Apache Hadoop YARN: Yet Another Resource Negotiator

Vinod Kumar Vavilapalli[h] Arun C Murthy[h] Chris Douglas[m] Sharad Agarwal[i]
Mahadev Konar[h] Robert Evans[y] Thomas Graves[y] Jason Lowe[y] Hitesh Shah[h]
Siddharth Seth[h] Bikas Saha[h] Carlo Curino[m] Owen O'Malley[h] Sanjay Radia[h]
Benjamin Reed[f] Eric Baldeschwieler[h]

[h]: hortonworks.com, [m]: microsoft.com, [i]: inmobi.com, [y]: yahoo-inc.com, [f]: facebook.com

Abstract

The initial design of Apache Hadoop [1] was tightly focused on running massive, MapReduce jobs to process a web crawl. For increasingly diverse companies, Hadoop has become the *data and computational agorá* —the de facto place where data and computational resources are shared and accessed. This broad adoption and ubiquitous usage has stretched the initial design well beyond its intended target, exposing two key shortcomings: 1) tight coupling of a specific programming model with the resource management infrastructure, forcing developers to abuse the MapReduce programming model, and 2) centralized handling of jobs' control flow, which resulted in endless scalability concerns for the scheduler.

In this paper, we summarize the design, development, and current state of deployment of the next generation of Hadoop's compute platform: *YARN*. The new architecture we introduced decouples the programming model from the resource management infrastructure, and delegates many scheduling functions (e.g., task fault-tolerance) to per-application components. We provide experimental evidence demonstrating the improvements we made, confirm improved efficiency by reporting the experience of running YARN on production environments (including 100% of Yahoo! grids), and confirm the flexibility claims by discussing the porting of several programming frameworks onto YARN viz. Dryad, Giraph, Hoya, Hadoop MapReduce, REEF, Spark, Storm, Tez.

SoCC'13, 1–3 Oct. 2013, Santa Clara, California, USA.
ACM 978-1-4503-2428-1. http://dx.doi.org/10.1145/2523616.2523633

1 Introduction

Apache Hadoop began as one of many open-source implementations of MapReduce [12], focused on tackling the unprecedented scale required to index web crawls. Its execution architecture was tuned for this use case, focusing on strong fault tolerance for massive, data-intensive computations. In many large web companies and startups, Hadoop clusters are the common place where operational data are stored and processed.

More importantly, it became *the* place within an organization where engineers and researchers have instantaneous and almost unrestricted access to vast amounts of computational resources and troves of company data. This is both a cause of Hadoop's success and also its biggest curse, as the public of developers extended the MapReduce programming model beyond the capabilities of the cluster management substrate. A common pattern submits "map-only" jobs to spawn arbitrary processes in the cluster. Examples of (ab)uses include forking web servers and gang-scheduled computation of iterative workloads. Developers, in order to leverage the physical resources, often resorted to clever workarounds to sidestep the limits of the MapReduce API.

These limitations and misuses motivated an entire class of papers using Hadoop as a baseline for unrelated environments. While many papers exposed substantial issues with the Hadoop architecture or implementation, some simply denounced (more or less ingeniously) some of the side-effects of these misuses. The limitations of the original Hadoop architecture are, by now, well understood by both the academic and open-source communities.

In this paper, we present a community-driven effort to

move Hadoop past its original incarnation. We present the next generation of Hadoop compute platform known as YARN, which departs from its familiar, monolithic architecture. By separating resource management functions from the programming model, YARN delegates many scheduling-related functions to per-job components. In this new context, MapReduce is just one of the applications running on top of YARN. This separation provides a great deal of flexibility in the choice of programming framework. Examples of alternative programming models that are becoming available on YARN are: Dryad [18], Giraph, Hoya, REEF [10], Spark [32], Storm [4] and Tez [2]. Programming frameworks running on YARN coordinate intra-application communication, execution flow, and dynamic optimizations as they see fit, unlocking dramatic performance improvements. We describe YARN's inception, design, open-source development, and deployment from our perspective as early architects and implementors.

2 History and rationale

In this section, we provide the historical account of how YARN's requirements emerged from practical needs. The reader not interested in the requirements' origin is invited to skim over this section (the requirements are highlighted for convenience), and proceed to Section 3 where we provide a terse description of the YARN's architecture.

Yahoo! adopted Apache Hadoop in 2006 to replace the infrastructure driving its WebMap application [11], the technology that builds a graph of the known web to power its search engine. At the time, the web graph contained more than 100 billion nodes and 1 trillion edges. The previous infrastructure, named "Dreadnaught," [25] had reached the limits of its scalability at 800 machines and a significant shift in its architecture was required to match the pace of the web. Dreadnought already executed distributed applications that resembled MapReduce [12] programs, so by adopting a more scalable MapReduce framework, significant parts of the search pipeline could be migrated easily. This highlights the first requirement that will survive throughout early versions of Hadoop, all the way to YARN—**[R1:] Scalability**.

In addition to extremely large-scale pipelines for Yahoo! Search, scientists optimizing advertising analytics, spam filtering, and content optimization drove many of its early requirements. As the Apache Hadoop community scaled the platform for ever-larger MapReduce jobs, requirements around **[R2:] Multi-tenancy** started to take shape. The engineering priorities and intermediate stages of the compute platform are best understood in this context. YARN's architecture addresses many long-standing requirements, based on experience evolving the MapReduce platform. In the rest of the paper, we will assume general understanding of classic Hadoop architecture, a brief summary of which is provided in Appendix A.

2.1 The era of ad-hoc clusters

Some of Hadoop's earliest users would bring up a cluster on a handful of nodes, load their data into the Hadoop Distributed File System (HDFS)[27], obtain the result they were interested in by writing MapReduce jobs, then tear it down [15]. As Hadoop's fault tolerance improved, persistent HDFS clusters became the norm. At Yahoo!, operators would load "interesting" datasets into a shared cluster, attracting scientists interested in deriving insights from them. While large-scale computation was still a primary driver of development, HDFS also acquired a permission model, quotas, and other features to improve its multi-tenant operation.

To address some of its multi-tenancy issues, Yahoo! developed and deployed *Hadoop on Demand* (HoD), which used Torque[7] and Maui[20] to allocate Hadoop clusters on a shared pool of hardware. Users would submit their job with a description of an appropriately sized compute cluster to Torque, which would enqueue the job until enough nodes become available. Onces nodes become available, Torque would start HoD's 'leader' process on the head node, which would then interact with Torque/Maui to start HoD's slave processes that subsequently spawn a JobTracker and TaskTrackers for that user which then accept a sequence of jobs. When the user released the cluster, the system would automatically collect the user's logs and return the nodes to the shared pool. Because HoD sets up a new cluster for every job, users could run (slightly) older versions of Hadoop while developers could test new features easily. Since Hadoop released a major revision every three months,[1] The flexibility of HoD was critical to maintaining that cadence—we refer to this decoupling of upgrade dependencies as **[R3:] Serviceability**.

While HoD could also deploy HDFS clusters, most users deployed the compute nodes across a shared HDFS instance. As HDFS scaled, more compute clusters could be allocated on top of it, creating a virtuous cycle of increased user density over more datasets, leading to new insights. Since most Hadoop clusters were smaller than the largest HoD jobs at Yahoo!, the JobTracker was rarely a bottleneck.

HoD proved itself as a versatile platform, anticipating some qualities of Mesos[17], which would extend

[1] Between 0.1 and 0.12, Hadoop released a major version every month. It maintained a three month cycle from 0.12 through 0.19

the framework-master model to support dynamic resource allocation among concurrent, diverse programming models. HoD can also be considered a private-cloud precursor of EC2 Elastic MapReduce, and Azure HDInsight offerings—without any of the isolation and security aspects.

2.2 Hadoop on Demand shortcomings

Yahoo! ultimately retired HoD in favor of shared MapReduce clusters due to middling resource utilization. During the map phase, the JobTracker makes every effort to place tasks close to their input data in HDFS, ideally on a node storing a replica of that data. Because Torque allocates nodes without accounting for locality,[2] the subset of nodes granted to a user's JobTracker would likely only contain a handful of relevant replicas. Given the large number of small jobs, most reads were from remote hosts. Efforts to combat these artifacts achieved mixed results; while spreading TaskTrackers across racks made intra-rack reads of shared datasets more likely, the shuffle of records between *map* and *reduce* tasks would necessarily cross racks, and subsequent jobs in the DAG would have fewer opportunities to account for skew in their ancestors. This aspect of **[R4:] Locality awareness** is a key requirement for YARN.

High-level frameworks like Pig[24] and Hive[30] often compose a workflow of MapReduce jobs in a DAG, each filtering, aggregating, and projecting data at every stage in the computation. Because clusters were not resized between jobs when using HoD, much of the capacity in a cluster lay fallow while subsequent, slimmer stages completed. In an extreme but a very common scenario, a single reduce task running on one node could prevent a cluster from being reclaimed. Some jobs held hundreds of nodes idle in this state.

Finally, job latency was dominated by the time spent allocating the cluster. Users could rely on few heuristics when estimating how many nodes their jobs required, and would often ask for whatever multiple of 10 matched their intuition. Cluster allocation latency was so high, users would often share long-awaited clusters with colleagues, holding on to nodes for longer than anticipated, raising latencies still further. While users were fond of many features in HoD, the economics of cluster utilization forced Yahoo! to pack its users into shared clusters. **[R5:] High Cluster Utilization** is a top priority for YARN.

[2]Efforts to modify torque to be "locality-aware" mitigated this effect somewhat, but the proportion of remote reads was still much higher than what a shared cluster could achieve.

2.3 Shared clusters

Ultimately, HoD had too little information to make intelligent decisions about its allocations, its resource granularity was too coarse, and its API forced users to provide misleading constraints to the resource layer.

However, moving to shared clusters was non-trivial. While HDFS had scaled gradually over years, the JobTracker had been insulated from those forces by HoD. When that guard was removed, MapReduce clusters suddenly became significantly larger, job throughput increased dramatically, and many of the features innocently added to the JobTracker became sources of critical bugs. Still worse, instead of losing a single workflow, a JobTracker failure caused an outage that would lose *all* the running jobs in a cluster and require users to manually recover their workflows.

Downtime created a backlog in processing pipelines that, when restarted, would put significant strain on the JobTracker. Restarts often involved manually killing users' jobs until the cluster recovered. Due to the complex state stored for each job, an implementation preserving jobs across restarts was never completely debugged.

Operating a large, multi-tenant Hadoop cluster is hard. While fault tolerance is a core design principle, the surface exposed to user applications is vast. Given various availability issues exposed by the single point of failure, it is critical to continuously monitor workloads in the cluster for dysfunctional jobs. More subtly, because the JobTracker needs to allocate tracking structures for every job it initializes, its admission control logic includes safeguards to protect its own availability; it may delay allocating fallow cluster resources to jobs because the overhead of tracking them could overwhelm the JobTracker process. All these concerns may be grouped under the need for **[R6:] Reliability/Availability**.

As Hadoop managed more tenants, diverse use cases, and raw data, its requirements for isolation became more stringent, but the authorization model lacked strong, scalable authentication—a critical feature for multi-tenant clusters. This was added and backported to multiple versions. **[R7:] Secure and auditable operation** must be preserved in YARN. Developers gradually hardened the system to accommodate diverse needs for resources, which were at odds with the slot-oriented view of resources.

While MapReduce supports a wide range of use cases, it is not the ideal model for all large-scale computation. For example, many machine learning programs require multiple iterations over a dataset to converge to a result. If one composes this flow as a sequence of MapReduce jobs, the scheduling overhead will significantly delay the result [32]. Similarly, many graph al-

gorithms are better expressed using a bulk-synchronous parallel model (BSP) using message passing to communicate between vertices, rather than the heavy, all-to-all communication barrier in a fault-tolerant, large-scale MapReduce job [22]. This mismatch became an impediment to users' productivity, but the MapReduce-centric resource model in Hadoop admitted no competing application model. Hadoop's wide deployment inside Yahoo! and the gravity of its data pipelines made these tensions irreconcilable. Undeterred, users would write "MapReduce" programs that would spawn alternative frameworks. To the scheduler they appeared as map-only jobs with radically different resource curves, thwarting the assumptions built into to the platform and causing poor utilization, potential deadlocks, and instability. YARN must declare a truce with its users, and provide explicit **[R8:] Support for Programming Model Diversity**.

Beyond their mismatch with emerging framework requirements, typed slots also harm utilization. While the separation between *map* and *reduce* capacity prevents deadlocks, it can also bottleneck resources. In Hadoop, the overlap between the two stages is configured by the user for each submitted job; starting reduce tasks later increases cluster throughput, while starting them early in a job's execution reduces its latency.[3] The number of map and reduce slots are fixed by the cluster operator, so fallow map capacity can't be used to spawn reduce tasks and vice versa.[4] Because the two task types complete at different rates, no configuration will be perfectly balanced; when either slot type becomes saturated, the JobTracker may be required to apply backpressure to job initialization, creating a classic pipeline bubble. Fungible resources complicate scheduling, but they also empower the allocator to pack the cluster more tightly. This highlights the need for a **[R9:] Flexible Resource Model**.

While the move to shared clusters improved utilization and locality compared to HoD, it also brought concerns for serviceability and availability into sharp relief. Deploying a new version of Apache Hadoop in a shared cluster was a carefully choreographed, and a regrettably common event. To fix a bug in the MapReduce implementation, operators would necessarily schedule downtime, shut down the cluster, deploy the new bits, validate the upgrade, then admit new jobs. By conflating the *platform* responsible for arbitrating resource usage with the *framework* expressing that program, one is forced to evolve them simultaneously; when operators improve the allocation efficiency of the platform,

[3]This oversimplifies significantly, particularly in clusters of unreliable nodes, but it is generally true.

[4]Some users even optimized their jobs to favor either map or reduce tasks based on shifting demand in the cluster [28].

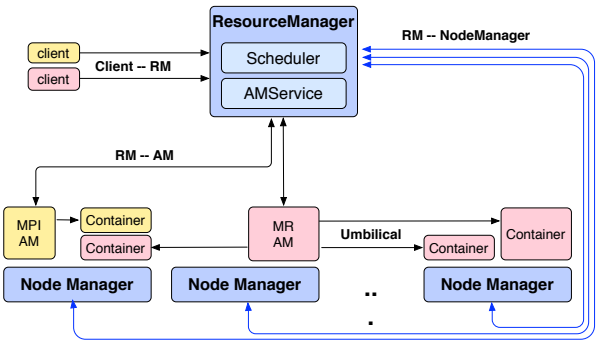


Figure 1: YARN Architecture (in blue the system components, and in yellow and pink two applications running.)

users must necessarily incorporate framework changes. Thus, upgrading a cluster requires users to halt, validate, and restore their pipelines for orthogonal changes. While updates typically required no more than recompilation, users' assumptions about internal framework details—or developers' assumptions about users' programs—occasionally created blocking incompatibilities on pipelines running on the grid.

Building on lessons learned by evolving Apache Hadoop MapReduce, YARN was designed to address requirements (R1-R9). However, the massive install base of MapReduce applications, the ecosystem of related projects, well-worn deployment practice, and a tight schedule would not tolerate a radical redesign. To avoid the trap of a "second system syndrome" [6], the new architecture reused as much code from the existing framework as possible, behaved in familiar patterns, and left many speculative features on the drawing board. This lead to the final requirement for the YARN redesign: **[R10:] Backward compatibility**.

In the remainder of this paper, we provide a description of YARN's architecture (Sec. 3), we report about real-world adoption of YARN (Sec. 4), provide experimental evidence validating some of the key architectural choices (Sec. 5) and conclude by comparing YARN with some related work (Sec. 6).

3 Architecture

To address the requirements we discussed in Section 2, YARN lifts some functions into a *platform* layer responsible for resource management, leaving coordination of logical execution plans to a host of *framework* implementations. Specifically, a per-cluster *ResourceManager* (RM) tracks resource usage and node liveness, enforces allocation invariants, and arbitrates contention among tenants. By separating these duties in the JobTracker's charter, the central allocator can use an abstract description of tenants' requirements, but remain ignorant of the

semantics of each allocation. That responsibility is delegated to an *ApplicationMaster* (AM), which coordinates the logical plan of a single job by requesting resources from the RM, generating a physical plan from the resources it receives, and coordinating the execution of that plan around faults.

3.1 Overview

The RM runs as a daemon on a dedicated machine, and acts as the central authority arbitrating resources among various competing applications in the cluster. Given this central and global view of the cluster resources, it can enforce rich, familiar properties such as fairness [*R10*], capacity [*R10*], and locality [*R4*] across tenants. Depending on the application demand, scheduling priorities, and resource availability, the RM dynamically allocates leases– called *containers*– to applications to run on particular nodes.[5] The container is a logical bundle of resources (e.g., $\langle 2\text{GB RAM}, 1\text{ CPU}\rangle$) bound to a particular node [*R4,R9*]. In order to enforce and track such assignments, the RM interacts with a special system daemon running on each node called the *NodeManager* (NM). Communications between RM and NMs are heartbeat-based for scalability. NMs are responsible for monitoring resource availability, reporting faults, and container lifecycle management (e.g., starting, killing). The RM assembles its global view from these snapshots of NM state.

Jobs are submitted to the RM via a public submission protocol and go through an admission control phase during which security credentials are validated and various operational and administrative checks are performed [*R7*]. Accepted jobs are passed to the scheduler to be run. Once the scheduler has enough resources, the application is moved from accepted to running state. Aside from internal bookkeeping, this involves allocating a container for the AM and spawning it on a node in the cluster. A record of accepted applications is written to persistent storage and recovered in case of RM restart or failure.

The ApplicationMaster is the "head" of a job, managing all lifecycle aspects including dynamically increasing and decreasing resources consumption, managing the flow of execution (e.g., running reducers against the output of maps), handling faults and computation skew, and performing other local optimizations. In fact, the AM can run arbitrary user code, and can be written in any programming language since all communication with the RM and NM is encoded using extensible communication protocols[6]—as an example consider the Dryad port we discuss in Section 4.2. YARN makes few assumptions about the AM, although in practice we expect most jobs will use a higher level programming framework (e.g., MapReduce, Dryad, Tez, REEF, etc.). By delegating all these functions to AMs, YARN's architecture gains a great deal of scalability [*R1*], programming model flexibility [*R8*], and improved upgrading/testing [*R3*] (since multiple versions of the same framework can coexist).

Typically, an AM will need to harness the resources (cpus, RAM, disks etc.) available on multiple nodes to complete a job. To obtain containers, AM issues resource requests to the RM. The form of these requests includes specification of locality preferences and properties of the containers. The RM will attempt to satisfy the resource requests coming from each application according to availability and scheduling policies. When a resource is allocated on behalf of an AM, the RM generates a lease for the resource, which is pulled by a subsequent AM heartbeat. A token-based security mechanism guarantees its authenticity when the AM presents the container lease to the NM [*R4*]. Once the ApplicationMaster discovers that a container is available for its use, it encodes an application-specific launch request with the lease. In MapReduce, the code running in the container is either a map task or a reduce task.[7] If needed, running containers may communicate directly with the AM through an application-specific protocol to report status and liveness and receive framework-specific commands– YARN neither facilitates nor enforces this communication. Overall, a YARN deployment provides a basic, yet robust infrastructure for lifecycle management and monitoring of containers, while application-specific semantics are managed by each framework [*R3,R8*].

This concludes the architectural overview of YARN. In the remainder of this section, we provide details for each of the main components.

3.2 Resource Manager (RM)

The ResourceManager exposes two public interfaces towards: 1) clients submitting applications, and 2) ApplicationMaster(s) dynamically negotiating access to resources, and one internal interface towards NodeManagers for cluster monitoring and resource access management. In the following, we focus on the public protocol between RM and AM, as this best represents the important frontier between the YARN platform and the various applications/frameworks running on it.

The RM matches a global model of the cluster state against the digest of resource requirements reported by

[5] We will refer to "containers" as the logical lease on resources and the actual process spawned on the node interchangeably.

[6] See: `https://code.google.com/p/protobuf/`

[7] In fact, the same code is spawned by TaskTrackers in Hadoop 1.x [*R10*]. Once started, the process obtains its payload from the AM across the network, rather than from the local daemon.

running applications. This makes it possible to tightly enforce global scheduling properties (different schedulers in YARN focus on different global properties, such as capacity or fairness), but it requires the scheduler to obtain an accurate understanding of applications' resource requirements. Communication messages and scheduler state must be compact and efficient for the RM to scale against application demand and the size of the cluster [*R1*]. The way resource requests are captured strikes a balance between accuracy in capturing resource needs and compactness. Fortunately, the scheduler only handles an overall resource profile for each application, ignoring local optimizations and internal application flow. YARN completely departs from the static partitioning of resources for maps and reduces; it treats the cluster resources as a (discretized) continuum [*R9*]—as we will see in Section 4.1, this delivered significant improvements in cluster utilization.

ApplicationMasters codify their need for resources in terms of one or more ResourceRequests, each of which tracks:

1. number of containers (e.g., 200 containers),
2. resources[8] per container ⟨2GB RAM, 1 CPU⟩,
3. locality preferences, and
4. priority of requests within the application

ResourceRequests are designed to allow users to capture the full detail of their needs and/or a roll-up version of it (e.g., one can specify node-level, rack-level, and global locality preferences [*R4*]). This allows more uniform requests to be represented compactly as aggregates. Furthermore, this design would allow us to impose size limits on ResourceRequests, thus leveraging the roll-up nature of ResourceRequests as a lossy compression of the application preferences. This makes communication and storage of such requests efficient for the scheduler, and it allows applications to express their needs clearly [*R1,R5,R9*]. The roll-up nature of these requests also guides the scheduler in case perfect locality cannot be achieved, towards good alternatives (e.g., rack-local allocation, if the desired node is busy).

This resource model serves current applications well in homogeneous environments, but we expect it to evolve over time as the ecosystem matures and new requirements emerge. Recent and developing extensions include: explicit tracking of gang-scheduling needs, and soft/hard constraints to express arbitrary co-location or disjoint placement.[9]

The scheduler tracks, updates, and satisfies these requests with available resources, as advertised on NM heartbeats. In response to AM requests, the RM generates containers together with tokens that grant access to resources. The RM forwards the exit status of finished containers, as reported by the NMs, to the responsible AMs. AMs are also notified when a new NM joins the cluster so that they can start requesting resources on the new nodes.

A recent extension of the protocol allows the RM to symmetrically request resources back from an application. This typically happens when cluster resources become scarce and the scheduler decides to revoke some of the resources that were given to an application to maintain scheduling invariants. We use structures similar to ResourceRequests to capture the locality preferences (which could be strict or negotiable). AMs have some flexibility when fulfilling such 'preemption' requests, e.g., by yielding containers that are less crucial for its work (for e.g. tasks that made only little progress till now), by checkpointing the state of a task, or by migrating the computation to other running containers. Overall, this allows applications to preserve work, in contrast to platforms that forcefully kill containers to satisfy resource constraints. If the application is non-collaborative, the RM can, after waiting for a certain amount of time, obtain the needed resources by instructing the NMs to forcibly terminate containers.

Given the prenominate requirements from section 2, it is important to point out what the ResourceManager is *not* responsible for. As discussed, it is not responsible for coordinating application execution or task fault-tolerance, but neither is is charged with 1) providing status or metrics for running applications (now part of the ApplicationMaster), nor 2) serving framework specific reports of completed jobs (now delegated to a per-framework daemon).[10] This is consistent with the view that the ResourceManager should only handle live resource scheduling, and helps central components in YARN scale beyond the Hadoop 1.0 JobTracker.

3.3 Application Master (AM)

An application may be a static set of processes, a logical description of work, or even a long-running service. The ApplicationMaster is the process that coordinates the application's execution in the cluster, but it itself is run in the cluster just like any other container. A component of the RM negotiates for the container to spawn this bootstrap process.

The AM periodically heartbeats to the RM to affirm its liveness and to update the record of its demand. After building a model of its requirements, the AM encodes its preferences and constraints in a heartbeat message to

[8] The resource vector is designed to be extensible.

[9] While the RM uses locality as a weight for placing containers, network bandwidth is not explicitly modeled and reserved as in Oktopus [5] or Orchestra [9].

[10] YARN does provide generic information about completed apps, containers etc. via a common daemon called Application History Server.

the RM. In response to subsequent heartbeats, the AM will receive a *container lease* on bundles of resources bound to a particular node in the cluster. Based on the containers it receives from the RM, the AM may update its execution plan to accommodate perceived abundance or scarcity. In contrast to some resource models, the allocations to an application are *late binding*: the process spawned is not bound to the request, but to the lease. The conditions that caused the AM to issue the request may not remain true when it receives its resources, but the semantics of the container are fungible and framework-specific [*R3,R8,R10*]. The AM will also update its resource asks to the RM as the containers it receives affect both its present and future requirements.

By way of illustration, the MapReduce AM optimizes for locality among map tasks with identical resource requirements. When running on HDFS, each block of input data is replicated on k machines. When the AM receives a container, it matches it against the set of pending map tasks, selecting a task with input data close to the container. If the AM decides to run a map task m_i in the container, then the hosts storing replicas of m_i's input data are less desirable. The AM will update its request to diminish the weight on the other $k-1$ hosts. This relationship between hosts remains opaque to the RM; similarly, if m_i fails, the AM is responsible for updating its demand to compensate. In the case of MapReduce, note that some services offered by the Hadoop JobTracker—such as job progress over RPC, a web interface to status, access to MapReduce-specific, historical data—are no longer part of the YARN architecture. These services are either provided by ApplicationMasters or by *framework* daemons.

Since the RM does not interpret the container status, the AM determines the semantics of the success or failure of the container exit status reported by NMs through the RM. Since the AM is itself a container running in a cluster of unreliable hardware, it should be resilient to failure. YARN provides some support for recovery, but because fault tolerance and application semantics are so closely intertwined, much of the burden falls on the AM. We discuss the model for fault tolerance in section 3.6.

3.4 Node Manager (NM)

The NodeManager is the "worker" daemon in YARN. It authenticates container leases, manages containers' dependencies, monitors their execution, and provides a set of services to containers. Operators configure it to report memory, CPU, and other resources available at this node and allocated for YARN. After registering with the RM, the NM heartbeats its status and receives instructions.

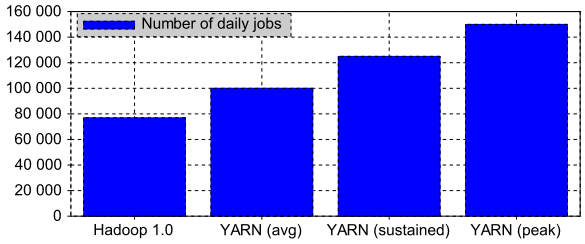
All containers in YARN– including AMs– are described by a *container launch context* (CLC). This record includes a map of environment variables, dependencies stored in remotely accessible storage, security tokens, payloads for NM services, and the command necessary to create the process. After validating the authenticity of the lease [*R7*], the NM configures the environment for the container, including initializing its monitoring subsystem with the resource constraints specified in the lease. To launch the container, the NM copies all the necessary dependencies– data files, executables, tarballs– to local storage. If required, the CLC also includes credentials to authenticate the download. Dependencies may be shared between containers in an application, between containers launched by the same tenant, and even between tenants, as specified in the CLC. The NM eventually garbage collects dependencies not in use by running containers.

The NM will also kill containers as directed by the RM or the AM. Containers may be killed when the RM reports its owning application as completed, when the scheduler decides to evict it for another tenant, or when the NM detects that the container exceeded the limits of its lease [*R2,R3,R7*]. AMs may request containers to be killed when the corresponding work isn't needed any more. Whenever a container exits, the NM will clean up its working directory in local storage. When an application completes, all resources owned by its containers are discarded on all nodes, including any of its processes still running in the cluster.

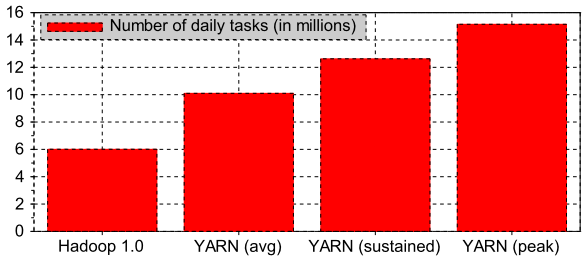
NM also periodically monitors the health of the physical node. It monitors any issues with the local disks, and runs an admin configured script frequently that in turn can point to any hardware/software issues. When such an issue is discovered, NM changes its state to be unhealthy and reports RM about the same which then makes a scheduler specific decision of killing the containers and/or stopping future allocations on this node till the health issue is addressed.

In addition to the above, a NM offers local services to containers running on that node. For example, the current implementation includes a *log aggregation* service that will upload data written by the application to `stdout` and `stderr` to HDFS once the application completes.

Finally, an administrator may configure the NM with a set of pluggable, *auxiliary services*. While a container's local storage will be cleaned up after it exits, it is allowed to promote some output to be preserved until the application exits. In this way, a process may produce data that persist beyond the life of the container, to be managed by the node. One important use case for these services are Hadoop MapReduce applications, for which intermediate data are transferred between map and reduce tasks using an auxiliary service. As mentioned earlier, the CLC allows AMs to address a payload to auxil-



(a) Daily jobs



(b) Daily tasks

Figure 2: YARN vs Hadoop 1.0 running on a 2500 nodes production grid at Yahoo!.

iary services; MapReduce applications use this channel to pass tokens that authenticate reduce tasks to the shuffle service.

3.5 YARN framework/application writers

From the preceding description of the core architecture, we extract the responsibilities of a YARN application author:

1. Submitting the application by passing a CLC for the ApplicationMaster to the RM.
2. When RM starts the AM, it should register with the RM and periodically advertise its liveness and requirements over the heartbeat protocol
3. Once the RM allocates a container, AM can construct a CLC to launch the container on the corresponding NM. It may also monitor the status of the running container and stop it when the resource should be reclaimed. Monitoring the progress of work done inside the container is strictly the AM's responsibility.
4. Once the AM is done with its work, it should unregister from the RM and exit cleanly.
5. Optionally, framework authors may add control flow between their own clients to report job status and expose a control plane.

Even a simple AM can be fairly complex; a distributed shell example with a handful of features is over 450 lines of Java. Frameworks to ease development of YARN applications exist. We explore some of these in section 4.2. Client libraries - YarnClient, NMClient, AMRMClient - ship with YARN and expose higher level APIs to avoid coding against low level protocols. An AM hardened against faults– including its own– is non-trivial. If the application exposes a service or wires a communication graph, it is also responsible for all aspects of its secure operation; YARN only secures its deployment.

3.6 Fault tolerance and availability

From its inception, Hadoop was designed to run on commodity hardware. By building fault tolerance into every layer of its stack, it hides the complexity of detection and recovery from hardware faults from users. YARN inherits that philosophy, though responsibility is now distributed between the ResourceManager and ApplicationMasters running in the cluster.

At the time of this writing, the RM remains a single point of failure in YARN's architecture. The RM recovers from its own failures by restoring its state from a persistent store on initialization. Once the recovery process is complete, it kills all the containers running in the cluster, including live ApplicationMasters. It then launches new instances of each AM. If the framework supports recovery—and most will, for routine fault tolerance—the platform will automatically restore users' pipelines [*R6*]. Work is in progress to add sufficient protocol support for AMs to survive RM restart. With this, AMs can continue to progress with existing containers while the RM is down, and can resync with the RM when it comes back up. Efforts are also underway to address high availability of a YARN cluster by having passive/active failover of RM to a standby node.

When a NM fails, the RM detects it by timing out its heartbeat response, marks all the containers running on that node as killed, and reports the failure to all running AMs. If the fault is transient, the NM will re-synchronize with the RM, clean up its local state, and continue. In both cases, AMs are responsible for reacting to node failures, potentially redoing work done by any containers running on that node during the fault.

Since the AM runs in the cluster, its failure does not affect the availability of the cluster [*R6,R8*], but the probability of an application hiccup due to AM failure is higher than in Hadoop 1.x. The RM may restart the AM if it fails, though the platform offers no support to restore the AMs state. A restarted AM synchronizing with its own running containers is also not a concern of the platform. For example, the Hadoop MapReduce AM will recover its completed tasks, but as of this writing, running tasks– and tasks that completed during AM recovery– will be killed and re-run.

Finally, the failure handling of the containers themselves is completely left to the frameworks. The RM collects all container exit events from the NMs and propagates those to the corresponding AMs in a heartbeat response. The MapReduce ApplicationMaster already lis-

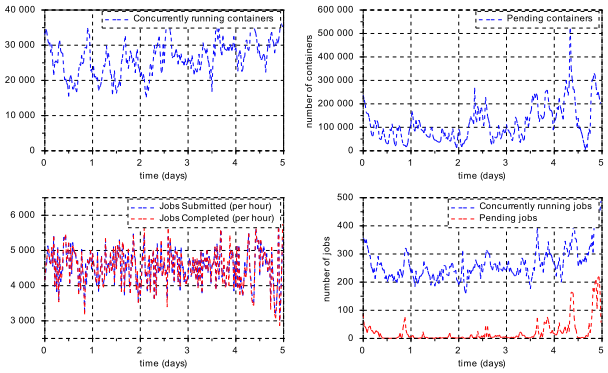


Figure 3: YARN jobs/containers statistics on a 2500 nodes production grid at Yahoo!.

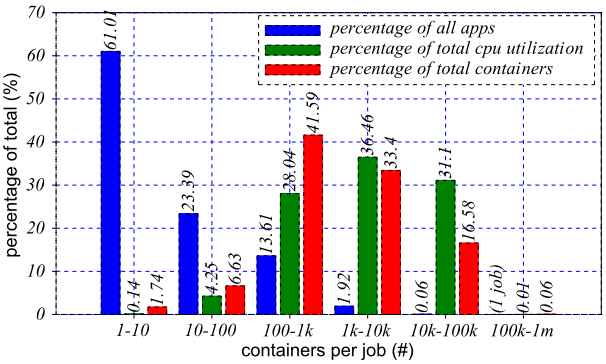


Figure 4: Job size distribution, and resource utilization

tens to these notifications and retries map or reduce tasks by requesting new containers from the RM.

With that, we conclude our coverage of the architecture and dive into YARN's real-world installations.

4 YARN in the real-world

We are aware of active use of YARN within several large companies. The following reports our experience running YARN at Yahoo!. We then follow it up with a discussion on few of the popular frameworks that have already been ported to YARN.

4.1 YARN at Yahoo!

Yahoo! upgraded its production grids from one of the stable branches of classic Hadoop to YARN. The statistics we report in the following are related to the last 30 days prior to upgrading for classic Hadoop, and the average statistics from the upgrade time (different for each grid) up until June 13th, 2013. An important caveat in interpreting the following statistics: the results we are about to present come from a real-life upgrading experience on large clusters, where the focus has been on maintaining the availability of a critical shared resource and not on scientific repeatability/consistency, as in a typical synthetic experiment. To help the reader easily interpret results, we will report a few global statistics, then focus on a large grid for which hardware has (mostly) not changed before and after the upgrade, and finally characterize the workload shift on that specific grid.

4.1.1 YARN across all clusters

In summary, after the upgrade, across all clusters, YARN is processing about **500,000 jobs daily**, for a grand total of over 230 compute-years every single day. The underlying storage exceeds 350 PB.

While one of the initial goals of YARN was to improve scalability, Yahoo! reported that they are not running clusters any bigger than 4000 nodes which used be the largest cluster's size before YARN. Significant gains in resource utilization delivered by YARN have increased the number of jobs that each grid can sustain. This has simply removed the need to scale further for the moment, and has even allowed the operational team to delay the re-provisioning of over 7000 nodes that have been decommissioned.

On the other side, it appears that the log aggregation component of YARN has increased the pressure on the HDFS NameNode, especially for large jobs. The NameNode is now estimated to be the scalability bottleneck in Yahoo's clusters. Fortunately, much work is ongoing in the community to both improve NameNode throughput and to limit the pressure on the NameNode by optimizing YARN log aggregation. Scalability concerns within YARN have been observed on a few large clusters with massive amounts of small applications, but recent improvements in heartbeat handling have mitigated some of these issues.

4.1.2 Statistics on a specific cluster

We will now focus on the experience of and statistics reported by Yahoo! grid operations team running YARN on one specific 2500 node grid.

Figure 2 shows the before/after load the operational team is comfortable running on a 2500 machine cluster. This is the busiest cluster at Yahoo! which is consistently being pushed close to its limit. In Figure 2a, we show a significant increase in the number of jobs running: **from about 77k** on the 1.0 version of Hadoop to roughly 100k jobs regularly on YARN. Moreover, a sustained throughput of 125k jobs per day was achieved on this cluster, which **peaks at about 150k jobs** a day (or almost about twice the number of jobs that represents the previous comfort-zone for this cluster). Average job size increased from 58 maps, 20 reducers to 85 maps, 16

reducers - note that the jobs include user jobs as well as other tiny jobs that get spawned by system applications like data copy/transfer via distcp, Oozie [19] and other data management frameworks. Similarly, the number of tasks across all jobs increased from 4M on Hadoop 1.0 to about 10M on average on YARN, with a sustained load of about 12M and a peak of about 15M - see Figure 2.

Another crucial metric to estimate efficiency of a cluster management system is the average resource utilization. Again, the shift in workload makes the statistics we are about to discuss less useful for direct comparison, but we observed a significant increase in CPU-utilization. On Hadoop 1.0, the estimated CPU utilization [11] was hovering around 320% or 3.2/16 cores pegged on each of the 2500 machines. **Essentially, moving to YARN, the CPU utilization almost doubled** to the equivalent of 6 continuously pegged cores per box and with peaks up to 10 fully utilized cores. In aggregate, this indicates that YARN was capable of keeping about 2.8*2500 = **7000 more cores completely busy running user code**. This is consistent with the increase in number of jobs and tasks running on the cluster we discussed above. One of the most important architectural differences that partially explains these improvements is the removal of the static split between map and reduce slots.

In Figure 3, we plot several job statistics over time: concurrently running and pending containers, jobs submitted, completed, running and pending. This shows the ability of the resource manager to handle large number of applications, container requests, and executions. The version of the CapacityScheduler used in this cluster does not use preemption yet—as this is a recently added feature. While preemption has not been tested at scale yet, we believe that careful use of preemption will significantly increase cluster utilization, we show a simple microbenchmark in Section 5.3.

To further characterize the nature of the workload running on this cluster, we present in Figure 4 a histogram for different size applications depicting: 1) the total number of applications in each bucket, 2) the total amount of CPU used by applications in that bucket, and 3) the total number of containers used by applications in that bucket. As observed in the past, while a large number of applications are very small, they account for a tiny fraction of cluster capacity (about 3.5 machines worth of CPU in this 2500 machines cluster). Interestingly, if we compare the slot utilization vs CPU utilization we observe that large jobs seem to use more CPU for each container. This is consistent with better tuning of large jobs (e.g., consistent use of compression), and possibly longer running tasks, thus amortizing startup overhead.

Overall, Yahoo! reports that the engineering effort put in to harden YARN into a production ready, scalable system, was well worth the effort. Plans to upgrade to the latest version, 2.1-beta, continue apace. After over 36,000 years of aggregated compute-time and several months of production stress-testing, Yahoo's operational team confirms that YARN's architectural shift has been very beneficial for their workloads. This is well summarized in the following quote: *"upgrading to YARN was equivalent to adding 1000 machines [to this 2500 machines cluster]"*.

[11] Computed by aggregating all metered CPU time and dividing by metered node uptime.

4.2 Applications and frameworks

A key requirement for YARN was to enable greater flexibility of programming model. This has been validated by the many programming frameworks that YARN has already attracted, despite its freshly-beta status at the time of this writing. We briefly summarize some projects either native to YARN or ported to the platform to illustrate the generality of its architecture.

Apache Hadoop MapReduce already works on top of YARN with almost the same feature-set. It is tested at scale, rest of ecosystem projects like Pig, Hive, Oozie, etc. are modified to work on top of MR over YARN, together with standard benchmarks performing at par or better compared to classic Hadoop. The MapReduce community has made sure that applications written against 1.x can run on top of YARN in a fully binary compatible manner (mapred APIs) or just by recompiling (source compatibility for mapreduce APIs).

Apache Tez is an Apache project (Incubator at the time of this writing) which aims to provide a generic directed-acyclic-graph (DAG) execution framework. One of its goals is to provide a collection of building blocks which can be composed into an arbitrary DAG (including a simple 2-stage (Map and Reduce) DAG to maintain compatibility with MapReduce). Tez provides query execution systems like Hive and Pig with a more natural model for their execution plan, as against forcing these plans to be transformed into MapReduce. The current focus is on speeding up complex Hive and Pig queries which typically require multiple MapReduce jobs, allowing to run as a single Tez job. In the future, rich features such as general support for interactive queries and generic DAGs will be considered.

Spark is an open-source research project from UC Berkeley [32], that targets machine learning and interactive querying workloads. The central idea of resilient distributed datasets (RDD) is leveraged to achieve significant performance improvements over classic MapReduce for this class of applications. Spark has been recently ported to YARN [**?**].

Dryad [18] provides DAG as the abstraction of execution flow, and it has been integrated with LINQ [31]. The

version ported to YARN is 100% native C++ and C# for worker nodes, while the ApplicationMaster leverages a thin layer of Java interfacing with the ResourceManager around the native Dryad graph manager. Eventually the Java layer will be substituted by direct interaction with protocol-buffer interfaces. Dryad-on-YARN is fully compatible with its non-YARN version.

Giraph is a highly scalable, vertex-centric graph computation framework. It was originally designed to run on top of Hadoop 1.0 as a Map-only job, where one map is special and behaves as coordinator. The port to YARN of Giraph is very natural, the execution coordinator role is taken by the ApplicationMaster, and resources are requested dynamically.

Storm is an open-source distributed, real-time processing engine, designed to scale across a cluster of machines and provide parallel stream processing. A common use case combines Storm for online computation and MapReduce as batch processor. By porting Storm on YARN a great deal of flexibility in resource allocation can be unblocked. Moreover, the shared access to the underlying HDFS storage layer simplifies the design of multi-framework workloads.

REEF meta-framework: YARN's flexibility comes at potentially significant effort for application implementors. Writing an ApplicationMaster and handling all aspects of fault tolerance, execution flow, coordination, etc. is a non-trivial endeavor. The REEF project [10] recognizes this and factors out several hard-to-build components that are common to many applications. This includes storage management, caching, fault-detection, checkpointing, push-based control flow (showcased experimentally later), and container reuse. Framework designers can build on top of REEF and more easily than directly on YARN and reuse many common services/libraries provided by REEF. REEF design makes it suitable for both MapReduce and DAG like executions as well as iterative and interactive computations.

Hoya is a Java-tool designed to leverage YARN to spin up dynamic HBase clusters[21] on demand. HBase clusters running on YARN can also grow and shrink dynamically (in our test cluster, RegionServers can be added/removed in less than 20 seconds). While the implications of mixing service and batch workloads in YARNare still being explored, early results from this project are encouraging.

5 Experiments

In the previous section, we established the real-world success of YARN by reporting on large production deployments and a thriving ecosystem of frameworks. In this section, we present more specific experimental results to demonstrate some of YARN's wins.

5.1 Beating the sort record

At the time of this writing, the MapReduce implementation on YARN is officially [12] holding both the Daytona and Indy GraySort benchmark records, sorting 1.42TB/min. The same system also reported (outside the competition) MinuteSort results sorting 1.61TB & 1.50TB in a minute, better than the current records. The experiments were run on 2100 nodes each equipped with two 2.3Ghz hexcore Xeon E5-2630, 64 GB memory, and 12x3TB disks each. The summary of results is provided in the following table:

Benchmark	Data Type	Data Size	Time	Rate
Daytona GraySort	no-skew	102.5 TB	72min	1.42TB/min
Daytona GraySort	skew	102.5 TB	117min	0.87TB/min
Daytona MinuteSort	no-skew	11497.86 GB	87.242 sec	-
Daytona MinuteSort	skew	1497.86 GB	59.223 sec	-
Indy MinuteSort	no-skew	1612.22 GB	58.027 sec	-

The full report [16] provides a detailed description of the experiments.

5.2 MapReduce benchmarks

MapReduce continues to be the most important and commonly used application on top of YARN. We are pleased to report that most of the standard Hadoop benchmarks perform better on YARN in Hadoop 2.1.0 compared to the current stable Hadoop-1 release 1.2.1. The summary of the MapReduce benchmarks on a 260 node cluster comparing 1.2.1 with 2.1.0 is provided below. Each slave node is running 2.27GHz Intel(R) Xeon(R) CPU totalling to 16 cores, has 38GB physical memory, and 6x1TB 7200 RPM disks each, formatted with ext3 file-system. The network bandwidth per node is 1Gb/sec. Each node runs a DataNode and a NodeManager with 24GB RAM allocated for containers. We run 6 maps and 3 reduces in 1.2.1, and 9 containers in 2.1.0. Each map occupies 1.5GB JVM heap and 2GB total memory, while each reduce takes 3GB heap 4GB total. JobTracker/ResourceManager run on a dedicated machine so is the HDFS NameNode.

Benchmark	Avg runtime (s)		Throughput(GB/s)	
	1.2.1	2.1.0	1.2.1	2.1.0
RandomWriter	222	228	7.03	6.84
Sort	475	398	3.28	3.92
Shuffle	951	648	-	-
AM Scalability	1020	353/303	-	-
Terasort	175.7	215.7	5.69	4.64
Scan	59	65	-	-
Read DFSIO	50.8	58.6	-	-
Write DFSIO	50.82	57.74	-	-

Table 1: Results from canonical Hadoop benchmarks

[12] Source: `sortbenchmark.org` as of June, 2013.

We interpret the results of these benchmarks as follows. The *sort* benchmark measures the end-to-end time for a 1.52TB (6GB per node) sort in HDFS, using default settings. The *shuffle* benchmark calibrates how fast the intermediate outputs from m maps are shuffled to n reduces using only synthetic data; records are neither read from nor written to HDFS. While the sort benchmark would typically benefit from improvements to the HDFS data path, both benchmarks perform better on YARN primarily due to significant improvements in the MapReduce runtime itself: map-side sort improvements, a reduce client that pipelines and batches transfers of map output, and a server-side shuffle based on Netty [3]. The *scan* and *DFSIO* jobs are canonical benchmarks used to evaluate HDFS and other distributed filesystems run under Hadoop MapReduce; the results in table 1 are a coarse measure of the effect attributable to HDFS in our experiments. Our access to the cluster was too brief to debug and characterize the middling performance from the 2.1.0 filesystem. Despite this noise, and even though YARN's design optimizes for multi-tenant throughput, its performance for single jobs is competitive with the central coordinator.

The *AM scalability* benchmark measures single-job robustness by saturating the AM with container bookkeeping duties. Table 1 includes two measurements of the MapReduce AM. The first experiment restricts available resources to match the slots available to the 1.x deployment. When we remove this artificial limit and allow YARN to use the full node, its performance improves significantly. The two experiments estimate the overhead of typed slots. We also attribute improved performance to more frequent node heartbeats and faster scheduling cycles, which we discuss in greater detail below. Since YARN is principally responsible for distributing and starting applications, we consider the scalability benchmark to be a critical metric.

Some architectural choices in YARN targeted bottlenecks we observed in production clusters. As discussed in section 2.3, typed slots harm throughput by creating an artificial mismatch between a fungible supply of resources on a node and the semantics of executing Hadoop MapReduce tasks. While section 4.1 covers the gains in aggregate workloads, we saw benefits to scheduling even single jobs. We attribute the bulk of this gain to improved heartbeat handling. In Hadoop-1, each node could heartbeat only every 30-40 seconds in large clusters due to coarse-grained locking in the JobTracker. Despite clever workarounds to lower latency, such as short-circuit paths for handling lightweight updates and adaptive delays to hasten important notifications, reconciling state in the JobTracker remained the principal cause of latency in large clusters. In contrast, NodeManagers heartbeat every 1-3 seconds. The RM code is more scalable, but it also solves a humbler set of constraints per request.

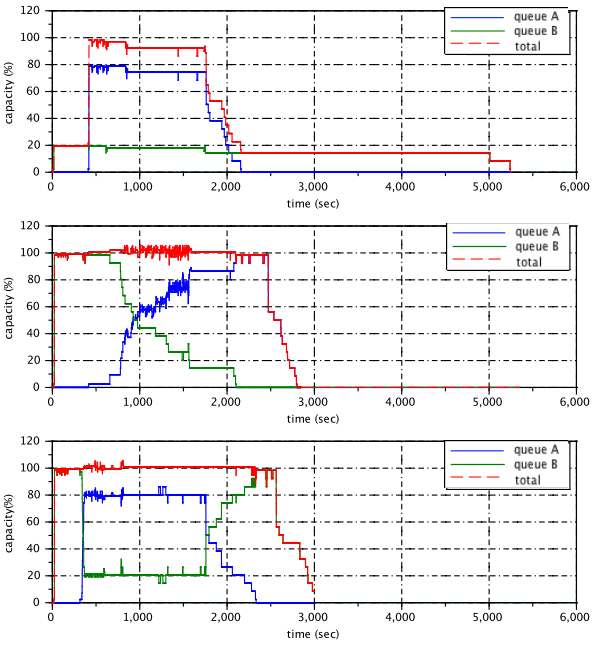


Figure 5: Effect of work-preserving preemption on the CapacityScheduler efficiency.

5.3 Benefits of preemption

In Figure 5, we demonstrate a recently-added feature in YARN: the ability to enforce global properties using work-preserving preemption. We ran experiments on a small (10 machine) cluster, to highlight the potential impact of work-preserving preemption. The cluster runs CapacityScheduler, configured with two queues A and B, respectively entitled to 80% and 20% of the capacity. A MapReduce job is submitted in the smaller queue B, and after a few minutes another MapReduce job is submitted in the larger queue A. In the graph, we show the capacity assigned to each queue under three configurations: 1) no capacity is offered to a queue beyond its guarantee (fixed capacity) 2) queues may consume 100% of the cluster capacity, but no preemption is performed, and 3) queues may consume 100% of the cluster capacity, but containers may be preempted. Work-preserving preemption allows the scheduler to overcommit resources for queue B without worrying about starving applications in queue A. When applications in queue A request resources, the scheduler issues preemption requests, which are serviced by the ApplicationMaster by checkpointing its tasks and yielding containers. This allows queue A to obtain all its guaranteed capacity (80% of cluster) in a few seconds, as opposed to case (2) in which the capacity rebalancing takes

about 20 minutes. Finally, since the preemption we use is checkpoint-based and does not waste work, the job running in B can restart tasks from where they left off, and it does so efficiently.

5.4 Improvements with Apache Tez

We present some rudimentary improvements when running a decision support query on Apache Hive running against Apache Tez (the integration in early stages at the time of this writing). Query 12 from TPC-DS benchmark [23], involves few joins, filters and group by aggregations. Even after aggressive plan level optimizations, Hive generates an execution plan consisting of multiple jobs when using MapReduce. The same query results in a linear DAG when executing against Tez, with a single Map stage followed by multiple Reduce stages. The query execution time when using MapReduce is 54 seconds on a 20 node cluster against 200 scale factor data, and this improves to 31 seconds when using Tez. Most of this saving can be attributed to scheduling and launching overheads of multiple MapReduce jobs and avoiding the unnecessary steps of persisting outputs of the intermediate MapReduce jobs to HDFS.

5.5 REEF: low latency with sessions

One of the key aspects of YARN is that it enables frameworks built on top of it to manage containers and communications as they see fit. We showcase this by leveraging the notion of container reuse and push-based communications provided by REEF. The experiment is based on a simple distributed-shell application built on top of REEF. We measure the client-side latency on a completely idle cluster when submitting a series of unix commands (e.g., `date`). The first command that is issued incurs the full cost of scheduling the application and acquiring containers, while subsequent commands are quickly forwarded through the Client and ApplicationMaster to already running containers for execution. The push-based messaging further reduces latency. The speedup in our experiment is very substantial, approaching three orders of magnitude—**from over 12sec to 31ms in average**.

6 Related work

Others have recognized the same limitations in the classic Hadoop architecture, and have concurrently developed alternative solutions, which can be closely compared to YARN. Among the many efforts the most closely resembling YARN are: Mesos [17], Omega [26], Corona [14], and Cosmos [8], maintained and used respectively by Twitter, Google, Facebook and Microsoft.

These systems share a common inspiration, and the high-level goal of improving scalability, latency and programming model flexibility. The many architectural differences are a reflection of diverse design priorities, and sometimes simply the effect of different historical contexts. While a true quantitative comparison is impossible to provide, we will try to highlight some of the architectural differences and our understanding of their rationale.

Omega's design leans more heavily towards distributed, multi-level scheduling. This reflects a greater focus on scalability, but makes it harder to enforce global properties such as capacity/fairness/deadlines. To this goal the authors seem to rely on coordinated development of the various frameworks that will be respectful of each other at runtime. This is sensible for a closed-world like Google, but not amenable to an open platform like Hadoop where arbitrary frameworks from diverse independent sources are share the same cluster.

Corona uses push based communication as opposed to the heartbeat based control-plane framework approach in YARN and other frameworks. The latency/scalability trade- offs are non-trivial and would deserve a detailed comparison.

While Mesos and YARN both have schedulers at two levels, there are two very significant differences. First, Mesos is an offer-based resource manager, whereas YARN has a request-based approach. YARN allows the AM to ask for resources based on various criteria including locations, allows the requester to modify future requests based on what was given and on current usage. Our approach was necessary to support the location based allocation. Second, instead of a per-job intra-framework scheduler, Mesos leverages a pool of central schedulers (e.g., classic Hadoop or MPI). YARN enables late binding of containers to tasks, where each individual job can perform local optimizations, and seems more amenable to rolling upgrades (since each job can run on a different version of the framework). On the other side, per-job ApplicationMaster might result in greater overhead than the Mesos approach.

Cosmos closely resembles Hadoop 2.0 architecturally with respect to storage and compute layers with the key difference of not having a central resource manager. However, it seems to be used for a single application type: Scope [8]. By virtue of a more narrow target Cosmos can leverage many optimizations such as native compression, indexed files, co-location of partitions of datasets to speed up Scope. The behavior with multiple application frameworks is not clear.

Prior to these recent efforts, there is a long history of work on resource management and scheduling - Condor [29], Torque [7], Moab [13] and Maui [20]. Our early Hadoop clusters used some of these systems, but

we found that they could not support the MapReduce model in a first-class way. Specifically, neither the data locality nor the elastic scheduling needs of map and reduce phases were expressible, so one was forced to allocate "virtual" Hadoop with the attendant utilization costs discussed in section 2.1. Perhaps some of these issues were due to the fact that many of these distributed schedulers were originally created to support MPI style and HPC application models and running coarse-grained non-elastic workloads. These cluster schedulers do allow clients to specify the types of processing environments, but unfortunately not locality constraints which is a key concern for Hadoop.

Another class of related technologies comes from the world of cloud infrastructures such as EC2, Azure, Eucalyptus and VMWare offerings. These mostly target VM-based sharing of a cluster, and are generally designed for long running processes (as the VM boot-times overheads are prohibitive).

7 Conclusion

In this paper, we summarized our recollection of the history of Hadoop, and discussed how wild adoption and new types of applications has pushed the initial architecture well beyond what it was designed to accomplish. We then described an evolutionary, yet profound, architectural transformation that lead to YARN. Thanks to the decoupling of resource management and programming framework, YARN provides: 1) greater scalability, 2) higher efficiency, and 3) enables a large number of different frameworks to efficiently share a cluster. These claims are substantiated both experimentally (via benchmarks), and by presenting a massive-scale production experience of Yahoo!—which is now 100% running on YARN. Finally, we tried to capture the great deal of excitement that surrounds this platform, by providing a snapshot of community activity and by briefly reporting on the many frameworks that have been ported to YARN. We believe YARN can serve as both a solid production framework and also as an invaluable playground for the research community.

8 Acknowledgements

We began our exposition of YARN by acknowledging the pedigree of its architecture. Its debt to all the individuals who developed, operated, tested, supported, documented, evangelized, funded, and most of all, *used* Apache Hadoop over the years is incalculable. In closing, we recognize a handful. Ram Marti and Jitendranath Panday influenced its security design. Dick King, Krishna Ramachandran, Luke Lu, Alejandro Abdelnur, Zhijie Shen, Omkar Vinit Joshi, Jian He have made or continue to make significant contributions to the implementation. Karam Singh, Ramya Sunil, Rajesh Balamohan and Srigurunath Chakravarthi have been ever helpful in testing and performance benchmarking. Rajive Chittajallu and Koji Noguchi helped in channeling requirements and insight from operations and user support points of view respectively. We thank Yahoo! for both supporting YARN via massive contributions and heavy usage and also for sharing the statistics on some of its clusters. We thank Dennis Fetterly and Sergiy Matusevych for providing some of the experimental results on Dryad and REEF, and Mayank Bansal for helping with 2.1.0 MapReduce benchmarks. Last but not the least, Apache Hadoop YARN continues to be a community driven open source project and owes much of its success to the Apache Hadoop YARN and MapReduce communities—a big thanks to all the contributors and committers who have helped YARN in every way possible.

A Classic Hadoop

Before YARN, Hadoop MapReduce clusters were composed of a master called JobTracker (JT) and worker nodes running *TaskTrackers* (TT). Users submitted MapReduce jobs to the JT which coordinated its execution across the TTs. A TT was configured by an operator with a fixed number of *map slots* and *reduce slots*. TTs periodically heartbeated into the JT to report the status of running tasks on that node and to affirm its liveness. On a heartbeat, the JT updated its state corresponding to the tasks running on that node, taking actions on behalf of that job (e.g.,scheduling failed tasks for re-execution), matching fallow slots on that node against the scheduler invariants, matching eligible jobs against available resources, favoring tasks with data local to that node.

As the central arbiter of the compute cluster, the JT was also responsible for admission control, tracking the liveness of TTs (to re-execute running tasks or tasks whose output becomes unavailable), launching tasks speculatively to route around slow nodes, reporting job status to users through a web server, recording audit logs and aggregate statistics, authenticating users and many other functions; each of these limited its scalability.

References

[1] Apache hadoop. http://hadoop.apache.org.

[2] Apache tez. `http://incubator.apache.org/projects/tez.html`.

[3] Netty project. http://netty.io.

[4] Storm. http://storm-project.net/.

[5] H. Ballani, P. Costa, T. Karagiannis, and A. I. Rowstron. Towards predictable datacenter networks. In *SIGCOMM*, volume 11, pages 242–253, 2011.

[6] F. P. Brooks, Jr. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[7] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounie, P. Neyron, and O. Richard. A batch scheduler with high level components. In *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, volume 2, pages 776–783 Vol. 2, 2005.

[8] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, Aug. 2008.

[9] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with orchestra. *SIGCOMM-Computer Communication Review*, 41(4):98, 2011.

[10] B.-G. Chun, T. Condie, C. Curino, R. Ramakrishnan, R. Sears, and M. Weimer. Reef: Retainable evaluator execution framework. In *VLDB 2013, Demo*, 2013.

[11] B. F. Cooper, E. Baldeschwieler, R. Fonseca, J. J. Kistler, P. Narayan, C. Neerdaels, T. Negrin, R. Ramakrishnan, A. Silberstein, U. Srivastava, et al. Building a cloud for Yahoo! *IEEE Data Eng. Bull.*, 32(1):36–43, 2009.

[12] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[13] W. Emeneker, D. Jackson, J. Butikofer, and D. Stanzione. Dynamic virtual clustering with xen and moab. In G. Min, B. Martino, L. Yang, M. Guo, and G. Rnger, editors, *Frontiers of High Performance Computing and Networking, ISPA 2006 Workshops*, volume 4331 of *Lecture Notes in Computer Science*, pages 440–451. Springer Berlin Heidelberg, 2006.

[14] Facebook Engineering Team. *Under the Hood: Scheduling MapReduce jobs more efficiently with Corona*. http://on.fb.me/TxUsYN, 2012.

[15] D. Gottfrid. *Self-service prorated super-computing fun*. http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun, 2007.

[16] T. Graves. *GraySort and MinuteSort at Yahoo on Hadoop 0.23*. http://sortbenchmark.org/Yahoo2013Sort.pdf, 2013.

[17] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: a platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association.

[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[19] M. Islam, A. K. Huang, M. Battisha, M. Chiang, S. Srinivasan, C. Peters, A. Neumann, and A. Abdelnur. Oozie: towards a scalable workflow management system for hadoop. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, page 4. ACM, 2012.

[20] D. B. Jackson, Q. Snell, and M. J. Clement. Core algorithms of the maui scheduler. In *Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*, JSSPP '01, pages 87–102, London, UK, UK, 2001. Springer-Verlag.

[21] S. Loughran, D. Das, and E. Baldeschwieler. *Introducing Hoya – HBase on YARN*. http://hortonworks.com/blog/introducing-hoya-hbase-on-yarn/, 2013.

[22] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[23] R. O. Nambiar and M. Poess. The making of tpcds. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 1049–1058. VLDB Endowment, 2006.

[24] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[25] O. O'Malley. *Hadoop: The Definitive Guide*, chapter Hadoop at Yahoo!, pages 11–12. O'Reilly Media, 2012.

[26] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 351–364, New York, NY, USA, 2013. ACM.

[27] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[28] T.-W. N. Sze. The two quadrillionth bit of π is 0! http://developer.yahoo.com/blogs/hadoop/two-quadrillionth-bit-0-467.html.

[29] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005.

[30] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Z. 0002, S. Anthony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using Hadoop. In F. Li, M. M. Moro, S. Ghandeharizadeh, J. R. Haritsa, G. Weikum, M. J. Carey, F. Casati, E. Y. Chang, I. Manolescu, S. Mehrotra, U. Dayal, and V. J. Tsotras, editors, *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 996–1005. IEEE, 2010.

[31] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[32] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.