A

Advancements in YARN Resource Manager

Konstantinos Karanasos, Arun Suresh, and Chris Douglas
Microsoft, Washington, DC, USA

Synonyms

Cluster scheduling; Job scheduling; Resource management

Definitions

YARN is currently one of the most popular frameworks for scheduling jobs and managing resources in shared clusters. In this entry, we focus on the new features introduced in YARN since its initial version.

Overview

Apache Hadoop (2017), one of the most widely adopted implementations of MapReduce (Dean and Ghemawat 2004), revolutionized the way that companies perform analytics over vast amounts of data. It enables parallel data processing over clusters comprised of thousands of machines while alleviating the user from implementing complex communication patterns and fault tolerance mechanisms.

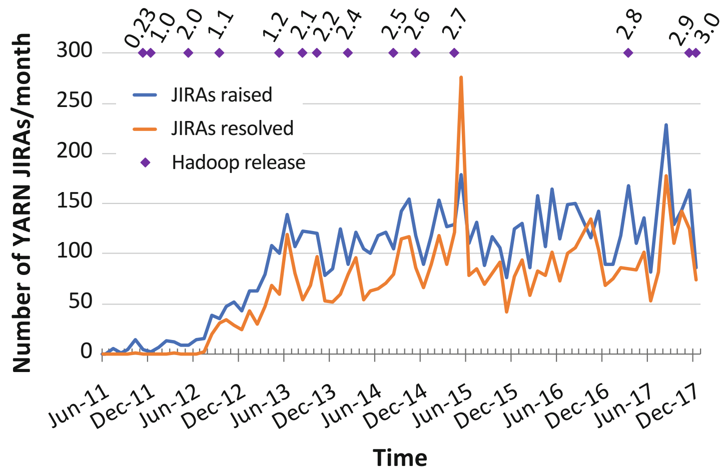
With its rise in popularity, came the realization that Hadoop's resource model for MapReduce, albeit flexible, is not suitable for every application, especially those relying on low-latency or iterative computations. This motivated decoupling the cluster resource management infrastructure from specific programming models and led to the birth of YARN (Vavilapalli et al. 2013). YARN manages cluster resources and exposes a generic interface for applications to request resources. This allows several applications, including MapReduce, to be deployed on a single cluster and share the same resource management layer.

YARN is a community-driven effort that was first introduced in Apache Hadoop in November 2011, as part of the 0.23 release. Since then, the interest of the community has continued unabated. Figure 1 shows that more than 100 tickets, i.e., JIRAs (YARN JIRA 2017), related to YARN are raised every month. A steady portion of these JIRAs are resolved, which shows the continuous community engagement. In the past year alone, 160 individuals have contributed code to YARN.

Moreover, YARN has been widely deployed across hundreds of companies for production purposes, including Yahoo! (Oath), Microsoft, Twitter, LinkedIn, Hortonworks, Cloudera, eBay, and Alibaba.

Since YARN's inception, we observe the following trends in modern clusters:

© Springer International Publishing AG 2018
S. Sakr, A. Zomaya (eds.), *Encyclopedia of Big Data Technologies*,
https://doi.org/10.1007/978-3-319-63962-8_207-1



Advancements in YARN Resource Manager, Fig. 1 Timeline of Apache Hadoop releases (on top) and number of raised and resolved tickets (JIRAs) per month on YARN

Application variety Users' interest has expanded from batch analytics applications (e.g., MapReduce) to include streaming, iterative (e.g., machine learning), and interactive computations.

Large shared clusters Instead of using dedicated clusters for each application, diverse workloads are consolidated on clusters of thousands or even tens of thousands of machines. This consolidation avoids unnecessary data movement, allows for better resource utilization, and enables pipelines with different application classes.

High resource utilization Operating large clusters involves a significant cost of ownership. Hence, cluster operators rely on resource managers to achieve high cluster utilization and improve their return on investment.

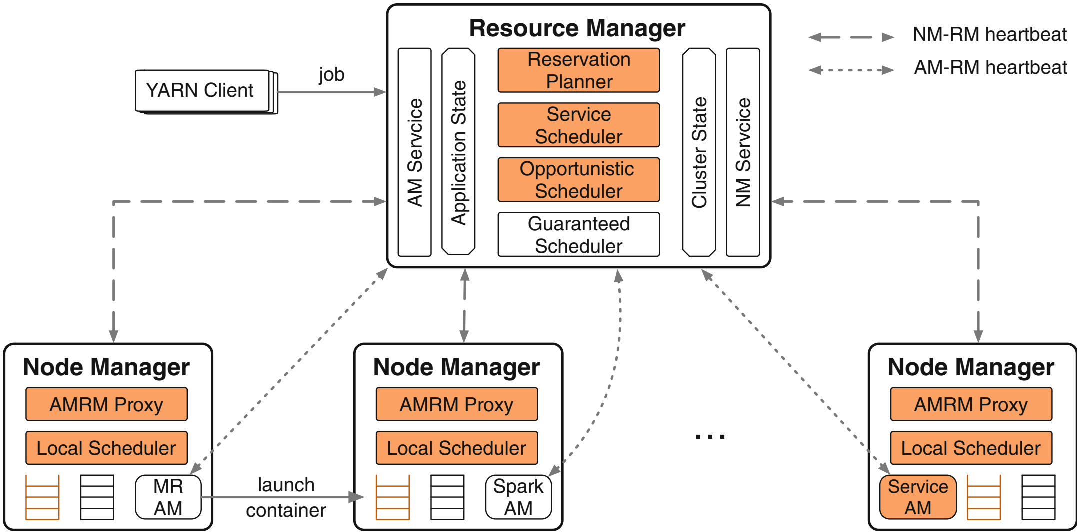
Predictable execution Production jobs typically come with Service Level Objectives (SLOs), such as completion deadlines, which have to be met in order for the output of the jobs to be consumed by downstream services. Execution predictability is often more important than pure application performance when it comes to business-critical jobs.

This diverse set of requirements has introduced new challenges to the resource management layer. To address these new demands, YARN has evolved from a platform for batch analytics workloads to a production-ready, general-purpose resource manager that can support a wide range of applications and user requirements over large shared clusters. In the remainder of this entry, we first give a brief overview of YARN's architecture and dedicate the rest of the paper to the new functionality that was added to YARN these last years.

YARN Architecture

YARN follows a centralized architecture in which a single logical component, the resource manager (RM), allocates resources to jobs submitted to the cluster. The resource requests handled by the RM are intentionally generic, while specific scheduling logic required by each application is encapsulated in the application master (AM) that any framework can implement. This allows YARN to support a wide range of applications using the same RM component. YARN's architecture is depicted in Fig. 2. Below we describe its main components. The new features, which appear in orange, are discussed in the following sections.

Node Manager (NM) The NM is a daemon running at each of the cluster's worker nodes. NMs are responsible for monitoring resource availability at the host node, reporting faults, and managing containers' life cycle (e.g., start, monitor, pause, and kill containers).



Advancements in YARN Resource Manager, Fig. 2 YARN architecture and overview of new features (in orange)

Resource Manager (RM) The RM runs on a dedicated machine, arbitrating resources among various competing applications. Multiple RMs can be used for high availability, with one of them being the master. The NMs periodically inform the RM of their status, which is stored at the *cluster state*. The RM-NM communication is heartbeat-based for scalability. The RM also maintains the resource requests of all applications (*application state*). Given its global view of the cluster and based on application demand, resource availability, scheduling priorities, and sharing policies (e.g., fairness), the *scheduler* performs the matchmaking between application requests and machines and hands leases, called *containers*, to applications. A container is a logical resource bundle (e.g., 2 GB RAM, 1 CPU) bound to a specific node.

YARN includes two scheduler implementations, namely, the Fair and Capacity Schedulers. The former imposes fairness between applications, while the latter dedicates a share of the cluster resources to groups of users.

Jobs are submitted to the RM via the *YARN Client* protocol and go through an admission control phase, during which security credentials are validated and various operational and administrative checks are performed.

Application Master (AM) The AM is the job orchestrator (one AM is instantiated per submitted job), managing all its life cycle aspects, including dynamically increasing and decreasing resource consumption, managing the execution flow (e.g., running reducers against the output of mappers), and handling faults. The AM can run arbitrary user code, written in any programming language. By delegating all these functions to AMs, YARN's architecture achieves significant scalability, programming model flexibility, and improved upgrading/testing.

An AM will typically need to harness resources from multiple nodes to complete a job. To obtain containers, the AM issues resource requests to the RM via heartbeats, using the *AM Service* interface. When the scheduler assigns a resource to the AM, the RM generates a lease for that resource. The AM is then notified and presents the container lease to the NM for launching the container at that node. The NM checks the authenticity of the lease and then initiates the container execution.

In the following sections, we present the main advancements made in YARN, in particular with respect to resource utilization, scalability, support for services, and execution predictability.

Resource Utilization

In the initial versions of YARN, the RM would assign containers to a node only if there were unallocated resources on that node. This **guaranteed** type of allocation ensures that once an AM dispatches a container to a node, there will be sufficient resources for its execution to start immediately.

Despite the predictable access to resources that this design offers, it has the following shortcomings that can lead to suboptimal resource utilization:

Feedback delays The heartbeat-based AM-RM and NM-RM communications can cause idle node resources from the moment a container finishes its execution on a node to the moment an AM gets notified through the RM to launch a new container on that node.

Underutilized resources The RM assigns containers based on the allocated resources at each node, which might be significantly higher than the actually utilized ones (e.g., a 4GB container using only 2 GB of its memory).

In a typical YARN cluster, NM-RM heartbeat intervals are set to 3 s, while AM-RM intervals vary but are typically up to a few seconds. Therefore, feedback delays are more pronounced for workloads with short tasks.

Below we describe the new mechanisms that were introduced in YARN to improve cluster resource utilization. These ideas first appeared in the Mercury and Yaq systems (Karanasos et al. 2015; Rasley et al. 2016) and are part of Apache Hadoop as of version 2.9 (Opportunistic scheduling 2017; Distributed scheduling 2017).

Opportunistic containers Unlike guaranteed containers, opportunistic ones are dispatched to an NM, even if there are no available resources on that node. In such a case, the opportunistic containers will be placed in a newly introduced **NM queue** (see Fig. 2). When resources become available, an opportunistic container will be picked from the queue, and its execution will start immediately, avoiding any feedback delays.

These containers run with lower priority in YARN and will be preempted in case of resource contention for guaranteed containers to start their execution. Hence, opportunistic containers improve cluster resource utilization without impacting the execution of guaranteed containers. Moreover, whereas the original NM passively executes conflict-free commands from the RM, a modern NM uses these two-level priorities as inputs to local scheduling decisions. For instance, low-priority jobs with non-strict execution guarantees or tasks off the critical path of a DAG, are good candidates for opportunistic containers.

The AMs currently determine the execution type for each container, but the system could use automated policies instead. The AM can also request promotion of opportunistic containers to guarantee to protect them from preemption.

Hybrid scheduling Opportunistic containers can be allocated centrally by the RM or in a distributed fashion through a local scheduler that runs at each NM and leases containers on other NMs without contacting the RM. Centralized allocation allows for higher-quality placement decisions and sharing policies. Distributed allocation offers lower allocation latencies, which can be beneficial for short-lived containers. To prevent conflicts, guaranteed containers are always assigned by the RM.

To determine the least-loaded nodes for placing opportunistic containers, the RM periodically gathers information about the running and queued containers at each node and propagates this information to the local schedulers too. To account for occasional load imbalance across nodes, YARN performs dynamic rebalancing of queued containers.

Resource overcommitment Currently, opportunistic containers can be employed to avoid feedback delays. Ongoing development also focuses on overcommitting resources using opportunistic containers (Utilization-based scheduling 2017). In this scenario, opportunistic containers facilitate reclaiming overcommitted resources on demand, without affecting the

performance and predictability of jobs that opt out of overcommitted resources.

Cluster Scalability

A single YARN RM can manage a few thousands of nodes. However, production analytics clusters at big cloud companies are often comprised of tens of thousands of machines, crossing YARN's limits (Burd et al. 2017).

YARN's scalability is constrained by the resource manager, as load increases proportionally to the number of cluster nodes and the application demands (e.g., active containers, resource requests per second). Increasing the heartbeat intervals could improve scalability in terms of number of nodes, but would be detrimental to utilization (Vavilapalli et al. 2013) and would still pose problems as the number of applications increases.

Instead, as of Apache Hadoop 2.9 (YARN Federation 2017), a **federation-based** approach scales a single YARN cluster to tens of thousands of nodes. This approach divides the cluster into smaller units, called **subclusters**, each with its own YARN RM and NMs. The federation system negotiates with subcluster RMs to give applications the experience of a single large cluster, allowing applications to schedule their tasks to any node of the federated cluster.

The state of the federated cluster is coordinated through the **State Store**, a central component that holds information about (1) subcluster liveliness and resource availability via heartbeats sent by each subcluster RM, (2) the YARN subcluster at which each AM is being deployed, and (3) policies used to impose global cluster invariants and perform load rebalancing.

To allow jobs to seamlessly span subclusters, the federated cluster relies on the following components:

Router A federated YARN cluster is equipped with a set of routers, which hide the presence of multiple RMs from applications. Each application gets submitted to a router, which, based on a policy, determines the subcluster for the AM to be executed, gets the subcluster URL from the State Store, and redirects the application submission request to the appropriate subcluster RM.

AMRM Proxy This component runs as a service at each NM of the cluster and acts as a proxy for every AM-RM communication. Instead of directly contacting the RM, applications are forced by the system to access their local AMRM Proxy. By dynamically routing the AM-RM messages, the AMRM Proxy provides the applications with transparent access to multiple YARN RMs. Note that the AMRM Proxy is also used to implement the local scheduler for opportunistic containers and could be used to protect the system against misbehaving AMs.

This federated design is scalable, as the number of nodes each RM is responsible for is bounded. Moreover, through appropriate policies, the majority of applications will be executed within a single subcluster; thus the number of applications that are present at each RM is also bounded. As the coordination between subclusters is minimal, the cluster's size can be scaled almost linearly by adding more subclusters. This architecture can provide tight enforcement of scheduling invariants within a subcluster, while continuous rebalancing across subclusters enforces invariants in the whole cluster.

A similar federated design has been followed to scale the underlying store (HDFS Federation 2017).

Long-Running Services

As already discussed, YARN's target applications were originally batch analytics jobs, such as MapReduce. However, a significant share of today's clusters is dedicated to workloads that include stream processing, iterative computations, data-intensive interactive jobs, and latency-sensitive online applications. Unlike batch jobs, these applications benefit from long-lived containers (from hours to months) to amortize

container initialization costs, reduce scheduling load, or maintain state across computations. Here we use the term *services* for all such applications.

Given their long-running nature, these applications have additional demands, such as support for restart, in-place upgrade, monitoring, and discovery of their components. To avoid using YARN's low-level API for enabling such operations, users have so far resorted to AM libraries such as Slider (Apache Slider 2017). Unfortunately, these external libraries only partially solve the problem, e.g., due to lack of common standards for YARN to optimize resource demands across libraries or version incompatibilities between the libraries and YARN.

To this end, the upcoming Apache Hadoop 3.1 release adds first-class support for long-running services in YARN, allowing for both traditional process-based and Docker-based containers. This service framework allows users to deploy existing services on YARN, simply by providing a JSON file with their service specifications, without having to translate those requirements into low-level resource requests at runtime.

The main component of YARN's service framework is the **container orchestrator**, which facilitates service deployment. It is an AM that, based on the service specification, configures the required requests for the RM and launches the corresponding containers. It deals with various service operations, such as starting components given specified dependencies, monitoring their health and restarting failed ones, scaling up and down component resources, upgrading components, and aggregating logs.

A **RESTful API server** is developed to allow users to manage the life cycle of services on YARN via simple commands, using framework-independent APIs. Moreover, a **DNS server** enables service discovery via standard DNS lookups and greatly simplifies service failovers.

Scheduling services Apart from the aforementioned support for service deployment and management, service owners also demand precise control of container placement to optimize the performance and resilience of their applications. For instance, containers of services are often required to be collocated (affinity) to reduce network costs or separated (anti-affinity) to minimize resource interference and correlated failures. For optimal service performance, even more powerful constraints are useful, such as complex intra- and inter-application constraints that collocate services with one another or put limits in the number of specific containers per node or rack.

When placing containers of services, cluster operators have their own, potentially conflicting, global optimization objectives. Examples include minimizing the violation of placement constraints, the resource fragmentation, the load imbalance, or the number of machines used. Due to their long lifetimes, services can tolerate longer scheduling latencies than batch jobs, but their placement should not impact the scheduling latencies of the latter.

To enable high-quality placement of services in YARN, Apache Hadoop 3.1 introduces support for rich placement constraints (Placement constraints 2017).

Jobs with SLOs

In production analytics clusters, the majority of cluster resources is usually consumed by **production jobs**. These jobs must meet strict Service Level Objectives (SLOs), such as completion deadlines, for their results to be consumed by downstream services. At the same time, a large number of smaller **best-effort jobs** are submitted to the same clusters in an ad hoc manner for exploratory purposes. These jobs lack SLOs, but they are sensitive to completion latencies.

Resource managers typically allocate resources to jobs based on *instantaneous* enforcement of job priorities and sharing invariants. Although simpler to implement and impose, this instantaneous resource provisioning makes it challenging to meet job SLOs without sacrificing low latency for best-effort jobs.

To ensure that important production jobs will have predictable access to resources, YARN was extended with the notion of *reservations*, which provide users with the ability to reserve resources over (and ahead of) time. The ideas around

reservations first appeared in Rayon (Curino et al. 2014) and are part of YARN as of Apache Hadoop 2.6.

Reservations This is a construct that determines the resource needs and temporal requirements of a job and translates the job's completion deadline into an SLO over predictable resource allocations. This is done ahead of the job's execution, aimed at ensuring a predictable and timely execution. To this end, YARN introduced a reservation definition language (RDL) to express a rich class of time-aware resource requirements, including deadlines, malleable and gang parallelism, and inter-job dependencies.

Reservation planning and scheduling RDL provides a uniform and abstract representation of jobs' needs. Reservation requests are received ahead of a job's submission by the **reservation planner**, which performs online admission control. It accepts all jobs that can fit in the cluster agenda over time and rejects those that cannot be satisfied. Once a reservation is accepted by the planner, the scheduler is used to dynamically assign cluster resources to the corresponding job.

Periodic reservations Given that a high percentage of production jobs are recurring (e.g., hourly, daily, or monthly), YARN allows users to define periodic reservations, starting with Apache Hadoop 2.9. A key property of recurring reservations is that once a periodic job is admitted, each of its instantiations will have a predictable resource allocation. This isolates periodic production jobs from the noisiness of sharing.

Toward predictable execution The idea of recurring reservations was first exposed as part of the Morpheus system (Jyothi et al. 2016). Morpheus analyzes inter-job data dependencies and ingress/egress operations to automatically derive SLOs. It uses a resource estimator tool, which is also part of Apache Hadoop as of version 2.9, to estimate jobs' resource requirements based on historic runs. Based on the derived SLOs and resource demands, the system generates recurring reservations and submits them for planning. This guarantees that periodic production jobs will have guaranteed access to resources and thus predictable execution.

Further Improvements

In this section, we discuss some additional improvements made to YARN.

Generic resources As more heterogeneous applications with varying resource demands are deployed to YARN clusters, there is an increasing need for finer control of resource types other than memory and CPU. Examples include disk bandwidth, network I/O, GPUs, and FPGAs.

Adding new resource types in YARN used to be cumbersome, as it required extensive code changes. The upcoming Apache Hadoop 3.1 release (Resource profiles 2017) follows a more flexible resource model, allowing users to add new resources with minimal effort. In fact, users can define their resources in a configuration file, eliminating the need for code changes or recompilation. The Dominant Resource Fairness (Ghodsi et al. 2011) scheduling algorithm at the RM has also been adapted to account for generic resource types, while *resource profiles* can be used for AMs to request containers specifying predefined resource sets. Ongoing work focuses on the isolation of resources such as disk, network, and GPUs.

Node labels Cluster operators can group nodes with similar characteristics, e.g., nodes with public IPs or nodes used for development or testing. Applications can then request containers on nodes with specific labels. This feature is supported by YARN's Capacity Scheduler from Apache Hadoop 2.6 on (Node labels 2017) and allows at most one label to be specified per node, thus creating nonoverlapping node partitions in the cluster. The cluster administrator can specify the portion of a partition that a queue of the scheduler can access, as well as the portion of a queue's capacity that is dedicated to a specific node partition. For instance, `queue A` might be restricted to access no more than 30% of the

nodes with public IPs, and 40% of queue A has to be on dev machines.

Changing queue configuration Several companies use YARN's Capacity Scheduler to share clusters across non-coordinating user groups. A hierarchy of queues isolates jobs from each department of the organization. Due to changes in the resource demands of each department, the queue hierarchy or the cluster condition, operators modify the amount of resources assigned to each organization's queue and to the sub-queues used within that department. However, queue reconfiguration has two main drawbacks: (1) setting and changing configurations is a tedious process that can only be performed by modifying XML files; (2) queue owners cannot perform any modifications to their sub-queues; the cluster admin must do it on their behalf.

To address these shortcomings, Apache Hadoop 2.9 (OrgQueue 2017) allows configurations to be stored in an in-memory database instead of XML files. It adds a RESTful API to programmatically modify the queues. This has the additional benefit that queues can be dynamically reconfigured by automated services, based on the cluster conditions or on organization-specific criteria. Queue ACLs allow queue owners to perform modifications on their part of the queue structure.

Timeline server Information about current and previous jobs submitted in the cluster is key for debugging, capacity planning, and performance tuning. Most importantly, observing historic data enables us to better understand the cluster and jobs' behavior in aggregate to holistically improve the system's operation.

The first incarnation of this effort was the application history server (AHS), which supported only MapReduce jobs. The AHS was superseded by the timeline server (TS), which can deal with generic YARN applications. In its first version, the TS was limited to a single writer and reader that resided at the RM. Its applicability was therefore limited to small clusters.

Apache Hadoop 2.9 includes a major redesign of TS (YARN TS v2 2017), which separates the collection (writes) from the serving (reads) of data, and performs both operations in a distributed manner. This brings several scalability and flexibility improvements.

The new TS collects metrics at various granularities, ranging from *flows* (i.e., sets of YARN applications logically grouped together) to jobs, job attempts, and containers. It also collects cluster-wide data, such as user and queue information, as well as configuration data.

The data collection is performed by collectors that run as services at the RM and at every NM. The AM of each job publishes data to the collector of the host NM. Similarly, each container pushes data to its local NM collector, while the RM publishes data to its dedicated collector. The readers are separate instances that are dedicated to serving queries via a REST API. By default Apache HBase (Apache HBase 2017) is used as the backing storage, which is known to scale to large amounts of data and read/write operations.

Conclusion

YARN was introduced in Apache Hadoop at the end of 2011 as an effort to break the strong ties between Hadoop and MapReduce and to allow generic applications to be deployed over a common resource management fabric. Since then, YARN has evolved to a fully fledged production-ready resource manager, which has been deployed on shared clusters comprising tens of thousands of machines. It handles applications ranging from batch analytics to streaming and machine learning workloads to low-latency services while achieving high resource utilization and supporting SLOs and predictability for production workloads. YARN enjoys a vital community with hundreds of monthly contributions.

A

Cross-References

- ▶ Hadoop
- ▶ Scheduling with Space-Time Soft Constraints in Heterogeneous Cloud Datacenters

Acknowledgements The authors would like to thank Subru Krishnan and Carlo Curino for their feedback while preparing this entry. We would also like to thank the diverse community of developers, operators, and users that have contributed to Apache Hadoop YARN since its inception.

References

Apache Hadoop (2017) Apache Hadoop. http://hadoop.apache.org

Apache HBase (2017) Apache HBase. http://hbase.apache.org

Apache Slider (2017) Apache Slider (incubating). http://slider.incubator.apache.org

Burd R, Sharma H, Sakalanaga S (2017) Lessons learned from scaling YARN to 40 K machines in a multi-tenancy environment. In: DataWorks Summit, San Jose

Curino C, Difallah DE, Douglas C, Krishnan S, Ramakrishnan R, Rao S (2014) Reservation-based scheduling: if you're late don't blame us! In: ACM symposium on cloud computing (SoCC)

Dean J, Ghemawat S (2004) MapReduce: simplified data processing on large clusters. In: USENIX symposium on operating systems design and implementation (OSDI)

Distributed scheduling (2017) Extend YARN to support distributed scheduling. https://issues.apache.org/jira/browse/YARN-2877

Ghodsi A, Zaharia M, Hindman B, Konwinski A, Shenker S, Stoica I (2011) Dominant resource fairness: fair allocation of multiple resource types. In: USENIX symposium on networked systems design and implementation (NSDI)

HDFS Federation (2017) Router-based HDFS federation. https://issues.apache.org/jira/browse/HDFS-10467

Jyothi SA, Curino C, Menache I, Narayanamurthy SM, Tumanov A, Yaniv J, Mavlyutov R, Goiri I, Krishnan S, Kulkarni J, Rao S (2016) Morpheus: towards automated slos for enterprise clusters. In: USENIX symposium on operating systems design and implementation (OSDI)

Karanasos K, Rao S, Curino C, Douglas C, Chaliparambil K, Fumarola GM, Heddaya S, Ramakrishnan R, Sakalanaga S (2015) Mercury: hybrid centralized and distributed scheduling in large shared clusters. In: USENIX annual technical conference (USENIX ATC)

Node Labels (2017) Allow for (admin) labels on nodes and resource-requests. https://issues.apache.org/jira/browse/YARN-796

Opportunistic Scheduling (2017) Scheduling of opportunistic containers through YARN RM. https://issues.apache.org/jira/browse/YARN-5220

OrgQueue (2017) OrgQueue for easy capacityscheduler queue configuration management. https://issues.apache.org/jira/browse/YARN-5734

Placement Constraints (2017) Rich placement constraints in YARN. https://issues.apache.org/jira/browse/YARN-6592

Rasley J, Karanasos K, Kandula S, Fonseca R, Vojnovic M, Rao S (2016) Efficient queue management for cluster scheduling. In: European conference on computer systems (EuroSys)

Resource Profiles (2017) Extend the YARN resource model for easier resource-type management and profiles. https://issues.apache.org/jira/browse/YARN-3926

Utilization-Based Scheduling (2017) Schedule containers based on utilization of currently allocated containers. https://issues.apache.org/jira/browse/YARN-1011

Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S, Saha B, Curino C, O'Malley O, Radia S, Reed B, Baldeschwieler E (2013) Apache Hadoop YARN: yet another resource negotiator. In: ACM symposium on cloud computing (SoCC)

YARN Federation (2017) Enable YARN RM scale out via federation using multiple RMs. https://issues.apache.org/jira/browse/YARN-2915

YARN JIRA (2017) Apache JIRA issue tracker for YARN. https://issues.apache.org/jira/browse/YARN

YARN TS v2 (2017) YARN timeline service v.2. https://issues.apache.org/jira/browse/YARN-5355