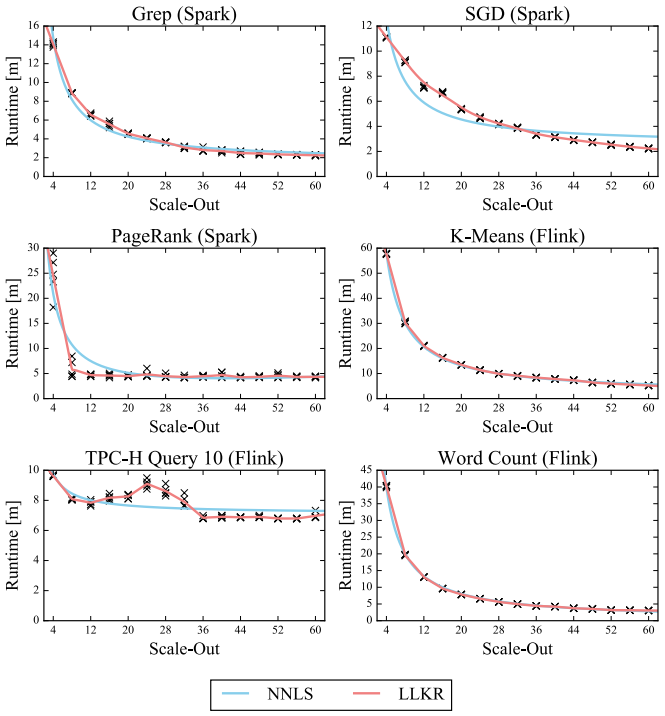

Fig. 5. Runtimes of 75 runs of the six benchmark jobs along with the fitted curves, where NNLS and LLKR are the parametric and nonparametric models, respectively.

To begin with, every dataset was fitted using the parametric and nonparametric regression models. Figure 5 summarizes the results for each of jobs. The parametric model provides a good fit for Grep, K-Means, and Word Count. Yet, it falls short when jobs exhibit more complicated runtime behaviors. This is the case for SGD, PageRank, and TPC-H Query 10. In these cases, the nonparametric model provides the better fit.

However, while the nonparametric model seems superior in the presence of lots of data, it might suffer from high variance when interpolation is done using only a few data points. For this reason, we evaluated the prediction performance of the models with different numbers of available training data points. In particular, for each model and number of training data points we calculated the mean relative prediction error using random sub-sampling cross-validation. For every fixed amount of training data points, random training points are selected from the dataset such that the scale-outs of the data points are pairwise different. Then, to perform an interpolation benchmark, a test point is randomly selected such that its scale-out lies in the range of the training points. The runtime prediction at the test scale-out is then compared with the

true runtime, calculating the relative prediction error. This random sub-sampling procedure is repeated 2000 times for every amount of training points used and the mean relative prediction error is reported. Figure 6 shows the results.

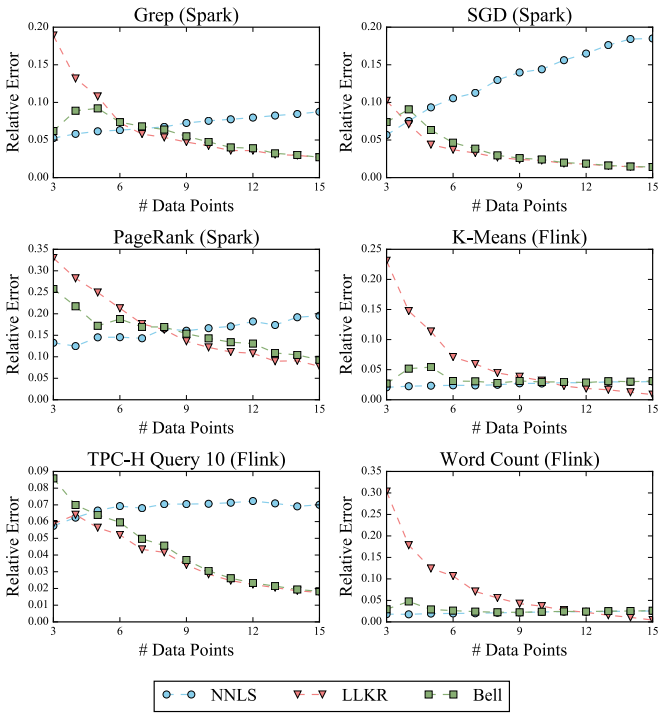

Fig. 6. Mean relative prediction error for each job as reported by repeated random sub-sampling cross-validation. NNLS and LLKR are the parametric and nonparametric models, respectively.

As expected, with increasing amounts of training data points and hence higher density of the dataset, the nonparametric method outperforms the parametric one. On the other hand, the nonparametric model is often outperformed by the parametric one for smaller datasets. Therefore, to combine the flexibility of the nonparametric model and the robustness of the parametric model it is crucial to incorporate mechanisms that switch optimally between the two models.

By using cross-validation to select between models, our approach is able to detect the better performing model with increasing amount of training points. At the same time, our approach keeps the relative error for small amounts of training data within reasonable bounds. In particular, for datasets where the parametric model already provides a good fit, Bell achieves an error that is close to the one of the parametric model.

To evaluate the overhead that Bell introduces to the submission and execution of jobs, we fitted the 75 data points of each of the six benchmark jobs 10 times. The median runtime for this ranges from 91 to 100 ms per job. Depending on the deployment of Bell and the workload repository, there is additional overhead for fetching the runtimes of previous runs, yet both overheads are relatively small compared to the seconds that it takes for a job to be scheduled and deployed as well as the minutes to hours that many jobs run.

V. Related Work

This section first summarizes related work on distributed dataflow systems and resource management for such frameworks. It then discusses work on selecting resources for data processing jobs according to runtime targets.

A. Distributed Dataflow Systems

MapReduce [1] proposed a programming and execution model for scalable and fault-tolerant data processing using parallel dataflows over interconnected commodity hardware. In MapReduce's execution model data is exchanged through a fault-tolerant distributed file system such as the Google File System [30] in-between alternating stages of Map and Reduce tasks. Systems like Dryad [12] and Nephele [31] extended this execution model by allowing arbitrary directed acyclic graphs of user-defined tasks. In Dryad and Nephele data can also be exchanged directly over network connections, without storing data on disk between subsequent tasks.

Systems like SCOPE [32], Nephele/PACTs [33], and Spark [2] provided larger sets of pre-defined operators including, for example, Joins. They also added declarative SQL-like programming languages and other features of parallel databases like automatic plan optimizations as, for example, the Stratosphere project [13] did for Nephele/PACTs.

Spark added an alternative to disk- and replication-based fault tolerance with its Resilient Distributed Datasets (RDDs) [34]. RDDs maintain enough linage information to recompute specific partitions in case of failures.

Systems like Flink [3], Spark, and Google's Dataflow [35] provide batch and stream processing in a single system. Flink also provides dedicated support for incremental processing for programs with sparse computational dependencies [36].

In summary, the available distributed dataflow frameworks provide considerably different feature sets. Therefore, different use cases match better to some systems than to others.

B. Resource Management Systems

Resource management systems for data processing frameworks such as YARN [4] and Mesos [5] allow users to run jobs of multiple data analytics frameworks on a single shared cluster. Users reserve parts of the clusters by specifying the required number and size of containers. Different distributed data processing frameworks can then run in these containers. YARN's design also moves scheduling functions towards per-job components for increased scalability.

Mesos [5] is a similar system, also enabling users to run jobs from multiple dataflow frameworks efficiently in a single shared cluster. Mesos offers a scheduling mechanism, in which the central scheduler offers individual frameworks a number of nodes, while the frameworks decide which of these offers to accept and which tasks to run on these resources. Therefore, Mesos also delegates some of the scheduling work to the frameworks. A key advantage of delegating container placement to processing systems is that the systems can optimize for goals like data locality with considerably more assumptions regarding the execution model.

C. Resource Allocation for Runtime Targets

Much of the work towards automatic resource allocation for analytics jobs has been specific to particular processing systems, while only a few works provide more general solutions.

1) Solutions For Specific Data Processing Systems: Aria [9] and Bazaar [19] both use simple MapReduce performance models to select resources for soft completion deadlines. Aria estimates the amount of execution slots necessary, Bazaar selects the number of instances and network bandwidth to assign to jobs. Bazaar uses short sample runs to profile jobs for its performance model. Aria was presented to work with both historic data and dedicated profiling, yet the authors later proposed using profiling to capture the impact of input sizes [17].

Elastisizer [18], which is part of the Starfish [37] system for automatically tuning MapReduce analytical clusters, answers cluster sizing queries of users for MapReduce jobs. Given a user's specification of the search space such as available resource types and framework configuration options as well as detailed job profiling information, Elastisizer simulates the runtime and costs using relative modeling [38].

AROMA [20] clusters previously executed MapReduce jobs based on their resource utilization. It then trains a performance model for each of these clusters that can be used to select from heterogenous resources and to configure different Hadoop MapReduce parameters. Incoming jobs are consequently profiled on subsets of the input data in a staging cluster and matched against one of the clusters to be used for provisioning and configuration.

The Jockey scheduler [11] was built for SCOPE. It automatically selects resources according to user-provided job utility functions that model deadlines and penalties. Jockey uses a simulator and detailed job statistics from previous runs to predict the runtime of a job's stages. Using precomputed simulations and online estimation of a job's progress, Jockey adapts resource allocations at runtime if a job is not performing as predicted.

Compared to these solutions, Bell does not assume a specific underlying execution model and therefore does not require detailed profiling information to be available.

2) General Solutions for Data Processing Systems: Bell is most related to systems such as Quasar [6] and Ernest [21].

Quasar models the performance implications of scale-up, scale-out, and job interference using both previously observed and dedicated sample runs. Quasar then jointly performs resource allocation and job placement. Further, Quasar monitors performance and adjusts resource allocation at runtime. In contrast, Bell leaves container placement to existing resource managers, is not based on dedicated isolated training, and does not expect systems to scale dynamically.

Ernest is a job submission tool that automatically allocates cloud resources for given runtime targets. When a job is submitted, Ernest runs the job on subsets of the inputs and different sets of resources, training a simple model of distributed computation. The combinations are selected using ideas from *optimal experiment design* [39]. In comparison, Bell is not based on isolated training runs, but aims to make the most of available workload data of recurring jobs. Moreover, while Bell's parametric regression model is based on the model proposed with Ernest, Bell automatically switches to nonparametric regression if the available data and prediction task allow this.

VI. Conclusion

This paper presented Bell, a submission tool for distributed dataflow jobs that automatically selects resources according to runtime targets. For this, Bell models the scale-out behavior of different jobs based on previous job executions. Therefore, Bell does not require isolated training runs in a staging cluster. It also does neither require instrumentation of the processing frameworks nor changes to the resource management system. Instead, since we implemented Bell as a job submission client for YARN, it can be used with existing Hadoop clusters.

Bell's technical core is its runtime prediction using parametric regression and nonparametric regression. Bell automatically chooses between these two methods depending on the available workload data for a job. As shown by our evaluation, Bell provides acceptable results for different analytics frameworks: the model selection approach we implemented overall outperforms the individual prediction models for the six data analytics jobs we implemented in Spark and Flink.

In the future, we also want Bell to decide the size of containers. Furthermore, since co-located jobs can interfere with each other in shared cluster setups, we need to incorporate job interference and container placement into predictions.

Nevertheless, Bell already helps users in making scale-out decisions for their runtime targets. Instead of effectively guessing the amount of resources, users can explicitly state their performance goals and have Bell perform the resource allocation.

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] 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.

[2] 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.

[3] 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, July 2015.

[4] 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.

[5] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. USENIX Association, March 2011.

[6] 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.

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

[8] 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.

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

[10] 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)*, ser. NSDI. USENIX Association, April 2012.

[11] 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.

[12] 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.

[13] 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, December 2014.

[14] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph Processing in a Distributed Dataflow Framework," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. USENIX Association, September 2014.

[15] I. Verbitskiy, L. Thamsen, and O. Kao, "When to Use a Distributed Dataflow Engine: Evaluating the Performance of Apache Flink," in *Proceedings of the IEEE International Conference on Cloud and Big Data Computing*, ser. CBDCom 2016. IEEE, July 2016.

[16] O.-C. Marcu, A. Costan, G. Antoniu, and M. S. Pérez, "Spark versus Flink: Understanding Performance in Big Data Analytics Frameworks," in *Proceedings of the IEEE 2016 International Conference on Cluster Computing*, ser. Cluster 2016. IEEE, September 2016.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the Outliers in Map-reduce Clusters Using Mantri," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. USENIX Association, September 2010.

[23] J. Tan, X. Pan, E. Marinelli, S. Kavulya, R. Gandhi, and P. Narasimhan, "Kahuna: Problem Diagnosis for MapReduce-based Cloud Computing Environments," in *2010 IEEE Network Operations and Management Symposium*, ser. NOMS 2010. IEEE, April 2010.

[24] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz, "Wrangler: Predictable and Faster Jobs Using Fewer Resources," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. ACM, November 2014.

[25] Z. Huang, B. Balasubramanian, M. Wang, T. Lan, M. Chiang, and D. H. K. Tsang, "RUSH: A RobUst ScHeduler to Manage Uncertain Completion-Times in Shared Clouds," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, ser. ICDCS 2016. IEEE, June 2016.

[26] W. S. Cleveland, "Robust Locally Weighted Regression and Smoothing Scatterplots," *Journal of the American Statistical Association*, vol. 74, no. 368, December 1979.

[27] Z. Ming, C. Luo, W. Gao, R. Han, Q. Yang, L. Wang, and J. Zhan, "BDGS: A Scalable Big Data Generator Suite in Big Data Benchmarking," in *Advancing Big Data Benchmarks: Proceedings of the 2013 Workshop Series on Big Data Benchmarking*, T. Rabl, N. Raghunath, M. Poess, M. Bhandarkar, H.-A. Jacobsen, and C. Baru, Eds. Springer, October 2014.

[28] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet Allocation," *The Journal of Machine Learning Research*, vol. 3, March 2003.

[29] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker Graphs: An Approach to Modeling Networks," *The Journal of Machine Learning Research*, vol. 11, March 2010.

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

[31] 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.

[32] 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, August 2008.

[33] 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.

[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.

[35] 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, August 2015.

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

[37] 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.

[38] M. P. Mesnier, M. Wachs, R. R. Sambasivan, A. X. Zheng, and G. R. Ganger, "Modeling the Relative Fitness of Storage," in *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '07. ACM, June 2007.

[39] Pukelsheim, Friedrich, *Optimal Design of Experiments*. SIAM, March 1993, vol. 50.