Dynamic Resource Allocation for Distributed Dataflows

vorgelegt von
Lauritz Thamsen, M.Sc.,
geb. in Braunschweig

von der Fakulät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr. Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Tilmann Rabl
Gutachter:	Prof. Dr. Odej Kao
Gutachter:	Prof. Dr. Andreas Polze
Gutachter:	Prof. Dr. César A. F. De Rose

Tag der wissenschaftlichen Aussprache: 4. Mai 2018

Berlin 2018

Acknowledgments

I want to take the opportunity to thank a number of people, who supported me and helped shape this thesis.

First, I want to thank my advisor Odej Kao for the valuable advice and guidance over the last years, for the opportunities to present my work at international conferences and to other research groups, and for the helpful feedback I received on drafts of this thesis. I also want to thank Cesar de Rose and Andreas Polze for agreeing to review this thesis and for the feedback I received when I was allowed to present my results prior to submitting this thesis.

I am grateful to Jossekin Beilharz, Thomas Renner, and Ilya Verbitskiy for critically reading this thesis and for helping me improve the presentation of my results with many practical suggestions and comments. Jossekin even took the time to review the preliminary version of this thesis while we were on the road for two weeks, attending conferences in Hong Kong and Taipei. Thomas and Ilya, on the other hand, were not only very supportive in the preparation of this thesis, but also closely worked with me in the last years and I am thankful for the good collaboration. I would also like to give credit to the students who worked with me over the last years, especially to Jannis Koch and also to Benjamin Rabier, Julian Böhm, and Sascha Wolke. Furthermore, I want to thank Uta Hartmann, Oliver Buck, and Jana Bechstein for proofreading this thesis and helping me improve the style.

I am thankful to my colleagues and former colleagues at TU Berlin. I especially want to thank Tobias Herb, Andreas Kunft, Alexander Alexandrov, and Asterios Katsifodimos here. They helped me get into the research field of this thesis when I first started at TU Berlin with a background in interactive programming systems and only a rudimentary understanding of scalable data analytics.

I also want to take this opportunity to thank Robert Hirschfeld and his research group, of which I was allowed to be part of during my Master's program at Hasso-Plattner-Institut and with whom I wrote all my first academic texts, arguably starting my academic endeavors about seven years ago.

Finally, I want to thank my family and friends, especially my parents and Uta, for their understanding and support.

Abstract

Distributed dataflow systems enable users to process large datasets in parallel on clusters of commodity nodes. Users temporarily reserve resources for their batch processing jobs in shared clusters through containers. A container in this context is an abstraction of a specific amount of resources, typically a number of virtual cores and an amount of memory. For their production batch jobs, users often have specific runtime targets and need to allocate containers accordingly. However, estimating the performance of distributed dataflow jobs is inherently difficult due to the many factors the performance depends on such as programs, datasets, systems, and resources. Additionally, there is significant performance variance in the execution of distributed dataflows in shared large commodity clusters. For these reasons, users often over-provision resources considerably to ensure the runtime targets of their production jobs are met. This behavior leads to unnecessary low resource utilizations and thereby generates needless costs.

This thesis presents novel methods for predicting the performance of distributed dataflow jobs and for allocating minimal sets of resources predicted to meet users' runtime targets. The core question addressed by this thesis is how minimal resources can be allocated automatically for a given runtime target and a production batch job of a distributed dataflow framework. To this end, this thesis contributes (1) two models for capturing the scale-out behavior of distributed dataflow jobs, a simple parameterized model of distributed processing and a nonparametric model able to interpolate arbitrary scale-out behavior given dense training data, and a method for automatically choosing between these two models, (2) different measures of the similarity between job executions and methods for selecting similar previous executions of a job as a basis for accurate performance prediction, and (3) a method for continuously monitoring a running job's progress towards its runtime target and dynamically adjusting resource allocations based on per-stage runtime predictions. The overall solution we present in this thesis supports multiple distributed dataflow systems through the use of black-box models and can be deployed on a per-application basis in existing cluster setups.

The methods presented in this thesis have been implemented in prototypes, experimentally evaluated on a commodity cluster using exemplary distributed dataflow jobs, and peer-reviewed for publication at renowned international conferences. For the experiments, we used jobs from the domains of search, relational processing, machine learning, and graph processing. We further used different datasets of these domains, ranging from 1 to 745.5 gigabytes, and up to 60 cluster nodes.

Zusammenfassung

Verteilte Datenflusssysteme erlauben es Nutzern, große Datenmengen parallel auf Computerclustern zu verarbeiten. Nutzer reservieren für ihre Analyseprogramme Ressourcen mittels sogenannter Container. Diese Container repräsentieren eine bestimmte Menge an Ressourcen, zum Beispiel eine bestimmte Anzahl an Prozessorkernen und eine Menge Hauptspeicher. Für produktiv eingesetzte Analyseprogramme haben Nutzer oft spezifische Laufzeitvorgaben. Es ist jedoch schwierig, das Laufzeitverhalten von verteilten Datenflussprogrammen vorher abzuschätzen, weil dieses von sehr vielen Faktoren beeinflusst wird. Einen wesentlichen Einfluss auf das Laufzeitverhalten haben neben den Programmen dabei die Datensätze, die Systeme und die Ressourcen. Zudem variiert die Ausführungsgeschwindigkeit von verteilten Datenflussprogrammen erheblich in von vielen Nutzern gemeinsam verwendeten Commodity Clustern. Daher reservieren Nutzer häufig deutlich mehr Ressourcen als erforderlich, um sicherzustellen, dass Laufzeitanforderungen eingehalten werden. Diese Vorgehensweise führt allerdings zu unnötig niedriger Ressourcenauslastung und dadurch zu unnötigen Kosten.

Diese Doktorarbeit präsentiert neue Methoden zur Vorhersage der Laufzeit von verteilten Datenflussprogrammen und zur Reservierung minimaler zur Einhaltung von Laufzeitvorgaben nötiger Ressourcen. Die Forschungsfrage dieser Doktorarbeit ist demnach, wie minimal nötige Ressourcen für gegebene Laufzeitanforderungen von produktiv eingesetzten verteilten Datenflussprogrammen automatisch ausgewählt werden können. Dazu leistet die Doktorarbeit die folgenden Beiträge. (1) Es werden zwei Modelle zur Beschreibung des Skalierungsverhaltens von verteilten Datenflussprogrammen vorgestellt sowie eine Methode, um automatisch zwischen den beiden Modellen zu wählen. (2) Es werden mehrere verschiedene Maße für die Ähnlichkeit zweier Ausführungen des gleichen Datenflussprogramms präsentiert, sowie Methoden um genau diejenigen ähnlichen vorangegangenen Ausführungen als Basis für die Laufzeitvorhersage von Programmen auszuwählen, die eine hohe Vorhersagegenauigkeit versprechen. (3) Es wird eine Methode vorgestellt, die mittels Laufzeitvorhersagen für die einzelnen Teilschritte von Datenflussprogrammen abschätzt, ob ein aktuell laufendes Programm die Laufzeitvorgabe ungefähr einhalten wird, und die Menge an reservierten Ressourcen ansonsten entsprechend dynamisch anpasst. Die Lösung, die in dieser Doktorarbeit präsentiert wird, unterstützt durch den Einsatz von Blackbox-Modellen verschiedene verteilte Datenflusssysteme und kann für einzelne Anwendungen in bestehenden Cluster-Aufbauten verwendet werden.

Die vorgestellten Methoden wurden prototypisch implementiert, experimentell mit beispielhaften Datenflussprogrammen sowie großen Datensätzen auf einem Commodity Cluster evaluiert und im Rahmen von Publikationen auf mehreren renommierten internationalen Konferenzen begutachtet. Für die Experimente wurden unter anderem Programme aus den Domämen relationale Datenverarbeitung, maschinelles Lernen, und Graphanalyse verwendet. Außerdem wurden verschiedene bis zu 745,5 Gigabyte große Datensätze und bis zu 60 Commodity Server verwendet.

Contents

1 Introduction **1**
- 1.1 Problem Definition . . . 3
- 1.2 Contributions . . . 5
- 1.3 Outline of the Thesis . . . 7

2 Background **9**
- 2.1 Distributed Data-Parallel Processing . . . 9
  - 2.1.1 Distributed Dataflows . . . 11
  - 2.1.2 Comparison to High-Performance Computing . . . 15
- 2.2 Shared Analytics Cluster Setup . . . 17
  - 2.2.1 Distributed File Systems . . . 18
  - 2.2.2 Resource Management Systems . . . 19
  - 2.2.3 Co-Located Cluster Setup . . . 21

3 Related Work **23**
- 3.1 Distributed Dataflow Systems and Related Distributed Systems . . . 23
  - 3.1.1 Distributed Dataflow Systems . . . 24
  - 3.1.2 Systems Used in Conjunction with Distributed Dataflow Systems . . . 26
  - 3.1.3 Related Parallel and Distributed Computing Systems . . . 29
- 3.2 Runtime Prediction and Resource Allocation for Runtime Targets . . . 32
  - 3.2.1 Pure Runtime and Progress Estimation . . . 33
  - 3.2.2 System-Specific Automatic Resource Allocation . . . 33
  - 3.2.3 Resource Allocation Based on Black-Box Prediction Models . . . 35
- 3.3 Adaptive Resource Management . . . 37

4 Problem and Concepts **39**
- 4.1 Problem and State of the Art . . . 39
- 4.2 Assumptions and Requirements . . . 42
  - 4.2.1 Batch Processing Jobs . . . 42
  - 4.2.2 Distributed Dataflow Systems . . . 42
  - 4.2.3 Dedicated Analytics Clusters . . . 44
  - 4.2.4 Requirements for a Practical Solution . . . 45
- 4.3 Approach and Methods . . . 45
  - 4.3.1 Solution Overview . . . 45
  - 4.3.2 Application to Iterative Jobs . . . 49

4.4 System Architecture . . . 50
4.4.1 Architecture Overview . . . 51
4.4.2 Prototype Components . . . 52
4.4.3 Integration with YARN and Spark . . . 55

5 Modeling the Scale-Out Behavior of Batch Jobs 59
5.1 Scaling out Distributed Dataflows . . . 59
5.2 Scale-Out Models for Distributed Dataflows . . . 64
5.2.1 Parametric Regression . . . 64
5.2.2 Nonparametric Regression . . . 66
5.2.3 Automatic Model Selection . . . 67
5.3 Evaluation . . . 67
5.3.1 Cluster Setup . . . 67
5.3.2 Experiments . . . 68
5.3.3 Results . . . 69

6 Estimating Job Runtimes Based on Similar Previous Executions 73
6.1 Predicting Job Performance Based on Previous Executions . . . 74
6.2 Assessing the Similarity of Job Executions . . . 76
6.2.1 Similarity Measures . . . 76
6.2.2 Similarity Quality . . . 80
6.2.3 Training Job-Specific Thresholds and Weights . . . 83
6.3 Estimating the Remaining Runtime of Recurring Iterative Jobs . . . 84
6.3.1 Estimate Inference . . . 85
6.3.2 Final Estimate . . . 86
6.3.3 Outlier Iterations . . . 86
6.4 Evaluation . . . 87
6.4.1 Cluster Setup . . . 87
6.4.2 Experiments . . . 87
6.4.3 Results . . . 89

7 Allocating Resources for Jobs With Runtime Targets 95
7.1 Stage-Wise Runtime Prediction . . . 96
7.2 Selecting Resources for Runtime Targets . . . 97
7.2.1 Resource Allocation Based on Predicted Runtimes . . . 97
7.2.2 Selecting Resources on Job Submission . . . 98
7.2.3 Adjusting Allocations at Runtime . . . 99
7.2.4 Selecting Resources for Jobs with Insufficient Training Data . . . 101
7.3 Evaluation . . . 101
7.3.1 Cluster Setup . . . 101
7.3.2 Experiments . . . 102
7.3.3 Results . . . 103

8 Conclusion 107

1 Introduction

Many organizations have to work with increasingly large datasets. The cost of storing a large volume of data has decreased considerably. Therefore, more data can be saved for later analysis. Furthermore, there is also more data being generated. A major reason for this is the digitalization with the Internet of Things as a driving force. Large numbers of sensors as increasingly deployed in manufacturing and urban infrastructures continuously record and emit data. For example, the Spanish city of Santander currently uses the continuous measurements of 15.000 sensors to monitor and analyze traffic conditions, noise, and air quality [1]. User-generated content is another reason for the increasingly large volumes of data, especially in combination with enormous user bases. Today, some Internet companies have billions of users. This many users generate immense datasets. Companies process this data for their core business and also to gain further insights. Google, for example, processes hundreds of terabytes of Web pages to improve Web search [2]. The company also uses millions of clicks as a basis for recommending articles on its news aggregation platform [3]. Furthermore, researchers analyzed Web search logs of millions of users of multiple search engines to detect previously unknown pharmaceutical side effects [4].

The increasing size of datasets and decreasing prices of commodity hardware, especially when compared to specialized parallel computers, have led to clusters of computers being used widely for working with large datasets. Companies, researchers, and the open source community developed distributed systems for storing and processing large datasets using clusters of shared-nothing commodity nodes. Distributed data-parallel processing systems support users in developing and executing distributed programs. These systems offer high-level programming abstractions that hide the complexities of parallel programming from users. Moreover, they provide efficient fault-tolerant distributed execution of programs.

A particularly popular class of systems for general-purpose distributed data-parallel processing are distributed dataflow systems like MapReduce [5], Spark [6], and Flink [7]. Users of these systems create programs from a set of operators, configure these data transformation tasks with sequential user code, and then connect the tasks to form directed job graphs. The distributed dataflow systems subsequently execute jobs in parallel and distributed across a set of connected shared-nothing commodity nodes. Tasks are executed data-parallely, so parallel task instances process partitions of the data. Further, depending on the semantics of operators, also instances of subsequent tasks in the job graphs can potentially process elements simultaneously, adding pipeline parallelism.

In comparison to the technology used for high-performance computing (HPC), distributed dataflow systems arguably make it easier to develop scalable data-parallel programs that efficiently analyze large datasets in parallel using large sets of commodity cluster nodes. This is due to the high-level programming abstractions and comprehensive distributed runtime environments of distributed dataflow systems. The programming model is restricted to a set of pre-defined operators, yet for these operators, the systems provide efficient implementations, data partitioning and parallelization, communication and synchronization, distributed task management, as well as monitoring and fault tolerance.

A prerequisite for executing a distributed dataflow program with a certain level of parallelism is using a set of compute resources providing that level of parallelism. Usually, multiple users share clusters to increase the resource utilization. In this case, resource management systems like Mesos [8] and YARN [9] manage the cluster nodes and users reserve shares of the available resources for their jobs via containers. A container is an abstraction of resources, used for resource negotiation and scheduling. It represents for example a number of virtual cores and an amount of main memory. Users typically reserve tens to hundreds of containers for their jobs, while containers of multiple jobs share the cluster and nodes, typically without resource isolation.

Users often have specific performance requirements for their production data processing jobs [10, 11]. Twitter, for instance, aims for a target latency of ten minutes for updating their search completion indices based on terabytes of log data [12]. Missing such runtime targets negatively effects the usability of services. Moreover, in case of agreed upon service level objectives (SLOs), typically expressed in the form of service level agreements (SLAs), missing runtime targets also entails financial penalties.

However, how specific levels of parallelism and resource allocations translate to job runtimes is often not straightforward for distributed dataflows. First, the speed-up of additional compute capacities is limited by ingestion rates, partitioning, and synchronization overheads. For example, when the available network links are fully saturated for most of a job's runtime, using more cores and memory will not speed up the job's execution significantly. Second, even when jobs can be scaled out to meet a particular runtime target, estimating beforehand how many containers are actually necessary to meet this goal is difficult. The scale-out behavior of distributed dataflow jobs depends on many factors related to the programs, datasets, systems, and resources used. The scale-out behavior is also often not completely straightforward. Due to data skew and the overheads of distributed communication and synchronization, it is fully possible that allocating more containers will lead to longer runtimes. Furthermore, failures and interference with concurrently running workloads can add considerable runtime variance [11, 13, 14].

Given runtime targets despite the difficulty of anticipating distributed dataflow performance, users defensively over-provision resources for their important production jobs. This behavior is problematic as it leads to poor overall cluster utilizations. In fact, a study of a production cluster at Twitter [15] shows that the aggregate CPU utilization

was consistently below 20%, despite resource reservations close to 80% of the total capacities. Memory utilization was between 40–50%, yet still differed considerably from the reserved capacities of also close to 80% of the total memory. Similarly, an analytics cluster at Google only achieved a CPU utilization of 25–35% and memory utilization of 40%, even though reservations exceeded 75% and 60% of the available capacities [13]. Resource utilizations this low suggest large potential for optimization. Moreover, improvements in this area will have huge impacts. Organizations would be able to save money on initial infrastructure investments, operational and maintenance costs, as well as costs for energy consumption.

At the same time, many production batch jobs run repeatedly. These are batch jobs that periodically update core data structures of organizations, scheduled for instance on a daily or hourly basis [10, 14, 16]. They account for up to 60% of the overall jobs running on dedicated analytics clusters and typically have defined performance requirements [11]. This presents an opportunity to collect runtime statistics for recurring jobs and model their scale-out behavior based on previous executions. Such scale-out models allow to predict the runtimes of a job for specific sets of resources. Based on these predictions, a minimal scale-out and therefore number of containers can be chosen automatically for a user's runtime target. Containers then translate to temporarily reserved resources, typically cores and memory, on specific worker nodes. Furthermore, monitoring of the actual job performance and dynamic adjustments based on runtime statistics can be used to address runtime variance.

Accurate runtime prediction and automatic resource allocation are especially important for distributed dataflow jobs. First, due to the data-parallel execution model and efficient scalable runtime environments of distributed dataflow systems, a multitude of scale-outs and thus numbers of containers can be used for jobs. Second, distributed dataflow systems make it easier to develop efficient data-parallel programs that make use of large sets of cluster resources, allowing even users without extensive knowledge of parallel programming, distributed systems, and data-parallel processing to create such programs. At the same time, even expert users do not always fully understand system and workload dynamics [17, 18]. Thus, systems arguably should alleviate users from having to allocate resources for their jobs and runtime targets themselves. Moreover, accurate runtime prediction can also be immensely useful for resource management systems. Similarly as in the area of HPC [19, 20], accurate runtime predictions can be used to go beyond merely selecting the next job from a queue of submitted jobs to planning schedules ahead based on the predicted runtimes of jobs and future availability of resources.

1.1 Problem Definition

The topic of this thesis is resource allocation for production batch jobs of distributed dataflow frameworks. The research question of this thesis is

"Given a runtime target for a production batch job of a distributed dataflow framework, how can minimally necessary sets of resources be allocated automatically?".

The problem embodied in this research question and addressed in this thesis is twofold:

Runtime Prediction The runtime of distributed dataflow jobs is difficult to predict due to the many factors that determine the performance of distributed dataflows, for which often not even full statistical information are available before execution, rendering even detailed performance models obsolete without sample runs. This first part of the problem therefore asks how runtimes of distributed dataflow systems can be predicted based on samples.

Runtime Variance There is considerable variance in the runtimes of distributed dataflow jobs in shared commodity clusters. This is due to worker failures, updated data and code, as well as varying degrees of data locality and interference with co-located workloads. Runtime variance can be addressed by provisioning for the worst case, but the goal of this thesis is to allocate minimal necessary resources. This second part of the problem therefore asks how runtime variance can be addressed when minimal sets of resources are allocated for distributed dataflow jobs based on runtime prediction models.

The idea of our solution is to let users explicitly state their runtime targets, use scale-out models that allow to predict the runtimes of distributed dataflow jobs, train these models on selected similar previous executions of recurring jobs, and then allocate minimal sets of resources predicted to meet users' runtime targets, while addressing performance variance with continuous monitoring and dynamic adjustments.

We make a number of assumptions, while addressing the problem:

Recurring Batch Jobs We assume that production batch jobs with specified runtime targets are executed repeatedly, for example on a daily or hourly basis.

Distributed Dataflow Systems We assume that batch jobs of distributed dataflow systems are scalable and consist of multiple stages that can be monitored, modeled, and executed with different sets of resources.

Shared Homogeneous Clusters We assume that production batch jobs with specified runtime targets run in shared homogeneous commodity clusters.

We specifically do not assume a particular distributed dataflow framework and no particular application domain. We also do not assume the availability of a dedicated staging cluster to profile new jobs. Furthermore, we do not expect jobs to run in isolation or otherwise fully predictable performance. We also do not assume control over job admission, the execution order of jobs, or container placement. Instead, we focus on setting the scale-out in terms of number of containers, which we refer to as resource allocation in this thesis, for single distributed dataflow jobs.

1.2 Contributions

This thesis proposes a set of solutions to the problem described above. These solutions make contributions in three areas: modeling the performance of distributed dataflows, selecting similar previous job executions as a basis for runtime prediction models, and allocating resources for distributed dataflow jobs dynamically based on predicted runtimes.

In the first area, the thesis presents a black-box approach for modeling the performance of distributed dataflow jobs. The idea is to find a function that describes the scale-out behavior of a job based on a couple of example executions. We present two different models and a method for dynamically selecting between models. First, a parameterized model that provides reasonable predictions from a few data points. Second, a model using non-parametric regression that, given dense training data, can be used to accurately interpolate arbitrary scale-out behaviors. Our method for selecting between models incorporates the available training data and the prediction task.

In the second area, the thesis presents an approach for selecting samples for accurate performance estimation using the similarity between executions of distributed dataflow jobs. Our approach is to continuously match a job to similar previous executions based on all available statistics, including statistics on the current job execution collected at runtime, allowing more accurate estimations as a job progresses. This way, a current job is matched to previous executions based on its actual runtime behavior. We first present a set of similarity measures. We then present methods for assessing the overall similarity between job executions. Specifically, we automatically train weights and thresholds on all previous executions of a job to give weight to those similarities that provide accurate estimations for a particular job. Finally, we present a method for estimating the progress of iterative distributed dataflow jobs that makes use of our similarity matching techniques.

In the third area, the thesis presents a practical application of the methods for performance modeling and for selecting sample runs as a basis for prediction models. Here, we present a method for automatically selecting minimal sets of resources predicted to meet the runtime targets of recurring distributed dataflow jobs. We use scale-out models for each of a job's stages, trained on previous executions of jobs, and then dynamically allocate resources for each job stage using the scale-out models. The overall solution we present here monitors jobs at runtime, predicts the remaining runtime at each of a job's synchronization barriers, and dynamically adjusts resource allocations to meet the job's runtime target despite variance in job performance.

We have implemented the methods presented in this thesis in a prototype of a system for dynamic resource allocation, which we call *Ellis*. Ellis makes use of a component for scale-out modeling, called *Bell*, and a component for selecting similar previous executions as training data, called *Cutty*. We integrated Ellis with YARN and Spark. Moreover, we evaluated all three prototypes experimentally with several exemplary Spark and Flink applications, large real-word as well as synthetic datasets, and up to 60 nodes of a commodity cluster.

Central parts of this thesis have been published as follows:

1. L. Thamsen, I. Verbitskiy, J. Beilharz, T. Renner, A. Polze, and O. Kao. "Ellis: Dynamically Scaling Distributed Dataflows to Meet Runtime Targets". In: *Proceedings of the 2017 IEEE International Conference on Cloud Computing Technology and Science*. CloudCom 2017. IEEE, 2017.
2. J. Koch, L. Thamsen, F. Schmidt, and O. Kao. "SMiPE: Estimating the Progress of Recurring Iterative Distributed Dataflows". In: *Proceedings of the 18th International Conference on Parallel and Distributed Computing, Applications and Technologies*. PDCAT 2017. IEEE, 2017.
3. L. Thamsen, I. Verbitskiy, F. Schmidt, T. Renner, and O. Kao. "Selecting Resources for Distributed Dataflow Systems According to Runtime Targets". In: *Proceedings of the IEEE International Performance Computing and Communications Conference*. IPCCC 2016. IEEE, 2016.

Additionally, the following publications are related to this thesis:

1. L. Thamsen, I. Verbitskiy, B. Rabier, and O. Kao. "Learning Efficient Co-locations for Scheduling Distributed Dataflows in Shared Clusters". In: *Services Transactions on Big Data* 5.1., 2018.
2. T. Renner, L. Thamsen, and O. Kao. "Adaptive Resource Management for Distributed Data Analytics Based on Container-level Cluster Monitoring". In: *Proceedings of the 6th International Conference on Data Science, Technology and Applications*. DATA 2017. SCITEPRESS, 2017.
3. L. Thamsen, B. Rabier, F. Schmidt, T. Renner, and O. Kao. "Scheduling Recurring Distributed Dataflow Jobs Based on Resource Utilization and Interference". In: *Proceedings of the 6th 2017 IEEE International Congress on Big Data*. BigData Congress 2017. IEEE, 2017.
4. T. Renner, L. Thamsen, and O. Kao. "CoLoc: Distributed Data and Container Colocation for Data-Intensive Applications". In: *Proceedings of the 2016 IEEE International Conference on Big Data*. IEEE BigData 2016. IEEE, 2016.
5. L. Thamsen, T. Renner, M. Byfeld, M. Paeschke, D. Schröder, and F. Böhm. "Visually Programming Dataflows for Distributed Data Analytics". In: *Proceedings of the 2016 IEEE International Conference on Big Data*. BigData 2016. IEEE, 2016.
6. 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*. CBDCom 2016. IEEE, 2016.
7. L. Thamsen, T. Renner, and O. Kao. "Continuously Improving the Resource Utilization of Iterative Parallel Dataflows". In: *Proceedings of the IEEE International Conference on Distributed Computing Systems Workshops*. ICDCSW 2016. IEEE, 2016.
8. T. Herb, L. Thamsen, T. Renner, and O. Kao. "Aura: A Flexible Dataflow Engine for Scalable Data Processing". In: *Andreas Knüpfer, Tobias Hilbrich, Christoph Nietham-*

mer, José Gracia, Wolfgang E. Nagel, Michael M. Resch (eds.), Tools for High Performance Computing 2015. Springer, 2016.

9. T. Renner, L. Thamsen, and O. Kao. "Network-Aware Resource Management for Scalable Data Analytics Frameworks". In: *Proceedings of the 2015 IEEE International Conference on BigData*. BigData 2015. IEEE, 2015.
10. A. Alexandrov, A. Kunft, A. Katsifodimos, F. Schüler, L. Thamsen, O. Kao, T. Herb, and V. Markl. "Implicit Parallelism Through Deep Language Embedding". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. ACM, 2015.

1.3 Outline of the Thesis

The remainder of this thesis is structured as follows.

Chapter 2 presents the necessary background on distributed data processing and analytics clusters. First, we present distributed data-parallel processing systems with a particular focus on distributed dataflow systems and differences to the area of HPC. Second, we describe the typical setup of shared analytics clusters. We explain the main concepts behind distributed file systems and resource management systems. We also describe how these systems are used in conjunction with distributed dataflow systems.

Chapter 3 presents related work. First, we present systems for distributed data-parallel processing, focussing on distributed dataflow systems, but also discussing the applicability of the methods presented in this thesis to other parallel computing systems. Second, we present work on predicting the runtimes of distributed dataflow jobs and on allocating resources based on predicted runtimes. Third, we present systems that monitor the performance of distributed processing systems and use collected runtime statistics to adaptively improve resource usage.

Chapter 4 describes the problem in detail and gives an overview of our solution. First, we explain how the difficulty of estimating the performance of distributed dataflows leads to low cluster utilizations and unnecessary costs, arguing that systems should select resources automatically for users' runtime targets. We also identify critical limitations with current solutions to the problem. Second, we discuss our central assumptions, from which we derive the requirements for our solution. Third, we present our overall approach and the central methods to solve the presented problem. Finally, we present the system architecture and the implementation of our prototype components.

Chapter 5 presents methods for modeling the performance of distributed dataflow jobs. First, we describe what happens when distributed dataflow jobs are scaled out to more compute resources and which factors impact whether scaling out a particular job actually decreases its runtime. Second, we present two different models that, given sufficient training samples, can be used to capture the scale-out behavior of distributed dataflow jobs. Third, we present a method for automatically choosing between the mod-

els, given both training data and a prediction task. Finally, we present an evaluation of the presented methods using six distributed dataflow jobs and a cluster of 60 nodes.

Chapter 6 presents methods for selecting similar previous executions of distributed dataflow jobs as a basis for performance estimation. First, we describe the general approach of using similar previous executions for estimating the performance of jobs. Second, we present multiple measures that can be used to compare job executions such as the input data size and stage runtimes. We also propose methods for assessing the overall similarity of executions and the usefulness of similarity measures for performance estimation. Furthermore, we explain how we train thresholds and weights for each job automatically based on all of its previous executions. Third, we show a method for estimating the remaining runtimes of running iterative distributed dataflow jobs using the previously presented methods. Finally, we present an evaluation of the presented methods using three iterative distributed dataflow programs, nine different datasets, and a cluster of 40 nodes.

Chapter 7 presents methods for dynamic resource allocation for distributed dataflow jobs with specific runtime targets. First, we show how we use scale-out models for individual job stages to predict the runtime of distinct parts of a distributed dataflow job. Second, we explain how sets of resources can be allocated based on predicted stage runtimes, both initially and after assessing a job's progress at the synchronization barriers in-between job stages. Finally, we present an evaluation of Ellis, the prototype in which we implemented these methods for dynamic resource allocation, using four distributed dataflow jobs and a cluster of 60 nodes.

Chapter 8 concludes this thesis by summarizing our results and identifying directions for future work.

2 Background

Contents

2.1 Distributed Data-Parallel Processing 9
2.1.1 Distributed Dataflows 11
2.1.2 Comparison to High-Performance Computing 15
2.2 Shared Analytics Cluster Setup 17
2.2.1 Distributed File Systems 18
2.2.2 Resource Management Systems 19
2.2.3 Co-Located Cluster Setup 21

This chapter presents the background of this thesis. We first describe distributed data-parallel processing, focusing on distributed dataflow systems and highlighting differences between these systems and the area of high-performance computing. Subsequently, we present the typical setup of dedicated shared cluster infrastructures for data-parallel processing and the main ideas behind distributed file systems and resource management systems.

2.1 Distributed Data-Parallel Processing

Parallel computing requires parallel hardware, parallel problems, and parallel programs. Parallel hardware can be a single parallel computer, for example one equipped with multiple cores or processors, but also a large cluster of connected multi-core machines. Due to the increasing size of datasets and the low prices for commodity hardware compared to high-capacity parallel hardware, clusters of locally connected shared-nothing commodity nodes are often used for storing and processing large datasets. These computing environments require distributed systems for processing large datasets in parallel.

Computing problems can be solved in parallel when multiple input elements can be processed independently or multiple different programming steps can be executed simultaneously. Processing multiple input elements in parallel following the same program is called *data parallelism*. In contrast, simultaneously executing multiple different programs that work together to solve a problem is called *task parallelism*. When large datasets have to be processed, problems are often data-parallel. In this case, the input data can be split up among parallel workers, typically running on multiple connected nodes, to be processed in parallel. The parallel workers need to be synchronized and

exchange data when multiple elements with specific characteristics are to be combined, otherwise they can process partitions of the entire data independently. Furthermore, even combining groups of elements can happen in parallel, as long as there are multiple groups that should be combined. This way, using distributed data-parallel processing systems to run data-parallel programs on a set of parallel computers such as a commodity cluster can significantly speed up computations. The two categories *data-intensive* and *compute-intensive* are also used to distinguish problems [21]. Data-intensive applications are I/O-bound, predominantly reading and transferring large amounts of data, while compute-intensive applications are CPU-bound, primarily devoting time to computation.

A popular class of systems for general-purpose distributed data-parallel processing are distributed dataflow systems [22]. Prominent examples of this class of systems are MapReduce [5], SCOPE [23], Spark [6], and Flink [7]. In distributed dataflow systems, data is processed as it flows through a graph of data-parallel operators. Data-parallel instances of the operators run on homogeneous shared-nothing commodity cluster nodes. The data usually gets re-partitioned and distributed among these data-parallel task instances multiple times during the execution of a distributed dataflow job. Distributed dataflow systems combine functional programming elements and techniques known from parallel databases. Users create programs by supplying sequential code to the second-order function Map and Reduce. Such user-code operators and database operators like Joins are then connected to form entire dataflow job graphs. Data partitioning and data-parallel operator implementations allow for effective parallel execution. Furthermore, the systems include fault-tolerant and efficient distributed runtime environments to run jobs on commodity clusters. Other techniques known from parallel databases that have been applied with distributed dataflow systems include declarative programming abstractions such as SQL-like query languages and automatic query optimization. The main conceptional difference that remains to actual parallel databases are that distributed dataflow systems only process data and do not persistently manage data. Instead of storing and indexing the datasets, distributed dataflow systems typically ingest input datasets from distributed file systems, usually adhoc and without access to comprehensive data statistics. Distributed dataflow systems have been developed for data-intensive applications, yet workloads can also be compute-intensive [24].

Besides distributed dataflow systems, there are numerous other kinds of distributed data-parallel processing systems. Many of these are domain-specific solutions such as systems specifically for distributed graph processing [25, 26] or distributed machine learning [27, 28]. However, this thesis focuses on runtime prediction, effective training data selection, and dynamic resource allocation for distributed dataflow systems and therefore we discuss these systems in detail in the following section. Afterwards, we discuss major differences to the area of HPC in regards to suitable problems, used hardware, and applied programming abstractions.

2.1.1 Distributed Dataflows

Distributed dataflows are graphs of connected data-parallel operators, which execute user-defined functions on a set of shared-nothing commodity cluster nodes. Distributed dataflow systems offer high-level programming abstractions and include efficient fault-tolerant distributed runtime environments for developing and executing data-parallel processing jobs. The systems allow users to create scalable data-parallel programs from sequential building blocks. Users select and connect operators such as Map, Reduce, and Join into dataflow job graphs. Map and Reduce are second-order functions that execute user-defined functions (UDFs). There are also variants of these two operators, for example pre-defined aggregations for computing sums. Join and Cross are examples for operators that combine two dataflows into one. Figure 2.1 shows an example of a dataflow graph. Two different input datasets are read-in and pre-processed by two different operators, before the data is joined and aggregated and stored again.

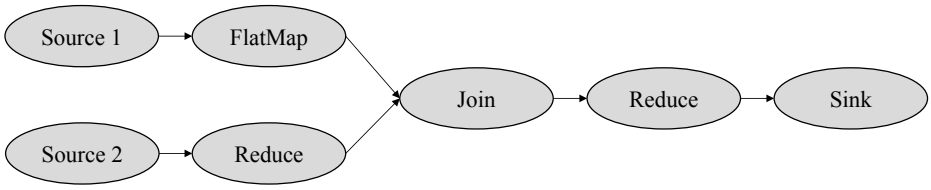


Figure 2.1: An exemplary dataflow graph.

The distributed dataflow systems take care of task parallelization, distributed execution, and fault tolerance. For each configured operator, the systems create a number of data-parallel task instances. Each of these instances is supplied with and processes a partition of the data. The number of data-parallel task instances is called degree of parallelism (DOP). Figure 2.2 shows the previous example dataflow job graph parallelized to a DOP of two.

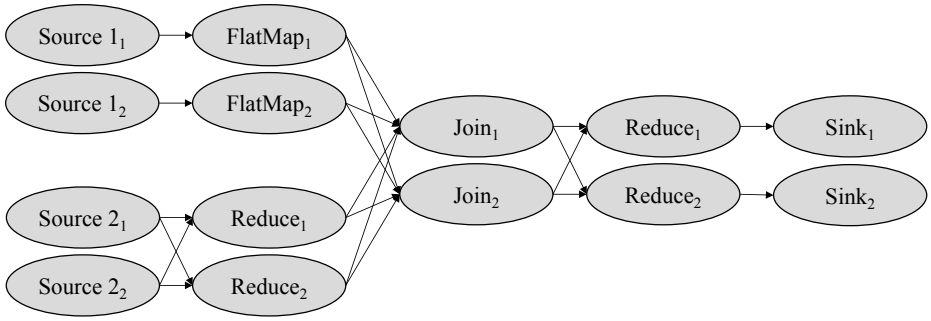


Figure 2.2: A parallelized version of the exemplary dataflow graph.

Task instances of Map operators take a single element as input, transform it, and produce a single output element for each input element. Since the Map instances process each input element independently, the input data can be assigned arbitrarily to the data-

parallel task instances of a specific Map task. Besides data parallelism, the behavior of Map operators allows for pipeline parallelism: Multiple subsequent Map tasks can run concurrently and in parallel, once preceding Map task instances produced input for subsequent ones. Operators like Reduce and Join, in comparison to Map, merge multiple elements with the same key. Task instances of these operators require all elements with the same key to be available before they can start outputting results. Therefore, all previous tasks have to be finished, effectively synchronizing the parallel dataflows and thereby adhering to the bulk synchronous parallel (BSP) model [29]. That is, pipelines of operators can be executed simultaneously up to a pipeline-breaking operator like a Reduce or a Join. The pipeline-breaking operators separate distributed dataflow jobs into multiple subsequent *stages*. Usually a job consists of multiple such stages, while only one stage is executed at a time. The other stages of the job are either inactive or not even scheduled and deployed to particular resources yet. Since all instances of the predecessor task or tasks have to be finished before a pipeline-breaking operator can be executed, the slowest instance of the predecessor task determines the overall runtime of a stage. Task instances that are considerably slower than the other task instances of a particular operator are called *stragglers*.

Even if operators like Join and Reduce do not allow for continued pipeline parallelism, these operators are still executed data-parallely: Multiple data-parallel task instances join or reduce groups of elements with the same key values in parallel. For this, as explained, these operators need to receive all elements with the same key values. This is achieved by partitioning the data using a partitioning function, which assigns elements with the same key to the same data-parallel task instance. In general, partitions for data-parallel processing are either created by reading in multiple splits of the input data in parallel from a distributed file system or received from predecessor tasks in the dataflow graph. Predecessor tasks either transmit existing partitions, transferring elements to one instance of the subsequent task, or re-partition the data, transferring elements to multiple instances of the subsequent task following a particular partitioning function. The latter case, in which typically all instances of a currently running task transmit elements to all instances of a subsequent task in order to re-partition the data, is called *shuffling*.

Many algorithms are iterative in nature. Examples include machine learning algorithms such as K-Means or Stochastic Gradient Descent (SGD) and graph processing algorithms like PageRank or Connected Components. In iterative algorithms, the same steps are executed repeatedly. This is also true for iterative distributed dataflow jobs. These distributed dataflow jobs repeat the same part of the job, usually multiple but at least one stage, until a given number of iterations has been performed or until a termination criteria has been met. Figure 2.3 shows an exemplary parallelized dataflow, in which two stages are repeatedly executed.

Many iterative algorithms need to examine less and less data in subsequent iterations. This behavior is called *convergence* and the elements that have to be processed in the next iteration are often called *active records*. For example, when computing the connected components of a graph using label propagation, only those vertices that have been as-

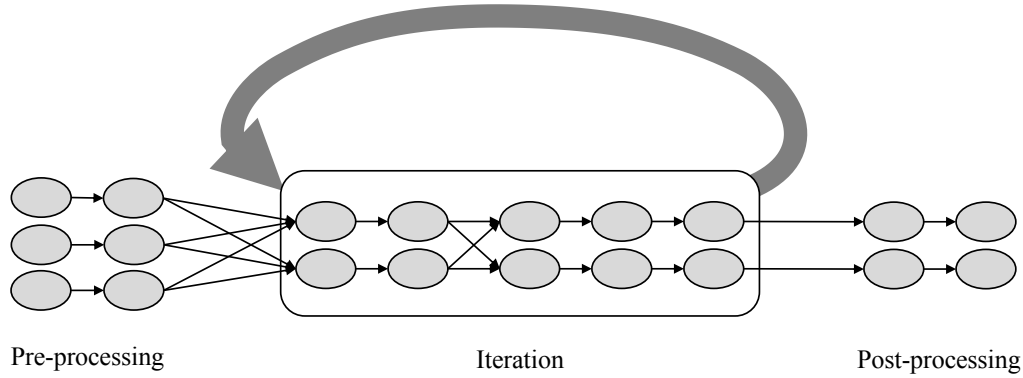


Figure 2.3: An iterative parallel dataflow graph.

signed to a different component in the last iteration have to be considered in the next iteration [30]. When this property of algorithms is used to actually process less data with subsequent iterations, this is called *incremental* or *delta* processing as opposed to *bulk* processing. Distributed dataflow systems synchronize the parallel dataflows in-between iterations, adhering to the BSP model. That is, all parallel task instances of the last distributed dataflow stage of the last iteration stop before the task instances of the first stage of the next iteration start.

Distributed dataflow jobs are executed by a number of workers. These workers are processes running on connected machines, typically shared-nothing commodity cluster nodes yet also potentially virtual machines. Distributed dataflow systems typically assume homogeneous capacities and also usually do not incorporate details on networks beyond assuming connections between all workers. In particular, distributed dataflow systems usually assume access to the same processing capacities in terms of cores and memory on each worker. That is, each worker provides the same number of so called *execution slots*, each with access to the same amount of memory and usually one processing core. The systems then assign either single task instances or pipelines of subsequent task instances to execution slots. Some systems schedule and deploy each stage separately. Other systems schedule and deploy entire dataflow jobs at once, leaving it to the operating system which thread has work and is scheduled and which thread is waiting for input.

Figure 2.4 shows a simple exemplary distributed dataflow job that is scheduled and deployed to a node. In this example, a full pipeline of task instances of two subsequent stages is scheduled and deployed to a single execution slot of a worker node. Regardless of how pipelines and stages are rolled out, intermediate results in-between subsequent distributed dataflow stages are typically kept in-memory as far as possible, while information of the current partitioning is also usually maintained across stages and iterations.

Typically commodity hardware and standard software components are used for clusters that run distributed data analytics. Given the scales at which data is processed in par-

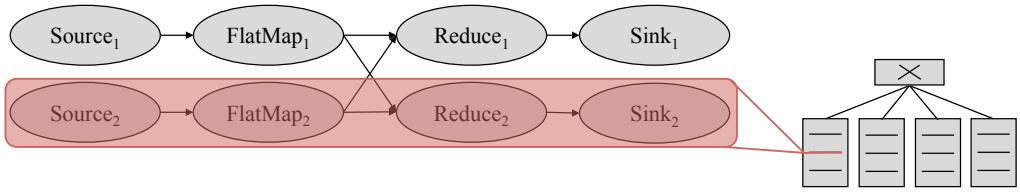


Figure 2.4: A scheduled and deployed distributed dataflow graph.

allel on connected worker nodes, failures are to be expected. Consequently, distributed dataflow systems usually implement mechanisms to achieve fault tolerance, so that not entire jobs have to be re-started in case of worker failures. One approach to fault tolerance is to snapshot intermediate data to durable storage such as to a distributed file system that stores each chunk of a file redundantly on multiple nodes. Another approach is based on tracking which operations were performed and which parts of the entire data was processed for each partition. This linage information then allows to re-compute particular lost partitions, which is considerable faster than re-running entire jobs when individual workers fail.

Besides failures, which make it necessary to read in snapshots or re-compute partitions of the data, there are multiple other reasons for particular task instances requiring more execution time than others task instances, making the slower task instances stragglers. For instance, unnoticed hardware problems might degrade the performance of a particular node and decrease the throughput of particular task instances. Furthermore, if partitioning schemes and the distribution of key values in a dataset result in more work being assigned to specific task instances, these task instances will also take longer than task instances with less work assigned. Another reason for the straggling performance of particular data-parallel task instances can be a low degree of *data locality*.

Data locality is the idea of ingesting input data locally. The degree of data locality then indicates how much of the data can be read-in locally. Data locality often has an impact of the performance of jobs, since the input data that is not locally available has to be transmitted and received over the network first. Receiving data over the network that is read-in on remote nodes usually takes longer than reading data from a local disk. Therefore, data-parallel task instances at the beginning of a distributed dataflow job that have been assigned parts of the inputs that are not locally available usually require more time to ingest their partitions, compared to task instances with local data access.

Given that distributed dataflow systems implement different mechanisms in regard to, for example, scheduling or fault tolerance, particular systems have their own strengths and use cases. On the level of developing distributed dataflow programs, the systems further offer different programming abstractions and sets of operators. Some provide, for example, only the second-order functions Map and Reduce, while others provide more operators. Some distributed dataflow systems offer declarative programming abstractions such as SQL-like query languages and domain-specific abstractions on top of the general-purpose operator-level abstractions. Furthermore, on the level of distributed

dataflow execution, there exist different physical implementations of logical operators and also different mechanisms for partitioning data among data-parallel instances. Moreover, as indicated before, there are different approaches to scheduling and deploying jobs that consist of multiple stages as well as for managing and exchanging intermediate results in-between stages. How intermediate results are handled and exchanged in-between stages is also usually connected to fault tolerance. Intermediate results can, for example, be exchanged through a distributed file system for fault tolerance. Other systems use linage instead of snapshotting to optimize the performance of the failure-free case. Aside from these matters, there is often existing code tied to a specific system. This code can be legacy code of an organization, yet also available algorithm implementations in the form of libraries. Moreover, another practical matter that leads to choosing one distributed dataflow system over the other are available connectors, data formats, and other integration code allowing to use a particular processing system with related distributed systems such as distributed file systems, databases, and messaging systems. That is, there is a class of similar distributed systems built around data-parallel operators executed on connected shared-nothing worker machines and programming abstractions that include second-order functions, yet there are numerous good reasons for users to choose a particular distributed dataflow system over another one for a specific data analytics task.

2.1.2 Comparison to High-Performance Computing

In the area of HPC, there is a long history of using high-performance computers and clusters to solve often highly compute-intensive problems with low-latency requirements [31]. Programmers use abstractions like Message Passing Interface (MPI) [32], Parallel Virtual Machine (PVM) [33], and OpenMP [34] to develop custom analysis and simulation programs that optimally harness the capabilities of the available parallel computing architecture. These hardware architectures can differ significantly, yet have in common that they provide high-performance capacities in terms of overall processing units and low-latency networks. These hardware architectures and the programming technology used for HPC allow for a wide range of problems to be solved efficiently, including compute-intensive problems that exhibit a significant level of coupling between tasks.

Compared to the methods and workloads in the area of HPC, distributed data-parallel processing with distributed dataflow systems differs in three aspects:

- Problems solved with distributed dataflow systems typically exhibit less coupling and more data parallelism than typical HPC workloads.
- The computing infrastructures used for distributed dataflow jobs are usually comprised of large numbers of commodity nodes and commodity network technology, while high-performance parallel computers and interconnects are used for HPC.

- The programming abstractions of distributed dataflow systems are more high-level and declarative as well as adhering to more restricted programming models, compared to programming technology used for HPC.

Workloads in the area of HPC are often more coupled than the workloads naturally expressed as distributed dataflows and more often compute-intensive [35–37]. Thus, HPC workloads are intensive less due to a large amount of data that can be processed independently, yet more due to a large amount of dependent computations that need to be performed. Examples for more tightly coupled HPC workloads include many simulations as, for example, from the domain of computational fluid dynamics, in which simulated particles have an effect on other particles, requiring the exchange of many messages between parallel workers and fine-grained updates to their state [38]. In contrast, distributed dataflow systems focus more on scalable data-parallel processing and, thus, managing large amounts of independent data-parallel tasks. The strategy of distributed dataflow systems, which is efficiently processing large inputs by partitioning and re-partitioning data among a set of workers that otherwise work independently, is not directly applicable for many HPC workloads. Moreover, the main performance objective for HPC workloads is typically low latency, while for distributed data-parallel processing jobs the main objective usually is high throughput.

Hardware used in the area of HPC is usually high-performance hardware, equipped with considerable computing capabilities in terms of cores, main memory, and network bandwidth [35, 36]. These massively-parallel computers feature a substantial number of processors, each with access to a large amount of shared memory, often with some areas of the overall main memory being local to a processor while other areas are less so, so that access to memory is not uniform. The processors are also typically connected by high-speed interconnects. Usually, these infrastructures are hosted and operated in dedicated computing centers, often running programs of a certain scientific domain such as climate simulations. Typically, users with extensive knowledge about the particular hardware architectures and the parallel programs that run on these environments operate HPC infrastructures and are also involved in allocating resources for jobs, since achieving optimal performance requires configuration and fine-tuning of many parameters. In contrast, distributed dataflow systems explicitly target commodity nodes and networks [2, 36]. That is, distributed dataflows are executed on large numbers of homogeneous shared-nothing commodity nodes connected by commodity network technology like Ethernet. Compared to the massively-parallel computers used for HPC, these commodity infrastructures are typically less reliable. Furthermore, users usually only decide the scale-out and number of containers to be used for their distributed dataflow jobs, while the systems take care of generating parallel executions plans and executing these plans on commodity nodes. Aside of this, there is some configuration of the distributed dataflow system in regards to resource usage, which is also not trivial, yet is often done for an entire cluster, not on a per-job basis[1]. These differences make HPC in-

[1] Flink for example allows to configure how much of the allocated memory should be used for network buffers, which is a setting for an entire Flink cluster and changes require re-

frastructures the better fit for workloads that are more tightly coupled and have higher low-latency requirements.

With HPC users typically use lower-level programming abstractions and languages compared to the high-level languages and programming abstractions used with distributed dataflow systems [21]. The programming models used for HPC typically offer message passing and global communication primitives [37, 39]. Users explicitly express parallelism as well as communication and synchronization of parallel and distributed processes. Distributed dataflow systems provide much more rigid programming models, more framework in terms of available operators that only need to be connected to form dataflow job graphs, and also much more comprehensive distributed runtime environments that for example also implement mechanisms for fault tolerance. In comparison, it is usually possible to achieve higher performance with the programming models and languages used for HPC, while the space of problems that can be expressed with HPC programming technology is also larger than what naturally fits distributed dataflows. However, this typically involves much more programming effort and in turn also possibilities to make mistakes.

In comparison, distributed dataflow systems have been explicitly designed for clusters of shared-nothing commodity hardware and data-intensive workloads. In this context, distributed dataflow systems are easier to use than many other forms of parallel and distributed programming due to their high-level programming models and comprehensive runtime environments. Users are only required to write sequential code for their distributed data-parallel processing jobs. This high level of abstraction and ease of use comes at the cost of performance, when compared to programs that have been hand-coded and carefully tuned for a specific computing environment. At the same time, there have been extensive efforts to automatically optimize the execution of distributed dataflow jobs on the level of data-parallel query plans in regard to operator selection, operator order, and optimal partitioning [10, 40–44]. Furthermore, not all classes of problems naturally fit the programming models and abstractions offered by distributed dataflow systems. Still, distributed dataflow systems are used increasingly for data-parallel processing of large datasets in many domains, including for example relational processing [23, 40, 45–48], graph processing [49], and machine learning [50–52]. This popularity of distributed dataflow systems is arguably due to the fact that they allow even users without extensive knowledge of parallel programming, distributed systems, and data-parallel processing to develop and run massively parallel distributed data processing programs.

2.2 Shared Analytics Cluster Setup

This section presents the two types of systems usually used in conjunction with distributed data-parallel processing systems, distributed file systems and resource management systems, as well as how these systems run co-located on the same cluster nodes.

starting Flink. `https://ci.apache.org/projects/flink/flink-docs-release-1.3/setup/config.html#background`, accessed 2018-02-24.

2.2.1 Distributed File Systems

Distributed file systems split large files into smaller blocks. These blocks are then distributed among and stored on multiple nodes. That is, files are stored on multiple cluster nodes and each node stores blocks of multiple files. This way, all disks in the cluster are used for storing large files. Moreover, if a single node stops working unexpectedly, each block is still available on other nodes. Thus, this design provides fault tolerance through redundancy. Furthermore, storing each file redundantly across multiple nodes provides parallel access and thereby scalability.

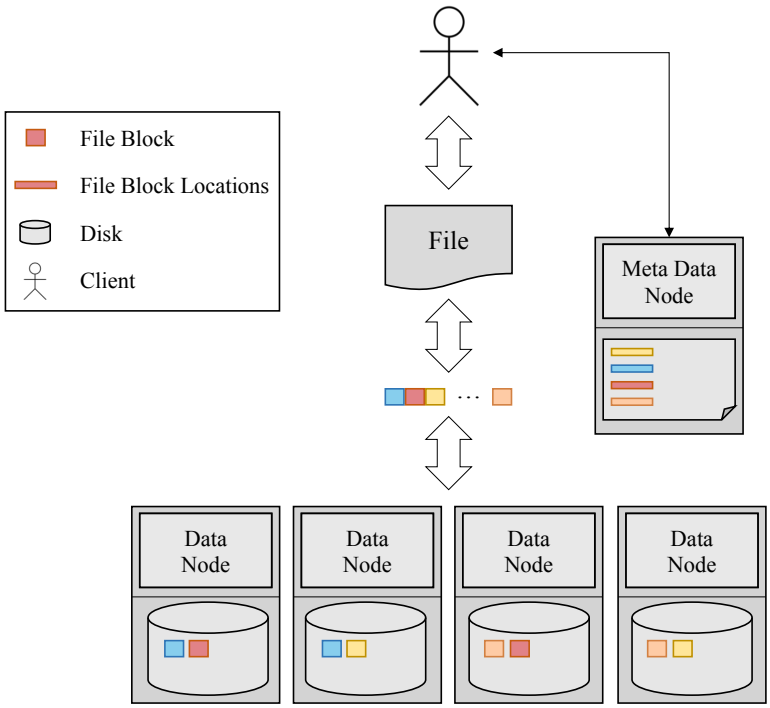


Figure 2.5: Chunks of a large file are stored in a distributed file system with a central metadata server.

Figure 2.5 exemplifies the main idea of distributed file systems. Four *Data Nodes* store blocks of a large file, while a *Meta Data Node* tracks the necessary location information to access the entire file. For this, the Meta Data Node keeps track of the specific locations of each individual block and its duplicates.

When users want to access a file stored in a distributed file system, they must obtain all individual data blocks of the file. For this, they first request block locations from the Meta Data Node and then request each individual block from the Data Nodes specified in the Meta Data Node's response. When users want to write files to a distributed file system on the other hand, they first request the locations for storing data from the Meta Data Node, before storing the splits of a large file directly on the Data Nodes. Large

files are often written and read not by centralized clients, but by distributed data-parallel systems. These systems read and write files on multiple Data Nodes in parallel.

When a particular block should be read on a particular Data Node, the block must be transferred over the network when it is not available on that Data Node. However, to obtain optimal performance data access should be coordinated in a way such that the data is read locally as much as possible, moving computations to the data for a high degree of data locality, if necessary. When a data-parallel processing system writes output to the distributed file system, it writes blocks simultaneously on a number of Data Nodes. Optimizing for both local data access and fault tolerance, a typical strategy for duplicating new blocks is storing one copy locally, another copy on a node of the same rack, and a third copy on a node of different rack.

2.2.2 Resource Management Systems

Many organizations that regularly have to process large datasets operate dedicated clusters of machines for their data processing jobs. These dedicated cluster infrastructures are typically shared by many jobs, expressed in one or multiple different distributed data-parallel processing systems. For sharing the cluster resources among jobs and processing systems, resource management systems are used. Instead of permanently running a single data processing system on the cluster, which may or may not support running multiple jobs in parallel, a resource management system runs permanently on the cluster nodes while individual processing systems run in temporarily reserved containers. That is, without resource management, a single processing system manages the whole cluster. This, however, forces users to use a specific distributed data processing system. Resource managers, on the other hand, let users choose processing systems with each reservation, as long as the particular system is integrated with the resource manager. A reservation can be valid for the runtime of a single job or for multiple related jobs, for instance when interactively analyzing the same dataset with multiple jobs.

Containers in this context are not necessarily Linux containers[2], even though some resource management systems do support the usage of Linux containers. Instead, in this context, containers are foremost logical leases of resources, allocated on a node. These containers have a specific size, representing for instance a number of cores and an amount of main memory. Users determine both how many of these containers should be reserved for a job and the size of the containers. Depending on the size of containers and node capabilities multiple containers run on a single node. Containers that are co-located on the same node share the resources of the node. Even though some resource management systems provide support for resource isolation, containers are frequently used without resource isolation. Specifically, when no means of resource isolation are installed, containers that have been allocated on the same node share the cores, network links, and disks, while main memory is usually statically allocated to particular containers. Sharing resources without resource isolation is often beneficial for the overall cluster throughput

[2] `https://linuxcontainers.org/`, accessed 2018-02-16.

as the resource demands of long-running analytics jobs often fluctuate significantly over the runtime of a job [24, 53]. Thus, when multiple jobs share resources freely, they can benefit from statistical multiplexing [13, 54]. For instance, when a Job A temporarily has a high demand for I/O and is running co-located to a Job B that already finished ingesting its input data and now is mostly CPU-intensive for a time, Job A will benefit from access to more I/O resources than would be its fair share while Job B will be able to take advantage of access to more cores than actually reserved, when no resource isolation prevents this. At the same time, two jobs that both predominantly require disk access will interfere significantly with each other, especially with access to hard disks which provide the highest read speeds for sequential access and often less throughput in case of concurrent reads from multiple threads. How much jobs benefit from shared resources or otherwise exhibit interference depends on which jobs share resources and also the resources used. Sharing and interference can have a significant impact on job runtimes and therefore lead to runtime variance.

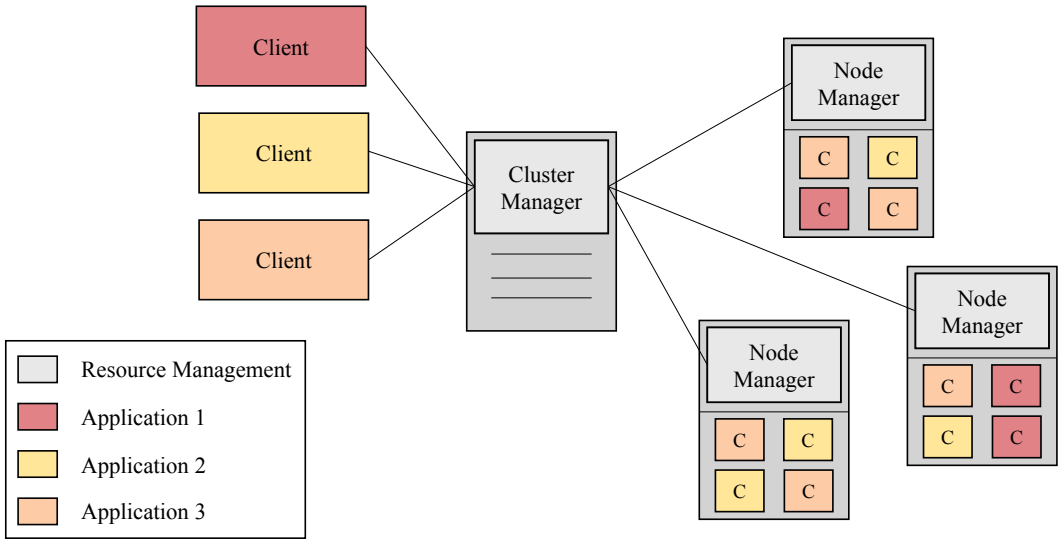


Figure 2.6: Cluster resource management with containers.

Figure 2.6 shows the general idea behind resource managers. The resource management system's master process, called *Cluster Manager*, runs on one node, while the other nodes host applications in containers. On these worker nodes runs a process called *Node Manager*. Node Managers are responsible for monitoring resource availability, for reporting failures, and for starting and stopping containers on the nodes they run on. The Cluster Manager is responsible for scheduling containers onto workers as well as for managing and monitoring the lifecycle of entire distributed applications. For this, the Cluster Manager needs a global view of the available resources and running applications. It, therefore, communicates with the Node Managers processes. In Figure 2.6, three clients each have a job running on the shared cluster and each job uses a number of containers.

Jobs are usually submitted via a specified submission protocol and go through an admission control phase. In this phase security and administrative checks are performed. If these checks are successful and jobs accepted, the Cluster Manager allocates resources to the jobs. Usually, the job first receives a single container. This first container then runs the job's *master* process, which negotiates the allocation of more resources with the Cluster Manager. The application's master then starts the distributed processing system. Specifically, it starts and monitors the system's worker processes in all additionally allocated containers. In this procedure, there are two levels of scheduling. First, the Resource Manager schedules containers onto its worker nodes. Second, the distributed processing system schedules its worker processes onto the set of allocated containers. This approach is advantageous, since the distributed processing system can make more assumptions about the execution model and also the submitted jobs, so it is usually in a better position to optimize for aspects like data locality. Beyond this centralized design for resource management systems there are also decentralized architectures, in which multiple Resource Manager processes are used for scalability and fault tolerance.

2.2.3 Co-Located Cluster Setup

Dedicated analytics clusters typically run multiple distributed systems for storing and processing large datasets in conjunction. The general software stack for this setup is depicted in Figure 2.7. A distributed file system is used for storing large datasets using the disks of all cluster nodes. By using a resource management system on top, cluster resources can be allocated to and thus shared among multiple data processing systems and jobs. That is, on top of the resource management system run applications, each potentially using a different distributed processing system. The distributed processing systems and jobs are executed in temporarily reserved containers. For this, the systems are started within these containers for the runtime of a job or a longer session.

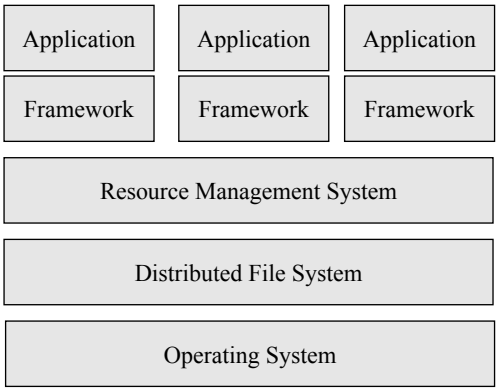


Figure 2.7: The typical software stack used with dedicated analytics clusters.

Typically, the worker processes of both the distributed file system and the resource management system run on the same nodes. This way, the jobs and processing systems

running on the cluster resources can have local access to files stored in the distributed file system. This is important as reading input data from local disks and also writing results locally takes less time than remote access, for which data must be first send over the network. Figure 2.8 depicts the execution of jobs in temporarily reserved containers with local access to files stored in a distributed file system that runs co-located to the resource management systems providing the containers. Four cluster nodes run three applications and also host parts of distributed files. The applications have local access to the parts of files that are stored on the nodes they have resource reservations on.

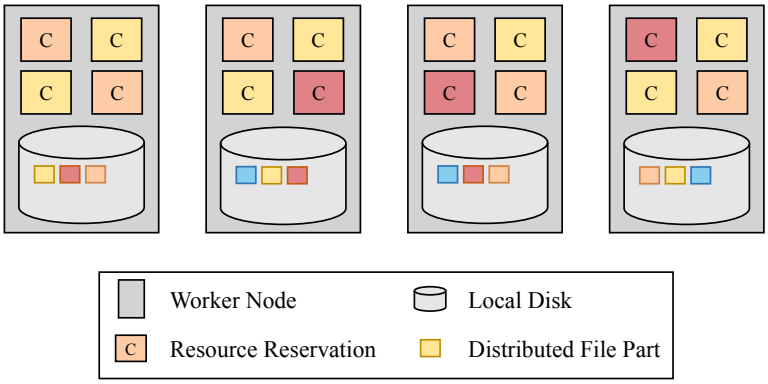


Figure 2.8: Using a resource management system and a co-located distributed file system on cluster nodes.

How much of the input data can be read locally depends on many factors: the size of input files and the size of blocks, the number of redundant copies of each block, the number of containers, and the actual placement of both data blocks and containers. Yet, the degree of data locality can have a significant impact on the runtime of jobs and since this degree varies, data locality is a major reason for runtime variance when running data processing jobs in shared large dedicated clusters [55, 56].

3 Related Work

Contents

3.1 Distributed Dataflow Systems and Related Distributed Systems 23
3.1.1 Distributed Dataflow Systems 24
3.1.2 Systems Used in Conjunction with Distributed Dataflow Systems 26
3.1.3 Related Parallel and Distributed Computing Systems 29
3.2 Runtime Prediction and Resource Allocation for Runtime Targets . . . 32
3.2.1 Pure Runtime and Progress Estimation 33
3.2.2 System-Specific Automatic Resource Allocation 33
3.2.3 Resource Allocation Based on Black-Box Prediction Models . . . 35
3.3 Adaptive Resource Management 37

First, we present distributed dataflow systems, the class of systems for which the approach and methods presented in this thesis were designed for. We also present systems typically used in conjunction with distributed dataflow systems, namely distributed file systems and resource management systems. Concluding this first section, we describe related distributed systems for massively parallel computation and the applicability of our solution to these systems.

Second, we present work on predicting runtimes and allocating resources based on such predictions for distributed dataflow systems, explaining major differences to the approach and methods presented in this thesis.

Third, we present related work on adaptive resource management for distributed dataflow jobs. The work presented in this third category is related to our approach and methods in its objectives, automatically optimizing resource usage and adherence to performance goals for distributed dataflow jobs, yet does not apply runtime prediction for allocating resources according to runtime targets.

3.1 Distributed Dataflow Systems and Related Distributed Systems

This section first presents distributed dataflow systems, then the systems typically used in conjunction with distributed dataflow systems, and finally other examples of systems for using highly-parallel computing resources.

Distributed dataflow systems are essential related work as the approach and methods presented in this thesis have been designed for these systems. That is, the models, the similarity matching, and our approach for predicting the runtimes of and selecting resources for job stages target distributed dataflow systems.

As distributed dataflow systems are typically used together with distributed file systems and resource management systems in shared cluster setups, we further describe these systems. The interactions of distributed dataflow systems with both distributed file systems and resource management systems, when multiple jobs run simultaneously on cluster resources, are a major reason for runtime variance. These interactions are, therefore, a motivation for our approach to matching previous runs with a similar runtime behavior as explained in Chapter 6 and our approach to dynamic resource allocation as explained in Chapter 7. Furthermore, we present a system architecture that integrates our approach and methods on the level of the resource management system, thereby supporting different distributed dataflow systems.

Finally, we describe major distributed data-parallel processing systems that do not follow a dataflow model, parallel databases built for analyzing large datasets, and notable other systems used for massively parallel computation.

3.1.1 Distributed Dataflow Systems

Distributed dataflow systems are frameworks for scalable data analytics. The systems provides efficient fault-tolerant distributed runtime environments for executing graphs of connected data-parallel operators on a set of shared-nothing commodity cluster nodes. Typically, distributed dataflow systems execute jobs consisting of multiple stages of data-parallel operators. Such stages consist of subsequent dataflow operators that in principle can run simultaneously.

Accurate scale-out modeling, runtime prediction, and resource allocation based on predicted runtimes is important for distributed dataflows. Programs of distributed dataflow systems are typically highly scalable. This is due to the data-parallel execution model of distributed dataflow systems and the distributed runtime environments of these systems, which include efficient implementations of data-parallel operators and partitioning mechanisms. Consequently, a large number of different scale-outs can be reasonably used for a distributed dataflow job. However, also distributed dataflow jobs exhibit increased overheads with increasing numbers of parallel workers and do not always have completely straightforward scale-out behavior, for example due to skew in the data. Selecting an inadequate scale-out for a job can, therefore, result in lower resource utilization and unnecessary costs. At the same time, distributed dataflow systems make it considerably easier to develop efficient distributed data-parallel processing programs, even for users without extensive knowledge of parallel programming, distributed systems, and data-parallel processing. However, even these users need to allocate an adequate amount of resources for their jobs and performance requirements. For these reasons, accurate runtime prediction and automatic resource allocation are arguably especially important for

distributed dataflow systems and the overall solution presented in this thesis, thus, focuses on distributed dataflow systems.

The key assumptions that we make about distributed dataflow systems on a conceptional level are explained in Section 4.2.2, while practical considerations for using our prototype system Ellis with existing systems are described in Section 4.4.

MapReduce [2, 5] introduced a programming model based on the second-order functions Map and Reduce. Users provide UDFs to these operators, which are then executed in parallel on connected distributed workers. The execution model of MapReduce is based on the alternating execution of the two operators. First, the Map operator is executed data-parallelly, then Reduce is executed data-parallelly. For fault tolerance the intermediate results are exchanged by storing and reading elements to disks in-between the two phases. The Map tasks sort the results by key, so that the Reduce tasks can efficiently read and then reduce defined groups of elements. MapReduce is a limited distributed dataflow system as it does not support general job graphs. A major implementation of the programming and execution model presented in [5] is part of Hadoop[1], which we will refer to as Hadoop MapReduce.

Dryad [57] and Nephele [58] also execute data-parallel tasks on connected worker machines, yet allow to connect these in a general Directed Acyclic Graph (DAG). These tasks can be connected by different communication channels such as, for example, using network channels or disks. Both Dryad and Nephele, however, do not provide a set of predefined operators and differ in this to all subsequent distributed dataflow systems we present here.

SCOPE [23] also allows to use a general DAG for dataflow jobs, yet furthermore provides a set of pre-defined operators, including for example Map, Reduce, and Join. Moreover, SCOPE incorporates optimizer techniques from parallel databases to generate optimized query plans [41, 59]. Consequently, developers can use a high-level scripting language similar to SQL to write SCOPE programs.

Stratosphere [60] is a platform for distributed data-parallel processing built on top of the Nephele runtime, adding a set of pre-defined operators with Nephele/PACTs [61]. Furthermore, the Stratosphere platform uses optimization techniques to generate efficient query plans from a high-level scripting language [42, 44, 62]. Stratosphere also supports incremental processing of converging iterative dataflow programs natively [30], effectively allowing cyclic dataflow graphs.

Naiad [63] is a similar distributed dataflow system in that it too supports incremental processing of converging iterative programs natively. Beyond this Naiad can also incrementally re-compute results when inputs change, using an approach called *Differential Dataflow* [64].

Spark [6, 65] is a similar system as SCOPE and Stratosphere, also allowing to create dataflow programs using operators like Map, Reduce, and Join connected to general

[1] `https://hadoop.apache.org/`, accessed 2018-02-16.

acyclic job graphs. Spark's key feature is its abstraction for distributed datasets called Resilient Distributed Datasets (RDDs) [66]. RDDs implement fault tolerance using lineage. This can speed up distributed computation significantly, compared to snapshotting intermediate data to disk. Furthermore, RDDs can be cached for faster interactive and iterative processing. Spark also provides higher-level programming abstractions, including for processing relational data, in which case Spark also optimizes query plans automatically [47, 48]. We used Spark for the experiments presented in Chapter 5, 6, and 7.

Spark Streaming [67] is a distributed streaming system that is based on the batch processing engine of Spark. For this, Spark Streaming uses micro-batches.

Flink [7], which originated from Stratosphere, uses a unified distributed execution engine for both batch and stream processing. It provides low latency stream processing, including exactly-once-guarantees, but can also be used for batch processing. Flink implements fault tolerance for distributed stream processing by periodically taking snapshots of the operators' state. We used Flink for the experiments presented in Chapter 5.

Google's Dataflow [68] is a distributed stream processing system, which also can be used for batch processing as well. Dataflow is unique in that it includes a set of concepts and core principles for dataflow systems in the context of unbounded, unordered data stream inputs.

Given this considerable number of similar data analytics systems, Tez [69] was developed as a framework for building distributed dataflow systems.

3.1.2 Systems Used in Conjunction with Distributed Dataflow Systems

This section describes systems typically used together with distributed dataflow systems. We first present distributed file systems, then resource management systems.

3.1.2.1 Distributed File Systems

Distributed file systems are systems that provide distributed data storage by splitting large files into small blocks, which are then stored redundantly across a set of connected shared-nothing commodity nodes. The blocks can be accessed in parallel on these *data nodes*, as done by distributed data-parallel processing systems. How much of a large file is locally accessible to a distributed data-parallel processing system, which is called the degree of data locality, can however vary, especially when files are distributed across a large set of nodes and resources on only a subset of nodes are used for a distributed data-parallel processing job. Even when the algorithms that schedule containers and task instances take data locality into account, multiple reasons can lead to varying degrees of data locality. For example, the resources with the highest degree of data locality might not currently have sufficient available capacities in shared cluster infrastructures. Thus, data

locality varies, leading to runtime variance that needs to be addressed when distributed dataflow jobs should meet runtime targets with minimal resources.

Google File System (GFS) [70] is a distributed file system optimized for distributed data processing. That is, GFS focuses primarily on very large files and high throughput for sequential reads and writes. The file system uses a single master node. This master node handles the metadata and coordinates data storing as well as data access. A major implementation of the design of GFS is part of Hadoop and called Hadoop Distributed File System (HDFS) [71]. We used HDFS for the experiments presented in Chapter 5, 6, and 7.

Ceph [72] is a distributed file system that provides a near-POSIX interface to its clients. The file system manages metadata using a distributed metadata cluster for improved scalability.

GlusterFS [73] provides a fully POSIX-compliant distributed file system. It does not maintain any metadata servers. Instead, data is located algorithmically using an elastic hashing algorithm.

Alluxio/Tachyon [74] is an in-memory distributed file system. As such, it provides high read and write performance for data-local tasks. Tachyon does not rely on replication for fault tolerance. Instead, lineage data is used to recompute lost datasets. In addition, a checkpointing algorithm is used to provide an upper bound for the time necessary to recompute data.

There are also other distributed data storages that can be used with distributed dataflow systems. Spark, for example, is able to ingest data from distributed file systems, distributed databases, and distributed object stores[2]. Moreover, the methods we use to predict runtimes and to address runtime variance do not assume input data to be read in from a distributed file system. Specifically, while we recognize varying degrees of data locality as a major reason for runtime variance, which does occur with distributed file systems, we do not make any assumptions about exact data distribution or replication. At the same time, we do expect distributed dataflow systems to be typically used in conjunction with distributed file systems and, therefore, that runtime variance due to varying data locality needs to be addressed.

3.1.2.2 Resource Management Systems

Resource management systems allow fine-grained sharing of cluster resources among multiple jobs, users, and distributed processing frameworks. That is, users temporarily reserve cluster resources in so called containers, which are logical leases of resources on particular nodes. These containers are often used without resource isolation to achieve high degrees of overall resource utilization despite the fluctuating resource usage of sin-

[2] Spark has multiple source connectors including for example for the local file systems, HDFS, Cassandra, and Amazon S3 `https://spark.apache.org/docs/2.0.0/programming-guide.html#external-datasets`, accessed 2018-02-19.

gle distributed data-parallel jobs. Yet, the overall cluster utilization and job performance depend on which job combinations share resources as there can be significant interference between workloads. This interference between co-located jobs in shared dedicated analytics clusters has been identified as a major reason for runtime variance [11, 14, 75], which needs to be addressed when minimal resources are to be used to meet runtime targets.

The resource management system Mesos [8] allows users to run jobs of multiple distributed processing frameworks efficiently in a single shared cluster. Mesos uses 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. When these offers are accepted, the frameworks schedule tasks onto allocated containers. A key advantage of delegating container placement to processing frameworks is that the frameworks can optimize for goals like data locality with considerably more assumptions regarding the execution model.

YARN [9] was developed to allow Hadoop clusters to be used with other distributed dataflow frameworks than just Hadoop MapReduce. 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. Similar to Mesos, YARN also moves scheduling functions towards per-job components for increased scalability. In contrast to Mesos, however, processing frameworks do not receive resource offers, but request resources from YARN. We integrated our prototype system Ellis with YARN as explained in Section 4.4.3 and used YARN also for the experiments presented in Chapter 7.

Besides multi-tenancy, the main goals of YARN and Mesos were thus enabling users in using multiple frameworks and in providing more scalable scheduling solutions. Other work towards more scalable schedulers for analytics workloads include Omega [76], Apollo [77], and Borg [78]. All three solutions are based on decentralized schedulers that are loosely coordinated using optimistic concurrency control.

Omega [76] is based on a lock-free protocol where all participating distributed schedulers can operate in parallel. In Omega all schedulers have full knowledge of the cluster state by holding a copy of the shared state. Acquiring resources is based on optimistic concurrency. Schedulers transactionally update the shared state and rollback when collisions are detected.

Apollo [77] is a resource manager where each scheduler has access to the global cluster state. When a task is scheduled, it is put into the task queue of the respective node. Each node maintains a wait-time matrix that estimates when specific resource configurations will be available given the tasks in the queue. The job schedulers combine this matrix with other factors like data locality and task priority when making scheduling decisions. Similar to Omega, Apollo relies on optimistic concurrency control. However, Apollo features a late collision resolution where task assignments are checked after they are already put into the queue.

Borg [78] uses a centralized resource manager that is replicated to achieve high availability and multiple distributed task schedulers. Submitted jobs are put into a pending queue. The distributed schedulers use the shared cluster state to process the job queue asynchronously and inform the master about task assignments. Optimistic concurrency is employed as the master checks the assignments.

We present our solution in the context of resource management systems and container reservations. However, our approach and methods are also applicable for allocating entire nodes or even virtual machines from a cloud provider, given recurring distributed dataflow jobs and homogeneous worker capabilities, so only the scale-out needs to be determined. Even when entire nodes are reserved, there is runtime variance due to for instance shared network links in hierarchical datacenter networks, failures of workers, and potentially varying degrees of data locality. Similarly, while virtual machines provide resource isolation, there is still interference with co-located workloads, for example when accessing shared I/O [79]. Therefore, even in these execution environments, runtime variance would need to be addressed. The methods we use to address runtime variance, namely matching of similar previous executions as a basis for runtime prediction models and dynamic adjustments of resource allocations after assessing a job's progress towards its runtime target, are not specific to resource-managed clusters and job execution in temporarily reserved containers. At the same time, we do expect production batch jobs that are scheduled periodically, such as on a daily or even hourly basis, and have specific runtime targets to typically run in dedicated shared clusters.

3.1.3 Related Parallel and Distributed Computing Systems

This section describes related systems for massively parallel computation besides distributed dataflow systems. We present other distributed data-parallel processing systems, describe parallel databases, and discuss high-performance computing systems. For each of these three categories we highlight differences to distributed dataflow systems and describe the applicability of our approach and methods.

3.1.3.1 Other Distributed Data-parallel Processing Systems

Many systems similar to distributed dataflow systems have been developed for large-scale distributed data-parallel computations. These systems have key characteristics in common with distributed dataflows: Data-parallel computations are executed on data that has been partitioned among parallel workers, typically running on clusters of commodity nodes. Moreover, these systems also usually provide restricted programming models that work in defined subsequent steps or iterations. For example, PowerGraph [26] requires users to express algorithms in its Gather, Apply, Scatter (GAS) programming model, while BSP steps are called *super-steps*. At the same time, these systems are fundamentally different in that they do not adhere to a dataflow execution model. Specifically, there is no re-partitioning and shuffling of the intermediate data in-between sub-

sequent stages of data-parallel operators. Instead of data flowing through distributed data-parallel tasks in this way, data is typically partitioned initially and then messages are exchanged between the workers that hold partitions of the data. Examples for distributed data-parallel processing systems include for instance multiple systems for the analysis of large graph structures such as Pregel [25] and the mentioned PowerGraph. Other examples of distributed data-parallel processing systems include systems for distributed machine learning such as Parameter Server [28], GraphLab [80], Distributed GraphLab [27].

The modeling and the similarity matching techniques we presented in Chapter 5 and 6 are principally applicable as long as there is scalable distributed data-parallel processing and, therefore, the question of how many resources to use for a job with a specific runtime target. However, without re-partitioning and shuffling of the intermediate data in-between subsequent stages, dynamically allocating resources for each stage as Ellis does is not directly applicable. Moreover, the similarity measures of Cutty that capture the runtime behavior of jobs such as the runtimes and the average resource utilizations of stages are only applicable when there are defined subsequent processing steps like stages. Still, it should be possible to use the approach and methods presented in this thesis to allocate resources initially for distributed data-parallel processing systems besides distributed dataflow systems. That is, Bell can be used to model the scale-out behavior of entire jobs, instead of using it at the granularity of stages as presented in Section 7.1. Such models could be trained on similar previous executions of jobs selected using the methods we presented in Section 6.2 for Cutty, only using similarity measures applicable for the execution model at hand. Trained job-level scale-out models could then be used to allocate resources for jobs on submission, following the greedy approach presented for Ellis in Section 7.2.1.

A special case are distributed data-parallel processing systems that do not provide a dataflow programming model, yet in the end translate programs to the execution model of distributed dataflows such as GraphX [49, 81], a distributed graph processing system built on top of Spark. For these systems our approach and methods should be fully applicable, including runtime adjustments after estimating remaining runtimes, when integrated on the level of distributed dataflow execution.

3.1.3.2 Parallel Databases

Database systems allow to store and to query data. The most prominent group are relational database management systems (RDBMSs), which are database systems that operate on relational data and offer declarative query languages. Indexing and automatic plan optimization of declarative queries are used to make query execution efficient. Some of these systems are optimized for transactional workloads, which consist of large numbers of insertions, updates, and retrievals of single elements. Other systems are instead optimized for analytical workloads, which consist of more complex queries, typically aggregating large groups of elements. This latter class of RDBMSs is more related to distributed dataflow systems than the class optimized for transactional data management.

Parallel RDBMSs provide a single-instance interface, yet run on multiple machines that jointly store and query data. Teradata and Tandem [82, 83] are examples of early commercial parallel RDBMSs. More recent systems include Greenplum [84] and Aster Data [85]. These systems support massively parallel query execution and also feature efficient fault-tolerant runtimes. Furthermore, they support more generic programming models than pure SQL, so that users can include user code into queries. Moreover, some databases feature full dataflow engines such as AsterixDB [86] with its Hyracks distributed dataflow engine [87], effectively combining distributed dataflow systems with data management capabilities such as storage and indexing.

At the same time, many techniques for efficient data-parallel processing that have been developed for databases and parallel databases have been transferred to distributed dataflow systems. These techniques include high-level and declarative query languages, automatic query optimization, as well as distributed data-parallel operators and partitioning methods. Examples of distributed dataflow systems that support these techniques include Dryad with DryadLINQ [40], Hadoop MapReduce using Pig [45] or Hive [46, 88], Spark using Shark [47] or SparkSQL [48], SCOPE [41, 59], and the Stratosphere platform [42, 44, 62].

The main difference between parallel databases and distributed dataflow system remains that databases also store and manage data, while distributed dataflow systems only provide distributed data-parallel processing. The approach and methods presented in this thesis is in general applicable to processing engines that work similarly to distributed dataflow engines. Thus, when there are subsequent job stages of data-parallel operators that can be modeled and monitored as well as data that is being re-partitioned and shuffled between these subsequent stages, so different scale-outs and sets of resources can be used for each stage, our solution should be usable for dynamic resource management for parallel database engines.

3.1.3.3 Systems for High-Performance Computing

HPC can be characterized as using high performance hardware with low-level programming abstractions for problems with low latency requirements. Users typically express both parallelism and inter-process communication explicitly. In comparison, distributed dataflow systems feature a more restricted data-parallel programming model, automatic task parallelization and distribution, typically a set of pre-defined data-parallel operators, and also more comprehensive fault-tolerant distributed runtime environments. The problems solved with HPC technology also often exhibit more coupling between dependent computations, compared to the problems that naturally fit distributed dataflow abstractions.

Among many techniques used for HPC, Message Passing Interface (MPI) [32] is the most prominent. MPI is a specification of an interface for message passing among parallel workers. It supports point-to-point and global communication primitives. Implementations of this standardized interface are available for different hardware platforms, in-

cluding for different methods for communication, ranging from TCP and shared memory to high-performance networks such as InfiniBand. That is, while MPI can be used with commodity hardware, there are also optimized implementations for high-performance computers and networks. Distributed dataflow systems, in comparison, have been explicitly designed for shared-nothing commodity nodes and Ethernet [2, 36]. Furthermore, MPI only defines message passing among workers, which makes it applicable to more classes of problems, in comparison to the restricted programming model that distributed dataflow systems offer.

The approach and methods presented in this thesis are not generally applicable to MPI programs. First, we assume the data parallelism and the scalability of distributed dataflows that results from the restricted programming model, automatic parallelization, and effective data partitioning. This scalability not only first poses the question of which scale-out and number of containers to use for a job and a given runtime target, but is the basis for our scale-out models, specifically the parameterized model of distributed processing that we use. Second, we assume an execution of stages as well as re-partitioning and shuffling in-between subsequent stages. This execution model allows to monitor, model, match previous executions, and allocate resources at the granularity of individual stages. Due to these assumptions our approach is not generally applicable to MPI and other HPC technologies such as OpenMP [34] and PVM [33].

3.2 Runtime Prediction and Resource Allocation for Runtime Targets

This section presents approaches and systems, including the state of the art, for estimating the progress, predicting runtimes, and allocating resources according to runtime targets for distributed dataflow jobs. As these solutions are alternatives to the approach and methods presented in this thesis, we compare them to Ellis, Bell, and Cutty.

First, we present work on purely estimating the runtime or progress of a distributed dataflow job. The major difference to this work is that, while Ellis also estimates the progress by predicting the remaining runtime of a distributed dataflow job, Ellis further uses its estimations to dynamically allocate resources to meet a specific runtime target.

Second, we present work on allocating resources for runtime targets that is specific to particular distributed dataflow systems. The major difference to this work is that we use black-box models to capture the scale-out behavior of jobs and, thereby, support different distributed dataflow systems.

Third, we present work on allocating resources for runtime targets using black-box prediction models. These solutions are most comparable to our approach and methods. For each of these approaches and systems, we therefore describe major differences to Ellis, our main prototype system. The shortcomings in comparison to our solution can be summarized as ineffective training, static allocation, and impractical scope as explained in Section 4.1.

3.2.1 Pure Runtime and Progress Estimation

This section presents approaches and systems for estimating the runtime or progress of distributed dataflow jobs. Compared to our approach and methods, the related work presented here is either specific to MapReduce, application domain-specific, or otherwise not generally applicable to distributed dataflow jobs, while Ellis uses black-box models that support different distributed dataflow systems and application domains. Moreover, Ellis uses its runtime prediction and progress estimation to dynamically manage resource allocations in order to meet a specific runtime target despite considerable performance variance.

Parallax and ParaTimer estimate the runtime of MapReduce jobs. Parallax predicts the runtime of sequences of MapReduce jobs compiled from Pig programs [45], which is an SQL-like programming abstraction compiling to Hadoop MapReduce programs. The runtime prediction of Parallax uses a simple model of parallelism, cardinality estimates from Pig's query optimizer, and profiling runs on user-defined samples of the input data. ParaTimer builds upon Parallax. It extends Parallax in that it allows a general DAG of MapReduce jobs, not just sequences. For this, ParaTimer identifies the critical path in a job graph 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 scenarios.

PREDIcT [89] estimates the runtimes of 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: the distributed dataflow job is executed on a subgraph and key characteristics such as the number of messages sent per iteration are extrapolated to the whole dataset. However, this only works if the subgraph preserves the relevant properties for the given algorithm. Also, any algorithm parameters must be adjusted for the sample run by the user.

There is also work on estimating the progress of distributed dataflow jobs by propagating the progress through the job graph [90]. In particular, this approach relies on adding specific markers to the input datasets, which are then used to detect when a particular percentage of the original input data has reached a certain task in the dataflow graph. The approach supports multiple inputs to tasks by averaging the progress information received by all predecessor tasks and then propagating this average to all successor tasks. The applicability of this approach is, however, limited as it can only be used for fully pipelined jobs, yet does not support jobs containing operators like a Reduce or a Join. That is, the approach can only indicate the progress of a single stage of a distributed dataflow job.

3.2.2 System-Specific Automatic Resource Allocation

This section presents approaches and systems for resource allocation based on predicted runtimes, yet these solutions use white-box models supporting only specific distributed

dataflow systems. In comparison, Ellis uses black-box models for capturing the scale-out behavior of distributed dataflow jobs by relying on Bell, as explained in Chapter 5, and also uses similarity matching techniques generally applicable to distributed dataflow systems by using Cutty for selecting training data, as explained in Chapter 6. Furthermore, the system architecture presented in this thesis integrates our approach and methods on the level of the resource management system, not a specific distributed dataflow system.

Aria [16] uses a simple MapReduce performance model to select resources given an input data size and runtime target. Specifically, Aria estimates the amount of execution slots necessary for meeting runtime targets, using its system-specific performance model and detailed runtime statistics from previous runs. Aria also estimates the impact of failures on job runtimes and it implements a SLO-based scheduler for Hadoop that submits jobs ordered by their deadline. Aria was first presented to work with both historic data and dedicated profiling, yet the authors later built on their work [91], adding dedicated sample runs to explore how well jobs scale for different input sizes.

Elastisizer [92], part of the Starfish [93] system for automatically tuning Hadoop clusters, answers cluster sizing queries of users for MapReduce jobs. Given detailed job profiling information, Elastisizer simulates the runtime and costs of executing a job using a certain configuration using relative modeling [94]. For this, the user needs to specify the search space such as available resource types and system configuration options.

AROMA [17] is a system for provisioning MapReduce jobs with deadlines in clouds. AROMA first 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 heterogeneous resources and to configure different system parameters. The performance model is based on regression accompanied by a systematic feature selection. 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.

Bazaar [95] also uses a simple MapReduce performance model to assign resources to jobs. It predicts job runtimes using a function of the scale-out, the network bandwidth, and input size. Bazaar uses sample runs to profile jobs and gather necessary runtime statistics for its performance model. It then selects the number of instances and network bandwidth to assign to a MapReduce job given a runtime target. Bazaar uses additional slack to address performance variance and failures.

The Jockey resource allocator and scheduler [14] was built for SCOPE [23]. Jockey uses a simulator and detailed job statistics from previous runs to predict the runtime of a job's stages. For selecting resources, users have to specify a utility function, which is used to model deadlines and penalties. Using precomputed simulations and an estimation of a job's progress at runtime, Jockey adapts resource allocations dynamically when a job is not performing as predicted.

OptEx [96] estimates the runtime of Spark jobs given the size of the input data, the number of iterations, and the number of nodes. OptEx has developers categorize jobs into distinct application classes and select a representative job per category. OptEx then

creates a profile per job category by running the representative job. Each of these profile is then used to extrapolate from short sample runs on a single node to larger scales for jobs of the category.

3.2.3 Resource Allocation Based on Black-Box Prediction Models

This section presents approaches and systems that can be used for automatic resource allocation, are generally applicable for distributed dataflow systems, and use runtime prediction to select scale-outs that meet runtime targets. These solutions are the most similar related work to the approach and methods presented in this thesis. We therefore provide a short comparison to using Ellis with Cutty and Bell for each of these approaches and systems.

Quasar [15] uses classifications to determine the impact of resources and interference with other workloads when scheduling jobs to heterogeneous cluster nodes. Quasar takes users performance requirements into account and selects resources for these performance goals. For batch analytics jobs, Quasar lets users express runtime targets, while Quasar also supports latency-critical user-facing applications, for which latency requirements can be expressed. Quasar jointly performs resource allocation and assignment, before continuously monitoring application performance at runtime and, if necessary, adjusting resource allocations dynamically. For this, the system re-classifies jobs when their performance deviates from expressed goals and adjusts resource allocations at runtime when classification assignments change. Quasar does, however, assume full control over both resource allocation and assignment. It further does not use models specific for distributed dataflow systems, yet is a fully generic system for all kinds of applications running in datacenters. Moreover, using Cutty we select similar previous executions as a basis for performance prediction, while Quasar requires dedicated profiling runs.

Ernest [97] is a job submission tool that automatically allocates cloud resources for distributed dataflow jobs with given runtime targets. When a job is submitted, Ernest runs the job on subsets of the input data and different sets of resources, training a simple model of distributed processing. The combinations of samples are selected using ideas from *optimal experiment design* [98]. In comparison, Ellis does not rely on isolated dedicated training runs, but aims to make the most of available workload data of recurring jobs using Bell and Cutty. Moreover, while Bell's parametric regression model is based on the model proposed with Ernest, Bell automatically switches to nonparametric regression when the available data and prediction task allow this. Ellis, furthermore, also monitors jobs at runtime and, if necessary, adjusts resource allocations to meet runtime targets despite performance variance. For this, Cutty also enables us to select previous runs with a similar runtime behavior, so the current cluster state is reflected in updated predictions at runtime.

PerfOrator [18] predicts the runtime performance of distributed dataflow jobs based on multiple profiles. First, PerfOrator executes a set of pre-defined queries to model different hardware resources, capturing the time it takes to perform reads, writes, and

data shuffling. Second, it takes as input a parallelization profile of the distributed dataflow system, currently supporting Hadoop MapReduce and Tez, but with the possibility to support further systems by describing the parallelism of their execution models. Third, Perforator profiles incoming jobs on samples of the data. Finally, it uses all three profiles and non-linear regression to translate performance requirements into resource allocations. PerfOrator requires more information than many black-box systems such as runtimes and CPU cycles spent processing for each stage measured in profiling runs. PerfOrator is, however, then able to make predictions for job runtimes and resource skylines on different hardware resources, whereas our solution focuses on homogeneous clusters. In contrast to PerfOrator, Ellis monitors job progress at runtime and adjusts allocations dynamically, if necessary, thereby addressing performance variance. Furthermore, we use Cutty to match similar previous executions of recurring production jobs and avoid dedicated profiling runs. Cutty also allows to update prediction models at runtime based on previous executions with similar runtime behavior, effectively incorporating the current cluster state into predictions at runtime.

Justice [99] is a resource allocator and admission control system for resource management systems like YARN and Mesos, aiming to fulfill completion deadlines in shared resource-constrained cluster environments by resource allocation, admission control, selecting from queued jobs, and stopping running jobs. Based on historic data from previously executed jobs, Justice uses a black-box prediction model to allocate resources for jobs. Justice prevents jobs likely to miss their deadline from starting and consuming resources wastefully. It also tracks jobs that already violate their deadline and selectively drops some of these to avoid further waste of resources. In comparison to our approach and methods, Justice optimizes the overall fulfillment of job deadlines in a resource-constrained cluster environment, not meeting a single job's runtime target with minimal resources. Consequently, Justice also assumes control over job admission, the execution order of jobs, and the ability to stop jobs. In contrast to our methods, Justice does neither adjust prediction models nor resource allocations for specific jobs at runtime.

CherryPick [100] is a system for allocating cloud resources for distributed analytics jobs according to users' performance goals. Given a user's constraints and a representative workload, CherryPick searches for the optimal cloud configuration in terms of number of virtual machines and type of virtual machine instance. A user's constraints can include a cost budget and a maximum running time as well as a preferred instance type and constraints for the overall cluster size. CherryPick builds a performance model using dedicated sample runs. For this, it uses the Bayesian Optimization framework along with a Gaussian Process. Since Bayesian Optimization can estimate a confidence interval for its predictions, CherryPick can judge which cloud configuration it should profile next and when to stop sampling. Compared to our solution, CherryPick aims at selecting among heterogeneous cloud resources on the basis of dedicated profiling runs, while we use Bell and Cutty to model the scale-out behavior of recurring jobs based on similar previous executions. Furthermore, Ellis continuously monitors a job's progress towards

its runtime target and addresses runtime variance by adapting resource allocations dynamically, if necessary.

3.3 Adaptive Resource Management

This section presents related work on adaptive resource management for distributed dataflow jobs. The approaches and systems presented here use runtime statistics, either recorded in previous runs of recurring jobs or dedicated profiling runs, to model the behavior of distributed dataflow jobs. The resulting job profiles are subsequently used to adaptively schedule containers and tasks for optimal data locality, interference between co-located workloads, bandwidth between task instances that exchange most data, or resource utilization. Compared to our approach and methods, the work presented in this section is related in its objectives, aiming at automatically configuring resource usage for optimal utilization and adherence to performance goals, yet does not estimate the runtimes of jobs or allocate resources according to runtime targets, in contrast to Ellis.

SCOPE [23] alters the task parallelism at runtime based on recorded statistics [43, 101]. SCOPE also uses these statistics to continuously optimize program plans, to select optimal physical operators, and to adapt the partitioning at runtime. These optimizations are, however, only based on data statistics and the solution selects the configuration of the next stage based on statistics from the previous stage. That is, SCOPE does not employ scale-out models trained on previous or sample runs.

ThroughputScheduler[102] is a scheduler for Hadoop workloads that uses a Bayesian learning scheme to adaptively match jobs to nodes with heterogeneous compute capabilities. It derives server capabilities by initially running a set of probe jobs on the available hardware. The requirements of submitted jobs, on the other hand, are determined on the fly. For this, the ThroughputScheduler uses a MapReduce-specific performance model.

Paragon [103] profiles incoming jobs, matches them to classes of similar jobs with respect to the impact of heterogeneous hardware and interference with co-located workloads, and uses these classifications to assign jobs to specific cluster resources. Paragon performs collaborative filtering, specifically singular value decomposition, to identify similarities between new and previously executed jobs. It then minimizes interference and maximizes resource utilization when scheduling applications based on its model of the impact of heterogeneity and workload interference.

In the context of distributed dataflow systems that process continuous inputs, there is work for Storm [104] that adaptively places and migrates tasks at runtime based on recorded traffic statistics [105]. This online task scheduler for Storm continuously monitors the traffic between tasks of the jobs graphs and repeatedly calculates which assignment of tasks to worker nodes yields the minimal inter-node network traffic. When there is a new schedule with less network traffic, the online scheduler applies it transparently to the Storm cluster.

Gemini [106] is an adaptive scheduler for the YARN resource management system. Gemini creates a model of the performance improvement and the fairness loss when combinations of jobs share cluster resources without resource isolation. This model reflects the degree of complimentary resource demands of job combinations. It uses historic data on the executed workload of a cluster to train this model. Gemini then automatically calculates the fairness loss of using a policy that only aims at optimizing the resource utilization of the cluster instead of Dominant Resource Fairness [107] and then decides whether fairness should be sacrificed for performance based on a user's setting of required fairness.

Nephele Streaming [108] is a distributed dataflow system based on Nephele [58], yet for processing continuous input streams. Nephele Streaming takes latency constraints into account and optimizes the resource usage adaptively towards fulfillment of these SLOs, while also attempting to use as few resources as possible. For this Nephele Streaming mainly employs three strategies: adaptive output buffer sizing, dynamic task chaining, and dynamic scaling [109]. Dynamic buffer sizing adaptively sets the size of output buffers, which are filled before output elements are send to another worker. Smaller output buffer sizes translate to lower latencies as output is sooner transmitted to subsequent tasks. However, this reduces throughput as more packets need to be transmitted. Task chaining merges certain tasks into the same thread, eliminating the need for queues and data handover, which reduces the latency, yet also at the cost of throughput, as fewer threads run in parallel. Dynamic scaling, which Nephele Streaming does automatically based on a queueing theoretic latency model, uses more compute resources to alleviate compute bottlenecks. That is, dynamic scaling is an optimization that does not decrease throughput, but increases costs.

Morpheus [11] is a system for resource allocation and scheduling of periodically running jobs. Morpheus infers deadlines for these repeatedly scheduled batch jobs from execution logs. It then uses recorded resource utilization data from previous executions of these jobs to infer the maximal usable resources for the jobs, effectively assuming overprovisioning by users. Subsequently, Morpheus allocates these resource skylines for subsequent runs. Morpheus further attempts to run scheduled jobs together with the same other periodical jobs to decrease runtime variance due to interference between co-located workloads. Morpheus also monitors jobs at runtime and dynamically adjusts resource allocations when resource usage deviates from expected demand.

Another approach for scheduling recurring batch jobs adaptively based on resource usage and interference applies reinforcement learning [53]. The approach learns over time which combinations of recurring jobs exhibit the least interference and achieve the highest resource utilization when running co-located on the same nodes. For its measure of co-location goodness, the approach takes CPU, disk, and network usage as well as I/O wait into account. It then uses its model of co-location goodness to schedule the job that utilizes the available resources best given the jobs currently running on the cluster.

4 Problem and Concepts

Contents

4.1 Problem and State of the Art 39
4.2 Assumptions and Requirements 42
4.2.1 Batch Processing Jobs 42
4.2.2 Distributed Dataflow Systems 42
4.2.3 Dedicated Analytics Clusters 44
4.2.4 Requirements for a Practical Solution 45
4.3 Approach and Methods 45
4.3.1 Solution Overview 45
4.3.2 Application to Iterative Jobs 49
4.4 System Architecture 50
4.4.1 Architecture Overview 51
4.4.2 Prototype Components 52
4.4.3 Integration with YARN and Spark 55

This chapter describes the problem and concepts of this thesis. We first present the problem we address with this thesis in detail, including the state of the art and its critical limitations. Subsequently, we discuss observations that we treat as assumptions for a practical solution of the identified problem. Then we continue with an overview of our methods, which are presented in full detail in the following three chapters. The chapter concludes with presenting our system architecture, the implementation of our prototype components, and how these integrate with existing systems.

4.1 Problem and State of the Art

Estimating the performance of distributed dataflow jobs upfront is difficult [11, 17, 18, 95]. This is due to the many factors the runtime performance depends on, including complex task dependencies, arbitrary user-defined functions, program parameters, dataset characteristics, properties of physical hardware as well as virtualization, and system configuration. Anticipating the impact that each of these factors has on job performance is hard. Moreover, for adhoc data processing, there is often little information available for some of these factors before processing. For example, there are often no detailed

data statistics available for files processed directly from distributed file systems. However, without detailed knowledge of data characteristics such as key value distributions, which would indicate data skew, it is unclear how well a particular data partitioning scheme will work for a specific level of parallelism.

Moreover, besides the difficulty of estimating runtimes, there is also significant variance in job performance [11, 14, 110]. Even when running the same program on the same dataset with an equal set of containers, the runtime of the job can vary considerably. A study of the variance of recurring jobs on a production cluster at Microsoft [14] showed that even runs that processed similar-sized input datasets exhibit considerable variance: the executions of 50% of the recurring jobs had a coefficient of variation of 0.20 for executions processing inputs that differed at most 10% in size. That is, the executions of half of the jobs deviate from the mean runtime by on average 20%. Some of this variance is due to factors emerging from sharing of resources like varying degrees of data locality [55, 56], competing access to data [75, 111], and interference between jobs through resource contention as well as adjacent usage of spare resources [14, 53]. Besides this sharing-induced variance, there is also inherent variance in runtimes due to stragglers, worker failures, and updates to data and code [11, 13, 112–118].

However, users often have specific requirements for the performance of their production jobs [10, 11, 14]. One real-world example for an expressed requirement from the company Twitter is updating the indices for search query completion on terabytes of log data within a maximum of ten minutes [12]. These SLOs are regularly formulated in SLAs, which are SLOs that have been negotiated and agreed upon by the stakeholders, typically with financial penalties due in case of noncompliance. Thus, missing performance objectives often has not just a negative effect on the usability of services, but also a financial impact. Even if SLOs have not been formally recorded, users often have clear expectations for the performance of their jobs. Arguably, expressing a runtime target for a job is also much more tangible for users than selecting resources [95].

Given specific runtime targets despite the fact that users have a hard time in anticipating the performance of distributed dataflows, users considerably over-provision resources for their important production jobs. This results in low overall utilization of cluster resources, with industry-wide utilization being estimated to be somewhere between 6% and 12% [15]. Studies of the resource utilization of cluster resources used exclusively for scalable data analytics at particular companies report higher average resource utilization, specifically values ranging from below 20% to 35% for CPU utilization and 40% to 50% for memory utilization [13, 15]. However, the reserved capacities were still reported to be two to five times higher than the actually used capacities. Resource utilization this low does not only yield unnecessarily high costs for operation and maintenance, but also needlessly large infrastructures in the first place. Furthermore, over-provisioning is problematic in regard to energy consumption and scalability. Overall, users often have no problem in meeting their runtime targets, yet they over-provision significantly and, thus, are ineffective in allocating minimally necessary sets of resources for their goals.

A lot of previous work addressed the problem of meeting SLOs, especially runtime targets, despite the difficulty of anticipating the performance of distributed dataflow jobs. These previous efforts fall short in at least one of the following four categories:

System-specific models A lot of these systems [14, 16, 17, 91, 92, 95, 96] were designed for specific distributed dataflow systems, while resource-managed clusters typically run multiple distributed dataflow systems [8, 9]. Moreover, even detailed performance models are imprecise given the runtime variance in shared clusters, yet capturing detailed statistics requires extensive instrumentation and therefore can impose overheads on job runtimes. For example, capturing detailed key value distributions is an instrumentation that adds significant overheads [119], but these statistics are important for many cost models [41].

Ineffective training Many systems that model job performance are based on dedicated isolated profiling runs using small scale-outs and samples of the input [15, 18, 91, 95–97], even though extrapolating to full datasets and larger scale-outs is not straightforward [89]. These training runs of course impose an overhead not just in terms of time, but also in terms of cluster resources required solely for model training. Some systems instead model performance using previous executions as training data, yet apply simplistic mechanisms for selecting relevant previous runs as a basis for prediction models [11, 99]. These systems match only metrics available before job execution, not capturing factors that are hard to predict but reflected in runtime statistics and can have a significant impact on runtimes. Examples include the convergence of iterative jobs, which can for instance depend not just on dataset characteristics, but also considerably on program parameters.

Static allocation Many of the previously presented systems only statically allocate resources once prior to execution and do not monitor the performance of jobs at runtime [16–18, 91, 92, 95–97, 99, 100]. However, given the inherent runtime variance of distributed dataflow jobs in shared environments, these systems either need to conservatively over-provision resources or users have to accept runtime target violations due to performance variance.

Impractical scope Some approaches also assume control over more than just resource allocation [11, 15, 16, 99]. These systems for example additionally schedule jobs or place containers onto nodes, yet switching to entirely different schedulers or even resource managers is usually an immense effort for organizations.

To the best of our knowledge there is no practical solution that (1) supports distributed dataflow jobs in general, (2) using runtime prediction models trained on specifically selected samples, and that (3) also monitors jobs at runtime and dynamically adjusts resource allocations, if necessary. In addition to these three main points of criticism of the state of the art, solutions also fall short as (4) they arguably unnecessarily require control over more than just resource allocation.

4.2 Assumptions and Requirements

In this section we first discuss those observations that we treat as assumptions regarding batch processing jobs, distributed dataflow systems, and dedicated analytics clusters. Then, we derive requirements for a practical solution for the previously described problem of users having difficulties to allocate minimally necessary sets of resources for their runtime targets.

4.2.1 Batch Processing Jobs

For batch data processing jobs we make two key observations:

- Users often have runtime targets for their production batch jobs.
- Production batch jobs are often recurring such as scheduled periodically on an hourly, daily, or weekly basis.

Users often have specific performance requirements or goals for their batch production jobs [10, 11, 14]. A study of the analytics batch jobs executed on a Microsoft cluster not only shows that most jobs that are executed are production jobs, but also that most complaints in form of ticket escalation are related to job performance or performance predictability [11]. Some of these performance objectives stem from contractual agreements with external partners, typically ensuing financial penalties if missed. Other performance objectives are derived from usability requirements. For example, user-facing online content such as current search trends has to be up-to-date to be useful. Otherwise, users will eventually stop using the services of a vendor and move on to competing offers, ensuing revenue loss. Even if neither external partners nor end-users are directly affected, users of analytics clusters often have notions of desired runtimes. At the very least, it is almost always easier for them to state runtime targets than to decide on appropriate resources for their jobs.

Production jobs are often recurring. These are typically periodically scheduled batch jobs that run repeatedly on updated or at least similar datasets [10, 11, 14, 16]. Such jobs have been reported to make up 60% of the larger clusters at Microsoft [11]. Of these jobs more than 40% run on a daily basis, while other frequently used periods are fifteen minutes, an hour, and twelve hours. Another study showed that recurring jobs make up 40.32% of the jobs as well as 39.71% of the cluster hours on all production clusters used for Microsoft's Bing service [10].

4.2.2 Distributed Dataflow Systems

For distributed dataflow systems we make two key observations:

- Distributed dataflow systems provide scalable data-parallel processing, so that a multitude of scale-outs can be reasonably used for jobs.

- Distributed dataflow systems run most batch jobs in multiple stages and allow to use different resources for stages.

Distributed dataflow systems process large datasets with data-parallel operators. For each data-parallel operator, a number of parallel task instances is scheduled, deployed, and executed on a set of connected shared-nothing commodity nodes. Typically, systems allow to set the degree of parallelism (DOP) for a job, a stage, or a single operator. Worker nodes offer execution slots to run a specific number of parallel task instances. Typically nodes offer as many of these execution slots to a distributed dataflow job as processing cores were reserved for the job. This data-parallel execution model of distributed dataflow systems is implemented by distributed runtime environments, which include efficient implementations of data-parallel operators and effective data partitioning mechanisms. Consequently, a multitude of different scale-outs can be reasonably used for a distributed dataflow job. Furthermore, depending on the input data, the job, and the resources different scale-outs yield different runtimes for a job.

Multiple subsequent operators that are executed together are called a stage of a distributed dataflow job. Most batch jobs consist of multiple such stages. The extent of a stage is limited by operators that require particular elements to be available at the same task instances and, thus, usually shuffling of the data. Some systems actually schedule, deploy, and run the stages of a job separately. Others schedule and deploy entire jobs at once, yet still only one stage is typically executed at any time. Distributed dataflow systems at least allow to set the DOP for each stage. Some also allow to allocate different sets of resources for each stage. However, even with systems that do not support this kind of dynamic scaling, every job can be split into multiple jobs to have its stages run separately and on different sets of resources. This allows to dynamically scale the resource usage of all distributed dataflow jobs that have multiple stages.

We do not assume a specific distributed dataflow system. There are many different distributed dataflow systems used for scalable general-purpose processing of large datasets. These implement for example different methods for fault tolerance, ranging from snapshotting intermediate results to annotating lineage for selective re-computation or even no fault tolerance at all. Such design decisions have an impact on the performance of jobs, reducing throughput for the failure-free case or requiring more recovery in case of failures. Distributed dataflow systems also provide different programming interfaces. Users might already be familiar with one interface or have existing code tied to a specific system. In addition, the systems come with different libraries of already implemented algorithms. Furthermore, in production settings, systems often have to be integrated with specific resource management systems, distributed file systems, messaging systems, and databases. All these factors are important reasons for users to choose one system over another, possibly one system for one job and a different one for the next, or even multiple systems for a pipeline of jobs. Therefore, multiple dataflow systems should be supported by a resource allocation system.

4.2.3 Dedicated Analytics Clusters

For dedicated analytics clusters we make two key assumptions:

- Dedicated analytics clusters are built using large numbers of homogeneous commodity nodes and standard software.
- Resources of dedicated analytics clusters are typically shared by multiple users and jobs.

When data processing is at the core of an organization's processes, organizations usually operate their own cluster infrastructures. These dedicated clusters typically consist of large numbers of shared-nothing commodity machines. The clusters are often entirely homogeneous or at least have subsets that are homogeneous. In fact, many distributed dataflow systems expect worker nodes to be homogeneous. Besides using commodity hardware for nodes and networks, standard software is used for the operating system, virtualization, cluster resource management, and distributed file systems. Compared to specialized parallel computers, this setup is less expensive, but also typically less reliable. Given the scale of infrastructures, failures are to be expected and distributed data processing systems consequently implement mechanisms to tolerate hardware and software failures. However, even when recovered, failures at the granularity of workers and tasks introduce inherent variance to job runtimes. Moreover, it is hard to provide throughput guarantees with commodity hardware and standard software. For these reasons, the performance provided by compute resources in these clusters is not completely predictable, but varies.

To achieve high resource utilization and in turn cost-efficiency, clusters are typically shared among multiple users running multiple jobs. Users reserve parts of the resources of a shared cluster via containers, which are an abstraction of resources, for example a number of virtual cores and amount of main memory. These containers run on specific nodes. On large clusters users typically have container reservations that span only subsets of the entire cluster. Therefore, data locality varies depending on the distribution of input datasets and the specific container reservations. Moreover, containers are often executed without isolation, even when containers of multiple jobs run on the same nodes. Given the fluctuating resource demands of long-running analytics jobs, running multiple jobs co-located in this way increases resource utilization and overall throughput due to statistical multiplexing [13, 54]. Yet, this also leads to interference between workloads. Thus, the performance of jobs can also vary depending on which other jobs are concurrently executed.

We do not assume the availability of a dedicated staging cluster to profile new jobs. Such a dedicated cluster adds costs. The dedicated staging cluster infrastructure also needs to be bought, operated, and maintained. Furthermore, profiling new jobs prior to their execution adds an overhead to the runtime of jobs. This overhead can be reduced significantly when profiling runs are executed using a sample of the actual input. However, accurately extrapolating from sample inputs and small scale-outs to processing the

actual input datasets at scale is not trivial. This is consequently done either with simplifying models [97] or for specific domains [89].

We do not assume control over job admission, execution order of jobs, or container placement. That is, we only focus on resource allocation, not resource management in its entirety. Switching to completely different schedulers or resource managers is a significant effort for organizations. There are also many considerations besides runtime targets to be taken into account when scheduling applications in cluster environments such as fairness, job priorities, and data locality. For these reasons, we focus on selecting the scale-out of a single distributed dataflow job and present a system architecture that allows our solution to be used for the resource negotiation of a single application.

4.2.4 Requirements for a Practical Solution

We argue that certain requirements for practical solutions that manage resource allocations on the basis of users' runtime targets follow from our observations on jobs, systems, and cluster setups. The four key requirements we identified are:

- Solutions should use black-box models to generally support distributed dataflow systems, so users can continue to choose the most appropriate tool for their tasks.
- Solutions should carefully select training data for performance models, independent of whether they use dedicated profiling runs or learn from previous executions of recurring jobs, so that predictions are accurate.
- Solutions should continuously monitor jobs and dynamically adjust resource allocations, so that the performance variance of distributed dataflows in shared cluster environments is addressed.
- Solutions should be designed as pluggable tools that focus on resource allocation, so users can benefit from solutions even with existing cluster setups, given the immense organizational effort of switching cluster software.

Based on these requirements, we developed our approach, methods, and system architecture.

4.3 Approach and Methods

In this section we first present our approach and an overview over our methods for dynamically allocating resources for distributed dataflow jobs. We then describe the application of our approach to iterative distributed dataflow jobs.

4.3.1 Solution Overview

Users often have specific runtime targets for their production batch jobs, yet at the same time it is difficult for them to allocate sets of resources that meet their demands without

reverting to significant over-provisioning. For this reason, we argue that users should not be required to make resource reservations themselves, but should be able to express their constraints to a system that automatically reserves minimally necessary sets of resources for their runtime targets. This system should use a runtime prediction model that supports multiple distributed dataflow systems, should train its model on carefully selected similar executions, and should address runtime variance continuously through dynamic scaling.

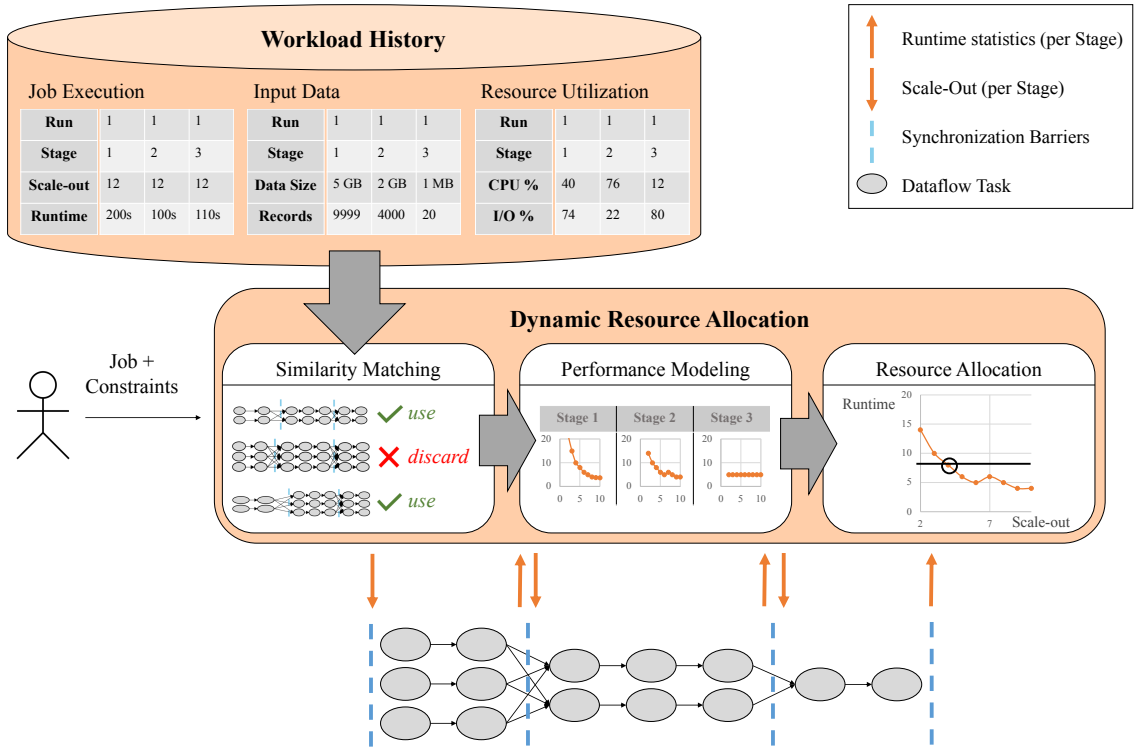


Figure 4.1: Dynamic resource allocation based on similar previous executions of recurring batch jobs to meet users' runtime targets.

Figure 4.1 shows an overview of our approach. A user submits a job with additional constraints for both the resources and the job runtime. A system for resource allocation receives this as input. The system selects resources for the submitted job and constraints in a three step process:

1. First, it matches the submitted job to similar previous executions. Similar previous executions are available for recurring batch jobs and allow to predict the runtime of a job.
2. Second, it uses matched previous executions to train a black-box runtime prediction model for the submitted job. Specifically, the system models the scale-out behavior of the job's stages to be able to predict the runtimes of distinct parts of the job.

3. Lastly, the system uses the scale-out models of stages to select minimal sets of resources predicted to provide the required runtime. This is done for each stage: After each stage the system decides the next stage's scale-out based on the currently elapsed and predicted remaining time.

The system uses a black-box model for predicting the performance of a job based on actually observed runtimes. This is possible for jobs that are repeatedly executed, such as daily or weekly batch jobs. Since many important production jobs are typically scheduled periodically, runtime statistics on previous executions are available for a large fraction of jobs, especially those jobs that users have clear performance expectations for. The system uses these statistics to carefully match similar previous executions as a basis for runtime prediction models.

The system continuously monitors the running job, models its performance based on matched previous executions, and adapts reserved resources for subsequent stages at the synchronization barriers in-between stages, if necessary. As more runtime statistics become available during the execution of a job, the system can match similar previous executions more accurately based on the actual runtime behavior. It then re-trains the scale-out models of the running job on the new selection of similar previous executions at runtime. This way, the prediction of the remaining runtime of a job can become more accurate as a job progresses.

In the following we briefly describe our main methods, before we present how we implemented our approach and methods in multiple components and as a pluggable job submission tool.

4.3.1.1 Workload History

Various information on the execution of jobs is used for matching similar previous executions and modeling the scale-out behavior of these. This information is recorded for all executions on a cluster and stored in a central repository. For modeling the scale-out behavior of the individual stages, we require the runtimes and scale-outs of the stages of a job. To effectively match similar previous executions, which form the basis for the scale-out modeling, we additionally require information on the input dataset, the convergence, and the utilization of resources. For distributed dataflow or resource management systems that do not log these information, additional monitoring of the distributed jobs is required.

4.3.1.2 Similarity Matching

Since our approach is to model a recurring job's performance based on its previous executions, our system selects similar previous runs of the submitted job from the workload history database, assuming that these allow to accurately predict the runtime of the submitted job. However, it is usually not the exact same job that is executed, even with

periodically running batch jobs. Usually at least the dataset and selections of resources, but possibly also program parameters and system configurations are slightly different. Such static factors are considered when selecting previous executions to predict the performance of a submitted job. Yet, it is hard to estimate beforehand what effect changes to these factors will have on a job's runtime. However, the effect of these changes is reflected in runtime statistics like stage runtimes and convergence. For this reason, we not only use static factors like statistics on the input dataset, but also match statistics on the current execution of a job like stage runtimes. Therefore, the matching of similar previous executions becomes more accurate as a job progresses and gets closer to its runtime target. In turn, when matched previous executions are continuously used to update prediction models, predictions become more accurate as well. Since different similarity measures are important for the runtime behavior of different jobs, we use job-specific thresholds and weights for combining multiple measures into an overall assessment of similarity. Moreover, we train these parameters on the history of each job, so users do not have to configure these heuristics manually.

4.3.1.3 Performance Modeling

Our approach for runtime prediction is to model the scale-out behavior of jobs. In particular, we model the scale-out behavior of individual job stages to be able to predict the runtimes of distinct parts of jobs for certain resource allocations. This way, we can for example predict the runtime of the remaining stages at the synchronization barriers in-between stages. We use black-box models, modeling the scale-out behavior only using the scale-outs and runtimes of previous executions of recurring jobs. To optimally support both different distributed dataflow systems and varying densities of available training data, we use two different regression models. First, we use a simple parameterized model of distributed processing, applicable to distributed dataflows. Second, we also use nonparametric regression, which can be used to accurately fit arbitrary scale-out behavior, given dense training data. Yet, training data density can vary significantly when previous executions of periodically scheduled batch jobs are used as samples. Moreover, nonparametric regression is not usable for extrapolation. Therefore, our system automatically chooses between the two regression models.

4.3.1.4 Resource Allocation

Given constraints for both the resources and the runtime, the system automatically selects a minimal necessary set of resources for a submitted job. For this, we use the scale-out models of the job stages, which link used resources to resulting runtimes. In particular, our approach is to search for the smallest scale-out within the user-provided bounds that is predicted to meet the user's runtime target, assuming more resources ensue more costs. Since this search space, namely the possible values for the scale-out, is discrete, we greedily search for such a scale-out, starting with the minimal scale-out and stopping

with the maximal scale-out specified by the user. If no scale-out within these bounds is predicted to meet the runtime target, we select the scale-out with the lowest predicted overall job runtime.

4.3.1.5 Runtime Adjustments

Our approach to monitoring a job's progress towards its runtime target is to predict the remaining runtime for the current resource allocation using the scale-out models of the individual job stages. Then we compare the predicted remaining runtime to the elapsed runtime and the runtime target. If the job is predicted to meet its runtime target, we do not change the scale-out. However, if the job is predicted to exceed the runtime target, we search for a scale-out that is predicted to meet the target. If instead the job's predicted overall runtime is below the runtime target, we search for a smaller scale-out to free surplus resources. This assessment happens at each synchronization barrier in-between subsequent job stages such as in-between subsequent iterations of iterative jobs. First, stages are distinct parts of jobs, which we can model separately and thus predict runtimes for. Second, distributed dataflow systems allow to use different DOPs for separate stages. Also, at synchronization barriers in-between stages the tasks of the previous stage have finished, while the tasks of the subsequent stage have not yet been started. Thus, there is no operator state to be migrated, only intermediate data, which often has to be shuffled for subsequent stages anyway. Stage barriers consequently allow to scale jobs with maintainable overheads [120]. Yet, as changing the scale-out of distributed dataflows at runtime is not entirely without overheads, even at synchronization barriers, we only adjust resource allocations if the predicted remaining runtime deviates significantly and add additional slack for establishing a new scale-out.

4.3.2 Application to Iterative Jobs

A particularly interesting class of distributed dataflow jobs for applying our approach are iterative workloads. This class includes, for example, many widely used machine learning and graph analysis algorithms. Iterative distributed dataflow jobs execute stages repeatedly, either for a fixed number of iterations or until a convergence criteria is met. That is, not only are many important algorithms iterative, but iterative distributed dataflow jobs often execute a high number of overall stages. Since we monitor and predict remaining runtimes on the basis of stages, iterative jobs therefore provide numerous points for assessing a job's progress and adapting allocations dynamically, if necessary. Moreover, since selection and aggregation reduce the data size considerably, subsequent stages in non-iterative distributed dataflow programs are often shorter, yet iterations of iterative distributed dataflow programs often have similar lengths. Thus, adapting the resources during the runtime of iterative jobs can have a large impact, even after a few stages have already finished execution. This is especially true for bulk iterations, which process the same dataset in each iteration.

Besides bulk iterations, there are also delta iterations. Delta iterations process datasets incrementally, considering a decreasing number of active elements from one iteration to the next. 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 [30] but also yields considerably different runtimes for iterations, even though the same dataflow stages are executed.

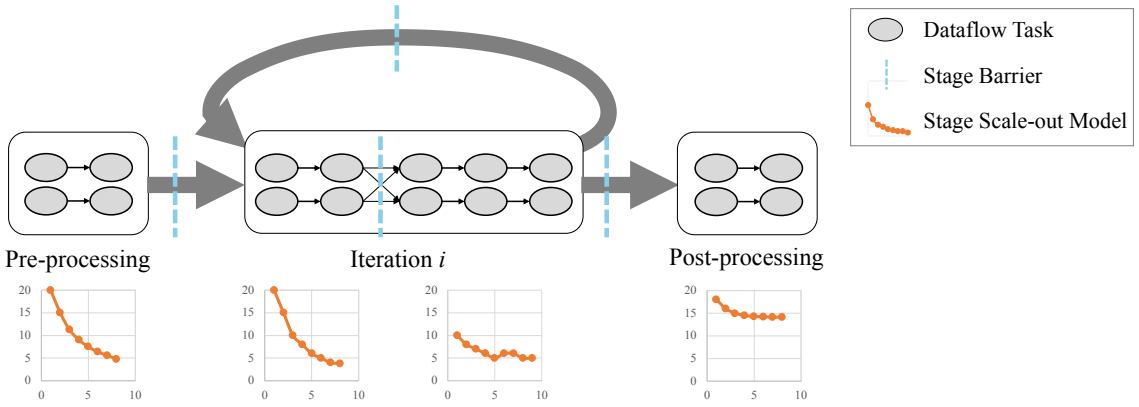


Figure 4.2: Stage-wise modeling of an iterative dataflow.

Figure 4.2 shows our approach applied to iterative jobs. The scale-out behavior of the stages of each iteration is modeled, enabling our solution to predict the runtime of the remaining iterations and adapt resource allocations in-between iterations, if necessary. We apply our approach directly to repeatedly executed stages since the runtimes of iterations can vary significantly, especially in case of incremental processing with delta iterations. Therefore, we do not create a single model for repeatedly executed stages, but model all stages separately.

Even modeling the stages of subsequent iterations separately is still an approximation. 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. This is addressed by matching similar previous executions of the iterative job also based on its actual runtime behavior.

4.4 System Architecture

In this section we first present an architecture overview for implementing our approach to dynamic resource allocation as a job submission tool. Then, we describe the features and the implementation of our main prototype components. Finally, we show how we integrated our prototype components with YARN and Spark.

4.4.1 Architecture Overview

Dynamic resource allocation for resource-managed shared analytics clusters can be implemented as a job submission tool. Figure 4.3 shows a system architecture for this implementation strategy. The system for resource allocation is used for submitting jobs to the resource management system. It selects and adjusts the resource allocation of a single job using available interfaces for job submission, progress monitoring, and dynamic scaling. This design has the advantage that the solution can be used by a single user and for a single job, without requiring invasive changes to the cluster setup. Therefore, our system works with existing cluster setups, specifically resource managers and schedulers.

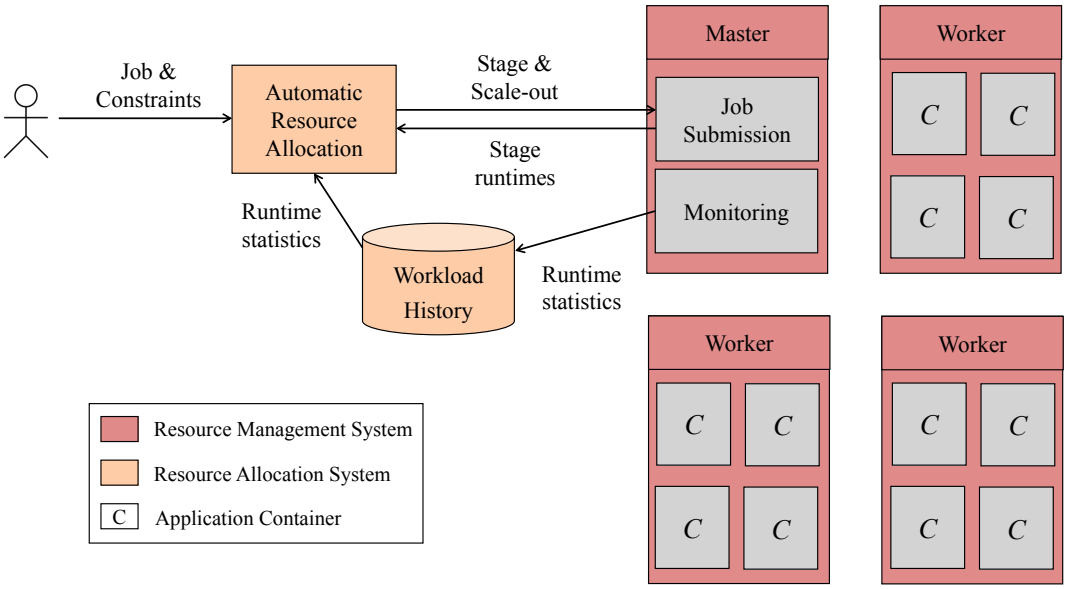


Figure 4.3: Automatic resource allocation based on historic workload data implemented as a job submission tool for resource-managed clusters.

Our allocation system wraps the cluster manager's existing submission tool and translates abstract user-level performance constraints into specific resource allocations. A user can then submit a job with constraints for the runtime and resources to our allocation system. The user specifies a runtime target as well as a minimal and a maximal number of containers. Based on these constraints, our system selects the scale-outs that it communicates to the cluster manager.

As we model and predict the performance of recurring distributed dataflow jobs based on similar previous executions, our allocation system requires access to a repository of workload statistics. This monitoring data on previously executed job runs can either be provided by the cluster manager or by an additional monitoring system. In either case, our allocation system is connected to a database containing workload statistics.

Our allocation system selects resources for each of a job's stages at runtime to mitigate the impact of runtime variance. It either submits a job stage-by-stage with specific scale-outs or dynamically manages resource allocations for a job submitted in its entirety. While it is possible to split each job along its stages and to have each stage be scheduled, deployed, and executed as a separate job using different levels of parallelism and resources, system support for dynamic scaling has advantages. When dynamic scaling is supported, distributed workers remain running and also keep the intermediate data in memory as far as possible. In either case, our system adapts the current resource allocation at the synchronization barriers in-between stages and, therefore, requires notifications about the end of stages from the cluster manager.

4.4.2 Prototype Components

We implemented our approach and methods in a set of prototype components. These components are shown in Figure 4.4. On the one side, there is *Ellis*, the prototype system implementing our approach to automatic resource allocation. On the other side, there is *Freamon*, the monitoring system we developed to record resource utilization statistics for the containers of distributed applications.

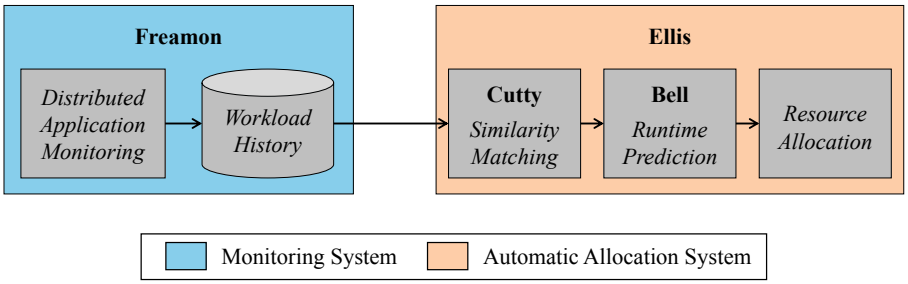


Figure 4.4: Components for automatic resource allocation.

Given a runtime target for a recurring job, Ellis initially selects and dynamically adjusts resource allocations. For this, Ellis contains logic for selecting resources based on predicted runtimes. For these runtime predictions Ellis uses the components *Bell* and *Cutty*. Bell is a system for modeling the scale-out behavior of distributed dataflow jobs, using a black-box approach which supports multiple distributed dataflow systems instead of relying on a system-specific model. Cutty is a system for selecting those similar previous executions of a recurring job that allow accurate predictions of the runtime of the job. Ellis uses Cutty to select similar previous executions as training data for Bell.

Since we built Ellis for applications running on YARN, which does not record all the runtime statistics we are using to select training data and model performance on, we use our own monitoring system Freamon. Freamon records not only the runtimes of jobs and individual job stages, but also records the resource utilization at the granularity of

application containers. Freamon stores these statistics in a database and makes them available to Ellis.

In the following we briefly describe the features and the implementation of Bell, Cutty, and Ellis.

4.4.2.1 Component Descriptions

Bell is a system for modeling the scale-out behavior of distributed dataflow jobs based on previous executions of recurring jobs. Bell learns how the runtime of a job depends on the scale-out from available samples. It uses a black-box model and, thus, is a general solution that works with multiple analytics frameworks, acknowledging the fact that users need to be able to choose the best tool for their tasks. In particular, Bell fits a function of the scale-out that can be used for predicting the job runtime using regression. As the number of previous runs that are available for training a job model varies even for periodically scheduled jobs, Bell aims to make the most of the historic data for each job. For this reason, Bell uses two different models and automatically chooses between them: highly flexible nonparametric regression and also parametric regression with a simple model of distributed processing, applicable to distributed dataflows. That is, Bell uses one model for interpolating arbitrary scale-out behavior, given dense training data, and another model as a robust fallback, to be able to provide reasonable predictions from a few data points and to extrapolate. Bell automatically chooses between these two models depending on the prediction task and the available historic data using cross-validation.

Cutty is a system for selecting similar previous executions of distributed dataflow programs based on their similarity to a currently running job, matching those previous executions that provide accurate performance estimation. Executions of the same program can exhibit considerably different runtime behavior in terms of runtimes, resource utilization, and convergence depending on input datasets, program parameters, system configurations, and specific resources used. Selecting executions with a similar runtime behavior is, however, a prerequisite for accurate prediction models. Using techniques we proposed with a system for estimating the runtimes of recurring iterative dataflow jobs, Cutty matches previous executions based on multiple similarity measures. Statistics for some of these measures are available offline, yet some measures are based on runtime statistics such as the runtimes of stages and convergence rates. These measures are only available at runtime. Cutty incorporates runtime statistics as they become available, allowing the selection of similar previous executions to become more accurate as jobs progress and get closer to their runtime targets. Since it depends on the job which of the different similarity measures are most useful for selecting previous executions as a basis for performance estimation, Cutty automatically trains thresholds and weights for the similarity measures on all of a job's previous executions.

Ellis is a system for initially selecting and dynamically adjusting resource allocations based on scale-out models. Given a user's runtime target, Ellis uses Bell and Cutty to select as few resources as necessary to meet the target. Specifically, Ellis uses Cutty to select

similar previous executions of a job. Ellis then provides data on these previous executions to Bell and uses Bell to create separate scale-out models for each of the job's stages. Subsequently, Ellis is able to predict the runtimes of separate stages and thus also the runtime of the remaining stages for a running job. Ellis predicts the remaining runtime at all synchronization barriers in-between stages. 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 runtime target. Ellis then adjusts resource allocations at stage barriers, effectively scaling the distributed dataflow jobs dynamically and thereby addressing the inherent variance in job performance. As Ellis supplies runtime statistics for the currently running job to Cutty and has Bell update models at runtime, predictions become increasingly accurate as jobs progress and more runtime statistics can be matched against previous executions.

The methods implemented with each of these three components are described in detail in the following three chapters.

4.4.2.2 Component Implementation

For the implementation of our prototype components, we used the Scala programming language. Key libraries we employed for our automatic resource allocation system are *BOBYQAOptimizer*, which is part of the Apache Commons library[1] and which we used for Cutty's optimization of the similarity matching, and *Breeze*[2], which is a numerical processing library for Scala that we used to implement the regression models of Bell. Other than that, Ellis uses a database connector for access to a database with statistical data on previous executions.

Our monitoring system Freamon consists of distributed monitoring agents, which we implemented using *Akka*[3], and a central database for recorded statistics, for which we use the *MonetDB* column-store database[4] [121]. The distributed monitoring agents use different Linux tools for recording resource utilization statistics of the processes of application containers:

- The virtual file system `/proc` for recording CPU and memory utilization.
- *Nethogs*[5] for recording the network utilization.
- *PidStat*[6] for recording the disk and CPU utilization, broken down into CPU utilization attributed to users and the system.

[1] `https://commons.apache.org/`, accessed 2017-11-15.
[2] `https://github.com/scalanlp/breeze`, accessed 2017-11-15.
[3] `https://akka.io/`, accessed 2017-11-15.
[4] `https://www.monetdb.org/`, accessed 2017-11-15.
[5] `https://github.com/raboof/nethogs`, accessed 2017-11-15.
[6] `https://github.com/sysstat/sysstat`, accessed 2017-11-15.

- *Dstat*[7], which itself is a frontend for multiple Linux resource utilization monitoring tools and which we use to also capture the overall resource utilization of worker nodes.

By default, Freamon uses only `/proc` for recording CPU and memory utilization. All other monitoring is optional.

4.4.3 Integration with YARN and Spark

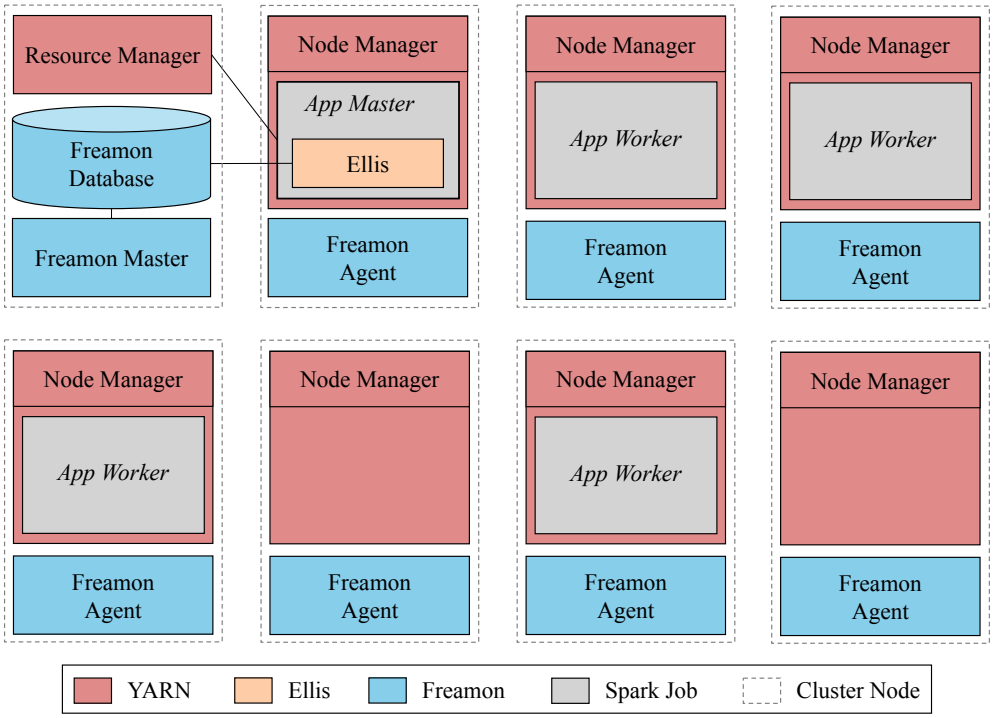


Figure 4.5: Deployment with a Spark application on a YARN cluster.

We integrated our prototype components with YARN and Spark to demonstrate the practicability of our approach and evaluate our methods. Following the general approach of implementing dynamic resource allocation as a pluggable job submission tool, we integrated Ellis with YARN and Spark on a per-job basis. Specifically, we implemented Ellis as a system to be used from an Application Master, which is the first process started for each YARN application. The Application Master process is responsible for resource negotiation and contains an application's driver program, which executes the stages of a distributed dataflow job. Since YARN does not provide application monitoring on the necessary granularity, we also developed Freamon. Freamon is a distributed monitoring

[7] `http://dag.wiee.rs/home-made/dstat/`, accessed 2017-11-15.

system that periodically records resource utilization statistics for all the containers of a YARN application. Ellis is connected to Freamon's central database component.

Figure 4.5 shows how Ellis and Freamon are deployed with a YARN cluster and used by a Spark application. There are six cluster nodes. One acts as master node, while five act as worker nodes. The master node runs YARN's master, which is called Resource Manager. It also runs the master of our monitoring system, which coordinates its worker agents and manages the central database that contains recorded workload statistics. Each of the worker nodes runs YARN's worker manager and Freamon's worker agent. In the example shown in Figure 4.5, five YARN worker execute a single container each, belonging to a single Spark YARN job. One worker runs the Application Master of this job, which is started first, negotiates the resources for the Spark job, and contains the job's driver program. The Application Master integrates Ellis for selecting resource allocations.

Ellis is used on job submission to initially select the number of containers the job uses. Moreover, Ellis is used after each completed stage. At this point, Ellis assesses whether the job is still predicted to finish within the bounds of a runtime target with the current resource allocation. If this is not the case, Ellis searches for a scale-out that is predicted to yield an overall job runtime within these bounds. If a job is executing faster than predicted, Ellis also searches for a smaller scale-out predicted to meet the target, so surplus resources become available for other YARN jobs. Ellis accesses Freamon's database for information on previous executions, which it uses to model and predict the runtime of job stages.

4.4.3.1 Integration through a Spark Job Listener

Ellis is integrated into Spark as a `JobListener`. The Spark system notifies this listener when a stage ends. The listener can be added to the driver program of a Spark job to have Ellis manage resource allocations dynamically. Listing 4.1 shows this for an exemplary Spark job.

Listing 4.1: The beginning of an exemplary Spark job that integrates Ellis by using our job listener.

```
1   val appConf = new PageRankArgs(args)
2   val appSignature = "PageRank"
3
4   val sparkConf = new SparkConf().setAppName(appSignature)
5   val sparkContext = new SparkContext(sparkConf)
6   val listener = new StageScaleOutPredictor(
7       sparkContext,
8       appSignature,
9       appConf.dbPath(),
10      appConf.minContainers(),
11      appConf.maxContainers(),
12      appConf.maxRuntime())
13  sparkContext.addSparkListener(listener)
```

In Line 6, a `StageScaleOutPredictor` is created at the beginning of the job. This is the Spark listener subclass that integrates Ellis. The `StageScaleOutPredictor` is initialized with the current execution context, application metadata, the path to Freamon's database, and the user's performance constraints. These constraints are a minimal and a maximal number of Spark workers, which are called executors, as well as a target runtime. Finally, in Line 13, the listener instance is added to the execution context, which activates the listener.

When the listener is created, it first computes an initial scale-out for the job as part of its initialization routine. The listener is also called whenever a stage ends as it overrides the `onJobEnd` hook. In this procedure, the listener assesses the job progress and adjusts the current resource allocation as described above.

This listener can be used for a single job, requiring modest changes to the driver program of the job. In fact, not counting the line-by-line enumeration of the arguments supplied when creating our listener, only two lines need to be changed to use Ellis for a Spark job.

4.4.3.2 Dynamic Scaling of a Spark YARN Job

YARN allows running applications to request more containers and to return some of the reserved containers. Specifically, while a job is running, the job's Application Master can re-negotiate resources with YARN's Resource Manager and thereby adapt the current allocation. Spark also supports dynamic scaling at runtime. There is an interface to change the number of executors that run a job. When Spark is executed on YARN, both features are integrated. A driver program can request more executors for its job at runtime, for which Spark then requests more containers from YARN, before it first starts new executors in these containers and then has these executors run additional parallel task instances. A driver program can also discard executors, which Spark then stops, before returning the container resources to YARN again.

Spark's dynamic scaling is by default managed by an auto-scaling policy, which computes the number of executors for a job based on currently pending tasks and idle executors. We deactivate this policy to manage the number of executors based on predicted runtimes.

When a Spark executor shuffles data as it does before new stages start, the executor sorts the intermediate results by key and serves these to all the executors of the next stage. Usually when an executor is discarded before its shuffle outputs have been fetched by all the executors of the next stage, Spark would recompute the outputs of the discarded executors. To make sure executors are decommissioned gracefully, an external shuffle service can be used. An external shuffle service is a standalone application, running on all worker nodes outside Spark's executor processes. It manages the shuffle output independently of the availability of executor processes. If an external shuffle service is used, executors will fetch shuffle outputs from this service instead of from each other,

allowing executors to be decommissioned as soon as they finished processing and before the subsequent stage has finished reading in the entire intermediate data.

5 Modeling the Scale-Out Behavior of Batch Jobs

Contents

5.1 Scaling out Distributed Dataflows 59
5.2 Scale-Out Models for Distributed Dataflows 64
5.2.1 Parametric Regression 64
5.2.2 Nonparametric Regression 66
5.2.3 Automatic Model Selection 67
5.3 Evaluation 67
5.3.1 Cluster Setup 67
5.3.2 Experiments 68
5.3.3 Results 69

This chapter presents Bell, which we published in [122] (© 2016 IEEE). Bell is a system for predicting the runtime of distributed dataflow jobs. We use a black-box approach for modeling the scale-out behavior of jobs based on previous executions. Therefore, Bell supports different distributed dataflow systems. In particular, Bell uses two different univariate regression models to find a function that captures the scale-out behavior of a job. The function takes a scale-out as input and outputs the job's runtime. Bell uses one model for interpolating arbitrary scale-out behavior, given dense training data. It uses another model as a robust fallback and for extrapolation, able to provide reasonable predictions from a few data points. Depending on the available training samples and the prediction task, Bell automatically chooses between these two models.

This chapter first describes how distributed dataflow jobs can be scaled out to more cluster resources. The chapter then presents the two regression models Bell uses for modeling the scale-out behavior of distributed dataflows and how Bell automatically selects between the two models using cross-validation. Finally, the chapter presents an evaluation of Bell using six different exemplary Flink and Spark jobs.

5.1 Scaling out Distributed Dataflows

Distributed dataflows are data-parallel operator pipelines, where operators execute UDFs on a single stream of inputs or even merge multiple dataflows using operators like Joins.

Each operator is executed data-parallelly, so that work is distributed among data-parallel task instances of an operator. Each of the data-parallel task instances processes a partition of the data. The task instances are executed by worker processes on connected nodes. Worker processes on multi-core nodes run multiple data-parallel task instances in parallel, using multiple threads. The data parallelism is increased when jobs are executed on more resources, usually in direct relation to the number of available processing units. Consequently, task instances potentially process smaller partitions, when a higher scale-out is used while the input stays the same. Therefore, scaling out distributed dataflows to more resources can speed up the execution of distributed dataflow jobs as more execution units work on smaller partitions.

Resources are usually reserved through containers. Containers in this context are an abstraction of compute capabilities for scheduling and reservation. To make a container reservation, users specify the amount and the size of containers. The size of a container could, for example, specify the amount of memory and the amount of cores users want to use with each container. While main memory is assigned to a particular container, usage of the other resources such as CPU cores, disks, and the network is typically not isolated. In fact, containers are often just processes running data-parallel task instances of a job. It is only the distributed dataflow system's scheduling of task instances that gives meaning to the specification of for example a particular number of reserved CPU cores. The systems denote each available core as an execution slot and assign data-parallel task instances to each of these slots.

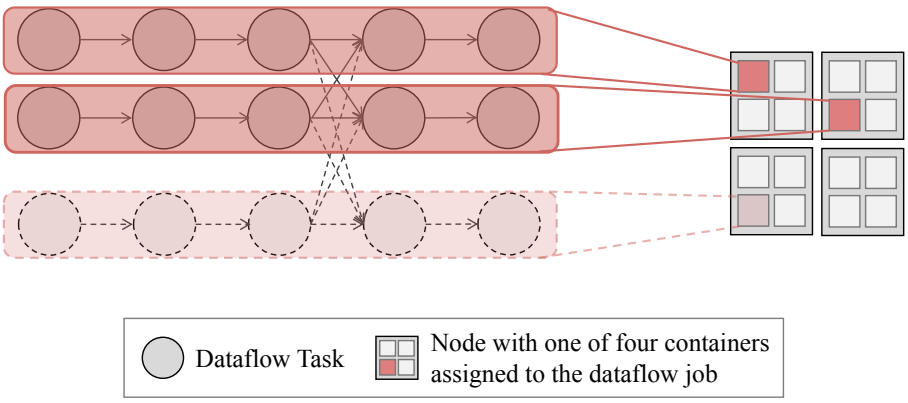


Figure 5.1: Scaling out a distributed dataflow job.

Figure 5.1 shows a distributed dataflow job that is deployed on a cluster of four nodes that each run four containers. In this example the dataflow job uses the same level of parallelism for each of its task, so each task has the same number of data-parallel task instances. Moreover, each container offers capabilities for a single execution slot, to which the distributed dataflow system schedules pipelines of subsequent task instances. In the example, a scale-out of two is used for the job at first. Therefore, the job first runs two pipelines of subsequent task instances in two containers, shown in more saturated colors and with solid lines in Figure 5.1. Adding another container and also a parallel task

instance for each task is shown in less saturated colors and with dashed lines. How much a particular job is sped up by using more containers depends on the program, the input datasets, the distributed dataflow system, and the computing infrastructure.

Since we are interested in predicting the runtime of a given job, the problem size is fixed by the given input datasets and an algorithm implementation in a particular distributed dataflow framework. That is, we are interested in the strong scaling behavior, the relationship between the scale-out and a job's runtime, which we call *scale-out behavior*. The scale-out refers to the specific number of containers used for executing the job. We assume that containers run on homogeneous hardware, so each node and the network connections between nodes principally provide the same performance. A job is defined by running the same algorithm implementation using the same distributed dataflow system. To be able to estimate the runtime of jobs using a particular scale-out we need to capture the *relative speed up* [123] that is achieved by scaling out the same job onto more containers. In particular, we want to find a function f for a given job that takes the scale-out x as input and returns the runtime t.

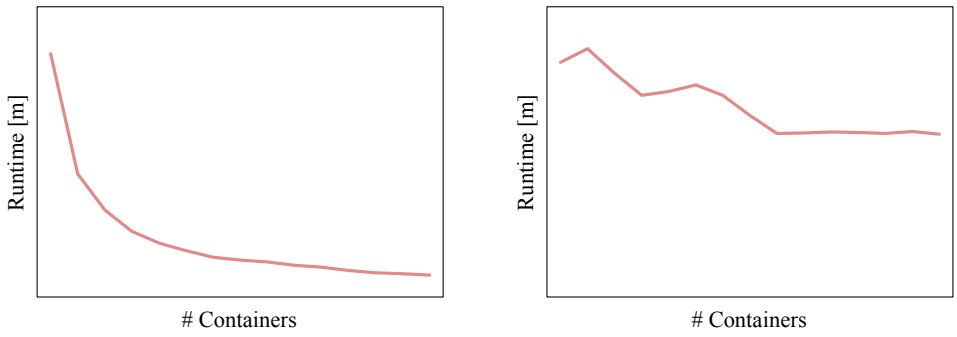


Figure 5.2: Two examples for the scale-out behavior of distributed dataflow jobs.

Figure 5.2 shows two examples for the scale-out behavior of distributed dataflow jobs. One job exhibits a scale-out behavior that is close to linear relative speed-up, which is an easily predictable scale-out behavior. The other job's scale-out behavior is more complicated, with some higher scale-outs even yielding significantly higher runtimes than lower scale-outs, which can for example result from a job's partitioning.

Whether increasing the scale-out for a particular distributed dataflow job speeds up its execution depends on many factors. The scale-out behavior of a job is determined by the program, the input data, the distributed dataflow system, and the resources used:

Programs A prerequisite for speeding up a job through scaling it out to more resources is that the job is mainly data-parallel [5, 124, 125]. Necessary serial computation on a single element or a single group of elements does not benefit from increasing the parallelism. The job has to execute the same operations on many input elements, so that the elements can be effectively split up among parallel workers and processed independently. This is the case for stateless operators like Maps and Filters, which compute on single elements. Some operators such as aggregations or Joins

need groups of elements to be available, requiring synchronization and communication among parallel workers. If data is not already partitioned into the necessary groups, the data needs to be shuffled between all workers or even send to a single point of aggregation. This requires either all-to-all or all-to-one communication, before each group can be processed in parallel. Whether a program scales well with more resources depends on the UDFs of operators and the necessary communication between parallel task instances. For example, if a job only consists of a single data-parallel Map operator that searches for a particular string in a large text dataset, the job will scale well with more compute resources, only limited by the rate at which data can be ingested by the distributed dataflow system. If instead a job iteratively executes stages that mainly require communication between workers, using more worker nodes and therefore more cores and memory will not necessarily speed up the job considerably. This is for example often the case for iterative graph algorithms that use label propagation, computing for instance the connected components of a graph. In general, programs that predominantly require CPU resources are known to scale well [24, 126].

Datasets The work needs to be distributed equally among data-parallel task instances. That is, each data-parallel task instance should receive and process roughly the same number of elements, assuming that the computation on each element has comparable computational costs. Partitions of similar size can always be established for operators that do not require particular partitions. Whether similarly sized partitions can be established for operators that do require particular partitions, be it for reducing a group of elements or for joining elements of multiple dataflows with the same keys, depends on the input data. In particular, it depends on the value distributions of the keys used for grouping or joining elements. First, the level of parallelism is limited by the number of different key values. Second, how the elements distribute across groups and how these groups are assigned to data-parallel task instances can lead to significantly different workloads among parallel workers. Yet, all workers are synchronized with each shuffling for such partition-dependent operators. These operators need to wait for all elements with a particular key. Therefore, they need to wait for all predecessor task instances to finish and for all the input data to be transmitted. In these steps, the slowest task instance determines the overall runtime. Therefore, the overall throughput is reduced when individual workers receive considerable more work than others. The values in particular fields of elements are also crucial for operators that filter or select only some elements. Such operators include Filters and Joins. Without detailed knowledge about both value distributions and the selectivity of the operators, estimating the workload for subsequent operators is difficult.

Systems The scale-out behavior also depends on the distributed dataflow system. Different distributed dataflow systems apply slightly different execution models. For example, how parallel task instances exchange data and synchronize is determined by a system's execution model. Many distributed dataflow systems apply a BSP

model for their parallel computations. Yet, systems can also relax this model to some extent for specific workloads such as iterative converging algorithms, which makes these jobs more scalable [127]. System may also provide different operator implementations and different partitioning methods. Furthermore, some systems automatically optimize execution plans and even incorporate the scale-out into the plan optimization. That is, with a different scale-out a job could use a different order of operators or a different operator implementation. This can have a significant impact on the job's performance. Moreover, distributed dataflow systems implement a range of different approaches to fault tolerance. Some systems exchange intermediate results between workers through distributed file systems that store the data redundantly on disk for fault tolerance. Other systems instead optimize the failure-free case, only annotating intermediate data with linage information to be able to re-compute particular partitions. These design decisions determine which resources are used for a job and, therefore, how well a job scales with more compute resources.

Resources When distributed dataflow jobs are scaled out, more compute resources become available to the jobs. That is, more memory and cores are available for processing the same overall number of elements. Therefore, given data is partitioned effectively, each execution unit processes a smaller partition. At the same time, ingestion rates are often limited. The input data for a job is usually stored on the disks of particular nodes and reading in this data can be the bottleneck of a job. If the nodes already read-in elements at maximal capacity and slower than elements can be processed by workers, using even more compute resources will not speed up the computation. Instead using more compute resources will only reduce the utilization of CPU cores. Similarly, the network can be the bottleneck of a job. For example, when hierarchical networks aggregate traffic on higher-level links and these are already fully saturated, adding more compute resources that also exchange data over these links, will also not speed up a job's execution. Thus, depending on the cluster network, the costs of communication and synchronization between workers can vary considerably. More generally, the relative speed of computation compared to the speed of ingress and egress of a particular computing infrastructure impacts the scale-out behavior of jobs. These ratio between compute and I/O speeds often also depends on system configurations. For example, users can adjust how much memory should be used for buffers and how much for task state or pick among multiple compression methods.

In summary, the actual performance and scaling behavior depends on many factors: programs, datasets, systems and system configurations, and hardware resources. Consequently, estimating the runtime of a specific scale-out of a job statically is difficult. Such upfront estimation would require a complex model as well as detailed statistics on both the program and the input datasets. An estimation system would then need to accurately model the selectivity of tasks, partitioning among parallel task instances, average task runtimes, and critical paths in job graphs. However, even such a detailed model

would only be accurate for a specific distributed dataflow system, while detailed statistics typically require instrumentation and dedicated profiling runs. For these reasons, we propose to learn the scale-out behavior from actual executions of a job using black-box models. These black-box models of course still need to accurately capture the scale-out behavior of distributed dataflow jobs.

5.2 Scale-Out Models for Distributed Dataflows

Bell learns how the runtime of a job depends on the scale-out from given samples. The scale-out models need to support multiple distributed dataflow systems, since various distributed dataflow systems are used today, each with its own feature set and available libraries. Therefore, Bell uses a black-box model, fitting a function that predicts the runtime of a job only based on the scale-out. Moreover, Bell either uses a simple parameterized model of distributed processing, applicable to distributed dataflow systems, or nonparametric regression, which is able to fit arbitrary scale-out behaviors.

We use regression to find a function that models the scale-out behavior based on provided examples. Formally, regression is the task to estimate the relationship between an output variable $\hat{y}$, given an input x such that $\hat{y}$ is close to the true value y for a defined error measure. A regression model is represented by a function $f : X \to Y$ and learned by a regression algorithm. Bell uses both parametric and nonparametric regression. Parametric regression fits a parameterized model to the training data by choosing weights that are optimal for a defined error measure. Nonparametric regression instead fits the data assuming locally defined behavior, without taking a parameterized model as input.

Bell trains these regression models for recurring jobs using the available historic data. This data, specifically the scale-outs and runtimes of previous executions, is consequently required as input to Bell, before it can be used for runtime prediction. Using cross-validation, Bell automatically selects between its models based on the available training data. Since nonparametric regression is not usable for extrapolation, this model selection step is only necessary for interpolation.

5.2.1 Parametric Regression

Parametric regression requires a parameterized model in addition to training data and then learns the model's parameters such that the model optimally fits the training data according to a defined error measure. As such a parameterized model, Bell uses Equation 5.1, which is based on the model proposed with Ernest [97], another system for predicting the runtimes of distributed dataflow systems. In comparison to the model proposed with Ernest, we use no factor for the scale of the datasets, since Bell is used to train the model on similar previous executions instead of relying on isolated profiling runs on samples of the actual input data.

$$f = \theta_0 + \theta_1 \cdot \frac{1}{containers} + \theta_2 \cdot \log(containers) + \theta_3 \cdot containers \tag{5.1}$$

Equation 5.1 is a model of distributed computation and communication applicable to distributed dataflows. It consists of four additive terms, each representing an aspect of parallel computation and communication of a certain number of *containers* that collaboratively process data. Each term has one parameter θ_i to be learned for some training data. As shown in the equation, the four terms represent:

- serial computation, independent of any parallelism through multiple containers
- parallel computation, split between the containers working in parallel
- communication patterns for step-wise aggregation such as tree-based aggregation
- overheads that increase with the number of containers such as for scheduling and all-to-one communication.

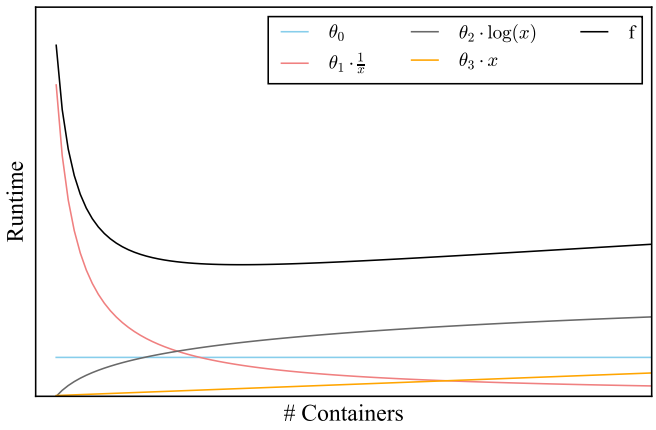


Figure 5.3: Exemplary model terms of Bell's parameterized model of distributed processing.

Figure 5.3 shows example graphs for each of the four model terms and the resulting overall model. For this demonstration, we selected the following values for the parameter θ_i: $\theta_0 = 2.5$ for constant serial computation time (blue), $\theta_1 = 20$ for reciprocal parallel computation time (red), $\theta_2 = 1.5$ for logarithmic time for step-wise aggregation in (gray), and $\theta_3 = 0.05$ for linearly increasing time for scaling overheads in (orange). The overall model of the runtime f is shown in black.

Since all terms of the model represent costs in terms of runtime, each one only adds to the overall runtime of a job. Therefore, the parameters θ_i should all be non-negative. For this reason, Bell uses non-negative least square (NNLS) to estimate the parameters when fitting a curve to given training data points.

Parametric regression, especially with a simple univariate model, can be used for both interpolation and extrapolation. Moreover, compared to nonparametric models it allows to make reasonable predictions from a few data points.

However, the parameterized model is a simple model of distributed processing. It is, consequently, applicable to all distributed dataflow systems and robust. Yet, its modeling accuracy when applied to real jobs running on real systems, even given high density training data, can be limited. For example, it is possible that specific scale-outs result in an overall parallelism, for which the data partitioning is considerably less effective than nearby scale-outs, which this simple model does not capture well. To be able to model such scale-out behavior more accurately, given dense training data, Bell also applies nonparametric regression.

5.2.2 Nonparametric Regression

In addition to the parameterized model of distributed processing, Bell also uses nonparametric regression, since the parametric model might not capture all possible scale-out behaviors well. Nonparametric regression does not require a specific model. Instead it infers the model automatically by assuming locally defined behavior in the training data. In line with the parametric model, this also supports different distributed dataflow systems.

Bell uses local linear regression (LLR) with a Gaussian kernel [128] to estimate the regression function. To give an intuition, Figure 5.4 shows a dataset, for which the regression function is evaluated at a specific point, shown in gray, by fitting a linear function to the surrounding samples. The samples used for this linear regression are weighted by a Gaussian kernel of a specific width. As a result, the smaller the distance of the samples is to the evaluated point, the more weight is given to them. Bell selects the kernel width automatically using cross-validation.

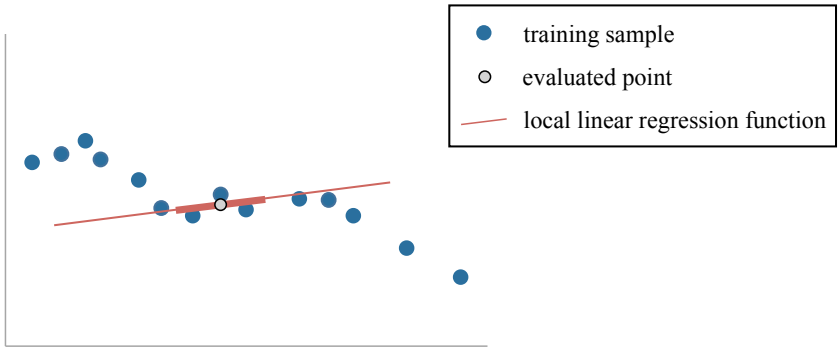


Figure 5.4: Local linear regression.

Nonparametric regression requires dense training data to fit functions that accurately model the scale-out behavior of a job. This is due to the locally optimized fitted curves.

Also, nonparametric regression can only be used for interpolating, not extrapolating. However, when interpolating the scale-out behavior of a distributed dataflow job based on dense training data, nonparametric regression allows to model arbitrary scale-out behavior accurately.

5.2.3 Automatic Model Selection

With parametric and nonparametric regression Bell uses two different methods for modeling the scale-out behavior of distributed dataflow jobs. Given dense training data, nonparametric regression allows to accurately model arbitrary scale-out behavior. Yet, when only few training data points are available, parametric regression might provide better results. Moreover, as we use Bell to model the scale-out behavior of recurring jobs based on previous executions of the jobs, we need to expect training data of varying density. Therefore, Bell first performs a model selection step to decide which prediction model to use for the given training data, aiming to make the most of the available historic data for a specific job.

Bell selects between the two regression models using cross-validation. We repeatedly learn and test both models, using the majority of available data points for training and the remaining points for testing the prediction accuracy. Bell then selects the model with the smallest cross-validation error for interpolation. Specifically, Bell performs k-fold cross-validation, given k unique scale-outs in the available data. However, the two folds for the smallest and largest scale-out are not used as test folds, since the goal is to assess the interpolation performance. They are only used for training. All other folds are used either for training or testing in this cross-validation. By partitioning our dataset this way, we test the prediction performance for scale-outs that have not already been used for training.

Nonparametric regression is only usable for interpolation and not for extrapolation, due to the local optimization. Therefore, Bell uses only parametric regression for estimating the runtime for lower or higher number of containers than have been previously used.

5.3 Evaluation

We evaluated our approach to scale-out modeling, which we implemented with Bell, with six different exemplary distributed dataflow jobs using both Flink and Spark, showing both the resulting scale-out models and reporting mean relative errors.

5.3.1 Cluster Setup

All experiments were done on 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. For the experiments we used Hadoop 2.7.1, Flink 1.0.3, and Spark 2.0.0. We used Spark's GraphX and MLlib libraries in the versions 1.6.0 and 1.1.0 respectively.

5.3.2 Experiments

We evaluated Bell with six different jobs and five datasets. We executed each of these jobs repeatedly, using different scale-outs, to have a history of runs for each job. We then fitted the two models that Bell supports, the simple parameterized model of distributed processing and nonparametric regression, showing how both individual models support different jobs. We also utilized this history of runs for a random sub-sampling cross-validation. We used different numbers of randomly selected training samples and a randomly selected test sample to show how the individual models and Bell perform with different amounts of available training data in terms of relative prediction error. Bell automatically selects between the two individual models in an initial model selection step.

The jobs we used cover different application domains, including search, relational queries, graph processing, and supervised as well as unsupervised machine learning. The datasets cover different types of data, various data sizes, and are generated using different data generators.

5.3.2.1 Jobs

Table 5.1 shows the six benchmark jobs we used, three Spark and three Flink jobs. Except SGD and PageRank, all jobs come with the examples of the frameworks. The SGD job we used is from Spark's MLlib, a library for scalable machine learning. The PageRank job is from GraphX [49], a graph processing system implemented on top of Spark.

Table 5.1: 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

5.3.2.2 Datasets

For the five datasets used in these experiments, we used five data generators. For graph and text data, we used generators from the Big Data Generator Suite (BDGS) [129]. These data generators scale real datasets while preserving key characteristics of the data. For relational data we used the data generator of the TPC-H benchmark suite[1]. In addition, we implemented a data generator for three-dimensional points and one for multi-dimensional feature vectors. Using these data generators, we produced the following five datasets:

Wiki The Wiki dataset was generated using the text generator of BDGS, which applies latent dirichlet allocation (LDA) [130] 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 [131] and a real graph of linked Web pages as a basis. 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 of the generator 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, resulting in 10 GB of data.

Points The Points dataset was generated using our own generator to produce 4,216,562,650 three-dimensional points following a Gaussian mixture model of five normal distributions with random cluster centers and equal variances, resulting in 50 GB of data.

5.3.3 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.

To begin with, every dataset was fitted using the parametric and nonparametric regression models. Figure 5.5 summarizes the results for each of the jobs. NNLS and LLR are the parametric and nonparametric models, respectively. The parametric model provides a good fit for Grep, K-Means, and Word Count. Yet, it falls short when jobs exhibit more

[1] `http://www.tpc.org/tpch/`, accessed 2018-02-19.

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.

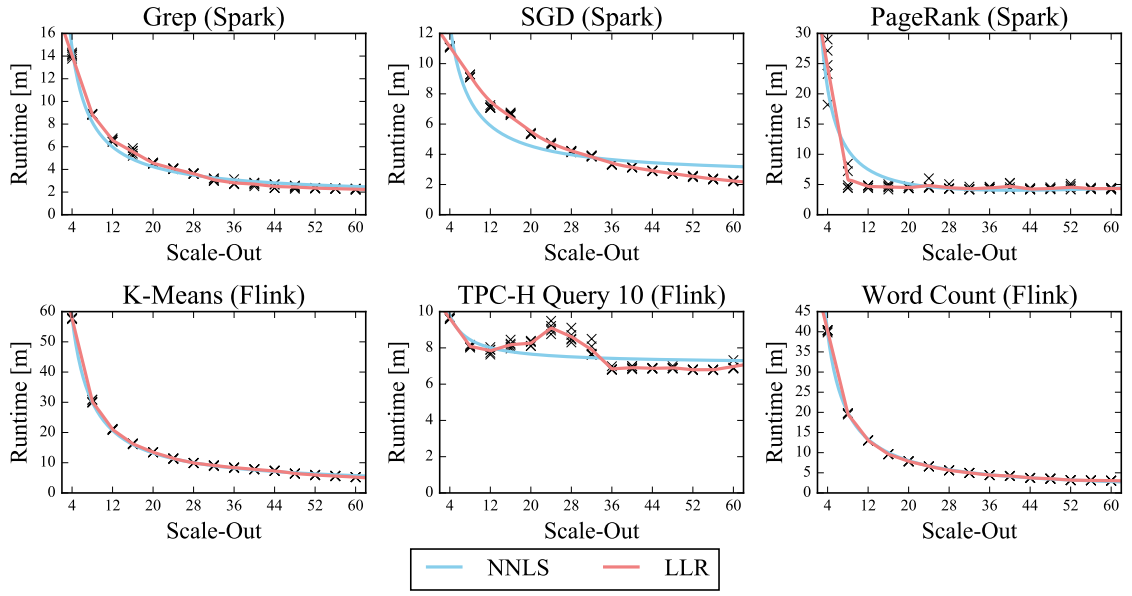


Figure 5.5: Runtimes of 75 runs of the six benchmark jobs along with the fitted curves. © 2016 IEEE [122].

However, while the nonparametric model seems superior in the presence of these amounts of training 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 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 5.6 shows the results. NNLS is again the parametric and LLR the nonparametric model. Bell is performing model selection via cross-validation beforehand.

As expected, with increasing amounts of training data points and hence higher density of the training dataset, the nonparametric method outperforms the parametric one. On the other hand, the nonparametric model is outperformed by the parametric one for smaller datasets.

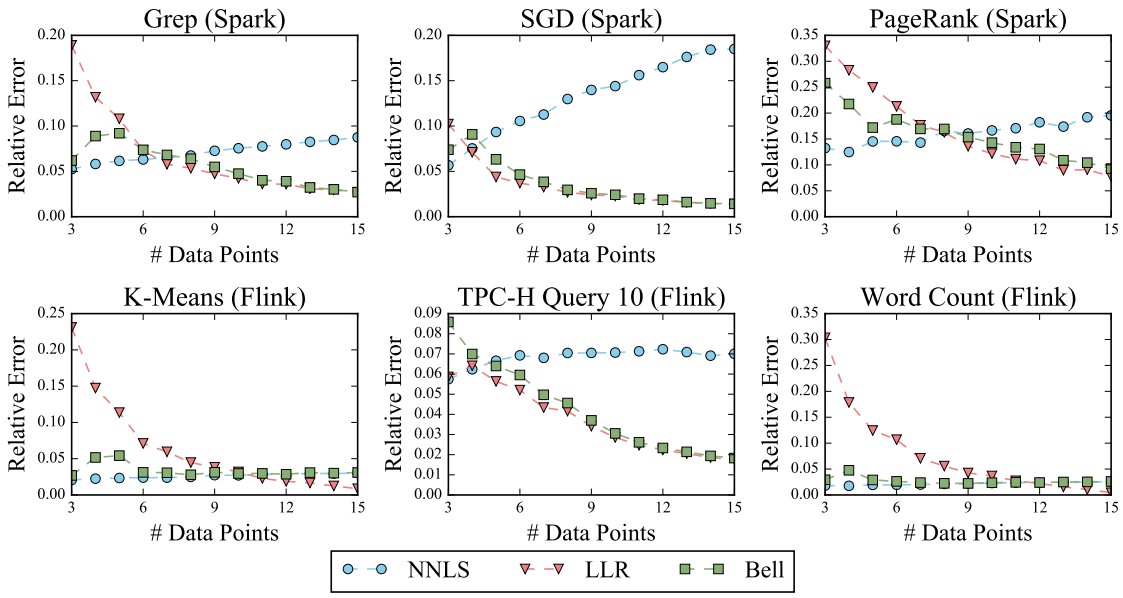


Figure 5.6: Mean relative prediction error for each benchmark job as reported by the repeated random sub-sampling cross-validation. © 2016 IEEE [122].

We also calculated the averages of the mean prediction errors for every amount of training data points over all six benchmark jobs. These averages are shown in Figure 5.7 and Table 5.2. Compared to the parametric model, Bell has a slightly higher mean prediction error with less than six training data points, but then outperforms the model. Compared to the nonparametric model, Bell has a significantly lower mean prediction error when there are less than seven training data points, while it performs similarly

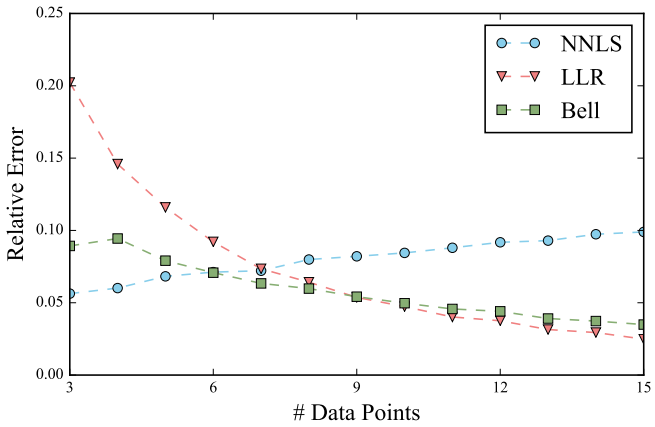


Figure 5.7: Mean relative prediction error over all benchmark jobs, averaging the mean relative errors of the repeated random sub-sampling cross-validations.

to the model for datasets with a higher density. For our benchmark jobs Bell needs on average ten similar previous runs for a mean relative prediction error of less than 5%. Moreover, for all tested amounts of available training data Bell predicts the runtimes of the benchmark jobs on average with a mean relative prediction of less than 10%.

Table 5.2: Mean Relative Prediction Error Over All Benchmark Jobs.

# Data Points	NNLS [%]	LLR [%]	Bell [%]
3	5.63	20.23	8.93
4	6.01	14.59	9.44
5	6.83	11.59	7.91
6	7.12	9.21	7.08
7	7.21	7.35	6.34
8	7.99	6.42	5.98
9	8.21	5.37	5.42
10	8.45	4.73	4.97
11	8.80	4.02	4.57
12	9.18	3.75	4.41
13	9.30	3.15	3.91
14	9.74	2.94	3.73
15	9.90	2.49	3.49

We conclude that it is crucial to combine the flexibility of the nonparametric model with the robustness of the parametric model and, therefore, to incorporate a mechanism that switches automatically between these two models. By using cross-validation to select between models, our approach is able to detect the better performing model with increasing amounts of training points. At the same time, our approach keeps the relative prediction 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 a relative error that is closer to the one of the parametric model than the nonparametric model.

Finally, to evaluate the overhead that using Bell can introduces to resource management, we fitted the 75 data points of each of the six benchmark jobs 10 times. The median runtime for this microbenchmark 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.

6 Estimating Job Runtimes Based on Similar Previous Executions

Contents

6.1 Predicting Job Performance Based on Previous Executions 74
6.2 Assessing the Similarity of Job Executions 76
6.2.1 Similarity Measures 76
6.2.2 Similarity Quality 80
6.2.3 Training Job-Specific Thresholds and Weights 83
6.3 Estimating the Remaining Runtime of Recurring Iterative Jobs 84
6.3.1 Estimate Inference 85
6.3.2 Final Estimate 86
6.3.3 Outlier Iterations 86
6.4 Evaluation 87
6.4.1 Cluster Setup 87
6.4.2 Experiments 87
6.4.3 Results 89

This chapter presents Cutty and SMiPE, which we published in [132] (© 2017 IEEE). Cutty is a system that selects previous runs of a job as a basis for performance estimation. Specifically, Cutty matches previous runs of a job based on their similarity to the current job. It uses multiple different measures of similarity including statistics on the input data, job stages, and resource usage. Some of these measures, for example basic information on the input datasets, are available offline. Yet, estimating the impact of differences in these measures on the actual runtime behavior of jobs is usually not straightforward. Other factors, such as stage runtimes and resource utilization statistics, are only available when a job runs but do effectively capture the runtime behavior of a job. Cutty uses as many measures as are available to match job executions. In turn, Cutty's similarity matching is able to become more accurate as a job progresses and more statistics become available. Since similarity in different measures can be useful for accurately estimating a job's runtime, we use thresholds and weights when combining individual similarity measures into an aggregated overall measure of similarity. Cutty trains these parameters for each job and thereby automatically adapts to the characteristics of jobs.

We originally developed Cutty for SMiPE, as published in [132]. SMiPE is a system that estimates the progress of iterative distributed dataflow jobs based on similar previous job

executions. For selecting these similar previous executions, SMiPE uses Cutty. SMiPE estimates the remaining runtime of iterative distributed dataflows by extrapolating the runtime behavior of the current execution on the basis of the iteration runtimes of the matched previous executions. It creates an estimate for each similar previous executions, before creating a weighted overall estimation from the individual estimates.

Cutty can be used with different performance estimation systems, including Bell, the runtime prediction system we presented in the previous chapter. Cutty is more general than SMiPE in that it supports non-iterative jobs by matching statistics on stages, instead of iterations. We use Cutty as a component of Ellis, as presented in the overall system architecture in Chapter 4.4, without the estimation component of SMiPE, to select training data for predicting the runtimes of general distributed dataflow jobs with Bell.

This chapter first describes the general approach of predicting the runtime of a distributed dataflow job based on similar previous executions. Second, the chapter presents the methods we implemented with Cutty: different similarity measures and how we assess the overall similarity, including how weights and thresholds used for combining individual similarity measures into an overall assessment of similarity can be trained automatically on the history of a job. Then, the chapter describes how SMiPE estimates the remaining runtime of iterative distributed dataflow jobs based on the iteration runtimes of similar previous job executions. Finally, the chapter presents an evaluation of Cutty and SMiPE and a workload of multiple Spark jobs.

6.1 Predicting Job Performance Based on Previous Executions

When a job is executed repeatedly, previous executions of the job can be used to model and predict the performance of the job, assuming the job will behave as before. However, it is usually not the exact same job that is executed, even with periodically running batch jobs. Usually the dataset and the selected resources, but possibly also the program code, parameters, and system configurations, differ slightly. Yet, it is hard to estimate the effect that such changes have on the performance of a job. For this reason, multiple runtime prediction systems use black-box approaches to performance modeling. These systems require only basic information about sample executions for their models. For instance, the prediction system we presented in the previous chapter trains its models exclusively using pairs of scale-outs and runtimes. These black-box prediction systems only assume that a job will behave similar to the examples used for training. However, this requires that a job's performance is modeled on previous executions that actually behaved similar and thus allow to accurately estimate the performance of the job that is to be executed. Therefore, it is essential that previous job executions with a similar runtime behavior are selected as a basis for black-box models.

Before a job is executed only limited statistics are available to match similar previous executions. These offline statistics concentrate on factors like the input data and the

resources used. As soon as a job executes, though, statistics on the runtime behavior become available. Important runtime statistics are for example the runtime of stages, dataset convergence across subsequent stages, and resource utilization. Such runtime statistics reflect the impact that updates of datasets, changes to programs, and adapted system configurations have on the performance of distributed dataflow jobs. Therefore, runtime statistics allow to match previous executions based on the actual runtime behavior. Moreover, such runtime statistics also reflect changes to the cluster state. For instance, failures and congested resources in shared commodity clusters can impact the runtimes of stages significantly.

Multiple statistical measures can be used to assess the similarity between two job executions. Yet, it is not clear that a similarity in a particular measure actually correlates with a similar runtime behavior and in turn similar job runtimes. For a job that only searches for particular elements in the input, the size of the input datasets will significantly determine the job's runtime. While for relational queries the selectivity and thus convergence across subsequent stages is an important factor for the runtime, which is often highly dependent on particular program parameters. This is similar for iterative jobs that compute incrementally on a shrinking active dataset. We approach this problem in two ways. First, we combine multiple measures of similarity into an overall measure. Second, we use weights and thresholds when combining multiple measures, taking into account that some measures are more important than others and that a certain extent of similarity is required for a previous execution to be useful for an estimation task. These weights and thresholds can be set for each job as it depends on the job which factors have a significant impact on its runtime behavior.

In fact, it depends on both the job and the estimation system, whether a particular measure of similarity is useful for estimating the runtime behavior of a distributed dataflow job. We define that a particular similarity measure is useful for performance estimation, when a high similarity in the measure correlates with similar runtime behavior and therefore accurate estimations. We call this characteristic of a measure of similarity between job executions *similarity quality*. Similarity quality is not universal, but valid for a particular estimation system and a particular job history. To assess the similarity quality a specific similarity measure has, we thus require a particular estimation system. Specifically, we test the accuracy of estimations when one job execution is estimated based on another one. Given an estimation for one of the executions based on the other one and the actual performance for the estimated execution, we compute the accuracy of the estimation and use this for assessing the similarity quality of the similarity measure. If the computed similarity quality suggests that a similarity measure is not useful for selecting previous executions, this does not necessarily mean that the measure is never useful. It is possible that the particular estimation system is not sophisticated enough to make use of the properties that the similarity captures.

Given the entire history of executions for a recurring job, we can automatically determine weights and thresholds that combine individual similarity measures into an overall

similarity measure with a high similarity quality. This overall similarity measure can then be used to match previous executions of a job that provide accurate runtime estimations.

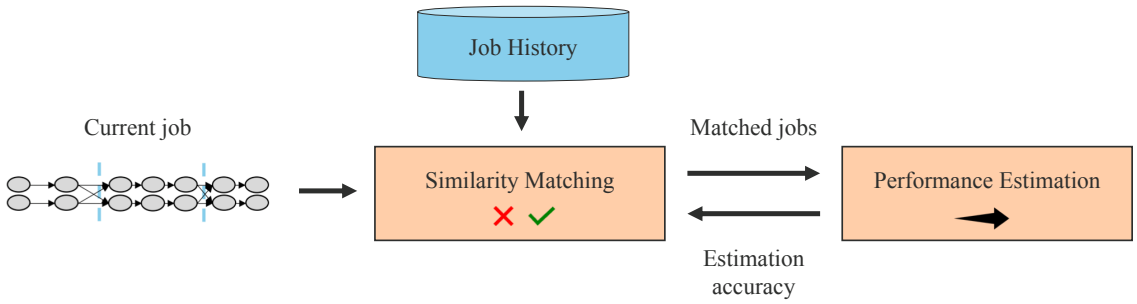


Figure 6.1: Using similar previous executions of recurring jobs as a basis for performance estimation, while configuring the similarity matching based on feedback from the estimation system.

Figure 6.1 shows our idea of similarity matching for performance estimation. First, for a current job we select previous executions in a step we call *similarity matching*. These executions are the input to a *performance estimation* step, which uses the matched previous executions to model and estimate the performance of the current job. The estimation system returns the estimation accuracy to the matching system. This feedback is used by the matching system to determine the usefulness of similarity measures. Based on this assessment, the matching system automatically configures weights and thresholds used for combining multiple similarity measures into an aggregated overall similarity measure. Specifically, the matching systems trains these parameters using the entire history of previous executions of a job.

6.2 Assessing the Similarity of Job Executions

In this section we describe Cutty, a system for selecting previous executions that are similar to a currently running job with the goal of performance estimation. First, we present five different measures for the similarity between job executions. Afterwards, we present a method for assessing whether a similarity measure is useful for performance estimation, which is the property we call similarity quality. Finally, we show how we use the similarity quality and all previous executions of a job to automatically set the thresholds and weights used for combining individual similarity measures.

6.2.1 Similarity Measures

Various measures can be used to rate the similarity of two job executions. These measures are statistics on factors that either determine or capture the performance of distributed dataflow jobs such as statistics on the program, the input data, and the resource allocation or statistics on the execution of particular stages and the resource utilization.

With our matching system Cutty, we use five similarity measures:

- Input Similarity,
- Runtime Similarity,
- Scale-Out Similarity,
- Convergence Similarity,
- and Resource Utilization Similarity.

These similarity measures are used to compare a current job execution ex_{cur} to a previous execution ex_{prev}. Four of these five similarity measures cover runtime statistics. For these measures, we record and compare statistics for job stages. That is, we compare the current job execution to previous executions at a particular stage x_{cur} and assume that statistics on already finished stages are also available for the current job.

6.2.1.1 Input Similarity

The Input Similarity compares the input datasets. Recurring jobs often process the same evolving datasets. The datasets these jobs process are consequently similar in both size and other key characteristics. As a measure of the similarity of the inputs we compare the absolute numbers of the records in the input datasets.

The Input Similarity is defined as

$$sim\left(ex_{cur}, ex_{prev}\right) \quad = \quad max\left(0,\ 1\ -\ \frac{\mid records\left(ex_{cur}\right) - records\left(ex_{prev}\right)\ \mid}{records\left(ex_{cur}\right)}\right),$$

where $records(ex)$ returns the number of records in the input dataset of execution ex. We use the maximum function to ensure that the similarity values are in the range between 0 and 1.

6.2.1.2 Runtime Similarity

The Runtime Similarity compares the runtimes of stages up to the last stage finished by the current job execution. The intuition behind this measure is that if the runtimes of stages have been similar in two executions so far, they are probably going to be similar for the remaining stages as well. For the Runtime Similarity measure we calculate the relative deviations of the runtimes of the stages of the two executions for each stage finished by the current execution. We then calculate the average of these deviations of the stage runtimes. If the two executions were not executed using the same number of containers, we first use a scale-out model to adjust the stage runtimes of the previous execution. Specifically, we use Bell, which we presented in the previous chapter, to model and adjust the stage runtimes.

The Runtime Similarity is defined as

$$sim\left(ex_{cur}, ex_{prev}, x_{cur}\right) \quad = \quad max\left(0,\ 1\ -\ \frac{1}{x_{cur}} \quad \sum_{i=1}^{x_{cur}} \Delta_r\left(ex_{cur}, ex_{prev}, i\right)\right),$$

where

$$\Delta_r\left(ex_{cur}, ex_{prev}, x\right) \quad = \quad \frac{\mid runtime\left(ex_{cur}, x\right) - runtime\left(ex_{prev}, x\right) \mid}{runtime\left(ex_{cur}, x\right)}.$$

The stage-wise average of all relative deviations is taken. As less deviation means greater similarity, we subtract this average from 1. We again use the maximum function to ensure that the similarity values are between 0 and 1.

6.2.1.3 Scale-Out Similarity

The Scale-Out Similarity compares the number of containers used for execution. Although using a model of the scale-out behavior of a job allows to adjust runtimes according to their scale-out, this may introduce inaccuracies, making it worth to compare jobs also in regard to their resource allocations. In particular, we compare the average number of containers used for the stages up to the last stage finished by the current job execution.

The Scale-Out Similarity is defined as

$$sim\left(ex_{cur}, ex_{prev}, x_{cur}\right) \quad = \quad \frac{min\left(containers_{avg}\left(ex_{cur}, x_{cur}\right),\ containers_{avg}\left(ex_{prev}, x_{cur}\right)\right)}{max\left(containers_{avg}\left(ex_{cur}, x_{cur}\right),\ containers_{avg}\left(ex_{prev}, x_{cur}\right)\right)},$$

where $containers_{avg}\left(ex, x\right)$ returns the average number of containers used by all stages up to the current stage x. The minimum and maximum functions are again used for a value between 0 and 1.

6.2.1.4 Convergence Similarity

The Convergence Similarity compares the number of records processed by each stage of an execution. The convergence behavior is mainly influenced by the input datasets and the algorithm parameters, but can also depend on the partitioning and the parallelism. Similar to the idea behind the runtime similarity, we assume that if the convergence is similar to the convergence in previous executions up to a stage, the convergence will also be similar in the remaining stages. For this similarity measure, we use the average of the differences in the number of records actively processed by each of the job stages, up to the last stage finished by the current execution.

Similar to the Runtime Similarity, the Convergence Similarity is defined as

$$sim\left(ex_{cur}, ex_{prev}, x_{cur}\right) \quad = \quad max\left(0,\ 1\ -\ \frac{1}{x_{cur}} \quad \sum_{i=1}^{x_{cur}} \Delta_a\left(ex_{cur}, ex_{prev}, i\right)\right),$$

where

$$\Delta_a\left(ex_{cur}, ex_{prev}, x\right) \quad = \quad \frac{\mid activeRecords\left(ex_{cur}, x\right) - activeRecords\left(ex_{prev}, x\right) \mid}{activeRecords\left(ex_{cur}, x\right)}$$

and $activeRecords\left(ex, x\right)$ returns the number of records actively processed by stage x of execution ex.

6.2.1.5 Resource Utilization Similarity

The Resource Utilization Similarity captures how executions utilize hardware resources. We use the utilizations of CPU cores, disks, and network links. For these resources, we record the average utilization for every container and each stage. For comparison between two executions that were executed on the same amount of containers, we use the values from all containers of the previously finished stage. In particular, we sort these averages and compare them pairwise, averaging the deviations between all pairs. This is based on the observation that how the hardware utilization is distributed across workers is more characteristic of an execution than a global average. If, however, two executions were executed using different numbers of containers, we fall back to comparing the global average.

We define $util_{container}\left(ex, x, hw, c\right)$ as the arithmetic mean of all utilization values of the hardware component hw attributed to a container c and gathered during execution ex, from the beginning of the execution until stage x. We further define $utilSeq\left(ex, x, hw\right)$ as the sorted sequence of the values $util_{container}(ex, x, hw, c)$ for all containers used by a job execution. It is then possible to compare the values of two executions individually, in the same way as runtimes are compared for the Runtime Similarity.

The Resource Utilization Similarity of a hardware component hw is defined as

$$sim\left(ex_{cur}, ex_{prev}, x_{cur}\right) \quad = \quad max\left(0,\ 1\ -\ \frac{1}{containers} \quad \sum_{i=1}^{containers} \Delta_h\left(ex_{cur}, ex_{prev}, x_{cur}, i\right)\right),$$

where $containers$ is the number of containers used in the execution and

$$\Delta_h\left(ex_{cur}, ex_{prev}, x, c\right) \quad = \quad \frac{\mid utilSeq\left(ex_{cur}, x, hw\right)[c] - utilSeq\left(ex_{prev}, x, hw\right)[c] \mid}{utilSeq\left(ex_{cur}, x, hw\right)[c]}.$$

Note that this comparison is only possible for two executions that have been using the same number of containers for the entire duration of their execution. If this is not

the case, we fall back to comparing the global average utilization. For this, we define $util_{avg}\,(ex, x, hw)$ as the arithmetic mean of the $util_{container}$ values of all containers used in the execution ex for a hardware component hw and up to a stage x. The global average of the Resource Utilization Similarity of a hardware component hw is then defined as

$$sim\left(ex_{cur}, ex_{prev}, x_{cur}\right) \quad = \quad max\left(0,\ 1\ - \Delta_h\left(ex_{cur}, ex_{prev}, x_{cur}\right)\right)\ ,$$

with

$$\Delta_h\left(ex_{cur}, ex_{prev}, x_{cur}\right) \quad = \quad \frac{\mid util_{avg}\left(ex_{cur}, x_{cur}, hw\right) - util_{avg}\left(ex_{prev}, x_{cur}, hw\right)\ \mid}{util_{avg}\left(ex_{cur}, x_{cur}, hw\right)}\ .$$

6.2.2 Similarity Quality

The similarity measures capture similarities between job executions that are potentially useful for selecting previous executions as samples for an estimation system. Whether a particular similarity measure is useful depends on both the job and the estimation systems. The performance of some jobs highly depends on convergence, while others depend more on the size of the input and the resources used. Moreover, whether jobs have to be similar in a particular dimension for accurate estimations also depends on the estimation system. For example, a black-box estimation system that only uses pairs of scale-outs and runtimes depends much more on the similarity of its training data than a white-box system that incorporates the differences into its model. To capture the usefulness similarity measures have for a job and an estimation system, we use what we call similarity quality: Similarity measures have a high quality for selecting previous executions as a basis for estimating the performance of jobs, when a high similarity in the measure also yields a high average estimation accuracy. Thus, similarity quality is a measure of the relationship between similarity and accuracy. Consequently, we use the similarity quality to assess which similarity measures are useful in estimating the performance of a job with a particular estimation system.

To determine the similarity quality a similarity measure has for a job and an estimation system, we analyze the job's history. Specifically, we compute the similarity between all previous executions and the estimation accuracy when one execution is estimated based solely on the other. For this, we simulate that an execution j from the job history is currently running and has just finished a particular stage x. Then, we select another execution k from the job history. First, we calculate s, the value of a similarity measure for the similarity between j and k. Then we request an estimate from the estimation system for the remaining runtime after stage x of execution j, based only on k. We know the actual remaining runtime of j and, therefore, can calculate the estimation accuracy a. We repeat this process for all job executions k and j in the history, using different values for the current stage x. As a result we get P, a set of points (s, a), where s is the similarity between the simulated current execution j and the selected execution k, and where a is the estimation accuracy when the runtime of j is estimated solely based on k. The points in P

show the relationship of the similarity in a specific similarity measure and the estimation accuracy, when the runtime of executions is estimated based on previous executions, for a specific job and a specific estimation system.

We define three metrics based on the points (s, a) in P for the similarity quality:

$$\begin{aligned} \text{cumulative histogram } h(t) &:= avg\{\, a \mid (s,a) \in P, \quad s \geq t \,\}, \\ \text{point share } n(t) &:= \frac{|\{\, a \mid (s,a) \in P, \quad s \geq t \,\}|}{|P|}, \text{ and} \\ \text{normalized similarity quality } q(i) &:= h(y), \text{ with } n(y) = 1 - i\,. \end{aligned}$$

These metrics indicate the similarity quality of a selection t, selecting points (s, a) of P where $s \geq t$. First, there is the cumulative histogram $h(t)$, which is the average accuracy of points (s, a) for a specific selection threshold t. Second, there is the point share $n(t)$, which is the share of points selected for a specific selection threshold t. Third, there is a measure we call *normalized similarity quality* $q(i)$, which combines the histogram and the point share into one measure to make similarities more comparable.

A similarity measure is useful for a job and an estimation system, if it is possible to find a threshold t for which the histogram $h(t)$ indicates a high average accuracy, while the point share $n(t)$ indicates that enough previous executions will be matched as a basis for performance estimation. We use the histogram and point share for training weights and thresholds.

Figure 6.2 shows the points (s, a) of P of an example similarity measure and job. Besides showing the points in P, the chart also shows the cumulative histogram and the point share. If the similarity threshold t is set to a high value such as t_2, the histogram shows that a high average accuracy can be expected. However, the point share would be very low. Therefore, it might be difficult to find previous executions similar enough to be matched for a current execution. However, estimating the performance of a current job based on similar previous executions is not possible, when no or too few previous executions are selected. In comparison, a lower value t_1 still yields accurate estimations, but is expected to match considerably more executions. We therefore need to consider both the average accuracy and the point share that a specific selection threshold t yields.

The normalized similarity quality combines the histogram and the point share of a similarity measure into a single metric. It normalizes the histogram on the point share, returning the average accuracy of each point share. The normalized similarity quality is shown in Figure 6.3 for the same exemplary similarity measure and job we used for Figure 6.2.

The input of the normalized similarity quality metric, $0 < i < 1$, denominates the share of points (s, a) that are most similar. We deduct i from 1 as we compare the average

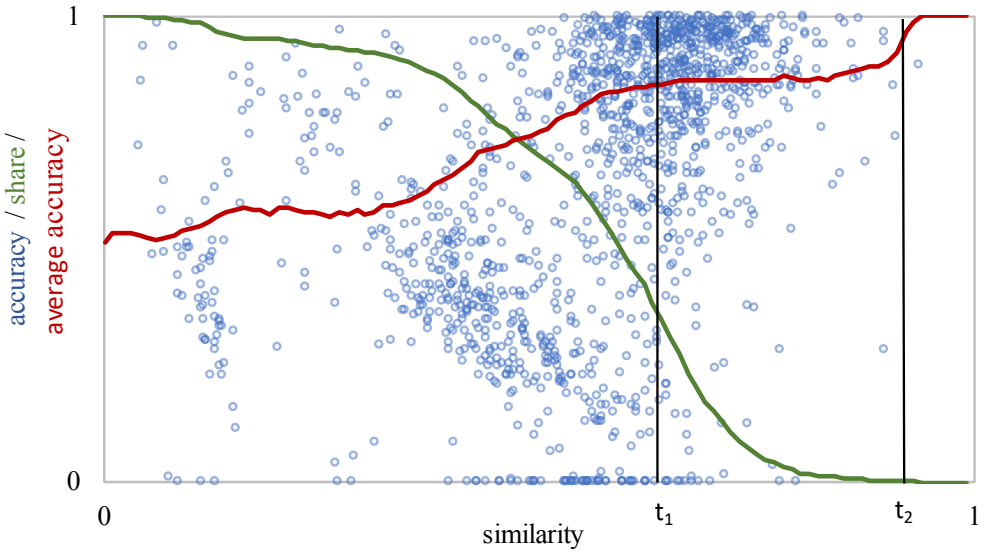


Figure 6.2: Similarity quality histogram (red), point share (green), and points (blue) of an example similarity measure. © 2017 IEEE [132].

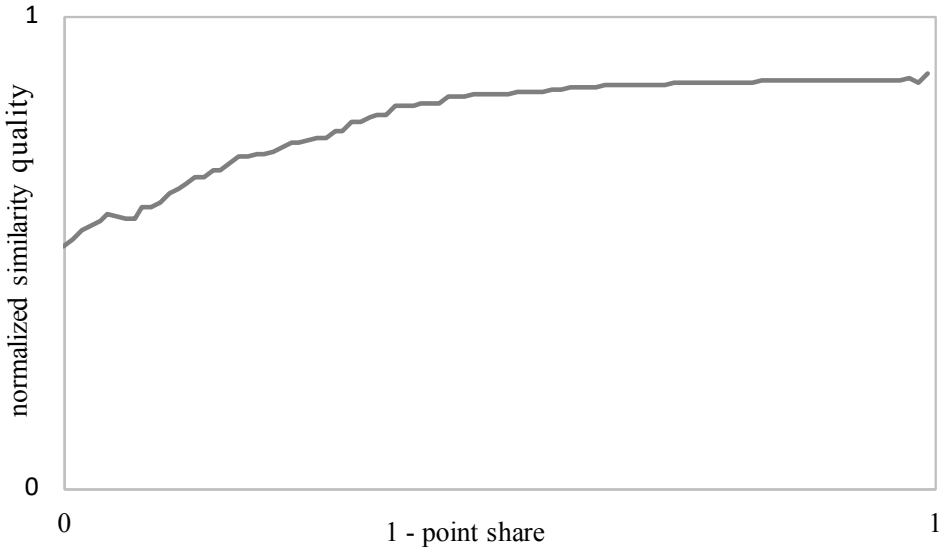


Figure 6.3: Normalized similarity quality of the example similarity measure.

accuracy of a particular point share, which increases as the similarity threshold increases and is limited by 1. For example, using the value 0.75 for i means we are comparing the average accuracy of points of a point share of 0.25, which are the 25% most similar exe-

cutions. Consequently, the normalized similarity quality allows to compare the average accuracy that similarity measures have at the same relative similarity, for example the average accuracy of the most similar 25% of executions, and we thus use the normalized similarity quality for the evaluation of similarity measures.

6.2.3 Training Job-Specific Thresholds and Weights

We are using multiple individual similarity measures to assess the similarity of two executions of a job. We combine these individual measures into an overall similarity measure using thresholds and weights. That is, each similarity has a threshold t and a weight w. These two parameters of each individual similarity measure configure which jobs are matched by Cutty. The thresholds determine which jobs are discarded. The weights of the individual similarities determine the overall similarity. That is, the overall similarity measure is a weighted average of the individual similarity measures. Instead of setting the weights and thresholds manually, we make use of a job's history and optimization to select values for these parameters automatically.

6.2.3.1 Similarity Thresholds

The similarity thresholds determine which jobs are matched for the estimation system. There is a threshold for each individual similarity measure, discarding jobs that are not similar enough in a particular dimension. Additionally, there is a threshold for the overall similarity measure, discarding jobs that are not similar enough in total.

We use the job history and the similarity quality measures to train the thresholds. We set a minimum average accuracy h_{min}. Additionally, we set a minimum point share n_{min} to avoid having a threshold for which not enough executions are matched. Then, for each similarity measure we search for and choose the highest threshold t that fulfills both $h(t) \geq h_{min}$ and $n(t) \geq n_{min}$. That is:

$$t = max\left\{t \in [0,1] \,|\, h(t) \geq h_{min} \wedge n(t) \geq n_{min}\right\}.$$

We choose the highest threshold t that fulfills both constraints in order to maximize the average estimation accuracy. If all executions are discarded, the thresholds of the similarity measures are lowered. This is achieved by multiplying each threshold by a factor $f_{red} < 1$. This procedure is repeated until at least one similar job execution can be matched, but not more than m times. If still no executions can be matched after m executions of this procedure, the estimation system returns an error.

6.2.3.2 Similarity Weights

For the executions not discarded, a weighted average is calculated from the individual similarity values. We make use of optimization to find weights that provide the

highest average accuracy. That is, an optimal solution must be found for the vector $w = (w_1, w_2, ..., w_n)$, where each w_i weights an individual similarity measure. We make use of simulations for this: For every execution of a job, we simulate the execution is currently running at a particular stage, using different values as current stage, and estimate the remaining runtime based on all other executions of the job, matched with different weights. To find optimal weights we use numerical optimization, specifically Powell's BOBYQA algorithm [133]. We use the mean relative estimation error of the set of simulations as objective function o:

$$o(w) = avg \left\{ \frac{|actual - estimate(w)|}{actual} \mid (estimate(w), actual) \in S(w) \right\}$$

The set $S(w)$ contains the results of the simulations in the form $(estimate(w), actual)$. $estimate(w)$ is the result of the estimation algorithm using w as the weights vector. $actual$ is the actual remaining runtime known from the job history. As BOBYQA allows negative solutions, the objective function is adjusted by a large value, thereby forcing positive solutions. We perform the optimization repeatedly with different initial values for the weights. The result is a weights vector w yielding a minimized mean relative estimation error.

Using Powell's BOBYQA algorithm has the advantage that the objective function is specified as a black box. That is, no assumptions about its form such as linearity are required.

6.3 Estimating the Remaining Runtime of Recurring Iterative Jobs

In this section we describe SMiPE, a system for estimating the remaining runtime of currently running iterative distributed dataflow jobs. SMiPE estimates the runtime of the remaining iterations of the current job on the basis of the iteration runtimes of similar previous executions. For selecting these similar previous executions, SMiPE uses Cutty, the similarity matching system we presented in the previous section.

SMiPE is a system specifically for estimating the remaining runtime of iterative distributed dataflow jobs. As such, it does not operate at the granularity of stages, but matches job executions and estimates the remaining runtime based on previous and remaining iterations. This allows, for example, to more accurately match the convergence of incremental iterative processing and also to detect outlier iterations.

Figure 6.4 shows a conceptual overview of SMiPE's approach. The approach is divided into three steps: (1) similarity matching, (2) estimation inference, and (3) final estimate. As SMiPE uses Cutty for similarity matching, we only describe how SMiPE infers individual estimates and then aggregates these into a final estimate in the following.

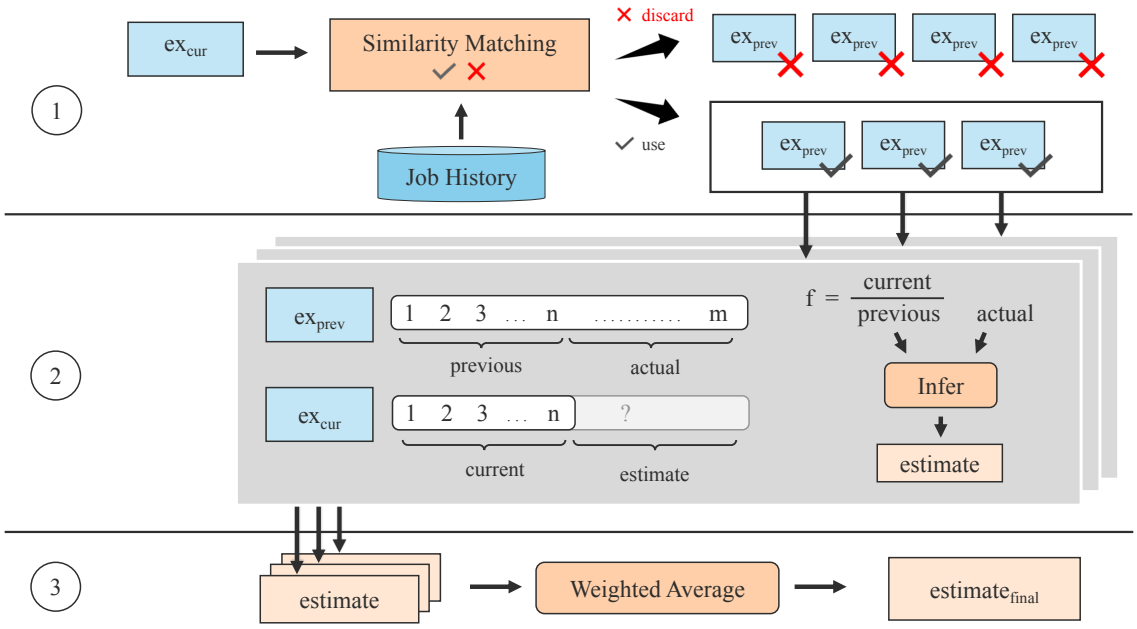


Figure 6.4: Overview of estimating the remaining runtime of currently running iterative distributed dataflow jobs based on similar previous executions. © 2017 IEEE [132].

6.3.1 Estimate Inference

SMiPE estimates the remaining runtime of the current execution of an iterative job based on matched similar previous executions. Specifically, SMiPE assumes that the remaining iterations of the current job will have runtimes comparable to the iterations of the matched previous executions. Based on each matched execution, SMiPE creates an estimate for the runtime of the remaining iterations. These individual estimates are later combined into a final estimate.

SMiPE calculates the estimate based on a single matched execution as follows. First, SMiPE assumes that the current execution will have the same number of iterations as the matched execution. The current job execution probably behaves slightly different than the historic job execution, since it usually will not be the exact same job and the exact same situation in the cluster. For this reason, we first compare the differences in iteration runtimes of the current execution and the matched previous execution, by calculating $f_i = runtime_{current} / runtime_{similar}$ for each finished iteration i of the current execution. We assume that the differences in runtimes up to the current iteration will on average also be true for the remaining iterations. Thus, we compute the average difference for all the iterations up to the current iteration x_{cur}: $f_{total} = \frac{\sum_{i=1}^{x_{cur}} f_i}{i}$. As the overall estimate for the remaining runtime of the current job, we now take the runtimes of the matched previous execution's remaining iterations and adjust the sum of these runtimes by f_{total}.

To account for the differences in resource allocations, the runtimes of the matched previous executions that used a different scale-out are adjusted to be applicable for estimating the runtime of the current execution. For this, we use a scale-out model that captures how runtimes change with the number of containers. Specifically, we use Bell, which we presented in the previous chapter, to model the scale-out behavior of iterations. We use the scale-out model for calculating the factor by which the runtimes of two executions differ and adjust the runtime of the similar previous execution by this factor. We perform the adjustment iteration-wise and, thus, support jobs that dynamically adjust resource allocations during the execution. The adjustment of the matched previous execution is not only done for the iterations up to the current iteration x_{cur}, but also for the remaining iterations. By default, SMiPE assumes that the current execution will continue using the same resource allocation as in the current iteration. If the estimator was, however, to be used in conjunction with a system that dynamically manages resource allocations, the dynamic allocation system could supply the number of nodes as a parameter to the estimator, which would in turn be able to estimate the remaining runtime based on the actually used resources.

Lastly, SMiPE weighs iterations, giving more weight to more recent iterations, assuming that recent trends in iterations runtimes due to for example changes in the cluster state will hold for the following iterations. SMiPE uses the inverse function $1/x$ for weighting more recent iterations.

6.3.2 Final Estimate

The set of estimates based on individual matched previous executions, which we calculated in the previous step, are now combined into a final estimate using a weighted average. As weights, the overall similarity values of the matched executions are used. That is, matched previous executions more similar to the current execution contribute more to the final estimate.

Due to the thresholds used in selecting previous executions, the differences in the overall similarity values can be relatively small. For this reason, the overall similarity values can be adjusted using the function sim^b ($b > 1$) for the final estimate, increasing the difference between weights to make the weighting more effective. By default SMiPE uses the value $b = 50$.

6.3.3 Outlier Iterations

Sometimes executions have iterations with runtimes that deviate drastically from expected values. This can have many reasons such as resource congestion due to interference between co-located workloads or failures. We call these iterations *outlier iterations*. Outlier iterations can have a negative effect on the accuracy of the estimation system.

Outliers before the current iteration lead to distorted iteration factors as the factors are based on iteration-wise differences in runtimes. For this reason, SMiPE ignores outlier iterations in the calculation of the average iteration factor. Specifically, we consider an iteration factor value f_i to be an outlier, if it is greater than n times the mean value of all iteration factors.

Outliers after the current iteration lead to distorted values for the remaining runtime. These outliers therefore also negatively impact the estimation accuracy. SMiPE addresses this issue by reducing the final similarity value of executions with outliers in iterations after the current iterations. For that, the outliers in the remaining iterations are counted. An iteration is considered an outlier if its runtime is more than n times that of the previous iteration. The final similarity value sim_{final} is then reduced in relation to the number of outliers by multiplying it with

$$1 - \frac{\text{number of outliers in iterations after } i_{cur}}{\text{number of iterations after } i_{cur}}.$$

Executions with outliers consequently contribute less to the overall estimation without being discarded completely.

6.4 Evaluation

We evaluated Cutty and SMiPE with multiple exemplary Spark jobs. First, we present the similarity quality of the individual similarity measures, showing that different similarity measures are useful for estimating the performance of jobs. Second, we show the overall estimation accuracy when SMiPE is used to estimate the remaining runtime of currently running iterative distributed dataflow jobs based on previous executions that were matched by Cutty.

6.4.1 Cluster Setup

All experiments were done on a cluster of 40 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.

For the experiments, we used Linux (Kernel 3.10.0), Java 1.8, Spark 2.0.0, and Hadoop 2.7.1. With Spark we used the libraries GraphX and MLlib in the versions 1.6.0 and 1.1.0, respectively. For the collection of hardware statistics, we used Dstat 0.7.2.

6.4.2 Experiments

Table 6.1 gives an overview of all experiments, showing the algorithms, datasets, and parameters we used. Each setting was executed on up to 40 nodes and repeated mul-

tiple times: four times for SGD, six times for PageRank, and nine times for Connected Components.

Table 6.1: Overview of Benchmark Jobs.

Algorithm	Input Dataset	Size	Parameters
PageRank	LiveJournal	1.00 GB	d=.01, d=.001, d=.0001
	Kronecker24	1.52 GB	d=.01, d=.001
	Kronecker25	3.43 GB	d=.01, d=.001
	Wiki	5.74 GB	d=.09, d=.01
Connected Components	LiveJournal	1.00 GB	–
	Kronecker24	1.52 GB	
	Kronecker25	3.43 GB	
	Wiki	5.74 GB	
SGD	50-100M	99.4 GB	step size = 1.0 convergence delta = 0.001
	25-250M	124.1 GB	
	25-500M	247.9 GB	
	50-500M	497.0 GB	
	50-750M	745.5 GB	

6.4.2.1 Jobs

We used the algorithms PageRank, Connected Components, and SGD in the experiments. PageRank calculates the importance of every vertex within a graph. The Connected Components algorithm finds all connected components of a graph using iterative label propagation. SGD iteratively finds the minimum or maximum of an objective function, using gradient approximation for increased efficiency. We used Spark's GraphX implementation for PageRank and Connected Components and Spark's library MLlib for SGD. For PageRank, we used different convergence deltas d.

6.4.2.2 Datasets

We used both real-world and generated datasets of different sizes:

Wiki The Wiki dataset is a real graph of encyclopedia pages taken from the English Wikipedia, with edges representing links between pages. The dataset is part of the KONECT graph collection [134].

LiveJournal The LiveJournal dataset is a real graph from the LiveJournal social network, with edges representing friendship relationships. It is also part of the KONECT graph collection.

Kronecker The two Kronecker datasets we used were generated with the graph generator of the BDGS generator suite [129] with 24 and 25 iterations, respectively. We call these two datasets *Kronecker24* and *Kronecker25*.

Feature-Points The Feature-Points datasets were 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. The generated datasets contain 100 to 750 million points and 25 or 50 features. The dataset name *50-100M* stands for a dataset with 50 features and 100 million points.

6.4.3 Results

In the following we present the results of our evaluation of Cutty and SMiPE.

6.4.3.1 Similarity Measures

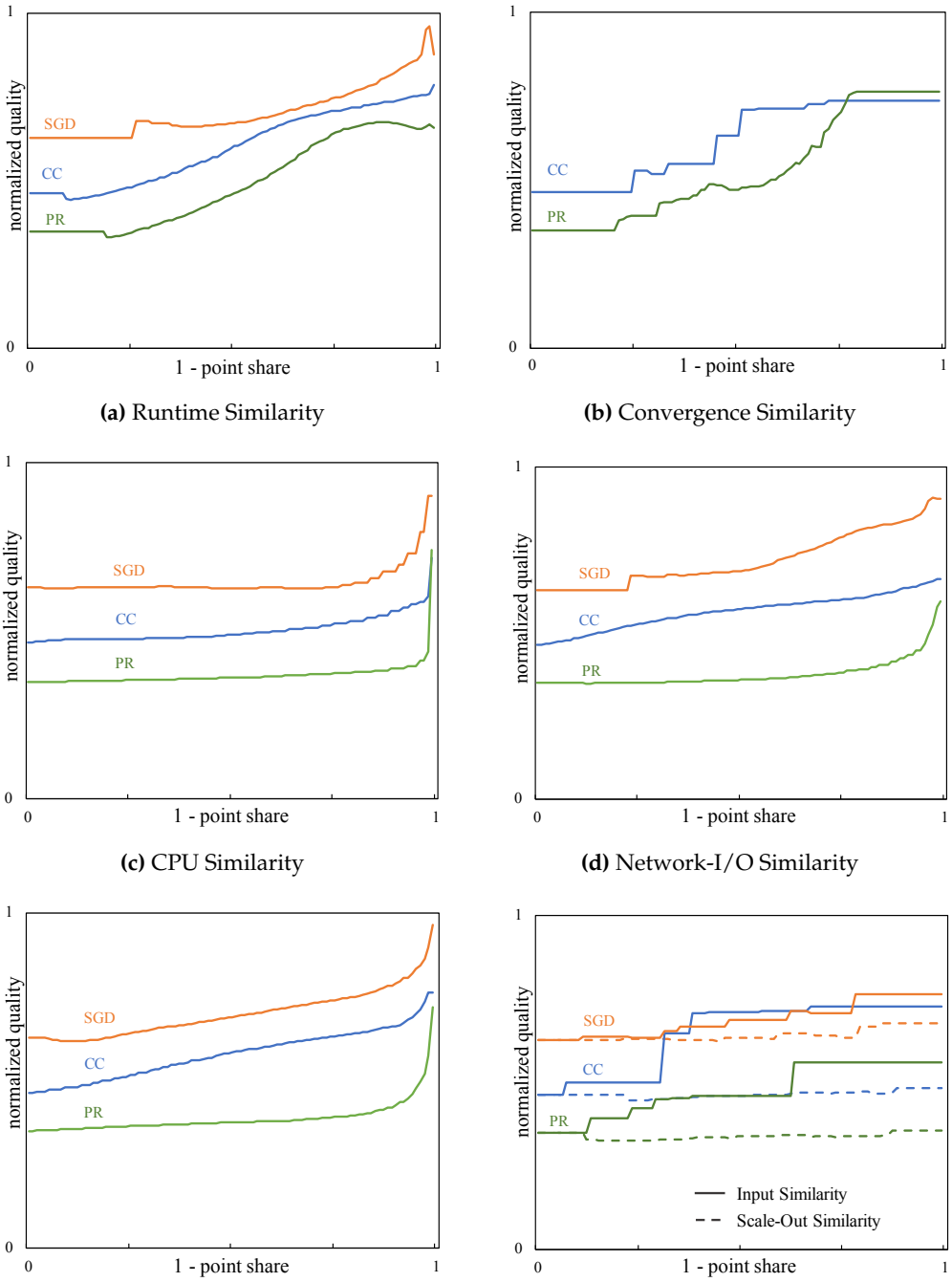
We evaluated the similarity measures by using the normalized similarity quality measure that we defined. In the following, it is simply referred to as similarity quality. We evaluated each of the three jobs using all previous executions. The results are shown in Figure 6.5. CC and PR are the Connected Components and Page Rank algorithms, respectively.

Figure 6.5 a) shows that the Runtime Similarity increases the expected accuracy significantly with increasingly small point shares. The differences in the accuracy are greater for PageRank and Connected Components than for SGD.

The Convergence Similarity, shown in 6.5 b), is compared for PageRank and Connected Components only. The measure captures the convergence behavior of executions by comparing the number of active records per iteration. PageRank and Connected Components both converge in that they work on a shrinking dataset of active records. SGD, in contrast, processes all records in each iteration. Our results suggest that the Convergence Similarity is effective for both PageRank and Connected Components, but more so for PageRank. This is not surprising as in the PageRank experiments multiple convergence delta values were used, which have a significant impact on the job runtime. Distinguishing jobs according to the convergence is therefore important for PageRank.

We present the Resource Utilization Similarity in Figure 6.5 c) to e) for CPU, network, and disk. For CPU, only very high point share values lead to a higher expected accuracy. This effect is most visible for PageRank, where the line is flat until it rises sharply for a point share value close to 1. The network and disk I/O charts show a higher expected accuracy also for lower point share values. These two measures seem effective for all three algorithms.

Figure 6.5 f) shows both the Input Similarity (solid lines) and the Scale-Out Similarity (dotted lines). The Input Similarity is able to distinguish executions according to their datasets for all algorithms. That is, the lines in the chart are not smooth due to the pair-



(a) Runtime Similarity

(b) Convergence Similarity

(c) CPU Similarity

(d) Network-I/O Similarity

(e) Disk-I/O Similarity

(f) Input and Scale-Out Similarity

Figure 6.5: Quality charts of the similarity measures. © 2017 IEEE [132].

wise comparison of the limited number of different datasets. This effect is more notable for Connected Components and PageRank than for SGD. The Scale-Out Similarity only leads to a minimal increase in the average accuracy for higher point share values for all algorithms. A reason for this is that the scale-out property is rather unspecific. For example, the group of executions with the same scale-out will include executions with all input datasets and all algorithm parameters. Yet, the runtimes in that group of executions are different and the mean accuracy of estimations on the basis of this group will be low in turn.

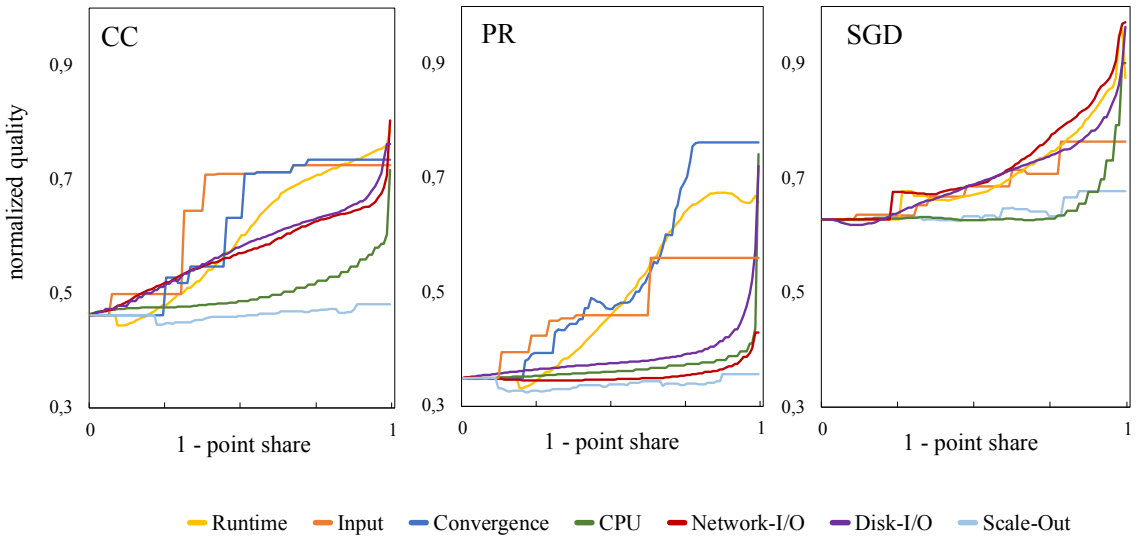


Figure 6.6: Summary of all similarity quality charts. © 2017 IEEE [132].

Figure 6.6 shows the similarity quality charts of all similarity measures for each algorithm. We observe that each algorithm has a distinct profile that determines which similarity measures are the most useful. For Connected Components, both the Input and the Convergence Similarity look useful. For PageRank, the Convergence Similarity seems most useful. For SGD, the Network Similarity looks most useful. This highlights the importance of the parameter training, which allows Cutty to automatically adapt to algorithm-specific profiles.

6.4.3.2 Overall Accuracy

For every execution in the job history, we selected representative sample iterations and calculated the relative error of the runtime estimation by SMiPE. Depending on the execution's total number of iterations, up to three sample iterations were chosen with equally spaced distance between them and an additional padding of 15% of the total iterations at the beginning and the end of the job execution. For each selected sample iteration, we first used SMiPE to estimate the remaining runtime using all executions of the job as job history. We then compared the estimate to the known actual runtime to calculate the

relative estimation error for each selected sample iteration. To judge the overall accuracy, we calculated the mean relative estimation error for each of the algorithms and datasets.

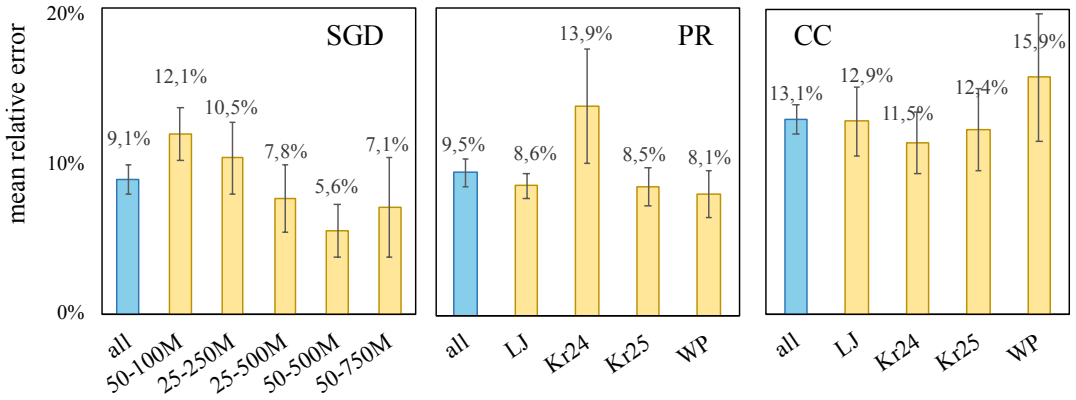


Figure 6.7: Mean relative estimation errors by algorithm and dataset. © 2017 IEEE [132].

Figure 6.7 shows the mean estimation relative error for each of the datasets we used (yellow) and in total for all algorithms (blue). CC is again the Connected Components algorithm and PR is Page Rank. The whiskers in the graph represent the 95% confidence interval. SGD has the lowest mean relative error with 9.1%, followed by PageRank with 9.1%. The mean relative error of Connected Components is 13.1%. This is arguably due to the short runtimes of Connected Components, so that even estimates with a low absolute error yield high relative errors. The mean absolute error for Connected Components is only 1.1 seconds.

For individual datasets, the mean relative estimation errors range from 5.6% to 12.1% for SGD, from 11.5% to 15.9% for Connected Components, and from 8.1% to 13.9% for PageRank. The highest relative mean estimation error is reported for using PageRank on the Kronecker24 dataset. The reason for this high error lies in the specific convergence behavior of PageRank with Kronecker24. The number of active records in the dataset remains almost constant for the entire execution, but then decreases sharply towards the end of the job. Thus, the Convergence Similarity cannot distinguish executions with different delta parameters until the very end of the execution. Thus, executions with lower parameters are considered similar, yet ultimately require more iterations and therefore have a longer overall runtime, leading to considerable estimation errors.

Figure 6.8 shows the iteration-wise relative estimation error for all iterations for the three algorithms. Each line represents executions with the same input dataset and the same algorithm parameters. The x-axis shows the iteration, relative to the total number of iterations executed for the job, so the curves are comparable regardless of the total number of iterations executed. The accuracies are mostly between 5% and 15% in the middle of the executions. Both algorithms exhibit considerably higher errors in the beginning and in the end of the execution. In the beginning, only little runtime statistics

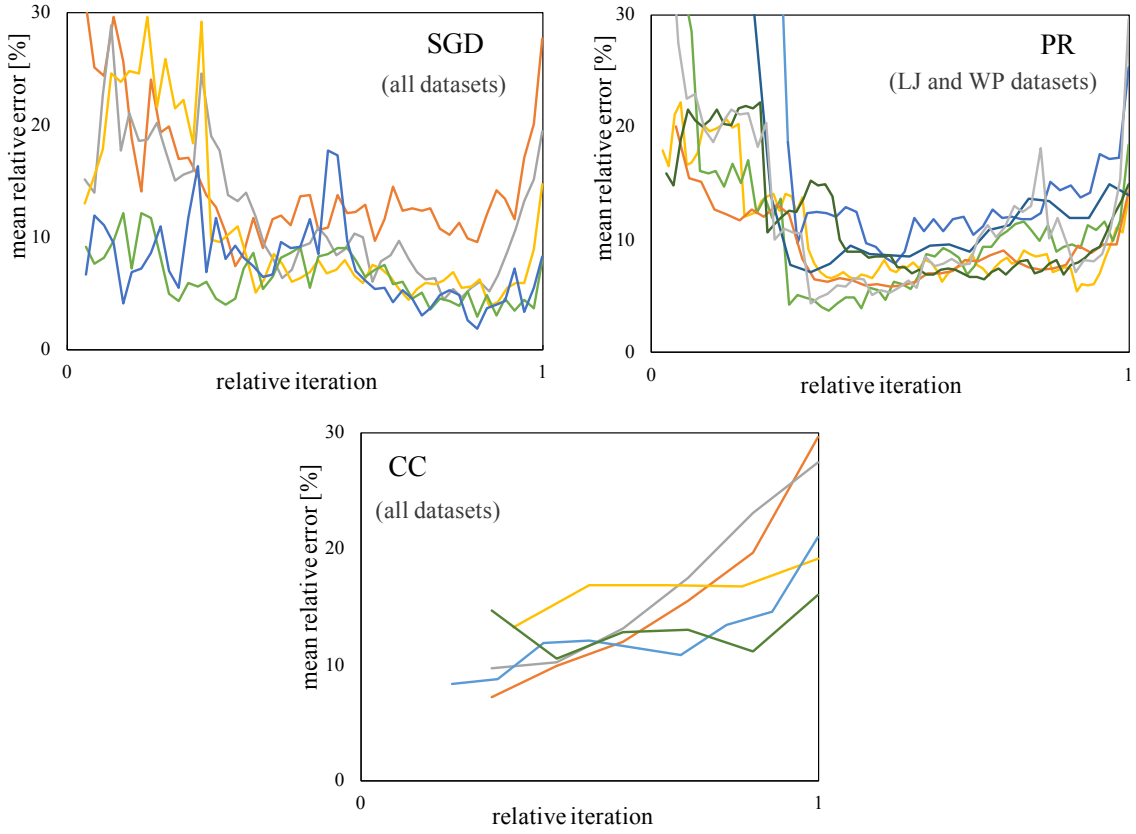


Figure 6.8: Iteration-wise estimation accuracy. © 2017 IEEE [132].

are available for a running execution, so the matching of similar job executions is initially inaccurate. As the job executions progress and more runtime statistics become available, the matching can better distinguish between job executions. It takes around 30% of the total iterations until the estimation reaches peak accuracy. The higher errors at the end of the executions are due to the smaller remaining runtimes, so even small absolute errors lead to higher relative errors.

7 Allocating Resources for Jobs With Runtime Targets

Contents

7.1 Stage-Wise Runtime Prediction 96

7.2 Selecting Resources for Runtime Targets 97

7.2.1 Resource Allocation Based on Predicted Runtimes 97

7.2.2 Selecting Resources on Job Submission 98

7.2.3 Adjusting Allocations at Runtime 99

7.2.4 Selecting Resources for Jobs with Insufficient Training Data . . . 101

7.3 Evaluation 101

7.3.1 Cluster Setup 101

7.3.2 Experiments 102

7.3.3 Results 103

This chapter presents Ellis, which we published in [135] (© 2017 IEEE). Ellis is a system that dynamically manages the resource allocations of distributed dataflow jobs according to users' runtime targets. For this, Ellis uses Bell, the runtime prediction system we presented in Chapter 5. Specifically, Ellis uses Bell to model the scale-out behavior of individual job stages to be able to predict the remaining runtime of a running job after each stage. Based on these predictions, Ellis selects resources that are predicted to meet a given runtime target. Ellis also dynamically adjusts resource allocations, if the predicted remaining runtime considerably exceeds the runtime target. Thereby, Ellis addresses the runtime variance exhibited by distributed dataflow jobs.

This chapter first explains how modeling the scale-out behavior of individual stages of distributed dataflows allows to predict the remaining runtime of a job. Second, we present in detail how Ellis initially selects resources based on predicted runtimes and adjusts resource allocations dynamically in-between stages. We also discuss the strategy Ellis applies for selecting resources when predictions are not possible due to the absence of sufficient training data. Third, the chapter presents an evaluation of Ellis using four different exemplary Spark jobs.

7.1 Stage-Wise Runtime Prediction

The key idea is to use a scale-out model not for the entire job, but for the individual stages of a distributed dataflow job. This allows us to predict the runtime of distinct parts of a job for particular resource allocations. Thus, we can assess the progress of a job towards a given runtime target by predicting the runtime of the remaining stages. At each synchronization barrier in-between subsequent stages, Ellis sums up the predicted runtimes of all the remaining stages under the current resource allocations and compares the result to the job's runtime target.

Distributed dataflow jobs can often be scaled efficiently at synchronization barriers in-between stages [120]. That is, subsequent stages can have different levels of parallelism, changing the scale-out of a job during its execution. At stage synchronization barriers, previously running operators finished and new ones are yet to start. With some distributed dataflow systems, jobs are always scheduled, deployed, and executed stage-by-stage. Yet, even if this is not the case, no running task state has to be migrated to new workers. Such task state includes state of UDFs and also internal state of operator implementations such as hash tables of Joins. However, when the level of parallelism is changed in-between subsequent stages, only the intermediate data resulting from one stage and to be consumed by the next stage has to be transferred to a new set of workers. Yet, at this point data is often shuffled anyway as stages usually end when further pipelining in-between subsequent tasks is no longer possible. This is the case for operators like Joins and group-wise aggregations that need all elements with the same key to be available at the same task instance. If the data is not already partitioned this way, the intermediate results need to be shuffled before the subsequent stage. Shuffling to a slightly larger or smaller set of nodes instead of to the exact same nodes is often possible with a maintainable overhead. Therefore, Ellis models the individual stages of jobs both to assess a job's progress at runtime and also to dynamically scale jobs, if necessary.

If the predicted runtime of all the remaining stages deviates considerably from the runtime target, so that the job would either finish later than required or could potentially do with less resources, Ellis searches for a different scale-out. In particular, Ellis searches for the smallest scale-out that is within the bounds of the specified minimal and maximal numbers of workers and also predicted to meet the runtime target.

As presented in Chapter 5, we use Bell for modeling the scale-out behavior of distributed dataflows. Bell uses a black-box approach for modeling, only taking the scale-outs and runtimes of previous executions of jobs as training data to fit functions for these datasets using regression. Moreover, Bell uses two models, a simple parameterized model and nonparametric regression, and automatically chooses between these models based on the prediction task und available training data.

Ellis uses Bell to model the scale-out behavior of each stage, as shown in Figure 7.1. That is, Bell is used to create a separate model for each stage, fitting the given training data, using either the parameterized model or interpolating the data points using non-parametric regression.

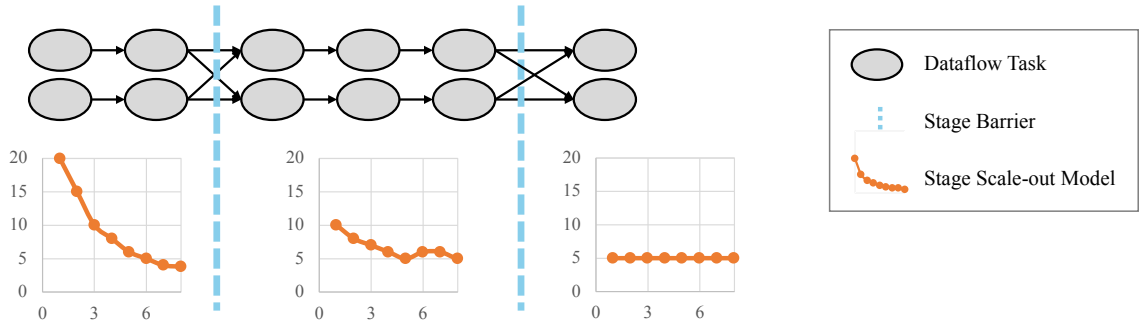


Figure 7.1: Modeling the scale-out behavior of the stages of a distributed dataflow job.

For selecting training data, Ellis uses Cutty. Cutty, which we presented in Chapter 6, selects similar previous executions of recurring jobs using multiple similarity measures. Some of these measures, such as information on the input datasets or the set of resources, are available offline. Other measures, including statistics on the runtime and resource utilization, are only available online. Thus, before a job is executed, we train scale-out models on previous executions that are similar in regard to measures that can be assessed offline, while at runtime we add runtime statistics as they become available, matching previous executions based on their actual runtime behavior. For assessing the similarity of the runtime behavior we compare stage runtimes, resource utilization of stages, and the convergence of jobs. That is, we continuously improve the scale-out models of jobs at runtime by selecting those previous executions as training data that actually behaved similar.

7.2 Selecting Resources for Runtime Targets

This section first presents the process of selecting resources for a job we use with Ellis. Then we explain in detail how resources are initially selected when a job is submitted and how resource allocation are adapted at runtime, if necessary.

7.2.1 Resource Allocation Based on Predicted Runtimes

Ellis uses a greedy approach for selecting resources based on the predicted runtimes of dataflow stages, conforming to three user-provided constraints. These three constraints are the runtime target as well as a minimum and a maximum scale-out. When no bounds for the scale-out are provided by the user, the system uses at least a single resource and maximally all available resources. Ellis searches for the smallest number of resources in-between the scale-out bounds predicted to provide a runtime below the runtime target. For predicting the runtime of each stage, Ellis uses Bell, as explained before.

Figure 7.2 depicts how Ellis selects resources. The curved blue line represents the overall predicted runtime for all stages of a distributed dataflow job, the horizontal orange

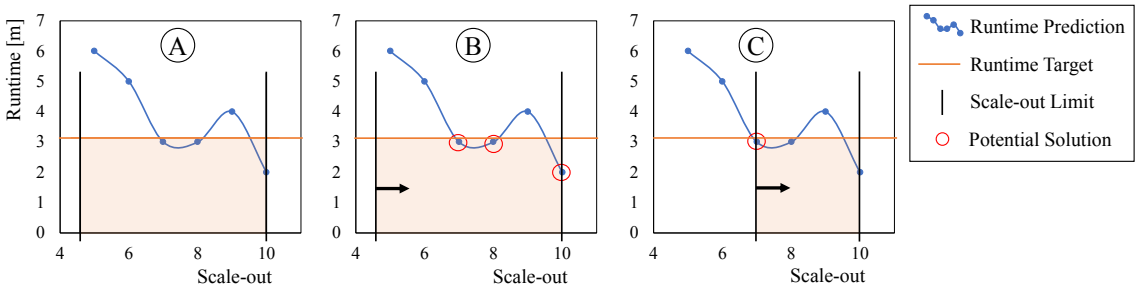


Figure 7.2: Process used by Ellis to select resources. © 2017 IEEE [135].

line shows the job's runtime target, and the vertical black lines stand for the scale-out limits provided by the user. Ellis sums up the predicted runtime of all stages as it searches for a single scale-out for all stages, since scaling a job at runtime is not without costs. The initial situation is shown in (A) of Figure 7.2. As shown in (B), Ellis computes the predicted overall runtime for the job for each scale-out, starting with the minimal scale-out and testing each discrete scale-out one by one. Ellis continues this process until the predicted overall job runtime is below the runtime target, as shown in (C). Basically, Ellis intercepts the three constraints with the runtime prediction functions to create a set of potential solutions. From this set, Ellis selects the smallest set of resources, assuming that more resources have higher costs than less resources.

At the beginning of a job's execution, Ellis uses this process to select resources for all stages of a job. At stage barriers, when the predicted runtime of the remaining stages for the current resource allocation significantly exceeds the runtime target, Ellis only considers the remaining stages.

7.2.2 Selecting Resources on Job Submission

On job submission, Ellis searches for the smallest scale-out within the bounds of a minimal and a maximal scale-out, for which the predicted runtime of all stages is below the runtime target. If there is none, Ellis selects the scale-out with the minimal predicted runtime for running all stages.

More formally, let $\mathbf{x}^{(\alpha)}$ be the scale-outs of stage α and $\mathbf{y}^{(\alpha)}$ the corresponding runtimes. We use regression to find a function f_α that fits this data. Algorithm 1 then shows the procedure we use to select the initial scale-out on job submission. The OnJobStart procedure takes the runtime constraint C, the minimal scale-out x_{min}, and the maximal scale-out x_{max} as input. The procedure first fits a regression function for the available training data. It then computes the set of scale-outs within the range of the provided minimal and maximal scale-outs predicted to meet the runtime target. If this solution set is not empty, the procedure uses the minimal scale-out in the solution set. If the solution set is instead empty, the procedure uses the scale-out with the minimal predicted runtime.

Algorithm 1 Procedure executed on job submission.

```
1: procedure OnJobStart(C, x_min, x_max)
2:     f ← Σ_α f_α
3:     X ← {x_min, ..., x_max}
4:     D ← {x ∈ X | f(x) < C}
5:     if D ≠ ∅ then
6:         x* ← min D
7:     else
8:         x* ← arg min_{x∈X} f(x)
9:     SetScaleOut(x*)
```

7.2.3 Adjusting Allocations at Runtime

In-between the stages of a running job, when the distributed dataflows are synchronized for specific operators like Joins and aggregations, which require all elements of the same key to be available, Ellis assesses the progress of jobs towards their runtime targets. At these synchronization barriers, the previous pipeline of tasks is finished and a new one starts. This subsequent stage of tasks can be started with a different level of parallelism, often with little overhead as elements need to be shuffled for the semantics of the following operator anyway.

Ellis assesses the current job's progress by predicting the runtime of the remaining stages for the current resource allocation, which is possible since we model the scale-out behavior of individual stages. The predicted remaining runtime is then compared to the actual remaining time, which is the runtime target minus the time the job has already been running. If the predicted remaining runtime considerably exceeds the actual remaining time, Ellis searches for the smallest scale-out for which the job is predicted to finish before the runtime target. If there is no such scale-out, Ellis selects the scale-out that is predicted to yield the lowest overall runtime for the remaining stages. If the predicted remaining runtime is considerably below the actual remaining time, Ellis searches for a scale-out that is lower than the current scale-out, in order to release surplus resources. If there is no such scale-out, the job's execution is continued with the current reservation.

More formally, let $\mathbf{x}^{(\alpha)}$ again be the scale-outs of stage α and $\mathbf{y}^{(\alpha)}$ the corresponding runtimes. We find a function f_α that fits this data using regression. We estimate the remaining runtime after stage α' given a scale-out x' by summing over the respective stage-wise predictions. That is, $f_{>\alpha'}(x') = \sum_{\alpha>\alpha'} f_\alpha(x')$. Algorithm 2 then shows the procedure that is used after each stage α, for estimating a job's progress towards its runtime target and if necessary adjusting the scale-out. The OnStageEnd procedure takes the following inputs: the runtime constraint C, the minimal scale-out $x_{\min}$, the maximal scale-out $x_{\max}$, the latest completed stage α', the current scale-out x, the time the job already ran t, the relative slack s_{tr}^+ and absolute slack s_{ta}^+ for triggering a dynamic scaling, and the relative slack s_{o} for establishing a new scale-out.

Algorithm 2 Procedure executed in-between all stages.

```
1:  procedure OnStageEnd(C, x_min, x_max, α', x, t, s_tr, s_ta, s_o)
2:      f_{>α'} ← Σ_{α>α'} f_α
3:      r ← f_{>α'}(x)                                  ▷ remaining runtime prediction
4:      if r > (C − t) · (1 + s_tr) + s_ta then
5:          X ← {x_min, ..., x_max}
6:          D ← {x' ∈ X | f_{>α'}(x') < s_o · (C − t)}
7:          if D ≠ ∅ then
8:              x* ← min D
9:          else
10:             x* ← arg min_{x'∈X} f_{>α'}(x')
11:         SetScaleOut(x*)
12:     else if r < (C − t) · (1 − s_tr) − s_ta then
13:         X ← {x_min, ..., x − 1}
14:         D ← {x' ∈ X | f_{>α'}(x') < s_o · (C − t)}
15:         if D ≠ ∅ then
16:             x* ← min D
17:             SetScaleOut(x*)
```

The procedure first fits a regression function for the available training data. It uses this function to predict the remaining runtime under the current scale-out x. Then there are two cases, for which we search for different scale-outs:

- If the predicted remaining runtime exceeds the remaining time to the runtime constraint by both the relative and absolute slack for triggering a dynamic scaling ($r > (C - t) \cdot (1 + s_{tr}) + s_{ta}$), the procedure searches for a new scale-out predicted to meet the runtime target with some slack for the overhead of the dynamic scaling (s_o). In this case, the procedure first computes the subset of the scale-outs within the bounds of the minimal and the maximal scale-out for which the overall runtime is predicted to be below the runtime constraint, reduced by the factor $0 < s_o \leq 1$ to compensate for the overheads of establishing a new scale-out. If this solution set D is not empty, the procedure selects the smallest scale-out in the solution set. If the solution set D is instead empty, the procedure selects the scale-out with the smallest predicted remaining runtime.
- If instead the predicted remaining runtime is below the remaining time to the runtime constraint by both slacks for triggering the scale-out ($r < (C - t) \cdot (1 - s_{tr}) - s_{ta}$), the procedure searches for a smaller scale-out, so resources are freed. It does so by computing the same solution set as before, only using the range between the minimal scale-out x_{min} and the current scale-out x as subset of valid scale-outs. If the solution set D is not empty, the smallest scale-out is selected. Otherwise, the procedure does not set a new scale-out.

If the overall runtime is predicted to be within the bounds of the two slacks s_{tr}^{+} and s_{ta}^{+} of the runtime constraint C, no dynamic scaling is triggered.

7.2.4 Selecting Resources for Jobs with Insufficient Training Data

Since we model the scale-out behavior of distributed dataflows using previous executions of recurring jobs, there might not always be sufficient training data to effectively train prediction models. In that case, we aim to select scale-outs that both meet a job's runtime target and explore the solution space, so subsequently not only more training data is available, but also training samples that cover the range of valid scale-outs between the user-provided limits. For this, we use a binary-search like approach as long as there are less than three selected samples for training a prediction model.

- If there are zero training samples for a job, use the maximal scale-out, assuming it has the best chances of meeting the job's runtime target.
- If there is one training sample, use the mean scale-out in-between the minimal and the maximal scale-out: $m \leftarrow \frac{x_{\text{min}}+x_{\text{max}}}{2}$.
- If there are two training samples, we consider the previous running times:
  - If the previous runtimes all where below the runtime target, we select a smaller scale-out than before and use the average in-between the user-provided minimum as well as the mean m: $\frac{x_{\text{min}}+m}{2}$.
  - If the previous runtimes were not all below the runtime target, we select a larger scale-out than in previous runs and use the average in-between the mean m and the user-provided maximum: $\frac{m+x_{\text{max}}}{2}$.

If there are more training samples, we train and use a scale-out model for runtime prediction and resource allocation as described in the previous sections.

7.3 Evaluation

We evaluated how well our approach addresses the variance in distributed dataflow performance by measuring how much less runtime target violations occur when Ellis assesses a job's progress and adjusts allocations at runtime in comparison to only using Ellis for initially selecting resources.

7.3.1 Cluster Setup

All experiments were done on 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. We used Linux (Kernel 3.10.0), Java 1.8.0, Hadoop 2.7.3, and Spark 2.1.0 for these experiments.

7.3.2 Experiments

To evaluate the effectiveness of our dynamic adjustments, we compared using Ellis only for the initial resource allocation to also using it for dynamic adjustments of the resource allocation in-between stages. In the following, we call using Ellis only initially *static* mode and using Ellis also in-between stages *dynamic* mode.

We started Ellis with an empty history database. Then, an initial ten runs were performed without dynamic adjustments for each of the benchmark jobs. The runtimes of these ten runs was used as training data for all experiments with each job. For comparison, we did 50 static and 50 dynamic runs. Overall we evaluated the effectiveness of our dynamic adjustments for four iterative Spark jobs and three different generated datasets. Each job used a scale-out range from 4 to 50 nodes.

7.3.2.1 Jobs

We used four Spark jobs as benchmarks, namely Multilayer Perceptron (MLP), Gradient Boosted Trees (GBT), SGD, and K-Means. Table 7.1 shows the jobs and the respective input parameters. The implementations of the jobs are based on Spark MLlib [52], a library for implementing distributed machine learning algorithms. All jobs were taken from the examples bundled with the library.

Table 7.1: 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

7.3.2.2 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[1].

Vandermonde A regression dataset with 20 features. The dataset was generated using our own generator by explicitly computing the Vandermonde matrix. For this, data

[1] `http://scikit-learn.org/`, accessed 2017-08-03.

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.

7.3.3 Results

The runtime of the 50 static and dynamic runs of each job is summarized in Figure 7.3. All jobs show less spikes above the target runtime when using Ellis for dynamic adjustments. In addition, if there are constraint violations, their magnitude is lower.

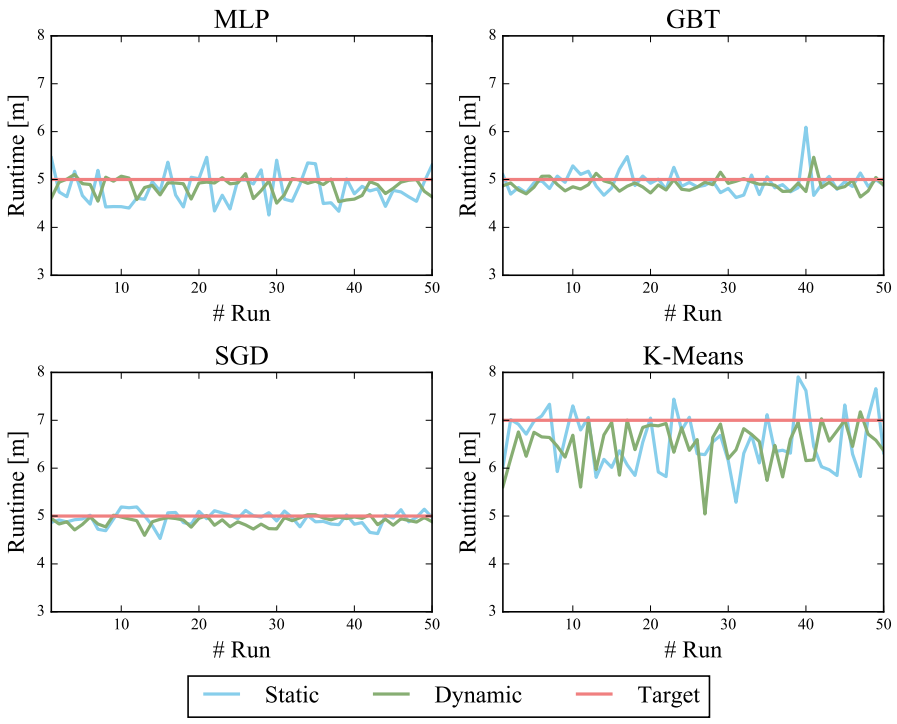


Figure 7.3: Comparison of the runtimes of 50 runs of using Ellis only to allocate resources initially (static) to also using Ellis for dynamic adjustments in-between stages (dynamic) and to the runtime target (red line). © 2017 IEEE [135].

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 $R = \sum_i y_i \cdot x_i$ 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 runtime constraint we introduce two metrics. The constraint violation count $CVC = \sum_{j \in \{j | Y_j > C\}} 1$ 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 CVS $= \sum_{j \in \{j|Y_j > C\}} (Y_j - C)$ summarizes the magnitude of the violations. Table 7.2 shows the constraint violation metrics for each of the four benchmark jobs. For every job both the amount and the intensity of constraint violations are reduced considerably.

Table 7.2: Constraint Violation Metrics.

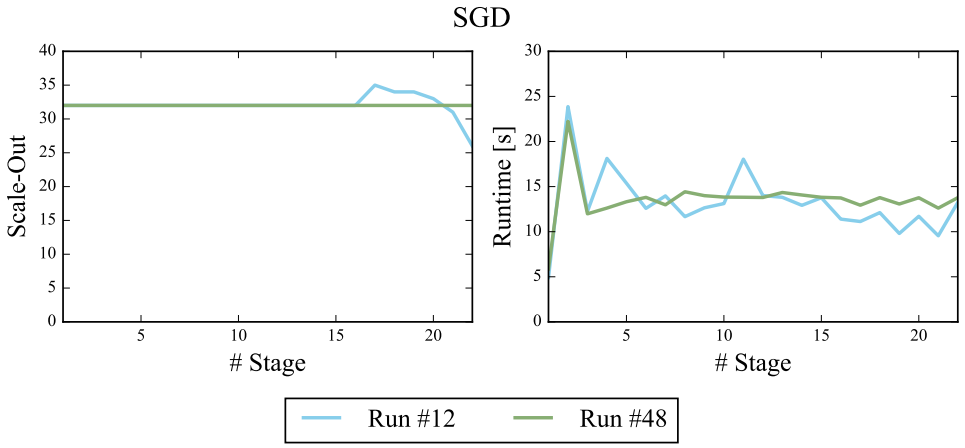
Job	CVC Static	CVC Dynamic	CVS Static [s]	CVS Dynamic [s]
MLP	13	9	198249	27099
GBT	15	7	192811	56732
SGD	20	5	98586	6631
K-Means	14	5	241925	13219

We also calculated the ratios of the metrics for the static and the dynamic runs, dividing the metrics of the dynamic runs by the metrics of the static runs. Table 7.3 shows these ratios for the metrics of our four benchmarks jobs. The ratios show that for every job the amount and the intensity of constraint violations is reduced, when Ellis estimates the progress towards the given runtime targets and based on these estimations adjusts resource allocations at runtime, if necessary. 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 static 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.

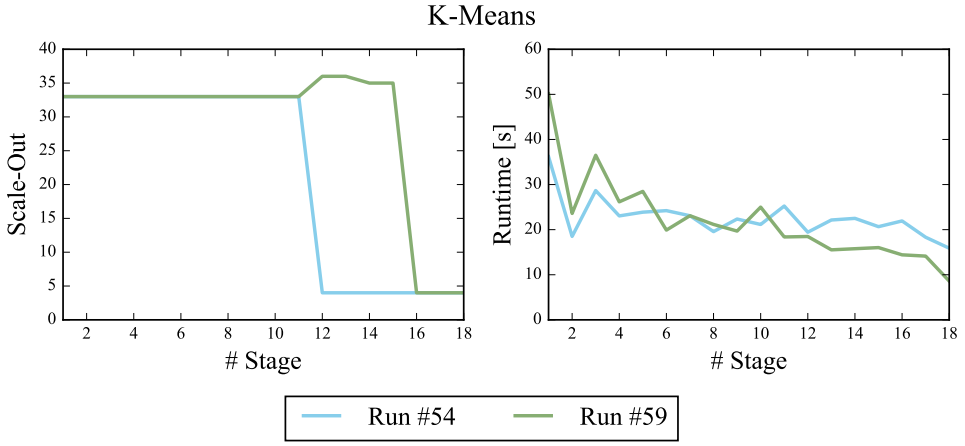
Table 7.3: Constraint Violations and Resource Usage Ratios.

Job	R Ratio	CVC Ratio	CVS Ratio
MLP	1.2704	0.6923	0.1367
GBT	0.9985	0.4667	0.2942
SGD	0.9475	0.2500	0.0673
K-Means	0.8003	0.3571	0.0546

To give an intuition of the dynamic scalings triggered by Ellis to meet a given runtime target, Figure 7.4 shows two exemplary runs of SGD and 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



(a) Two runs of the SGD job.



(b) Two runs of the K-Means job.

Figure 7.4: Exemplary runs of two jobs, showing the scale-out selected for each stage in the left chart and the corresponding runtime in the right chart. © 2017 IEEE [135].

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.

8 Conclusion

This thesis presented an approach and methods for automatically allocating minimal sets of resources for production batch jobs of distributed dataflow systems with runtime targets. The approach is based on the idea that repeatedly executed batch jobs present an opportunity to learn a job's scale-out behavior on a granularity that allows predicting the runtimes of individual job stages. Given such fine-grained scale-out models, it is not only possible to allocate an initial set of resources that is predicted to meet a given runtime target, but also to continuously monitor jobs by predicting the remaining runtime after each of a job's stages. If the predicted remaining runtime then deviates significantly from a job's runtime target, the scale-out models of the remaining stages can be used to dynamically select a new scale-out that is predicted to provide a job runtime within the bounds of the given runtime target.

The thesis made contributions in three areas. First, we presented two black-box regression models for capturing the scale-out behavior of distributed dataflow jobs, a simple parameterized model of distributed processing and a nonparametric model able to interpolate arbitrary scale-out behavior. We also showed how cross-validation can be used to select between these models automatically. Second, we presented methods for selecting those previous executions of recurring jobs as training data for estimation systems that allow accurate estimation. We presented multiple measures for assessing the similarity of distributed dataflow job executions and a method for training similarity matching parameters on the execution history of a job, automatically optimizing the similarity matching based on estimation accuracy. Third, we presented a method for continuously monitoring and dynamically scaling distributed dataflow jobs with runtime targets. Using a scale-out model for each of a distributed dataflow job's stages, the remaining runtime is predicted after each stage to select a new scale-out when jobs do not perform as initially predicted.

We implemented prototypes of the methods presented in this thesis and evaluated them using a number of exemplary Spark and Flink jobs, datasets of different sizes and domains, and a commodity cluster of 60 nodes. We showed that our methods for predicting the runtimes of distributed dataflow jobs using two models and automatic model selection provide a higher mean accuracy than each of the individual models. Furthermore, we demonstrated that our methods for selecting similar previous executions as a basis for estimating the runtimes of distributed dataflow jobs can be used to estimate remaining runtimes with a mean relative estimation error of 9.1-13.1%. Finally, we showed that dynamic resource allocation on the basis of stage-wise runtime prediction can help

significantly in meeting runtime targets. For our test workload the number of runtime constraint violations was reduced by 30.7-75.0% and the magnitude of runtime constraint violations by 70.6-94.5% through the dynamic adjustments of resource allocations, while using between 27% more and 20% less resources for the execution of the jobs. These results show that the presented approach and methods can be used for automatically selecting minimal sets of resources that meet runtime targets. That is, users no longer need to accurately estimate the runtimes of their distributed dataflow jobs, which is a difficult task, or considerably over-provision resources, which incurs needless costs. Moreover, due to our runtime monitoring and dynamic adjustments, it is no longer necessary to provision resources for the worst case of significantly variable performance.

Although this thesis already addresses central aspects of automatically allocating minimal sets of resources for distributed dataflow jobs with specific runtime targets, there are several interesting directions for further investigation. These directions can be derived from the key limitations of our current solution. The first key limitation is the dependence on the availability of previous executions of jobs, where we need on average ten similar previous runs, each with a different scale-out, for a mean relative prediction error of less than 5% for our benchmark jobs. Therefore, we would like to investigate methods that go beyond the simple binary-search approach we currently apply for exploring the impact of scale-outs when there are only few previous runs as examples. The second key limitation is the assumption of homogeneous cluster resources, currently requiring training and usage of separate models when there are multiple types of resources available. Thus, we would like to investigate how the impact of scale-up can be modeled more effectively, so our solution becomes more usable when there are heterogeneous resources. There is also a third key limitation, namely that we assume distributed dataflow systems with stages that can be provisioned separately. However, our black-box methods are arguably already generic and broadly applicable, while distributed dataflow systems are majorly popular for general-purpose data-parallel processing. Therefore, we are predominantly interested in addressing the first two limitations. Improvements in these directions would make our solution more useful for end-users that run their individual analytics jobs on resources temporarily leased from cloud providers, helping to further democratize access to massively parallel computation and analysis of large datasets, in line with the promises of cloud computing and distributed dataflow systems.

Nevertheless, the results of this thesis already show that the presented approach and methods for modeling the scale-out behavior, predicting runtimes, and allocating resources for distributed dataflow jobs at the granularity of stages can effectively alleviate users from the difficult task of estimating the runtime behavior of their jobs, while runtime targets are met to a high degree and without significant over-provisioning. Moreover, accurate runtime prediction can also be considerably useful for resource management systems, allowing to plan schedules ahead. Consequently, applying our results in practice promises increased efficiency and reduced costs for distributed data analytics on large shared clusters.

Bibliography

[1] B. Cheng, S. Longo, F. Cirillo, M. Bauer, and E. Kovacs. "Building a Big Data Platform for Smart Cities: Experience and Lessons from Santander". In: *2015 IEEE International Congress on Big Data*. BigDataCongress '15. IEEE, 2015.

[2] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Communications of the ACM* 51.1 (2008).

[3] A. S. Das, M. Datar, A. Garg, and S. Rajaram. "Google News Personalization: Scalable Online Collaborative Filtering". In: *Proceedings of the 16th International Conference on World Wide Web*. WWW '07. ACM, 2007.

[4] R. W. White, N. P. Tatonetti, N. H. Shah, R. B. Altman, and E. Horvitz. "Web-Scale Pharmacovigilance: Listening to Signals From the Crowd". In: *Journal of the American Medical Informatics Association* 20.3 (2013).

[5] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation*. OSDI '04. USENIX Association, 2004.

[6] 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*. HotCloud'10. USENIX Association, 2010.

[7] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. "Apache Flink: Stream and Batch Processing in a Single Engine". In: *IEEE Data Engineering Bulletin* 38.4 (2015).

[8] 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*. NSDI '11. USENIX Association, 2011.

[9] 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*. SOCC '13. ACM, 2013.

[10] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. "Re-optimizing Data Parallel Computing". In: *In Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*. NSDI '12. USENIX Association, 2012.

[11] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, Í. 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*. OSDI '16. USENIX Association, 2016.

[12] G. Mishne, J. Dalton, Z. Li, A. Sharma, and J. Lin. "Fast Data in the Era of Big Data: Twitter's Real-time Related Query Suggestion Architecture". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. ACM, 2013.

[13] 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*. SoCC '12. ACM, 2012.

[14] 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*. EuroSys '12. ACM, 2012.

[15] 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*. ASPLOS '14. ACM, 2014.

[16] 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*. ICAC '11. ACM, 2011.

[17] 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*. ICAC '12. ACM, 2012.

[18] 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*. SoCC '16. ACM, 2016.

[19] M. Hovestadt, O. Kao, A. Keller, and A. Streit. "Scheduling in HPC Resource Management Systems: Queuing vs. Planning". In: *Job Scheduling Strategies for Parallel Processing*. Ed. by D. Feitelson, L. Rudolph, and U. Schwiegelshohn. Springer, 2003.

[20] X. Zheng, Z. Zhou, X. Yang, Z. Lan, and J. Wang. "Exploring Plan-Based Scheduling for Large-Scale Computing Systems". In: *2016 IEEE International Conference on Cluster Computing*. CLUSTER '16. IEEE, 2016.

[21] A. M. Middleton. "Data-Intensive Technologies for Cloud Computing". In: *Handbook of Cloud Computing*. Ed. by B. Furht and A. Escalante. Springer, 2010.

[22] S. Babu and H. Herodotou. "Massively Parallel Databases and MapReduce Systems". In: *Foundations and Trends in Databases* 5.1 (2013).

[23] 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". In: *Proceedings of the VLDB Endowment* 1.2 (2008).

[24] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. "Making Sense of Performance in Data Analytics Frameworks". In: *12th USENIX Symposium on Networked Systems Design and Implementation*. NSDI '15. 2015.

[25] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. "Pregel: A System for Large-Scale Graph Processing". In: *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*. PODC '09. ACM, 2009.

[26] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI '12. USENIX Association, 2012.

[27] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud". In: *Proceedings of the VLDB Endowment* 5.8 (2012).

[28] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. "Scaling Distributed Machine Learning with the Parameter Server". In: *11th USENIX Symposium on Operating Systems Design and Implementation*. OSDI '14. USENIX Association, 2014.

[29] L. G. Valiant. "A Bridging Model for Parallel Computation". In: *Communications of the ACM* 33.8 (1990).

[30] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. "Spinning Fast Iterative Data Flows". In: *Proceedings of the VLDB Endowment* 5.11 (2012).

[31] G. Bell and J. Gray. "What's Next in High-performance Computing?" In: *Communications of the ACM* 45.2 (2002).

[32] D. W. Walker. "The Design of a Standard Message Passing Interface for Distributed Memory Concurrent Computers". In: *Parallel Computing* 20.4 (1994).

[33] V. S. Sunderam. "PVM: A Framework for Parallel Distributed Computing". In: *Concurrency: Practice and Experience* 2.4 (1990).

[34] L. Dagum and R. Menon. "OpenMP: an Industry Standard API for Shared-memory Programming". In: *IEEE Computational Science and Engineering* 5.1 (1998).

[35] A. Katal, M. Wazid, and R. H. Goudar. "Big Data: Issues, Challenges, Tools and Good Practices". In: *2013 Sixth International Conference on Contemporary Computing*. IC3 '13. IEEE, 2013.

[36] S. Jha, J. Qiu, A. Luckow, P. Mantha, and G. C. Fox. "A Tale of Two Data-Intensive Paradigms: Applications, Abstractions, and Architectures". In: *2014 IEEE International Congress on Big Data*. BigDataCongress '14. IEEE, 2014.

[37] G. Fox, J. Qiu, S. Jha, S. Ekanayake, and S. Kamburugamuve. "Big Data, Simulations and HPC Convergence". In: *Proceedings of the Workshop on Big Data Benchmarks 2015*. Ed. by T. Rabl, R. Nambiar, C. Baru, M. Bhandarkar, M. Poess, and S. Pyne. WBDB 2015. Springer, 2016.

[38] S.J. Lawson and M. Woodgate and R. Steijl and G.N. Barakos. "High Performance Computing for Challenging Problems in Computational Fluid Dynamics". In: *Progress in Aerospace Sciences* 52 (2012).

[39] J. Dongarra, T. Sterling, H. Simon, and E. Strohmaier. "High-Performance Computing: Clusters, Constellations, MPPs, and Future Directions". In: *Computing in Science Engineering* 7.2 (2005).

[40] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI '08. USENIX Association, 2008.

[41] J. Zhou, P. A. Larson, and R. Chaiken. "Incorporating Partitioning and Parallel Plans Into the SCOPE Optimizer". In: *2010 IEEE 26th International Conference on Data Engineering*. ICDE 2010. IEEE, 2010.

[42] F. Hueske, M. Peters, M. J. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. "Opening the Black Boxes in Data Flow Optimization". In: *Proceedings of the VLDB Endowment* 5.11 (2012).

[43] N. Bruno, S. Jain, and J. Zhou. "Continuous Cloud-Scale Query Optimization and Processing". In: *Proceedings of the VLDB Endowment* 6.11 (2013).

[44] A. Rheinländer, A. Heise, F. Hueske, U. Leser, and F. Naumann. "SOFA: An Extensible Logical Optimizer for UDF-heavy Data Flows". In: *Information Systems* 52.C (2015).

[45] 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*. SIGMOD '08. ACM, 2008.

[46] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. "Hive: A Warehousing Solution over a Map-Reduce Framework". In: *Proceedings of the VLDB Endowment* 2.2 (2009).

[47] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. "Shark: SQL and Rich Analytics at Scale". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. ACM, 2013.

[48] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. "Spark SQL: Relational Data Processing in Spark". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. ACM, 2015.

[49] 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*. OSDI '14. USENIX Association, 2014.

[50] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. "SystemML: Declarative Machine Learning on MapReduce". In: *2011 IEEE 27th International Conference on Data Engineering*. ICDE '11. IEEE, 2011.

[51] M. Boehm and M. W. Dusenberry and D. Eriksson and A. V. Evfimievski and F. M. Manshadi and N. Pansare and B. Reinwald and F. R. Reiss and P. Sen and A. C. Surve and S. Tatikonda. "SystemML: Declarative Machine Learning on Spark". In: *Proceedings of the VLDB Endowment* 9.13 (2016).

[52] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. "MLlib: Machine Learning in Apache Spark". In: *The Journal of Machine Learning Research* 17.1 (2016).

[53] L. Thamsen, B. Rabier, F. Schmidt, T. Renner, and O. Kao. "Scheduling Recurring Distributed Dataflow Jobs Based on Resource Utilization and Interference." In: *2017 IEEE International Congress on Big Data*. BigDataCongress '17. IEEE, 2017.

[54] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. "Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling". In: *Proceedings of the 5th European Conference on Computer Systems*. EuroSys '10. ACM, 2010.

[55] Z. Guo, G. Fox, and M. Zhou. "Investigation of Data Locality in MapReduce". In: *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. CCGrid 2012. IEEE, 2012.

[56] 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*. BigData 2016. IEEE, 2016.

[57] 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*. EuroSys '07. ACM, 2007.

[58] 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*. MTAGS '09. ACM, 2009.

[59] J. Zhou, N. Bruno, M.-C. Wu, P. A. Larson, R. Chaiken, and D. Shakib. "SCOPE: Parallel Databases Meet MapReduce". In: *The VLDB Journal* 21.5 (2012).

[60] 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". In: *The VLDB Journal* 23.6 (2014).

[61] 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*. SoCC '10. ACM, 2010.

[62] 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 2012. VLDB Endowment, 2012.

[63] 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. 2013.

[64] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. "Differential Dataflow". In: *Proceedings of the 6th Conference on Innovative Data Systems Research*. CIDR '13. CIDR 2013, 2013.

[65] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. "Apache Spark: A Unified Engine for Big Data Processing". In: *Commununications of the ACM* 59.11 (2016).

[66] 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*. NSDI '12. USENIX Association, 2012.

[67] 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*. SOSP '13. ACM, 2013.

[68] 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". In: *Proceedings of the VLDB Endowment* 8.12 (2015).

[69] B. Saha and H. Shah and S. Seth and G. Vijayaraghavan and A. Murthy and C. Curino. "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*. SIGMOD '15. ACM, 2015.

[70] S. Ghemawat, H. Gobioff, and S.-T. Leung. "The Google File System". In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. SOSP '03. ACM, 2003.

[71] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. "The Hadoop Distributed File System". In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*. MSST 2010. IEEE, 2010.

[72] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. "Ceph: A Scalable, High-Performance Distributed File System". In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI '06. USENIX Association, 2006.

[73] A. Davies and A. Orsaria. "Scale Out with GlusterFS". In: *Linux Journal* 2013.235 (2013).

[74] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. "Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks". In: *Proceedings of the ACM Symposium on Cloud Computing*. SOCC '14. ACM, 2014.

[75] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. "Scarlett: Coping with Skewed Content Popularity in Mapreduce Clusters". In: *Proceedings of the Sixth Conference on Computer Systems*. EuroSys '11. ACM, 2011.

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

[77] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. "Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI '14. USENIX Association, 2014.

[78] 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*. EuroSys '15. ACM, 2015.

[79] D. Battre, N. Frejnik, S. Goel, O. Kao, and D. Warneke. "Evaluation of Network Topology Inference in Opaque Compute Clouds through End-to-End Measurements". In: *2011 IEEE 4th International Conference on Cloud Computing*. CLOUD 2011. IEEE, 2011.

[80] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein. "GraphLab: A New Framework for Parallel Machine Learning". In: *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*. UAI '10. AUAI Press, 2010.

[81] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. "GraphX: A Resilient Distributed Graph System on Spark". In: *First International Workshop on Graph Data Management Experiences and Systems*. GRADES '13. ACM, 2013.

[82] S. Englert, J. Gray, T. Kocher, and P. Shah. "A Benchmark of NonStop SQL Release 2 Demonstrating Near-linear Speedup and Scaleup on Large Databases". In: *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '90. ACM, 1990.

[83] H. Zeller. "Parallel Query Execution in NonStop SQL". In: *Compcon Spring '90. Intellectual Leverage. Digest of Papers. Thirty-Fifth IEEE Computer Society International Conference.* IEEE, 1990.

[84] F. M. Waas. "Beyond Conventional Data Warehousing: Massively Parallel Data Processing with Greenplum Database". In: *Business Intelligence for the Real-Time Enterprise*. Ed. by M. Castellanos, U. Dayal, and T. Sellis. Springer, 2009.

[85] E. Friedman, P. Pawlowski, and J. Cieslewicz. "SQL/MapReduce: A Practical Approach to Self-describing, Polymorphic, and Parallelizable User-defined Functions". In: *Proceedings of the VLDB Endowment* 2.2 (2009).

[86] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y.-S. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. Tsotras, R. Vernica, J. Wen, and T. Westmann. "AsterixDB: A Scalable, Open Source BDMS". In: *Proceedings of the VLDB Endowment* 7.14 (2014).

[87] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. "Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing". In: *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*. ICDE '11. IEEE, 2011.

[88] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. "Hive - a Petabyte Scale Data Warehouse Using Hadoop". In: *2010 IEEE 26th International Conference on Data Engineering*. ICDE 2010. IEEE, 2010.

[89] A. D. Popescu, A. Balmin, V. Ercegovac, and A. Ailamaki. "PREDIcT: Towards Predicting the Runtime of Large Scale Iterative Analytics". In: *Proceedings of the VLDB Endowment* 6.14 (2013).

[90] M. Höger and O. Kao. "Progress Estimation in Parallel Data Processing Systems". In: *Proceedings of the IEEE International Conference on Cloud and Big Data Computing*. CBDCom 2016. IEEE, 2016.

[91] 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*. Middleware '11. Springer, 2011.

[92] 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*. SOCC '11. ACM, 2011.

[93] 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 5th Conference on Innovative Data Systems Research*. CIDR '11. CIDR 2011, 2011.

[94] 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*. SIGMETRICS '07. ACM, 2007.

[95] 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*. SoCC '12. ACM, 2012.

[96] S. Sidhanta, W. Golab, and S. Mukhopadhyay. "OptEx: A Deadline-Aware Cost Optimization Model for Spark". In: *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. CCGrid 2016. IEEE, 2016.

[97] 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*. NSDI '16. USENIX Association, 2016.

[98] F. Pukelsheim. *Optimal Design of Experiments*. Vol. 50. SIAM, 1993.

[99] S. Dimopoulos and C. Krintz and R. Wolsk. "Justice: A Deadline-aware, Fair-share Resource Allocator for Implementing Multi-analytics". In: *2017 IEEE International Conference on Cluster Computing*. CLUSTER '17. IEEE, 2017.

[100] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. "CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics". In: *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*. NSDI '17. USENIX Association, 2017.

[101] N. Bruno, S. Agarwal, S. Kandula, B. Shi, M.-C. Wu, and J. Zhou. "Recurring Job Optimization in SCOPE". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12. ACM, 2012.

[102] S. Gupta, C. Fritz, B. Price, R. Hoover, J. Dekleer, and C. Witteveen. "ThroughputScheduler: Learning to Schedule on Heterogeneous Hadoop Clusters". In: *Proceedings of the 10th International Conference on Autonomic Computing*. ICAC '13. USENIX Association, 2013.

[103] C. Delimitrou and C. Kozyrakis. "Paragon: QoS-aware Scheduling for Heterogeneous Datacenters". In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. ACM, 2013.

[104] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. "Storm@Twitter". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. ACM, 2014.

[105] L. Aniello, R. Baldoni, and L. Querzoni. "Adaptive Online Scheduling in Storm". In: *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*. DEBS '13. ACM, 2013.

[106] Z. Niu, S. Tang, and B. He. "Gemini: An Adaptive Performance-Fairness Scheduler for Data-Intensive Cluster Computing". In: *2015 IEEE 7th International Conference on Cloud Computing Technology and Science*. CloudCom 2015. IEEE, 2015.

[107] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI '11. USENIX Association, 2011.

[108] B. Lohrmann, D. Warneke, and O. Kao. "Nephele Streaming: Stream Processing Under QoS Constraints at Scale". In: *Cluster Computing* 17.1 (2014).

[109] B. Lohrmann, P. Janacik, and O. Kao. "Elastic Stream Processing with Latency Guarantees". In: *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems*. ICDCS '15. 2015.

[110] 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 '16. IEEE, 2016.

[111] C. L. Abad, Y. Lu, and R. H. Campbell. "DARE: Adaptive Data Replication for Efficient Cluster Scheduling". In: *2011 IEEE International Conference on Cluster Computing*. CLUSTER '11. IEEE, 2011.

[112] 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*. HotOS '09. USENIX Association, 2009.

[113] 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*. MASCOTS '09. IEEE, 2009.

[114] K. V. Vishwanath and N. Nagappan. "Characterizing Cloud Computing Hardware Reliability". In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. ACM, 2010.

[115] 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*. NOMS 2010. IEEE, 2010.

[116] 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*. OSDI '10. USENIX Association, 2010.

[117] 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*. NSDI '13. USENIX Association, 2013.

[118] 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*. SOCC '14. ACM, 2014.

[119] T. Mendt. "Cardinality Estimation in Shared-Nothing Parallel Data Flows ". MA thesis. Technische Universität Berlin, 2015.

[120] L. Thamsen, T. Renner, and O. Kao. "Continuously Improving the Resource Utilization of Iterative Parallel Dataflows". In: *2016 IEEE 36th International Conference on Distributed Computing Systems Workshops*. ICDCSW 2016. IEEE, 2016.

[121] P. A. Boncz, M. Zukowski, and N. Nes. "MonetDB/X100: Hyper-Pipelining Query Execution". In: *Proceedings of the Second Biennal Conference on Innovative Data Systems Research*. Vol. 5. CIDR '05. CIDR 2005, 2005.

[122] 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, 2016.

[123] S. Sahni and V. Thanvantri. "Performance Metrics: Keeping the Focus on Runtime". In: *IEEE Parallel Distributed Technology: Systems Applications* 4.1 (1996).

[124] W. D. Hillis and G. L. Steele. "Data Parallel Algorithms". In: *Communications of the ACM* 29.12 (1986).

[125] J. Hartman and D. Sanders. "Data Parallel Programming: A Transition From Serial to Parallel Computing". In: *Proceedings of the Twenty-Fourth SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '93. ACM, 1993.

[126] 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*. CBDCom 2016. IEEE, 2016.

[127] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. "Exploiting Bounded Staleness to Speed Up Big Data Analytics". In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX ATC'14. USENIX Association, 2014.

[128] W. S. Cleveland. "Robust Locally Weighted Regression and Smoothing Scatterplots". In: *Journal of the American Statistical Association* 74.368 (1979).

[129] 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*. Ed. by T. Rabl, N. Raghunath, M. Poess, M. Bhandarkar, H.-A. Jacobsen, and C. Baru. Springer, 2014.

[130] D. M. Blei, A. Y. Ng, and M. I. Jordan. "Latent Dirichlet Allocation". In: *The Journal of Machine Learning Research* 3 (2003).

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

[132] J. Koch, L. Thamsen, F. Schmidt, and O. Kao. "SMiPE: Estimating the Progress of Recurring Iterative Distributed Dataflows". In: *The 18th International Conference on Parallel and Distributed Computing, Applications and Technologies*. PDCAT '17. IEEE, 2017.

[133] M. J. Powell. "The BOBYQA Algorithm for Bound Constrained Optimization without Derivatives". In: *Cambridge NA Report NA2009/06, University of Cambridge, Cambridge* (2009).

[134] J. Kunegis. "KONECT: The Koblenz Network Collection". In: *Proceedings of the 22Nd International Conference on World Wide Web*. WWW '13. ACM, 2013.

[135] L. Thamsen, I. Verbitskiy, J. Beilharz, T. Renner, A. Polze, and O. Kao. "Ellis: Dynamically Scaling Distributed Dataflows to Meet Runtime Targets". In: *Proceedings of the 2017 IEEE 9th International Conference on Cloud Computing Technology and Science*. CloudCom 2017. IEEE, 2017.