The Grand CRU Challenge

Marcel Blöcher
Department of Computer Science, TU Darmstadt
Darmstadt, Germany
bloecher@dsp.tu-darmstadt.de

Malte Viering
Department of Computer Science, TU Darmstadt
Darmstadt, Germany
viering@dsp.tu-darmstadt.de

Stefan Schmid
Department of Computer Science, Aalborg University
Aalborg, Denmark
Department of Computer Science, TU Berlin
TU Berlin, Germany
schmiste@cs.aau.dk

Patrick Eugster
Department of Computer Science, TU Darmstadt
Darmstadt, Germany
Department of Computer Science, Purdue University
West Lafayette, USA
peugster@dsp.tu-darmstadt.de

ABSTRACT

One of the main objectives of any cluster management system is the maximization of *cluster resource utilization (CRU)*. In this paper, we argue that there is a dilemma underlying the challenge of maximizing CRU, as soon as *network resources* enter the picture. In contrast to local resources which can be handled in a more isolated fashion, global network resources are namely shared, and their allocation is intertwined with that of local resources. For effective resource management, either applications thus have to learn more about the infrastructure, or the resource manager has to understand application semantics – both options violate the separation of applications from the underlying infrastructure strived for by resource managers. This paper makes the case for a resource management system that addresses the dilemma, and presents first ideas.

CCS CONCEPTS

• **Information systems → Computing platforms**; • **Networks** → *Cloud computing*; *In-network processing*; • **Computer systems organization** → *Cloud computing*; • **Software and its engineering** → *Cloud computing*;

KEYWORDS

cloud computing, cluster scheduling, resource manager architecture

ACM Reference format:
Marcel Blöcher, Malte Viering, Stefan Schmid, and Patrick Eugster. 2017. The Grand CRU Challenge. In *Proceedings of HotConNet '17, Los Angeles, CA, USA, August 25, 2017,* 5 pages.
https://doi.org/10.1145/3094405.3094407

1 INTRODUCTION

Highly virtualized compute infrastructures and datacenters promise a high *cluster resource utilization* (CRU) and hence low costs, by multiplexing multiple applications over the shared physical infrastructure. Indeed, today there are several frameworks supporting a broad variety of applications.

However, state-of-the-art resource managers (RMs) like Apache Mesos [6] or Hadoop YARN [10] come with a major drawback: they do not consider *network resources* [8]. The main difference to node-local resources is that network resources are *shared* between hosts (and thus containers scheduled on them), and hence their allocation is intertwined with that of local resources. As a consequence, resources can not easily be dealt with in a piecemeal fashion anymore.

This leads to a dilemma: for taking informed resource scheduling decisions, either cloud applications have to learn more about the underlying infrastructure and resource bottlenecks in order to effectively use a set of independent resources to schedule an entire "graph" of containers, or the RM has to understand more of the application semantics and performance goals. Both of these speak against a clear separation between applications on the one side, and resource management on the other side, as it is currently achieved.

By the same token, even if the RM supports multi-dimensional resource *changes* at runtime (we are only aware of one system providing such functionality, namely Kraken [5]), it is difficult for the application to find the right upgrade plan (an open problem in [5]) to break down the complexity of allocation by having applications outline their requirements incrementally (e.g., by adding containers and corresponding bandwidth requirements to previously scheduled containers one-by-one). For instance, upgrading the computational resources may be of little help if the application is bottlenecked by the network. With different types of resources often depending on each other and having to be jointly optimized [1, 2, 7], the application is essentially forced to resort to a blind trial-and-error.

Figure 1 gives an example for the importance of taking into account all the resources involved in the execution of an application: our streaming case study is based on an Apache Kafka setup, and shows that increasing one type of resource (e.g., computation) may not influence the performance of a streaming application which is currently bottlenecked by another resource (e.g., network bandwidth).

HotConNet '17, August 25, 2017, Los Angeles, CA, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5058-7/17/08…$15.00
https://doi.org/10.1145/3094405.3094407

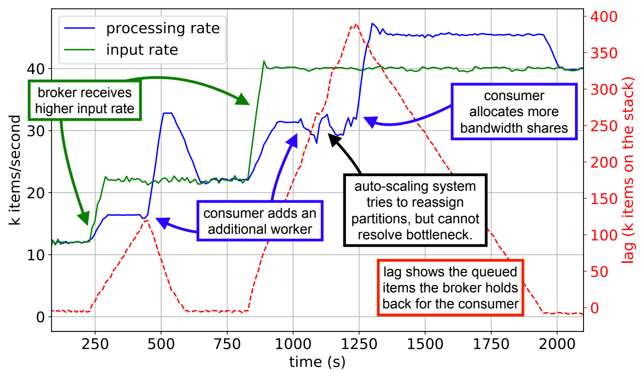


Figure 1: Two-dimensional resource scaling: An Apache Kafka broker system with a single consumer. Item input rate (*green*) shows the number of items the broker receives ready for forwarding. Processing rate (*blue*) gives the processing rate across all worker. Lag (*red*) shows the number of items the broker caches in consequence of slow processing rate. As the input rate changes, the scaling system tries to react with more workers and network shares. Without interactive negotiation of both the RM and the application, a scaling system has to guess which resource may improve the performance.

An intuitive approach to maximize CRU is to simply share more information between the application and the RM. However, this is challenging in practice, for three reasons:

(1) It is desirable to respect the separation of the different roles. Unnecessary dependencies between roles should be avoided in general.
(2) Given that the underlying resource allocation problem is inherently multi-dimensional, the naïve approach to simply expose all possible resource upgrade options becomes combinatorial and expensive.
(3) Different resources can be very different in nature: while computational resources are typically local in nature and can hence be debugged and elastically changed efficiently (e.g., by increasing the number of cores allocated to a container), the provisioning of network resources is an inherently non-local task.

This paper explores the dilemma inherent in today's resource management systems, and initiates the discussion of solutions to the grand CRU challenge, by sharing *slightly* more information between applications and cluster management systems, while respecting the independence of the two roles and ensuring scalability. Indeed, we will argue that existing cluster scheduling systems do not provide efficient information sharing mechanisms, and either abstract from the actual scheduling information problem (e.g., [4]), focus on a single resource only (e.g., [3]), or assume a single point of scheduling (e.g., [7]). Especially in case of bandwidth hungry applications such as scale-out databases and batch processing applications or latency sensitive applications like streaming applications, insufficient information about the global network resources can lead to interference, conservative and inefficient resource reservations, or scalability issues [8].

2 THE GRAND CLUSTER RESOURCE UTILIZATION CHALLENGE

As the resource requirements of cloud applications are usually not limited to computational resources only, but, e.g., also include the network or the memory, an efficient cluster scheduling mechanism needs to account for these resources accordingly. Indeed, resources which are not explicitly modeled or accounted for, can lead to interferences and hence an unpredictable performance.

Indeed, cloud-based applications, including batch processing, streaming, and scale-out databases, generate a significant amount of network traffic and a considerable fraction of their runtime performance is due to network activity.

Ideally, in order to meet performance goals, a cluster scheduler needs the following information:

(1) **Cluster Information**
  (a) *Node-local resources:* Resources such as CPU, memory, ports, etc., and their usage.
  (b) *Network resources:* In particular, network topology including capacity, current usage, and allocation, middleboxes, etc.
(2) **Application Information**
  (a) *Performance goals:* Objectives of each tenant's application components.
  (b) *Resources:* Node-local resource usage and requirements of tenant's application components, performance of each tenant's application components, communication behavior of tenant's application components.

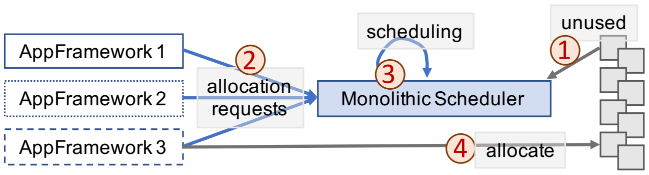
To be aware of this information during runtime is even more important due to the dynamic nature of both resource usage and requirements.

Naturally, the corresponding knowledge is partitioned between the RM and the application framework (AF). The former has the view on the topology from a cluster perspective across all tenants, including each tenant's allocation profile with timing aspects. The latter is aware of each task's performance and how each task interacts with each other (internal and external) component. This introduces a dilemma for cluster resource scheduling: without knowledge of both role's information, scheduling decisions are likely to be suboptimal.

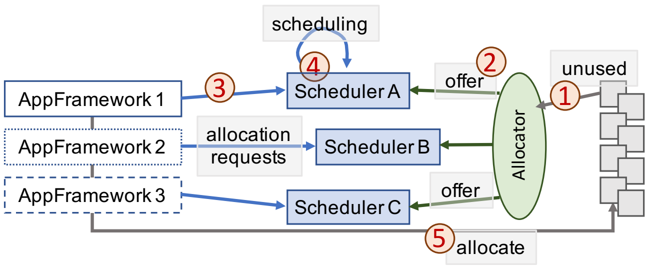
2.1 Design Space

To see the underlying design challenges, we first briefly review existing RM architectures. These coarsely fall into three categories [9]: *monolithic*, *two-level*, and *shared state*. Figure 2 shows all three architectures while focusing on the interaction between AFs, nodes (task executors), and the scheduler(s).

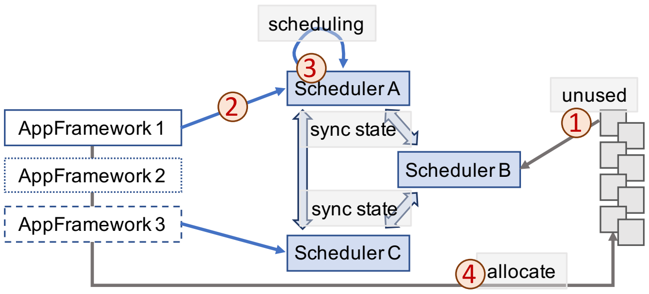
Monolithic scheduling (as used e.g. in YARN and Bazaar [7]) uses a global cluster state: A monolithic RM consists of a central component, the scheduler, which holds all responsibilities. This scheduler acts as the mediator between system components like node managers, which manage each node's processes. While simple and allowing to schedule all components from a logically centralized perspective, this architecture comes with the main drawback that it does not scale (as all requests and scheduling decisions have to pass the central scheduler). A second shortcoming is that all AFs use the same interfaces of the central scheduler, which makes multipath designs for different types of AFs even more complex. The scalability problem becomes worse in the presence of network resources.



(a) Monolithic



(b) Two-level



(c) Shared State

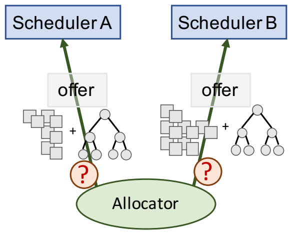
Figure 2: Overview on system components of all three scheduler architectures.

Two-level scheduling (as used e.g. in Mesos) separates the scheduling into two phases: a (central) allocator packs spare resources into a bundle and offers these resources to one of the requesting schedulers. This scheduler may apply AF specific scheduling decisions and decides how to proceed with the received offer. When the scheduler returns its decision, the allocator is able to repack the remaining resources into a new resource bundle. This design overcomes the limitation of the monolithic approach by supporting AF specific schedulers and by distributing the scheduling load. The drawback of the two-level architecture is its pessimistic concurrency model: Resource offering requires to lock resources until the scheduler replies. First, the allocator has to decide when to pack which of the spare resources as a bundle. If such bundles are too small, none of the waiting schedulers may use these resources, or fragmentation may occur [6]. Second, the questions which of the requesting schedulers should receive an offer next should ideally not only be based on, e.g., canonical max-min fairness policies like dominant resource fairness (DRF), but on already allocated resources of an application (and its associated scheduler). This issue becomes more severe for network resources like bandwidth, which has a cluster-spanning effect, as we will see in the following section.

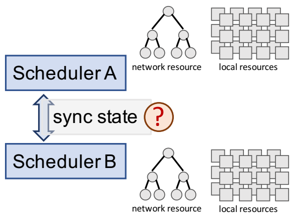
Shared state allocation tries to combine the advantages of the previously mentioned approaches, and to support application specific schedulers without blocking (but conflicts): All schedulers use their own shadow copy of the cluster state (i.e., offering overlapping resources). For coordination purposes of the actual allocations, the schedulers use a transaction like system: A scheduler does its scheduling decisions on the basis of its current state. When allocating resources, it commits its scheduling decision and checks with the global cluster state. Due to the shared nature, the private cluster state of the scheduler might be out of sync or another scheduler may have already claimed the preferred resources in the meantime. Thus, transactional approaches are employed: conflict resolution strategies like incremental commits or all-or-nothing are employed [9]. This distributed approach features similar benefits as the two-level design, it comes at the price of potentially introducing collisions and overheads for synchronizing private cluster state: A typical tradeoff of a shared state system is the state update frequency and ratio of scheduling collisions.

2.2 Limitations

The discussion so far mainly revolved around node-local (compute) resources, and none of the three RM architectures satisfies our requirements on information sharing when accounting for more global network resources. In the following, we omit the monolithic scheduler, based on the drawbacks mentioned so far.



(a) Two-level



(b) Shared state

Figure 3: Available state of two-level and shared state RM if considering node-local and network resources.

Figure 3 shows the available information each scheduler has. The two-level architecture (Figure 3(a)) expands the pessimistic concurrency if network resources are considered: An offer has to guarantee by definition exclusive access when it's received by a scheduler. For considering network resources between compute nodes of an offer, the offer should also contain network shares across them. However, this leads to cluster-spanning locks as each link connects more than two nodes. Furthermore, the allocator has to consider the network topology in addition to all other constraints for determining which scheduler gets which shares.

If the shared state architecture (Figure 3(b)) considers network resources, the shared state covers also these. By increasing the amount of information the shared state holds, collision ratio will rise further. To counteract, one could increase the frequency for updating the cluster state before scheduling. If the frequency is too low (or the information is stale when scheduling is performed), the scheduler may take wrong decisions due to outdated information, leading to more scheduling collisions. At the same time, by increasing the synchronization rate, the amount of state to transfer increases, especially when considering network updates. Schwarzkopf et al. [9] investigate different conflict modes: Commits can be treated as incremental or all-or-nothing if conflicts arise. The conflict detection may use a heuristic that declines if a resource was updated in the meantime, or accepts everything that fits to the current state. For the following evaluation, we use an incremental transaction mode which tries to commit as many allocations as possible.

3 ACCOUNTING FOR THE NETWORK

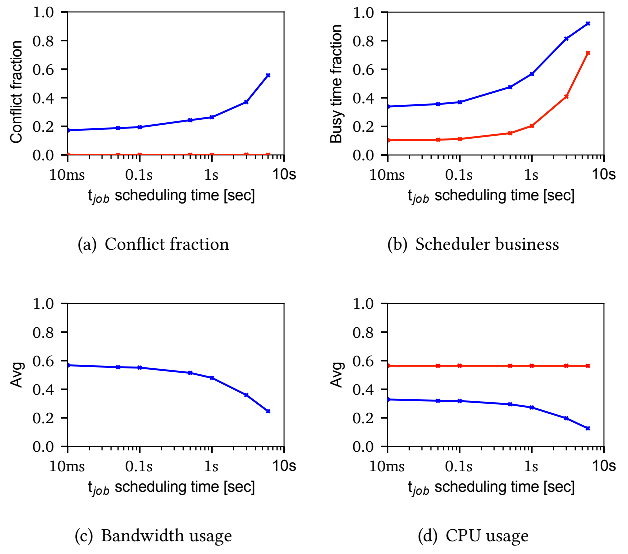
In order to highlight the CRU dilemma when accounting for global network resources, we provide a first analysis and simulation study. We will see that while the two level architecture comes with the drawback of to pessimistic offers, shared state allocation leads to many conflicts. Finally we argue that a combination of the two architectures may offer a path toward tackling the CRU dilemma.

3.1 Inefficiencies due to Conflicts

Shared state allocation has two performance metrics: conflict ratio of scheduling attempts and the amount of information that needs to be shared for synchronizing the state. In our evaluation, we focus on the former and use the Omega simulator [9]. In order to consider network allocations, we modified the simulator as follows: Each job gets an additional resource unit of required bandwidth. This bandwidth specifies for each of its tasks the bandwidth that is required to communicate to all other tasks of this job. We use the virtual cluster (VC) [1] embedding for this purpose. Each task of each job has a bidirectional link of capacity b to a logical switch.

Our setup uses a small fat-tree topology of 6000 compute nodes without bandwidth over-subscription. To the best of our knowledge, there is currently no publicly available trace that provides bandwidth demands of applications. Hence we use a synthetic workload with on average 200 tasks per job and a VC bandwidth demand of each task.

Figure 4 shows the conflict ratio, scheduler business (busy time vs. idle time), and average resource usage. Before considering Omega with network awareness (*blue*), we run the simulation with only node-local resource scheduling (*red*). Each experiment varies the time required for doing the scheduling of a job (t_{job}). The numbers show that our setup of two Omega schedulers is able to handle the scheduling load.
When Omega considers network resources (*blue*), average network



(a) Conflict fraction (b) Scheduler business

(c) Bandwidth usage (d) CPU usage

Figure 4: Shared state scheduler (Omega) simulation with only node-local resources (*red*) and a modification to consider virtual cluster network allocations (*blue*). Performance as a function of scheduling time of a job (t_{job}) in seconds. All experiments use a Fat-Tree like topology with two schedulers running concurrently.

usage (Figure 4(c)) of each server's edge bandwidth is below 60%. Compared to the no-network setup, the conflict ratio increases significantly. Even for the lowest t_{job} the conflict fraction is near 20% and shows higher numbers of up to 60% conflicts when job scheduling time comes to 10s.

This experiment serves only as a first investigation on the issue of network resources and the shared state architecture. In particular, it is optimistic and we expect to see more conflicts when having more than two schedulers running on a cluster.

3.2 Resource Locking Inefficiencies

Two-level architectures entail a resource hoarding issue when considering network for task allocations. As discussed earlier, an offer gives a dedicated bundle of resources to a scheduler. The allocator blocks these resources during scheduling time. When including the network in resource offers, available bandwidth shares on links between affected machines need to be included in the specific offer. Due to the fact that links connect multiple machines, offers with network shares affect not only the machines that are in the specific offer. Additionally, there is an issue when tasks will unlock their resources, as the following example demonstrates.

We will illustrate this using an example. Figure 5 shows a simplified topology with corresponding machines. This example focuses on a single job that runs tasks connected with VC allocations. The VC allocation defines the required bandwidth for an edge as follows [1]: Each edge that divides the tasks of a job (with bandwidth b) into two sets S_1 and S_2, requires reservations of $\min(|S_1|, |S_2|) * b$. Consider a job j, and an offer o that does not contain the full network topology currently used by the running tasks of j. The corresponding scheduler s is processing o to find allocations for new tasks of j. In the meantime (3rd phase) a task of j finishes and frees some resources that affect network links l outside of o. Another scheduler may claim those freed link resources l. If now (4th phase) s commits its placement p based on o, we see that j was modified in the meantime, so we have to recalculate the bandwidth demands of j. Based on the new placed tasks in p, we may need previously freed resources l. However, the link resources l may be locked in the meantime. Consequently, s cannot use the resources of offer o as expected - the allocator has to decline this offer or at least parts of it that cause conflicts.

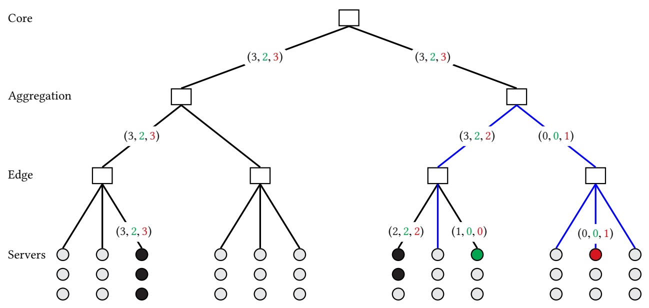


Figure 5: Resource offer conflict: This example shows 4 phases of a job's tasks placement. 1st: 6 tasks distributed on 3 racks (*black* and *green* nodes). Each affected edge shows the required bandwidth shares as first attribute. 2nd: The scheduler receives an offer containing both local and network resources (*blue*) for spawning new tasks. The scheduling decides to start another task on the *red* node. 3rd: In the meantime, the *green* task finishes and frees some resources, as given by the *green* bandwidth shares on the edges. 4th: The scheduler responses to the *blue* offer with its *red* task, which in turn, affects resources (edges) outside of the offer as shown by *red* bandwidth shares on the edges.

This behavior goes against with the idea of pessimistic resource offers. Normally, a single allocation that is based on an offer should not lead to conflicts. A possible modification to resolve this issue could be as follows: Referring to the previous example, 3rd phase, instead of freeing network resources, the allocator blocks the link resources as long as this job is running. Hence, when a task of j finishes, local resources are freed, but not its network resources: a job claims the network resources until all tasks are finished. Only when a job finishes, all related resources (including network) are freed. Then, network resources can be reassigned to a different job. As a consequence, each job does aggressive network resource hoarding. However, this collides with a key property of the two-level architecture: When a task finishes, the allocator receives the tokens for the affected resources. Then, the allocator decides which of the schedulers receive the resources as an offer.

One could think of another solution: Instead of hoarding the network resources as long as a job is running, a job hoards them only as long as some scheduling of this job is running (i.e., during "scheduling-mode"). However, this is impossible because the allocator offers resources to a scheduler and not to a specific job. Hence, from the perspective of an allocator, all running jobs of a scheduler are in "scheduling-mode" as long as the scheduler has some offers, i.e., all the time.

3.3 Towards a Better Tradeoff

None of the three investigated architectures fulfills the requirements to tackle the CRU dilemma. As soon as the scheduler accommodates node-local and network resources, all three architectures unveil a weakness due to their architectural design: The monolithic lacks ability to scale, the two-level becomes too pessimistic, and the shared state sees too many conflicts. Our first analysis reveals that two-level and shared state architecture are two extremes: the former has small disjoint information shares, whereas the latter focuses on conflict resolution if every scheduler has write access on all information. Neither design is suitable if combining node-local and network resource as required for the CRU dilemma.

We hence advocate a system that combines all three architectures by softening the hard constraints of each.

ACKNOWLEDGMENTS

This work has been funded by the German Research Foundation (DFG) as part of project B2 within the Collaborative Research Center (CRC) 1053 – MAKI – Multi-Mechanisms Adaptation for the Future Internet, ERC grant FP7-617805, NSF CSR grant 1618923, as well as Aalborg University's PreLytics project.

REFERENCES

[1] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. 2011. Towards predictable datacenter networks. In *ACM SIGCOMM*.
[2] D. Xie et al. 2012. The only constant is change: incorporating time-varying network reservations in data centers. In *ACM SIGCOMM*.
[3] Andrew D Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. 2012. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 99–112.
[4] Carlo Fuerst, Stefan Schmid, Lalith Suresh, and Paolo Costa. 2015. Kraken: Towards Elastic Performance Guarantees in Multi-tenant Data Centers. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. ACM, 433–434.
[5] Carlo Fuerst, Stefan Schmid, Lalith Suresh, and Paolo Costa. 2016. Kraken: Online and Elastic Resource Reservations for Multi-tenant Datacenters. In *Proc. 35th IEEE Conference on Computer Communications (INFOCOM)*.
[6] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center.. In *NSDI*, Vol. 11. 22–22.
[7] Virajith Jalaparti, Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. 2012. Bridging the tenant-provider gap in cloud services. In *Proc. 3rd ACM Symposium on Cloud Computing (SOCC)*. ACM.
[8] Jeffrey C Mogul and Lucian Popa. 2012. What we talk about when we talk about cloud network performance. *ACM SIGCOMM Computer Communication Review* 42, 5 (2012), 44–48.
[9] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 351–364.
[10] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 5.