Supporting Long-Running Applications in Shared Compute Clusters

Panagiotis Garefalakis

Department of Computing
Imperial College London

This dissertation is submitted for the degree of
Doctor of Philosophy

June 2020

Abstract

Clusters that handle data-intensive workloads at a data-centre scale have become commonplace. In this setting, clusters are typically shared across several users and applications, and consolidate workloads that range from traditional analytics applications to critical services, stream processing, machine learning, and complex data processing applications. This constitutes a growing class of applications called *long-running*, consisting of containers that are used for durations ranging from hours to months. Even though long-running applications occupy a significant amount of resources in shared compute clusters today, there is currently rudimentary systems support that not only hinders application performance and resilience but also decreases cluster resource efficiency i.e., the effective utility extracted from cluster resources.

In this thesis, we describe two main areas that lack support for long-running applications in traditional system designs. First, the way modern data processing frameworks execute complex computation tasks as part of shared long-running containers is broken. Even though these frameworks enable users to combine different types of computation as part of the same application using high-level programming interfaces, they ignore their diverse latency and throughput requirements during execution. Second, existing systems that allocate resources for long-running applications in the form of containers lack an expressive interface that can capture their placement requirements in shared compute clusters. Such placements can be expressed by means of complex constraints and are critical for both the performance and resilience of long-running applications.

To target the aforementioned mismatch, we introduce our contribution of *unified dataflows with placement constraints*, an abstraction that enables the *efficient* execution and the *effective* placement of long-running applications in shared compute clusters. Our abstraction is realised as part of a novel execution framework and a new cluster manager following a two-scheduler design.

We first present NEPTUNE, a data processing framework that captures application execution requirements and employs coroutines as a lightweight mechanism for suspending and resuming tasks within long-running containers with sub-millisecond latency. NEPTUNE introduces scheduling policies that can dynamically prioritise tasks of heterogeneous jobs respecting their requirements in an *efficient* manner while achieving high resource utilisation.

We then introduce MEDEA, a cluster scheduler that enables the *effective* placement of containers in shared compute clusters. It follows an optimisation-based scheduling approach that is used for applications with sustained impact on cluster resources. MEDEA introduces powerful placement constraints with formal semantics that developers can use to capture container interactions within and across applications. MEDEA has been deployed in large production clusters and has been open-sourced as part of Apache Hadoop 3.1 release.

Acknowledgements

Foremost, I want to express my sincere gratitude to my supervisor Peter Pietzuch for his continuous support over the past five years. Peter's passion for research, inspiring counsel, and high standards have been crucial in shaping the core of this thesis.

I want to specially thank Konstantinos Karanasos, my industrial mentor, who guided and helped me in countless ways. Our research and non-research conversations challenged and motivated me while ultimately helped me grow both as a person and as a researcher. I am also grateful to my examiners, Seif Haridi and Giuliano Casale, for their insightful comments and suggestions that helped me improve this document after a lively discussion.

Next, I would like to thank Imperial College London and the High-Performance and Embedded Distributed Systems (HiPEDS) Centre for Doctoral Training (CDT) programme for supporting my studies. I also want to thank the fantastic members of Large-Scale Distributed Systems (LSDS) group at Imperial with whom I shared office all these years. Special thanks to Alexandros Koliousis, George Theodorakis, Jana Giceva, Holger Pirk, Christian Priebe, Joshua Lind, and Raul Castro Fernandez for all work done together, the technical discussions, and the off-topic conversations that made the past years such a pleasant experience.

The ideas for both the systems presented in this dissertation are heavily influenced by my internships at Microsoft back in 2016 and 2017 where I worked with the CISL group. Konstantinos Karanasos, Arun Suresh, Sriram Rao, Virajith Jalaparti, Avrilia Floratou, Ashvin Agrawal, Carlo Curino, Subru Krishnan all helped me with their invaluable advice and their technical expertise. Working with distinguished researchers from the industry helped me realise several opportunities for innovative systems targeting large-scale shared compute clusters.

Thanks to Ioanna, Christos, Salvatore, and the rest of my wonderful friends around the world for being by my side in the good days and the bad days. I will always be grateful to Nayia for making everything easier. Even though we met halfway through this journey, it wouldn't be the same without her love and support.

Finally and above all, I am profoundly grateful to my family who supported me through this difficult journey. I thank my wonderful parents Georgia and Giannis, and my sister Eirini for their endless love, unconditional support, and for their encouragement to pursue my dreams.

Declaration

This thesis presents my work in the Department of Computing at Imperial College London for the period between October 2015 and October 2019. Parts of the work described were done in collaboration with other researchers:

- **Chapter 4**: I proposed, implemented and evaluated the design of NEPTUNE execution framework. The metric collection and windowing mechanism was contributed to NEPTUNE by Christo Lolov as part of his MEng project at Imperial College London during the academic year 2018-2019.
- **Chapter 5**: I also led the development and evaluation of MEDEA cluster scheduler based on several production use-cases and requirements. The design of the system was result of many fruitful discussions with Arun Suresh, Konstantinos Karanasos and the open-source community as part of my collaboration with Microsoft's Cloud and Information Services Lab where I interned for the summers of 2016 and 2017.

I declare that the work presented in this thesis is my own, except where declared above.

Panagiotis Garefalakis
June 2020

©

The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution 4.0 International Licence (CC BY).

Under this licence, you may copy and redistribute the material in any medium or format for both commercial and non-commercial purposes. You may also create and distribute modified versions of the work. This on the condition that you credit the author.

When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes.

Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

Table of contents

1 Introduction **1**

1.1 The evolution of Big Data processing . . . 2

1.2 Research motivation . . . 4

1.3 Contributions . . . 5

1.3.1 Unified dataflows . . . 5

1.3.2 Placement constraints . . . 6

1.4 Related publications . . . 6

1.5 Outline . . . 7

2 Background **9**

2.1 Cluster workloads . . . 10

2.1.1 Workload types . . . 12

2.1.2 Workload scheduling . . . 13

2.2 Data processing systems . . . 15

2.2.1 Dataflow model . . . 17

2.2.2 Programming interface . . . 18

2.2.3 Execution model . . . 21

2.2.4 Evolution of the execution model . . . 22

2.2.5 Scheduling hybrid dataflow applications . . . 24

2.2.6 Data processing summary . . . 27

2.3 Cluster scheduling . . . 27

2.3.1 Long-running application performance 28
2.3.2 Long-running application resilience 31
2.3.3 Global cluster objectives 33
2.3.4 Cluster scheduling requirements 33
2.3.5 Cluster scheduler architectures 36
2.3.6 Cluster scheduling summary 40
2.4 Chapter summary 40

3 Unified Dataflows with Placement Constraints 41
3.1 Overview 42
3.2 Unified dataflows 43
3.2.1 Execution model 44
3.2.2 API 45
3.3 Placement constraints 46
3.3.1 Expressing constraints 47
3.3.2 Container tags 48
3.3.3 Node groups 49
3.4 Discussion 49
3.5 Summary 50

4 Neptune: An Execution Framework for Suspendable Tasks 53
4.1 Overview 55
4.2 Unified dataflow applications 56
4.2.1 Expressing stream/batch applications 56
4.2.2 Scheduling stream/batch applications 57
4.2.3 Executing stream/batch tasks 58
4.3 Suspendable tasks 59
4.4 Locality- and memory-driven scheduling 62

4.4.1 Metric windowing 63
4.4.2 Scheduling policy 64
4.5 Implementation 66
4.6 Evaluation 69
4.6.1 Experimental set-up 70
4.6.2 Application performance 72
4.6.3 Varying resource demands 76
4.6.4 Task suspension 79
4.7 Limitations and discussion 81
4.8 Summary 83

5 Medea: A Cluster Scheduler with Rich Placement Constraints 85
5.1 Overview 87
5.2 Expressing placement constraints 89
5.2.1 Discussion 92
5.3 Scheduling long-running applications 92
5.3.1 Overview 92
5.3.2 ILP-based scheduling 93
5.3.3 Heuristic-based scheduling 95
5.4 Implementation 96
5.5 Evaluation 99
5.5.1 Experimental set-up 99
5.5.2 Application performance 102
5.5.3 Application resilience 104
5.5.4 Global cluster objectives 105
5.5.5 Scheduling latency 108
5.6 Limitations and discussion 111
5.7 Summary 112

6 Conclusions and future work **115**

6.1 Thesis summary 117

6.2 Future work 119

References **121**

List of figures

2.1 Machines used for long-running applications (LRAs) in six analytics clusters at Microsoft . . . 11

2.2 Container scheduling event life-cycle over time describing states at the bottom and metrics at the top . . . 13

2.3 Task scheduling event life-cycle over time describing states at the bottom and metrics at the top . . . 14

2.4 Examples of different dataflow models consisting of tasks shown as circles and flows of data represented as edges . . . 17

2.5 Execution overview of a dataflow application (Blue circles represent operators while white ones indicate different tasks.) . . . 21

2.6 Evolution of stream and batch frameworks . . . 22

2.7 Resource usage over time for the malicious behaviour detection application under different scheduling policies . . . 24

2.8 Queueing and end-to-end latencies observed for the detection application implemented in Spark over varying sizes of batch jobs . . . 26

2.9 Memcached lookup latency with node affinity constraints . . . 29

2.10 HBase throughput with node anti-affinity constraints . . . 30

2.11 Impact of cardinality constraints on application performance . . . 31

2.12 Unavailable machines in a Microsoft cluster (With total we represent the unavailability percentage over all machines, while SU1–SU4 are unavailability percentages over specific service units, i.e., logical node groups.) . . . 32

2.13 Overview of cluster scheduler architectures (Circles represent tasks/containers while colours correspond to different task types e.g., LRA, batch; white boxes represent cluster machines.) 37

3.1 Overview of the unified dataflow model with placement constraints 41

3.2 Unified dataflow application example for a real-time malicious behaviour detection service (A batch job trains a model using historical data; a stream job performs inference to detect malicious behaviour in real-time.) 44

3.3 Container placement scenario, with a unified application (`DF`), HBase (`HB`), a key/value store (`KV`) and streaming job (`S1`) 46

4.1 NEPTUNE architecture (Executors maintain separate queues for running and suspended tasks, and periodically send heartbeats to the central scheduler. The scheduler assigns tasks to executors and decides to suspend already-running ones according to its scheduling policy.) 57

4.2 NEPTUNE coroutine mechanism (Caller initiates a coroutine using the `call` method. Coroutine is using a separate stack to store state and variables and can `yield` control back to the caller during execution.) 59

4.3 NEPTUNE coroutine composition (The executor runs a coroutine `ShuffleMapTask`, which instantiates a coroutine `SortShuffleWriter`. Both can yield to the executor when the `TaskContext` is paused.) 61

4.4 Task runtime metric captured over time (The executor reports inconsistent values between the first and second heartbeats as task runtime depends on deserialization time.) 64

4.5 NEPTUNE integration in Spark (As part of the master node, the NEPTUNE scheduler assigns tasks to worker nodes. On each worker node, NEPTUNE maintains a queue of running and suspended tasks.) 67

4.6 Yahoo Streaming Benchmark (streaming + batch) streaming query latency 73

4.7 Yahoo Streaming Benchmark (streaming + batch) throughput 74

4.8 LDA NYTimes dataset (training + inference) inference latency 75

4.9 LDA NYTimes dataset (training + inference) throughput 76

4.10 Performance impact of varying demands 77

4.11 Memory impact of varying job demands 78

4.12 Latency impact of increasing memory use 79
4.13 NEPTUNE task suspension mechanism latency breakdown using TPC-H queries . . . 80
4.14 Latency of task suspension mechanisms 81
4.15 Available machine memory in a Microsoft production cluster (The cluster is comprised of tens of thousands of machines with each group corresponding to different hardware configurations. Data is over four days.) 82

5.1 MEDEA scheduler design 88
5.2 LRA placement scenario, with two key/value store instances (KV1, KV2), a streaming application (S1) and Spark job instances (Spark) 90
5.3 ILP formulation 94
5.4 MEDEA architecture 97
5.5 Application performance (lower is better) 103
5.6 Application resilience over 15 days 105
5.7 Load balance with varying LRA utilisation 106
5.8 Constraint violations 107
5.9 Varying cluster size 109
5.10 Two-scheduler benefit 110
5.11 Task-based scheduling latency 111

List of tables

2.1 Storm/Memcached latencies (Latencies include Memcached lookup and total processing.) 29

2.2 Support for LRA requirements R1–R4 in existing schedulers (✧ indicates implicit support for constraints through static machine attributes and not by declaring explicit dependencies between containers; ✻ indicates a partially supported feature.) 35

4.1 Summary of all the executor metrics provided in NEPTUNE. Some of the metrics provide a cumulative and others provide a snapshot value. 68

5.1 Notation used in ILP formulation (constants appear above the dashed line, variables below) 93

5.2 YCSB workload properties (A–F) [26] 100

Chapter 1

Introduction

The internet today represents an enormous space where large volumes of data are captured and analysed. Datasets are tightly coupled with our everyday habits including our mobile, social, finance and digital life. The connection between humans and data is so strong that scientists claim we are now living in the era of big data [14, 92]. The analysis of such data is ranging from building indexes to help users find on-line content through search engines, predicting earthquakes and financial trends, suggesting user content through ad-targeting platforms and recommender systems, to genome research trying to cure deadly diseases [115]. To a great extent, analysing big data efficiently enables science and the world to access new undiscovered insights and make progress faster.

The explosive growth of big data and information processing has created an unprecedented need for storing huge data volumes. This has led to the development of several scalable storage systems [5, 155] and a new breed of databases [22, 34, 50, 62], often referred to as NoSQL data stores. These data stores usually explore weaker consistency models [11, 42, 85] that trade-off strong consistency for increased performance and scalability; while more recent data store designs also support transactions over such weaker consistency models [29].

Even though scalable data storage can be challenging, it is not the biggest issue anymore. Large internet companies such as Microsoft, Amazon, Facebook, and Google already store large amounts of information on a daily basis and are willing to store even more information that can potentially increase their value in the future. For example, Facebook was able to store data from its on-line user activity directly to its back-end storage service back in 2005 [43]. This shifted the focus of organisations from storage to scalable data processing technologies that allow their developers to process large volumes of data in an efficient manner.

1.1 The evolution of Big Data processing

Over the past decade, we have witnessed several distributed data processing frameworks [5, 7, 20, 32, 45, 64, 66, 114, 127, 150] that have enabled users to process vast amounts of data on clusters of machines [18, 31, 82, 135, 137]. In order to scale, these systems rely on shared-nothing architectures comprised of clusters of commodity machines [126]. The aggregate computation power of those machines enables distributed processing frameworks to handle increasing volumes of data in a cost-effective manner. This *scale out* approach is more attractive than using more powerful machine hardware to speed up computation, an approach also known as scale up, which can be a non-viable costly solution.

Early implementations of distributed data processing frameworks were targeting specific types of computation and use cases. For example, batch processing frameworks such as Apache Hadoop, Spark and Hive [5, 66, 107, 150] mainly focused on high processing throughput over large datasets, while stream processing frameworks such as Apache Storm, Flink and Heron [20, 45, 64, 127], focused more on providing low processing latency. In practice, however, this variety of frameworks posed a limitation both for the users and cluster operators. Users had to copy data across frameworks when building more complex workflows involving multiple systems, and also had to face unclear trade-offs picking the right framework for the right job. At the same time, cluster operators had to support and manage multiple systems with dependencies across compute clusters.

Acknowledging the limitation of using, deploying and maintaining multiple systems, distributed processing frameworks evolved to support and unify different types of data processing such as streaming, batch, and machine learning in a single execution engine [7, 13, 20, 41, 150]. This evolution is in line with the current momentum towards unified analytics in general, where modern systems try to combine large-scale data processing with state-of-the-art machine learning and artificial intelligence in an efficient manner [81, 93, 123].

Applications in these frameworks are realised as generic dataflow graphs, where nodes perform computation and edges represent the flow of data across the nodes. Dataflow computation is performed in parallel tasks that usually run as part of *long-running containers* in the cluster. These systems have also exposed unified programming interfaces for the users to express their computation through a single API. Users can now use unified APIs to express different types of computation such as batch and streaming. Examples include Spark's Structured Streaming API [9], Flink's Table API [49], and the external API layer of Apache Beam [13].

As the next step in this unification, users have begun to combine diverse computation jobs within more complex applications. For example, a real-time malicious behaviour detection service [88] can now combine a machine learning job training a model using historical data, and a low-latency stream job performing inference on real-time data, as part of a single unified application [48, 119]. Such unified application designs simplify application development and operationalisation: (i) users can

reuse code with consistent semantics e.g., a streaming and a batch version of a job running over the same data will produce the same results; (ii) avoid the complexity of interacting with external systems e.g., state sharing across jobs is usually provided by the framework; (iii) and have a single application to monitor and tune. However, despite the unified application support from the API point of view by some distributed processing frameworks, there is no support on the scheduling or execution side. These complex applications still follow a typical dataflow abstraction, ignoring specific job execution requirements.

In the era of unified analytics, a common challenge of modern data processing frameworks is capturing and satisfying the diverse job requirements of complex unified applications. These applications can now execute computation with different demands (i.e., low latency or high throughput processing) within the same shared long-running containers.

In addition to that, data processing frameworks require large clusters of commodity machines to be able to scale and provide results in a timely manner. Modern organisations operate such large data-centres that are typically shared across users and applications. In this environment, sophisticated clusters managers such as Hydra [31], YARN [135], Mesos [65], and Borg [137] carry out the on-demand allocation of resources such as memory, disk and CPU to applications in the form of containers. Cluster managers have thus allowed data processing frameworks to scale to hundreds of commodity machines while at the same time they have enabled the consolidation of a plethora of different workloads onto shared compute clusters as they are application-agnostic.

Workload consolidation is important to increase cluster resource efficiency, i.e., to extract as much computation as possible from the underlying resources available. However, this has led to a diverse set of applications running in shared compute clusters today. Apart from batch analytics jobs [5, 32, 129, 154] that were traditionally running in those clusters, workloads now include unified applications [48, 119], stream processing [64, 127], iterative computations [1], data-intensive interactive jobs [150], and latency-sensitive online applications [62, 94]. For their deployment, unlike batch jobs that typically use short-lived containers running for durations in the order of seconds, most of the modern applications benefit from long-running containers. These containers are allocated and used for durations ranging from hours to months, thus avoiding repeated container initialisation costs and reducing container scheduling load. We refer to this new growing class of applications that leverage long-lived containers as *long-running applications* or *LRAs*.

As LRAs use containers that run for longer periods of time, they have a sustained impact on cluster resources. Selecting the most suitable machines for their container placement is thus important not only for application performance and availability, but also for cluster efficiency. For instance, load imbalances in a cluster can potentially block future applications with specific machine requirements from running. In terms of LRA performance, containers on the same machine may suffer from interference, contending in shared resources, heavily affecting their throughput (see §2.3.1). Moreover, groups of machines failing together that are not uncommon in large compute clusters, can affect

application resilience — an LRA with a random container placement might lose multiple containers at once (see §2.3.2). Despite these observations, support for effective long-running application placement in existing cluster schedulers is still elementary [10, 78, 82, 137].

Long-running applications require precise control over container placement to unlock their full potential. A common challenge of modern cluster managers is to provide support for high-quality placement of LRAs that run along traditional applications in shared compute clusters while guaranteeing cluster resource efficiency.

1.2 Research motivation

In order to cope with the growing class of long-running applications, there is a need for scalable cluster management architectures that can support precise and high-quality container placement. At the same time, cluster management systems should take into account global cluster-wide goals (e.g., reduce load-imbalance) and effectively allocate resources (e.g., be cost-effective). Data-parallel processing systems already execute long-running applications that utilise shared clusters to analyse large volumes of data. However, modern unified applications and complex workloads require more sophisticated systems and abstractions that can understand the unique execution requirements of their applications and are able to efficiently utilise shared resources (e.g., extract the most computation from resources).

In general, there are fundamental mismatches between the needs of modern long-running applications and traditional data-centre system designs. The shortcomings of these systems include:

- **Lack of expressive container placement.** Even though shared compute clusters today consolidate a variety of workloads with several of them requiring precise control of their containers for performance or resilience, existing cluster managers lack support for placement constraints that can capture such requirements in an expressive and high-level manner. Such constraints are critical to achieve higher-quality resource allocations, satisfy application requirements and also guarantee the healthy operation of the cluster.

- **Lack of unified data computation abstractions.** Even though data processing frameworks evolved to support different types of computation such as batch and streaming within the same execution engine that can be expressed using a single unified API, current data processing frameworks lack understanding of the diverse execution requirements of unified applications. Data processing systems need to utilise new abstractions that can capture complex job requirements and also satisfy their demands during execution.

1.3 Contributions

To address the aforementioned challenges, in this thesis we introduce *unified dataflows with placement constraints*, a new abstraction for efficiently unifying dataflows with different computation demands in shared compute clusters. In addition to data and computation, they can capture dataflow execution requirements and placement constraints. Unified dataflows with placement constraints can therefore: (i) combine data-parallel programs with different execution requirements as part of the same application; and (ii) control the placement of their long-running containers in the cluster.

To satisfy diverse unified dataflow requirements at runtime, we introduce suspendable tasks and novel scheduling policies that dynamically prioritise tasks within long-running containers as part of a distributed dataflow framework. To control long-running containers during placement, we introduce rich placement constraints that capture container interactions. We implement placement constraints as part of a cluster manager that achieves high-quality placement on clusters consisting of thousands of machines.

1.3.1 Unified dataflows

Similar to traditional dataflow graphs, unified dataflows represent data and computation explicitly in a graph. A unified dataflow graph may consist of multiple sub-dataflows, with each possibly having specific execution requirements such as high-throughput or low-latency. In unified dataflows, these execution requirements are treated as first-class citizens. Users can implement complex dataflows using unified APIs and mark explicit dataflow requirements expressed as priorities. A dataflow framework that implements unified dataflows must be flexible in providing high-throughput and low-latency data processing according to user-expressed requirements.

We introduce novel scheduling policies that are aware of user-defined requirements and can preempt tasks within long-running containers. Using application-level preemption policies, we can control the execution of diverse tasks in a dataflow. To preempt tasks efficiently, we implement a lightweight task preemption mechanism allowing tasks to be suspended and resumed within milliseconds while preserving their execution progress. Our scheduling policies along with the efficient preemption mechanism enable dataflow platforms to dynamically prioritise tasks according to their needs while efficiently utilising container resources.

Unified dataflows expose execution requirements as first-class citizens to the system. This makes it possible to employ application-specific mechanisms and scheduling policies to dynamically prioritise tasks within long-running containers according to their needs.

1.3.2 Placement constraints

For their execution, unified dataflow platforms utilise long-lived containers in compute clusters that are shared with a variety of other workloads. To express dependencies across long-running containers during placement, unified dataflows make use of placement constraints. Placement constraints enable both users and cluster operators to satisfy specific performance and availability requirements of long-running applications in shared clusters. For example, one can minimise resource interference between specific containers types (e.g., memory intensive) by placing them in different machines. This is a typical anti-affinity constraint across nodes in the cluster.

We introduce expressive high-level constraints that enable users to specify container placements both within and across applications without requiring knowledge of the underlying cluster infrastructure. We show that a single cardinality constraint type is sufficient to capture a wide variety of cases related to application performance and resilience. Our rich, high-level constraints along with a two-scheduler design enable cluster managers to optimise the placement of long-running containers without impacting traditional short-running containers running in the clusters.

Rich, high-level placement constraints allow the concise representation of interactions across containers without revealing the underlying cluster infrastructure. Cluster managers can utilise such constraints to achieve higher-quality placement of LRAs in shared compute infrastructures.

1.4 Related publications

Parts of the work described in this thesis are part of peer-reviewed publications:

- **NEPTUNE: Scheduling Suspendable Tasks for Unified Stream/Batch Applications.** Panagiotis Garefalakis, Konstantinos Karanasos and Peter Pietzuch. 10th ACM Symposium on Cloud Computing (**SoCC**), Santa Cruz, CA, 2019 [52]
- **MEDEA: Scheduling of Long Running Applications in Shared Production Clusters.** Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh and Sriram Rao. 13th ACM European Conference on Computer Systems (**EuroSys**), Porto, Portugal, 2018 [51]

The following publications have also influenced the work described in this thesis, but did not directly contribute to:

- **Peering through the Dark: An Owl's View of Inter-job Dependencies and Jobs' Impact in Shared Clusters.** Andrew Chung, Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Panagiotis Garefalakis and Gregory R. Ganger. ACM International Conference on Management of Data (**SIGMOD**), Amsterdam, Netherlands, 2019 [25]

- **Java2SDG: Stateful Big Data Processing for the Masses.**
  Raul Castro Fernandez, Panagiotis Garefalakis and Peter Pietzuch.
  32nd IEEE International Conference on Data Engineering (**ICDE**), Helsinki, Finland, 2016 [46]

1.5 Outline

The remainder of this thesis is structured as follows:

- **Chapter 2** presents background and related work. It first describes basic concepts and terminology, and then gives an overview of workloads executed in shared compute clusters today with an emphasis on the growing class of long-running applications. The chapter then tracks the evolution of data processing frameworks, a particular type of long-running applications, from single purpose engines to unified execution engines, and showcases their limitations in terms of complex application scheduling. Finally, it traces the advancements in cluster management and lays the requirements a cluster scheduler must satisfy in order to effectively place long-lived containers. The requirements are based on observations from a series of long-running application experiments on a real pre-production cluster.

- **Chapter 3** describes unified dataflow graphs with placement constraints, the new abstraction proposed in this thesis for efficient data processing in shared compute clusters. The chapter explains model characteristics including dataflow requirements and placement constraints in detail using examples.

- **Chapter 4** describes NEPTUNE, an execution framework for unified dataflow applications that can dynamically prioritise tasks based on their user-defined dataflow requirements. The chapter focuses on the system aspects of NEPTUNE and describes the implementation details of its suspendable tasks and its novel scheduling policies on top of Apache Spark, one of the most widely used data processing frameworks today. Finally, in a range of experiments, it shows that our Spark-based implementation of NEPTUNE on a 75-node Microsoft Azure cluster can achieve close to optimal performance for both latency-sensitive and latency-tolerant dataflows with a modest memory impact. At the same time, NEPTUNE can better utilise long-running container resources in the cluster.

- **Chapter 5** describes MEDEA, a cluster scheduler with rich placement constraints that can support the placement of applications with long-running containers as well as traditional analytics jobs with short-lived containers. The chapter first explains the system components of MEDEA including its novel two-scheduler design and its powerful constraints API, and then describes the implementation details on top of Apache Hadoop YARN, one of the most popular open-source cluster managers today. Finally, over experiments we show that the YARN-based

implementation of MEDEA on a 400-node pre-production cluster can achieve placements of long-running applications that yield significant performance and resilience benefits compared to state-of-the-art schedulers today.

- **Chapter 6** summarises the contributions of this thesis and presents potential future research directions.

Chapter 2

Background

This chapter describes big data systems and concepts used in this thesis related to long-running applications (§2.1). We cover the evolution of data processing systems, describe their programming and execution model, and list their requirements (§2.2). We also trace the advancements of cluster management systems responsible for controlling shared compute clusters and give an intuition of their requirements in respect to long-running applications (§2.3).

In the era of big data, modern organisations rely on distributed data processing *applications* to improve the quality of their services. These applications can involve different types of computation such as batch processing, streaming, machine learning and artificial intelligence forming complex workloads. In this context, a *workload* represents a collection of resources and computation logic that delivers specific business value. Workloads typically consist of several jobs, and each *job* runs a sequence of tasks. A job *task* is the smallest form of computation processing a set of data. Based on their characteristics, workloads can be classified into different categories, based on their resource requirements (i.e., memory, IO or CPU intensive), inter-arrival times (i.e., daily, weekly), or traffic patterns (i.e., static, periodic, unpredictable) [4].

Data processing *frameworks* such as MapReduce, Spark and Hive have enabled developers to create such distributed applications that have the ability to process increasing volumes of data [32, 66, 150]. These frameworks offer user-friendly programming interfaces and are designed to run on shared-nothing commodity infrastructures (see §2.2). Applications can thus seamlessly scale on hundreds of machines that are cheaper to maintain and operate [12]. Larger versions of those *clusters* comprise of thousands of machines, potentially with different hardware characteristics, connected over the network [31, 137]. Using general-purpose cluster managers, the resources of each *machine* in the cluster can be *shared* across several frameworks, users and applications.

Cluster managers, or schedulers, such as Hydra [31], Kubernetes [82], YARN [135], Borg [137], and Mesos [65] carry out the on-demand allocation of machine resources to applications. Resources such

as CPU, memory and disk are packaged in the form of *containers* and are shared across applications running computation tasks within those containers. Schedulers typically place containers on machines based on simple scheduling algorithms, trying to make fast placement decisions. They can also offer advanced features as part of their scheduling logic such as capacity planning, fairness, support for hardware heterogeneity and more (see §2.3). To protect containers running on the same machine and competing for shared resources such as CPU, memory and network, some schedulers also support resource isolation mechanisms such as cgroups [82, 135].

However, the fast-paced evolution of frameworks and the diversification of applications that run in shared compute clusters pose new challenges for the systems that support them. Data processing frameworks, evolved from single purpose engines to unified execution engines with long-running containers, and cluster managers now consolidate a plethora of diverse workloads in clusters consisting of thousands of machines. In this thesis, we highlight some of the challenges and explore a new opportunity to address them with our contribution of *unified dataflows with placement constraints*.

More precisely, out of the diverse workloads running in shared compute clusters today, the focus of this thesis is on long-running applications. In this chapter, we highlight an important mismatch between cluster managers that were traditionally focusing on fast allocation decisions for applications with short-running containers and an increasing number of long-running applications with complex placements deployed in large-scale shared clusters. At the same time, applications started utilising long-running containers that can be used to run computation with diverse requirements. Another critical mismatch is between modern data processing systems that evolved to support different types of computation within those containers and the diverse execution requirements of applications developed on top of these systems that are currently ignored.

In §2.1, we briefly discuss the different types of workloads that are currently deployed in compute clusters based on existing studies and our analysis across six clusters within Microsoft, while in §2.1.2 we describe their scheduling life-cycles. Then in §2.2, we focus on the evolution of data-parallel processing systems and we experimentally demonstrate using a real-world application on Azure how existing scheduling policies fail to satisfy complex dataflow requirements during execution in §2.2.5. Driven by experiments on a 275-node pre-production cluster, we explore the new challenges that arise from placing the containers of long-running applications along other workloads in a shared cluster in §2.3. Finally, we list the requirements that a cluster scheduler must satisfy to efficiently support long-running applications in §2.3.4.

2.1 Cluster workloads

Large-scale data analysis provides the means for companies to improve product engagement and increase their revenues. For this purpose, companies have traditionally utilised batch analytics frameworks to perform off-line analysis of their data [32, 113, 154]. Batch data analysis involves jobs

that run computation tasks as part of short-lived containers in the cluster. Analysis results are not used in real-time and thus batch computation mostly focuses on high-throughput processing and does not usually yield strict latency requirements.

Besides batch analytics, an increasing number of applications currently relies on distributed systems to scale and handle increasing volumes of data [104]. To handle user-facing requests in real-time or process continuous streams of data, efficient on-line streaming services use long-lived containers with continuously running tasks to achieve low end-to-end latency [45, 47, 64, 98]. This richness of modern applications relying on distributed infrastructures for scalability and fault tolerance, led to increasingly varied workloads running in shared clusters today.

In fact, studies on publicly available cluster traces from several companies confirm the increasing workload diversity in terms of job runtimes and resource requirements [4, 87, 112]. In the publicly available Google trace, around 80% of the jobs utilise containers in the cluster that have durations of less than 12 minutes each, while the longest container duration is at least 29 days, which in this case is the duration of whole trace [112, 139]. In Alibaba clusters, there is significant reported imbalance in terms of container resource usage between long-running on-line services and off-line batch analytics [87].

In addition to existing studies, we perform an analysis of cluster workloads within Microsoft. For the analysis we use six analytics clusters, each comprised of tens of thousands of machines with a variety of hardware characteristics. In these clusters, thousands of applications utilise containers that are using shared cluster resources. When the active container durations of an application are above 12 hours, we mark it as a long-running application, also known as LRA. As shown in Figure 2.1, within Microsoft analytics clusters, at least 10% of each cluster's machines are used for LRAs while two are used exclusively by them. Our analysis, therefore, confirms that a substantial portion of cluster resources is dedicated to applications that use long-lived containers such as on-line services. In general, as the percentage of cluster resources used for LRAs is reported to be growing [31], support for LRAs in modern compute clusters is going to be even more crucial in the near future.

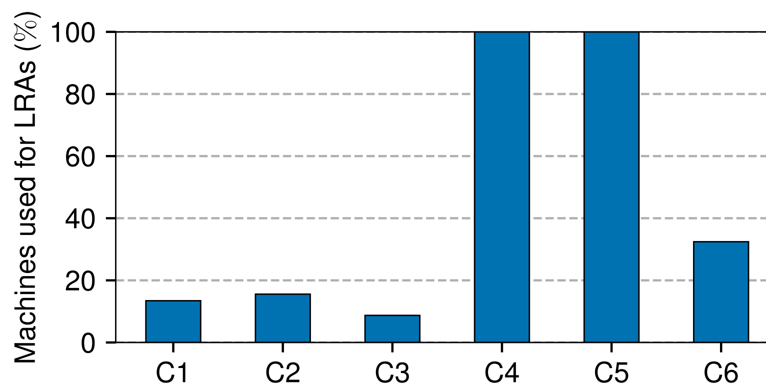


Figure 2.1 Machines used for long-running applications (LRAs) in six analytics clusters at Microsoft

Placing LRAs with long-lived containers along with batch jobs utilising short-lived containers in shared compute clusters is both appealing and challenging. On the one hand, it can reduce cluster operational costs by increasing resource efficiency, avoid unnecessary data movement, and enable pipelines involving both classes of applications. On the other hand, it can add conflicting demands to the cluster scheduler managing the cluster as we describe in more detail in §2.3. Applications with short-lived containers such as batch jobs need to be placed with low-latency to achieve high processing throughput, while LRAs might need high-quality placement to achieve good performance and availability.

2.1.1 Workload types

Depending on the duration of their containers, cluster workloads can be classified as either batch or long-running, with the former using short-lived containers and the latter utilising long-running containers. Below we identify some key application scenarios for these two categories of workloads and describe their main characteristics.

Batch. Represent traditional workloads in production clusters that utilise short-running containers to execute tasks that run for durations ranging from seconds to minutes. Examples include frameworks such as MapReduce [5, 32], Scope [154] and Tez [129], performing off-line computation, e.g., extracting, transforming and loading data (ETL) in order to provide data insights and improve the quality of services. Frameworks that perform off-line computation with short-lived containers are not sensitive to machine failures as they can spawn new ones and continue data processing from the last checkpoint [20, 70, 106, 114]. Their containers have no strict placement requirements except optional data locality to avoid data movement. However, as batch computation involves large numbers of processing tasks spawning several containers, these frameworks require low container placement latency to avoid increasing their total task runtime (see §2.1.2).

Long-running. Apart from batch analytics applications, workloads in production clusters now include several long-running applications with containers running for durations ranging from hours to months. Below we identify different scenarios where LRAs are used. We base these scenarios on discussions with cluster operators from several companies and the open-source community:

- **Streaming systems** such as Storm [127], Samza [114], Heron [83], Flink [20], Millwheel [3], Kafka Streams [76], and SEEP [45] process data streams in near real-time via dataflows of long-running operators that are deployed using containers.
- **Interactive data-intensive applications** such as Spark [150], Impala [80], and Hive LLAP [19] employ long-standing workers, also known as executors, to avoid container start-up costs and to process data that resides in memory with low latency.

- **Latency-sensitive applications** such as HBase [62], ZooKeeper [69], and Memcached [94] serve requests using long-standing containers to achieve low end-to-end latency.
- **Machine learning frameworks** such as TensorFlow [1], Spark MLLib [125] and SystemML [15] use long-running executors to efficiently perform iterative computations.

In the analysis we perform within Microsoft's shared big data infrastructure (see Figure 2.1), we observe that, in most analytics clusters, tens of unique scenarios involve LRAs falling into one of the categories above. Some of these LRAs must meet strict latency requirements, scale to large volumes of data, or achieve high-availability. As their containers run for longer periods of time, they could tolerate longer placement latency than traditional batch workloads, however, they require higher-quality placement to meet their special requirements (see §2.3.1).

2.1.2 Workload scheduling

Both LRAs and batch analytics workloads utilise containers that execute computation tasks. To allocate containers, applications use cluster managers that are responsible for the placement of containers on shared machines. Figure 2.2 shows the container placement life-cycle. To execute tasks within those containers, applications or application frameworks such as Spark and Flink, use schedulers that take into account task characteristics, dependencies and available container resources when launching tasks. Figure 2.3 shows the task scheduling life-cycle. As we describe below, even though container scheduling takes place at cluster management level and task scheduling happens at the application level, their scheduling life-cycles share similar concepts.

Container scheduling life-cycle. Applications first submit container requests to a general-purpose cluster manager responsible for the placement, initialisation and monitoring of these containers. A container request waits to be considered for placement by the cluster scheduler. Next, the cluster manager triggers a scheduling algorithm to place the container according to specific policy or objective

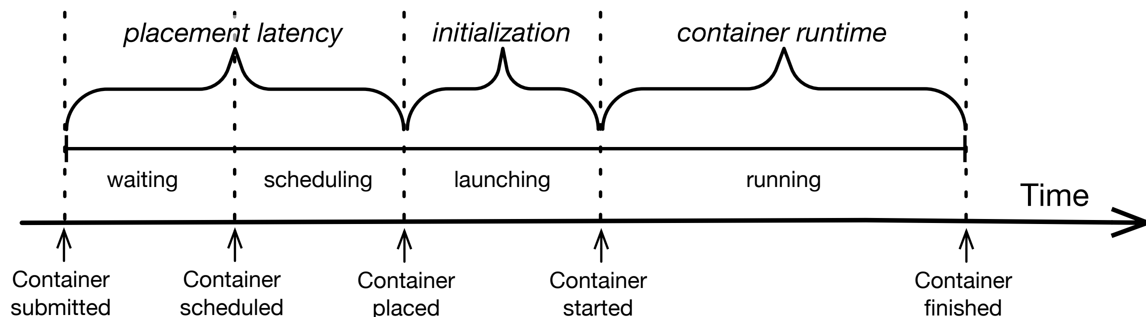


Figure 2.2 Container scheduling event life-cycle over time describing states at the bottom and metrics at the top

i.e., place the container in a machine in a way that balances load in the cluster. Upon container placement, the cluster manager starts container initialisation, also called localisation, downloading all the necessary binaries and dependencies to get the container ready for execution. Finally, the initialised container, which is now capable of executing tasks, is returned to the application.

Based on the event life-cycle described above, the main container metrics are: container placement latency, container initialisation latency and container runtime (see Figure 2.2). *Container placement latency* is the time between a container request is submitted to the scheduler until its placement on a machine. It represents the time it takes the scheduling algorithm to decide where to place a container in the cluster. *Container initialisation latency* is the time it takes the scheduler to install dependencies, download binaries and prepare the container on a specific machine. It represents the container set-up time. *Container runtime* is the time spent running application tasks and performing computation. Depending on the execution policy of the application, a container can be used to run a single application task [32] or multiple ones in order to avoid unnecessary container start-up costs [80, 150].

From the cluster management level, both LRAs and batch analytics follow the same container life-cycle transitions even though they have different needs. LRAs often require high-quality placement for their containers to satisfy their performance and resilience requirements without the need for fast placement decisions. Batch analytics containers must be placed with low latency as their completion time can be significantly affected by increased placement costs.

Task scheduling life-cycle. An application uses the acquired containers placed on cluster machines to execute tasks. Each task is a computation instance usually utilising a single core of a container. Tasks that are ready to execute are first submitted to an application specific scheduler that is responsible for task execution and monitoring. A task is waiting in a queue to be considered and for resources to become available. Next, the application scheduler uses a policy to decide the task that is going to be executed next within a container i.e., run the task that was earlier submitted, or the task with the highest priority in the first available container. Finally a task runs until completion in a container.

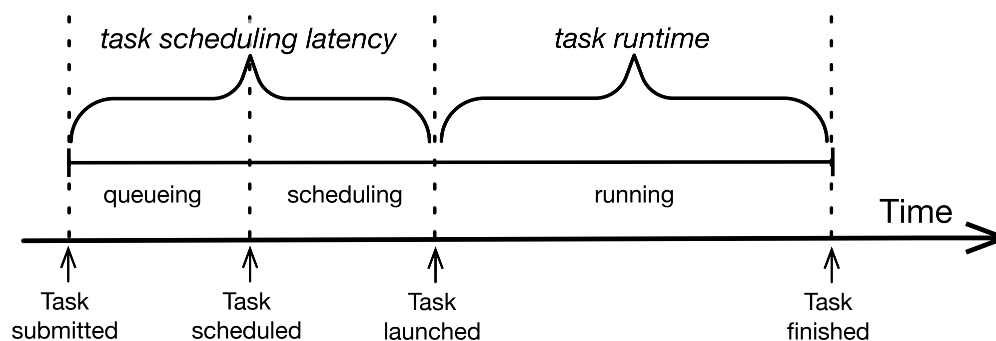


Figure 2.3 Task scheduling event life-cycle over time describing states at the bottom and metrics at the top

Main task life-cycle metrics include: task scheduling latency and task runtime (see Figure 2.3). *Task scheduling latency* is the time between a task being submitted to the application scheduler until being launched within a container. It includes the time that it takes the scheduler to decide where to run a task and also the time a task has to wait for resources to become available. *Task runtime* is the time spent running application specific computation in a container also including infinitely long tasks by LRAs such as database services or cache servers. Note here that a task does also deserialise some application code or data which is similar to a container initialisation stage. As the duration of that stage is typically short and cannot be avoided, it is not included in our figure.

In a cluster environment, application schedulers follow similar task life-cycle transitions even when their application requirements are entirely different. For example, traditional batch applications schedule a single task per short-running container for simplicity, LRA services schedule continuous tasks within long-lived containers to serve requests with low latency, while some data-intensive LRAs utilise long-running containers to execute several heterogeneous tasks with reduced initialisation costs. As we describe in more detail below, the diverse tasks of an LRA may have different requirements in terms of latency and throughput while being scheduled on the same shared long-running containers. Increased task queueing or scheduling latencies can thus significantly affect task requirements and consequently the goals of an entire LRA application.

2.2 Data processing systems

In the previous section, we classified shared cluster workloads and discussed their container and task scheduling life-cycles. This section describes big data processing systems. We first focus on their abstraction (§2.2.1), programming interface (§2.2.2) and execution model (§2.2.3) and then on how they adapted to support the heterogeneous jobs of modern data processing applications (§2.2.4). Finally, we show with experiments how existing data processing systems fail to satisfy modern application requirements (§2.2.5).

Over the past years, many parallel processing systems have been developed to enable users analyse large volumes of data by executing computation within short- or long-lived containers [5, 6, 7, 66, 127]. In order to scale processing, these systems employ a data-parallel model. In a *data-parallel* model, the same computation is performed over smaller chunks of data that are also known as partitions. This is different than the task-parallel model where different computation is performed over the same or different data. Essentially, task parallelism allows splitting memory or CPU intensive computation in smaller tasks while data parallelism allows splitting IO intensive computation to many smaller but identical tasks.

Big data workloads are processing increasing volumes of data and are traditionally bound by the amount of data they can transfer and process. Therefore, big data processing systems expose dataflow-based abstractions that accommodate data-parallel processing [32, 151]. By employing the dataflow

abstraction, processing systems extract tasks that can be executed in parallel from the application definition with minimal user intervention. As a result, users of the data processing systems do not have to explicitly choose the granularity of the tasks but only how to partition the data based on the available resources.

Data parallelism enables systems to scale throughput with the amount of resources available. Each application task is running computation over an individual portion (partition) of the total data as part of a container. As tasks can process data in parallel, one can reduce the total computation time by just adding more resources and spawning new tasks. By adding new tasks, each task will be responsible for processing a smaller partition of data.

To increase the amount of usable resources, data processing systems follow a scale out approach where they transparently implement the distributed logic to utilise the aggregate resources of many commodity machines connected over the network. To reduce total computation time and scale, one can just add machines of similar hardware characteristics to the network. This is a fundamentally different approach than scaling up resources and using more powerful hardware. Scaling up is usually a cost-ineffective approach where a slight increase in resources could multiply costs and most importantly is limited by Moore's law [116].

Most of the existing well-established distributed data processing systems are targeting a particular use case. They exploit domain specific knowledge to improve their performance while preserving system simplicity. For instance, batch analytics frameworks such as Scope [154] and Tez [113] are mainly targeting off-line computation over huge data sets spanning many machines in a cluster. They focus on improving data insights and performing ETL computations while quickly recovering from hardware failures. Stream processing frameworks like Flink [20] and MillWheel [3] are mainly targeting real-time computation for tasks like detecting malicious behaviour over clickstreams of data while maintaining low end-to-end latency. Machine learning frameworks such as Tensorflow [1] and Spark MLLib [125] focus on performing efficiently iterative computations and train models faster by caching intermediate results in memory.

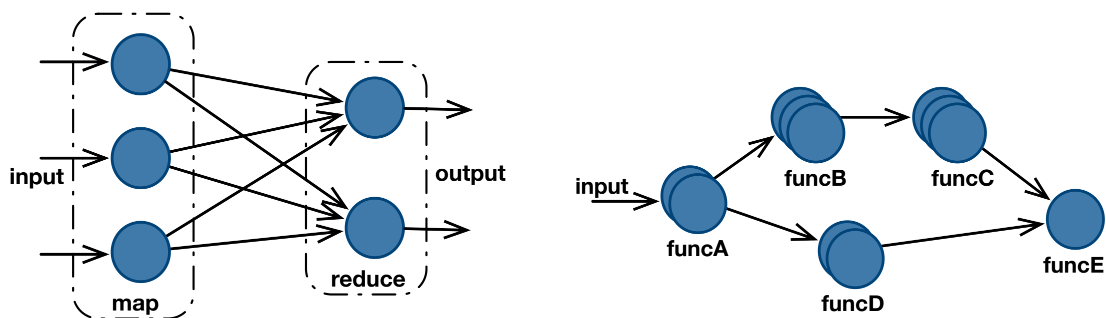
In essence, distributed data processing systems enable developers to transparently run scalable parallel computation over large sets of data. They hide the underlying complexity of data partitioning, network communication, coordination and fault tolerance from the users. As we describe next, they follow a dataflow model offering a variety of programming interfaces including lower-level interfaces such as functional or imperative APIs and higher-level abstractions like the SQL declarative language (§2.2.2). Using these interfaces users develop applications that perform computation ranging from batch processing to streaming, machine learning, and artificial intelligence (§2.2.4). Computation is represented as dataflows of operators that are executed within containers in clusters of machines (§2.2.3). In what follows, we describe the dataflow model in more detail.

2.2.1 Dataflow model

Applications in data processing systems are represented as dataflow graphs. Under the generic dataflow computation model, nodes or vertices perform the user-defined computation while edges represent the flow of data across the nodes. Nodes apply computation functions on data that can be stateless or stateful, storing the data locally. The output of an upstream node can be used as an input by a downstream node connected with an edge. As a result, edges in a directed graph define the flow of data. Using dataflow graphs, a system can easily identify elements that can run in parallel and thus provide a good representation of parallel computation. Implementations of the dataflow model in existing data processing systems include the *MapReduce dataflow model* and the *acyclic dataflow model*, both discussed below.

MapReduce dataflow model was proposed by Google in 2004 and influenced the first breed of big data processing systems [5, 33]. It offers a simple computation model consisting of map and reduce tasks. Map tasks process the input by applying a computation function on the input data and reduce tasks aggregate the processed map data and output results. As depicted in Figure 2.4a, dataflow graphs expressing MapReduce computation are quite simple. Users develop their applications using stateless map and reduce functions. The dataflow system first runs the map tasks applying user-defined map function over partitions of the input data. For every data element the map function returns zero or more key-value pairs. Then, the key-value pairs are sorted locally by key across all tasks. In the reduce phase, the dataflow system assigns a group of keys to each of the reduce tasks. A reduce task finally applies the user-defined reduce function and outputs the results. By partitioning the input data, map and reduce tasks can run in parallel as shown in the figure.

The simplicity of the MapReduce programming model made it a popular choice across several organisations. In addition to that, the ability to increase processing rate by scaling out to clusters of machines and being able to tolerate hardware failures fuelled its wide adoption. However, the



(a) MapReduce dataflow model consisting of map and reduce tasks

(b) Generic dataflow model forming a DAG of arbitrary computation functions

Figure 2.4 Examples of different dataflow models consisting of tasks shown as circles and flows of data represented as edges

simplicity of the model proved to have its own disadvantages as it can not express more complex applications. Higher-level frameworks such as Pig [53, 99] and Hive [68] tried to address this limitation by generating and managing multiple MapReduce jobs for more complex applications.

Acyclic dataflow model is a generalisation of the MapReduce dataflow model that allows to represent more complex applications. It models application computations as directed acyclic graphs or DAGs of tasks as shown in Figure 2.4b. Instead of relying on map and reduce functions, this model allows tasks of any user-defined function. Tasks that apply those functions can be interconnected in arbitrary ways forming a DAG, as long as no cycle exists in the graph. In an acyclic dataflow, tasks that perform on different partitions of the data can execute in parallel while tasks that depend on the output of other tasks can run as soon the former produce some output.

Dryad was one of the first distributed data processing systems that was based on the acyclic dataflow model [70]. It allows DAGs of tasks that perform user-defined functions. Tasks are connected with edges of communication channels such as files, TCP pipes, and shared-memory FIFOs in order to send data. Unlike MapReduce, Dryad allows users to write more complex dataflow graphs that are expressed as several connected sub-dataflows. Higher-level interfaces on top of Dryad evolved to expose more user-friendly declarative APIs that simplify expressing iterative computation [148].

Spark is a more recent big data processing system based on the acyclic dataflow model. Spark builds a DAG of stages for each application. Every stage groups several identical tasks that can run in parallel [150]. Spark performs in-memory computation by utilising special data structures called Resilient Distributed Datasets or RDDs [151]. Unlike MapReduce where computation tasks always store their output on disk, Spark maintains output in memory where possible. As a result, tasks consuming upstream output can directly stream it from memory resulting to orders of magnitude throughput improvements compared to MapReduce. Unlike MapReduce and Dryad that have to materialise intermediate results for fault tolerance, Spark maintains the lineage of data transformations. To recover from failure, Spark reapplies in parallel the data transformations recorded in the lineage graph and continues making progress.

2.2.2 Programming interface

Programming interfaces exposed by data processing systems allow developers to implement applications without worrying about the underlying dataflow execution complexity such as task parallelisation. Applications expressed using such interfaces are first translated to dataflows and then jobs with parallel tasks that can be directly executed by the runtime. Data processing frameworks expose one or more programming interfaces including lower-level functional, imperative or dataflow interfaces and more intuitive higher-level declarative programming abstractions. Modern frameworks even allow the mixture of those interfaces to implement applications. The above interface distinction to low- or high-level is made based on the programming abstraction (e.g., declarative, or imperative)

and not necessarily how close it is to the execution abstraction (e.g., map and reduce functions, or dataflow vertices). Below we describe such lower- and higher-level programming interfaces targeting distributed data processing systems.

Low-level interfaces

Lower-level programming interfaces include imperative style programming languages such as the ones exposed in FlumeJava [21], Pig Latin [99], Sawzall [105] and Piccolo [106]. FlumeJava is a Java library that exposes immutable collections supporting operations for data parallel processing. The operations exposed by each collection form a dataflow graph which is then optimised and translated to MapReduce jobs that can run in a distributed or local setting. Pig Latin is a domain specific language exposing special operators that can be used as part of statements. A sequence of statements or steps can be then combined as part of an application that runs on top of the MapReduce platform. Similar to Sawzall, this type of language flexibility enables the development of more complex application logic that can be translated to simpler MapReduce jobs.

Systems such as Heron [83], Flink [20], Storm [127] and SEEP [45] offer a dataflow programming abstraction that models applications as computation graphs, also called topologies. Topologies are defined as directed graphs where the vertices represent computations and the edges represent the data flow between the computation components. Applications in Storm and Heron can be defined using different programming languages such as Java or Python where developers implement their programs using specific APIs. The application topology, defining all the application logic, is then executed on top of the particular data processing runtime.

Low-level programming interfaces also include functional abstractions like the ones exposed by systems such as Spark [151], Nectar [61] and D-Streams [152]. For instance, Spark exposes RDDs as its main programming abstraction. They are a form of immutable, partitioned, implicitly distributed collection as well as a functional programming interface providing fault-tolerance in a transparent manner. The abstraction supports an expressive set of operations named transformations and actions that can be used to implement application logic. Transformations like map and filter are lazily evaluated merely constructing a computation graph, while actions like count trigger the actual computation. The computation graph is captured by an RDD and describes the distributed operations in coarse-grained manner which are then executed by the runtime.

The code example below uses the functional Spark RDD API to first load the contents of a potentially distributed log file in an RDD, then run a filter operation keeping only the lines that start with an error and finally run a count action returning the total number of errors.

```
1 val lines = sc.textFile("/var/log/syslog")
2 val error = lines.filter(s => s.startsWith("ERROR"))
3 println(s"Total errors: ${error.count()}")
```

High-level abstractions

Higher-level interfaces on top of data processing systems include the declarative programming model where developers express the computation logic and the desired result without describing implementation details such as the transformations on data. SQL is a popular declarative language that has been widely used over the past decades to interface with database systems. More recently, we witnessed significant efforts to bring similar interfaces on top of distributed dataflow systems as they are more intuitive to users.

Examples include systems such as Pig [53] and Hive [130] offering SQL-style interfaces to developers. They both run on top of the MapReduce platform and translate SQL applications to graphs of MapReduce jobs. They also allow more complex applications to be expressed as a series of simpler jobs with data dependencies across them. Shark [141] was a similar effort for Apache Spark by replacing the physical plan generator of Apache Hive to generate RDDs instead of MapReduce jobs. Impala was another system offering SQL-style interface to the users on top of Apache Hadoop but implementing its own execution runtime instead of relying on MapReduce or Spark jobs [80].

Spark SQL [8] is a replacement of Shark, providing better integration between relational SQL code and traditional Spark processing code. It is based on the DataFrames abstraction which is conceptually equivalent to a table in a relational database i.e., a collection of rows with named columns. DataFrames capture the high-level computations to be executed. Spark computes a query plan and then the query plan is optimised using a relational query optimiser named Catalyst. Finally, for the execution the optimised higher-level DataFrame is simply translated to an RDD representation that the Spark engine can execute.

The code example described above using the functional Spark API can be rewritten using the declarative SQL interface as:

```
1 lines.createOrReplaceTempView("lines")
2 val error = spark.sql("select * from lines
3                  where value like 'ERROR%'")
4 println(s"Total errors : ${error.count()}")
```

Declarative interfaces found great success among data analysts as they can now implement big data applications with higher-level programming abstractions they are already familiar with. At the same time, particular lower-level interfaces proved to be better fit for particular types of computation than others. For example, the dataflow abstraction can precisely represent streaming computation while functional abstractions provide a more concise representation of ETL or batch computation.

To utilise the benefits of all interfaces, modern distributed dataflow frameworks such as Spark and Flink, allow users to develop applications using a mixture of higher-level SQL-style relational queries with lower-level functional or imperative code. In addition to that, we also witnessed a shift towards

the unification of APIs to capture efficiently different types of computations using a single interface, as we discuss in more detail in §2.2.4. Applications implemented using higher, lower, or unified APIs are translated by frameworks to dataflow computation abstractions that can be natively executed by the dataflow execution engines on the entire cluster.

2.2.3 Execution model

In §2.2.1, we described how data processing systems utilise the dataflow model to scale out to increasing data sizes by parallelising computation and IO over clusters of machines. In §2.2.2, we discussed how users can take advantage of lower-, higher- and mixed-level interfaces to intuitively express their applications. Now we explain how dataflow frameworks execute translated applications using DAG abstractions regardless of the implementation interface.

In a dataflow engine, an application consists of one or more *jobs*, and each job is converted to a *directed acyclic graph* (DAG) of operators as shown in Figure 2.5. Operators in the DAG are usually grouped into *stages*. Each stage corresponds to a collection of *tasks* where every task performs computation over different partitions of data. Within each stage, many operators can be fused together, a technique also called operator chaining or pipelining in systems like Apache Spark. At each stage boundary, data is written to a local cache residing in memory or disk by tasks in the upstream stages and then transferred over the network to tasks in the downstream stages.

A scheduler is a data-driven component responsible for assigning tasks to containers in the cluster. For execution, a container receives a computation task and a reference to its input data. Scheduling decisions in dataflow systems are data-driven. Given the application dataflow graph and the already completed operators, the scheduler first decides which operator in the graph is ready for execution (has satisfied dependencies) and it then spawns parallel tasks on particular machines in the cluster that implement the computation logic of that operator. As a result, an *application scheduler* assigns tasks to containers respecting their data dependencies but also taking into account their resource needs and data locality [54, 58, 149].

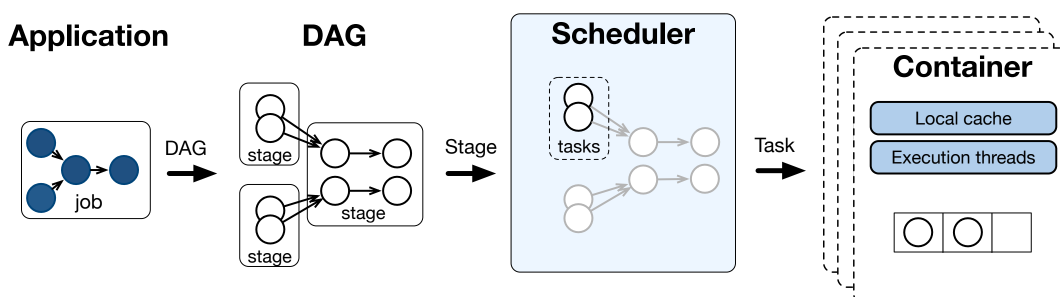


Figure 2.5 Execution overview of a dataflow application (Blue circles represent operators while white ones indicate different tasks.)

Depending on the execution model of the dataflow system, tasks are either executed on a container as long running (*continuous operator* model) [20, 45, 64, 127] or are scheduled incrementally as soon as their dependencies are met (*bulk-synchronous* model) [32, 70, 136, 150]. Systems utilising the continuous operator model can usually achieve lower latency by immediately processing a stream of data, while systems of the bulk-synchronous execution model make it easier to achieve high throughput by applying optimised execution techniques within each batch of data [16].

For efficiency reasons, most recent dataflow frameworks follow an *executor model* in which tasks are dispatched to long-running containers or executors. This is different than the traditional model of instantiating a new container for every task in frameworks like MapReduce. Executors are deployed on machines in the cluster, also called worker nodes, and typically run for the entire lifetime of an application thus avoiding repeated container scheduling latencies and initialisation costs. Every executor provides task *slots* that represent the compute capabilities of the node and each slot can execute a single task.

2.2.4 Evolution of the execution model

Distributed dataflow frameworks following the execution model described above have enabled users to analyse vast amounts of data using clusters of machines. User computation is expressed as a graph of operators that process flows of data expressed in a variety of lower-level, higher-level and more recently mixed and unified programming interfaces (see §2.2.2). In this section, we trace the evolution of those frameworks as depicted in Figure 2.6.

Early dataflow systems followed a particular execution model targeting a use case with very specific requirements. For example, some of the first dataflow systems were dedicated to batch processing while other systems focused on streaming computation. Examples of the former include Apache Hadoop [5], Hive [66], Spark [150], and Presto [107], while examples of the latter include Apache Storm [127], Flink [20], and S4 [98]. Batch computation systems mainly focused on providing *high processing throughput*, whereas streaming systems had also to achieve *low processing latency*.

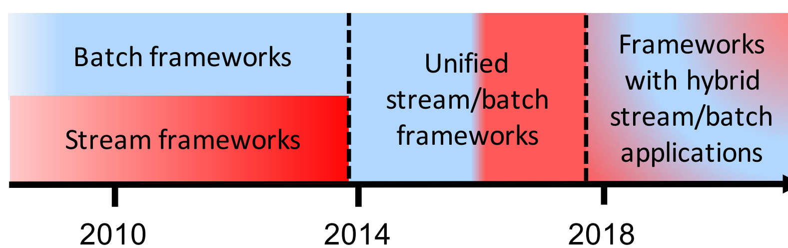


Figure 2.6 Evolution of stream and batch frameworks

This dichotomy was a limitation to the users who had to use multiple dataflow frameworks to accommodate their needs. At the same time, cluster operators had to logically separate cluster resources and make sure each framework had enough resources to scale. Acknowledging the limitation of having to manage multiple systems and copy data across them, both for users and cluster operators, dataflow platforms evolved to support both stream and batch applications within the same framework. Using a *unified stream/batch framework* such as Spark [150] or Flink [20], both stream and batch computation can be expressed and run as part of a single execution engine (see Figure 2.6).

As part of the evolution of unified analytics in general, several unified APIs evolved on top of existing popular frameworks [7, 20, 150], such as Spark's Structured Streaming [9], Flink's Table API [49], or Apache Beam [13] that was introduced as part of an external layer. Unified APIs are high-level programming interfaces that enable users to seamlessly express stream and batch computation using a single API instead of using separate APIs for each type of computation.

Structured Streaming, for example, is a high-level unified API based on Spark DataFrames. Users describe the computation query and the input that can be either bounded (batch) or unbounded (streaming) data. The system then runs the query incrementally with tasks operating on micro-batches of data, while maintaining enough state to recover from failure. Using a single programming interface users can develop applications that were traditionally handled by separate systems.

As the next step in this unification, over the past few years, users have begun to combine latency-sensitive stream jobs with latency-tolerant batch jobs as part of the same *hybrid stream/batch application* (see Figure 2.6). In the remaining of this thesis, we use the term "*stream/batch*" to refer to these applications where "stream" describes the latency-sensitive jobs of the application and "batch" to the remaining ones. Examples of such applications include on-line machine learning, real-time data transformation and serving, low-latency event monitoring and reporting.

For example, a stream/batch application implementing a real-time service to detect malicious behaviour would use a batch job to train a machine learning model over historical data while at the same time it would use a stream job performing inference over the trained model to detect malicious behaviour on real-time data [48, 119]. Stream/batch applications enable all jobs of an application to execute as part of the same containers, share computation logic and application state across their jobs as described in more detail in §3.2. Traditionally, a stream/batch application like the one described above, would have to be executed by separate engines: a batch engine for training the model, and a stream engine to perform the inference as dictated by the "lambda" architecture [91].

As we discuss in §2.2.5, even though existing dataflow platforms evolved to provide the logical abstractions for unified applications to express their stream/batch jobs, there is no scheduling support for these hybrid applications. Execution engines are unaware of stream/batch job requirements as these applications are captured as typical dataflows.

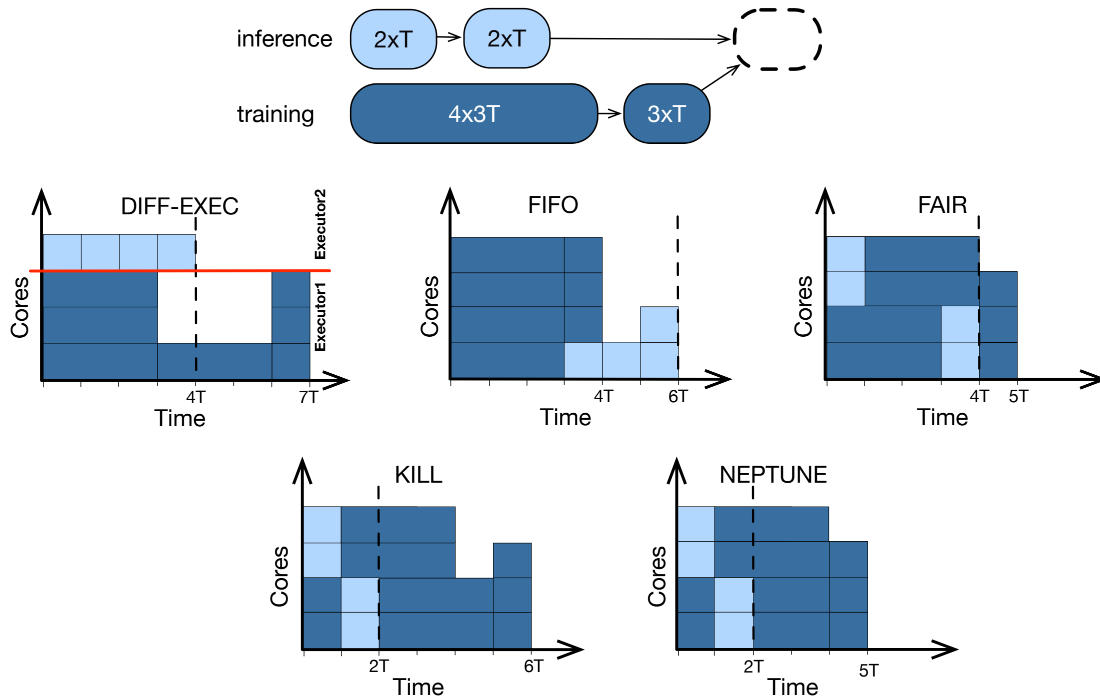


Figure 2.7 Resource usage over time for the malicious behaviour detection application under different scheduling policies

2.2.5 Scheduling hybrid dataflow applications

Dataflow platforms such as Apache Spark provide API support for hybrid stream/batch applications. However, they have well-known limitations in terms of scheduling and execution especially when both stream and batch tasks operate on micro-batches as we explain below [120, 121, 122]. In this section, we show how the different jobs of a stream/batch application are handled by existing distributed dataflow platforms that schedule tasks incrementally.

Figure 2.7 illustrates the scheduling problem that a distributed dataflow system faces when dealing with stream/batch applications. In our example, the stream/batch application implements a malicious behaviour detection service . It consists of a real-time inference job and a historical data training job. Rectangles on the simplified DAG at the top of the figure represent job stages consisting of tasks. Each task in a stage requires a single resource unit. Execution time on the x-axis is normalised in time units (T). Light blue rectangles represent tasks of the (stream) inference job while dark blue ones represent (batch) training tasks. Note that training tasks require up to 3 times more time and 2 times more resources than the inference tasks and can thus introduce significant queueing delays to the latter.

In an ideal scheduling scenario for this application, stream tasks would start execution immediately to achieve the lowest latency, which is $2T$. The remaining resources would be used by batch tasks to achieve high resource utilisation and throughput.

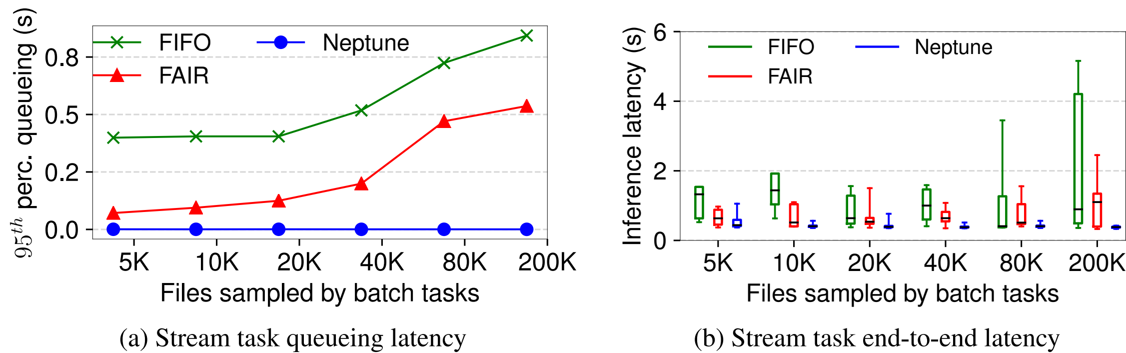
A basic approach that avoids queueing and achieves low latency employs *static resource allocation* in which each job uses a dedicated set of resources (DIFF-EXEC in the figure). This corresponds to approaches that either use separate engines for stream and batch ("lambda" architecture), or use a single engine but submit separate applications for each job. This is a common solution in existing production environments that use general-purpose resource managers and have dedicated resource queues for latency-critical jobs [23, 30, 35].

In our example, we assign 25% of the resources to the stream job executor and the rest to the batch job executor. Although the latency for the stream job is low, it is still $2\times$ higher than the optimal (that is $2T$), as separate executors cannot guarantee low latency. More importantly, as resources cannot be shared across jobs, the application completes in $7T$, which is the worst of all scenarios. Jobs could be allowed to go over capacity, but that would result in either higher queueing delays for stream jobs or wasted work for batch jobs, as we discuss below. Allowing jobs to go over capacity could increase utilisation, however, using separate job executors does not allow sharing of application state or logic across jobs, which is a fundamental weakness of this approach.

By default, unified stream/batch engines such as Spark [8] and Flink [20] schedule applications with multiple jobs on shared executors in a FIFO fashion, based on job priorities and job submission times. Every job is divided into stages and each job gets priority on all available resources sequentially as long as its stages have tasks to launch. When jobs at the head of the queue use only part of the cluster, subsequent jobs can start running immediately. If they consume all available resources, later jobs may be delayed significantly. In our example, if the training job gets triggered first, it will consume all cluster resources and when the inference job arrives, it will get queued behind it, leading to an increased latency of $6T$.

FAIR is an alternative scheduling policy that runs tasks on shared executors in a round-robin fashion. All jobs receive an equal, possibly weighted, share of resources. Stream jobs submitted while a batch job is running can obtain resources right away and experience good response times, without waiting for the batch job to finish. In our example, FAIR achieves better utilisation (the application completes in $5T$) and reduces the stream job's response time by $2T$, compared to FIFO. FAIR, however, cannot guarantee low queueing delays. The achieved latency is $2\times$ higher than the optimal, because FAIR treats all jobs as equal without respecting job latency requirements.

To avoid queueing delays for the latency sensitive stream tasks completely, it is possible to employ non work-preserving *preemption* by killing tasks of batch jobs when needed. This mechanism can be combined with any of the above scheduling strategies. In Figure 2.7, KILL coupled with FAIR scheduling policy shows minimal queueing, but requires restarting tasks and losing their progress. This policy side-effect leads to the second worst effective resource utilisation and thus throughput as



(a) Stream task queueing latency

(b) Stream task end-to-end latency

Figure 2.8 Queueing and end-to-end latencies observed for the detection application implemented in Spark over varying sizes of batch jobs

the application completes in $6T$. Even if jobs support work-preserving preemption, one would have to account for the extra overhead of checkpointing task state.

Instead of killing batch tasks, we would like to suspend them in favour of higher-priority stream tasks and resume them when free resources are available both minimising queueing latency and wasted work. This is exactly what we achieve with NEPTUNE, our solution described in Chapter §4. NEPTUNE's approach is depicted in the lower right part of Figure 2.7. It dynamically prioritises stream tasks by *suspending* running batch tasks, minimising queueing delays ($2T$ runtime for the stream job). When resources become available, batch tasks are resumed without losing progress, thus achieving high resource utilisation ($5T$ for application completion) without any checkpointing overhead.

Measuring queueing delay. To assess the effect of different scheduling policies on queueing and end-to-end latencies in practice, we also run a real-world stream/batch application using a cluster of 4 Azure E4s_v3 virtual machines with 4 CPU cores, 32 GB of memory, and 60 GB of SSD storage each. The implementation of the stream/batch application described above consists of a real-time inference and a historical training job in Spark. More details about the implementation can be found in Listing 3.1. For this experiment, we run the application with varying sizes of training jobs by increasing the number of files they consume from the NYTimes dataset, while a small subset of the dataset was used for inference [67].

Figure 2.8a shows the 95th percentile of queueing latency for the stream tasks of the application. The results indicate that Spark's default policies, FIFO and FAIR, cannot achieve low queueing delays for the latency-sensitive stream tasks as we increase the training load and, thus, resource utilisation. Ideally, we would like minimal queueing delays, even under high resource utilisation. As we show in §4.6, NEPTUNE can achieve minimal queueing delays by preempting lower priority tasks when needed, also leading to significant gains both in terms of end-to-end latencies and task throughput.

Figure 2.8b shows the end-to-end inference latency of the application over increasing training load. The results show that Spark's default scheduling policies cannot guarantee low end-to-end latency due to increased task queueing delays. Even under low training load, FIFO and FAIR yield latencies of several seconds for the upper percentile. In contrast, NEPTUNE results in sub-second end-to-end inference latency distributions even under high resource utilisation by efficiently suspending lower priority batch training tasks.

2.2.6 Data processing summary

To sum up, there are numerous dataflow frameworks for batch or streaming computation, which expose a variety of programming models and transparently provide scalability and fault-tolerance (§2.2.2). The recent demand for unified analytics, as described in §2.2.4, resulted in several unified APIs on top of those frameworks [7, 20, 150], such as Spark's Structured Streaming [9], Flink's Table API [49], or external layers such as Apache Beam [13].

However, as we experimentally demonstrated in §2.2.5, these systems only provide the logical abstractions for stream/batch jobs running on shared resources. Even though more critical jobs can be prioritised with static policies such as fair scheduling [147], this does not avoid queueing delays and thus cannot guarantee latency or throughput requirements. In Chapter §4, we propose NEPTUNE, a unified execution framework to support stream/batch applications that can efficiently share long-running executors. NEPTUNE is aware of explicit job requirements and can dynamically prioritise latency-critical jobs of these applications while effectively utilising cluster resources.

2.3 Cluster scheduling

After discussing the evolution of big data processing systems from single purpose to unified engines and describing the system mismatch in regard to modern application needs and existing job scheduling approaches, we focus on cluster management systems. Cluster managers such as YARN [135], Mesos [65], Borg [137] and Kubernetes [82] enable resource sharing across users, frameworks and applications in large compute clusters. They package resources such as CPU, disk and memory as containers that are allocated on-demand by applications. To reduce interference across containers running on the same machine, they usually offer performance isolation mechanisms such as cgroups [84].

In general, the role of a cluster manager is to place containers on cluster machines in a way that machines are efficiently shared, container and thus application performance is not hindered, and the majority of application- and cluster-level requirements are satisfied. However, meeting these goals is increasingly hard as cluster workloads grow richer, workload requirements become more diverse and clusters grow in size.

As we discussed in §2.1, a substantial portion of production clusters today is dedicated to LRA workloads. They are placed along batch analytics workloads in shared compute clusters to reduce operational costs (see Figure 2.1). As LRA containers run for longer durations compared to traditional batch applications, their placement is important and has a sustained impact on cluster resources. As we experimentally demonstrate below, support for LRAs in existing schedulers is limited and simple placement constraints such as affinity and anti-affinity that are supported by some schedulers today are necessary but not sufficient to satisfy their requirements.

In this section, we explore the new challenges that cluster schedulers, especially in production environments, face due to long-running workloads. First, we motivate the importance of LRA placement for the application performance (§2.3.1) and resilience (§2.3.2). Then, we describe particular types of global cluster objectives imposed by cluster operators (§2.3.3) and provide a summary of requirements for modern cluster managers (§2.3.4). Finally, we study the main system architectures that these cluster schedulers follow (§2.3.5).

2.3.1 Long-running application performance

To study the impact of container placement on application performance, we run a series of experiments on a 275-node pre-production cluster within Microsoft. Each cluster machine has with 8 CPU cores, 128 GB of memory, 3 TB of HDD storage and is connected on a 10 Gbps network. On the cluster we deploy various LRAs such as HBase, TensorFlow, and Storm while controlling their placement using constraints such as affinity, anti-affinity, and cardinality. We experiment with constraints targeting containers both within an application (i.e., intra-application constraints) and across applications (i.e., inter-application constraints).

Affinity. It is often beneficial to collocate the containers of an LRA on the same node or group of nodes as other containers, e.g., to reduce network traffic between communicating containers within the same or across different applications.

To observe the impact of *intra-* and *inter-application affinity* on application performance, we deploy a Storm (streaming) application on the 275-node cluster using YARN, a popular open-source cluster manager. The application identifies top-k trending hash-tags on Twitter over a 60-second sliding window using an input stream of 6,000 tweets per second, similar to the typical average number of tweets reported by Twitter [134]. The resulting hash-tags are then combined with user profiles loaded from Memcached, a key-value store that stores a total of two million user profiles. For the deployment, we use five supervisors for Storm application and a single instance for Memcached.

We compare three different container placement scenarios: (i) a container placement with no constraints (no-constraints); (ii) a placement where all Storm containers live on the same node (intra-only); and (iii) a placement where both Storm and Memcached containers live on the same node (intra-inter).

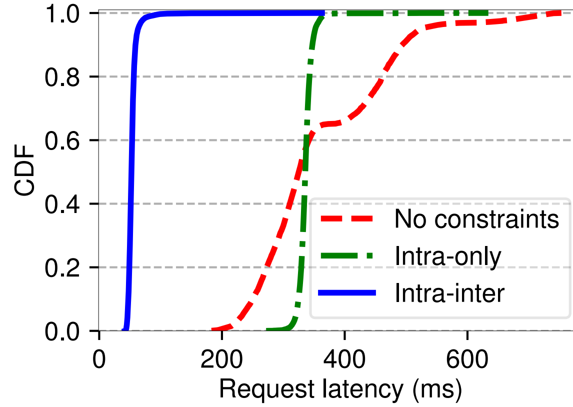


Figure 2.9 Memcached lookup latency with node affinity constraints

	Memcached latency (ms)			**Processing latency**
	mean	stdev	median	average (ms)
No-constraints	372.0	129.5	327.4	1100
Intra-only	361.7	96.0	337.0	755
Intra-inter	78.8	90.6	53.0	150

Table 2.1 Storm/Memcached latencies (Latencies include Memcached lookup and total processing.)

Figure 2.9 shows the latency distribution of the Memcached lookups, and Table 2.1 includes both the Memcached lookup latency and the total end-to-end latency of the Storm/Memcached topology.

Our results show that intra-only placement leads to an end-to-end latency improvement of 31% over no-constraints due to reduced network costs. However, as we can observe from Figure 2.9, this strategy cannot improve mean Memcached lookup latency. On the other hand, with intra-inter placement, we can reduce mean Memcached lookup latency by 4.6× over intra-only; end-to-end latency by 5× over intra-only and by 7.6× over no-constraints (see Table 2.1). Therefore, we conclude that both intra- and inter-application affinity constraints are crucial to achieve the best application performance.

Anti-affinity. To minimise resource interference between LRAs, it may be desirable to place the containers of an application on different machines through *intra-* and *inter-application anti-affinity*.

To validate the performance benefit of such constraints, we deploy 40 HBase instances with 30 region servers each, occupying 30% of the 275-node clusters' total memory. We use the YCSB benchmark [26] with a dataset of 1 billion records (1 TB) and submit six YCSB workloads through multiple clients to generate load for the HBase instances. To emulate a shared production environment, we also submit GridMix batch jobs [59] that use 60% of the cluster's memory.

We first compare the following placement scenarios: (i) a placement with no-constraints; and (ii) a container placement with anti-affinity constraints to avoid collocating region servers of the same or different HBase instances on the same node (anti-affinity). As Figure 2.10 shows, no-constraints achieves 34% lower throughput compared to anti-affinity, as it can lead to collocated region servers,

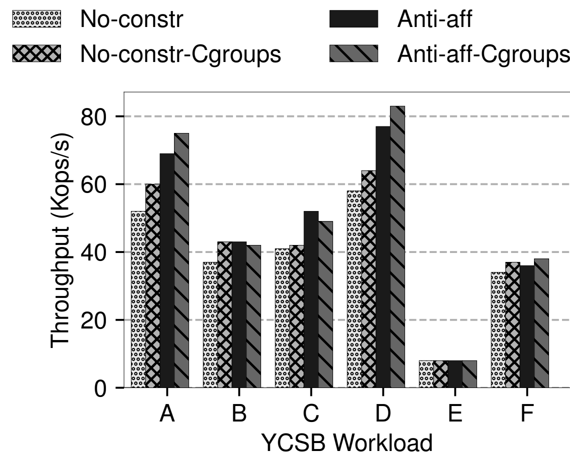


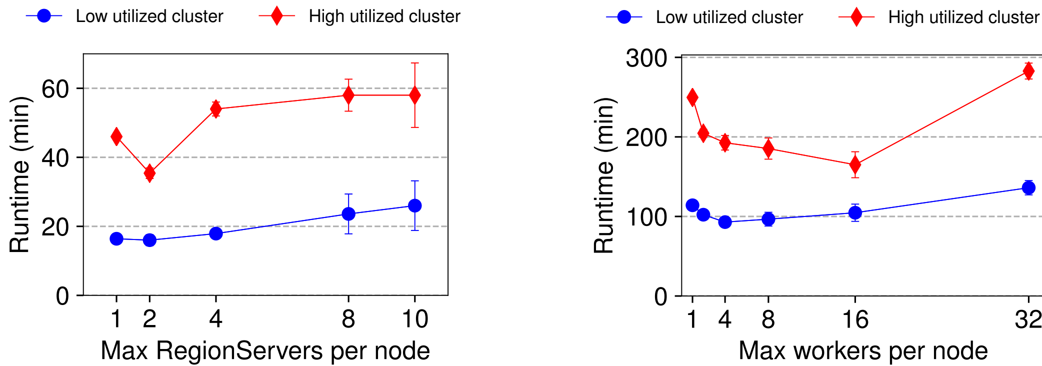
Figure 2.10 HBase throughput with node anti-affinity constraints

competing for CPU and I/O resources. Our experiment also reveals that no-constraints incurs increased tail latency compared to anti-affinity by up to 3.9× for the 99th percentile.

Furthermore, we repeat the above experiment using the *cgroups* [84] mechanism to assess whether resource isolation mechanisms are sufficient to improve performance, instead of using placement constraints. As shown in Figure 2.10, although cgroups improve the throughput of no-constraints by 20%, they cannot match the performance of anti-affinity. The isolation offered by cgroups cannot prevent interference between resources not managed by the OS kernel, such as CPU caches and memory bandwidth. Hence, anti-affinity is required for optimising LRA performance. Moreover, combining placement constraints like anti-affinity with resource isolation mechanisms such as cgroups leads to the highest performance gains.

Cardinality. The affinity and anti-affinity constraints represent the two extremes of the collocation spectrum where we either collocate everything or nothing. To strike a balance between the two, we experiment with more flexible *cardinality* constraints, which set a limit on the number of collocated containers on a node or group of nodes in the shared cluster. Even though finding the right cardinality is hard and depends on both the application and cluster state as we show below, in this experiment we want to see if applications can benefit from more flexible constraints and not how to automatically infer them.

Figure 2.11a reports the time required to run all YCSB workloads using 10 HBase region servers (RS) with full anti-affinity (i.e., 1 RS per node) to full affinity (i.e., all 10 RS on one node). Similarly, Figure 2.11b shows the time required for TensorFlow to complete a machine learning workflow with one million iterations using 32 workers, each time with a varying maximum workers per node. We again use GridMix for additional cluster load: 5% of the total cluster memory for the "low" utilised cluster in the figures and 70% for the "high" utilised cluster.



(a) HBase total runtime with cardinality constraints (b) TensorFlow runtime with cardinality constraints

Figure 2.11 Impact of cardinality constraints on application performance

In our experiments in the highly utilised cluster, we observe that collocating up to 16 TensorFlow workers on a node as shown in Figure 2.11b, reduces runtime by 42% compared to the affinity placement with a maximum cardinality of 32, and by 34% compared to the anti-affinity placement with a maximum cardinality of 1. A second observation is that the cardinality that leads to optimal runtimes can vary based on the specific application and the current cluster load. Indeed, in the TensorFlow experiment of Figure 2.11b, the most optimal cardinality value is 16 for the highly utilised cluster and 4 for the the less utilised one, while in the HBase experiment of Figure 2.11a, the optimal cardinality is 2 for the highly utilised cluster and either 1 or 2 for the low utilised cluster as they yield similar runtimes. Based on these results, we make the crucial observation that affinity and anti-affinity constraints, albeit beneficial, are not sufficient, and tighter placement control using cardinality constraints is required to achieve optimal application performance.

2.3.2 Long-running application resilience

Besides the impact on application performance as experimentally demonstrated above, placement constraints such as affinity, anti-affinity, and cardinality heavily affect application resilience in shared compute clusters. To study the effect on application resilience, we use data from a shared production cluster within Microsoft reporting the availability of machines over a period of four days.

Unavailable machines in large clusters of commodity machines are quite common. This is due to machine failures, scheduled maintenance, OS and application upgrades, or machines being re-purposed. For administrative reasons, cluster operators typically split clusters into: (i) *fault domains*, that are machines with a higher likelihood of joint failure (e.g., racks); and (ii) *upgrade domains*, which are machines that are scheduled to be upgraded together. In some production clusters within Microsoft, node groups called “service units” account for both upgrades and failures [72].

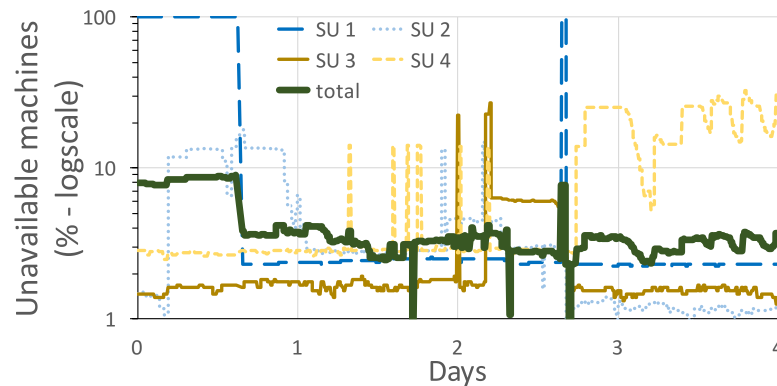


Figure 2.12 Unavailable machines in a Microsoft cluster (With total we represent the unavailability percentage over all machines, while SU1–SU4 are unavailability percentages over specific service units, i.e., logical node groups.)

Figure 2.12 shows the percentage of machine unavailability in a production cluster within Microsoft consisting of tens of thousands of machines (total) over the a day period. Figure also shows the percentage of machine unavailability within four random service units of the cluster (SU1–SU4), comprising a couple of thousand machines each. We observe that: (i) unavailability in a service unit is usually below 3% but can spike to 25% or even 100%; (ii) there is a strong correlation of unavailability within a service unit; and (iii) service units tend to fail asynchronously (e.g., when SU1 is 100% unavailable, total machine unavailability is only 8%).

As a result, deploying an application with a random placement plan on the cluster, could cause multiple container failures at once, which can in turn impact its recovery time or performance. Such a placement plan hurts LRAs in particular because their containers are by definition long-lived and thus the failure probability increases over time. Hence, application owners must be able to spread containers across fault and upgrade domains on the cluster using for example *anti-affinity* constraints across node groups.

Even though spreading containers is crucial for long running application resilience, applications owners should not be required to explicitly refer to specific nodes or domains (e.g., fault, or upgrade domains) when requesting containers as: (i) these domains change over time, e.g., with the addition or removal of nodes in these groups; and (ii) it is cumbersome to enumerate all domains of a cluster with thousands of machines. Moreover, in cloud environments, it may even not be feasible to expose the underlying nodes as the cluster operator will not be able to reveal the cluster configuration for security and business reasons.

2.3.3 Global cluster objectives

In addition to the above application requirements affecting performance and availability, cluster operators must also specify constraints (usually expressed as placement constraints) that guarantee the smooth operation of the cluster and are considered higher-priority than typical application constraints. Besides local application constraints, such as restricting the number of network-intensive containers per node, the scheduling of LRAs must meet *global objectives* targeting to:

- **Minimise constraint violations.** Satisfying the placement constraints of all applications may not be possible, especially in a heavily loaded cluster. In this scenario, instead of avoiding placement completely or ignoring placement violations, the number and the extent[1] of constraint violations should be minimised.
- **Minimise resource fragmentation.** It is undesirable to leave too few free resources on a node as they might remain un-utilised. There might be cases where resources left on a node are not even enough for a single container instance.
- **Balance node load.** By balancing the node load, applications can expand their allocated resources on a given node when needed and better accommodate unpredictable load spikes during runtime.
- **Minimise number of machines used.** A low number of machines used can be achieved by better resource consolidation (packing) and it can thus reduce the operating cost of a cluster in a cloud environment.

Note that some of the above objectives may be conflicting, for example minimising constraint violations and load imbalance, while others may be irrelevant in a specific scenario, for instance minimising the number of machines for an on-premises cluster. The cluster operator should be able to determine the objectives to be used and their relative importance. At the same time, supporting such global objectives should not affect the scheduling of traditional batch jobs, which are more sensitive to container allocation latencies due to their shorter container runtimes (see §2.1.2).

2.3.4 Cluster scheduling requirements

Based on our observations so far (§2.3.1–§2.3.3), we summarise and list the requirements a modern cluster manager should meet to effectively schedule long-running applications in shared compute clusters:

[1] Consider a constraint to place no more than 5 HBase containers on a rack. Placing instead 10 containers is a more extensive violation than placing 6.

- **[R1] Expressive placement constraints:** The cluster manager should support intra- and inter-application (anti-)affinity and cardinality constraints to express dependencies between containers during placement both within and across applications.
- **[R2] High-level constraints:** The application constraints must be *high-level*, in the sense that they should be agnostic of the cluster organisation, and capable of referring to both current and future LRA containers.
- **[R3] Global objectives:** The cluster manager should also be able to meet global optimisation objectives imposed by the cluster operator that can guarantee the smooth operation of the cluster.
- **[R4] Scalability and scheduling latency:** Although LRAs can tolerate higher scheduling latencies compared to batch applications, supporting LRAs should not affect the scheduling latency for containers of task-based applications or affect the scalability of the scheduling infrastructure.

Table 2.2 highlights the features of existing cluster managers with some support for LRAs in respect to our requirements and their architectures (see §2.3.5). We do not list cluster managers that do not provide support for LRAs. Most existing schedulers only provide rudimentary support for placement constraints [55, 65, 128, 135, 137] and MEDEA is our work fully supporting LRAs, described in Chapter §5. Existing schedulers typically rely on machine attributes to allow users to express affinity or anti-affinity constraints in an *implicit* manner: e.g., placement of two application containers on a node with a given hostname or on machines with a GPU/FPGA. Similar placement techniques have been used by schedulers to reduce network congestion and achieve better data locality [60, 73]. However, as these attributes are statically assigned to machines, they can not be used to specify expressive (e.g., inter-application, cardinality) and high-level (i.e., not relying on machine names) LRA constraints.

Condor [128] is one of the first schedulers supporting constraints. It provides a classAd mechanism that allows resource providers to describe declaratively each individual resource with expressions built on a collection of attributes e.g., disk, memory, or architecture. Using the same mechanism, applications can express their resource allocation preferences by submitting a classAd specifying *Constraint* and *Rank* expressions for desired resources. The former serves as a hard constraint (a filter on resources) while the later can be considered as soft constraint (the allocation will not fail if not satisfied), an expression of preference over individual machines. The matchmaking algorithm then evaluates each consumer classAd against each resource classAd and picks the highest ranked resource. Unfortunately, Condor's matchmaking algorithm relies on constraints expressed using a static set of attributes exposing the underlying infrastructure and lacking flexibility.

System	Type	[R1] Expressive constraints between containers					[R2] High-level constraints	[R3] Global objectives	[R4] Low-latency container allocation
		affinity	anti-affinity	cardinality	intra	inter			
YARN [135]	Two-level	✧	–	–	✧	–	–	–	✓
Slider [118]	Two-level	✧	✧	–	✧	–	–	–	–
Borg [137]	Centralised	✧	✧	–	✧	✧	–	✻	✓
Kubernetes [137]	Centralized	✓	✓	–	✓	✓	✓	✻	✓
Mesos [65]	Two-level	✧	–	–	✧	–	–	–	–
Marathon [89]	Two-level	✓	✓	✓	✓	–	–	–	–
Aurora [10]	Two-level	✧	✓	✓	✓	–	–	–	–
TetriSched [133]	Centralized	✧	✧	✧	✓	–	–	✻	✓
Firmament [57]	Centralized	✧	✧	✧	✓	–	–	✻	✓
MEDEA §5	Centralized	✓	✓	✓	✓	✓	✓	✓	✓

Table 2.2 Support for LRA requirements R1–R4 in existing schedulers (✧ indicates implicit support for constraints through static machine attributes and not by declaring explicit dependencies between containers; ✻ indicates a partially supported feature.)

More recent schedulers include YARN [135] that was initially designed for task-based jobs, and currently supports only affinity to specific nodes/racks [78]. Slider [118] and ongoing extension efforts [144] enable LRAs in YARN that supports only simple intra-application constraints.

Prophet [142] is another scheduler on top of YARN that improves LRA scheduling based on historical data, but requires jobs to be recurring. Aurora [10] and Marathon [89] add support for LRAs to Mesos two-level scheduler. They enable intra-application, (anti-)affinity and cardinality constraints (e.g., limiting the number of containers per node or rack), but by being external to Mesos, they follow an offer-based model that makes it hard to optimise for global cluster objectives (see §2.3.5).

Google's Borg [137] schedules both LRAs and task-based jobs. Borg also supports scoring of nodes during scheduling, which can emulate a restricted version of global objectives, however, only support for affinity to machines with specific attributes is mentioned. Kubernetes supports explicit intra- and inter-application high-level constraints between containers, but not cardinality, and lacks full support for global objectives by considering only one container request at a time.

Firmament [57] uses a graph-based approach similar to Quincy [71], formulating scheduling preferences of different jobs as a minimum-cost maximum-flow model. Firmament supports simple placement constraints such as affinity to specific machines. Adding more constraints would require (an exponential number of) additional vertices in the scheduling graph, increasing scheduling latency and limiting scalability.

TetriSched [133] supports various placement *and* time constraints, but the exact machines/racks of the placement have to be specified. TetriSched allows reservations of resources and introduces a new Space-Time Request Language (STRL) that enables jobs to express their preferences in space-time manner. Expressions in STRL are compiled and converted to an Mixed Integer Linear Programming (MILP) formulation used by the scheduling algorithm to place the entire cluster workload. This can be problematic under high load, as either the scheduling latency or the placement quality gets compromised.

2.3.5 Cluster scheduler architectures

Higher-quality container placement using sophisticated scheduling in general has proven to be essential for achieving higher cluster utilisation [31, 137], more predictable application performance [39, 153] and increased application resilience [117]. However, to satisfy the LRA requirements and global objectives described above, a cluster manager must be able to capture and support complex constraints as part of its scheduling logic that can significantly increase placement latency.

In reality, sophisticated cluster schedulers implement complex algorithms with single- or multi-dimensional resource fitting to support special constraints. Yet, the higher placement latency, which may be in the order of seconds, is unacceptable for particular types of workloads such as batch jobs

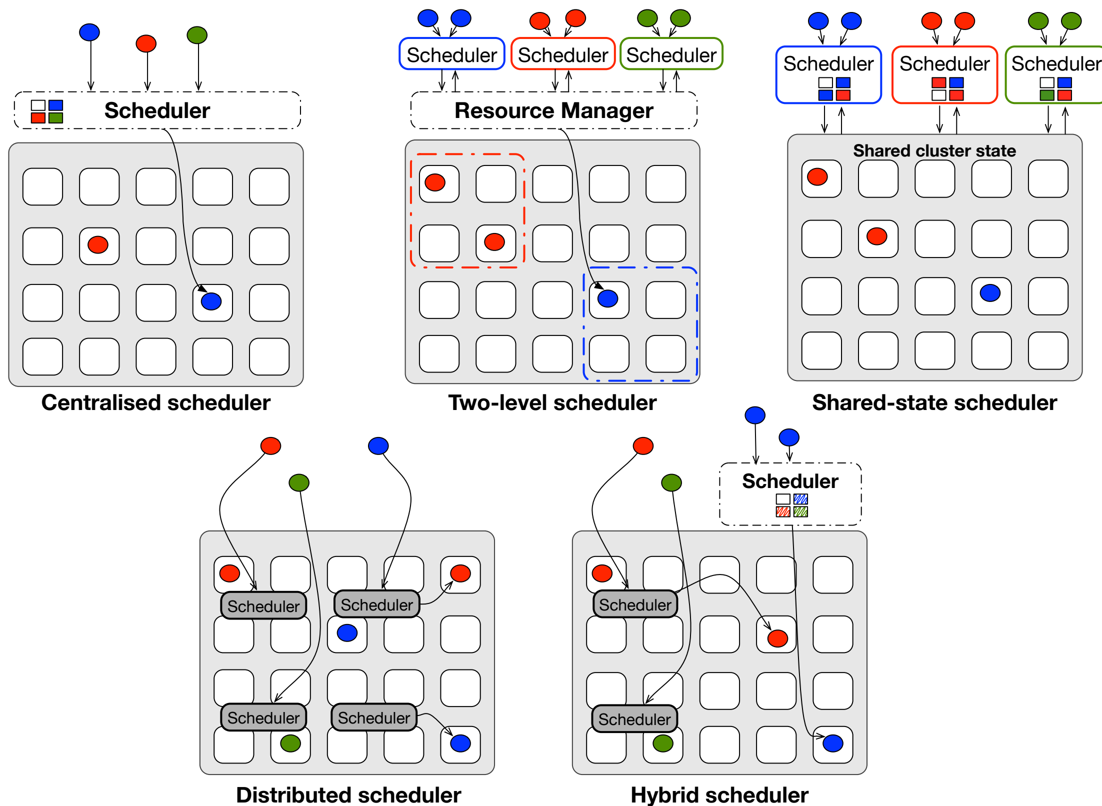


Figure 2.13 Overview of cluster scheduler architectures (Circles represent tasks/containers while colours correspond to different task types e.g., LRA, batch; white boxes represent cluster machines.)

with shorter container runtimes. As a result, prevailing cluster scheduler architectures today usually trade off between low scheduling latency and high quality container placement even though the ideal scheduler should achieve both. Figure 2.13 provides a high-level view of popular cluster scheduling architectures used by existing systems.

Centralised. In this architecture, schedulers use the same scheduling logic to choose placements for all workloads in the cluster. They maintain cluster state information in a centralised component including pending containers, running containers and machine utilisation. Centralised schedulers achieve high quality placement by implementing sophisticated scheduling algorithms and using up-to-date cluster state information.

However, complex scheduling algorithms can increase container placement latency and limit their scalability [100]. Therefore, existing approaches either use simple scheduling heuristics or early-termination techniques. For instance, Google's Borg [137] and its open-source variant Kubernetes [82], build complex multi-resource models for each node in the cluster but perform greedy scoring to choose the final placement across randomly sampled nodes, accounting for constraints and load balancing. In a similar manner, Facebook's Bistro [56] runs expensive scoring and path selection computations asynchronously but performs placement using simple heuristics to reduce placement latency.

TetriSched [133] and Alsched [132] are two centralised schedulers that use MILP to place containers in the cluster. They support complex placement and time constraints but they require the exact machines/racks to be defined as part of their constraints exposing the cluster infrastructure. As workloads have to be holistically placed using the MILP algorithm, when clusters grow larger or they are under high-load, scheduling with MILP can take tens of seconds. As a result, these schedulers follow an early-termination approach using the best placement they have thus far, trading placement quality for scheduling latency.

Other centralised schedulers such as Quincy [71] and Firmament [57] choose not to compromise on placement quality. They both consider the entire cluster workload when placing containers using a graph based approach. However, adding more constraints or nodes in their scheduling graph could lead to increasing scheduling latencies. Quasar takes into account both resource heterogeneity and interference during placement. Even though it can lead to near-optimal placement for long-running applications, it can also impact the latency of other shorter jobs [39].

Two-level. The two-level scheduling architecture utilises a simpler resource manager that allocates containers, and a number of application specific schedulers on top of it that are aware of their independent needs. The application schedulers implement the scheduling logic of workloads with diverse requirements such as batch, streaming and machine learning but these applications are only aware of their own allocated resources i.e., they do not have a view of the whole cluster state information (see Figure 2.13).

Apache Mesos [65] is a popular two-level scheduler, first developed to accommodate the deployment requirements of Apache Spark [150]. In Mesos, resources are still managed centrally but scheduling decisions are delegated to framework schedulers. Frameworks receive resource offers from the central component that can be either accepted or turned down. Framework schedulers can implement their own complex scheduling algorithms without affecting the latency or the scalability of other workloads running in the cluster. However, they are only aware of the resources they hold or they have been previously offered that can be problematic in cases in which, for instance, we are optimising for global objectives.

Shared-state. The shared-state architecture shares some similarities with the two-level schedulers as they were both designed to support independent and diverse workloads sharing compute clusters. Unlike two-level schedulers, they allow multiple application schedulers with different scheduling logic to have a complete view of cluster state, which is not always up-to-date.

Omega is such a scheduler introducing partially distributed shared state [117]. It supports multiple schedulers that can run simultaneously in the cluster with each one of them dealing with a fraction of the total cluster workload. They may follow different implementations and they use an optimistic concurrency control model to update shared cluster state with the latest resource allocations. Every scheduler can mutate the state by issuing concurrent transactions that can either succeed or fail.

Failures lead to extra resource allocation attempts (retries). As state can be out-of-date leading to several retries, this model works best for workloads that maintain their container allocation for reasonably long durations.

Distributed. The distributed architecture includes schedulers that want to achieve extreme scalability and low-latency allocations following a decentralised approach. Such distributed architectures lack state sharing or expensive coordination and leverage worker nodes that either pull tasks directly from distributed schedulers or maintain a queue of tasks locally to minimise the period that machines remain idle.

Sparrow is a scheduler following a fully distributed architecture [100]. It intentionally uses a simple scheduling logic that can choose container placements at scale utilising randomly sampled information. Apollo is another example of a distributed scheduler that maintains a local queue of tasks assigned to each node [18]. It estimates wait times for each node and lazily propagates node future resource availability to schedulers in the form of a wait-time matrix. However, distributed schedulers usually achieve poor quality placement as their simple scheduling logic is bound by partial and often stale cluster state information.

Hybrid. Finally, there are scheduling architectures that try to achieve the best of both worlds: the high quality placement of centralised schedulers and the low scheduling latency of distributed schedulers. They split cluster workloads across a centralised component and multiple distributed ones based on job requirements or job historical information (see Figure 2.13).

Mercury is a hybrid cluster scheduler that offers two types of containers, guaranteed and opportunistic [77]. Applications deployed using Mercury can use a mixture of guaranteed containers that incur no queueing delays and are never preempted, and opportunistic containers that land in a special node queue, are lower priority and provide no queueing delay guarantees. Yaq is a cluster scheduler built on top of Mercury that uses task runtime statistics to reorder worker queues and reduce short-running container latency [111].

The Hawk hybrid scheduler utilises historical data to assign workloads to schedulers. Long-running containers are higher priority and are assigned to the centralised scheduler in order to achieve better quality placement while short-running containers are assigned to one of the distributed schedulers [35]. To avoid possible queueing delays for shorter workloads, Hawk dedicates a portion of the cluster to short-running containers. Eagle is an extension of Hawk hybrid scheduler that utilises state probing techniques to improve the latency of the lower priority workloads by lazily propagating cluster state information [36].

It is clear that scheduler architectures trade off between placement quality and fast allocation decisions. Even the more advanced hybrid architectures such as Yaq [111] and Hawk [35] offer quality placement only to a part of the cluster workload based on historical information. Unfortunately, obtaining such historical runtime information has proven to be a non-trivial task [102] and only works if workloads

are recurring. As we show in Chapter §5, using a centralised architecture with a two-scheduler design, it is possible to scale and achieve both low allocation latency and high quality placement where it matters the most without relying on historical data.

2.3.6 Cluster scheduling summary

To sum up, there are several practical use cases of LRAs in production environments that require complex high-level placement constraints. In particular we observe that precise control of container placement is key for optimising LRA performance (§2.3.1) and resilience (§2.3.2) while simple affinity and anti-affinity constraints are not enough. Moreover, when placing containers, the cluster scheduler must achieve the global optimisation objectives defined by the operators that are important for the healthy operation of the cluster (§2.3.3).

Existing schedulers only offer rudimentary support to LRAs (see §2.3.4), following different scheduling architectures choosing between low allocation latency and high quality placement (§2.3.5). In Chapter §5, we propose MEDEA, a new cluster manager with a novel two-scheduler design that enables the placement of both long- and short-running containers. MEDEA can achieve both high-quality placement for LRAs and low placement latency for task-based jobs that are traditionally running in large shared production clusters consisting of thousands of machines.

2.4 Chapter summary

To conclude, in this chapter we introduced basic concepts and terminology related to long-running applications, a popular workload type utilising long-running containers running in big data clusters today. We traced the advancements of data processing systems, a popular type of LRAs, and we described their programming and execution model. We also focused on cluster management systems that are responsible for allocating resources to long-running applications in shared compute clusters and categorised them based on their architectures.

Using data and use-cases from a production environment within a large internet company, we experimentally demonstrated that expressive container placement is crucial both for LRA performance and resilience. We described how existing cluster schedulers offer limited support to LRAs and trade-off between allocation latency and placement quality. Experimenting with a data-intensive application running on a cloud environment, we showed that data processing frameworks that evolved to support the heterogeneous jobs of modern long-running applications fail to satisfy application requirements and effectively utilise cluster resources. Based on our observations, we described the requirements of modern data processing and cluster scheduling systems targeting the new era of long-running applications in shared compute clusters.

Chapter 3

Unified Dataflows with Placement Constraints

In the previous chapter, we discussed the mismatch between the growing class of long-running applications and the systems that support their placement and execution in shared compute clusters. Unlike traditional workloads with short-running containers, long-running applications require precise control of their container placements to satisfy their performance and resilience needs and global cluster objectives. At the same time, modern long-running dataflow applications that execute heterogeneous computation tasks within long-lived containers in the cluster need to capture and satisfy their diverse requirements at runtime.

In this thesis, we introduce a unified execution model for the new era of long-running applications with explicit placement constraints that enables developers to: (i) combine data-parallel computation with different execution requirements as part of the same application; and (ii) control the placement of their long-running containers in the cluster. This chapter describes the new abstraction called *unified dataflows with placement constraints*.

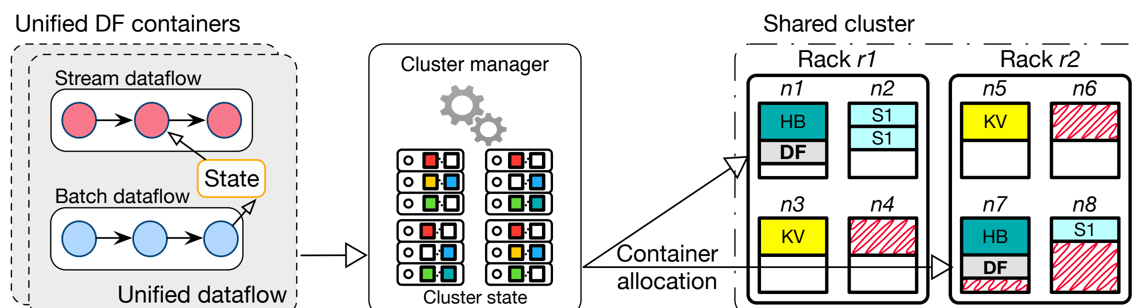


Figure 3.1 Overview of the unified dataflow model with placement constraints

A unified dataflow graph can represent complex applications that combine computation jobs with diverse requirements in a single dataflow. Figure 3.1 shows a stream dataflow and a batch dataflow as part of a single unified application. Unified dataflow graphs execute within long-running containers and also express their strict container placement requirements to optimise their performance and resilience. A cluster manager is responsible for their container placement in shared compute clusters. The goal of unified dataflows with placement constraints is twofold: (i) achieve *efficient* execution of dataflows within containers by taking into account their requirements during execution (see §3.2); and (ii) achieve *effective* placement across long-running containers in shared compute clusters by capturing container interactions during placement (see §3.3).

In this chapter, we present the unified dataflow model and explain with examples dataflow requirements and placement constraints. Our model is realised as part of two systems: NEPTUNE and MEDEA. Chapter §4 presents NEPTUNE, a novel execution framework for unified dataflow applications that can satisfy diverse job execution requirements within containers, and Chapter §5 presents MEDEA, a cluster manager with explicit support for complex, high-level placement constraints that can effectively capture interactions across containers.

We first provide an overview of the unified dataflow model with placement constraints (§3.1). We then define unified dataflows more formally (§3.2), describe their execution model (§3.2.1), and API (§3.2.2). Using a running example, we describe the semantics of our placement constraints (§3.3) as part of constraint expressions (§3.3.1) utilising container tags (§3.3.2), and node groups (§3.3.3). Finally, we discuss the limitations of our model in (§3.4)

3.1 Overview

A unified dataflow graph represents a unified application that consists of one or more *dataflow jobs*. Each dataflow job is a graph formed by *task elements* that express computation, represented as circles in Figure 3.1. Task elements receive collections of input data, also called dataflows, do some processing over the data, and produce output that is in turn sent downstream. Edges in the left part of Figure 3.1 represent the flow of the data between tasks. Dataflow jobs can also have *state elements*, representing the state of computation. State is depicted as a highlighted rectangle in the figure. A state element is associated with a specific task element in a dataflow job, but can be shared across tasks of different dataflow jobs that are part of the same unified application.

Unified dataflow graphs by design allow jobs to share computation logic and state without relying on external systems, however, different jobs of the same unified dataflow may have different latency and throughput requirements. Each job in a unified dataflow can be thus tagged with a *priority* representing these requirements. For example, in Figure 3.1, the stream dataflow in red represents a high priority job while batch in blue represents a lower priority one. Priority is automatically propagated to each

task element of the job (see §3.2). Priorities are then used by the runtime responsible for dataflow task execution to efficiently share resources across jobs depending on their needs.

Unified dataflow graphs are also designed to scale their computation as part of large shared compute clusters. To that end, jobs that are part of unified dataflow graphs, submit requests to a resource manager (or scheduler) to execute their computation tasks. The scheduler allocates cluster resources in the form of *long-running containers* that can be used for the execution of multiple task elements across jobs within the same unified application e.g., allocate 2 containers with 2 CPUs and 4 GB of RAM each.

To optimise application performance and resilience, unified dataflows require precise control of their container placements, and they achieve that by using complex constraints. Placement constraints can capture interactions between containers, both within and across applications in the cluster using the concept of *container tags* and *node groups* (see §3.3).

In Figure 3.1, a typical placement constraint would be to place `HBase` (`HB`) key/value store containers on the same node as the unified dataflow application (`DF`) which is a consumer of `HBase` records in order to reduce network traffic. Resource requests, along with placement constraints submitted by the application are handled by the cluster manager that can schedule applications with long-running containers and achieve high-quality placement.

3.2 Unified dataflows

Next, we formally define unified dataflow graphs, each consisting of one or more dataflow jobs, and each job forming a sub-graph of task and state elements.

Definition 3.2.1. Unified dataflow graph. A unified dataflow graph involves a number of dataflow jobs, d. Every dataflow job consists of task elements, c, expressing computation, and state elements, e, representing computation state. Dataflows, f, are edges between task elements, $f = (c_i, c_j)$ that propagate data. Within the same dataflow, a task element can be connected with a single state element e in a way that if $(c_i, e_j) \in F$, and $(c_i, e_k) \in F$, then $e_j = e_k$.

Every dataflow can be associated with a priority that is then propagated to its tasks. We define the priority function $\pi_d(c) \rightarrow \mathbb{N}$ where $\pi_d(c)$ is the priority of a specific task c of a dataflow job D. Using the priority function $\pi_d(c)$, we can compare the priorities of task elements across jobs within a unified dataflow which is necessary for satisfying their diverse requirements.

Example: A unified application consists of a stream dataflow d_1 with low latency requirements, and shares state with a batch dataflow d_2 with high throughput requirements. During unified dataflow execution, for any task element c: $\pi_d(c_{d1}) > \pi_d(c_{d2})$.

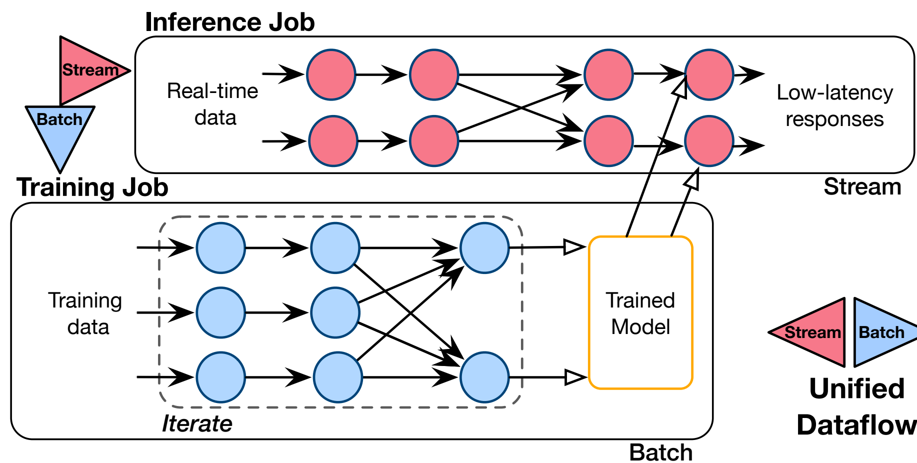


Figure 3.2 Unified dataflow application example for a real-time malicious behaviour detection service (A batch job trains a model using historical data; a stream job performs inference to detect malicious behaviour in real-time.)

3.2.1 Execution model

Figure 3.2 shows a unified dataflow application example for a real-time malicious behaviour detection service, as implemented in Listing 3.1. It consists of two dataflow jobs, a stream and a batch job, each with a number of task elements performing computation. The lower-priority blue batch tasks receive historical training data as input and perform model training, while the latency-sensitive red stream tasks receive real-time data from Kafka and perform inference to detect malicious behaviour.

The unified application maintains the trained model as state. The model is periodically updated by the batch job and is shared with the stream job that uses the latest model to perform real-time detection with the most up-to-date data. State sharing, along with application logic sharing, facilitates application development and management. In addition, by combining different job types as part of the same unified dataflow, cluster resources can be used more efficiently. Instead of being statically partitioned, they can be shared across jobs depending on their needs and their priority.

For efficiency reasons, instead of instantiating a new container for launching each task element, unified dataflows follow an *executor model* in which task elements are dispatched to long-running containers. A cluster manager is usually responsible for the allocation and placement of the executor containers on worker nodes in a shared cluster. Long-lived executors typically run for the entire lifetime of a unified dataflow application.

Listing 3.1: Unified application for real-time malicious behaviour detection

```
1  val trainData = context.read("malicious-train-data")
2  val pipeline = new Pipeline().setStages(Array(
3      new OneHotEncoderEstimator(/* range column to vector */),
4      new VectorAssembler(/* merge column vectors */),
5      new Classifier(/* select estimator */)))
6  val pipelineModel = pipeline.fit(trainData)
7  val streamingData = context
8      .readStream("kafkaTopic")
9      .groupBy("userId") /* some aggregations omitted */
10     .schema() /* input schema */
11 val streamRates = pipelineModel
12     .transform(streamingData) /* apply model to input */
13 streamRates.start() /* start streaming */
```

(Lines 1–6: Batch; lines 7–12: Stream)

3.2.2 API

Unified dataflows can be expressed using existing unified dataflow APIs such as Spark's Structured Streaming or Flink's Table API [9, 49]. To illustrate unified dataflows, we implement the application described above using Spark's Structured Streaming API. Listing 3.1 thus implements the real-time malicious behaviour detection service depicted in Figure 3.2, which is a production service at a popular e-commerce company [88]. The service is realised as a unified stream/batch application consisting of two dataflow jobs. A batch job that performs model training by iterating over the historical training data to reflect the latest changes in the `model` (or `pipelineModel`) which is kept as state in-memory (lines 1–6). The state is then shared in memory with the second job which is a real-time streaming job using the model for real-time prediction (lines 7–13). The streaming job computes a series of aggregations on the real-time Kafka input and then queries the model to detect malicious behaviour (line 12).

In terms of requirements, the output of the latency-sensitive streaming job must be computed with low latency to ensure fast reaction times. At the same time, the latency-tolerant batch job must run with high throughput to efficiently utilise the remaining resources and reflect the most recent state in the model, as the training data is continuously updated. In unified dataflows these requirements can be captured as priorities. Even though omitted from Listing 3.1 (as it not part of the standard Structured Streaming API), we could mark the batch priority as "low" (line 6), and the stream priority as "high"(line 13) using appropriate positive integer numbers. This will enable the execution engine of the dataflow to share resources across tasks depending on their needs.

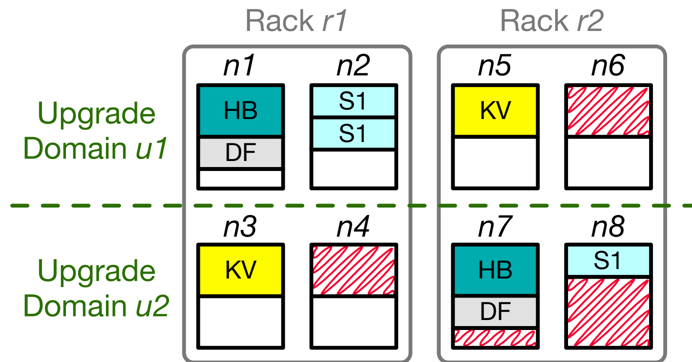


Figure 3.3 Container placement scenario, with a unified application (`DF`), HBase (`HB`), a key/value store (`KV`) and streaming job (`S1`)

3.3 Placement constraints

Placement decisions for long-lived executors or long-running containers in general, have a sustained impact on applications and cluster resources due to their long lifetimes. Thus, our model explicitly supports powerful placement constraints that capture container interactions. Container interactions are taken into account during placement by the cluster manager that is also responsible for maintaining a global view of all the active constraints in the cluster — more details about the constraint management are provided in §5.4. Our constraints rely on the notion of *container tags* (see §3.3.2) and *node groups* (see §3.3.3) to be expressive and high-level without requiring knowledge of (or exposing) the underlying cluster configuration (see §3.3.1).

Figure 3.3 shows a toy cluster consisting of 2 racks and 4 nodes per rack shared across 2 upgrade domains (groups of nodes that are upgraded together). `HBase` and `KV` are deployed key/value store applications with data partitioned across multiple containers in the cluster. `S1` is a streaming application with three containers spread across racks. The `HBase` application instance is using 2 containers, and is deployed along with a unified dataflow application, `DF`. The unified dataflow application instance `DF` is directly consuming data from `HBase` while resources used by other workloads in the cluster are marked with red.

In this simple scenario, some of the applications have specific requirements that can be translated to explicit container placement constraints. For instance, to lower the chances of multiple containers failing together and increase resilience, each container of the key/value store `KV` should be placed across different fault and upgrade domains (groups of nodes that fail or are upgraded together e.g., a rack), a typical *anti-affinity* constraint. To reduce network traffic and improve performance, all `DF` containers should be placed on the same node (collocated) as the `HB` containers from which they are consuming data from, a typical *affinity* constraint. To achieve low latency, no more than 2 latency sensitive streaming `S1` containers should be placed on the same node, a more generic *cardinality* constraint.

3.3.1 Expressing constraints

To accommodate such constraints, our model relies on *container tags* and *node groups*, both maintained centrally as part of cluster state (see §5.3.1). Container *tags* enable users to refer to containers in the same or different applications that are possibly not even yet deployed in the cluster (see §3.3.2). *Node groups* enable users to target specific node sets that can be either physical, such as a rack, or logical, such as a fault or upgrade domain (see §3.3.3). Using container tags and node groups together, application developers can define constraint expressions that are *high-level* and *expressive*. These expressions rely on placement constraints that use a single cardinality constraint type with a lower (c_{min}) and an upper (c_{max}) threshold to capture a variety of cases (e.g., affinity, or cardinality) both across and within applications running in the cluster. Even though in this subsection we focus more on the semantics of the placement constraints, §5.2 provides a detailed explanation of their syntax using several real-world examples.

Definition 3.3.1. Placement expressions. As part of the unified dataflow model, we support constraint expressions of the following form:

$$C = \{\texttt{subject_tag}, \texttt{tag_constraint}, \texttt{node_group}\}$$

where `subject_tag` is a tag or conjunction of tags that identifies the containers subject to the constraint (see §3.3.2); `tag_constraint` is a generic cardinality constraint of the form $\{\texttt{c_tag}, c_{min}, c_{max}\}$ where `c_tag` is a container tag (or conjunction of tags), and c_{min}, c_{max} are positive integers; and `node_group` is a node group (see §3.3.3). The `tag_constraint` can also be a boolean expression of multiple tag constraints — currently we do not support negation.

Example: In Figure 3.3, we can use a placement expression to specify constraints of the form: "place containers with tag `DF` of a unified dataflow application in the same `node` as `HBase` memory critical containers", like: $C_{example} = \{\texttt{df}, \{\texttt{hb} \wedge \texttt{memory_critical}, 1, \infty\}, \texttt{node}\}$

Definition 3.3.2. Placement constraints. The semantics of a constraint C is that each of the containers with `subject_tag` should be placed on a node belonging to a node set $\mathscr{S} \in \texttt{node_group}$, such that $c_{min} \leq \gamma_{\mathscr{S}}(\texttt{c_tag}) \leq c_{max}$ holds for the tag cardinality function of tag $\mathscr{S}$ as defined in §3.3.2. In words, *each* of the containers with `subject_tag` should be placed on a node of a particular logical set of nodes such that the cardinality of the `c_tag` is greater or equal to c_{min} and less or equal to c_{max}.

As we explain in more detail in §5.2, this generic cardinality constraint type is sufficient to express a wide range of intra- and inter-application constraints. Unlike existing approaches that rely on static machine attributes (see §2.3.4), our constraints are high-level and expressive utilising the abstractions of node groups and container tags that change as part of the application life-cycle. Our model also allows expressing more complex placements by combining arbitrary constraints with boolean

operators — for more specifics in regard to treating such complex constraints or resolving constraint conflicts see §5.3.2.

3.3.2 Container tags

Each container request r of an application is associated with a set of tags $\mathcal{T}_r$. Tags are a simple yet powerful mechanism for constraints to refer to containers of the same or different, possibly not yet deployed, applications in the cluster. For example, a constraint can use tag `hb` to refer to a current or future container of an HBase application.

More precisely, in our model we allow application owners to attach *tags* to *containers*, which is more expressive than existing models for container placement that attach attributes only to cluster *machines* (see §2.3). Without container tags, application owners would have to request specific *machines* that satisfy their requirements before even deploying their applications to the cluster which is cumbersome and in some cases not even possible (as in cloud environments).

Example: An HBase Master container can have the following tags: `appID:0023`[1] to denote the ID of the application; `hb` for the application type which in this cases is HBase; `hb_m` for the container's role which is master; and `memory_critical` as a resource specification.

Definition 3.3.3. Tag sets. Tags of running containers on nodes form sets that are called *node tag sets*. A tag set of a node $\mathcal{T}_n$ is the union of tags of the containers running on node n at a given moment. The node tag set is *dynamic* in the sense that when a container is allocated on node n, its tags are automatically added to $\mathcal{T}_n$ and when the container finishes execution, the associated tags are automatically removed.

Given that each tag in $\mathcal{T}_n$ can be associated with multiple containers on node n, we also define the *tag cardinality* function $\gamma_n : \mathcal{T}_n \rightarrow \mathbb{N}$, where $\gamma_n(t)$ is the number of occurrences of a tag $t \in \mathcal{T}_n$ on node n.

Example: Consider two HBase containers deployed on a node n_1: one master with tags $\{\texttt{hb}, \texttt{hb_m}\}$ and one region server with tags $\{\texttt{hb}, \texttt{hb_rs}\}$. Then, $\mathcal{T}_{n1}=\{\texttt{hb}, \texttt{hb_m}, \texttt{hb_rs}\}$, with $\gamma_{n1}(\texttt{hb})=2$ and $\gamma_{n1}(\texttt{hb_m})=\gamma_{n1}(\texttt{hb_rs})=1$.

Note that the fact that dynamic tag sets are more flexible, allows us to define a subset of a node tag set statically, e.g., to identify nodes with specific hardware capabilities, such as GPUs or FPGAs. Therefore, our tag model can also express the static machine attributes offered by existing container placement approaches [65, 135].

[1] To avoid naming conflicts, namespaces can be used in tags, such as the predefined `appID` that defines the namespace of application IDs.

Definition 3.3.4. Node-group tag sets. In a similar manner, to capture the associated container tags within a group of nodes (logical or psychical), we define the *tag set* $\mathscr{T}_S$ of a *set of nodes* S, such as a data-centre rack, or an upgrade domain, to be the union of tag sets of the nodes that are part of S.

Example: Let nodes n_1 (from the above example) and n_2 belong to rack r_1, and assume $\mathscr{T}_{n2}=\{\texttt{hb}, \texttt{hb_rs}\}$ with $\gamma_{n2}(\texttt{hb})=\gamma_{n2}(\texttt{hb_rs})=1$. Then, $\mathscr{T}_{r1}=\{\texttt{hb}, \texttt{hb_m}, \texttt{hb_rs}\}$, with $\gamma_{r1}(\texttt{hb})=3$, $\gamma_{r1}(\texttt{hb_m})=1$, and $\gamma_{r1}(\texttt{hb_rs})=2$.

3.3.3 Node groups

As we briefly mentioned above, nodes in the cluster are part of *node groups*, with the simplest predefined node groups being a `node` and a `rack`.

Definition 3.3.5. Node groups. Cluster operators can register node groups to specify logical, possibly overlapping, categories of node sets. A *node group* $\mathscr{K}_P$ is thus defined as a *set of nodes* P like a rack, or an upgrade domain, formed by the union of nodes that are part of set P.

A node set belonging to the node group `node` includes a single element that corresponds to a cluster node; a `rack` node set contains all nodes of a physical rack. Other node group examples are fault and upgrade domains, nodes that are failing together and that are being upgraded together respectively.

Example: In Figure 3.3, node sets $\{n1, n2, n3, n4\}$ and $\{n5, n6, n7, n8\}$ belong to the same rack node group, while node sets $\{n1, n2, n5, n6\}$ and $\{n3, n4, n7, n8\}$ belong to the same upgrade domain node group.

Node groups allow constraints to be expressed independently of the cluster's underlying organisation. They thus allow operators to not reveal their cluster infrastructure. For example, a constraint requiring to "place `hb` containers of the same application in different upgrade domains" does not need to be aware of the cluster's upgrade domains or perform any actions when upgrade domains change. Node groups also allow for more succinct constraint definitions—in their absence, the above constraint would have to enumerate all possible node combinations through a disjunction.

Node groups together with tags play a key role in enabling high-level constraints.

3.4 Discussion

The model of *unified dataflows with placement constraints* described in this chapter is generic, and can be realised as part of existing production systems as we explain in more detail in the subsequent chapters. We now discuss some of the limitations of our model.

Unified dataflow priorities. Unified dataflow graphs allow jobs to express their computation and latency requirements as priorities. These dataflow priorities can be used to effectively allocate resources to jobs during execution. Although our model relies on users to define dataflow priorities, by supporting just two distinct job categories, stream and batch, we can tackle well-known priority scheduling issues such as indefinite blocking, or starvation. For instance, in a unified application with a latency-sensitive stream job that can utilise all the available resources, a latency-tolerant batch job could be waiting for resources indefinitely. In Chapter §4, we describe a simple anti-starvation policy, guaranteeing the progress of jobs, that can lead to more fair resource sharing in scenarios where applications consist of both shorter- and longer-running jobs.

Even though operating system schedulers have been dealing with arbitrary priority levels across users and processes for decades [79], supporting more than two job priorities in the context of unified dataflows could be challenging. As dataflow jobs with higher priority levels need their share of resources, they can cause frequent switching (or preemption) across jobs with lower priority levels with the latter being blocked indefinitely. A solution to this problem is job ageing, where priorities of job tasks increase the longer they wait. However, as dataflow tasks are not as fine-grained as processes, they can spend significant time (several milliseconds) switching from running to suspended state, eventually ending up spending more time changing state than doing useful work. More importantly, an ageing mechanism causing frequent state switching would make it substantially harder to argue about job response times even for the higher-priority jobs that have strict latency requirements. This is the main reason we ended up using just two priority levels for our unified dataflow implementation, high and low, for stream and batch jobs respectively, reducing scheduling complexity while covering all of our production use cases (see §4.3).

Placement constraint expressiveness. Even though in our placement constraint model we wanted to make sure we can express all their practical use cases, there are some specific cases that can not be directly expressed. A particular limitation of our constraint model is that we cannot express constraints of the form "spread all spark containers across 3 racks", i.e., impose cardinality on the number of node sets rather than the number of containers per node set. However, as we explain in more detail in §5.2.1, given that we can indirectly achieve similar placements using the cardinality constraints i.e., place up to 3 spark containers on each rack, we decided to not complicate our syntax to support these constraints.

3.5 Summary

To sum up, this chapter introduced unified dataflow graphs with placement constraints, a new execution model that enables developers to combine diverse data-parallel computation as part of the same application and control the placement of their long-running containers in the cluster. Unified dataflow graphs with placement constraints capture: (i) dataflow execution requirements using explicit dataflow

priorities; and (ii) complex container placement constraint expressions using tags and logical node groups.

Implemented as part of an novel execution framework, unified dataflows can achieve *efficient* execution of computation tasks within long-running containers. Using application-specific preemption mechanisms an execution framework scheduler can dynamically prioritise tasks based on their requirements while efficiently utilising container resources (Chapter §4).

Exposing the expressive constraints of our model as part of a cluster manager can enable the *effective* placement of long-running containers respecting application requirements. Using a novel cluster scheduler that can capture complex container interactions during placement can lead to higher-quality container placement in shared compute clusters (Chapter §5).

Chapter 4

Neptune: An Execution Framework for Suspendable Tasks

In the previous chapter, we described *unified dataflow graphs with placement constraints*, an abstraction that represents execution requirements explicitly as part of the dataflow model and also captures the placement of containers with expressive high-level constraints. Similar to existing unified APIs exposed by modern dataflow frameworks [7, 13, 20, 41, 150], unified dataflows allow users to combine latency-sensitive stream and latency-tolerant batch jobs as part of the same dataflow. However, unlike existing approaches, by capturing execution requirements, they allow the dataflow engine to achieve efficient task execution within long-running containers.

This chapter focuses on the system aspects of unified dataflow graphs and more precisely on how to realise unified dataflows as part of a distributed execution framework. Supporting unified stream/batch applications with unified dataflows that share the same runtime presents several challenges. In particular, the stream jobs of these applications must be executed with minimum delay to achieve low latency, which means that batch jobs may have to be preempted. At the same time, the throughput of batch jobs should not be compromised in favour of the stream jobs that could potentially lead to batch job starvation. Finally, resources managed by the execution framework, usually in the form of long-lived executors, must be efficiently shared across jobs depending on their needs, maintaining high resource utilisation.

As discussed in §2.2.4, prior to unified stream/batch applications, stream and batch jobs had to be deployed separately in a cluster and cooperate with each other through general-purpose resource managers such as YARN [135] or Mesos [65]. This approach often resulted in: (i) low utilisation of resources when dedicating parts of the cluster to each system using static resource allocations [65, 135, 137]; (ii) loss of progress when having to kill batch jobs to prioritise stream ones using non-work preserving preemption [30]; (iii) or high and unpredictable preemption times when checkpointing the state of preemptable jobs [23, 37].

Opportunity. In this chapter, we argue that the fact that batch and stream jobs can now be part of a single long-running application and share the same execution runtime provides a unique opportunity that was not available when using multiple engines or separate applications. By employing application-specific preemption mechanisms and efficient scheduling policies, a scheduler can dynamically prioritise stream jobs without unduly penalising lower-priority batch jobs. Existing schedulers for stream/batch applications, such as Spark's FAIR scheduler [124], do not take advantage of this opportunity and only prioritise jobs based on static priorities (see §2.2.5). Static priorities may introduce unpredictable queueing delays for latency-sensitive tasks, a limitation that is well-known by the community [120, 121, 122].

Requirements. However, to adequately support unified stream/batch applications as part of the same runtime, an execution framework must address several challenges:

1. *Capture diverse requirements* of stream and batch jobs regarding latency[1] and throughput. First and foremost, users should be able to express their unified stream/batch applications using existing unified APIs and only use simple configuration to disseminate job requirements to the execution framework.

2. *Minimise queueing delays for latency-sensitive jobs*, even under high resource utilisation. When all resources are occupied, a streaming job may have to wait for other tasks to finish before starting processing, which leads to increased latency due to queueing delays. An execution framework should thus dynamically prioritise latency-sensitive tasks to avoid delays.

3. Ensure *high throughput and resource utilisation* despite the heterogeneity of job types. At the same time, the throughput of batch jobs should not be compromised in favour of higher-priority stream jobs. Lower-priority batch jobs should also make progress, efficiently utilising the remaining resources and avoid starvation.

In this chapter, we describe how NEPTUNE, a new execution framework for unified stream/batch applications addresses these challenges. In §4.1 we give an overview of NEPTUNE, while in §4.2 we describe how unified applications are expressed, scheduled, and executed. Subsequently, in §4.3, we introduce *suspendable tasks*, an efficient preemption mechanism for suspending tasks and resuming them without loss of progress. In §4.4, we discuss how suspendable tasks enable NEPTUNE to dynamically prioritise latency-sensitive jobs while achieving high resource utilisation using a novel scheduling policy and leveraging real-time cluster metrics. In §4.5, we describe NEPTUNE's implementation details on top of Apache Spark, one of the most widely used distributed dataflow frameworks today. Our experimental evaluation demonstrating the benefits of NEPTUNE on a 75-node Azure compute cluster follows in §4.6. Finally, in §4.7, we discuss NEPTUNE's limitations.

[1] In a dataflow system, latency is the time between receiving a new record and producing output for this record.

4.1 Overview

Similar to modern dataflow frameworks, NEPTUNE allows users to express unified application jobs in a variety of lower- and higher-level APIs that share the same runtime for their execution. Unlike existing dataflow frameworks, users also express their execution requirements by tagging specific application jobs as lower or higher priority using a simple job configuration property. We make the decision of treating job execution priorities as an explicit unified dataflow property that the data processing system is aware of and hence can use when making scheduling decisions. Unified applications along with their requirements are submitted to a centralised scheduler that builds an execution plan for each job and then executes its tasks. The scheduler in NEPTUNE also enacts policies that suspend and resume preemptable tasks according to their user-defined requirements and higher-level objectives such as balancing executor load. We now give an overview of NEPTUNE's main components.

API. NEPTUNE allows users to express unified stream/batch applications by combining latency-sensitive stream jobs with latency-tolerant batch jobs that are represented as unified dataflows. It supports a variety of APIs that users can leverage to express their computation logic, as we explain in §4.2.1. In addition to that, users can define their diverse job execution requirements by explicitly tagging dataflow jobs with a single line of extra configuration per job. The distinct execution requirements of the dataflow jobs, part of the unified dataflow, are used by NEPTUNE when scheduling (see §4.2.2) and executing (see §4.2.3) computation tasks within long-lived executors.

Runtime. NEPTUNE uses *suspendable tasks* to minimise the queueing delays for the latency-sensitive tasks of stream/batch applications during execution. Suspendable tasks are preemptable tasks that can preserve their work when interrupted without losing execution progress. They employ lightweight coroutines to pause and resume execution within milliseconds. Using coroutines, simple or more complex processing tasks can be implemented as suspendable tasks efficiently while being transparent to the user. In §4.3, we show how processing tasks can be implemented with coroutines and briefly describe the benefits of this mechanism over other approaches.

Scheduler. NEPTUNE uses a *centralised scheduler* to assign the suspendable computation tasks of a unified dataflow to *long-running executors*. The scheduler is aware of the diverse task execution requirements and enacts scheduling policies that trigger task suspension and resumption, which are executed locally on each executor. To improve scheduling decisions, the centralised scheduler in NEPTUNE also takes into account executor metrics describing the current state. Metrics are periodically transmitted to the scheduler by the executors in the form of heartbeats. In §4.4, we describe in detail how the locality- and memory-aware scheduling policy, our best performing scheduling policy, takes into account executor metrics to guarantee job requirements and satisfy higher-level objectives such as balancing cluster load.

4.2 Unified dataflow applications

In §2.2, we discussed how distributed dataflow frameworks allow users to develop scalable and fault tolerant applications. They simplify data processing by hiding the underlying complexity of parallel code execution and fault-tolerance. For application development, these frameworks usually allow the mixture of higher-level relational queries with lower-level functional or imperative code. Lately, we also observe an effort towards the unification of APIs. Unified APIs enable users to combine latency-sensitive stream jobs with latency-tolerant batch jobs as part of the same unified application using a single API. Unified APIs in existing frameworks include Spark's Structured Streaming, and Flink's Table API, as well as the external unified API of Apache Beam [13].

In this section, we describe how NEPTUNE leverages such APIs to represent stream/batch applications as unified dataflows while treating their execution requirements as first-class citizens. This enables the dataflow framework to utilise more evolved scheduling policies that can take into account task execution requirements and utilise resources more efficiently. Below, we describe how unified dataflow applications in NEPTUNE are expressed, scheduled, and executed.

4.2.1 Expressing stream/batch applications

NEPTUNE allows users to express unified stream/batch applications represented as unified dataflows using both lower-level and higher-level APIs. As NEPTUNE is implemented on top of Apache Spark, applications can be expressed with Spark's lower-level Resilient Distributed Dataset (RDD) API or its higher-level DataSet, DataFrame and unified Structured Streaming APIs (see §2.2.2). Using these APIs, users can define computation jobs that are part of the same application and share a common execution environment, also known as a *context*. The context maintains the connection to the application executors and it is used to submit application jobs and broadcast variables. Executors are responsible for running computation tasks that are scheduled for execution and not continuously running. At any point in time, each executor can only be assigned to a single active application context.

NEPTUNE also enables users to specify their execution requirements for a stream/batch application by tagging each dataflow job using a simple local job property. At job submission time, a job is configured with a `neptune.priority` property, which is set to a `high` or `low` value. For instance, the stream jobs of a unified dataflow are marked as `high` while the batch as `low`, expressing their execution priority. This and other local properties, for example to select a job resource pool/queue under FAIR scheduling, are inheritable variables, maintained at the job level and are automatically transmitted to each child stage and task. The property propagation is achieved through the common application context when submitting a job, which passes the properties down to the job scheduler that is responsible for stage execution.

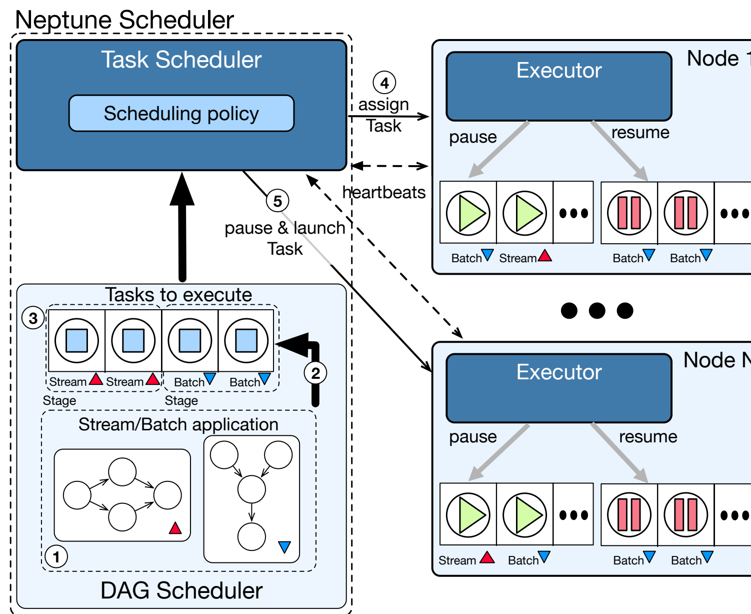


Figure 4.1 NEPTUNE architecture (Executors maintain separate queues for running and suspended tasks, and periodically send heartbeats to the central scheduler. The scheduler assigns tasks to executors and decides to suspend already-running ones according to its scheduling policy.)

4.2.2 Scheduling stream/batch applications

The NEPTUNE scheduler builds an execution plan for each dataflow job and also implements policies that suspend and resume tasks according to their requirements. The scheduler is comprised of two main components as shown in Figure 4.1: the DAG Scheduler and the Task Scheduler.

The DAG Scheduler computes the execution plan of each dataflow job in the form of a directed acyclic graph (DAG) (step 1). The compilation step that translates a user-submitted application to a set of DAGs is omitted from the figure. Each DAG vertex represents a stage, which groups tasks that can execute the same computation in parallel. During the execution of the application, the DAG Scheduler keeps track of the tasks that are already completed. When all data dependencies of a stage are met, it adds the tasks of that stage to a priority queue of tasks ready to be executed next (step 2). Within the task queue, tasks are ordered based on whether they are stream or batch, as dictated by their user-defined execution priority (step 3). For example, a stage of a latency-sensitive stream job is always placed before a stage of a latency-tolerant batch job in the queue. For stages that are not tagged by the user, NEPTUNE uses by default a FIFO ordering policy, prioritising tasks based on their submission time. However, a different ordering policy for these stages can be used instead, such as FAIR, by changing a configuration parameter.

Tasks that are ready-to-execute are then passed to the Task Scheduler. To execute tasks of stream stages without delay, tasks in NEPTUNE are implemented as *suspendable* (see §4.3). Any task that

is suspendable can be interrupted and later resumed with no loss of execution progress. When a new task arrives, it is immediately executed if there are sufficient free resources available in the executors (step 4). Otherwise, lower-priority batch tasks that are already running may be suspended, as dictated by the *scheduling policy* (see §4.4), to free up resources for higher-priority stream tasks to start their execution immediately (step 5).

To avoid losing the progress made by a preempted task, when a stream task terminates, the oldest suspended task is scheduled for execution. Suspended tasks in NEPTUNE are maintained locally on the same worker because using them on different workers would require task migration. This would incur the extra costs of transferring input data or intermediate outputs and would also require a policy to decide when and where to migrate. Since NEPTUNE preempts batch tasks (which are of arbitrary length) to prioritise the execution of stream tasks (which are typically short) and reacts in milliseconds to maintain high resource utilisation, it does not support task migration by design.

4.2.3 Executing stream/batch tasks

Task execution in NEPTUNE is performed by a set of executors with each one of them running as part of a long-lived container. Executors are also responsible for task suspension and resumption, as dictated by the job scheduler. A typical executor runs for the entire lifetime of an application, while under dynamic resource allocation mode, executors can be scaled up or down in a coarse-grained manner e.g., scale up when the application is in need for extra resources or scale down when executors are being idle for a long (configurable) period of time. Each executor has c CPU cores and can run up to c concurrent tasks. Periodically, executors also send heartbeats back to the scheduler with information about their task state and performance metrics such as memory usage, CPU utilisation, garbage collection, and disk spilling activity. As we describe in §4.4, this information can be used by the centralised scheduler to make more involved scheduling decisions.

Executors maintain two separate task queues, one for the running and one for the suspended tasks, as shown in Figure 4.1. For executing tasks, each executor has a thread pool that by default contains the same number of threads as the available cores on the executor. When a task is assigned to an executor, it is added to the running task queue and is immediately executed by a thread of the local thread pool assuming that there are available cores in the executor. Otherwise, when a stream task preempts a lower-priority running task, the executor first marks the batch task as paused, moves it from the running to the suspended queue, and prepares the new task for execution with low delay. As tasks complete and resources are freed, the scheduler resumes suspended tasks that reside in the suspended queue. For resumption, the oldest suspended task is picked to be resumed first. The executor then moves the tasks from the suspended to the running queue and continues computation without losing any execution progress.

4.3 Suspendable tasks

In the previous section, we discussed how NEPTUNE supports unified stream/batch applications as part of its' high-level system design and capture their execution requirements. To minimise execution delays for higher-priority dataflow jobs in a work-preserving manner, tasks in NEPTUNE are implemented as suspendable. Now, we describe the implementation details of these preemptable tasks and how existing computation tasks can be translated to suspendable tasks while being transparent to the user.

One of the main requirements of NEPTUNE is to minimise the queueing delay for stream jobs while having a low impact on lower-priority batch jobs. To achieve that, we need an efficient preemption mechanism that: (i) suspends and resumes tasks fast and without loss of execution progress; (ii) is transparent to users, i.e., does not force users to use specific APIs for their applications or to re-implement their computation jobs; and (iii) does not require a bespoke programming model, e.g., that requires users to trigger periodic checkpoints within tasks [44].

First, we considered a classical approach that would rely on thread synchronisation primitives for task suspension. In this approach, worker threads that execute tasks would use the *wait* and *notify* primitives to suspend and resume their computation. These thread synchronisation primitives typically rely on system calls. The actual preemption is performed by the OS scheduler, which moves threads between a wait and a running queue in kernel space. The wait call asks the OS scheduler to move the current task thread to the wait queue, while the notify call asks the scheduler to move one of the waiting threads from the wait queue to the run queue to be scheduled when possible. However, as in similar studies under different context [138], we noticed that, as the number of threads increases, thread synchronisation with OS involvement becomes a bottleneck and introduces extra scheduling delays (see §4.6.4).

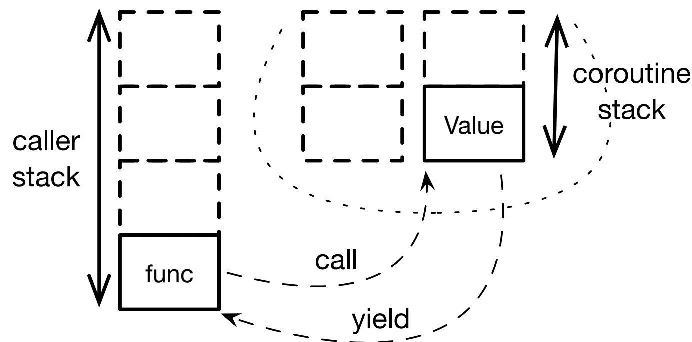


Figure 4.2 NEPTUNE coroutine mechanism (Caller initiates a coroutine using the `call` method. Coroutine is using a separate stack to store state and variables and can `yield` control back to the caller during execution.)

As a scalable and efficient mechanism for task suspension, we decided to use stackful *coroutines* instead [108]. Coroutines are resumable functions that generalise traditional functions or subroutines by having the ability to suspend during their execution and then resume later on without losing any progress. They can be used in combination with subroutines and thus selectively mark only parts of a program as suspendable. Similar to traditional functions, coroutines are invoked by a caller but have the ability to *yield* back values to the caller before completion, explaining why they suspended. A regular function without yield points can thus be considered a special case of a coroutine that does not suspend execution and returns to the caller only after completion.

Unlike traditional functions, coroutines maintain their own stack holding local variables that is separate from their caller's (caller stack) as shown in Figure 4.2. This stack allows separately written or nested coroutines to be able to call each other and yield back to the same caller. To create a new coroutine, the caller uses a special *call* keyword that creates a new coroutine instance with a new stack that can be automatically started. A started coroutine instance can then suspend execution at *yield points* that are part of its computation. When a coroutine suspends execution, it returns a *status handle* back to the caller, reporting the reason of the suspension. The status handle can be later used by the caller to resume execution from the previous execution frame without loss of progress.

Unlike checkpointing mechanisms used by existing systems to suspend execution [23, 37], coroutines can suspend tasks quickly (in milliseconds) because the state of execution is saved on the coroutine stack represented as arrays in memory (managed by the runtime). A coroutine call stack is a series of execution frames, each storing a pointer to the coroutine object, the position in the coroutine where the last yield occurred, the local variables and the return values. In fact, as the implementation is constrained by the JVM platform where arrays may contain either object references or primitive values (not both), coroutines in NEPTUNE use both a reference and a value stack to store coroutine descriptors, program counters and local variables based on their type. As mentioned above, stackful coroutines also support method composition, meaning they can call each other and return to the same

Listing 4.1: Collect coroutine task

```
1  val _collect_: (TaskContext, Iterator[T]) -> (Int, Array[T]) =
2    coroutine {(context: TaskContext, itr: Iterator[T]) => {
3          val result = new mutable.ArrayBuffer[T]
4          while (itr.hasNext) { /* iterate records */
5            result.append(itr.next) /* append record to dataset */
6            if (context.isPaused()) { /* check task context */
7              yieldval(0) /* yield value to caller */
8            }
9          }
10         result.toArray /* return result dataset */
11   }
```

resume site. This is important for implementing computation tasks composed of functions that call each other (e.g., shuffle tasks, as explained below). More importantly, coroutines can directly replace existing subroutine tasks by maintaining the same API, offering task preemption transparently to the users.

NEPTUNE uses stackful coroutines to implement *suspendable tasks*, which have a yield point after the processing of each record. As an example, Listing 4.1 shows the implementation of the `collect` task that returns all elements of a distributed dataset. Notice the yield point in line 7. The coroutine task receives the `taskContext` and a record iterator as arguments (line 2). In the iteration loop (lines 4–7), the new record is first appended to the result dataset (line 5). Every task is associated with an individual task context, which determines whether the task will be suspended, as dictated by the scheduling policy. If the task context is marked as paused, it returns a value back to the caller to express whether the task was suspended gracefully (line 7). Otherwise, the task continues execution. In a similar manner, coroutines can implement simple task logic, such as aggregations, or more complex logic, such as nested coroutine calls. Since yield points are only triggered sporadically when lower-priority tasks are marked as paused, they introduce a negligible overhead to the overall task computation time.

The component responsible for running the suspendable tasks is the executor. Figure 4.3 shows how an executor creates a more complex suspendable `ShuffleMapTask` compared to the collect task described above. A shuffle operation in general is redistributing the data of a distributed dataset so that they can be grouped differently across partitions. It is usually triggered when sorting or changing the grouping of data, and consists of `ShuffleMapTasks` followed by reduce tasks. A `ShuffleMapTask` writes out a shuffle file for every logical block (or partition) of the data across executors. Then, every

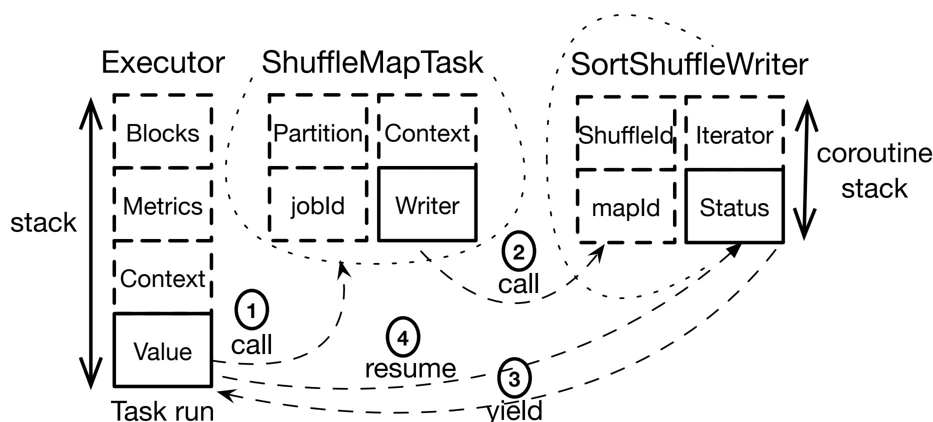


Figure 4.3 NEPTUNE coroutine composition (The executor runs a coroutine `ShuffleMapTask`, which instantiates a coroutine `SortShuffleWriter`. Both can yield to the executor when the `TaskContext` is paused.)

partition is fetched by a reduce task that will perform the shuffle aggregation (e.g., joining on a key). A suspendable `ShuffleMapTask` instance in NEPTUNE is created by invoking a *call* function (step 1). The suspendable task uses a coroutine stack to store its context, local variables, and state, including the `jobId` and `partition` needed for the functionality of the shuffle computation. Depending on the number of output partitions, type of aggregation and serializer, the `ShuffleMapTask` internally calls a `Writer` method either from a serialized sorted (i.e., `BypassMergeSortShuffleWriter`) or deserialized and sort (i.e., `SortShuffleWriter`) ShuffleWriter class (step 2). The `Writer` is another coroutine with a separate stack storing its local variables, and state, including the `ShuffleId`, `mapId` and the record `iterator`. The `Writer` coroutine can return to the same caller as the `ShuffleMapTask`. Once launched, the `Writer` instance executes up to a point and then receives a *pause* call from the executor (omitted from the figure). The `Writer` then *yields* a value back to the executor, indicating the reason for its suspension (step 3). The executor can then *resume* the same task instance (up to the next yield point or to completion) or invoke another function (step 4).

The above is a typical example of function composition that can be expressed with stackful coroutines. Coroutine composition enables both the calling and called coroutines to yield back to the same base caller, adding multiple suspension points. This functionality is necessary for suspending complex tasks like the ones needed for dataflow computation.

4.4 Locality- and memory-driven scheduling

The previous section introduced suspendable tasks, a mechanism that can provide efficient task preemption. In this section, we describe the logic controlling this mechanism, the scheduler component responsible for deciding *which* suspendable tasks to preempt and *when*. These scheduling decisions are crucial to: (i) achieve low latency for stream tasks i.e., guarantee that stream tasks can bypass batch tasks; (ii) cause minimum disruption to batch tasks; (iii) satisfy task locality preferences; and (iv) balance cluster load.

As NEPTUNE builds on top of Apache Spark, it also inherits its execution model. An application submitted for execution in Spark maps to a single driver process and a set of executors that are distributed across nodes in a cluster. The driver translates the application jobs to stages of tasks based on their dependencies and data locality preferences while it is also responsible for formulating an execution plan for those tasks. The executors perform work by running computation tasks, and they also store cached (task) data locally. The task scheduler component of the driver launches tasks on executors based on their available resources as soon as their dependencies are met. Tasks are operating on micro-batches, meaning that they processes larger streams of data as a series of smaller batches of data. Scheduling decisions in Spark are limited to launching a task (or stage of tasks) when there are enough available resources (and task dependencies are met), and killing a task when there is a failure we can not recover from. To keep the execution state up-to-date and make progress, executors notify

the scheduler about task completion or failure asynchronously while they also send usage metrics periodically back to the driver in the form of heartbeats.

NEPTUNE extends the scheduling logic described above with decisions about which tasks to suspend and when as the tasks we introduced are suspendable. Moreover, to improve such decisions, it captures a plethora of executor metrics that are now stored over configurable time windows (not just the latest values), as described in §4.4.1. The locality- and memory-aware (LMA) policy, our best performing scheduling policy, uses these metrics to avoid task suspension on executors with high-memory pressure and improve task performance, as described in §4.4.2.

4.4.1 Metric windowing

In NEPTUNE, the centralised scheduler component receives executor metrics via periodic heartbeats that capture their state including the status of the tasks running within executors. Heartbeats are reported at configurable time intervals, determined by the user using an application-level configuration parameter. Typically, a few seconds interval is recommended to capture the most up-to-date metrics, but this value may increase with cluster size to avoid heartbeat congestion. When a heartbeat reaches the scheduler, the executor measurements are added to the appropriate metrics window following a validation step checking for incorrect or missing values.

From the executor side, a heartbeat triggers the collection of all the metric measurements as exposed by the executor metrics provider service. At any point in time, an executor metric yields a single value, with some values being cumulative, and others offering a snapshot view at the time of collection. For example, the heap memory usage is the amount of memory used by the executor at that specific point in time. On the other hand, the time spent doing garbage collection is the time spent doing garbage collection since the launch of the executor process. For cumulative values, we also perform a metric unrolling step in order to determine the change since the last recorded measurement at the scheduler side.

Once metrics are captured and unrolled they are stored per executor in a metrics window data structure. Each executor metrics window maintains a sliding-window per metric of the last w valid values, with w also being configurable. The metrics data structure simplifies operations over sliding-windows such as exposing the moving average of each metric. The scheduling policy then utilises such operations to prioritise executors based on, for instance, memory-pressure, using the percentage of memory used or the percentage of time spent doing garbage collection on each executor.

Metric challenges. Capturing metrics can also be challenging as metric values can occasionally be incorrect or inconsistent. Figure 4.4 shows how a *complex metric*, reporting executor task duration, can occasionally report decreasing values. The duration of a running task is the amount of time spent in computation. Time periods, such as the task deserialization should not be included in the task

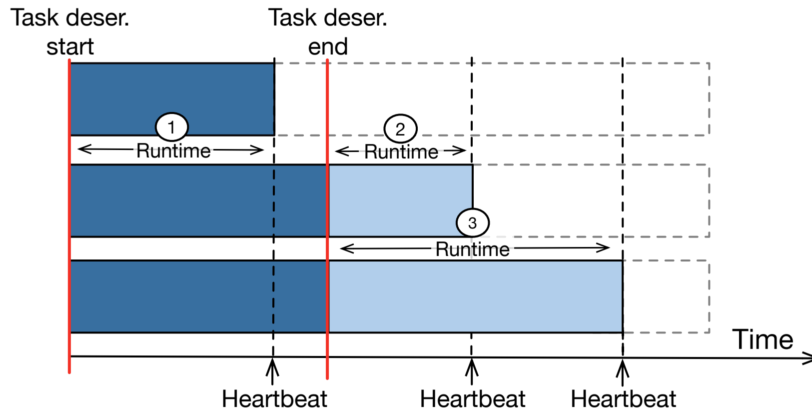


Figure 4.4 Task runtime metric captured over time (The executor reports inconsistent values between the first and second heartbeats as task runtime depends on deserialization time.)

duration. The deserialization time is the amount of time taken to unpack task dependencies before starting the actual computation.

As shown in Figure 4.4, in the event of a heartbeat received while deserialization is still happening (step 1), the duration of the task will be reported as the difference between the heartbeat's timestamp and the task deserialization start timestamp. The reason is the executor has not finished deserializing the task and thus does not know how much time it should consider as duration. At the arrival of a second heartbeat (step 2), the task has already been deserialized and has started computation. However, the reported duration of the task may be lower than the duration value of the first heartbeat, as the executor will now deduct the task deserialization time. This can lead to cases where the executors report decreasing task durations that is not feasible. This could potentially lead to wrong scheduling decisions if, for example, we use task runtime as an indication of executor load. To avoid such inconsistencies without extra complex logic on the executor side, we address the issue by ignoring decreasing values of the specific metric in the scheduler validation step. This allows values to eventually catch up, which in the above example will occur when the third heartbeat is reported (step 3) and the task duration becomes greater than the first heartbeat.

4.4.2 Scheduling policy

In NEPTUNE, scheduling policies can take advantage of the executor metrics stored over configurable time windows when deciding *where* (in which executors) to preempt tasks. By default, NEPTUNE uses the LMA (*locality- and memory-aware*) scheduling policy described below to less aggressively preempt tasks on executors with high memory-pressure. As suspendable tasks use coroutines that keep state in memory, they can cause increased memory pressure, which may have an impact on task and thus application performance [101].

Algorithm 1: LMA scheduling policy

```
   Input: List Executors,            // In descending preference order
 1          List ExecutorsMetricsWindow          // Executor metrics
 2 Upon event HEARTBEAT hb from EXECUTOR e do
 3 |  ExecutorsMetricsWindow[e].push(hb.metrics)
 4 |  Executors.updateOrdering
 5 return
 6 Upon event SUBMIT Task t do
   |  // Cache local executor
 7 |  Executor t_exec ← hostToExecutor.get(t.cacheLocation)
 8 |  if t_exec is None or t_exec.freeMemory < threshold then
   |  |  // Executor with the least pressure
 9 |  |  t_exec ← Executors.head
10 |  if t_exec has availableCores then
   |  |  // Launch task t on free cores
11 |  |  t_exec.Launch(t)
12 |  else
   |  |  // Suspend task t_p and launch t
13 |  |  Task t_p ← t_exec.tasks.filter(LowPriority)
14 |  |  t_exec.PauseAndLaunch(t, t_p)
15 return
```

In more detail, users develop stream/batch applications, represented as unified dataflows with several computation jobs. As described in §4.2.2, when a new job stage of a dataflow is submitted, it is first added to a queue of tasks pending execution. Once the stage dependencies are met, a list of tasks is submitted to run. Batch tasks run immediately as long as there are enough free resources on the executors, otherwise they wait for resources to become available. For stream tasks, NEPTUNE's scheduler uses the LMA scheduling policy, outlined in Algorithm 1.

The locality- and memory-aware (LMA) scheduling policy primarily considers the stream task's locality preferences[2] (shown as `cacheLocation`) and the executors' memory conditions (stored as part of `ExecutorsMetricsWindow`). To detect executors with high memory pressure that may have a negative impact on task performance, it uses metrics such as memory usage, disk spilling, and garbage collection activity. These metrics are stored per executor and based on their values the executors are sorted in ascending order. For each of the above metrics, if the mean value on executor A is lower than the mean of the same metric on executor B, we will prefer the former. Memory usage is the metric that is representing the latest state of each executor and is thus considered first, while disk spilling and garbage collection metrics are considered second as they describe activities that happened in the past. These metrics are collected periodically through the heartbeats between the executors and

[2]These are locality preferences to nodes that already hold locally the data that will be used for computation.

the scheduler (line 2) and are smoothened over configurable windows (line 3). When a heartbeat is received, it triggers an update on the metrics data structure that maintaining a preference order of executors to be used for preemption as explained below (line 4).

When deciding where to place a task, LMA first checks whether the task has a preference to a specific executor. If so, and the executor's free memory is above a threshold (we use twice the task's memory demand to make sure there is sufficient memory in the executor), this executor is picked; otherwise, LMA selects the executor with the least memory pressure (lines 7–9). If the selected executor has available CPU cores, the algorithm sends a *launch* message to the executor (line 11); otherwise, it selects a batch task to preempt (line 13), and sends a *pause & launch* message (line 14). This message suspends the batch task and starts the new one in a single network round trip.

In a unified dataflow with a large number of higher-priority stream tasks that can fully utilise the executors, the lower-priority batch tasks could completely starve waiting for resources. To prevent low priority jobs from being suspended indefinitely, NEPTUNE also implements a simple *anti-starvation* mechanism (omitted from the pseudo-code). For each task, NEPTUNE tracks the number of times that it has been suspended. If the task has been paused more than a given number of times, it is run uninterrupted for a period of time, ensuring progress of every stage. Note that the same mechanism, along with application-specific knowledge, can be used to bound the delay incurred by important batch tasks, such as tasks that update shared state.

4.5 Implementation

In this section, we describe how our unified dataflow execution approach can be realised in a system implementation as part of an existing distributed data processing system.

We implemented NEPTUNE by extending Apache Spark version 2.4.0, one of the most widely-used data processing frameworks, while our changes also apply to more recent Spark versions that maintain the same architecture (e.g., Spark 3.0). Suspendable tasks in NEPTUNE are implemented as coroutines using an open-source Scala library-extension: https://github.com/storm-enroute/coroutines. Our changes are in the Spark scheduler and execution engine, while NEPTUNE's code is publicly available as open-source at: https://github.com/lsds/Neptune. NEPTUNE's integration with Spark is depicted at Figure 4.5 with the modified Spark components and the new ones in blue.

Spark includes a `Driver` that runs on a master node and an `Executor` on each worker node, all running as separate processes. The `Executor` uses a thread pool for running tasks. An application consists of stages of tasks, scheduled to run an independent set of executors that run tasks and store data for that application. Every application maintains a unique `SparkContext`, the entry-point for job execution. In a specific SparkContext, multiple parallel jobs can run simultaneously if they are submitted from separate threads.

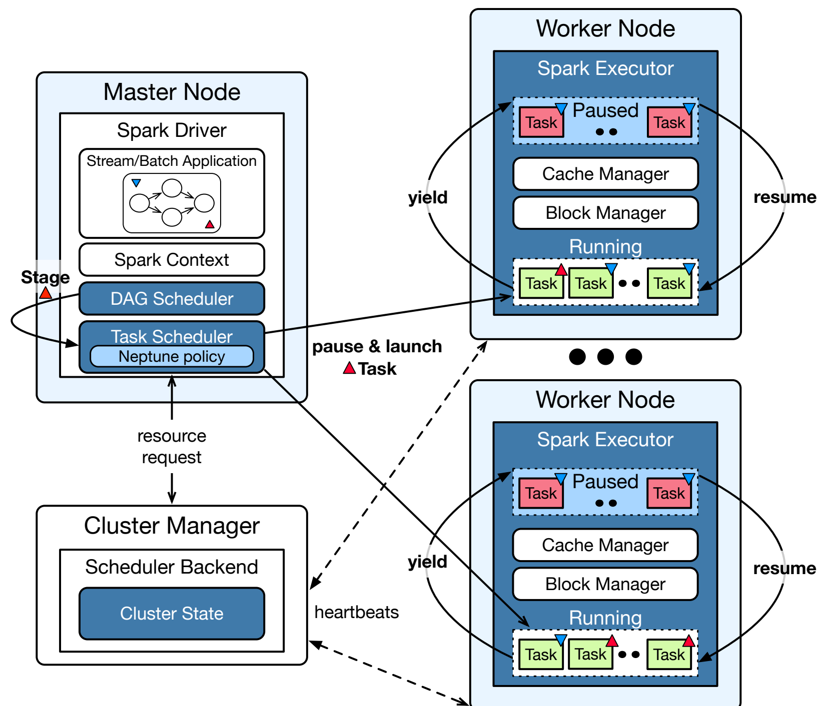


Figure 4.5 NEPTUNE integration in Spark (As part of the master node, the NEPTUNE scheduler assigns tasks to worker nodes. On each worker node, NEPTUNE maintains a queue of running and suspended tasks.)

DAG scheduler. On application submission, the application's `SparkContext` hands over a logical plan to the `DAGScheduler`, which translates it to a directed acyclic graph (DAG). Each DAG is split into stages at shuffle boundaries. The `DAGScheduler` maintains which RDDs and stage outputs are materialised, determines the locations for running each task, and decides on an execution schedule. NEPTUNE extends the `DAGScheduler` to allow stages with different requirements, expressed as priority levels (see §4.2.1). NEPTUNE currently supports two levels, high and low, for stream and batch jobs, respectively.

Task scheduler. Each DAG stage consists of a set of tasks and these tasks are submitted for execution by the `TaskScheduler`. New tasks ready for execution are serialized and sent to workers using an RPC. NEPTUNE extends the `TaskScheduler` to support the suspension of running tasks and the launch of new ones (through a *pause & launch task* message).

Scheduling policy. NEPTUNE uses Spark's existing policy for scheduling batch tasks and the LMA policy, described in §4.4.2, to determine where to place stream tasks and when to suspend running batch tasks. In the LMA policy, stream tasks suspend batch tasks based on their cache locality preferences and also avoid preemption on executors with high memory pressure based on real-time cluster metrics. As tasks terminate and the paused task queue in a worker node is not empty, the oldest suspended task is resumed, as long as there are no stream tasks pending.

Metric name	Description
1. JVMHeapMemory	The memory available in the JVM heap at runtime
2. JVMOffHeapMemory	The memory available in the JVM off-heap space and is not subject to garbage collection
3. OnHeapExecutionMemory	The portion of the reserved part of the JVM heap which is currently used to store execution information
4. OffHeapExecutionMemory	The portion of the reserved part of the JVM off-heap space which is currently used to store execution information
5. OnHeapStorageMemory	The portion of the reserved part of the JVM heap which is currently used for storage memory
6. OffHeapStorageMemory	The portion of the reserved part of the JVM off-heap space which is currently used for storage memory
7. OnHeapUnifiedMemory	The sum of OnHeapExecutionMemory and OnHeapStorageMemory
8. OffHeapUnifiedMemory	The sum of OffHeapExecutionMemory and OffHeapStorageMemory
9. DirectPoolMemory	The memory shared between the OS and the JVM so that objects are not copied into the JVM (a costly operation)
10. MappedPoolMemory	The memory which is occupied by files used by the JVM
11. ProcessTreeJVMVMemory	The virtual memory a JVM process believes it has controls
12. ProcessTreeJVMRSSMemory	The physical memory a JVM process currently controls
13. ProcessTreePythonVMemory	The virtual memory a Python process believes it controls
14. ProcessTreePythonRSSMemory	The physical memory a Python process currently controls
15. ProcessTreeOtherVMemory	The virtual memory non-JVM-or-Python processes believe they control
16. ProcessTreeOtherRSSMemory	The physical memory non-JVM-or-Python processes currently control
17. MinorGCCount	The cumulative count of how many times a minor garbage collections
18. MinorGCTime	The cumulative time taken by all minor garbage collections to this point
19. MajorGCCount	The cumulative count of how many times a major garbage collections
20. MajorGCTime	The cumulative time taken by all major garbage collections to this point
21. ExecutorGCTime	The amount of time finished executor tasks spent collecting garbage
22. ExecutorDuration	The amount of time finished tasks ran for
23. ExecutorAndTasksGCTime	The amount of time finished tasks, currently running and paused tasks spent collecting garbage
24. ExecutorAndTasksDuration	The amount of time finished tasks ran and paused for
25. ExecutorAndTaskMemory BytesSpilled	The number of bytes from main memory spilled by finished, currently running and paused tasks
26. ExecutorAndTaskDisk BytesSpilled	The number of bytes actually written out to disk by finished, currently running and paused tasks
27. OnHeapTotalMemory	The total size of on heap memory available

Table 4.1 Summary of all the executor metrics provided in NEPTUNE. Some of the metrics provide a cumulative and others provide a snapshot value.

Cluster state. Our LMA scheduling policy (see §4.4.2) requires additional information from each executor such as disk spilling and garbage collection activity to take memory pressure into account. NEPTUNE augments Spark's cluster state by having executors piggyback such metrics onto the heartbeat messages. A detailed list of the metrics captured in NEPTUNE is provided in Table 4.1. On the first column of the table we list the names of the provided metrics, while on the second column we provide a description of the metrics, reporting either a cumulative or a snapshot value. Framework's cluster manager, that is part of the centralised scheduler, maintains a sliding-window moving average per metric.

Executor. On a new task launch, an executor deserializes the task content that is a stream of bytes, creates a `TaskMemoryManager`, initialises a runner, and finally executes the task on a thread pool. NEPTUNE extends the executor with an extra suspended task queue and the transition logic between task states. A pause & launch call suspends a running task and marks the `TaskContext` as paused, yielding in the next record iteration. The executor then adds the task in the paused queue and frees the unused resources from the `TaskMemoryManager`. For cached data partitions and intermediate shuffle outputs, executors provide a local cache via the `CacheManager` and the `BlockManager` components. The `CacheManager` is responsible for caching tables and query results for subsequent executions. It utilises the `BlockManager` that acts as a lower-level local cache providing interfaces for putting and retrieving blocks of data both locally and remotely. These blocks may represent partitions, intermediate shuffle outputs, or broadcasts.

Suspendable tasks. A task is the smallest unit of execution associated with an RDD partition running on the executors. Every task in Spark, and consequently NEPTUNE, has the notion of locality, inherited by the implementation of the underlying RDD. For example, when using HadoopRDDs to read data from HDFS, the locality preferences are based on the nodes where the HDFS blocks reside. A task can be either: a `ResultTask` that executes a function on the RDD partition records and sends the output back to the driver; or a `ShuffleMapTask` that computes records on the RDD partition and writes the results to the `BlockManager` for use by later stage tasks. To support suspendable tasks in a way that is completely transparent to the user, NEPTUNE re-implements all basic `ResultTask` and `ShuffleMapTask` functionality in Spark across programming interfaces using coroutines (see §4.3).

4.6 Evaluation

In this section, we evaluate NEPTUNE, the prototype implementation of our execution framework for unified stream/batch applications. NEPTUNE is based on our new execution model, unified dataflows, a new model to perform efficient execution of dataflows within long-running containers respecting explicit dataflow requirements. First we provide an overview of the section.

Overview of evaluation. We compare NEPTUNE with existing distributed dataflow systems and various scheduling policies that are described in detail along with our evaluation set-up in §4.6.1. Then we analyse the performance of our system focusing on our execution requirements including: minimising queueing and thus latency for latency-sensitive jobs of a stream/batch application; ensuring high throughput regardless the job type; and achieving high resource utilisation and thus resource efficiency. We study the performance of NEPTUNE under different scenarios:

- We first compare the execution performance of stream/batch applications with NEPTUNE and with existing solutions. We use typical application benchmarks including a machine learning (ML) training/inference application [67], similar to the one depicted in Figure 3.2, and the YSB streaming benchmark [24] (§4.6.2)
- We then measure the performance impact in terms of throughput and latency with varying resource demands. By increasing the resource demands of our higher-priority stream tasks we observe how NEPTUNE adapts to the new requirements and to what extent lower-priority jobs are affected (§4.6.3)
- Finally, using TPC-H, a decision-support benchmark as micro-benchmark [131], we explore the efficiency and scalability of NEPTUNE's task suspension mechanism using coroutines. We also compare with alternative task preemption approaches based on traditional thread synchronisation primitives (§4.6.4).

4.6.1 Experimental set-up

Cluster set-up. For our experiments, we use a cluster of 75 Azure E4s_v3 virtual machine (VM) instances. Each VM has 4 CPU cores, 32 GB of memory, and 60 GB of SSD storage. On each cluster VM, we run executors that use 4 slots for executing tasks, the same as their number of cores. Note here that in a production setting, at least one core of each VM would be dedicated to monitoring, reporting, or storage services. As we did not use any extra services for our experiments, we decided to use all available cores for task execution. We deploy Apache Spark version 2.4.0 as the baseline for our experiments. Apache Spark supports defining stream/batch application using its Structured Streaming API, but its execution engine is unaware of the various dataflow execution requirements. For all our experiments, we use the same JVM, heap size, and garbage collection flags. Finally, before taking measurements, we also warm up the JVM.

Workloads. We employ the following workloads:

1. The *Yahoo Streaming Benchmark* (YSB) models an ad-account environment that runs analytics on a stream of ad impressions [24]. In this environment, a producer inserts records in a stream and the benchmark groups the events into 10-second windows per ad campaign. It then measures

how long it takes for all events in the window to be processed. Our stream/batch application combines two concurrent instances of YSB, a latency-sensitive and a latency-tolerant one.

2. The *machine learning (ML) training/inference* application uses online Latent Dirichlet Allocation (LDA) to perform topic modeling and inference, similar to our unified dataflow example of Figure 3.2. LDA is an unsupervised machine learning method that uncovers hidden semantics ("topics") from a group of documents, each represented as a group of tokens. Each token w_{ij} in the document is associated with a latent topic assignment $z_{ij} \in \{1, \ldots, K\}$, where K is the number of topics, i indexes the documents and j indexes each token in the document. We use the NYTimes dataset for training and a small subset for inference [67]. The dataset has 300k documents, 100k words, 1k topics and 99.5 million tokens. We use the online variational Bayes LDA algorithm with parameter values miniBatchFraction=0.05 and maxIterations=50. Gibbs sampling is used to infer document topic assignments z_{ij}, as implemented in Spark MLlib [125]. Similar to YSB, our stream/batch application has a latency-sensitive inference job and a latency-tolerant training job.

3. *TPC-H* [131] is a decision support benchmark with 22 analytical relational queries that address several "choke points" as part of their computation logic such as aggregates, large joins and arithmetic computations [17]. For the data generation, we use a scale factor of 10 (SF10) and data stored in the Parquet format [103].

Comparisons. We compare the following systems:

- NEPTUNE (or NEP): This is our system using the *locality and memory-aware* (LMA) scheduling policy, as described in §4.4.2. The LMA scheduling policy respects task locality preferences and load-balances preempted tasks across nodes in the cluster.

- NEP-LB: This is NEPTUNE with a simpler scheduling policy that load balances task preemption across nodes based on the nodes' memory condition but ignores task data locality preferences. As a result, high-priority tasks using such data may have to perform a remote fetch from another executor.

- FIFO and FAIR: These are the two non-preemptive policies available in Spark. FIFO prioritises tasks based on job submission time while FAIR assigns resources to jobs proportionally to their weight. For FAIR, we configure the weight for stream jobs to be equal to the job parallelism (higher-priority) and for batch jobs to be equal to one.

- KILL: This is a preemptive scheduling policy that we implemented on top of Spark, combined with the FAIR scheduling policy. It allows lower-priority batch tasks to be preempted (killed) in order to minimise queueing delays for higher-priority stream tasks. Killed tasks can reuse any input data cached on the executor, but they lose any progress that they have done as they are restarted.

- DIFF-EXEC: This policy represents how Spark runs different priority tasks in separate executors for isolation similar to static resource allocation. We deploy two executors per machine, each using half the CPU cores. One executor is used for high-priority tasks and the other for low-priority ones.
- PRI-ONLY: We execute only the stream tasks of a unified stream/batch application. This provides the ideal latency scenario for higher-priority stream tasks, as they can execute in complete isolation.

Our implementation of NEPTUNE is based on Apache Spark 2.4.0.

4.6.2 Application performance

First, we want to study the effect of NEPTUNE on end-to-end application performance in terms of latency and throughput for different applications. For this experiment, we use the 75-instance Azure cluster described above and run two different stream/batch applications: the Yahoo Streaming Benchmark with two job instances, a latency-sensitive one and a latency-tolerant one, and the ML application consisting of a latency-tolerant training job and a latency-sensitive inference job.

The goal for NEPTUNE is to minimise end-to-end latency for higher-priority stream tasks with the least impact on the throughput of latency-tolerant tasks. At the same time, NEPTUNE should be able to a scale to a significant number of machines without introducing extra overhead.

Yahoo Streaming Benchmark (YSB). We use the YSB application to measure end-to-end latency and throughput as used by others to evaluate similar system properties [136]. We measure latency for each window as the difference between the last event processed and the window end time; throughput is the total number of records processed over time. Our stream/batch application consists of two YSB jobs. The latency-sensitive stream job has a parallelism of 44 occupying 15% of cluster resources and generating thousands of events per second. The latency-tolerant batch job has enough parallelism to occupy all cluster resources and generates millions of events per second.

Figure 4.6 shows the end-to-end latency for the stream job of the unified stream/batch application using different systems and scheduling policies. For this figure, we use box plots in which the lower/upper parts of the box represent the 25th/75th percentiles, respectively; the middle line is the median; and the whiskers are the 5th and 99th percentiles, respectively.

We can observe that, from the default Spark scheduling policies (FIFO and FAIR), FAIR has a lower and tighter distribution. In more detail, the FAIR scheduling policy reduces the 75th percentile latency for the stream job by 5%, and the 99th percentile latency by 37% compared to FIFO. However, the tail latency still remains more than 2$\times$ higher than for PRI-ONLY where the stream jobs run in complete isolation.

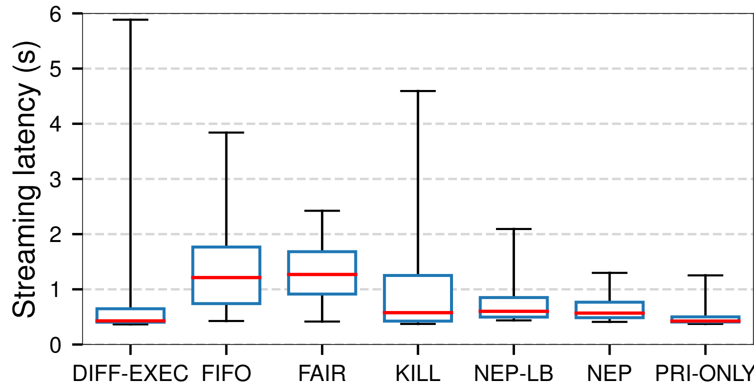
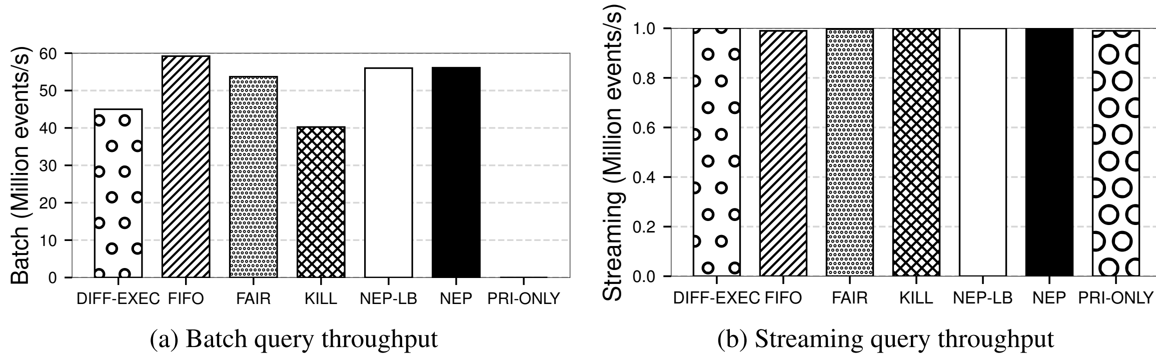


Figure 4.6 Yahoo Streaming Benchmark (streaming + batch) streaming query latency

A source of extra latency for the FAIR policy is queueing, with tasks waiting for resources to become available. To minimise queueing for stream tasks, we implemented the preemption-enabled KILL scheduling policy on top of Spark that is allowed to kill lower-priority tasks in favour of stream tasks. By preempting batch tasks, KILL reduces the median latency by 54% compared to the non-preemptive FAIR, but the 99th percentile is almost 2× higher than FAIR. The reason for the high tail latency is twofold: (i) KILL cannot preempt more than a weighted share of resources as inherited by the FAIR weight configuration, which is ineffective when too many stream tasks wait for execution; and (ii) the housekeeping process of killing tasks in a Spark executor involves releasing locks and cleaning up allocated memory and pages in the block manager, which is time-consuming. It is thus clear that, with KILL, we can not achieve end-to-end latencies that are close to ideal, which is running the stream job in isolation.

NEPTUNE with the LMA scheduling policy (NEP) achieves latencies comparable to PRI-ONLY. The simpler NEP-LB policy that does not consider cache preferences achieves 61% worse latency compared to NEP for the 99th percentile. The reason behind the worse latency is cache preferences that affect task latencies for jobs that rely on executors to store data, especially at the higher percentiles. Finally, DIFF-EXEC which represents the static allocation scenario where each job runs on a different executor, reduces the median latency but increases the tail latency. We observe high tail latencies when executors on the same machine interfere with each other competing for the same resource such as memory bandwidth or network. NEPTUNE does not face resource interference by having a single executor per machine and using an effective task scheduling policy.

As scheduling policies prioritise or preempt job tasks over others, they also affect job throughput as shown in Figure 4.7. The throughput of the batch query over different scheduling policies is depicted in Figure 4.7a. The default Spark scheduling policy, FIFO, that is unaware of priorities and fairness, achieves the highest throughput, as batch jobs consume all cluster resources when scheduled. Spark's FAIR policy improved stream latency compared to FIFO comes at the cost of decreased batch throughput by 10% due to the prioritisation of stream over batch tasks. As expected, due to the



(a) Batch query throughput

(b) Streaming query throughput

Figure 4.7 Yahoo Streaming Benchmark (streaming + batch) throughput

termination of batch tasks and loss of their execution progress, KILL achieves the worst throughput, namely 32% lower than FIFO and 25% lower than FAIR.

As shown in Figure 4.7a, even though DIFF-EXEC runs the batch jobs on separate executors, it experiences a 24% lower throughput than FIFO. This is because DIFF-EXEC relies on a static allocation of resources with up to 85% of executors used for batch jobs while FIFO can use 100% of the executors. However, DIFF-EXEC throughput is still 12% higher than KILL, meaning that restarting tasks without preserving their progress has the worst impact in terms of throughput. Finally, both the NEP and NEP-LB policies achieve a throughput comparable to the best performing FIFO policy (within 5%).

For the streaming job, Figure 4.7b shows identical throughput for all configurations. The reason for the identical results is the micro-batching model Spark uses by default. The micro-batching model accumulates streaming events and periodically triggers processing for all available data. As a result, over time the non resource-demanding stream job computation is amortised, resulting in fixed throughput but varying latency. Thus, for a share of resources, Spark can achieve similar throughput whether tasks are scheduled immediately or queued.

ML training/inference application. The next stream/batch application that we evaluate performs topic modeling and consists of a training and an inference job. The training job uses the online variational Bayes LDA algorithm to generate a new model and, after a number of iterations, it stores the model in HDFS distributed storage using all available resources. The inference job loads the latest trained model in cache and infers topics for the received documents using Gibbs sampling, consuming up to 15% of all cluster resources.

Figure 4.8 shows the latency distribution achieved for the inference job across different configurations. As LDA training tasks are computationally intensive, queueing inference tasks behind training tasks has a significant impact on task latency. We observe that the FAIR scheduling policy reduces the 99th percentile latency by 13% compared to FIFO. However, FAIR still yields almost 3× higher latency than running the inference job in isolation (PRI-ONLY) for the 99th percentile. Note that the preemptive KILL policy is able to reduce the median latency by 18% compared to FAIR and 20% compared to

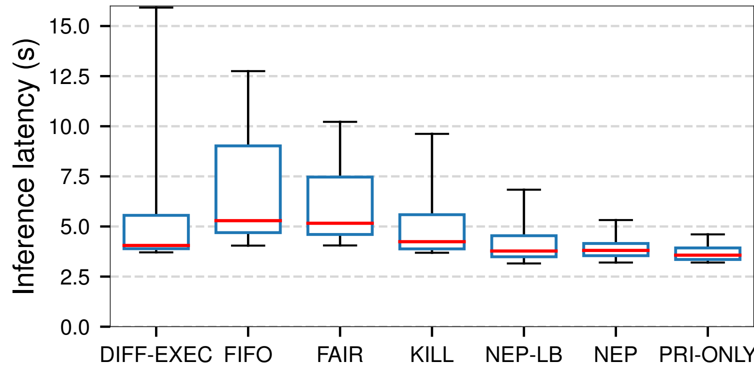


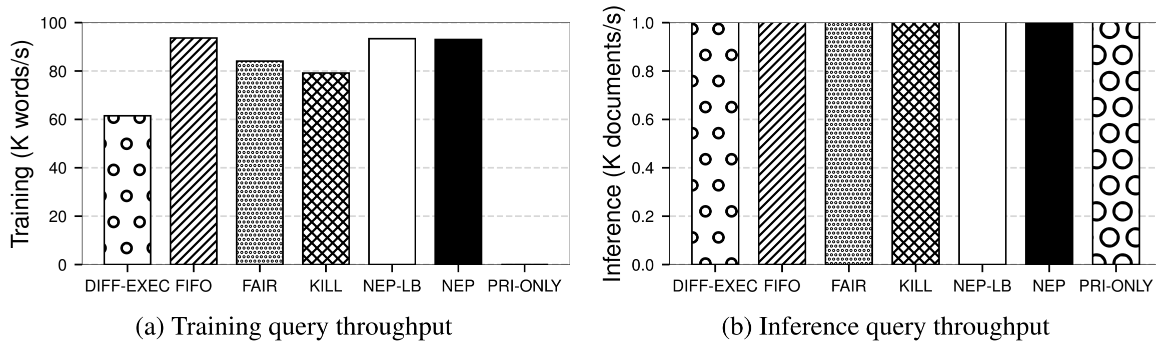
Figure 4.8 LDA NYTimes dataset (training + inference) inference latency

FIFO, but its 99th percentile latency is 55% higher than running the job in isolation (PRI-ONLY). As explained above for the YSB application, KILL cannot preempt more than a weighted share of resources and faces the extra overhead of housekeeping terminated tasks.

NEP with the LMA scheduling policy achieves a latency that is just 6% and 13% higher than PRI-ONLY for the median and the 99th percentiles, respectively. NEPTUNE without cache awareness (NEP-LB) has a 29% worse latency for the 99th percentile compared to NEP. For the LDA stream/batch application, it is again obvious that task cache preferences are affecting the task tail latency. Finally, although running the jobs in different executors (DIFF-EXEC) reduces median latency by 23% and 21% compared to FIFO and FAIR, respectively, it incurs high tail latencies due to executor interference that are almost 2× higher than NEP for the 99th percentile.

The effect of scheduling policies in terms of job throughput is shown in Figure 4.9. More precisely, Figure 4.9a reports the achieved throughput for the training job. Similar to the YSB application, FIFO yields the highest throughput by entirely ignoring job priorities. FAIR achieves 10% lower throughput compared to FIFO by prioritising inference tasks over training using static job priorities that are expressed as weights. KILL that is based on the static priorities of FAIR achieves throughput that is 15% lower than FIFO. KILL has 5% lower throughput than FAIR, as tasks have to be restarted after preemption. DIFF-EXEC, which uses only up to 85% of available executors, achieves a 34% lower throughput compared to FIFO. Unlike YSB, in this application, DIFF-EXEC performs worse than KILL because reduced resources have more impact on computationally intensive training jobs. Finally, both NEP and NEP-LB achieve throughput comparable to the best performing FIFO policy, with an overhead of just 1%.

Figure 4.9b shows the inference throughput across configurations which is identical, similar to the YSB application above. As Spark uses a micro-batching model that accumulates events and periodically triggers processing over all available data, over time, the less resource-demanding stream job computation is amortised, resulting in fixed throughput but with varying latency.



(a) Training query throughput

(b) Inference query throughput

Figure 4.9 LDA NYTimes dataset (training + inference) throughput

In summary, in a cluster of 75 Azure nodes, NEPTUNE is able to scale and achieve latencies that are comparable to the ideal PRI-ONLY policy for the latency-sensitive jobs. At the same time, NEPTUNE reduces the throughput of latency-tolerant jobs by less than 5% compared to the best performing FIFO policy.

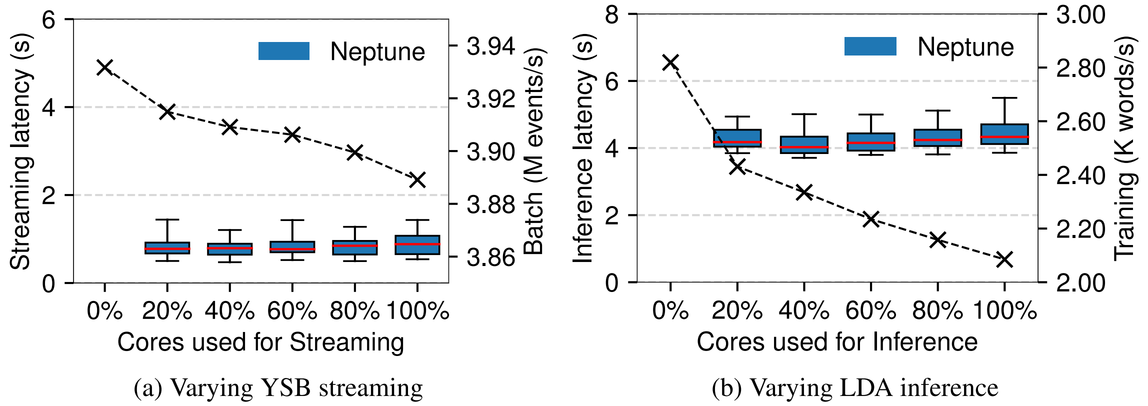
4.6.3 Varying resource demands

Next we want to evaluate the performance of NEPTUNE in terms of throughput and latency under varying resource demands. We explore how the increasing resource demands of higher-priority stream jobs affect system performance and memory utilisation, as more lower-priority tasks have to be suspended concurrently. We also focus on a memory constrained environment to evaluate the effectiveness of our locality- and memory-aware (LMA) scheduling policy.

The goal for NEPTUNE is to maintain low latency for higher-priority stream jobs even when it has to suspend the entire cluster. At the same time, the LMA scheduling policy should be able to avoid memory-constrained executors that can potentially hurt end-to-end job latency.

Different job resources. The applications evaluated in §4.6.2 only use part of the cluster resources for the stream jobs (15% of the entire cluster). In this experiment, we use the same stream/batch applications deployed in a cluster of 4 Azure VMs, while we increase the stream job resource demands from 0% to 100% of cluster resources. Batch jobs in our experiment still use all available resources. We use NEPTUNE to execute jobs and measure latency, throughput, and memory consumption.

Stream latency. We run the YSB application with increasing resource demands, and we observe that NEPTUNE maintains low latency across all percentages as shown with box plots in Figure 4.10a. As the lower-priority tasks that must be suspended increase, up to the point that the entire cluster must be suspended for 100% cores used, the low latency maintained by NEPTUNE shows the effectiveness of our preemption mechanism. The higher number of preempted tasks, however, incurs a penalty in batch job throughput, depicted as a dashed line in the figure. For instance, when stream tasks ask for



(a) Varying YSB streaming (b) Varying LDA inference

Figure 4.10 Performance impact of varying demands

40% and 100% of the cluster, batch throughput drops by 0.8% and 1.1%, respectively, compared to optimal (0%).

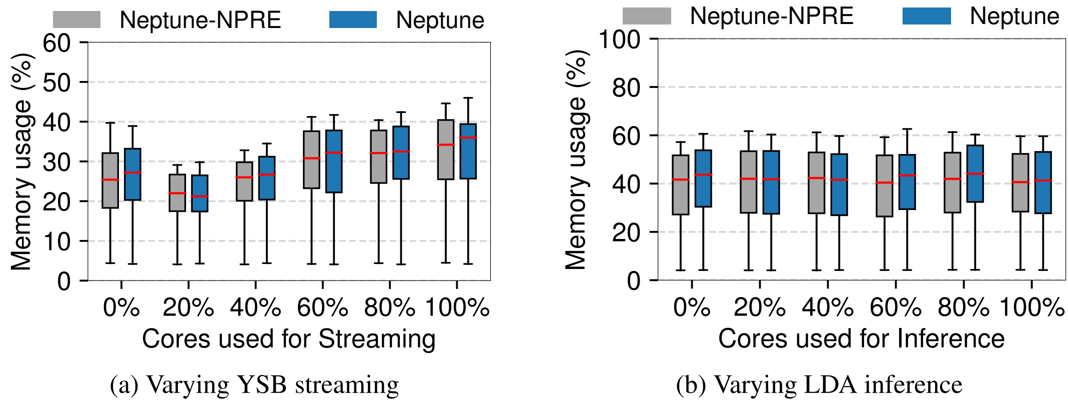
Similarly, increasing inference resources in the LDA application from 0% to 100% does not affect latency in NEPTUNE, as shown in Figure 4.10b. However, when inference tasks demand 40% and 100% of the cluster, training throughput drops by 17% and 26%, respectively. As inference tasks of this application have longer runtimes compared to YSB stream tasks, they have a higher impact on lower-priority jobs in terms of throughput.

Memory overhead. We also explore the effect of task preemption in NEPTUNE by measuring memory usage. As suspendable tasks in NEPTUNE use coroutines that maintain a stack of execution frames in-memory, it is important to validate that their state is efficiently garbage collected and does not lead to significant memory-pressure that could cause job slowdowns.

For this experiment, we compare two systems as shown in Figure 4.11: NEPTUNE with LMA scheduling policy and plain NEPTUNE without preemption, depicted as Neptune-NPRE. As shown in Figure 4.11a, when the YSB streaming tasks are first introduced in the system, the task scheduler schedules fewer batch tasks. As a result, memory usage drops by 25% for the 99th percentile when cores used for streaming are 20%, for NEPTUNE, both with and without preemption. It then steadily increases in line with the stream job demands from 40% to 100%. Across the full range, however, NEPTUNE's memory usage is no more than 2% for the 99th percentile compared to Neptune-NPRE.

In a similar manner, for the machine learning training/inference (LDA) application that is shown in Figure 4.11b, even though there is no drop in memory usage when inference tasks are first introduced in the system, the increase is no more than 1.5% for the 99th percentile compared to Neptune-NPRE.

Taking into account the memory usage distributions across increasing stream job demands with and without preemption enabled, we conclude that NEPTUNE's suspension mechanism only introduces a modest memory overhead and thus cannot directly lead to job slow-downs.



(a) Varying YSB streaming

(b) Varying LDA inference

Figure 4.11 Memory impact of varying job demands

Memory pressure. In the next experiment, we investigate the effectiveness of NEPTUNE under memory pressure. To achieve this, we synthetically create a memory-constrained environment where some executors have limited available memory and selecting these executors to run high-priority tasks could hurt their performance. For this experiment we use the same 4 Azure VMs, and we reduce each worker's JVM size to 1 GB. We use a single YSB stream instance utilising 30% of the cluster resources and, as the batch workload, we use an increasing number of joins, joining millions of rows. In this memory-constrained environment, the worker's average memory utilisation increases from 20% to 40% and 80% with 2, 4, and 6 joins, respectively.

Figure 4.12 shows the end-to-end latency for the streaming (higher-priority) job across different scheduling policies, including the default Spark FIFO and FAIR policies, our enhanced KILL policy with preemption, and NEPTUNE with (NEP) and without (NEP-LB) cache locality awareness.

In more detail, with 2 concurrent joins and low memory pressure (20% average memory utilisation), all policies achieve similar mean latencies of around 100 ms (Figure 4.12). As we increase the number of joins to 4, with almost half of each worker's memory utilised on average by batch tasks, NEPTUNE achieves 99th latencies that are 1.9× lower than FIFO and 2.8× lower than KILL, respectively. Finding the least memory-constrained executor in NEPTUNE helps higher-priority tasks achieve lower and more predictable tail latencies (thus the tighter latency distribution).

The effect is even more pronounced with 6 joins — NEPTUNE achieves 99th percentile latencies that are 3× lower than FIFO and 2× lower than FAIR, respectively. NEP-LB, ignoring individual task locality preferences but taking into account executor memory conditions, performs similar to NEPTUNE in this context, with 99th latencies that are just 2% and 12% higher for 4 and 6 joins, respectively. On the other hand, FIFO and FAIR scheduling policies that ignore cluster state when scheduling tasks can lead to the highest tail latencies of the experiment, with values of 4 and 3 seconds, respectively.

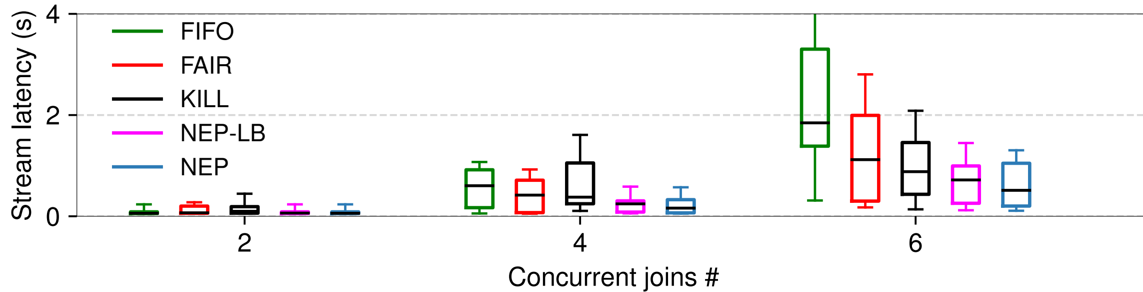


Figure 4.12 Latency impact of increasing memory use

We conclude that using real-time executor metrics, NEPTUNE manages to predict accurately the memory pressure of each individual executor in the cluster, thus reducing memory bottlenecks to obtain lower tail latencies for the tasks for which it matters the most.

4.6.4 Task suspension

Finally, we want to evaluate the effectiveness and efficiency of NEPTUNE's coroutine-based task suspension mechanism. Using the TPC-H workload [131] consisting of 22 analytical queries with different complexity and computation needs and using solely batch tasks we first measure how fast coroutines can yield, and then compare how coroutines scale as the parallelism increases compared to traditional thread synchronisation suspension primitives.

For our suspension mechanism, we want to be able to pause and resume tasks with low latency regardless the computation task complexity. Moreover, it should possible able to scale to machines with a large number of cores and not be heavily affected by the increasing parallelism within an executor.

Suspension latency. First, we run the TPC-H benchmark on a cluster of 4 Azure VMs and measure the task duration distribution for each query. Using a custom `TaskEventListener` that we implemented, we re-run the benchmark and continuously transition tasks from `PAUSED` to `RESUMED` states until completion, while measuring the latency for each transition. As coroutine tasks can yield after each record iteration, a task that does heavy per-record computations takes longer to get suspended. By continuously triggering yield points, we measure the worst case scenario in terms of transition latency for each query task.

Figure 4.13 shows the task runtime and pause/resume latency distributions for all 22 queries; whiskers represent the 5th and 99th percentiles, respectively. We observe that, although queries have different task runtimes (grey boxes) ranging from 100s of milliseconds to 10s of seconds, NEPTUNE can pause (red boxes) and resume (blue boxes) tasks with sub-millisecond latencies, showing the effectiveness of our mechanism.

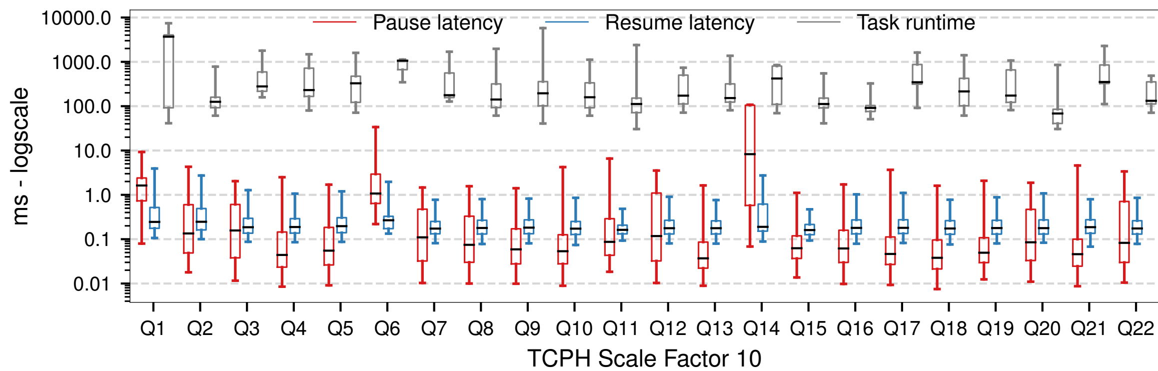


Figure 4.13 NEPTUNE task suspension mechanism latency breakdown using TPC-H queries

An exception is Q14 with a 75th percentile pause latency of 100 ms. After a deeper investigation, we observe that this query reads from the `lineitem` table and has no filters other than on the date. Given that this table is partitioned by date, the Parquet reader that operates at the partition level has to consume full partitions before the corresponding Spark tasks can yield. So in this particular case, the pause latency is affected by the partition size or, more precisely, by how fast NEPTUNE can ingest the data partition.

In general, when dealing with file partitions, the less selective the query, the longer it takes to consume each partition (less/no filters are pushed down to the reader), which results in increased suspension times. This problem can be mitigated by re-partitioning the table, increasing the task parallelism, or improving readers to be more fine-grained which is orthogonal to our approach.

Coroutines vs. thread synchronisation. Finally, we want to compare NEPTUNE's coroutine-based task preemption with an alternative approach based on traditional thread synchronisation primitives, depicted as ThreadSync. Thread synchronisation relies on the preemption performed by the OS scheduler and is implemented in NEPTUNE using thread *wait* and *notify* calls. As discussed in §4.3, with *wait* and *notify* calls, the actual preemption is performed by the OS scheduler, which moves threads between a wait and a running queue in kernel space.

In Figure 4.14, we compare the preemption mechanism of NEPTUNE using Coroutines with the alternative mechanism of ThreadSync. We run TPC-H queries on a 32-core Azure VM using the custom `TaskEventListener` used in the previous experiment, alternating tasks from `PAUSED` to `RESUMED` states, while increasing the parallelism of the executor from 2 to 64 (double the number of cores).

The results for the first TPC-H query are shown in Figure 4.14, with a similar behaviour for the rest of the queries. With up to 8 parallel tasks, both Coroutines and ThreadSync mechanisms can achieve low pause latency. However, as the parallelism increases, the overhead of ThreadSync rises: the 99th percentile of yield latency increases by 2.6× for 16 threads compared to Coroutines. The reason for the jump in latency is the OS scheduler that must continuously arbitrate between wait/run queues,

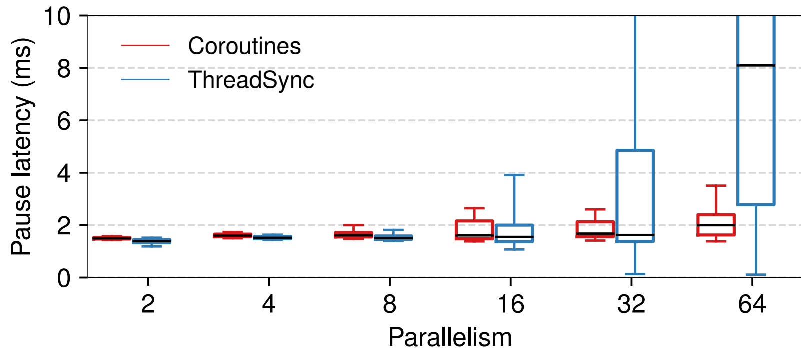


Figure 4.14 Latency of task suspension mechanisms

which is similar to the results from other studies [138]. ThreadSync exhibits worse scaling compared to coroutines, which completely bypasses the OS.

Note here that each stream task typically has a runtime of a few 10s of milliseconds. At the same time, the gains from Coroutine task suspension compared to ThreadSync are in the range of a few milliseconds and can be as high as 600 ms for the 99th percentile for 64 threads (omitted from Figure 4.14). Therefore, the overhead of suspending a task with thread synchronisation primitives can be a significant part of a stream task's runtime (up to orders of magnitude higher in the worst-case), considerably affecting its latency. Moreover, as stream jobs consist of multiple tasks, potentially triggering 100s of task suspensions, the overheads accumulate.

4.7 Limitations and discussion

In the following paragraphs, we highlight some of NEPTUNE 's current limitations and initiate a discussion on their significance and how they can be addressed as part of future work.

Memory pressure. Although suspendable tasks in NEPTUNE are both efficient to suspend/resume and transparent to users, they keep state in memory, which may increase memory pressure on executors. However, we observe that typical production clusters today tend not to be constrained by memory.

Figure 4.15 shows the available machine memory in a Microsoft production cluster comprised of tens of thousands of machines. We took measurements every few seconds over the course of four days, and we plot the CDF of the available machine memory, grouped by machines with the same hardware configuration. The available memory in 90% of the machines is at least 75 GB and in some configurations over 150 GB. In addition to that, tasks in this cluster typically require less than 10 GB of memory each, which means that several more tasks can be started at each machine in terms of memory. Similar trends hold at other companies. For example, in production Spark clusters at Facebook, 95% of the machines utilise no more than 70% of the machine's memory [2].

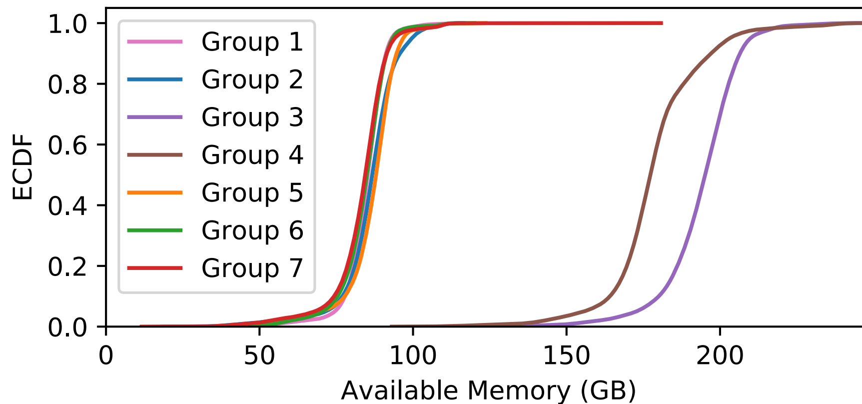


Figure 4.15 Available machine memory in a Microsoft production cluster (The cluster is comprised of tens of thousands of machines with each group corresponding to different hardware configurations. Data is over four days.)

In cloud deployments, it is common for users to over-provision their memory resources [27] and, if not, they can always allocate more memory to their VMs. NEPTUNE takes advantage of this flexibility in terms of memory availability to reduce the latency of higher-priority tasks that would otherwise have to face extra queueing delays. Moreover, as NEPTUNE limits the number of tasks simultaneously suspended on each executor by the number of CPU cores and not the amount of memory, running out of memory is rarely an issue in practice. In cases when memory does become the bottleneck, NEPTUNE's LMA scheduling policy, described in §4.4.2, can ensure that machines with the least memory pressure are picked for task suspension. In memory-constrained environments, NEPTUNE could also utilise a checkpointing alternative that, even though it is slower than coroutines, it does not keep any state in memory.

Checkpointing. NEPTUNE adds preemption support to Apache Spark, which is an in-memory framework, but does not currently support suspended task checkpointing. The main reason is that whenever there is checkpointing to disk, task latency will be affected significantly, which defeats NEPTUNE's main goal of providing millisecond-latency preemption. Note that, by design, NEPTUNE can suspend tasks of different jobs, as long as they belong to the same application and share executors.

In the future, we could employ existing techniques to checkpoint NEPTUNE's suspended tasks to release memory under high memory pressure conditions [96]. An alternative system design, relying on coarser-grained checkpointing, would be to have separate JVMs per task and use OS containers to suspend and resume tasks.

Coroutine efficiency. Another noteworthy point is how fast coroutines can pause and resume. Several recent systems in the context of databases [74, 109] and machine learning [110] use coroutines to reduce the latency of operations but NEPTUNE is the first system to do so for a distributed dataflow framework. In NEPTUNE, coroutine pause latency is determined by the amount of processing per

data record before the coroutine is able to yield. Although task optimisations such as function pipelining in Spark can increase yield times, we find that the difference is typically negligible in practice (see §4.6.4).

In our experiments, we only observe increased yield times for the handling of large file partitions when queries are less selective. As we explained in the previous section, this is caused by the file reader mechanism in Spark and can be mitigated. NEPTUNE could automatically detect such problematic tasks and avoid potential issues, for example by re-partitioning data or increasing task parallelism.

Applicability. Even though NEPTUNE is built on top of Apache Spark, we believe that our design, as shown in Figure 4.1, is applicable to other dataflow frameworks for stream/batch applications, such as Apache Flink [20] and Storm [127]. These dataflow systems follow a similar architecture to NEPTUNE. We opted to have Neptune's first implementation on Spark, hence we cannot claim concrete gains on other systems. However, we expect Flink, which uses a continuous-operator execution model, to benefit even more from task suspension, as operators typically run for longer durations compared to scheduled tasks in Spark. By introducing dynamic scheduling decisions and suspending continuous tasks we could utilise resources more efficiently (e.g., the idle periods of continuous operators could be used to run lower-priority training tasks).

Extending NEPTUNE and integrating the unified dataflow model with other execution engines is thus another future direction towards the goal of developing an architecture for general-purpose unified dataflow execution. Unified dataflow architectures would expose unified APIs similar to the ones supported today for Spark's Structured Streaming, Flink's Table API, as well as the external unified API of Apache Beam, but would also respect the diverse execution requirements of unified applications, efficiently utilising cluster resources.

4.8 Summary

In this chapter, we presented NEPTUNE, an execution framework that realises unified dataflows on top of Apache Spark. It supports stream/batch applications that share long-running executors and can dynamically prioritises latency-critical jobs, while effectively utilising application resources. With its suspendable task implementation based on coroutines and smart scheduling policies, the design of NEPTUNE paves the way for truly unified stream/batch applications.

Our experimental evaluation demonstrates the benefits of NEPTUNE for stream/batch applications. On a 75-node Azure cluster using machine learning and streaming benchmarks, NEPTUNE reduces end-to-end latencies by up to 3$\times$ compared to Spark and increases the throughput by up to 1.5$\times$. Using the locality- and memory-aware (LMA) policy, NEPTUNE achieves close to optimal performance for both latency-critical and latency-tolerant tasks with a modest memory impact. Through micro-benchmarks,

we confirm that suspendable tasks can efficiently pause and resume with sub-millisecond latencies and that, in scenarios with memory pressure, LMA scheduling policy can better utilise resources.

Chapter 5

Medea: A Cluster Scheduler with Rich Placement Constraints

As we have shown in the previous chapter, the explicit execution requirements of *unified dataflows* as part of a novel dataflow framework enable users to define jobs with diverse computation in the same application and achieve efficient task execution within the same shared executors. One remaining question is how to support the rich placement constraints of our model expressing the placement of an application's long-running executors. Our goal is to enable users and cluster operators achieve more effective, high-quality placements of long-running workloads in shared compute clusters.

This chapter describes the system aspects of our placement constraints and more precisely how to realise high-level and expressive constraints in a scalable and performant manner at the cluster manager level. Since cluster schedulers are application-agnostic, they have enabled the consolidation of a plethora of diverse workloads onto thousands of machines. However, supporting placement constraints for these workloads as part of a production-hardened cluster manager presents several challenges. Simply adding complex constraints to the cluster manager could introduce delays that are unreasonable for traditional batch analytics workloads focusing on low-latency allocations. On the other hand, workloads that run for longer periods of time (LRAs) can tolerate some scheduling delays and would actually benefit from higher-quality container placement. In addition to constraints, cluster schedulers must satisfy their own global cluster objectives ranging from sharing policies and fairness to load-balancing and packing. At the same time, configuring them in a production setting is a time and labour intensive process.

As we discussed in §2.1, cluster schedulers that handle a variety of diverse data-intensive workloads at a data-centre scale are increasingly common [31, 77, 137]. Their workloads range from batch, that typically benefit from short-lived containers running for durations in the order of seconds, to long-running such as unified dataflows that utilise *long-lived containers* running for hours to months. For workloads with longer durations, container placement is especially important for

satisfying requirements such as application performance, resilience and global cluster objectives (see §2.3.1–§2.3.3). As part of our unified dataflow model, we capture container interactions within and across applications that, if properly utilised, can lead to an effective placement of their long-running containers in the cluster.

Opportunity. In this chapter, we argue that for the effective placement of long-running containers, there is a need for a cluster scheduler with explicit support for complex high-level constraints. The scheduler should achieve high-quality placements for LRAs in general within shared infrastructures without impacting the scheduling latency of traditional jobs with short-lived containers. Despite the fact that long-running workloads occupy a significant portion of cluster resources in production clusters today, support for LRAs in existing schedulers is rudimentary [10, 78, 82, 137]. Existing schedulers such as Mesos [65] and Borg [137] only support constraints implicitly through static machine attributes, which prohibits defining constraints without revealing the underlying cluster infrastructure. Kubernetes [82] supports explicit intra- and inter-application high-level constraints between containers, but it does not support cardinality, which is critical to control the level of container collocation and thus resource interference (see §2.3.1). As a result, in existing cluster managers, the scheduling requirements of LRAs remain largely unsupported.

Requirements. In order to efficiently support the container placement of long-running applications, a cluster manager must meet several requirements (summarised as **R1-R4** in §2.3.4):

1. *Support powerful and high-level constraints* to capture interactions between containers and unlock the full potential of applications. Precise control of container placement is key for optimising the *performance* and *resilience* of LRAs. Simple affinity and anti-affinity constraints such as collocating containers to reduce network costs, or separating them to minimise resource interference or chance of failure, are necessary but not sufficient. Constraints should also be agnostic of the underlying cluster organisation (requirements **R1** and **R2**).

2. *Achieve high-quality placement* and allow cluster operators to optimise for cluster objectives. When placing LRA containers, the cluster scheduler must achieve global optimisation objectives, such as minimising the violation of placement constraints, the resource fragmentation, any load imbalance, or the number of machines used. The cluster operator should be able to determine the objectives to be used and define their relative importance (requirement **R3**).

3. *Not impact system scalability or scheduling latency* of the traditional applications that use short-lived containers in shared production clusters. Due to their long lifetimes, LRAs can tolerate longer scheduling latencies than batch jobs, but their complex placement should not impact the scheduling latency of applications with short-lived containers that are running in the cluster (requirement **R4**).

In this chapter, we describe MEDEA, a new cluster scheduler motivated by the above requirements, which enables the placement of both long- and short-running containers in shared compute clusters in a scalable manner. We start with an overview of MEDEA and its components in §5.1. In §5.2, we discuss how MEDEA enables application developers to *express* powerful placement constraints across LRA containers with formal semantics using container tags and node groups. In §5.3, we describe how MEDEA *schedules* LRAs achieving high-quality placement and global objectives by formulating the placement of LRAs with constraints as an integer linear program (ILP) (§5.3.2). We also investigate heuristic alternatives that can trade placement quality for lower scheduling latency in §5.3.3. In §5.4, we describe MEDEA's open-source implementation on top of Apache Hadoop YARN, one of the most widely deployed cluster schedulers used by companies such as Microsoft, Yahoo!, Twitter, LinkedIn, EBay, Hortonworks, and Cloudera. Our experimental evaluation of MEDEA on a 400-node pre-production cluster follows in §5.5. Finally, in §5.6, we discuss MEDEA's limitations.

5.1 Overview

Similar to existing cluster managers, MEDEA supports the scalable and low-latency scheduling of "traditional" applications with short-running containers, also referred to as *task-based* jobs. Unlike existing schedulers, MEDEA fully supports the scheduling requirements of LRAs (**R1–R4** in §2.3.4), including their complex high-level constraints that are crucial for achieving effective and high-quality placement for long-lived containers.

To support both LRAs and traditional applications in shared compute clusters, MEDEA uses a two-scheduler centralised design for placing containers consisting of: (i) a dedicated *LRA scheduler*; and (ii) a traditional *task-based scheduler*, as shown in Figure 5.1. In MEDEA, the dedicated *LRA scheduler* is responsible for placing long-running containers, while accounting for constraints and achieving placements that satisfy global cluster objectives, such as balancing cluster load or minimising the number of used machines. Constraints are stored in the *constraint manager* component that provides a central view of all the enabled constraints in the cluster. At the same time, the traditional *task-based scheduler* places task-based jobs with low latency, reusing existing task-based scheduler implementations and facilitating the adoption of MEDEA in production settings without the need for a labour-intensive system reconfiguration. Next we discuss the key components of our design.

LRA interface. When an application owner submits a request to MEDEA, similar to other schedulers, they specify the required containers (e.g., "10 containers with 2 CPUs and 4 GB RAM each"). These resource demands, along with some simple constraints (e.g., data locality) are sufficient for task-based jobs. For LRAs, MEDEA introduces a rich API for expressive placement constraints that can capture interactions between containers satisfying requirements **R1–R2**. Applications that use the more evolved constraints API are handled by the LRA scheduler, while the ones using the simpler container

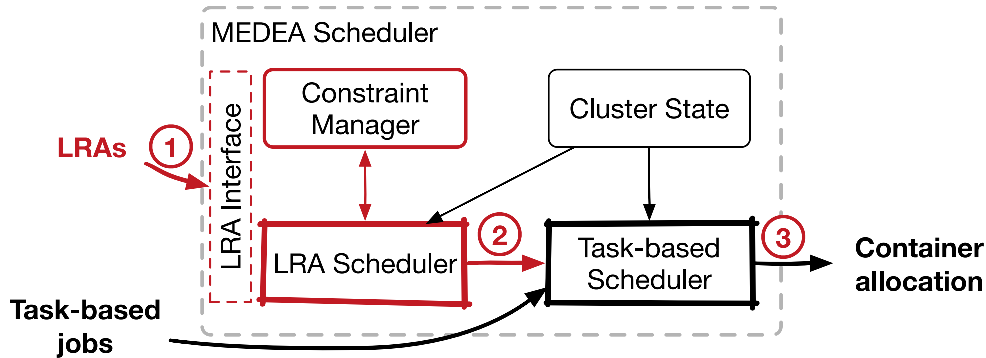


Figure 5.1 MEDEA scheduler design

request API are handled by the traditional task-based scheduler. A description of our supported constraints is given in §5.2.

LRA scheduler. The LRA scheduler uses an online optimisation-based algorithm that, given the current cluster condition, including already running LRAs and task-based jobs, determines the efficient placement of newly submitted LRA container requests. The scheduler is invoked at regular configurable intervals to place all LRAs submitted during the latest interval. Our scheduling algorithm, described in §5.3, takes into account multiple LRA container requests at once to satisfy their placement constraints and attain global optimisation objectives (requirement **R3**). By considering multiple containers at once, MEDEA achieves placements that lead to significantly fewer constraints violations compared to considering only a single container request at a time (see §5.5.4).

Task-based scheduler. MEDEA's two-scheduler design removes the burden of handling complex placement decisions as part of the scheduling logic from the task-based scheduler. As a result, the scheduling latency for task-based jobs remains low without extra overheads (requirement **R4**). Moreover, this design allows the reuse of existing production-hardened task-based schedulers. This minimises the changes required in the existing scheduling infrastructure and is crucial for adopting MEDEA in production.

Constraint manager. To manage placement constraints, we introduce a new central component in which all constraints—both from the application owners and cluster operators—are stored. This allows MEDEA to have a global view of all active constraints and to expose them to application owners or cluster operators that can add, remove or modify constraints.

Scheduling life-cycle. The scheduling life-cycle of an LRA in MEDEA can be summarised in the following steps: users submit LRAs along with their placement requirements to the LRA scheduler through the LRA interface (step 1 in Figure 5.1). The LRA scheduler makes placement decisions considering cluster state and existing constraints stored as part of the constraint manager. Then, the placement decisions made by the LRA scheduler are passed to the task-based scheduler (step 2).

Finally, the task-based scheduler performs the actual resource allocations on cluster machines in the form of containers (step 3).

This approach avoids the challenge of conflicting placements faced by existing multi-level [65, 117] and distributed [18, 77, 100, 111] schedulers. In these designs, different schedulers operating on the same cluster state may arrive at conflicting decisions. They then resolve scheduling conflicts by pessimistic [65] or optimistic [117] concurrency control, or by queueing tasks at worker nodes [18, 77, 100, 111], MEDEA bypasses this problem by having a single scheduler performing the actual allocations. We further discuss placement conflicts in §5.6.

Performing all allocations through the task-based scheduler also allows us to achieve (weighted) fairness and respect application priorities across both LRAs and task-based jobs.

5.2 Expressing placement constraints

In §2.3, we discussed how container placement is important for the performance and the resilience of applications, especially ones with long-lived containers. We also showed that simple affinity and anti-affinity constraints, albeit beneficial, are not enough to control the level of container collocation in the cluster. We finally argued that, besides being expressive, placement constraints should also be high-level, not revealing the cluster infrastructure.

We presented our approach to provide high-level and expressive constraints in §3. It can capture container interactions as part of a *unified dataflow model* based on the notion of container tags and logical node groups. As described in more detail in §3.3, container tags provide a mechanism to refer to containers in the same or different LRAs (see §3.3.2), while node groups are used to target specific node sets in the cluster (see §3.3.3).

In this section, we describe how MEDEA, our novel cluster scheduler, can enable powerful placement constraints leveraging container *tags* and *node groups*. Below, we explain the syntax and semantics of our constraints, followed by a discussion of our constraint model (§5.2.1).

MEDEA allows application owners and cluster operators to specify placement constraints of the following form:

$$C = \{\texttt{subject_tag}, \texttt{tag_constraint}, \texttt{node_group}\}$$

where `subject_tag` is a tag of the containers subject to the constraint, `tag_constraint` is a constraint of the form $\{\texttt{c_tag}, c_{min}, c_{max}\}$ where `c_tag` is a container tag, and c_{min}, c_{max} are positive integers, and `node_group` is a logical node group. To support more complex scenarios, the `tag_constraint` can also be a boolean expression formed by multiple tag constraints (we do not support negation yet).

For a constraint C to be satisfied, *each* of the containers with a `subject_tag` should be placed on a node belonging to a node set $\mathscr{S} \in \texttt{node_group}$, such that $c_{min} \leq \gamma_{\mathscr{S}}(\texttt{c_tag}) \leq c_{max}$ holds for the tag

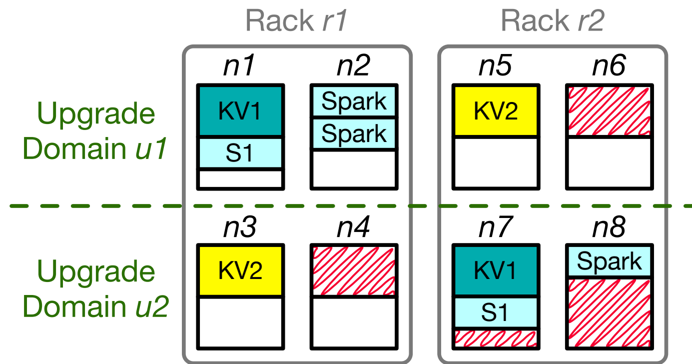


Figure 5.2 LRA placement scenario, with two key/value store instances (KV1, KV2), a streaming application (S1) and Spark job instances (Spark)

cardinality function of $\mathscr{S}$ (see §3.3.2). The number of containers of a target tag c_tag in a particular node group should be in the range between c_{min} and c_{max} for the constraint to be satisfied.

Figure 5.2 shows a small 2 rack cluster with 8 nodes split across 2 upgrade domains. Two instances of HBase (with tags hbase, KV1, memory sensitive mem), each using 2 containers, deployed, along with a Storm streaming job, S1, consuming data from KV1. Spark analytics jobs are also utilising resources in the cluster. Marked with red are the resources used by other workloads in the cluster.

As we demonstrate with several examples below, using a *single generic cardinality constraint type* and properly configuring the cardinality range variables c_{min} and c_{max}, we can express a wide range of intra- and inter-application LRA constraints and effectively place a variety of applications running in our cluster example:

1. We can express *affinity constraints* with $c_{min} = 1$ and $c_{max} = \infty$:

   Example: Constraint $C_{af} = \{$storm, $\{$hbase $\wedge$ mem, $1, \infty\}$, node$\}$ requests each container with tag storm to be placed in the same node with at least one container with tags hbase and mem. If we want to restrict the constraint to a specific application instance, for example KV1 with application[1] ID 0023, we would use $C_{af'} = \{$appID:0023 $\wedge$ storm, $\{$appID:0023 $\wedge$ hbase $\wedge$ mem, $1, \infty\}$, node$\}$. This constraint is particularly useful to a storm application communicating with hbase and mem applications; collocating their containers will reduce network costs.

2. We can express *anti-affinity constraints* with $c_{min} = 0$ and $c_{max} = 0$:

   Example: $C_{aa} = \{$KV2, $\{$KV2, $0, 0\}$, upgrade_domain$\}$ requests each hbase container from application instance KV2, to be placed in a different upgrade domain from all its other containers. The anti-affinity constraint here is used to separate containers of applications and minimise the

[1]For convenience, we automatically attach some predefined tags to each container, e.g., the ID of the LRA that it belongs to.

chance of them competing with each other for the same machine resources causing resource interference.

3. Using other values for c_{min} and c_{max} allow us to express generic *cardinality constraints*:

   Example: C_{ca} = {`storm`, {`spark`, 0, 5}, `rack`} requests each `storm` container to be placed in a rack that has no more than five `spark` containers deployed. With affinity and anti-affinity being the two extremes, cardinality allows to define the level of container collocation in a specific node group. Cardinality is particularly useful in cases in which collocating containers up to a point is known to be beneficial for the applications.

   An interesting additional feature is that we can choose `subject_tag` to be a static tag, such as `FPGA` or `GPU`, to denote machines equipped with FPGAs or GPUs, respectively.

   Moreover, if we want to specify constraints within the same group of containers (for instance from the same framework) the `subject_tag` and `tag_constraint` can use the same tags:

   Example: C_{cg} = {`spark`, {`spark`, 3, 10}, `rack`} can be used by the cluster operator to allow no fewer than three and no more than five Spark containers in a rack. This type of constraint can be particularly useful when we know that a specific range of collocated containers in a node group yields the best performance results.

Constraint dependencies. Expressed constraints make extensive use of tags, which depend on already deployed containers and that do not have to be statically predefined. As a result, both intra- and inter-application constraints can be expressed. Note, however, that if a constraint must target a specific container of another deployed application, its application and container ID are required. To address this problem, application owners can submit the two applications together. Otherwise, an independent service that exposes the deployed applications, their tags and their containers can be used.

Compound constraints. To express more complex placements, multiple constraints can be combined with boolean operators. Such constraints are specified in disjunctive normal form (DNF), allowing any arbitrary combination of constraints.

Soft constraints and weights. Each constraint can be associated with a weight. An application can use different weights to determine the relative importance of its constraints, e.g., to request node or rack affinity, with a preference for the former. By default, the constraints in MEDEA are soft, i.e., the scheduler will try to satisfy the constraint but will not deny placement in case of constraint violations (see §5.3). In our encountered practical scenarios, soft constraints capture better the expected allocation behaviour by users. MEDEA can emulate hard constraints through the use of weight values.

5.2.1 Discussion

The constraint syntax above is the result of discussions with multiple companies such as Microsoft and Hortonworks, and the open-source community. In our model, we wanted to make sure that we could express all their practical use cases.

Even though we cannot express constraints of the form "spread all spark containers across 3 racks", (see §3.4) we can indirectly achieve similar placements using the cardinality constraints i.e., place up to 3 spark containers on each rack. We decided to not complicate our syntax to support these constraints.

MEDEA currently only allows constraints on LRAs. However, in §5.6, we discuss how we could allow task-based jobs to express constraints targeting LRAs, e.g., to have a map/reduce job be placed on the same rack as a Memcached application.

Note also that in MEDEA we focus on building the scheduling infrastructure that enables placement constraints. Automatically inferring the most appropriate and beneficial constraints for each application and cluster is the focus of future work.

5.3 Scheduling long-running applications

The previous section introduced the syntax of our placement constraints, relying on the notion of container tags and node groups to be expressive and high-level. In this section, we describe our heuristic- and optimisation-based scheduling algorithms that receive resource requests utilising the constraint syntax as input and provide high-quality container placements as output.

We first give an overview of our scheduling approach in §5.3.1, and then present our ILP-based scheduling algorithm for LRAs. As we discussed in §2.3.3, container placement is crucial to satisfy global cluster objectives, such as balancing cluster load and reducing resource fragmented machines. Our ILP scheduling algorithm optimises placements for such cluster objectives, as we describe in §5.3.2. Finally, in §5.3.3, we consider heuristic-based algorithms that trade placement quality for scheduling latency.

5.3.1 Overview

When the LRA scheduler is invoked (at configurable intervals), it considers the following information: (i) the container requests along with the placement constraints of the newly submitted LRAs; (ii) the constraints of already deployed LRAs and the cluster operator constraints that are accessible via the constraint manager component (see Figure 5.1); and (iii) the available resources at each node such as

Symbol	Description
k	Number of LRAs to be placed
N	Number of cluster nodes
T_i	Number of containers of LRA i
R_n^f, R_n^u	Free, used resources of node n[2]
m	Total number of constraints
w_i	Weights of components in objective function
B_n, D_n	Sufficiently large integers, used in inequalities
S_i	1 if all containers of LRA i are placed; 0 otherwise
X_{ijn}	1 if container j of LRA i placed at node n; 0 otherwise
r_{ij}	Resource demand of container j of LRA i
r_{min}	Minimum resource demand
$c_{min}^{l,v}$, $c_{max}^{l,v}$	Violation of cardinalities c_{min}, c_{max} for constraint C_l
v_c^l	Violation for constraint C_l
z_n	1 if free resources $\geq r_{min}$ after placement; 0 otherwise

Table 5.1 Notation used in ILP formulation (constants appear above the dashed line, variables below)

memory and CPU in the cluster. It then determines the cluster node at which each LRA container should be placed.

When determining the LRA container placement, our scheduling algorithm attempts to: (i) place all the containers of an LRA (no half-placed applications); (ii) satisfy the placement constraints of the newly submitted LRAs as well as the constraints of the previously deployed ones, and the ones submitted by the cluster operator; (iii) respect resource capacities of all the nodes in the cluster; and (iv) optimise for global cluster objectives such as balancing cluster load and minimising resource fragmentation (see §2.3.3).

As mentioned in §5.1, the LRA scheduler is invoked at regular intervals that are configured by the cluster operator. Longer intervals may increase the scheduling latency for LRAs, but can allow multiple LRAs to be considered together and achieve better container placements, thus improving placement quality. We experimentally study this trade-off between scheduling latency and placement quality in §5.5.

5.3.2 ILP-based scheduling

LRA placement is an optimisation problem under a set of constraints, and can thus be expressed as an integer linear programming (ILP) problem. For completeness, Figure 5.3 provides the ILP

$$\text{maximise} \quad \frac{w_1}{k}\sum_{i=1}^{k} S_i + \frac{w_2}{m}\sum_{l=1}^{m} v_c^l + \frac{w_3}{N}\sum_{n=1}^{N} z_n \tag{5.1}$$

subject to:

$$\forall i,j: \quad \sum_{n=1}^{N} X_{ijn} \leq 1 \tag{5.2}$$

$$\forall n: \quad \sum_{i=1}^{k}\sum_{j=1}^{T_i} r_{ij} \cdot X_{ijn} \leq R_n^f \tag{5.3}$$

$$\forall i: \quad \sum_{n=1}^{N}\sum_{j=1}^{T_i} X_{ijn} - T_i S_i = 0 \tag{5.4}$$

$$\forall n: \quad \sum_{i=1}^{k}\sum_{j=1}^{T_i} r_{ij} \cdot X_{ijn} - B_n(1 - z_n) \leq R_n^f - r_{min} \tag{5.5}$$

For each constraint $C_l = \{\texttt{s_tag}, \{\texttt{c_tag}, c_{min}^l, c_{max}^l\}, \texttt{G}\}$,
$\forall$ container $t_{i_s j_s} \in \texttt{s_tag}$, $\forall$ node set $\mathscr{S} \in \texttt{G}$:

$$\sum_{n \in \mathscr{S}} \left(\sum_{\substack{i,j:\texttt{tag} \in t_{ij} \\ t_{ij} \neq t_{i_s j_s}}} X_{ijn} + D_n(1 - X_{i_s j_s n}) \right) - c_{min}^l + c_{min}^{l,v} \geq 0 \tag{5.6}$$

$$\sum_{n \in \mathscr{S}} \left(\sum_{\substack{i,j:\texttt{tag} \in t_{ij} \\ t_{ij} \neq t_{i_s j_s}}} X_{ijn} - D_n(1 - X_{i_s j_s n}) \right) - c_{max}^l - c_{max}^{l,v} \leq 0 \tag{5.7}$$

$$v_c^l = \frac{c_{min}^{l,v} + 1}{c_{min}^l + 1} + \frac{c_{max}^{l,v} + 1}{c_{max}^l + 1} \tag{5.8}$$

Figure 5.3 ILP formulation

formulation, relying on the notation from Table 5.1. Below, we give a more intuitive description of the optimisation problem.

Consider k LRAs that are submitted by application owners in the latest scheduling interval and must be scheduled on a cluster that consists of N-nodes.

Objective. Our objective function as defined by Equation 5.1 and has three components: (1) it aims to place as many of the k LRAs as possible; (2) it minimises the number of constraint violations of

[2]For simplicity, we use a single scalar value for the cluster resources. However, our model can be extended to use a vector of resources instead, i.e., having separate equations for each resource type.

these LRAs (more on this below); and (3) it avoids cluster resource fragmentation by minimising the number of nodes left with too few resources (Equation 5.5).

Note that in order to combine these components linearly, independently of their range or units, we normalise each component to take values from 0 to 1. We also use weights (w_1–w_3) to assign different priorities to components. The cluster administrator is responsible for setting these weights based on the desired cluster behaviour. In Equation 5.1, we include the components that we use in our evaluation clusters within Microsoft, but additional ones can be easily added, such as load imbalance or minimising the number of nodes used for placement.

We also make sure that each container of an LRA is placed at most once, respecting node capacities. More importantly, we make sure that we place either all or none of an LRA's containers, as it would be more challenging to deal with half-placed applications (Equation 5.2 to Equation 5.4).

Placement constraints and violations. For each placement constraint that belongs to a newly submitted or already deployed LRA or to the cluster operator, we use two inequalities to impose the constraint semantics (see §5.2): one for the minimum c_{min} and one for the maximum c_{max} cardinality (Equation 5.6 and Equation 5.7). In case of a constraint violation, which is more common in heavily utilised clusters or when dealing with restrictive constraints, we also quantify the extend of the violation relative to the values of c_{min} and c_{max} using Equation 5.8. By minimising the extend of constraint violation of an LRA, we can guarantee that even when the ideal placement cannot be achieved, our algorithm provides the best placement alternative.

Resolution of constraint conflicts. When placing an LRA, the set of constraints considered may include conflicts, e.g., one constraint requesting no more than three Spark containers in a rack, and another one at least four. In such cases, cluster operator constraints are always considered higher-priority and override the application constraints, as long as they are more restrictive. In case of conflicting application constraints that are of the same priority, our ILP formulation favours the placement that minimises the number of violations.

Compound constraints. To support more complex, compound constraints that are expressed in disjunctive normal form, or DNF (§5.2). MEDEA treats each DNF conjunct as a separate constraint and adds an extra inequality in the formulation. As a result, MEDEA can guarantee that at least one of these constraints is finally met.

5.3.3 Heuristic-based scheduling

Even though MEDEA uses the ILP scheduling algorithm only to place the LRA containers that are associated with placement constraints, as the cluster size or the constraint complexity increases or under high cluster load, the integer linear problem complexity may lead to high scheduling latency.

To avoid compromising placement quality by terminating the ILP solver early [133], we also consider simpler scheduling approaches using heuristics.

We experiment with our heuristics in order to examine whether simpler approaches than ILP are sufficient to achieve high quality LRA placements. Next, we discuss the ones that gave us the best results in practice. An experimental comparison between the heuristics and the ILP scheduling algorithm is given in §5.5.

The **tag popularity** heuristic prioritises the placement of containers that are associated with tags appearing in the largest number of constraints. In this heuristic, we track the number of times each tag is associated with existing constraints across the whole cluster, as stored in the constraint manager component. When new LRA container requests are received, they are ordered based on their most popular tag (highest number of occurrences) and the top ones are satisfied first. The intuition is that such container requests with popular tags often have more complex constraints and are thus harder to place in a cluster.

The **node candidates** heuristic first calculates the number of nodes N_c on which a given container is allowed to be placed, subject to all placement constraints. Then, it places the container with the smallest N_c, as that container has the least placement flexibility. Note that N_c values need to be recalculated after each container placement. We avoid this by recalculating N_c only for containers whose placement opportunities were affected in the previous iteration. We keep track of the affected containers by their tags, so a container placed in the previous iteration only triggers the recalculation of node candidates for the containers with tags in common.

5.4 Implementation

In this section, we describe how our two-scheduler design with support for placement constraints can be realised in a system implementation as part of an existing cluster manager. Our implementation of MEDEA is built as an extension to Apache Hadoop YARN [135], following the design described in §5.1. Moreover, we open-sourced our implementation, which is now included in the Apache Hadoop 3.1 release with the detailed open-sourcing effort tracked in [145].

We start by describing the main components of YARN's architecture, as shown in Figure 5.4. YARN follows a centralised architecture where a single logical component, the *Resource Manager* (RM), allocates resources on *Node Managers* (NMs) for applications submitted to the cluster. Applications submit generic resource requests that are handled by the RM, while any specific scheduling logic required by each application is implemented as part of the *Application Master* that any framework can extend. This enables YARN to support a wide variety of applications using the same centralised resource management component with generic scheduling logic.

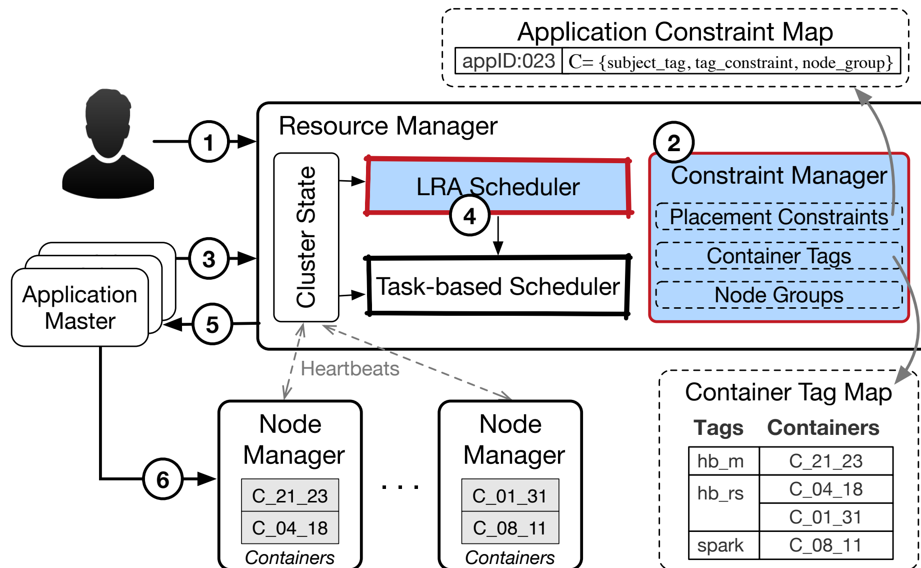


Figure 5.4 MEDEA architecture

Resource Manager. The main role of the RM is to arbitrate resources among submitted applications. RM usually runs on a dedicated cluster machine, while for high-availability multiple RMs can be deployed, with one of them serving as the master. The node managers periodically report their status to the RM through a heartbeat mechanism for scalability. At any point in time, the RM maintains all the pending resource requests of all submitted applications. Given the cluster state and based on application demand, priorities, and sharing policies (e.g., fairness), the RM scheduler is responsible to find the best match of cluster machines to application requests. It then hands leases on containers to applications. A container forms a package of resources on a particular node (e.g., 4 GB of RAM and 2 CPU on NODE_1).

YARN includes two publicly available scheduler implementations, the FAIR Scheduler [147] and the Capacity Scheduler [146]. The FAIR Scheduler imposes fairness across applications; the Capacity Scheduler focuses on ensuring predefined shares of cluster resources to groups of users (capacity planning). When applications are submitted to the RM and before they are considered by the scheduler, they also go through an admission control phase. During that phase, the user credentials of an application are validated and a variety of operational checks are performed.

Application Master. Each submitted application initiates an AM, which acts as the resource orchestrator for the application. The AM manages all the resource life-cycle aspects of the application, including managing the execution flow e.g., initiating reducer containers after map tasks are done, dynamically increasing and decreasing resource consumption according to the application needs, and handling faults. The AM can run arbitrary user code, written in a plethora of programming languages. By delegating the above functionality to the AM component, YARN's architecture can achieve significant flexibility and also scalability. Complex logic can be part of the AM.

To run an application, an AM typically utilises resources from multiple nodes in the cluster. To obtain containers, the AM issues resource requests to the RM via the heartbeat mechanism. When the RM scheduler assigns a resource to the AM, it also generates a lease. The AM is then notified and presents the container lease to the NM for launching the container on that machine. The NM authenticates the lease and initiates the container localisation and execution process.

Node Manager. Each worker node in the cluster runs a daemon process called a node manager. Node managers are responsible for: (i) launching containers and managing their life-cycle including start, end, status, and killing of containers on each machine; and (ii) monitoring resource availability at the host machine such as available CPU, disk and memory metrics, while reporting possible faults.

MEDEA. To implement MEDEA, we mainly extend YARN's resource manager. Figure 5.4 shows in blue the new components added to YARN's RM, namely the *constraint manager* (CM) and the *LRA scheduler*. The CM component is responsible for storing container tags and node groups, along with application-specific and cluster-wide placement constraints as described in §5.2. The LRA scheduler determines the placement of LRA containers to nodes, given the placement constraints and the cluster condition. To this end, it uses either the ILP-based (relying on the CPLEX solver [28]) or the heuristic-based scheduling algorithms from §5.3. For the task-based scheduler that handles applications with no placement constraints, we use YARN's Capacity Scheduler [146]. The FAIR Scheduler [147] can be used instead to ensure fairness across applications, simply by changing a configuration parameter.

LRA life-cycle. Figure 5.4 also shows the life-cycle of an LRA in MEDEA:

1. The client first submits an LRA to the cluster, including a set of tags for its containers and a set of placement constraints.

2. When the Resource Manager receives the LRA, the Constraint Manager validates and stores the associated constraints, and then the job-specific application manager (AM) is initialised.

3. The AM petitions the RM for cluster resources based on the resource requirements of its containers: These LRA requests are intercepted by a special pre-processor component (omitted from the figure) and are then forwarded to the LRA scheduler.

4. The LRA scheduler takes into account the relevant constraints, the cluster's node groups and the current container tags stored in the CM, as well as the available resources in the cluster. It finds the best placement for the LRA containers based on the pre-configured objective function (see §5.3.2). This placement is then passed to the task-based scheduler that performs the actual allocation (see §5.1).

5. When the task-based scheduler performs the actual container allocation, the RM notifies the AM about the new container leases.

6. The AM finally uses the leases to dispatch the containers of an application for execution to the node managers.

5.5 Evaluation

In this section, we evaluate MEDEA, our cluster manager prototype with explicit support for rich placement constraints that can achieve high-quality container placement for LRAs as well as fast container allocations for traditional task-based jobs. MEDEA is the first cluster manager that provides support for the expressive constraints of our unified dataflow model to effectively place its long-running containers. We first give an overview of the section.

Overview of evaluation. We first describe the experimental set-up of our evaluation along with the different systems, workloads, and placement constraints used for our comparisons (§5.5.1). We then study the impact of MEDEA in a real cluster environment and analyse the benefits in terms of application performance and resilience compared to existing state-of-the-art systems (§5.5.2–§5.5.3). We also focus on MEDEA meeting our pre-defined global cluster objectives including resource fragmentation, load balancing and high placement quality, as LRAs occupy an increasing amount of resources in the cluster, as constraints grow in complexity, or as we increase the period of our scheduling actions (§5.5.4). Finally, using simulations, we investigate whether MEDEA can maintain low scheduling latency as clusters grow in size and as cluster utilisation increases (§5.5.5). In the experiments below, we want to study:

- the impact of MEDEA on LRA performance (§5.5.2) and resilience (§5.5.3) as part of a real production environment with shared cluster workloads;
- the achieved global objectives of MEDEA (§5.5.4) and the different trade-offs between our optimisation based algorithm and the variety of heuristic-based ones; and
- the scheduling latency of our LRA scheduler as we increase the percentage of LRAs in the cluster and as we scale the simulated cluster size up to 5000 machines (§5.5.5).

5.5.1 Experimental set-up

Cluster set-up. To evaluate MEDEA on a real deployment, we use a pre-production cluster within Microsoft consisting of 400 machines grouped into 10 racks. Each machine has a dual quad-core Intel Xeon E5-2660 CPU with hyper-threading (HT), 128 GB of RAM, and ten 3 TB data drives, not configured in a RAID (JBOD). The network supports 10 Gbps within and 6 Gbps across racks.

Simulation. To experiment with multiple configurations and an even higher number of machines, we use a simulator that executes MEDEA with simulated machines, merely ignoring RPCs and

task execution. To drive the simulations, we use GridMix [59], an open-source benchmark that uses workload traces for generating synthetic jobs for Hadoop clusters. We extend its synthetic workload functionality to produce LRAs with custom constraints. For the simulation, we are also using Tez [129] as the execution framework for the short-running batch jobs.

Workload. To emulate a real shared cluster workload, we use the following applications:

- HBase [62] application instances, with ten workers each. We use the YCSB benchmark [26] with one billion records, for a total of 1 TB of data, and submit six YCSB workloads, A–F, using one YCSB client per HBase instance. YCSB can be configured to produce specific access pattern, I/O size, and read/write ratios, and performs 1 KB accesses from a single table with configurable read/write ratios using: Zipfian (featuring record popularity), Latest (featuring record freshness), or Uniformly-random probability distributions. The details of the particular workloads (A–F) can be found in Table 5.2.
- TensorFlow [1] framework instances, each with 8 workers and 2 parameter servers. For TensorFlow, we run machine learning workloads that involve one million iterations per run.
- Batch Tez [129] task-based jobs, generated using the GridMix synthetic workload generator [59], resembling some production workloads within Microsoft. For the generation of production workloads, we are using traces similar to the ones used in the evaluation of Yaq [111] cluster scheduler, each consisting of 100 tasks per job.

For the deployment of the above applications, we use <2 GB, 1 CPU> containers for the HBase and TensorFlow workers, <4 GB, 1 CPU> containers for the TensorFlow master-workers, and <1 GB, 1 CPU> containers for the rest.

Workload	Operations	Distribution	Application
A-Update heavy	Read: 50% Update: 50%	Zipfian	User session store recording user's recent actions
B-Read heavy	Read 95% Update 5%	Zipfian	Photo tagging: add a tag is an update, but most operations tag reads
C-Read only	Read 100%	Zipfian	User profile cache, where profiles are constructed elsewhere
D-Read latest	Read 95% Insert 5%	Latest	User status updates, people want to read the latest statuses
E-Short ranges	Scan 95% Insert 5%	Zipfian Uniform	Threaded conversations, each scan is for the posts in a given thread
F-Read write	Read 50% Write 50%	Zipfian	User database, user records are read-modify or store user activity

Table 5.2 YCSB workload properties (A–F) [26]

Placement constraints. For the deployment of the LRAs, we also assume some specific placement constraints. Unless otherwise specified, we use the following placement constraints when deploying the HBase and TensorFlow LRA instances.

- To minimise network traffic and reduce network costs, we use the intra-application affinity constraint that all workers within the same HBase or TensorFlow application instance should reside on the same rack.
- To minimise resource interference across applications that compete for the same resources, we impose the inter-application cardinality constraint that no more than two HBase, and no more than four TensorFlow workers, are placed on the same node.
- For the HBase application, we request an extra node affinity constraint between the HBase Master and the Thrift Server to reduce network communication costs. We also request a node anti-affinity constraint between the HBase Master and Secondary to increase application resilience.

Note here that the maximum cardinality used for each application was decided empirically after experimenting with different values. Our goal in MEDEA is to build the scheduling infrastructure for expressive placement constraints and not infer them (see also the discussion in §5.2.1 on automatically inferring constraints).

Comparisons. In our evaluation, we compare the following systems:

- MEDEA-ILP (or MEDEA): This is our optimisation-based (ILP) scheduling algorithm, as described in detail in §5.3.2. For the objective function (Equation 5.1), we use weights $w_1=1$, $w_2=0.5$ and $w_3=0.25$, i.e., we give higher priority to maximising the number of scheduled LRAs and then to minimising constraint violations and resource fragmentation. For triggering the scheduling algorithm, we use an interval of 10 secs.
- MEDEA-NC and MEDEA-TP: These are our heuristic-based algorithms as described in §5.3.3. For the heuristic-based scheduling algorithms, we still consider multiple LRAs at once, similar to MEDEA-ILP, using a 10 sec scheduling interval.
- SERIAL: This is another heuristic-based algorithm that, unlike MEDEA-NC and MEDEA-TP, does not order the LRA container requests when making scheduling decisions. This algorithm is also using the 10 sec scheduling interval.
- YARN: This is our constraint-unaware production-ready baseline. We use YARN, the default container scheduler within Microsoft's analytics clusters. Our novel scheduler, MEDEA, is also implemented on top of YARN.

- J-KUBE: Kubernetes [82] is the most complete system to date for supporting placement constraints (see Table 2.2). As it follows a different architecture than MEDEA, to have a fair comparison, we implement its scheduling algorithm as part of MEDEA's LRA scheduler. The main differences with MEDEA is that: (i) Kubernetes considers one container request at a time during scheduling; and (ii) supports affinity and anti-affinity but no cardinality constraints.
- J-KUBE++: This is the same scheduler implementation as J-KUBE above, but we extend its functionality to support cardinality constraints. Unlike affinity and anti-affinity constraints, cardinality can control the level of container collocation and interference, as we showed in §2.3.1, and are thus particularly important.

Our implementation of MEDEA is based on Apache Hadoop 2.7.2 [5]. For the deployment of LRAs in the cluster, we use Apache Slider 0.92.0 [118].

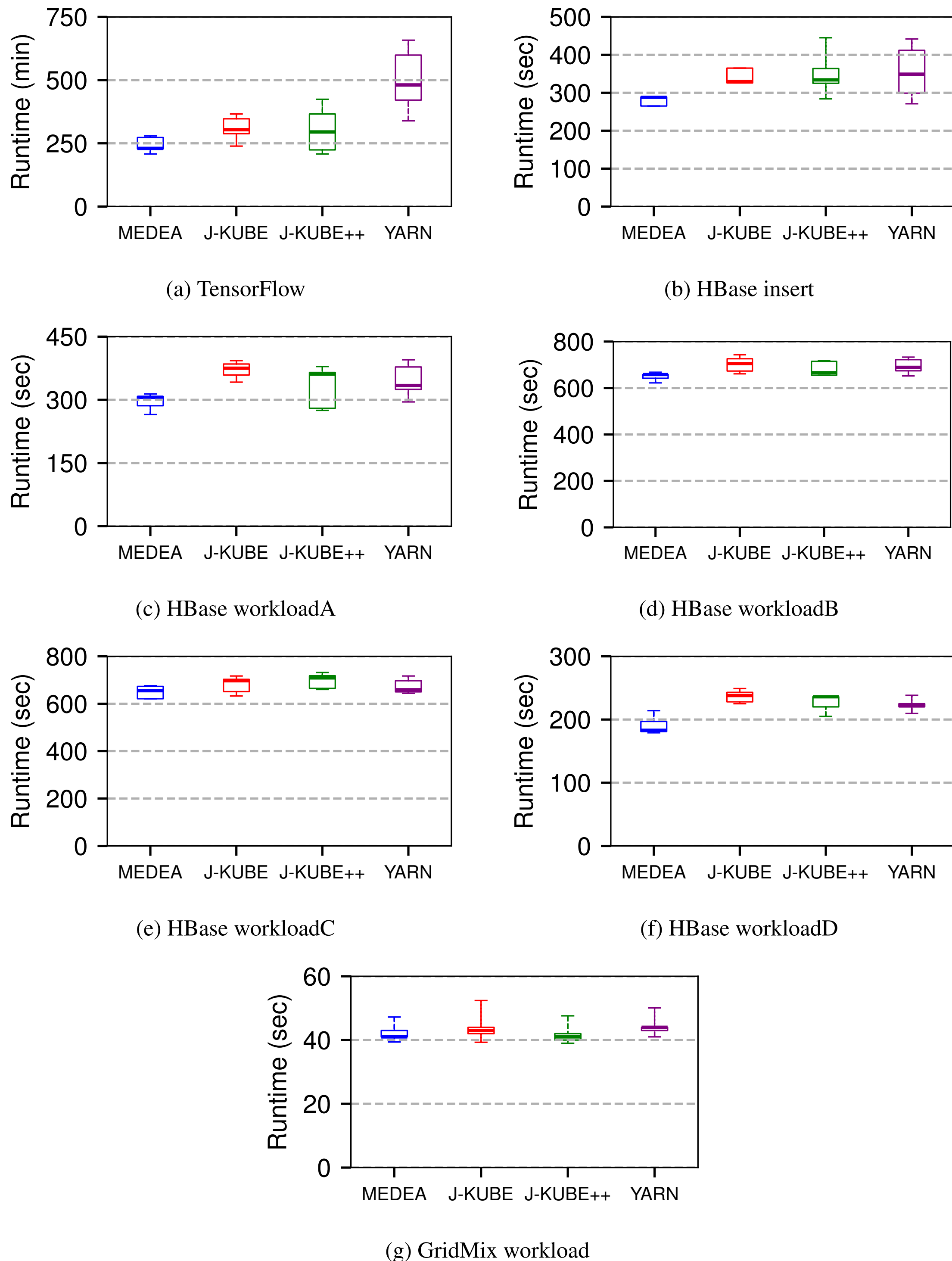
5.5.2 Application performance

We first want to study the impact of MEDEA on end-to-end application performance as part of a real environment. For this experiment, we use the 400-node pre-production cluster described above and deploy 45 TensorFlow and 50 HBase LRA instances using YARN, J-KUBE, J-KUBE++, and MEDEA. At the same time, we submit GridMix task-based jobs using production traces that account for 50% of the cluster's memory total.

The goal for MEDEA is to achieve high-quality placement for the containers of long-running applications that can improve the performance and thus reduce the total runtime of those applications. At the same time, the improved LRA placement and runtime should not affect the traditional task-based jobs that are running in the cluster.

The runtimes for LRAs are shown in Figure 5.5a to Figure 5.5f; Figure 5.5g shows the runtimes for task-based jobs. In more detail, Figure 5.5a illustrates the runtimes for our machine learning workflows on TensorFlow; Figure 5.5b shows the runtimes for data insertion on HBase; and Figure 5.5c to Figure 5.5f show the runtimes of Workload A to Workload D on HBase, respectively. To visualise the distribution of runtimes, we use box plots in our figures in which the lower/upper part of the box represents the 25th and 75th percentiles, respectively; the middle line is the median; and the whiskers are the 5th and 99th percentiles, respectively.

As we can observe in Figure 5.5a–Figure 5.5f, for all LRA workloads, MEDEA consistently outperforms J-KUBE across all percentiles. This is mainly due to J-KUBE's lack of support for cardinality constraints — anti-affinity constraints alone are not sufficient to achieve the best application performance (see §2.3.1). For instance, the median runtime is 32% longer on J-KUBE for TensorFlow and 23% longer for HBase Workload A compared to MEDEA, while the rest of the HBase workloads yield similar benefits.



(a) TensorFlow

(b) HBase insert

(c) HBase workloadA

(d) HBase workloadB

(e) HBase workloadC

(f) HBase workloadD

(g) GridMix workload

Figure 5.5 Application performance (lower is better)

To further study whether cardinality is the only reason for MEDEA s' benefit, we extend J-KUBE with support for cardinality constraints but still consider a single container request at a time. The resulting system with support for cardinality is J-KUBE++. In Figure 5.5a–Figure 5.5f, we observe that MEDEA still has improvements of up to 28% when compared to J-KUBE++. More importantly, J-KUBE++ leads to significant variability in runtimes, which is a detrimental issue in production clusters [75]. Even thought the 5th percentile of the distribution in J-KUBE++ is similar to MEDEA, the 99th percentile is up to 54% higher. By considering only one container request at a time, J-KUBE++ often leads to placements with many constraint violations (see §5.5.4), which are also not consistent across application instances. For instances with many violations, the benefit of MEDEA is higher.

MEDEA's benefits are even more pronounced when compared with the production-ready YARN configuration, that does not support constraints. MEDEA can achieve a runtime that is up to 2.1× shorter for the median and up to 2.4× shorter for the 99th percentile compared to YARN. More importantly, YARN leads to high runtime unpredictability, as some constraints are randomly satisfied for some LRAs and violated for the majority of them.

Finally, one of the main requirements for MEDEA was that the high-quality placement for LRAs should not come at the expense of task-based jobs running at the cluster. Figure 5.5g shows that MEDEA's benefits do affect task-based jobs, as the runtimes of task-based jobs from production workloads are consistently similar across all schedulers.

Overall, in a 400-node pre-production cluster, MEDEA significantly improves both the performance and the predictability of LRAs compared to state-of-the-art schedulers such as YARN and Kubernetes, without affecting the performance of task-based jobs.

5.5.3 Application resilience

Next, we want to assess the effectiveness of MEDEA for improving application resilience. Using MEDEA, we want to achieve high-quality placements for LRAs that guard them from losing multiple containers at once a situation that can potentially affect their performance and recovery.

For this experiment, we use real unavailability data from a Microsoft production cluster that is used by analytics applications. The cluster comprises a few tens of thousands of machines with similar hardware characteristics that are grouped into 25 logical node groups, also known as service units (see §2.3.2).

For each service unit we collect the number of unavailable machines due to failures, machine upgrades, maintenance, etc. We collect data for each hour over a period of 15 days. Then we generate LRAs, with 100 containers each, and place them with the simple intra-application anti-affinity constraint that containers of the same LRA should be spread across service units, using the MEDEA and J-

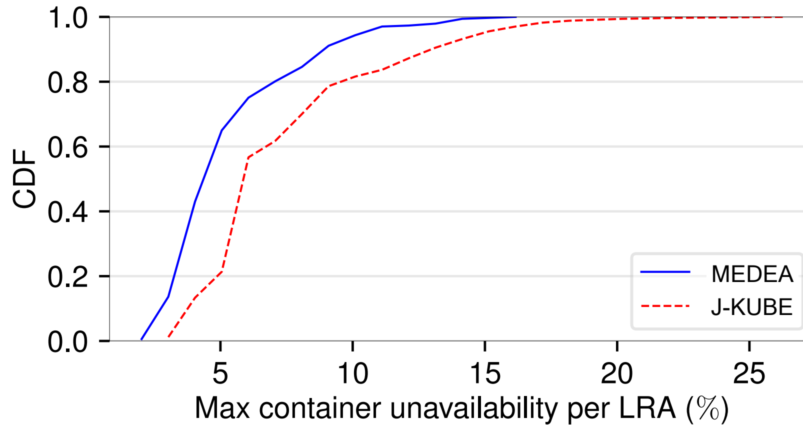


Figure 5.6 Application resilience over 15 days

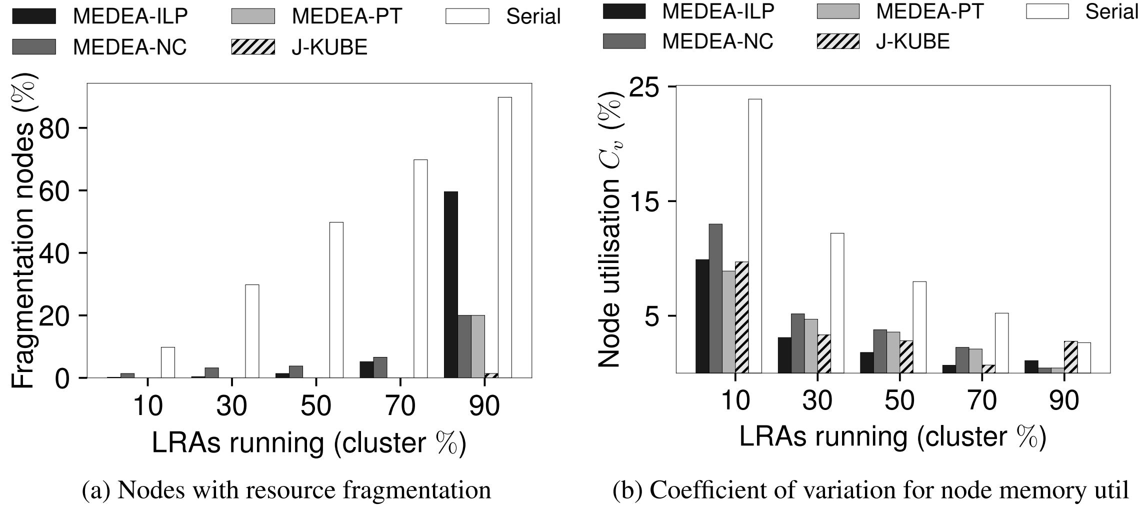
KUBE schedulers. Given the machine unavailability captured in our collected data and the container placements achieved by the schedulers, we pick for each hour the LRA with the highest percentage of unavailable containers. The intuition here is that a scheduler that can achieve a good placement without violating the anti-affinity constraint will not place more than 4 container on the same service unit. As a result, when a service unit fails, the LRA will not lose more than 4 containers at once.

As shown in Figure 5.6, unlike J-KUBE, MEDEA maintains a maximum unavailability of containers close to 4, even for higher percentiles. In general, MEDEA's placement leads to fewer anti-affinity constraint violations compared to J-KUBE, and this leads to lower container unavailability across all percentiles. MEDEA improves the median and maximum unavailability percentage by 16% and 24%, respectively, compared to J-KUBE, which is crucial for a production environment.

5.5.4 Global cluster objectives

We now focus on MEDEA's ability to satisfy global cluster objectives, as described in §2.3.3, using our scheduling algorithms presented in §5.3. In this experiment, we want to study the behaviour of various scheduling algorithms under different scenarios varying: LRAs and task-based jobs running in the cluster, LRA algorithm periodicity, and LRA constraint complexity. We compare the scheduling algorithms by their ability to reach global objectives such as balancing cluster load, minimising cluster resource fragmentation, and constraint violations.

For the experiments below, we use a simulated cluster of 500 machines with 8 CPU cores and 16 GB RAM each, split across 10 racks. We generate HBase instances using the constraints mentioned in §5.5.1 (intra-application rack affinity across workers, inter-application cardinality of 2 workers within nodes, intra-application master-thrift server node affinity, intra-application master-secondary node anti-affinity). We conduct five experiments and observe for each scheduling algorithm: (i) the percentage of containers that violate constraints; (ii) the percentage of nodes that are fragmented (i.e.,



(a) Nodes with resource fragmentation

(b) Coefficient of variation for node memory util

Figure 5.7 Load balance with varying LRA utilisation

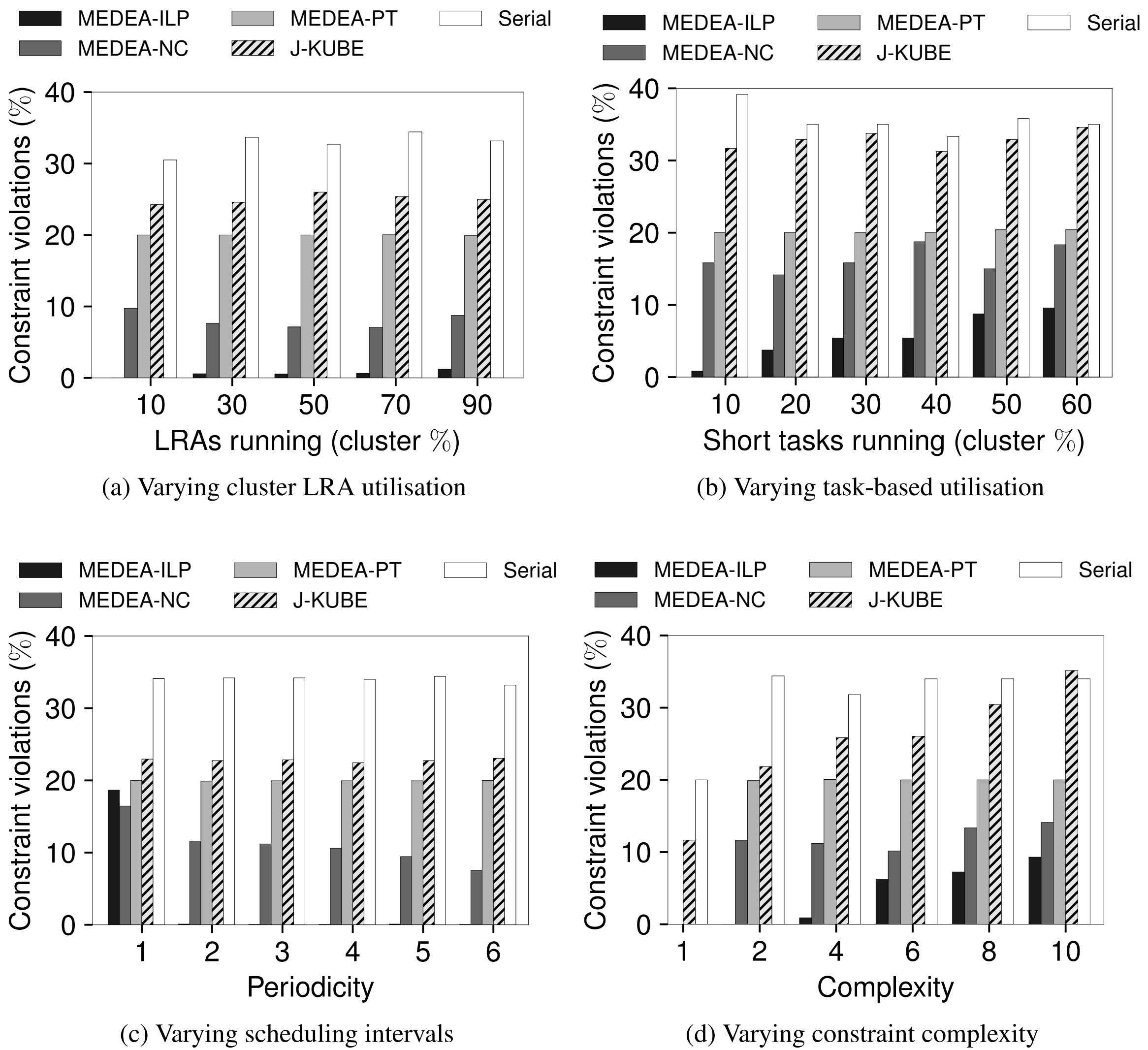
have less than 1 core/2 GB RAM and are not fully utilised); and (iii) the coefficient of variation of nodes' memory utilisation as a proxy for load imbalance.

First, we vary the number of submitted LRAs, accounting for a total cluster memory utilisation ranging from 10% to 90% — no task-based jobs are used in this experiment. Our results are shown in Figure 5.7a, Figure 5.7b, and Figure 5.8a. The scheduling interval for our MEDEA algorithms with periodicity including MEDEA-ILP, -NC, and -TP is such that two LRAs are considered on each scheduling cycle.

As we can observe in Figure 5.7a, all MEDEA algorithms, as well as J-KUBE, lead to a low number of fragmented nodes, except under high utilisations (90% of cluster resources). An exception is the SERIAL scheduling algorithm that consistently faces an increased number of fragmented nodes. In our experiments, we notice that, by considering requests sequentially (one at a time), it is not uncommon to end up in a situation in which the algorithm is forced to place a container in a machine that is close to fragmentation. This happens, for example, when the container to be placed is marked with an affinity constraint and the target container is already running on the highly utilised machine.

As shown in Figure 5.7b, all algorithms, apart from SERIAL, perform similarly in terms of cluster load imbalance. Again, considering container requests sequentially can lead the algorithm into a situation which it is forced to choose a machine that is over-utilised based on (bad) previous scheduling decisions. Notice here that load imbalance is more pronounced for low utilisations — each scheduling decision has more impact compared to higher utilised clusters where the load evens out.

MEDEA-ILP, thanks to its objective function and the consideration of multiple container requests at a time, leads to almost no constraint violations, even for 90% utilisation, as we can observe in Figure 5.8a. In contrast, our heuristic algorithms result in 10%–20% of violations, even for low



(a) Varying cluster LRA utilisation

(b) Varying task-based utilisation

(c) Varying scheduling intervals

(d) Varying constraint complexity

Figure 5.8 Constraint violations

utilisation, despite also considering multiple requests at a time. J-KUBE, handling one request at a time, exhibits even more violations. Note that, for all schedulers, constraint violations do not significantly increase with utilisation, as most of our constraints in this experiment are intra-application i.e., placements mostly violate the stricter intra-application constraints.

We also measure constraint violations using the the same simulated cluster as above and LRAs with a stable utilisation of 10%. At the same time, we use task-based jobs with varying utilisation from 10% to 60%. In Figure 5.8b, we observe similar trends for constraint violations as above: MEDEA-ILP yields less than 10% of violations even for the higher utilisation; the other scheduling algorithms lead to more than 15% and up to 40% of violations. Even though J-KUBE effectively balances cluster

load and minimises node fragmentation, it cannot optimise the placement of containers and leads to increased constraint violations compared to MEDEA-ILP.

In the next experiment, we use the simulated cluster and vary the scheduling interval, keeping a stable 10% LRA utilisation. By changing the scheduling interval, we affect the number of LRAs considered at each scheduler invocation (periodicity in the figure). With periodicity of 1, similar to the other scheduling algorithms, MEDEA-ILP also exhibits violations, as shown in Figure 5.8c. However, increasing periodicity reduces violations for MEDEA-ILP and -NC. This observation highlights the importance of periodicity — considering multiple container requests at a time for satisfying inter-application constraints.

Finally, we perform an experiment using the simulated cluster and LRAs that consistently utilise 10% of the cluster. We gradually increase the complexity of LRA placement constraints (complexity X means that we have affinity or cardinality inter-application constraints involving up to X LRAs). Figure 5.8d reports the violations for constraints, as we vary the constraint complexity. We can observe in the figure that, even with constraints involving 10 LRAs, MEDEA-ILP results in less than 10% of violations. MEDEA-NC and -TP heuristics also perform relatively well with less than $<$20% of violations. In contrast, J-KUBE has more than 20% of violations. By considering only one request at a time, it is difficult to satisfy the increasing complexity of inter-application constraints.

5.5.5 Scheduling latency

Finally, we want to investigate how our LRA scheduler scales in terms of scheduling latency and how it affects the scheduling of short-running jobs. We first use different scheduling algorithms to place LRAs and vary the simulated cluster size from 50 to 5000 machines.

Then, we focus on our best-performing MEDEA-ILP algorithm and schedule LRAs, as we increase the fraction of resources used by LRAs in the cluster. To conclude, we study how MEDEA impacts the scheduling of short-running containers using the publicly available Google cluster trace [139].

For MEDEA, we want the LRA scheduler to maintain low scheduling latency even at an increasing number of placed LRAs, or numbers of machines in the cluster. At the same time, the scheduling latency of short-running jobs should remain unaffected.

First, we use an increasing cluster size from 50 to 5000 machines, and we generate LRAs to consume 20% of the total cluster resources each time. Then, we measure the average scheduling latency for placing all containers of an LRA, as shown in Figure 5.9. As we can observe in the figure, our heuristic algorithms achieve the lowest latencies, with MEDEA-NC being more expensive. J-KUBE leads to higher latencies, due to the frequent scoring of nodes, even though we believe we can further optimise our implementation via smart caching of node scores. Although MEDEA-ILP has the highest latency, even with 5000 machines, the average latency is 850 ms, which is low compared to the typical

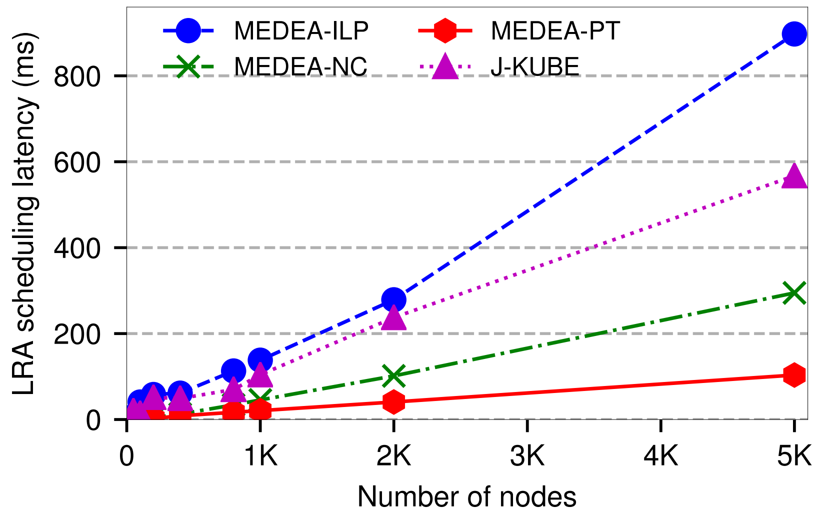


Figure 5.9 Varying cluster size

execution times of LRAs (hours, days or months). For larger clusters in which lower scheduling latency is a requirement, MEDEA-NC and J-KUBE can yield reasonable compromises in terms of latency and quality. In practical scenarios, MEDEA-ILP is the better choice because it combines relatively low latency with better placement quality.

Our two-scheduler approach in MEDEA is different from existing schedulers such as TetriSched [133] that use an ILP scheduling algorithm to place all requests and not just the ones with constraints. As we discuss below, this can be problematic under high load because the scheduling latency or the placement quality gets compromised. The Firmament [57] scheduler also follows similar logic using a single graph-based scheduler to support placement constraints and place all container requests. However, adding more machines or constraints would require (an exponential number of) additional vertices in the scheduling graph, consequently increasing scheduling latency.

In the following experiment, we want to assess the benefit of our two-scheduler design. For this purpose, we compare MEDEA-ILP using with a modified version that uses the solver for scheduling both long- and short-running containers in the cluster. We call this modified scheduler version ILP-ALL.

To study the benefit our two-scheduler design, we use a 256-machine simulated cluster and generate LRAs and task-based jobs that result in a fully-utilised cluster. At the same time, we increase the percentage of resources occupied by LRAs from 0% to 100% — 0% practically means that all cluster resources are fully occupied by task-based jobs and 100% that all resources are used by long-running applications.

As we vary the fraction of cluster resources used by LRAs, we measure the total scheduling latency for LRAs, as shown in Figure 5.10. The single-scheduler design (ILP-ALL) using the ILP scheduler to place both short- and long-running containers, results in an increased scheduling latency. More

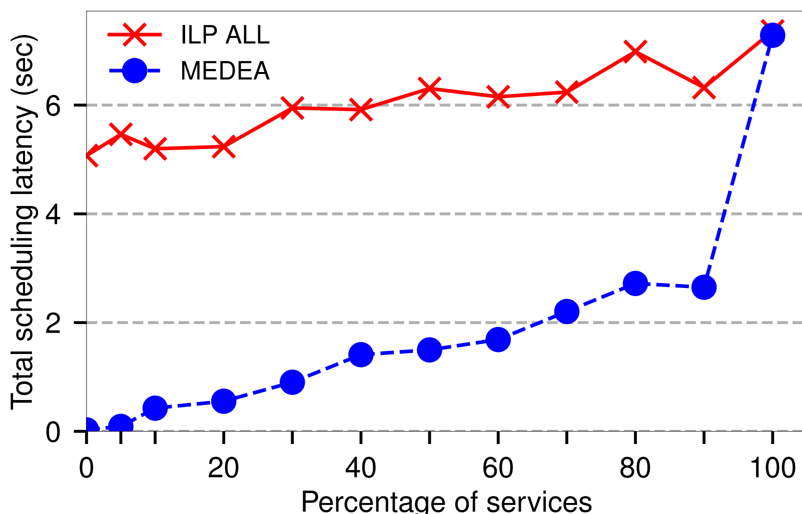


Figure 5.10 Two-scheduler benefit

precisely, the scheduling latency of ILP-ALL is 9.5× worse than MEDEA when LRAs utilise 20% of the cluster, while it is less than half when LRAs occupy 90% of cluster resources. This justifies the rationale behind MEDEA's use of the more expensive ILP solver to schedule only for the applications that affected by container placement the most, LRAs.

Finally, we study the impact of MEDEA's two-scheduler design on scheduling short-running containers using the Google cluster trace [139]. The goal of this experiment is to show that the LRA scheduler is not on the critical path of the task-based scheduler, and thus does not affect the scheduling latency of task-based (non-LRA) applications. To this end, we use the Google trace, which we speed up by a factor of 200× to increase the load on the scheduler in terms of scheduling decisions per second. Another reason for speeding up the trace is to examine whether the scheduler can keep up with this load in the same way as the original YARN scheduler. For this experiment, we simulated a 12,500 node cluster, as dictated by the trace.

In Figure 5.11, we report the scheduling latency achieved by MEDEA-ILP and YARN when placing the trace tasks using box plots in which the lower/upper part of the box represents the 25th and 75th percentiles, respectively; the middle line is the median; and the whiskers are the 5th and 99th percentiles, respectively. In the case of MEDEA, we add an additional 10% scheduling load coming from LRAs (scheduled by the ILP algorithm). As we can observe in the figure, despite the additional LRA load, MEDEA achieves scheduling latencies similar to those of YARN. For the lower percentiles, the latency distributions are similar for MEDEA and YARN while for the upper 99th percentile the difference is less than 20%. This shows that MEDEA's LRA scheduler does not impact the operation of the underlying task-based scheduler.

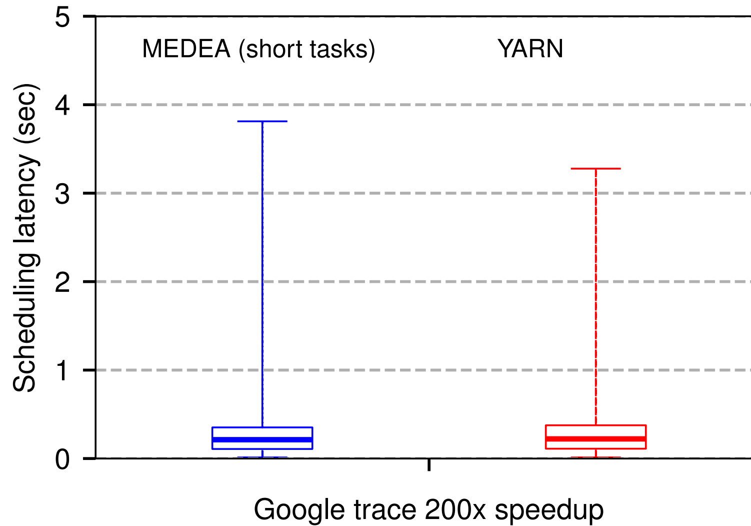


Figure 5.11 Task-based scheduling latency

5.6 Limitations and discussion

In the next paragraphs, we highlight some of the limitations of MEDEA and discuss how they can be addressed as part of future work.

Placement conflicts. Given MEDEA's two-scheduler design described in §5.1, the cluster state may have changed from the moment an LRA is submitted until its containers are allocated, not allowing the placement of the LRA due to conflicting allocations by task-based jobs. Possible solutions to this problem are: (i) killing containers of task-based jobs to free up resources for LRAs; (ii) relaxing some LRA constraints, e.g., by placing an LRA container on the corresponding rack instead of the specified node; and (iii) allowing LRAs to reserve cluster resources with constraints in advance. Given that these approaches could significantly impact the execution of task-based jobs and increase the scheduler's complexity. In the current version of MEDEA, we opt for the simpler approach of resubmitting the LRA in case of conflict. However, we can easily support container preemption or reservations in the future.

Container migration. Another interesting future direction is extending the ILP formulation to also allow container migration. This direction shares commonalities with some VM scheduling approaches that address resource overload and interference issues or non-viable placements through VM migration [63, 140]. To do so, the extended ILP algorithm will have to take into account the migration cost of existing long-running containers and their state. The objective function will include a new component with a separate weight targeting to minimise the number of executing long-running containers that will have to be migrated.

Container migration could be used as a reactive mechanism combined with MEDEA's proactive approach in achieving high quality LRA placements. This mechanism can be particularly useful under high cluster load, when LRAs enter and leave the system at high rates or when their resource demands change over time. As part of future work, we can extend our ILP formulation to enable migration and account for the extra migration cost in our objective function.

Constraints for task-based jobs. Our focus in MEDEA is on supporting placement constraints and providing high-quality placement for LRAs. However, task-based jobs may also require placement constraints to increase their performance (see also §5.2.1). These constraints can refer either to other task-based jobs or LRAs. We can address this by extending the task-based scheduler in MEDEA to support our expressive placement constraints in a heuristic fashion with no optimality guarantees. As a result, neither the scheduling latency of task-based jobs nor overload the (ILP-based) LRA scheduler would be affected.

Resource isolation. Collocated containers running on the same machine can still interfere with each other, even when achieving the most optimal container placement with MEDEA. Existing schedulers mitigate interference between collocated workloads by throttling low priority jobs in favour of LRAs [153], detecting interference though profiling [38, 39, 40], or using hardware-based isolation mechanisms [86]. In MEDEA, we are currently isolating containers on the same machine using cgroups [84]. However, in the future, several of the above techniques can be combined with MEDEA to achieve even better resource isolation.

5.7 Summary

In this chapter, motivated by the largely unexplored long-running application placement requirements, we presented MEDEA, a system for efficiently scheduling applications with long-running containers (LRAs). MEDEA is the first system to fully support complex high-level constraints that capture interactions both within and across LRAs. Placement constraints are crucial for the performance and resilience of applications. MEDEA follows a two-scheduler design, using an optimisation-based algorithm for high-quality placement of LRAs with constraints, and a traditional scheduler for placing task-based jobs, utilising shorter-running container, with low scheduling latency.

Our experimental evaluation highlights the benefits of MEDEA when placing LRAs. On a 400-node pre-production cluster, MEDEA reduces median runtime of HBase and TensorFlow workloads by up to 32% compared to our implementation of Kubernetes' scheduling algorithm (J-KUBE) and by 2.1× compared to YARN production-ready scheduler, while significantly reducing workload runtime variability too. At the same time, MEDEA improves application unavailability by up to 24% compared to J-KUBE. MEDEA leads to constraint violations of less than 10%, even for complex inter-application

constraints involving 10 LRAs. More importantly, it does not affect the scheduling latency or the performance of task-based jobs.

Chapter 6

Conclusions and future work

Over the past years, several distributed processing systems were developed by large internet companies and the community to facilitate the increasing data processing needs. These big data systems were used to analyse large volumes of data using clusters of commodity machines. Today, similar big data technologies are widely adopted by a variety of sectors, ranging from finance and healthcare to research and entertainment. As these technologies are becoming an integral part of our daily lives by predicting climate change and extreme weather, helping people with disabilities live better, or even assisting epidemiologists detect, analyse and treat deadly diseases, this wide adoption is only expected to grow in the future [95]. This growth will stress the limits of existing big data systems that will have to efficiently utilise compute resources and process even greater volumes of data.

The broad applicability of data-parallel processing systems has enabled scientists across a variety of domains ranging from systems and database research to artificial intelligence, machine learning and statistics to speed up computation and perform experiments at scale. As a side effect, it also led to the evolution of processing systems from single purpose engines to unified data processing platforms, combining large-scale data processing with state-of-the-art machine learning, query optimisation and artificial intelligence as part of a single data processing system [123]. Modern unified processing systems now expose user-friendly programming interfaces for developers to express a variety of computation as part of the same processing engine and then execute in the cluster as part of shared long-running containers.

At the same time, data processing platforms entail large compute clusters to be able to scale and accommodate the increasing processing demands putting more pressure in existing cluster infrastructures. Cluster managers that were traditionally carrying out the on-demand allocation of resources to simple analytics jobs, now have to deal with a plethora of different workloads, including the increasingly popular LRAs. LRA workloads comprise of modern applications and services that utilise long-running containers and have a sustained impact on cluster resources. Workloads with longer- or

shorter-lived containers are now consolidated on shared compute clusters to increase the effective utility extracted from existing resources.

Even though over the past few years big data systems and technologies have evolved to support an increasing variety of workloads, there are still many opportunities to further improve the abstractions, features, performance and efficiency of these systems. In this thesis, we argued that to efficiently support modern cluster workloads and more precisely the growing class of LRAs, we need to bridge the gap between the needs of modern data-centre applications and traditional big data system designs. Big data systems can be general purpose and application-agnostic but at the same time they should be able to provide the right abstractions and logic to effectively accommodate modern application requirements.

LRA workloads require precise control of their container placement to improve performance and resilience. However, existing cluster managers that consolidate a variety of workloads in shared compute clusters ignore the complex placement needs of modern applications. Unified dataflow applications, a particular type of LRAs, combine heterogeneous computation as part of the same application while utilising the same shared long-running containers during execution. Current data processing frameworks for unified applications lack complete support and understanding of their diverse execution requirements.

This thesis proposed a new model for data processing, *unified dataflow graphs with placement constraints*. Our model permits the execution of applications with diverse computation demands by efficiently exposing dataflow requirements as first-class citizens to the system. In addition to execution requirements, our model can capture expressive and high-level container interactions across and within applications running in shared compute clusters. This permits unified dataflow applications or LRAs in general to concisely express the placement requirements of their long-running containers in the cluster.

To evaluate the ideas described in this thesis we implemented NEPTUNE, a data processing framework for unified applications, and MEDEA, a cluster manager with explicit support for powerful placement constraints. Utilising unified dataflows with rich placement constraints, complex heterogeneous applications with diverse execution requirements optimise for high-throughput or low latency according the application needs. At the same time, applications achieve high-quality placement of their containers in shared compute clusters, leading to better application performance and resilience.

Unified dataflows with placement constraints is an effective way of unifying data-parallel processing for the new era of long-running applications scaling to large shared compute clusters.

6.1 Thesis summary

This thesis began by describing the significance of big data analytics and the increasing connection of big data with our everyday lives (Chapter §1). The explosive growth of big data created new ways of storing and managing large volumes of data and led to the development of several novel and scalable data storage technologies. Big internet companies that wanted to improve their services and increase their revenues, used user-generated content stored in their clusters and focused on providing more efficient processing of their invaluable data.

This triggered a new wave of innovation in the big data ecosystem with a plethora of data processing frameworks targeting a variety of workloads and use cases. Even though earlier implementations of those systems were targeting a specific type of computation such as batch processing and streaming, more recent data processing frameworks evolved to support different types of computation as part of the same engine and often exposing unified programming interfaces.

All these data processing frameworks are designed to scale their throughput by following a scale out approach, using the aggregate power of many commodity machines. In order to accommodate their needs, organisations operate large clusters managed by cluster schedulers that are shared across a number of users and applications. As cluster managers are application-agnostic, they enabled the consolidation of a variety of workloads in shared compute clusters to increase the resource efficiency of these infrastructures.

In the background chapter, we described the different workloads that execute in shared compute cluster today (Chapter §2). Using real production data we performed an analysis of the types of cluster workloads within Microsoft and identified particular application scenarios of the growing class of long-running workloads. After explaining the differences and similarities of container and task scheduling, we focused on data processing systems and described the computational model as well as the programming abstractions that they offer and finally showed their evolution. These systems are now expressive enough to capture heterogeneous types of computation within the same execution engine. Despite this evolution, by completely ignoring computation requirements, these systems face well-known limitations in terms of scheduling and execution that we demonstrated with experiments. We also traced the advancements in cluster management systems and demonstrated with experiments the limitations of existing cluster schedulers when dealing with the placement of long-running applications. Finally we categorised them based on their features and system architecture and provided a summary of identified missing of partially supported requirements in cluster scheduling systems.

To overcome these shortcomings, we introduced unified dataflow graphs with placement constraints in Chapter §3. In particular, we propose a new abstraction for efficient data processing in shared data-centres that can explicitly represent dataflow computation requirements in the dataflow graph as well as expressive and high-level placement constraints. The chapter also introduces the abstractions

of node groups and container tags in order to provide powerful and high-level placement constraints for containers in the cluster management level.

The new abstraction of unified dataflows has to respect user-defined computation requirements and also utilise container resources efficiently. Within a single unified application, a part of the computation might require high-throughput processing while another one low-latency responses. To address this challenge, we make dataflow execution requirements explicit. This means that the system is aware of the execution requirements of the application and can make more involved scheduling decisions. For example, it can dynamically prioritise computation based on the execution requirements while fully utilising given resources.

To effectively deploy unified dataflows within large compute clusters and satisfy their performance and resilience requirements, unified dataflows capture expressive and high-level placement constraints using node groups and container tags abstractions. In our model, placement constraints are also first-class citizens. The container allocation system utilises the placement constraints of our model to place long-running containers in the cluster respecting application requirements. At the same time, the container allocation system further optimises placement for global cluster objectives such as balancing cluster load, or minimising the number of machines used in the cluster.

In Chapter §4, we showed how we can implement unified dataflows with user-defined execution requirements as part of a novel computation framework, NEPTUNE. NEPTUNE uses suspendable tasks and novel scheduling policies to prioritise computation dynamically based on captured dataflow requirements and efficiently utilise container resources. In a range of experiments we demonstrated that the NEPTUNE execution framework is flexible and can achieve close to optimal performance in terms of both throughput and latency, while better utilising long-running container resources in the cluster. Our system is built on top of Apache Spark, a widely used data processing framework, and is publicly available as open-source.

Then, in Chapter §5, we showed how we can implement our rich placement constraints as part of MEDEA, a cluster manager that supports the placement of both short- and long-running containers in shared production clusters. MEDEA follows a novel two-scheduler design and a powerful constraints API that can support our expressive constraint model. We evaluated the implementation of MEDEA on top of Apache Hadoop YARN, a popular open-source cluster manager. In a range of experiments on a 400-node pre-production cluster, we showed that our YARN-based implementation of MEDEA can achieve placement of long-running containers that yield significant performance and resilience benefits compared to state-of-the-art systems, while not impacting the scheduling of traditional analytics workloads with short-running containers. Our system is open-sourced and available as part of Apache Hadoop 3.1 release.

6.2 Future work

During the writing of this thesis, we have also identified a number of opportunities for future work.

- **Data-driven scheduling.** In this thesis, we focused mostly on building the abstractions and the scheduling infrastructure that enables the effective placement and the execution of LRAs in modern shared compute clusters. However, automatically inferring application requirements could be an interesting future direction. For example, an alternative approach would be to utilise the historical knowledge on application behaviour. Unlike traditional trial-and-error approaches for finding the best placement constraints, such as the ones used in MEDEA, for each application we would use the application knowledge that is accumulated over time by the system through data collection.

  By using such historical information and applying data mining and machine learning principles, we could significantly improve the practicality of large-scale scheduling, as less user input would be needed, without compromising on the quality of the scheduling decisions [38, 39, 40]. In order to achieve that, we would have to avoid machine learning or data mining techniques with large computational overheads that would be unable scale to the large numbers of servers or applications that comprise modern data-centres. Moreover, for previously-unseen applications, this data-driven approach would still have to use some default values.

- **Hardware heterogeneity.** While both MEDEA and NEPTUNE have focused on workload heterogeneity, large clusters also suffer from hardware heterogeneity that occurs because servers are populated and replaced over time [38, 143]. Heterogeneity in hardware may cause performance variability to applications and is a well known issue in large production data-centres [90, 97, 143]. For example, a particular workload may run faster on an a specific generation processor, CPU architecture or CPU clock speed while another one might require a machine with a GPU or an FPGA.

  The main challenge here is to assign the most suitable resources to each workload, given the various requirements of applications, machine characteristics and cluster constraints. Thus, we would like to further focus on heterogeneity aware scheduling and the integration of both workload and cluster heterogeneity in a more generic model. This model could capture hardware requirements as an extra level of constraints.

- **Dynamic scalability in the cloud.** We proposed an execution model with explicit data processing and container placement requirements that operates in pre-defined cluster environments, however, our model could also benefit systems in more elastic environments. For example, systems that scale their capacity dynamically could also benefit from high quality placements using MEDEA's cardinality constraints as they are generic and provide an upper bound on the number of containers that are placed on the same group of nodes. Even though the constraints

are generic enough to support this scenario, an extended placement algorithm would be needed to natively support scaling (up or down) requests modelled as container migration decisions (see §5.6). Moreover, NEPTUNE's dynamic scheduling decisions during execution could directly benefit such systems, by more efficiently utilising resources and lowering job latency where it matters the most.

As step further, we could also focus on public clouds where computation is sold as utility and thus container allocation and application scalability can bring important cost benefits. We can extend MEDEA, our resource manager that is able control the amount of resources allocated to each applications, to change allocation decisions dynamically. For these dynamic decisions to be correct, input by the applications will be needed because they know precisely when they require more or less resources e.g., when they detect spikes in their load, or when they have too many unused containers.

For example, NEPTUNE could notify MEDEA when executor resources are over-utilised and MEDEA would decide the type (spot or on-demand instance), family (general, memory_optimised, etc.) and the size (small, medium, large) of the cloud instances to allocate. NEPTUNE would use the newly allocated resources to scale out and balance executor load. Thus, there should be a close interaction between the applications and the resource manager. We plan to investigate ways to coordinate them in a scalable and efficient manner.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, 2016.

[2] Sameer Agarwal and Ankit Agarwal. Scaling Apache Spark at Facebook. https://databricks.com/session/scaling-apache-spark-at-facebook. Spark and AI Summit, 2019.

[3] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *PVLDB*, 2013.

[4] George Amvrosiadis, Jun Woo Park, Gregory R Ganger, Garth A Gibson, Elisabeth Baseman, and Nathan DeBardeleben. On the Diversity of Cluster Workloads and its Impact on Research Results. In *USENIX ATC*, 2018.

[5] Apache Hadoop. http://hadoop.apache.org. Accessed: 2019-9-20.

[6] Apache Spark. http://spark.apache.org. Accessed: 2019-9-20.

[7] Apache Apex. http://apex.apache.org. Accessed: 2019-9-20.

[8] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*, 2015.

[9] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *SIGMOD*, 2018.

[10] Apache Aurora. http://aurora.apache.org. Accessed: 2019-9-20.

[11] Peter Bailis, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Bolt-on Causal Consistency. In *SIGMOD*, 2013.

[12] Luiz André Barroso and Urs Hölzle. The Datacenter as a Computer An Introduction to the Design of Warehouse-Scale Machines. *Synthesis Lectures on Computer Architecture*, 2009.

[13] Apache Beam. http://beam.apache.org. Accessed: 2019-9-20.

[14] The Age of Data and Opportunities. https://ibmbigdatahub.com/blog/age-data-and-opportunities. Accessed: 2019-9-20.

[15] Matthias Boehm, Michael Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick Reiss, Prithviraj Sen, Arvind Surve, and Shirish Tatikonda. SystemML: Declarative Machine Learning on Spark. *PVLDB*, 2016.

[16] Peter A Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, 2005.

[17] Peter Boncz, Thomas Neumann, and Orri Erling. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Technology Conference on Performance Evaluation and Benchmarking*, 2013.

[18] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *OSDI*, 2014.

[19] Jesús Camacho-Rodríguez, Ashutosh Chauhan, Alan Gates, Eugene Koifman, Owen O'Malley, Vineet Garg, Zoltan Haindrich, Sergey Shelukhin, Prasanth Jayachandran, Siddharth Seth, et al. Apache Hive: From MapReduce to Enterprise-grade Big Data Warehousing. In *SIGMOD*, 2019.

[20] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2015.

[21] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *ACM SIGPLAN Notices*, 2010.

[22] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 2008.

[23] Wei Chen, Jia Rao, and Xiaobo Zhou. Preemptive, Low Latency Datacenter Scheduling via Lightweight Virtualization. In *USENIX ATC*, 2017.

[24] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Tom Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. Benchmarking Streaming Computation Engines at Yahoo. *Engineering Yahoo! Blog*, 2015.

[25] Andrew Chung, Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Panagiotis Garefalakis, and Gregory R Ganger. Peering through the Dark: An Owl's View of Inter-job Dependencies and Jobs' Impact in Shared Clusters. In *SIGMOD*, 2019.

[26] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, 2010.

[27] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *SOSP*, 2017.

[28] CPLEX Optimizer. http://ibm.com/software/integration/optimization/cplex-optimizer. Accessed: 2019-9-20.

[29] Natacha Crooks, Youer Pu, Nancy Estrada, Trinabh Gupta, Lorenzo Alvisi, and Allen Clement. TARDiS: A Branch-and-Merge Approach To Weak Consistency. In *SIGMOD*, 2016.

[30] Carlo Curino, Djellel E. Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. Reservation-based Scheduling: If You're Late Don't Blame Us! In *SoCC*, 2014.

[31] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M. Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, Roni Burd, Sarvesh Sakalanaga, Chris Douglas, Bill Ramsey, and Raghu Ramakrishnan. Hydra: a Federated Resource Manager for Data-center Scale Analytics. In *NSDI*, 2019.

[32] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[33] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 2008.

[34] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *ACM SIGOPS operating systems review*, 2007.

[35] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid Datacenter Scheduling. In *USENIX ATC*, 2015.

[36] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Job-Aware Scheduling in eagle: Divide and Stick to your Probes. In *SoCC*, 2016.

[37] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Kairos: Preemptive Data Center Scheduling Without Runtime Estimates. In *SoCC*, 2018.

[38] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *SIGPLAN*, 2013.

[39] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *ASPLOS*, 2014.

[40] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters. In *SoCC*, 2015.

[41] Open Sourcing Delta Lake. https://bit.ly/2VY3bgF. Accessed: 2019-9-20.

[42] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks. In *SoCC*, 2014.

[43] Facebook Ginormous Data. https://bit.ly/361elmF. Accessed: 2019-9-20.

[44] Lu Fang, Khanh Nguyen, Guoqing Xu, Brian Demsky, and Shan Lu. Interruptible Tasks: Treating Memory Pressure As Interrupts for Highly Scalable Data-Parallel Programs. In *SOSP*, 2015.

[45] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Making State Explicit for Imperative Big Data Processing. In *USENIX ATC*, 2014.

[46] Raul Castro Fernandez, Panagiotis Garefalakis, and Peter Pietzuch. Java2SDG: Stateful Big Data Processing for the Masses. In *ICDE*, 2016.

[47] Apache Flink. http://flink.apache.org. Accessed: 2019-9-20.

[48] A Year of Blink at Alibaba: Apache Flink in Large Scale Production. https://bit.ly/2TyIFOW. Accessed: 2019-9-20.

[49] Apache Flink's Table API. https://ci.apache.org/projects/flink/flink-docs-stable/dev/table/tableApi.html. Accessed: 2019-9-20.

[50] Panagiotis Garefalakis, Panagiotis Papadopoulos, and Kostas Magoutis. ACaZoo: A Distributed Key-Value Store Based on Replicated LSM-Trees. In *SRDS*, 2014.

[51] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. MEDEA: Scheduling of Long Running Applications in Shared Production Clusters. In *EuroSys*, 2018.

[52] Panagiotis Garefalakis, Konstantinos Karanasos, and Peter Pietzuch. NEPTUNE: Scheduling Suspendable Tasks for Unified Stream/Batch Applications. In *SoCC*, 2019.

[53] Alan Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan Narayanam, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a High-Level Dataflow System on top of Map-Reduce: The Pig Experience. *PVLDB*, 2009.

[54] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *NSDI*, 2011.

[55] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Choosy: Max-Min Fair Sharing for Datacenter Jobs with Constraints. In *EuroSys*, 2013.

[56] Andrey Goder, Alexey Spiridonov, and Yin Wang. Bistro: Scheduling Data-Parallel Jobs Against Live Production Systems. In *USENIX ATC*, 2015.

[57] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *OSDI*, 2016.

[58] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: Packing and Dependency-aware Scheduling for Data-Parallel Clusters. In *OSDI*, 2016.

[59] Hadoop GridMix. http://hadoop.apache.org/docs/r1.2.1/gridmix.html. Accessed: 2019-9-20.

[60] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues Don't Matter When You Can JUMP Them! In *NSDI*, 2015.

[61] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A Thekkath, Yuan Yu, and Li Zhuang. Nectar: Automatic Management of Data and Computation in Datacenters. In *OSDI*, 2010.

[62] Apache HBase. http://hbase.apache.org. Accessed: 2019-9-20.

[63] Fabien Hermenier, Julia Lawall, and Gilles Muller. BtrPlace: A Flexible Consolidation Manager for Highly Available Applications. *IEEE TDSC*, 2013.

[64] Apache Heron. http://heronstreaming.io. Accessed: 2019-9-20.

[65] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, 2011.

[66] Apache Hive. http://hive.apache.org. Accessed: 2019-9-20.

[67] Matthew Hoffman, Francis R Bach, and David M Blei. Online Learning for Latent Dirichlet Allocation. In *NIPS*, 2010.

[68] Yin Huai, Ashutosh Chauhan, Alan Gates, Günther Hagleitner, Eric N. Hanson, Owen O'Malley, Jitendra Pandey, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. Major Technical Advancements in Apache Hive. In *SIGMOD*, 2014.

[69] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX ATC*, 2010.

[70] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys*, 2007.

[71] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *SOSP*, 2009.

[72] Michael Isard. Autopilot: Automatic Data Center Management. *ACM SIGOPS Operating Systems Review*, 2007.

[73] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. Network-Aware Scheduling for Data-Parallel Jobs: Plan When You Can. In *SIGCOMM*, 2015.

[74] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin J. Levandoski, and Gor V. Nishanov. Exploiting Coroutines to Attack the "Killer Nanoseconds". *PVLDB*, 2018.

[75] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Iñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *OSDI*, 2016.

[76] Kafka Streams. http://docs.confluent.io/5.3.0/streams. Accessed: 2019-9-20.

[77] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters. In *USENIX ATC*, 2015.

[78] Konstantinos Karanasos, Arun Suresh, and Chris Douglas. Advancements in YARN Resource Manager. *Encyclopedia of Big Data Technologies*, 2018.

[79] Judy Kay and Piers Lauder. A Fair Share Scheduler. *Communications of the ACM*, 1988.

[80] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR*, 2015.

[81] Lars Kroll, Klas Segeljakt, Paris Carbone, Christian Schulte, and Seif Haridi. Arc: An IR for Batch and Stream Programming. In *DBPL*, 2019.

[82] Kubernetes. http://kubernetes.io. Accessed: 2019-9-20.

[83] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter Heron: Stream Processing at Scale. In *SIGMOD*, 2015.

[84] LXC: LinuX Container. http://linuxcontainers.org. Accessed: 2019-9-20.

[85] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *SOSP*, 2011.

[86] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *ISCA*, 2015.

[87] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. Imbalance in the Cloud: an Analysis on Alibaba Cluster Trace. In *Big Data Conference*, 2017.

[88] Real-time Fraud Detection at Groupon. https://bit.ly/2Chmnv7. Accessed: 2019-9-20.

[89] Marathon. http://mesosphere.github.io/marathon. Accessed: 2019-9-20.

[90] Jason Mars and Lingjia Tang. Whare-Map: Heterogeneity in "Homogeneous" Warehouse-Scale Computers. In *SIGARCH*, 2013.

[91] Nathan Marz and James Warren. *Big Data: Principles and best practices of scalable real-time data systems*. New York; Manning Publications Co., 2015.

[92] Viktor Mayer-Schönberger and Kenneth Cukier. *Big Data: A Revolution That Will Transform How We Live, Work, and Think*. Houghton Mifflin Harcourt, 2013.

[93] Max Meldrum, Klas Segeljakt, Lars Kroll, Paris Carbone, Christian Schulte, and Seif Haridi. Arcon: Continuous and Deep Data Stream Analytics. In *Proceedings of Real-Time Business Intelligence and Analytics*, 2019.

[94] Memcached. http://memcached.org. Accessed: 2019-9-20.

[95] MLIR: Accelerating AI with open-source Infrastructure. https://www.blog.google/technology/ai/mlir-accelerating-ai-open-source-infrastructure/. Accessed: 2019-9-20.

[96] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R De Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.

[97] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-Clouds: Managing Performance Interference Effects for QoS-Aware Clouds. In *EuroSys*, 2010.

[98] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed Stream Computing Platform. In *IEEE International Conference on Data Mining Workshops*, 2010.

[99] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*, 2008.

[100] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, Low Latency Scheduling. In *SOSP*, 2013.

[101] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making Sense of Performance in Data Analytics Frameworks. In *NSDI*, 2015.

[102] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A Kozuch, and Gregory R Ganger. 3Sigma: Distribution-based cluster scheduling for runtime uncertainty. In *EuroSys*, 2018.

[103] Apache Parquet. https://parquet.apache.org. Accessed: 2019-9-20.

[104] Peter Pietzuch, Panagiotis Garefalakis, Alexandros Koliousis, Holger Pirk, and George Theodorakis. Do We Need Distributed Stream Processing? https://lsds.doc.ic.ac.uk/blog/do-we-need-distributed-stream-processing. Accessed: 2019-9-20.

[105] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 2005.

[106] Russell Power and Jinyang Li. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *OSDI*, 2010.

[107] Presto. http://prestodb.io. Accessed: 2019-9-20.

[108] Aleksandar Prokopec and Fengyun Liu. Theory and Practice of Coroutines with Snapshots. In *ECOOP*, 2018.

[109] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. Interleaving with Coroutines: A Practical Approach for Robust Index Joins. *PVLDB*, 2017.

[110] Aurick Qiao, Abutalib Aghayev, Weiren Yu, Haoyang Chen, Qirong Ho, Garth A. Gibson, and Eric P. Xing. Litz: Elastic Framework for High-Performance Distributed Machine Learning. In *USENIX ATC*, 2018.

[111] Jeff Rasley, Konstantinos Karanasos, Srikanth Kandula, Rodrigo Fonseca, Milan Vojnovic, and Sriram Rao. Efficient Queue Management for Cluster Scheduling. In *EuroSys*, 2016.

[112] Reiss, Charles and Tumanov, Alexey and Ganger, Gregory R and Katz, Randy H and Kozuch, Michael A. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *SoCC*, 2012.

[113] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun C. Murthy, and Carlo Curino. Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications. In *SIGMOD*, 2015.

[114] Apache Samza. http://samza.apache.org. Accessed: 2019-9-20.

[115] Neil Savage. Big data versus the big C. *Nature*, 2014.

[116] Robert R Schaller. Moore's Law: Past, Present, and Future. *IEEE Spectrum*, 1997.

[117] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *EuroSys*, 2013.

[118] Apache Slider (incubating). http://slider.incubator.apache.org. Accessed: 2019-9-20.

[119] Continuous Applications: Evolving Streaming in Apache Spark. https://bit.ly/2aJaSOr. Accessed: 2019-9-20.

[120] How to avoid Long running jobs blocking Short running jobs. https://bit.ly/2Ut05hb. Accessed: 2019-9-20.

[121] Mixing Long run periodic update jobs with Streaming. https://bit.ly/2UvRcTS. Accessed: 2019-9-20.

[122] Lightweight pipeline execution. https://bit.ly/2ElCWbx. Accessed: 2019-9-20.

[123] Spark+AI Summit. https://databricks.com/sparkaisummit/north-america. Accessed: 2019-9-20.

[124] Apache Spark Fair Scheduler Pools. https://spark.apache.org/docs/latest/job-scheduling.html#fair-scheduler-pools. Accessed: 2019-9-20.

[125] Spark MLLib. http://spark.apache.org/docs/latest/ml-guide.html. Accessed: 2019-9-20.

[126] Michael Stonebraker. The Case for Shared Nothing. *IEEE Database Eng. Bull.*, 1986.

[127] Apache Storm. http://storm.apache.org. Accessed: 2019-9-20.

[128] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Condor: a Distributed Job Scheduler. In *Beowulf cluster computing with Linux*, 2001.

[129] Apache Tez. https://tez.apache.org. Accessed: 2019-9-20.

[130] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2009.

[131] TPC-H Benchmark. http://www.tpc.org/tpch. Accessed: 2019-9-20.

[132] Alexey Tumanov, James Cipar, Gregory R. Ganger, and Michael A. Kozuch. alsched: Algebraic Scheduling of Mixed Workloads in Heterogeneous Clouds. In *SoCC*, 2012.

[133] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. TetriSched: Global Rescheduling with Adaptive Plan-ahead in Dynamic Heterogeneous Clusters. In *EuroSys*, 2016.

[134] Twitter Record. http://blog.twitter.com/2013/new-tweets-per-second-record-and-how. Accessed: 2019-9-20.

[135] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SoCC*, 2013.

[136] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and Adaptable Stream Processing at Scale. In *SOSP*, 2017.

[137] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *EuroSys*, 2015.

[138] Notify...oh, wait! I have a signal. https://bit.ly/2GMdFGW. Accessed: 2019-9-20.

[139] John Wilkes. More Google cluster data. Google research blog, November 2011. Posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html.

[140] Timothy Wood, Prashant J Shenoy, Arun Venkataramani, and Mazin S Yousif. Black-box and Gray-box Strategies for Virtual Machine Migration. In *NSDI*, 2007.

[141] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: SQL and rich Analytics at Scale. In *SIGMOD*, 2013.

[142] Guoyao Xu, Cheng-Zhong Xu, and Song Jiang. Prophet: Scheduling Executors with Time-Varying Resource Demands on Data-Parallel Computation Frameworks. In *ICAC*, 2016.

[143] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. *SIGARCH*, 2013.

[144] Simplified and first-class support for services in YARN. https://issues.apache.org/jira/browse/YARN-4692. Accessed: 2019-9-20.

[145] https://issues.apache.org/jira/browse/YARN-6592. Accessed: 2019-9-20.

[146] Apache Hadoop Capacity Scheduler. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html. Accessed: 2019-9-20.

[147] Apache Hadoop Fair Scheduler. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html. Accessed: 2019-9-20.

[148] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *OSDI*, 2008.

[149] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *EuroSys*, 2010.

[150] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *HotCloud*, 2010.

[151] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, 2012.

[152] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *SOSP*, 2013.

[153] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI2: CPU Performance Isolation for Shared Compute Clusters. In *EuroSys*, 2013.

[154] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Åke Larson, Ronnie Chaiken, and Darren Shakib. SCOPE: parallel databases meet MapReduce. *VLDB Journal*, 2012.

[155] ZooKeeper. http://zookeeper.apache.org. Accessed: 2019-9-20.