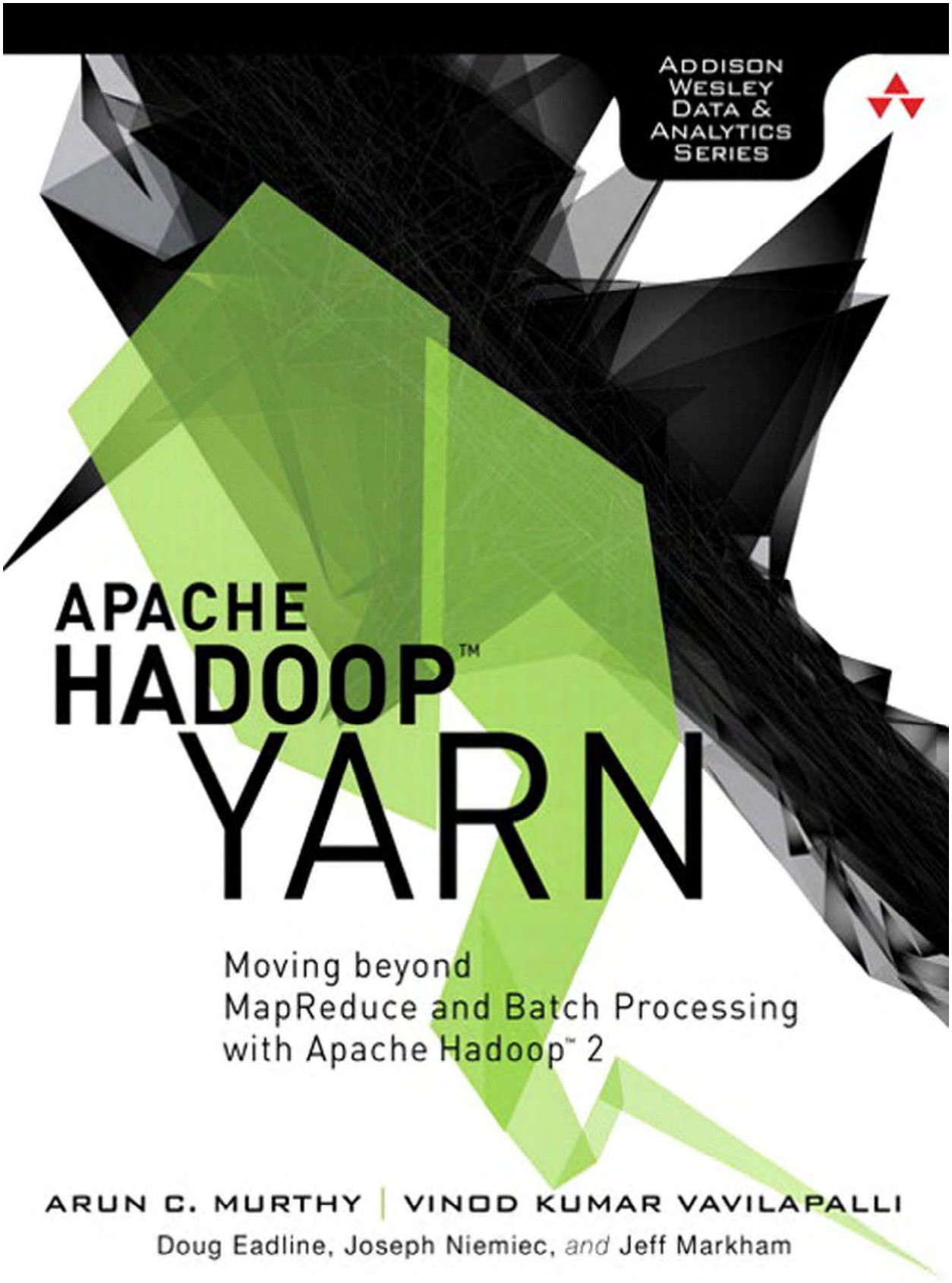


Petitioner has added romanettes to pages (a) -(c). Otherwise, it leaves the original page numbering.

MICROSOFT CORP.
EXHIBIT 1012

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the United States, please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Murthy, Arun C.
Apache Hadoop YARN : moving beyond MapReduce and batch processing with Apache Hadoop 2 / Arun C. Murthy, Vinod Kumar Vavilapalli, Doug Eadline, Joseph Niemiec, Jeff Markham.
pages cm
Includes index.
ISBN 978-0-321-93450-5 (pbk. : alk. paper)
1. Apache Hadoop. 2. Electronic data processing—Distributed processing. I. Title.
QA76.9.D5M97 2014
004'.36—dc23

2014003391

Copyright © 2014 Hortonworks Inc.

Apache, Apache Hadoop, Hadoop, and the Hadoop elephant logo are trademarks of The Apache Software Foundation. Used with permission. No endorsement by The Apache Software Foundation is implied by the use of these marks.

Hortonworks is a trademark of Hortonworks, Inc., registered in the U.S. and other countries.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

ISBN-13: 978-0-321-93450-5
ISBN-10: 0-321-93450-4
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, March 2014

Contents

Foreword by Raymie Stata xiii

Foreword by Paul Dix xv

Preface xvii

Acknowledgments xxi

About the Authors xxv

1 Apache Hadoop YARN: A Brief History and Rationale 1

Introduction 1

Apache Hadoop 2

Phase 0: The Era of Ad Hoc Clusters 3

Phase 1: Hadoop on Demand 3

HDFS in the HOD World 5

Features and Advantages of HOD 6

Shortcomings of Hadoop on Demand 7

Phase 2: Dawn of the Shared Compute Clusters 9

Evolution of Shared Clusters 9

Issues with Shared MapReduce Clusters 15

Phase 3: Emergence of YARN 18

Conclusion 20

2 Apache Hadoop YARN Install Quick Start 21

Getting Started 22

Steps to Configure a Single-Node YARN Cluster 22

Step 1: Download Apache Hadoop 22

Step 2: Set JAVA_HOME 23

Step 3: Create Users and Groups 23

Step 4: Make Data and Log Directories 23

Step 5: Configure core-site.xml 24

Step 6: Configure hdfs-site.xml 24

Step 7: Configure mapred-site.xml 25

Step 8: Configure yarn-site.xml 25

Step 9: Modify Java Heap Sizes 26

Step 10: Format HDFS 26

Step 11: Start the HDFS Services 27

Step 12: Start YARN Services **28**
Step 13: Verify the Running Services Using the Web Interface **28**
Run Sample MapReduce Examples **30**
Wrap-up **31**

3 Apache Hadoop YARN Core Concepts 33
Beyond MapReduce **33**
The MapReduce Paradigm **35**
Apache Hadoop MapReduce **35**
The Need for Non-MapReduce Workloads **37**
Addressing Scalability **37**
Improved Utilization **38**
User Agility **38**
Apache Hadoop YARN **38**
YARN Components **39**
ResourceManager **39**
ApplicationMaster **40**
Resource Model **41**
ResourceRequests and Containers **41**
Container Specification **42**
Wrap-up **42**

4 Functional Overview of YARN Components 43
Architecture Overview **43**
ResourceManager **45**
YARN Scheduling Components **46**
FIFO Scheduler **46**
Capacity Scheduler **47**
Fair Scheduler **47**
Containers **49**
NodeManager **49**
ApplicationMaster **50**
YARN Resource Model **50**
Client Resource Request **51**
ApplicationMaster Container Allocation **51**
ApplicationMaster–Container Manager Communication **52**

Managing Application Dependencies 53
LocalResources Definitions 54
LocalResource Timestamps 55
LocalResource Types 55
LocalResource Visibilities 56
Lifetime of LocalResources 57
Wrap-up 57

5 Installing Apache Hadoop YARN 59
The Basics 59
System Preparation 60
Step 1: Install EPEL and pdsh 60
Step 2: Generate and Distribute ssh Keys 61
Script-based Installation of Hadoop 2 62
JDK Options 62
Step 1: Download and Extract the Scripts 63
Step 2: Set the Script Variables 63
Step 3: Provide Node Names 64
Step 4: Run the Script 64
Step 5: Verify the Installation 65
Script-based Uninstall 68
Configuration File Processing 68
Configuration File Settings 68
core-site.xml 68
hdfs-site.xml 69
mapred-site.xml 69
yarn-site.xml 70
Start-up Scripts 71
Installing Hadoop with Apache Ambari 71
Performing an Ambari-based Hadoop Installation 72
Step 1: Check Requirements 73
Step 2: Install the Ambari Server 73
Step 3: Install and Start Ambari Agents 73
Step 4: Start the Ambari Server 74
Step 5: Install an HDP2.X Cluster 75
Wrap-up 84

6 Apache Hadoop YARN Administration 85
Script-based Configuration **85**
Monitoring Cluster Health: Nagios **90**
Monitoring Basic Hadoop Services **92**
Monitoring the JVM **95**
Real-time Monitoring: Ganglia **97**
Administration with Ambari **99**
JVM Analysis **103**
Basic YARN Administration **106**
YARN Administrative Tools **106**
Adding and Decommissioning YARN Nodes **107**
Capacity Scheduler Configuration **108**
YARN WebProxy **108**
Using the JobHistoryServer **108**
Refreshing User-to-Groups Mappings **108**
Refreshing Superuser Proxy Groups Mappings **109**
Refreshing ACLs for Administration of ResourceManager **109**
Reloading the Service-level Authorization Policy File **109**
Managing YARN Jobs **109**
Setting Container Memory **110**
Setting Container Cores **110**
Setting MapReduce Properties **110**
User Log Management **111**
Wrap-up **114**

7 Apache Hadoop YARN Architecture Guide 115
Overview **115**
ResourceManager **117**
Overview of the ResourceManager Components **118**
Client Interaction with the ResourceManager **118**
Application Interaction with the ResourceManager **120**

Interaction of Nodes with the ResourceManager **121**
Core ResourceManager Components **122**
Security-related Components in the ResourceManager **124**
NodeManager **127**
Overview of the NodeManager Components **128**
NodeManager Components **129**
NodeManager Security Components **136**
Important NodeManager Functions **137**
ApplicationMaster **138**
Overview **138**
Liveliness **139**
Resource Requirements **140**
Scheduling **140**
Scheduling Protocol and Locality **142**
Launching Containers **145**
Completed Containers **146**
ApplicationMaster Failures and Recovery **146**
Coordination and Output Commit **146**
Information for Clients **147**
Security **147**
Cleanup on ApplicationMaster Exit **147**
YARN Containers **148**
Container Environment **148**
Communication with the ApplicationMaster **149**
Summary for Application-writers **150**
Wrap-up **151**

8 Capacity Scheduler in YARN 153
Introduction to the Capacity Scheduler **153**
Elasticity with Multitenancy **154**
Security **154**
Resource Awareness **154**
Granular Scheduling **154**
Locality **155**
Scheduling Policies **155**
Capacity Scheduler Configuration **155**

Queues **156**
Hierarchical Queues **156**
Key Characteristics **157**
Scheduling Among Queues **157**
Defining Hierarchical Queues **158**
Queue Access Control **159**
Capacity Management with Queues **160**
User Limits **163**
Reservations **166**
State of the Queues **167**
Limits on Applications **168**
User Interface **169**
Wrap-up **169**

9 MapReduce with Apache Hadoop YARN 171
Running Hadoop YARN MapReduce Examples **171**
Listing Available Examples **171**
Running the Pi Example **172**
Using the Web GUI to Monitor Examples **174**
Running the Terasort Test **180**
Run the TestDFSIO Benchmark **180**
MapReduce Compatibility **181**
The MapReduce ApplicationMaster **181**
Enabling Application Master Restarts **182**
Enabling Recovery of Completed Tasks **182**
The JobHistory Server **182**
Calculating the Capacity of a Node **182**
Changes to the Shuffle Service **184**
Running Existing Hadoop Version 1 Applications **184**
Binary Compatibility of org.apache.hadoop.mapred APIs **184**
Source Compatibility of org.apache.hadoop.mapreduce APIs **185**
Compatibility of Command-line Scripts **185**
Compatibility Tradeoff Between MRv1 and Early MRv2 (0.23.x) Applications **185**

Running MapReduce Version 1 Existing Code **187**
Running Apache Pig Scripts on YARN **187**
Running Apache Hive Queries on YARN **187**
Running Apache Oozie Workflows on YARN **188**
Advanced Features **188**
Uber Jobs **188**
Pluggable Shuffle and Sort **188**
Wrap-up **190**

10 Apache Hadoop YARN Application Example 191
The YARN Client **191**
The ApplicationMaster **208**
Wrap-up **226**

11 Using Apache Hadoop YARN Distributed-Shell 227
Using the YARN Distributed-Shell **227**
A Simple Example **228**
Using More Containers **229**
Distributed-Shell Examples with Shell Arguments **230**
Internals of the Distributed-Shell **232**
Application Constants **232**
Client **233**
ApplicationMaster **236**
Final Containers **240**
Wrap-up **240**

12 Apache Hadoop YARN Frameworks 241
Distributed-Shell **241**
Hadoop MapReduce **241**
Apache Tez **242**
Apache Giraph **242**
Hoya: HBase on YARN **243**
Dryad on YARN **243**
Apache Spark **244**
Apache Storm **244**

REEF: Retainable Evaluator Execution Framework **245**
Hamster: Hadoop and MPI on the Same Cluster **245**
Wrap-up **245**

A Supplemental Content and Code Downloads 247
Available Downloads **247**

B YARN Installation Scripts 249
install-hadoop2.sh **249**
uninstall-hadoop2.sh **256**
hadoop-xml-conf.sh **258**

C YARN Administration Scripts 263
configure-hadoop2.sh **263**

D Nagios Modules 269
check_resource_manager.sh **269**
check_data_node.sh **271**
check_resource_manager_old_space_pct.sh **272**

E Resources and Additional Information 277

F HDFS Quick Reference 279
Quick Command Reference **279**
Starting HDFS and the HDFS Web GUI **280**
Get an HDFS Status Report **280**
Perform an FSCK on HDFS **281**
General HDFS Commands **281**
List Files in HDFS **282**
Make a Directory in HDFS **283**
Copy Files to HDFS **283**
Copy Files from HDFS **284**
Copy Files within HDFS **284**
Delete a File within HDFS **284**
Delete a Directory in HDFS **284**
Decommissioning HDFS Nodes **284**

Index 287

1

Apache Hadoop YARN: A Brief History and Rationale

In this chapter we provide a historical account of why and how Apache Hadoop YARN came about. YARN's requirements emerged and evolved from the practical needs of long-existing cluster deployments of Hadoop, both small and large, and we discuss how each of these requirements ultimately shaped YARN.

YARN's architecture addresses many of these long-standing requirements, based on experience evolving the MapReduce platform. By understanding this historical context, readers can appreciate most of the design decisions that were made with YARN. These design decisions will repeatedly appear in Chapter 4, "Functional Overview of YARN Components," and Chapter 7, "Apache Hadoop YARN Architecture Guide."

Introduction

Several different problems need to be tackled when building a shared compute platform. Scalability is the foremost concern, to avoid rewriting software again and again whenever existing demands can no longer be satisfied with the current version. The desire to share physical resources brings up issues of multitenancy, isolation, and security. Users interacting with a Hadoop cluster serving as a long-running service inside an organization will come to depend on its reliable and highly available operation. To continue to manage user workloads in the least disruptive manner, serviceability of the platform is a principal concern for operators and administrators. Abstracting the intricacies of a distributed system and exposing clean but varied application-level paradigms are growing necessities for any compute platform.

Hadoop's compute layer has seen all of this and much more during its continuous and long progress. It went through multiple evolutionary phases in its architecture. We highlight the "Big Four" of these phases in the reminder of this chapter.

- "Phase 0: The Era of Ad Hoc Clusters" signaled the beginning of Hadoop clusters that were set up in an ad hoc, per-user manner.
- "Phase 1: Hadoop on Demand" was the next step in the evolution in the form of a common system for provisioning and managing private Hadoop MapReduce and HDFS instances on a shared cluster of commodity hardware.
- "Phase 2: Dawn of the Shared Compute Clusters" began when the majority of Hadoop installations moved to a model of a shared MapReduce cluster together with shared HDFS instances.
- "Phase 3: Emergence of YARN"—the main subject of this book—arose to address the demands and shortcomings of the previous architectures.

As the reader follows the journey through these various phases, it will be apparent how the requirements of YARN unfolded over time. As the architecture continued to evolve, existing problems would be solved and new use-cases would emerge, pushing forward further stages of advancements.

We'll now tour through the various stages of evolution one after another, in chronological order. For each phase, we first describe what the architecture looked like and what its advancements were from its previous generation, and then wind things up with its limitations—setting the stage for the next phase.

Apache Hadoop

To really comprehend the history of YARN, you have to start by taking a close look at the evolution of Hadoop itself. Yahoo! adopted Apache Hadoop in 2006 to replace the existing infrastructure that was then driving its WebMap application—the technology that builds a graph of the known web to power its search engine. At that time, the web-graph contained more than 100 billion nodes with roughly 1 trillion edges. The previous infrastructure, named "Dreadnaught," successfully served its purpose and grew well—starting from a size of just 20 nodes and expanding to 600 cluster nodes—but had reached the limits of its scalability. The software also didn't perform perfectly in many scenarios, including handling of failures in the clusters' commodity hardware. A significant shift in its architecture was required to scale out further to match the ever-growing size of the web. The distributed applications running under Dreadnought were very similar to MapReduce programs and needed to span clusters of machines and work at a large scale. This highlights the first requirement that would survive throughout early versions of Hadoop MapReduce, all the way to YARN—[Requirement 1] Scalability.

- **[Requirement 1] Scalability**

  The next-generation compute platform should scale horizontally to tens of thousands of nodes and concurrent applications.

For Yahoo!, by adopting a more scalable MapReduce framework, significant parts of the search pipeline could be migrated easily without major refactoring—which, in

turn, ignited the initial investment in Apache Hadoop. However, although the original push for Hadoop was for the sake of search infrastructure, other use-cases started taking advantage of Hadoop much faster, even before the migration of the web-graph to Hadoop could be completed. The process of setting up research grids for research teams, data scientists, and the like had hastened the deployment of larger and larger Hadoop clusters. Yahoo! scientists who were optimizing advertising analytics, spam filtering, personalization, and content initially drove Hadoop's evolution and many of its early requirements. In line with that evolution, the engineering priorities evolved over time, and Hadoop went through many intermediate stages of the compute platform, including ad hoc clusters.

Phase 0: The Era of Ad Hoc Clusters

Before the advent of ad hoc clusters, many of Hadoop's earliest users would use Hadoop as if it were similar to a desktop application but running on a host of machines. They would manually bring up a cluster on a handful of nodes, load their data into the Hadoop Distributed File System (HDFS), obtain the result they were interested in by writing MapReduce jobs, and then tear down that cluster. This was partly because there wasn't an urgent need for persistent data in Hadoop HDFS, and partly because there was no incentive for sharing common data sets and the results of the computations. As usage of these private clusters increased and Hadoop's fault tolerance improved, persistent HDFS clusters came into being. Yahoo! Hadoop administrators would install and manage a shared HDFS instance, and load commonly used and interesting data sets into the shared cluster, attracting scientists interested in deriving insights from them. HDFS also acquired a POSIX-like permissions model for supporting multiuser environments, file and namespace quotas, and other features to improve its multitenant operation. Tracing the evolution of HDFS is in itself an interesting endeavor, but we will focus on the compute platform in the remainder of this chapter.

Once shared HDFS instances came into being, issues with the not-yet-shared compute instances came into sharp focus. Unlike with HDFS, simply setting up a shared MapReduce cluster for multiple users potentially from multiple organizations wasn't a trivial step forward. Private compute cluster instances continued to thrive, but continuous sharing of the common underlying physical resources wasn't ideal. To address some of the multitenancy issues with manually deploying and tearing down private clusters, Yahoo! developed and deployed a platform called Hadoop on Demand.

Phase 1: Hadoop on Demand

The Hadoop on Demand (HOD) project was a system for provisioning and managing Hadoop MapReduce and HDFS instances on a shared cluster of commodity hardware. The Hadoop on Demand project predated and directly influenced how the developers eventually arrived at YARN's architecture. Understanding the HOD architecture and its eventual limitations is a first step toward comprehending YARN's motivations.

To address the multitenancy woes with the manually shared clusters from the previous incarnation (Phase 0), HOD used a traditional resource manager—Torque—together with a cluster scheduler—Maui—to allocate Hadoop clusters on a shared pool of nodes. Traditional resource managers were already being used elsewhere in high-performance computing environments to enable effective sharing of pooled cluster resources. By making use of such existing systems, HOD handed off the problem of cluster management to systems outside of Hadoop. On the allocated nodes, HOD would start MapReduce and HDFS daemons, which in turn would serve the user's data and application requests. Thus, the basic system architecture of HOD included these layers:

- A ResourceManager (RM) together with a scheduler
- Various HOD components to interact with the RM/scheduler and manage Hadoop
- Hadoop MapReduce and HDFS daemons
- A HOD shell and Hadoop clients

A typical session of HOD involved three major steps: allocate a cluster, run Hadoop jobs on the allocated cluster, and finally deallocate the cluster. Here is a brief description of a typical HOD-user session:

- Users would invoke a HOD shell and submit their needs by supplying a description of an appropriately sized compute cluster to Torque. This description included:
  - The number of nodes needed
  - A description of a special head-process called the RingMaster to be started by the ResourceManager
  - A specification of the Hadoop deployment desired
- Torque would enqueue the request until enough nodes become available. Once the nodes were available, Torque started the head-process called RingMaster on one of the compute nodes.
- The RingMaster was a HOD component and used another ResourceManager interface to run the second HOD component, HODRing—with one HODRing being present on each of the allocated compute nodes.
- The HODRings booted up, communicated with the RingMaster to obtain Hadoop commands, and ran them accordingly. Once the Hadoop daemons were started, HODRings registered with the RingMaster, giving information about the daemons.
- The HOD client kept communicating with the RingMaster to find out the location of the JobTracker and HDFS daemons.
- Once everything was set up and the users learned the JobTracker and HDFS locations, HOD simply got out the way and allowed the user to perform his or her data crunching on the corresponding clusters.

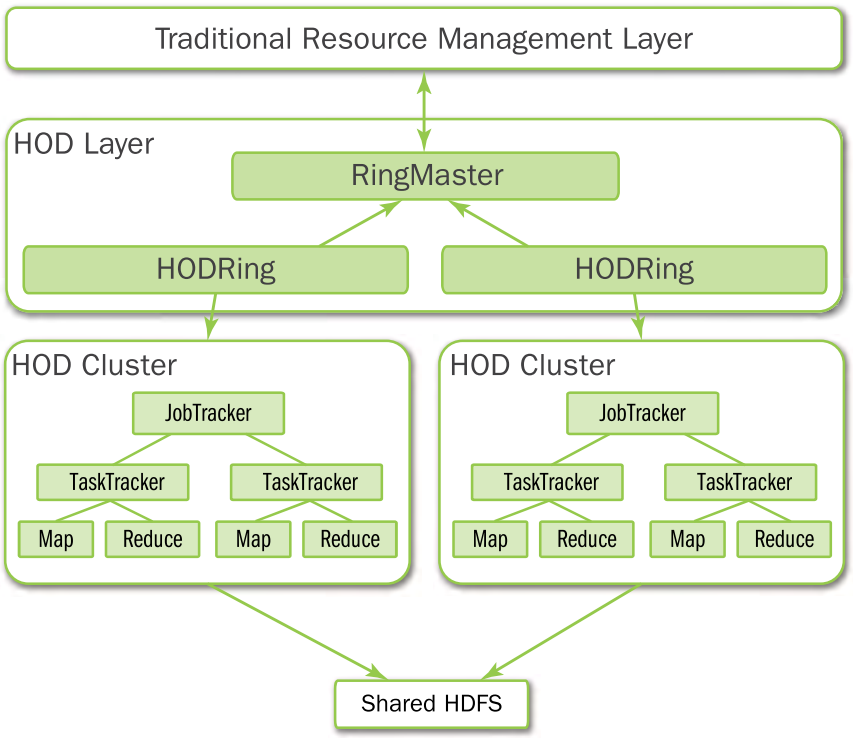


Figure 1.1 Hadoop on Demand architecture

- The user released a cluster once he or she was done running the data analysis jobs.

Figure 1.1 provides an overview of the HOD architecture.

HDFS in the HOD World

While HOD could also deploy HDFS clusters, most users chose to deploy the compute nodes across a shared HDFS instance. In a typical Hadoop cluster provisioned by HOD, cluster administrators would set up HDFS statically (without using HOD). This allowed data to be persisted in HDFS even after the HOD-provisioned clusters were deallocated. To use a statically configured HDFS, a user simply needed to point to an external HDFS instance. As HDFS scaled further, more compute clusters could be allocated through HOD, creating a cycle of increased experimentation by users over more data sets, leading to a greater return on investment. Because most user-specific MapReduce clusters were smaller than the largest HOD jobs possible, the JobTracker running for any single HOD cluster was rarely a bottleneck.

Features and Advantages of HOD

Because HOD sets up a new cluster for every job, users could run older and stable versions of Hadoop software while developers continued to test new features in isolation. Since the Hadoop community typically released a major revision every three months, the flexibility of HOD was critical to maintaining that software release schedule—we refer to this decoupling of upgrade dependencies as [Requirement 2] Serviceability.

- **[Requirement 2] Serviceability**

  The next-generation compute platform should enable evolution of cluster software to be completely decoupled from users' applications.

In addition, HOD made it easy for administrators and users to quickly set up and use Hadoop on an existing cluster under a traditional resource management system. Beyond Yahoo!, universities and high-performance computing environments could run Hadoop on their existing clusters with ease by making use of HOD. It was also a very useful tool for Hadoop developers and testers who needed to share a physical cluster for testing their own Hadoop versions.

Log Management

HOD could also be configured to upload users' job logs and the Hadoop daemon logs to a configured HDFS location when a cluster was deallocated. The number of log files uploaded to and retained on HDFS could increase over time in an unbounded manner. To address this issue, HOD shipped with tools that helped administrators manage the log retention by removing old log files uploaded to HDFS after a specified amount of time had elapsed.

Multiple Users and Multiple Clusters per User

As long as nodes were available and organizational policies were not violated, a user could use HOD to allocate multiple MapReduce clusters simultaneously. HOD provided the **list** and the **info** operations to facilitate the management of multiple concurrent clusters. The list operation listed all the clusters allocated so far by a user, and the info operation showed information about a given cluster—Torque job ID, locations of the important daemons like the HOD RingMaster process, and the RPC addresses of the Hadoop JobTracker and NameNode daemons.

The resource management layer had some ways of limiting users from abusing cluster resources, but the user interface for exposing those limits was poor. HOD shipped with scripts that took care of this integration so that, for instance, if some user limits were violated, HOD would update a public job attribute that the user could query against.

HOD also had scripts that integrated with the resource manager to allow a user to identify the account under which the user's Hadoop clusters ran. This was necessary because production systems on traditional resource managers used to manage accounts separately so that they could charge users for using shared compute resources.

Ultimately, each node in the cluster could belong to only one user's Hadoop cluster at any point of time—a major limitation of HOD. As usage of HOD grew along with its success, requirements around [Requirement 3] Multitenancy started to take shape.

- **[Requirement 3] Multitenancy**

  The next-generation compute platform should support multiple tenants to coexist on the same cluster and enable fine-grained sharing of individual nodes among different tenants.

Distribution of Hadoop Software

When provisioning Hadoop, HOD could either use a preinstalled Hadoop instance on the cluster nodes or request HOD to distribute and install a Hadoop tarball as part of the provisioning operation. This was especially useful in a development environment where individual developers might have different versions of Hadoop to test on the same shared cluster.

Configuration

HOD provided a very convenient mechanism to configure both the boot-up HOD software itself and the Hadoop daemons that it provisioned. It also helped manage the configuration files that it generated on the client side.

Auto-deallocation of Idle Clusters

HOD used to automatically deallocate clusters that were not running Hadoop jobs for a predefined period of time. Each HOD allocation included a monitoring facility that constantly checked for any running Hadoop jobs. If it detected no running Hadoop jobs for an extended interval, it automatically deallocated its own cluster, freeing up those nodes for future use.

Shortcomings of Hadoop on Demand

Hadoop on Demand proved itself to be a powerful and very useful platform, but Yahoo! ultimately had to retire it in favor of directly shared MapReduce clusters due to many of its shortcomings.

Data Locality

For any given MapReduce job, 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 doesn't know how blocks are distributed on HDFS, it allocates nodes without accounting for locality. The subset of nodes granted to a user's JobTracker will likely contain only a handful of relevant replicas and, if the user is unlucky, none. Many Hadoop clusters are characterized by a small number of very big jobs and a large number of small jobs. For most of the small jobs, most reads will emanate from remote hosts because of the insufficient information available from Torque.

Efforts were undertaken to mitigate this situation but achieved mixed results. One solution was to spread TaskTrackers across racks by modifying Torque/Maui itself and

making them rack-aware. Once this was done, any user's HOD compute cluster would be allocated nodes that were spread across racks. This made intra-rack reads of shared data sets more likely, but introduced other problems. The transfer of records between map and reduce tasks as part of MapReduce's shuffle phase would necessarily cross racks, causing a significant slowdown of users' workloads.

While such short-term solutions were implemented, ultimately none of them proved ideal. In addition, they all pointed to the fundamental limitation of the traditional resource management software—that is, the ability to understand data locality as a first-class dimension. This aspect of [Requirement 4] Locality Awareness is a key requirement for YARN.

- **[Requirement 4] Locality Awareness**

  The next-generation compute platform should support locality awareness—moving computation to the data is a major win for many applications.

Cluster Utilization

MapReduce jobs consist of multiple stages: a map stage followed by a shuffle and a reduce stage. Further, high-level frameworks like Apache Pig and Apache Hive often organize a workflow of MapReduce jobs in a directed-acyclic graph (DAG) of computations. Because clusters were not resizable between stages of a single job or between jobs when using HOD, most of the time the major share of the capacity in a cluster would be barren, waiting for the subsequent slimmer stages to be completed. In an extreme but very common scenario, a single reduce task running on one node could prevent a cluster of hundreds of nodes from being reclaimed. When all jobs in a colocation were considered, this approach could result in hundreds of nodes being idle in this state.

In addition, private MapReduce clusters for each user implied that even after a user was done with his or her workflows, a HOD cluster could potentially be idle for a while before being automatically detected and shut down.

While users were fond of many features in HOD, the economics of cluster utilization ultimately forced Yahoo! to pack its users' jobs into shared clusters. [Requirement 5] High Cluster Utilization is a top priority for YARN.

- **[Requirement 5] High Cluster Utilization**

  The next-generation compute platform should enable high utilization of the underlying physical resources.

Elasticity

In a typical Hadoop workflow, MapReduce jobs have lots of maps with a much smaller number of reduces, with map tasks being short and quick and reduce tasks being I/O heavy and longer running. With HOD, users relied on few heuristics when estimating how many nodes their jobs required—typically allocating their private HOD clusters based on the required number of map tasks (which in turn depends on the input size). In the past, this was the best strategy for users because more often than not, job latency was dominated by the time spent in the queues waiting for the

allocation of the cluster. This strategy, although the best option for individual users, leads to bad scenarios from the overall cluster utilization point of view. Specifically, sometimes all of the map tasks are finished (resulting in idle nodes in the cluster) while a few reduce tasks simply chug along for a long while.

Hadoop on Demand did not have the ability to grow and shrink the MapReduce clusters on demand for a variety of reasons. Most importantly, elasticity wasn't a first-class feature in the underlying ResourceManager itself. Even beyond that, as jobs were run under a Hadoop cluster, growing a cluster on demand by starting TaskTrackers wasn't cheap. Shrinking the cluster by shutting down nodes wasn't straightforward, either, without potentially massive movement of existing intermediate outputs of map tasks that had already run and finished on those nodes.

Further, whenever cluster allocation latency was very high, users would often share long-awaited clusters with colleagues, holding on to nodes for longer than anticipated, and increasing latencies even further.

Phase 2: Dawn of the Shared Compute Clusters

Ultimately, HOD architecture 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 management layer. This forced the next step of evolution—the majority of installations, including Yahoo!, moved to a model of a shared MapReduce cluster together with shared HDFS instances. The main components of this shared compute architecture were as follows:

- **A JobTracker**: A central daemon responsible for running all the jobs in the cluster. This is the same daemon that used to run jobs for a single user in the HOD world, but with additional functionality.
- **TaskTrackers**: The slave in the system, which executes one task at a time under directions from the JobTracker. This again is the same daemon as in HOD, but now runs the tasks of jobs from all users.

What follows is an exposition of shared MapReduce compute clusters. Shared MapReduce clusters working in tandem with shared HDFS instances is the dominant architecture of Apache Hadoop 1.x release lines. At the point of this writing, many organizations have moved beyond 1.x to the next-generation architecture, but at the same time multitudes of Hadoop deployments continue to the JobTracker/TaskTracker architecture and are looking forward to the migration to YARN-based Apache Hadoop 2.x release lines. Because of this, in what follows, note that we'll refer to the age of shared MapReduce-only shared clusters as *both* the past and the present.

Evolution of Shared Clusters

Moving to shared clusters from HOD-based architecture was nontrivial, and replacement of HOD was easier said than done. HOD, for all its problems, was originally designed to specifically address (and thus masked) many of the multitenancy issues

occurring in shared MapReduce clusters. Adding to that, HOD silently took advantage of some core features of the underlying traditional resource manager, which eventually became missing features when the clusters evolved to being native MapReduce shared clusters. In the remainder of this section, we'll describe salient characteristics of shared MapReduce deployments and indicate how the architecture gradually evolved away from HOD.

HDFS Instances

In line with how a shared HDFS architecture was established during the days of HOD, shared instances of HDFS continue to advance. During Phase 2, HDFS improved its scalability, acquired more features such as file-append, the new FileContext API for applications, Kerberos-based security features, high availability, and other performance features such as local short-circuit to data-node files directly.

Central JobTracker Daemon

The first step in the evolution of the MapReduce subsystem was to start running the JobTracker daemon as a shared resource across jobs, across users. This started with putting an abstraction for a cluster scheduler right inside the JobTracker, the details of which we explore in the next subsection. In addition, and unlike in the phase in which HOD was the norm, both developer testing and user validation revealed numerous deadlocks and race conditions in the JobTracker that were earlier neatly shielded by HOD.

JobTracker Memory Management

Running jobs from multiple users also drew attention to the issue of memory management of the JobTracker heap. At large clusters in Yahoo!, we had seen many instances in which a user, just as he or she used to allocate large clusters in the HOD world, would submit a job with many thousands of mappers or reducers. The configured heap of the JobTracker at that time hadn't yet reached the multiple tens of gigabytes observed with HDFS's NameNode. Many times, the JobTracker would expand these very large jobs in its memory to start scheduling them, only to run into heap issues and memory thrash and pauses due to Java garbage collection. The only solution at that time once such a scenario occurred was to restart the JobTracker daemon, effectively causing a downtime for the whole cluster. Thus, the JobTracker heap itself became a shared resource that needed features to support multitenancy, but smart scheduling of this scarce resource was hard. The JobTracker heap would store in-memory representations of jobs and tasks—some of them static and easily accountable, but other parts dynamic (e.g., job counters, job configuration) and hence not bounded.

To avoid the risks associated with a complex solution, the simplest proposal of limiting the maximum number of tasks per job was first put in place. This simple solution eventually had to evolve to support more limits—on the number of jobs submitted per user, on the number of jobs that are initialized and expanded in the JobTracker's memory at any time, on the number of tasks that any job might legally request, and on the number of concurrent tasks that any job can run.

Management of Completed Jobs

The JobTracker would also remember completed jobs so that users could learn about their status once the jobs finished. Initially, completed jobs would have a memory footprint similar to that of any other running job. Completed jobs are, by definition, unbounded as time progresses. To address this issue, the JobTracker was modified to start remembering only partial but critical information about completed jobs, such as job status and counters, thereby minimizing the heap footprint per completed job. Even after this, with ever-increasing completed jobs, the JobTracker couldn't cope after sufficient time elapsed. To address this issue, the straightforward solution of remembering only the last *N* jobs per user was deployed. This created still more challenges: Users with a very high job-churn rate would eventually run into situations where they could not get information about recently submitted jobs. Further, the solution was a per-user limit, so given enough users; the JobTracker would eventually exhaust its heap anyway.

The ultimate state-of-the-art solution for managing this issue was to change the JobTracker to *not* remember *any* completed jobs at all, but instead redirect requests about completed jobs to a special server called the JobHistoryServer. This server offloaded the responsibility of serving web requests about completed jobs away from the JobTracker. To handle RPC requests in flight about completed jobs, the JobTracker would also persist some of the completed job information on the local or a remote file system; this responsibility of RPCs would also eventually transition to the JobHistoryServer in Hadoop 2.x releases.

Central Scheduler

When HOD was abandoned, the central scheduler that worked in unison with a traditional resource manager also went away. Trying to integrate existing schedulers with the newly proposed JobTracker-based architecture was a nonstarter due to the engineering challenges involved. It was proposed to extend the JobTracker itself to support queues of jobs. Users would interact with the queues, which are configured appropriately. In the HOD setting, nodes would be statically assigned to a queue—but that led to utilization issues across queues. In the newer architecture, nodes are no longer assigned statically. Instead, **slots** available on a node are dynamically allocated to jobs in queues, thereby also increasing the granularity of the scheduling from nodes to slots.

To facilitate innovations in the scheduling algorithm, an abstraction was put in place. Soon, several implementations came about. Yahoo! implemented and deployed the Capacity scheduler, which focused on throughput, while an alternative called the Fair scheduler also emerged, focusing on fairness.

Scheduling was done on every node's heartbeat: The scheduler would look at the free capacity on this node, look at the jobs that need resources, and schedule a task accordingly. Several dimensions were taken into account while making this scheduling decision—scheduler-specific policies such as capacity, fairness, and, more importantly, per-job locality preferences. Eventually, this "one task per heartbeat" approach was changed to start allocating multiple tasks per heartbeat to improve scheduling latencies and utilization.

The Capacity scheduler is based on allocating capacities to a flat list of queues and to users within those queues. Queues are defined following the internal organizational structure, and each queue is configured with a guaranteed capacity. Excess capacities from idle queues are distributed to queues that are in demand, even if they have already made use of their guaranteed capacity. Inside a queue, users can share resources but there is an overarching emphasis on job throughput, based on a FIFO algorithm. Limits are put in place to avoid single users taking over entire queues or the cluster.

Moving to centralized scheduling and granular resources resulted in massive utilization improvements. This brought more users, more growth to the so-called research clusters, and, in turn, more requirements. The ability to refresh queues at run time to affect capacity changes or to modify queue Access Control Lists (ACLs) was desired and subsequently implemented. With node-level isolation (described later), jobs were required to specify their memory requirements upfront, which warranted intelligent scheduling of high-memory jobs together with regular jobs; the scheduler accordingly acquired such functionality. This was done through reservation of slots on nodes for high-RAM jobs so that they do not become starved while regular jobs come in and take over capacity.

Recovery and Upgrades

The JobTracker was clearly a *single point of failure* for the whole cluster. Whenever a software bug surfaced or a planned upgrade needed to be done, the JobTracker would bring down the whole cluster. Anytime it needed to be restarted, even though the submitted job definitions were persistent in HDFS from the clients themselves, the state of running jobs would be completely lost. A feature was needed to let jobs survive JobTracker restarts. If a job was running at the time when the JobTracker restarted, along with the ability to not lose running work, the user would expect to get all information about previously completed tasks of this job transparently. To address this requirement, the JobTracker had to record and create persistent information about every completed task for every job onto highly available storage.

This feature was eventually implemented, but proved to be fraught with so many race conditions and corner cases that it eventually couldn't be pushed to production because of its instability. The complexity of the feature partly arose from the fact that JobTracker had to track and store too much information—first about the cluster state, and second about the scheduling state of each and every job. Referring to [Requirement 2] Serviceability, the shared MapReduce clusters in a way had regressed compared to HOD with respect to serviceability.

Isolation on Individual Nodes

Many times, tasks of user Map/Reduce applications would get extremely memory intensive. This could occur due to many reasons—for example, due to inadvertent bugs in the users' map or reduce code, because of incorrectly configured jobs that would unnecessarily process huge amounts of data, or because of mappers/reducers spawning children processes whose memory/CPU utilization couldn't be controlled by the task JVM. The last issue was very possible with the Hadoop streaming framework,

which enabled users to write their MapReduce code in an arbitrary language that was then run under separate children processes of task JVMs. When this happened, the user tasks would start to interfere with the proper execution of other processes on the node, including tasks of other jobs, even Hadoop daemons like the DataNode and the TaskTracker. In some instances, runaway user jobs would bring down multiple DataNodes on the cluster and cause HDFS downtime. Such uncontrolled tasks would cause nodes to become unusable for all purposes, leading to a need for a way to prevent such tasks from bringing down the node.

Such a situation wouldn't happen with HOD, as every user would essentially bring up his or her own Hadoop MapReduce cluster and each node belonged to only one user at any single point of time. Further, HOD would work with the underlying resource manager to set resource limits prior to the TaskTracker getting launched. This made the entire TaskTracker process chain—the daemon itself together, with the task JVMs *and* any processes further spawned by the tasks themselves—to be bounded. Whatever system needed to be designed to throttle runaway tasks had to mimic this exact functionality.

We considered multiple solutions—for example, the host operating system facilitating user limits that are both static and dynamic, putting caps on individual tasks, and setting a cumulative limit on the overall usage across all tasks. We eventually settled on the ability to control individual tasks by killing any process trees that surpass predetermined memory limits. The TaskTracker uses a default admin configuration or a per-job user-specified configuration, continuously monitors tasks' memory usage in regular cycles, and shoots down any process tree that has overrun the memory limits.

Distributed Cache was another feature that was neatly isolated by HOD. With HOD, any user's TaskTrackers would download remote files and maintain a local cache only for that user. With shared clusters, TaskTrackers were forced to maintain this cache across users. To help manage this distribution, the concepts of a public cache, private cache, and application cache were introduced. A public cache would include public files from all users, whereas a private cache would restrict itself to be per user. An application-level cache included resources that had to be deleted once a job finished. Further, with the TaskTracker concurrently managing several caches at once, several locking problems with regard to the Distributed Cache emerged, which required a minor redesign/reimplementation of this part of the TaskTracker.

Security

Along with enhancing resource isolation on individual nodes, HOD shielded security issues with multiple users by avoiding sharing of individual nodes altogether. Even for a single user, HOD would start the TaskTracker, which would then spawn the map and reduce tasks, resulting in all of them running as the user who had submitted the HOD job. With shared clusters, however, the tasks needed to be run as the job owner for security and accounting purposes, rather than as the user running the TaskTracker daemon itself.

We tried to avoid running the TaskTracker daemon as a privileged user (such as root) to solve this requirement. The TaskTracker would perform several operations

on behalf of users, and running it as a privileged user would leak a lot of surface area that was vulnerable to attacks. Ultimately, we solved this problem by creating a setuid executable called **taskcontroller** that would be owned by root but runnable only by the TaskTracker. The TaskTracker would launch this executable with appropriate commands when needed. It would first run as root, do some very basic operations, and then immediately drop privileges by using setuid POSIX call to run the remaining operations as the user. Because this was very platform-specific code, we implemented a TaskController Java abstraction and shipped an implementation for Linux called LinuxTaskController with all the platform-specific code written in C.

The directories and files used by the task also needed to have appropriate permissions. Many of these directories and files were created by the TaskTracker, but were used by the task. A few were written by the user code but then used or accessed by the daemon. For security reasons, the permissions needed to be very strict and readable/writable by only the user or the TaskTracker. This step was done by making the taskcontroller first change the permissions from the TaskTracker to the user, and then letting the task run. Any files that needed to be read by the TaskTracker after the task finished had to have been created with appropriate permissions by the tasks.

Authentication and Access Control

As Hadoop managed more tenants, diverse use-cases, and raw data, its requirements for isolation became more stringent. Unfortunately, the system lacked strong, scalable authentication and an authorization model—a critical feature for multitenant clusters. This capability was added and backported to multiple versions of Hadoop.

A user can submit jobs to one or more MapReduce clusters, but he or she should be authenticated by Kerberos or a delegation mechanism before job submission. A user can disconnect after job submission and then reconnect to get the job status by using the same authentication mechanism. Once such an authenticated user sends requests to the JobTracker, it records all such accesses in an audit log that can be postprocessed for analyzing over time—thereby creating a kind of audit trail.

Tasks run as the user need credentials to securely talk to HDFS, too. For this to happen, the user needs to specify the list of HDFS clusters for a job at job submission either implicitly by input/output paths or explicitly. The job client then uses this list to reach HDFS and obtain credentials on users' behalf. Beyond HDFS, communication with the TaskTracker for both task heartbeats and shuffle by the reduce tasks is also secured through a JobToken-based authentication mechanism.

A mechanism was needed to control who can submit jobs to a specified queue. Jobs can be submitted to only those queues the user is authorized to use. For this purpose, administrators set up **Queue ACLs** before the cluster is initialized. Administrators can dynamically change a queue's ACL to allow a specific user or group to access it at run time. Specific users and groups, called the cluster administrators and queue administrators, are able to manage the ACL on the queue as well to access or modify any job in the queue.

On top of queue-level ACLs, users are allowed to access or modify only their own MapReduce jobs or jobs to which others have given them access via **Job ACLs**. A Job

ACL governs two types of job operations: viewing a job and modifying a job. The web UI also shows information only about jobs run by the current user or about those jobs that are explicitly given access to via Job ACLs.

As one can see, MapReduce clusters acquired a lot of security features over time to manage more tenants on the same shared hardware. This [Requirement 6] Secure and Auditable Operation must be preserved in YARN.

- **[Requirement 6] Secure and Auditable Operation**

  The next-generation compute platform should continue to enable secure and auditable usage of cluster resources.

Miscellaneous Cluster Management Features

So far, we have described in great detail the evolution of the central JobTracker daemon and the individual nodes. In addition to those, HOD made use of a few other useful features in the underlying resource manager such as addition and decommissioning of nodes that needed to be reimplemented in the JobTracker to facilitate cluster management. Torque also exposed a functionality to run an arbitrary program that could dynamically recognize any issues with specific nodes. To replace this functionality, TaskTrackers would run a similar **health-check** script every so often and figure out if a node had turned bad. This information would eventually reach the JobTracker, which would in turn remove this node from scheduling. In addition to taking nodes offline after observing their (poor) health status, heuristics were implemented to track task failures on each node over time and to blacklist any nodes that failed to complete a greater-than-mean number of tasks across jobs.

Evolution of the MapReduce Framework

In addition to the changes in the underlying resource management, the MapReduce framework itself went through many changes. New MapReduce APIs were introduced in an attempt to fill some gaps in the old APIs, the algorithm for running speculative duplicate JVMs to work around slow tasks went through several iterations, and new features like reusing JVMs across tasks for performance were introduced. As the MapReduce framework was tied to the cluster management layer, this evolution would eventually prove to be difficult.

Issues with Shared MapReduce Clusters

Issues with the shared MapReduce clusters developed over time.

Scalability Bottlenecks

As mentioned earlier, 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 and job throughput increased dramatically, but issues with memory management and coarse-grained locking to support many of the features added to the JobTracker became sources of significant

scalability bottlenecks. Scaling the JobTracker to clusters containing more than about 4000 nodes would prove to be extremely difficult.

Part of the problem arose from the fact that the JobTracker was keeping in memory data from user jobs that could potentially be unbounded. Despite the innumerable limits that were put in place, the JobTracker would eventually run into some other part of the data structure that wasn't limited. For example, a user job might generate so many counters (which were then not limited) that TaskTrackers would spend all their time uploading those counters. The JobTracker's RPCs would then slow down to a grinding halt, TaskTrackers would get lost, resulting in a vicious circle that ended only with a downtime and a long wild goose chase for the offending application.

This problem would eventually lead to one of the bigger design points of YARN—to not load any user data in the system daemons to the greatest extent possible.

The JobTracker could logically be extended to support larger clusters and heterogeneous frameworks, if only with significant engineering investments. Heartbeat latency could be as high as 200 ms in large clusters, leading to node heartbeat intervals of as much as 40 seconds due to coarse-grained locking of its internal data structures. This problem could be improved with carefully designed fine-grained locking. The internal data structures in the JobTracker were often inefficient but they could be redesigned to occupy less memory. Many of the functions of the JobTracker could also be offloaded to separate, multitenant daemons. For example, serving the status of historical jobs could be—and eventually was—offloaded to the separate service JobHistoryServer. In other words, evolution could ideally continue by iterating on the existing code.

Although logical in theory, this scheme proved infeasible in practice. Changes to the JobTracker had become extremely difficult to validate. The continuous push for ill-thought-out features had produced a working, scalable, but very fragile system. It was time to go back to the drawing board for a complete overhaul. Scalability targets also anticipated clusters of 6000 machines running 100,000 concurrent tasks from 10,000 concurrent jobs, and there was no way the JobTracker could support such a massive scale without a major rewrite.

Reliability and Availability

While the move to shared clusters improved utilization and locality compared to HOD, it also brought concerns for serviceability and availability into sharp focus. Instead of losing a single workflow, a JobTracker failure caused an outage that would lose all of the running jobs in a cluster and require users to manually resubmit and recover their workflows. Upgrading a cluster by deploying a new version of Hadoop in a shared cluster was a rather common event and demanded very careful planning. To fix a bug in the MapReduce implementation, operators would necessarily schedule a cluster downtime, shut down the cluster, deploy the new binaries, validate the upgrade, and then admit new jobs. Any downtime created a backlog in the processing pipelines; when the jobs were eventually resubmitted, they would put a significant strain on the JobTracker. Restarts sometimes involved manually killing users' jobs until the cluster recovered.

Operating a large, multitenant Hadoop cluster is hard. While fault tolerance is a core design principle, the surface exposed to user applications is vast. Given the various availability issues exposed by the single point of failure, it was critical to continuously monitor workloads in the cluster for offending jobs. All of these concerns may be grouped under the need for [Requirement 7] Reliability and Availability.

- **[Requirement 7] Reliability and Availability**

  The next-generation compute platform should have a very reliable user interaction and support high availability.

Abuse of the MapReduce Programming Model

While MapReduce supports a wide range of use-cases, it is not the ideal model for all large-scale computations. For example, many machine learning programs require multiple iterations over a data set to converge to a result. If one composes this flow as a sequence of MapReduce jobs, the scheduling overhead will significantly delay the result. Similarly, many graph algorithms are better expressed using a bulk-synchronous parallel model (BSP) with message passing to communicate between vertices, rather than the heavy, all-to-all communication barrier in a fault-tolerant, large-scale MapReduce job. This mismatch became an impediment to users' productivity, but the MapReduce-centricity in Hadoop allowed no other alternative programming model.

The evolution of the software wired the intricacies of MapReduce so deeply into the platform that it took a multiple months' effort to introduce job-level setup and cleanup tasks, let alone an alternative programming model. Users who were in dire need of such alternative models would write MapReduce programs that would spawn their custom implementations—for example, for a farm of web servers. To the central scheduler, they appeared as a collection of map-only jobs with radically different resource curves, causing poor utilization, potentially resource deadlocks, and instability. If YARN were to be the next-generation platform, it must declare a truce with its users and provide explicit [Requirement 8] Support for Programming Model Diversity.

- **[Requirement 8] Support for Programming Model Diversity**

  The next-generation compute platform should enable diverse programming models and evolve beyond just being MapReduce-centric.

Resource Model

Beyond their mismatch with emerging framework requirements, the typed slots on the TaskTrackers harmed utilization. While the separation between map and reduce capacity on individual nodes (and hence the cluster) prevented cross-task-type deadlocks, it also caused bottleneck resources.

The overlap between the map and reduce stages is configured by the user for each submitted job. Starting reduce tasks later increases cluster throughput, while starting them earlier in a job's execution reduces its latency. The number of map and reduce slots are fixed by the cluster administrators, so unused map capacity can't be used to spawn reduce tasks, and vice versa. Because the two task types can potentially (and more often than not do) complete at different rates, no configuration will ever be

perfectly ideal. When either slot type becomes completely utilized, the JobTracker is forced to apply back-pressure to job initialization despite the presence of available slots of the other type. Nonstatic definition of resources on individual nodes complicates scheduling, but it also empowers the scheduler to pack the cluster more tightly.

Further, the definition of slots was purely based on jobs' memory requirements, as memory was the scarcest resource for much of this time. Hardware keeps evolving, however, and there are now many sites where CPU has become the most scarce resource, with memory being available in abundance, and the concept of slots doesn't easily accommodate this conundrum of scheduling multiple resources. This highlights the need for a [Requirement 9] Flexible Resource Model.

- **[Requirement 9] Flexible Resource Model**

  The next-generation compute platform should enable dynamic resource configurations on individual nodes and a flexible resource model.

Management of User Logs

The handling of user logs generated by applications had been one of the biggest selling points of HOD, but it turned into a pain point for shared MapReduce installations. User logs were typically left on individual nodes by the TaskTracker daemon after they were truncated, but only for a specific amount of time. If individual nodes died or were taken offline, their logs wouldn't be available at all. Runaway tasks could also fill up disks with useless logs, and there was no way to shield other tasks or the system daemons from such bad tasks.

Agility

By conflating the platform responsible for arbitrating resource usage with the framework expressing that program, one is forced to evolve both structures simultaneously. While cluster administrators try to improve the allocation efficiency of the platform, it is the users' responsibility to help incorporate framework changes into the new structure. Thus, upgrading a cluster should not require users to halt, validate, and restore their pipelines. But the exact opposite thing happened with shared MapReduce clusters: While updates typically required no more than recompilation, users' assumptions about internal framework details or developers' assumptions about users' programs occasionally created incompatibilities, wasting more software development cycles.

As stated earlier, HOD was much better at supporting this agility of user applications. [Requirement 2] Serviceability covered this need for the next-generation compute platform to enable evolution of cluster software completely decoupled from users' applications.

Phase 3: Emergence of YARN

The JobTracker would ideally require a complete rewrite to fix the majority of the scaling issues. Even if it were successful, however, this rewrite would not necessarily

resolve the coupling between platform and user code, nor would it address users' appetite for non-MapReduce programming models or the dependency between careful admission control and JobTracker scalability. Absent a significant redesign, cluster availability would continue to be tied to the stability of the whole system.

Building on lessons learned by evolving Apache Hadoop MapReduce, YARN was designed to address the specific requirements stated so far (i.e., Requirement 1 through Requirement 9). However, the massive installed base of MapReduce applications, the ecosystem of related projects, the well-worn deployment practice, and a tight schedule could not tolerate a radical new user interface. Consequently, the new architecture and the corresponding implementation reused as much code from the existing framework as possible, behaved in familiar patterns, and exposed the same interfaces for the existing MapReduce users. This led to the final requirement for the YARN redesign: [Requirement 10] Backward Compatibility.

- **[Requirement 10] Backward Compatibility**

  The next-generation compute platform should maintain complete backward compatibility of existing MapReduce applications.

To summarize the requirements for YARN, we need the following features:

- **[Requirement 1] Scalability**: The next-generation compute platform should scale horizontally to tens of thousands of nodes and concurrent applications.
- **[Requirement 2] Serviceability**: The next-generation compute platform should enable evolution of cluster software to be completely decoupled from users' applications.
- **[Requirement 3] Multitenancy**: The next-generation compute platform should support multiple tenants to coexist on the same cluster and enable fine-grained sharing of individual nodes among different tenants.
- **[Requirement 4] Locality Awareness**: The next-generation compute platform should support locality awareness—moving computation to the data is a major win for many applications.
- **[Requirement 5] High Cluster Utilization**: The next-generation compute platform should enable high utilization of the underlying physical resources.
- **[Requirement 6] Secure and Auditable Operation**: The next-generation compute platform should continue to enable secure and auditable usage of cluster resources.
- **[Requirement 7] Reliability and Availability**: The next-generation compute platform should have a very reliable user interaction and support high availability.
- **[Requirement 8] Support for Programming Model Diversity**: The next-generation compute platform should enable diverse programming models and evolve beyond just being MapReduce-centric.

- **[Requirement 9] Flexible Resource Model**: The next-generation compute platform should enable dynamic resource configurations on individual nodes and a flexible resource model.
- **[Requirement 10] Backward Compatibility**: The next-generation compute platform should maintain completely backward compatibility of existing Map-Reduce applications.

Conclusion

That concludes our coverage of the history and rationale for YARN. We hope that it gives readers a perspective on the various design and architectural decisions that will appear and reappear in the remainder of this book. It should also give an insight into the evolutionary process of YARN; every major decision in YARN is backed up by a sound, if sometimes gory history.

2

Apache Hadoop YARN Install Quick Start

Apache Hadoop presents the user with a vast ecosystem of tools and applications. For those familiar with Hadoop version 1, there are two core components; the Hadoop Distributed File System and the integrated MapReduce distributed processing engine. Hadoop YARN is the new replacement for the monolithic MapReduce component found in version 1. The scheduling and resource management have been separated from the management of MapReduce pipelines. While Hadoop version 2 with YARN still provides full MapReduce capability and backwards compatibility with version 1, it also opens the door to many other "application frameworks" that are not based on MapReduce processing.

The acronym YARN is short for "Yet Another Resource Negotiator," which is a good description of what YARN actually does. Fundamentally, YARN is a resource scheduler designed to work on existing and new Hadoop clusters. The seemingly trivial split of resource scheduling from the MapReduce data flow opens up a whole new range of possibilities for Hadoop and Big Data processing. A separate scheduler allows for better utilization and scalability of the cluster, while simultaneously providing a platform for other non-MapReduce applications to take advantage of the Hadoop Distributed File System and run-time environment. A more detailed discussion of the new Hadoop YARN capabilities can be found in Chapter 3, "Apache Hadoop YARN Core Concepts."

From a larger vantage point, YARN can be viewed as a cluster-wide Operating System that provides the essential services for applications to take advantage of a large dynamic and parallel resource infrastructure. Applications written in any language can now take advantage of the combined Hadoop compute and storage assets within any size cluster.

Although motivated by the needs of large clusters, YARN is capable of running on a single cluster node or desktop machine. The instructions in this chapter will allow you to install and explore Apache Hadoop version 2 with YARN on a single machine.

Getting Started

A production Apache Hadoop system can take time to set up properly and is not necessary to start experimenting with many of the YARN concepts and attributes. This chapter provides a quick start guide to installing Hadoop version Hadoop 2.2.0 on a single machine (workstation, server, or a laptop).

A more complete description of other installation options, such as those required by a production cluster setup, is given in Chapter 5, "Installing Apache Hadoop YARN." Before we begin with the quick start, we will mention a few background details that will help with installation. These items include rudimentary knowledge of Linux, package installation, and basic system administration commands.

A basic Apache Hadoop version 2 system has two core components:

- The Hadoop Distributed File System (HDFS) for storing data
- Hadoop YARN for implementing applications to process data

Other Apache Hadoop components, such as Pig and Hive, can be added after the two core components are installed and operating properly.

Steps to Configure a Single-Node YARN Cluster

The following type of installation is often referred to as "pseudo-distributed" because it mimics some of the functionality of a distributed Hadoop cluster. A single machine is, of course, not practical for any production use, nor is it parallel. A small-scale Hadoop installation can provide a simple method for learning Hadoop basics, however.

The recommended *minimal* installation hardware is a dual-core processor with 2 GB of RAM and 2 GB of available hard drive space. The system will need a recent Linux distribution with Java installed (e.g., Red Hat Enterprise Linux or rebuilds, Fedora, Suse Linux Enterprise, OpenSuse, Ubuntu). Red Hat Enterprise Linux 6.3 is used for this installation example. A bash shell environment is also assumed. The first step is to download Apache Hadoop.

Note that the following commands and files are available for download from the book repository; see Appendix A for details.

Step 1: Download Apache Hadoop

Download the latest distribution from the Hadoop website (http://hadoop.apache.org/). For example, as root do the following:

```
# cd /root
# wget http://mirrors.ibiblio.org/apache/hadoop/common/hadoop-2.2.0/hadoop-
➥2.2.0.tar.gz
```

Next create and extract the package in `/opt/yarn`:

```
# mkdir -p /opt/yarn
# cd /opt/yarn
# tar xvzf /root/hadoop-2.2.0.tar.gz
```

Step 2: Set JAVA_HOME

For Hadoop 2, the recommended version of Java can be found at http://wiki.apache.org/hadoop/HadoopJavaVersions. In general, a Java Development Kit 1.6 (or greater) should work. For this install, we will use Open Java 1.6.0_24, which is part of Red Hat Enterprise Linux 6.3. Make sure you have a working Java JDK installed; in this case, it is the Java-1.6.0-openjdk RPM. To include `JAVA_HOME` for all bash users (other shells must be set in a similar fashion), make an entry in `/etc/profile.d` as follows:

```
# echo "export JAVA_HOME=/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0.x86_64/" > /etc/
➥profile.d/java.sh
```

To make sure `JAVA_HOME` is defined for this session, source the new script:

```
# source /etc/profile.d/java.sh
```

Step 3: Create Users and Groups

It is best to run the various daemons with separate accounts. Three accounts (yarn, hdfs, mapred) in the group hadoop can be created as follows:

```
# groupadd hadoop
# useradd -g hadoop yarn
# useradd -g hadoop hdfs
# useradd -g hadoop mapred
```

Step 4: Make Data and Log Directories

Hadoop needs various data and log directories with various permissions. Enter the following lines to create these directories:

```
# mkdir -p /var/data/hadoop/hdfs/nn
# mkdir -p /var/data/hadoop/hdfs/snn
# mkdir -p /var/data/hadoop/hdfs/dn
# chown hdfs:hadoop /var/data/hadoop/hdfs -R
# mkdir -p /var/log/hadoop/yarn
# chown yarn:hadoop /var/log/hadoop/yarn -R
```

Next, move to the YARN installation root and create the log directory and set the owner and group as follows:

```
# cd /opt/yarn/hadoop-2.2.0
# mkdir logs
# chmod g+w logs
# chown yarn:hadoop . -R
```

Step 5: Configure core-site.xml

From the base of the Hadoop installation path (e.g., `/opt/yarn/hadoop-2.2.0`), edit the `etc/hadoop/core-site.xml` file. The original installed file will have no entries other than the `<configuration> </configuration>` tags. Two properties need to be set. The first is the `fs.default.name` property, which sets the host and request port name for the NameNode (metadata server for HDFS). The second is `hadoop.http.staticuser.user`, which will set the default user name to hdfs. Copy the following lines to the Hadoop `etc/hadoop/core-site.xml` file and remove the original empty `<configuration> </configuration>` tags.

```
<configuration>
       <property>
              <name>fs.default.name</name>
              <value>hdfs://localhost:9000</value>
       </property>
       <property>
              <name>hadoop.http.staticuser.user</name>
              <value>hdfs</value>
       </property>
</configuration>
```

Step 6: Configure hdfs-site.xml

From the base of the Hadoop installation path, edit the `etc/hadoop/hdfs-site.xml` file. In the single-node pseudo-distributed mode, we don't need or want the HDFS to replicate file blocks. By default, HDFS keeps three copies of each file in the file system for redundancy. There is no need for replication on a single machine; thus the value of `dfs.replication` will be set to 1.

In `hdfs-site.xml`, we specify the NameNode, Secondary NameNode, and DataNode data directories that we created in Step 4. These are the directories used by the various components of HDFS to store data. Copy the following lines into Hadoop `etc/hadoop/hdfs-site.xml` and remove the original empty `<configuration> </configuration>` tags.

```
<configuration>
 <property>
   <name>dfs.replication</name>
   <value>1</value>
 </property>
 <property>
   <name>dfs.namenode.name.dir</name>
   <value>file:/var/data/hadoop/hdfs/nn</value>
 </property>
 <property>
   <name>fs.checkpoint.dir</name>
   <value>file:/var/data/hadoop/hdfs/snn</value>
 </property>
```

```
 <property>
   <name>fs.checkpoint.edits.dir</name>
   <value>file:/var/data/hadoop/hdfs/snn</value>
 </property>
 <property>
   <name>dfs.datanode.data.dir</name>
   <value>file:/var/data/hadoop/hdfs/dn</value>
 </property>
</configuration>
```

Step 7: Configure mapred-site.xml

From the base of the Hadoop installation, edit the etc/hadoop/mapred-site.xml file. A new configuration option for Hadoop 2 is the capability to specify a framework name for MapReduce, setting the mapreduce.framework.name property. In this install, we will use the value of "yarn" to tell MapReduce that it will run as a YARN application. First, copy the template file to the mapred-site.xml.

```
# cp mapred-site.xml.template mapred-site.xml
```

Next, copy the following lines into Hadoop etc/hadoop/mapred-site.xml file and remove the original empty <configuration> </configuration> tags.

```
<configuration>
<property>
   <name>mapreduce.framework.name</name>
   <value>yarn</value>
 </property>
</configuration>
```

Step 8: Configure yarn-site.xml

From the base of the Hadoop installation, edit the etc/hadoop/yarn-site.xml file. The yarn.nodemanager.aux-services property tells NodeManagers that there will be an auxiliary service called mapreduce.shuffle that they need to implement. After we tell the NodeManagers to implement that service, we give it a class name as the means to implement that service. This particular configuration tells MapReduce how to do its shuffle. Because NodeManagers won't shuffle data for a non-MapReduce job by default, we need to configure such a service for MapReduce. Copy the following lines to the Hadoop etc/hadoop/yarn-site.xml file and remove the original empty <configuration> </configuration> tags.

```
<configuration>
<property>
   <name>yarn.nodemanager.aux-services</name>
   <value>mapreduce_shuffle</value>
 </property>
 <property>
   <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
```

```
   <value>org.apache.hadoop.mapred.ShuffleHandler</value>
 </property>
</configuration>
```

Step 9: Modify Java Heap Sizes

The Hadoop installation uses several environment variables that determine the heap sizes for each Hadoop process. These are defined in the `etc/hadoop/*-env.sh` files used by Hadoop. The default for most of the processes is a 1 GB heap size; because we're running on a workstation that will probably have limited resources compared to a standard server, however, we need to adjust the heap size settings. The values that follow are adequate for a small workstation or server.

Edit the `etc/hadoop/hadoop-env.sh` file to reflect the following (don't forget to remove the "#" at the beginning of the line):

```
HADOOP_HEAPSIZE="500"
HADOOP_NAMENODE_INIT_HEAPSIZE="500"
```

Next, edit `mapred-env.sh` to reflect the following:

```
HADOOP_JOB_HISTORYSERVER_HEAPSIZE=250
```

Finally, edit `yarn-env.sh` to reflect the following:

```
JAVA_HEAP_MAX=-Xmx500m
```

The following line will need to be added to `yarn-env.sh`:

```
YARN_HEAPSIZE=500
```

Step 10: Format HDFS

For the HDFS NameNode to start, it needs to initialize the directory where it will hold its data. The NameNode service tracks all the metadata for the file system. The format process will use the value assigned to `dfs.namenode.name.dir` in `etc/hadoop/hdfs-site.xml` earlier (i.e., `/var/data/hadoop/hdfs/nn`). Formatting destroys everything in the directory and sets up a new file system. Format the NameNode directory as the HDFS superuser, which is typically the "hdfs" user account.

From the base of the Hadoop distribution, change directories to the "`bin`" directory and execute the following commands:

```
# su - hdfs
$ cd /opt/yarn/hadoop-2.2.0/bin
$ ./hdfs namenode -format
```

If the command worked, you should see the following near the end of a long list of messages:

```
INFO common.Storage: Storage directory /var/data/hadoop/hdfs/nn has been
➥successfully formatted.
```

Step 11: Start the HDFS Services

Once formatting is successful, the HDFS services must be started. There is one service for the NameNode (metadata server), a single DataNode (where the actual data is stored), and the SecondaryNameNode (checkpoint data for the NameNode). The Hadoop distribution includes scripts that set up these commands as well as name other values such as PID directories, log directories, and other standard process configurations. From the `bin` directory in Step 10, execute the following as user hdfs:

```
$ cd ../sbin
$ ./hadoop-daemon.sh start namenode
```

The command should show the following:

```
starting namenode, logging to /opt/yarn/hadoop-2.2.0/logs/hadoop-hdfs-namenode-
➥limulus.out
```

The secondarynamenode and datanode services can be started in the same way:

```
$ ./hadoop-daemon.sh start secondarynamenode
starting secondarynamenode, logging to /opt/yarn/hadoop-2.2.0/logs/hadoop-hdfs-
➥secondarynamenode-limulus.out
$ ./hadoop-daemon.sh start datanode
starting datanode, logging to /opt/yarn/hadoop-2.2.0/logs/hadoop-hdfs-datanode-
➥limulus.out
```

If the daemon started successfully, you should see responses that will point to the log file. (Note that the actual log file is appended with ".log," not ".out."). As a sanity check, issue a `jps` command to confirm that all the services are running. The actual PID (Java Process ID) values will be different than shown in this listing:

```
$ jps
15140 SecondaryNameNode
15015 NameNode
15335 Jps
15214 DataNode
```

If the process did not start, it may be helpful to inspect the log files. For instance, examine the log file for the NameNode. (Note that the path is taken from the preceding command.)

```
vi /opt/yarn/hadoop-2.2.0/logs/hadoop-hdfs-namenode-limulus.log
```

All Hadoop services can be stopped using the `hadoop-daemon.sh` script. For example, to stop the datanode service, enter the following (as user hdfs in the `/opt/yarn/hadoop-2.2.0/sbin` directory):

```
$ ./hadoop-daemon.sh stop datanode
```

The same can be done for the NameNode and SecondaryNameNode.

Step 12: Start YARN Services

As with HDFS services, the YARN services need to be started. One ResourceManager and one NodeManager must be started as user yarn (exiting from user hdfs first):

```
$ exit
logout
# su - yarn
$ cd /opt/yarn/hadoop-2.2.0/sbin
$ ./yarn-daemon.sh start resourcemanager
starting resourcemanager, logging to /opt/yarn/hadoop-2.2.0/logs/yarn-yarn-
➥resourcemanager-limulus.out
$ ./yarn-daemon.sh start nodemanager
starting nodemanager, logging to /opt/yarn/hadoop-2.2.0/logs/yarn-yarn-
➥nodemanager-limulus.out
```

As when the HDFS daemons were started in Step 1, the status of the running daemons is sent to their respective log files. To check whether the services are running, issue a `jps` command. The following shows all the services necessary to run YARN on a single server:

```
$ jps
15933 Jps
15567 ResourceManager
15785 NodeManager
```

If there are missing services, check the log file for the specific service. Similar to the case with HDFS services, the services can be stopped by issuing a stop argument to the daemon script:

```
./yarn-daemon.sh stop nodemanager
```

Step 13: Verify the Running Services Using the Web Interface

Both HDFS and the YARN ResourceManager have a web interface. These interfaces are a convenient way to browse many of the aspects of your Hadoop installation. To monitor HDFS, enter the following (or use your favorite web browser):

```
$ firefox  http://localhost:50070
```

Connecting to port 50070 will bring up a web interface similar to Figure 2.1.

A web interface for the ResourceManager can be viewed by entering the following:

```
$ firefox http://localhost:8088
```

A webpage similar to that shown in Figure 2.2 will be displayed.

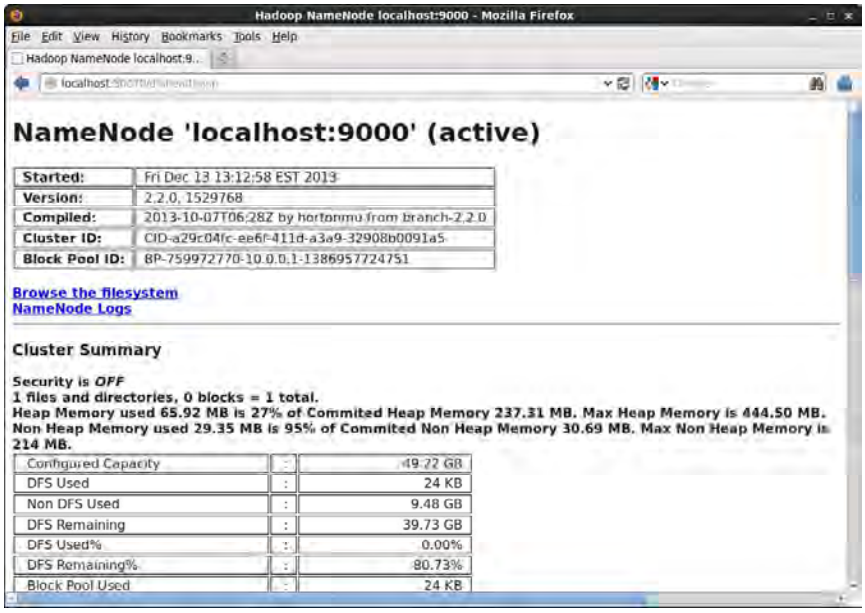


Figure 2.1 Webpage for HDFS file system

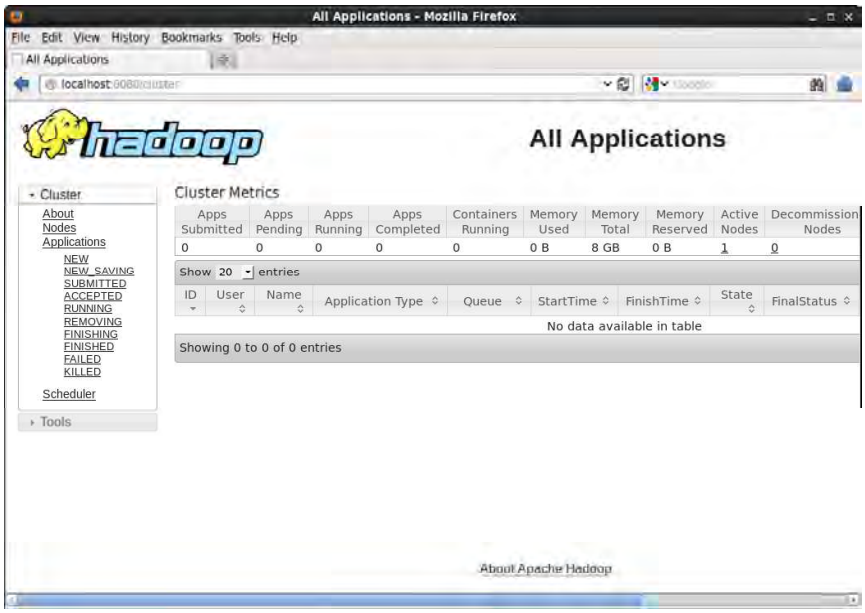


Figure 2.2 Webpage for YARN ResourceManager

Run Sample MapReduce Examples

To test your installation, run the sample "pi" program that calculates the value of pi using a quasi-Monte Carlo method and MapReduce. Change to user hdfs and run the following:

```
# su - hdfs
$ cd /opt/yarn/hadoop-2.2.0/bin
$ export YARN_EXAMPLES=/opt/yarn/hadoop-2.2.0/share/hadoop/mapreduce
$ ./yarn jar $YARN_EXAMPLES/hadoop-mapreduce-examples-2.2.0.jar pi 16 1000
```

If the program worked correctly, the following should be displayed at the end of the program output stream:

```
Estimated value of Pi is 3.14250000000000000000
```

This example submits a MapReduce job to YARN from the included samples in the `share/hadoop/mapreduce` directory. The master JAR file contains several sample applications to test your YARN installation. After you submit the job, its progress can be viewed by updating the ResourceManager webpage shown in Figure 2.2.

You can get a full list of examples by entering the following:

```
./yarn jar $YARN_EXAMPLES/hadoop-mapreduce-examples-2.2.0.jar
```

To see a list of options for each example, add the example name to this command. The following is a list of the included jobs in the examples JAR file.

- **aggregatewordcount**: An Aggregate-based map/reduce program that counts the words in the input files.
- **aggregatewordhist:** An Aggregate-based map/reduce program that computes the histogram of the words in the input files.
- **bbp:** A map/reduce program that uses Bailey-Borwein-Plouffe to compute the exact digits of pi.
- **dbcount**: An example job that counts the pageview counts from a database.
- **distbbp**: A map/reduce program that uses a BBP-type formula to compute the exact bits of pi.
- **grep:** A map/reduce program that counts the matches to a regex in the input.
- **join:** A job that effects a join over sorted, equally partitioned data sets.
- **multifilewc**: A job that counts words from several files.
- **pentomino:** A map/reduce tile laying program to find solutions to pentomino problems.
- **pi:** A map/reduce program that estimates pi using a quasi-Monte Carlo method.
- **randomtextwriter:** A map/reduce program that writes 10 GB of random textual data per node.