Addison Wesley Data & Analytics Series

Apache Hadoop™ YARN

Moving beyond MapReduce and Batch Processing with Apache Hadoop™ 2

Arun C. Murthy | Vinod Kumar Vavilapalli

Doug Eadline, Joseph Niemiec, *and* Jeff Markham

Apache Hadoop™ YARN

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

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

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

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

Foreword by Raymie Stata

William Gibson was fond of saying: "The future is already here—it's just not very evenly distributed." Those of us who have been in the web search industry have had the privilege—and the curse—of living in the future of Big Data when it wasn't distributed at all. What did we learn? We learned to measure everything. We learned to experiment. We learned to mine signals out of unstructured data. We learned to drive business value through data science. And we learned that, to do these things, we needed a new data-processing platform fundamentally different from the business intelligence systems being developed at the time.

The future of Big Data is rapidly arriving for almost all industries. This is driven in part by widespread instrumentation of the physical world—vehicles, buildings, and even people are spitting out log streams not unlike the weblogs we know and love in cyberspace. Less obviously, digital records—such as digitized government records, digitized insurance policies, and digital medical records—are creating a trove of information not unlike the webpages crawled and parsed by search engines. It's no surprise, then, that the tools and techniques pioneered first in the world of web search are finding currency in more and more industries. And the leading such tool, of course, is Apache Hadoop.

But Hadoop is close to ten years old. Computing infrastructure has advanced significantly in this decade. If Hadoop was to maintain its relevance in the modern Big Data world, it needed to advance as well. YARN represents just the advancement needed to keep Hadoop relevant.

As described in the historical overview provided in this book, for the majority of Hadoop's existence, it supported a single computing paradigm: MapReduce. On the compute servers we had at the time, horizontal scaling—throwing more server nodes at a problem—was the only way the web search industry could hope to keep pace with the growth of the web. The MapReduce paradigm is particularly well suited for horizontal scaling, so it was the natural paradigm to keep investing in.

With faster networks, higher core counts, solid-state storage, and (especially) larger memories, new paradigms of parallel computing are becoming practical at large scales. YARN will allow Hadoop users to move beyond MapReduce and adopt these emerging paradigms. MapReduce will not go away—it's a good fit for many problems, and it still scales better than anything else currently developed. But, increasingly, MapReduce will be just one tool in a much larger tool chest—a tool chest named "YARN."

In short, the era of Big Data is just starting. Thanks to YARN, Hadoop will continue to play a pivotal role in Big Data processing across all industries. Given this, I was pleased to learn that YARN project founder Arun Murthy and project lead Vinod Kumar Vavilapalli have teamed up with Doug Eadline, Joseph Niemiec, and Jeff Markham to write a volume sharing the history and goals of the YARN project, describing how to deploy and operate YARN, and providing a tutorial on how to get the most out of it at the application level.

This book is a critically needed resource for the newly released Apache Hadoop 2.0, highlighting YARN as the significant breakthrough that broadens Hadoop beyond the MapReduce paradigm.

—*Raymie Stata, CEO of Altiscale*

Foreword by Paul Dix

No series on data and analytics would be complete without coverage of Hadoop and the different parts of the Hadoop ecosystem. Hadoop 2 introduced YARN, or "Yet Another Resource Negotiator," which represents a major change in the internals of how data processing works in Hadoop. With YARN, Hadoop has moved beyond the MapReduce paradigm to expose a framework for building applications for data processing at scale. MapReduce has become just an application implemented on the YARN framework. This book provides detailed coverage of how YARN works and explains how you can take advantage of it to work with data at scale in Hadoop outside of MapReduce.

No one is more qualified to bring this material to you than the authors of this book. They're the team at Hortonworks responsible for the creation and development of YARN. Arun, a co-founder of Hortonworks, has been working on Hadoop since its creation in 2006. Vinod has been contributing to the Apache Hadoop project full-time since mid-2007. Jeff and Joseph are solutions engineers with Hortonworks. Doug is the trainer for the popular Hadoop Fundamentals LiveLessons and has years of experience building Hadoop and clustered systems. Together, these authors bring a breadth of knowledge and experience with Hadoop and YARN that can't be found elsewhere.

This book provides you with a brief history of Hadoop and MapReduce to set the stage for why YARN was a necessary next step in the evolution of the platform. You get a walk-through on installation and administration and then dive into the internals of YARN and the Capacity scheduler. You see how existing MapReduce applications now run as an applications framework on top of YARN. Finally, you learn how to implement your own YARN applications and look at some of the new YARN-based frameworks. This book gives you a comprehensive dive into the next generation Hadoop platform.

—*Paul Dix, Series Editor*

This page intentionally left blank

Preface

Apache Hadoop has a rich and long history. It's come a long way since its birth in the middle of the first decade of this millennium—from being merely an infrastructure component for a niche use-case (web search), it's now morphed into a compelling part of a modern data architecture for a very wide spectrum of the industry. Apache Hadoop owes its success to many factors: the community housed at the Apache Software Foundation; the timing (solving an important problem at the right time); the extensive early investment done by Yahoo! in funding its development, hardening, and large-scale production deployments; and the current state where it's been adopted by a broad ecosystem. In hindsight, its success is easy to rationalize.

On a personal level, Vinod and I have been privileged to be part of this journey from the very beginning. It's very rare to get an opportunity to make such a wide impact on the industry, and even rarer to do so in the slipstream of a great wave of a community developing software in the open—a community that allowed us to share our efforts, encouraged our good ideas, and weeded out the questionable ones. We are very proud to be part of an effort that is helping the industry understand, and unlock, a significant value from data.

YARN is an effort to usher Apache Hadoop into a new era—an era in which its initial impact is no longer a novelty and expectations are significantly higher, and growing. At Hortonworks, we strongly believe that at least half the world's data will be touched by Apache Hadoop. To those in the engine room, it has been evident, for at least half a decade now, that Apache Hadoop had to evolve beyond supporting MapReduce alone. As the industry pours all its data into Apache Hadoop HDFS, there is a real need to process that data in multiple ways: real-time event processing, human-interactive SQL queries, batch processing, machine learning, and many others. Apache Hadoop 1.0 was severely limiting; one could store data in many forms in HDFS, but MapReduce was the only algorithm you could use to natively process that data.

YARN was our way to begin to solve that multidimensional requirement natively in Apache Hadoop, thereby transforming the core of Apache Hadoop from a one-trick "batch store/process" system into a true multiuse platform. The crux was the recognition that Apache Hadoop MapReduce had two facets: (1) a core resource manager, which included scheduling, workload management, and fault tolerance; and (2) a user-facing MapReduce framework that provided a simplified interface to the end-user that hid the complexity of dealing with a scalable, distributed system. In particular, the MapReduce framework freed the user from having to deal with gritty details of fault tolerance, scalability, and other issues. YARN is just realization of this simple idea. With YARN, we have successfully relegated MapReduce to the role of merely one of the options to process data in Hadoop, and it now sits side-by-side by other frameworks such as Apache Storm (real-time event processing), Apache Tez (interactive query backed), Apache Spark (in-memory machine learning), and many more.

Distributed systems are hard; in particular, dealing with their failures is hard. YARN enables programmers to design and implement distributed *frameworks* while sharing a common set of resources and data. While YARN lets application developers focus on their business logic by automatically taking care of thorny problems like resource arbitration, isolation, cluster health, and fault monitoring, it also needs applications to act on the corresponding signals from YARN as they see fit. YARN makes the effort of building such systems significantly simpler by dealing with many issues with which a framework developer would be confronted; the framework developer, at the same time, still has to deal with the consequences on the framework in a framework-specific manner.

While the power of YARN is easily comprehensible, the ability to exploit that power requires the user to understand the intricacies of building such a system in conjunction with YARN. This book aims to reconcile that dichotomy.

The YARN project and the Apache YARN community have come a long way since their beginning. Increasingly more applications are moving to run natively under YARN and, therefore, are helping users process data in myriad ways. We hope that with the knowledge gleaned from this book, the reader can help feed that cycle of enablement so that individuals and organizations alike can take full advantage of the data revolution with the applications of their choice.

—*Arun C. Murthy*

Focus of the Book

This book is intended to provide detailed coverage of Apache Hadoop YARN's goals, its design and architecture and how it expands the Apache Hadoop ecosystem to take advantage of data at scale beyond MapReduce. It primarily focuses on installation and administration of YARN clusters, on helping users with YARN application development and new frameworks that run on top of YARN beyond MapReduce.

Please note that this book is *not* intended to be an introduction to Apache Hadoop itself. We assume that the reader has a working knowledge of Hadoop version 1, writing applications on top of the Hadoop MapReduce framework, and the architecture and usage of the Hadoop Distributed FileSystem. Please see the book webpage (http://yarn-book.com) for a list of introductory resources. In future editions of this book, we hope to expand our material related to the MapReduce application framework itself and how users can design and code their own MapReduce applications.

Book Structure

In Chapter 1, "Apache Hadoop YARN: A Brief History and Rationale," we provide a historical account of why and how Apache Hadoop YARN came about. Chapter 2, "Apache Hadoop YARN Install Quick Start," gives you a quick-start guide for installing and exploring Apache Hadoop YARN on a single node. Chapter 3, "Apache Hadoop YARN Core Concepts," introduces YARN and explains how it expands Hadoop ecosystem. A functional overview of YARN components then appears in Chapter 4, "Functional Overview of YARN Components," to get the reader started.

Chapter 5, "Installing Apache Hadoop YARN," describes methods of installing YARN. It covers both a script-based manual installation as well as a GUI-based installation using Apache Ambari. We then cover information about administration of YARN clusters in Chapter 6, "Apache Hadoop YARN Administration."

A deep dive into YARN's architecture occurs in Chapter 7, "Apache Hadoop YARN Architecture Guide," which should give the reader an idea of the inner workings of YARN. We follow this discussion with an exposition of the Capacity scheduler in Chapter 8, "Capacity Scheduler in YARN."

Chapter 9, "MapReduce with Apache Hadoop YARN," describes how existing MapReduce-based applications can work on and take advantage of YARN. Chapter 10, "Apache Hadoop YARN Application Example," provides a detailed walk-through of how to build a YARN application by way of illustrating a working YARN application that creates a JBoss Application Server cluster. Chapter 11, "Using Apache Hadoop YARN Distributed-Shell," describes the usage and innards of distributed shell, the canonical example application that is built on top of and ships with YARN.

One of the most exciting aspects of YARN is its ability to support multiple programming models and application frameworks. We conclude with Chapter 12, "Apache Hadoop YARN Frameworks," a brief survey of emerging open-source frameworks that are being developed to run under YARN.

Appendices include Appendix A, "Supplemental Content and Code Downloads"; Appendix B, "YARN Installation Scripts"; Appendix C, "YARN Administration Scripts"; Appendix D, "Nagios Modules"; Appendix E, "Resources and Additional Information"; and Appendix F, "HDFS Quick Reference."

Book Conventions

Code is displayed in a `monospaced` font. Code lines that wrap because they are too long to fit on one line in this book are denoted with this symbol: ➥.

Additional Content and Accompanying Code

Please see Appendix A, " Supplemental Content and Code Downloads," for the location of the book webpage (http://yarn-book.com). All code and configuration files used in this book can be downloaded from this site. Check the website for new and updated content including "Description of Apache Hadoop YARN Configuration Properties" and "Apache Hadoop YARN Troubleshooting Tips."

Acknowledgments

We are very grateful for the following individuals who provided feedback and valuable assistance in crafting this book.

- Ron Lee, Platform Engineering Architect at Hortonworks Inc, for making this book happen, and without whose involvement this book wouldn't be where it is now.
- Jian He, Apache Hadoop YARN Committer and a member of the Hortonworks engineering team, for helping with reviews.
- Zhijie Shen, Apache Hadoop YARN Committer and a member of the Hortonworks engineering team, for helping with reviews.
- Omkar Vinit Joshi, Apache Hadoop YARN Committer, for some very thorough reviews of a number of chapters.
- Xuan Gong, a member of the Hortonworks engineering team, for helping with reviews.
- Christopher Gambino, for the target audience testing.
- David Hoyle at Hortonworks, for reading the draft.
- Ellis H. Wilson III, storage scientist, Department of Computer Science and Engineering, the Pennsylvania State University, for reading and reviewing the entire draft.

Arun C. Murthy

Apache Hadoop is a product of the fruits of the community at the Apache Software Foundation (ASF). The mantra of the ASF is "Community Over Code," based on the insight that successful communities are built to last, much more so than successful projects or code bases. Apache Hadoop is a shining example of this. Since its inception, many hundreds of people have contributed their time, interest and expertise—many are still around while others have moved on; the constant is the *community*. I'd like to take this opportunity to thank every one of the contributors; Hadoop wouldn't be what it is without your contributions. Contribution is not merely code; it's a bug report, an email on the user mailing list helping a journeywoman with a query, an edit of the Hadoop wiki, and so on.

I'd like to thank everyone at Yahoo! who supported Apache Hadoop from the beginning—there really isn't a need to elaborate further; it's crystal clear to everyone who understands the history and context of the project.

Apache Hadoop YARN began as a mere idea. Ideas are plentiful and transient, and have questionable value. YARN wouldn't be real but for the countless hours put in by hundreds of contributors; nor would it be real but for the initial team who believed in the idea, weeded out the bad parts, chiseled out the reasonable parts, and took ownership of it. Thank you, you know who you are.

Special thanks to the team behind the curtains at Hortonworks who were so instrumental in the production of this book; folks like Ron and Jim are the key architects of this effort. Also to my co-authors: Vinod, Joe, Doug, and Jeff; you guys are an amazing bunch. Vinod, in particular, is someone the world should pay even more attention to—he is a very special young man for a variety of reasons.

Everything in my life germinates from the support, patience, and love emanating from my family: mom, grandparents, my best friend and amazing wife, Manasa, and the three-year-old twinkle of my eye, Arjun. Thank you. Gratitude in particular to my granddad, the best man I have ever known and the moral yardstick I use to measure myself with—I miss you terribly now.

Cliché alert: last, not least, many thanks to you, the reader. Your time invested in reading this book and learning about Apache Hadoop and YARN is a very big compliment. Please do not hesitate to point out how we could have provided better return for your time.

Vinod Kumar Vavilapalli

Apache Hadoop YARN, and at a bigger level, Apache Hadoop itself, continues to be a healthy, community-driven, open-source project. It owes much of its success and adoption to the Apache Hadoop YARN and MapReduce communities. Many individuals and organizations spent a lot of time developing, testing, deploying and administering, supporting, documenting, evangelizing, and most of all, using Apache Hadoop YARN over the years. Here's a big thanks to all the volunteer contributors, users, testers, committers, and PMC members who have helped YARN to progress in every way possible. Without them, YARN wouldn't be where it is today, let alone this book. My involvement with the project is entirely accidental, and I pay my gratitude to lady luck for bestowing upon me the incredible opportunity of being able to contribute to such a once-in-a-decade project.

This book wouldn't have been possible without the herding efforts of Ron Lee, who pushed and prodded me and the other co-writers of this book at every stage. Thanks to Jeff Markham for getting the book off the ground and for his efforts in demonstrating the power of YARN in building a non-trivial YARN application and making it usable as a guide for instruction. Thanks to Doug Eadline for his persistent thrust toward a timely and usable release of the content. And thanks to Joseph Niemiec for jumping in late in the game but contributing with significant efforts.

Special thanks to my mentor, Hemanth Yamijala, for patiently helping me when my career had just started and for such great guidance. Thanks to my co-author,

mentor, team lead and friend, Arun C. Murthy, for taking me along on the ride that is Hadoop. Thanks to my beautiful and wonderful wife, Bhavana, for all her love, support, and not the least for patiently bearing with my single-threaded span of attention while I was writing the book. And finally, to my parents, who brought me into this beautiful world and for giving me such a wonderful life.

Doug Eadline

There are many people who have worked behind the scenes to make this book possible. First, I want to thank Ron Lee of Hortonworks: Without your hand on the tiller, this book would have surely sailed into some rough seas. Also, Joe Niemiec of Hortonworks, thanks for all the help and the 11th-hour efforts. To Debra Williams Cauley of Addison-Wesley, you are a good friend who makes the voyage easier; Namaste. Thanks to the other authors, particularly Vinod for helping me understand the big and little ideas behind YARN. I also cannot forget my support crew, Emily, Marlee, Carla, and Taylor—thanks for reminding me when I raise my eyebrows. And, finally, the biggest thank you to my wonderful wife, Maddy, for her support. Yes, it is done. Really.

Joseph Niemiec

A big thanks to my father, Jeffery Niemiec, for without him I would have never developed my passion for computers.

Jeff Markham

From my first introduction to YARN at Hortonworks in 2012 to now, I've come to realize that the only way organizations worldwide can use this game-changing software is because of the open-source community effort led by Arun Murthy and Vinod Vavilapalli. To lead the world-class Hortonworks engineers along with corporate and individual contributors means a lot of sausage making, cat herding, and a heavy dose of vision. Without all that, there wouldn't even be YARN. Thanks to both of you for leading a truly great engineering effort. Special thanks to Ron Lee for shepherding us all through this process, all outside of his day job. Most importantly, though, I owe a huge debt of gratitude to my wife, Yong, who wound up doing a lot of the heavy lifting for our relocation to Seoul while I fulfilled my obligations for this project. 사랑해요!

About the Authors

Arun C. Murthy has contributed to Apache Hadoop full time since the inception of the project in early 2006. He is a long-term Hadoop committer and a member of the Apache Hadoop Project Management Committee. Previously, he was the architect and lead of the Yahoo! Hadoop MapReduce development team and was ultimately responsible, on a technical level, for providing Hadoop MapReduce as a service for all of Yahoo!—currently running on nearly 50,000 machines! Arun is the founder and architect of Hortonworks Inc., a software company that is helping to accelerate the development and adoption of Apache Hadoop. Hortonworks was formed by the key architects and core Hadoop committers from the Yahoo! Hadoop software engineering team in June 2011. Funded by Yahoo! and Benchmark Capital, one of the preeminent technology investors, Hortonworks has as its goal ensuring that Apache Hadoop becomes the standard platform for storing, processing, managing, and analyzing Big Data. Arun lives in Silicon Valley.

Vinod Kumar Vavilapalli has been contributing to Apache Hadoop project full time since mid-2007. At Apache Software Foundation, he is a long-term Hadoop contributor, Hadoop committer, member of the Apache Hadoop Project Management Committee, and a Foundation member. Vinod is a MapReduce and YARN go-to guy at Hortonworks Inc. For more than five years, he has been working on Hadoop and still has fun doing it. He was involved in Hadoop on Demand, Hadoop 0.20, Capacity scheduler, Hadoop security, and MapReduce, and now is a lead developer and the project lead for Apache Hadoop YARN. Before joining Hortonworks, he was at Yahoo! working in the Grid team that made Hadoop what it is today, running at large scale—up to tens of thousands of nodes. Vinod loves reading books of all kinds, and is passionate about using computers to change the world for better, bit by bit. He has a bachelor's degree in computer science and engineering from the Indian Institute of Technology Roorkee. He lives in Silicon Valley and is reachable at twitter handle @tshooter.

Doug Eadline, PhD, began his career as a practitioner and a chronicler of the Linux cluster HPC revolution and now documents Big Data analytics. Starting with the first Beowulf how-to document, Doug has written hundreds of articles, white papers, and instructional documents covering virtually all aspects of HPC. Prior to starting and editing the popular ClusterMonkey.net website in 2005, he served as editor-in-chief for *ClusterWorld* magazine, and was senior HPC editor for *Linux Magazine*. He has practical hands-on experience in many aspects of HPC, including hardware and software design, benchmarking, storage, GPU, cloud computing, and parallel computing. Currently, he is a writer and consultant to the HPC industry and leader of the Limulus Personal Cluster Project (http://limulus.basement-supercomputing.com). He is also author of *Hadoop Fundamentals LiveLessons* and *Apache Hadoop YARN Fundamentals LiveLessons* videos from Addison-Wesley.

Joseph Niemiec is a Big Data solutions engineer whose focus is on designing Hadoop solutions for many *Fortune 1000* companies. In this position, Joseph has worked with customers to build multiple YARN applications, providing a unique perspective on moving customers beyond batch processing, and has worked on YARN development directly. An avid technologist, Joseph has been focused on technology innovations since 2001. His interest in data analytics originally started in game score optimization as a teenager and has shifted to helping customers uptake new technology innovations such as Hadoop and, most recently, building new data applications using YARN.

Jeff Markham is a solution engineer at Hortonworks Inc., the company promoting open-source Hadoop. Previously, he was with VMware, Red Hat, and IBM, helping companies build distributed applications with distributed data. He has written articles on Java application development and has spoken at several conferences and to Hadoop user groups. Jeff is a contributor to Apache Pig and Apache HDFS.

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.

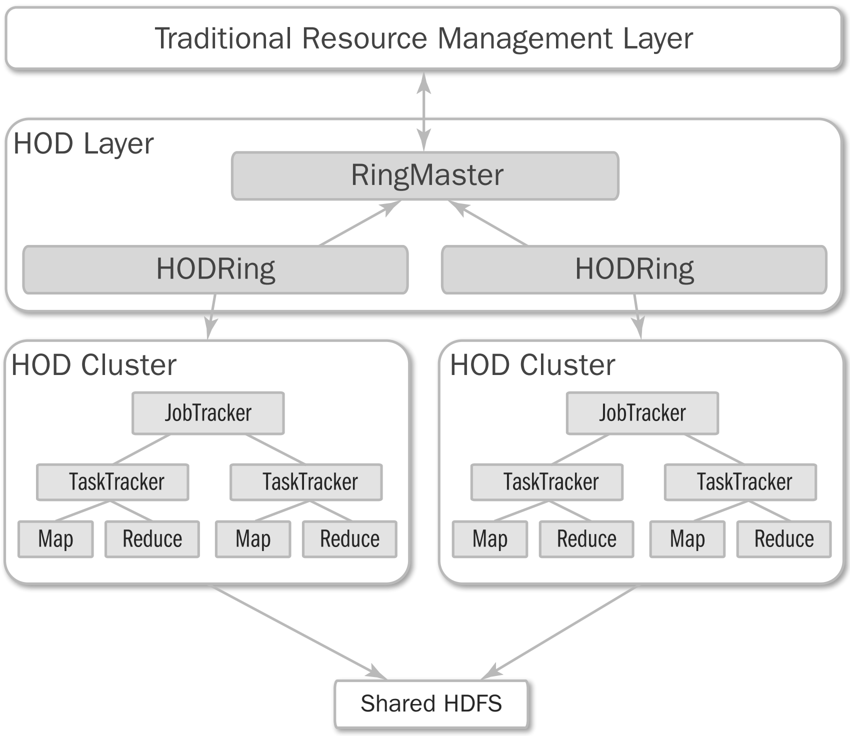


Figure 1.1 Hadoop on Demand architecture

- 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):

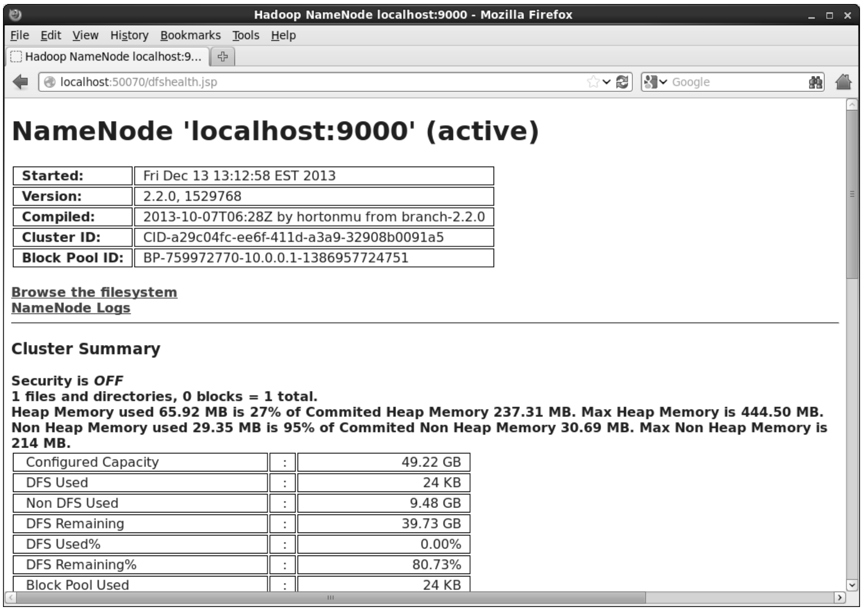
```
$ firefox  http://localhost:50070
```

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.


Hadoop NameNode localhost:9000 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Hadoop NameNode localhost:9...
localhost:50070/dfshealth.jsp
Google

NameNode 'localhost:9000' (active)

Started:	Fri Dec 13 13:12:58 EST 2013
Version:	2.2.0, 1529768
Compiled:	2013-10-07T06:28Z by hortonmu from branch-2.2.0
Cluster ID:	CID-a29c04fc-ee6f-411d-a3a9-32908b0091a5
Block Pool ID:	BP-759972770-10.0.0.1-1386957724751

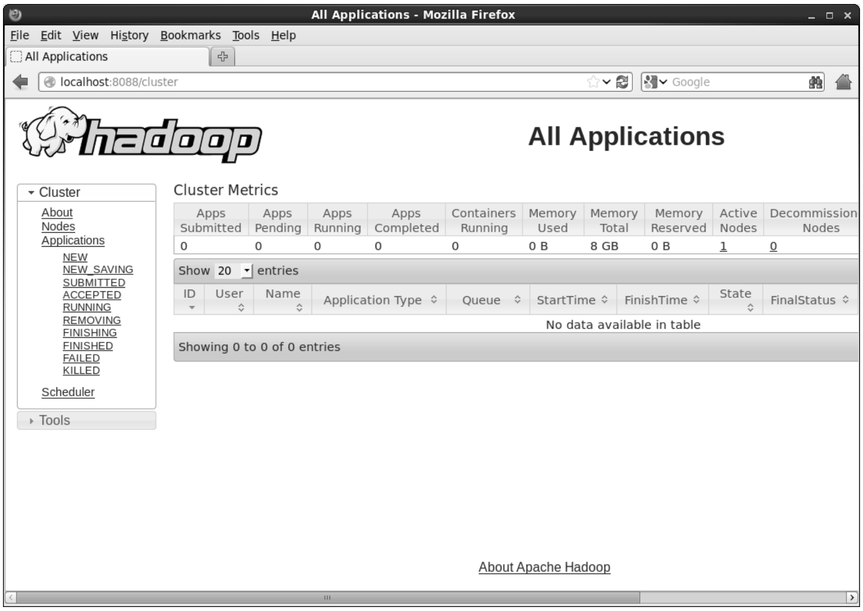
Browse the filesystem
NameNode Logs

Cluster Summary

Security is *OFF*
1 files and directories, 0 blocks = 1 total.
Heap Memory used 65.92 MB is 27% of Commited Heap Memory 237.31 MB. Max Heap Memory is 444.50 MB.
Non Heap Memory used 29.35 MB is 95% of Commited Non Heap Memory 30.69 MB. Max Non Heap Memory is 214 MB.

Configured Capacity	:	49.22 GB
DFS Used	:	24 KB
Non DFS Used	:	9.48 GB
DFS Remaining	:	39.73 GB
DFS Used%	:	0.00%
DFS Remaining%	:	80.73%
Block Pool Used	:	24 KB

Figure 2.1 Webpage for HDFS file system



All Applications

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	Active Nodes	Decommission Nodes
0	0	0	0	0	0 B	8 GB	0 B	1	0

Show 20 entries

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus
No data available in table								

Showing 0 to 0 of 0 entries

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.

- **randomwriter**: A map/reduce program that writes 10 GB of random data per node.
- **secondarysort**: An example defining a secondary sort to the reduce.
- **sort:** A map/reduce program that sorts the data written by the random writer.
- **sudoku:** A Sudoku solver.
- **teragen:** Generate data for the terasort.
- **terasort:** Run the terasort.
- **teravalidate:** Check the results of the terasort.
- **wordcount:** A map/reduce program that counts the words in the input files.
- **wordmean:** A map/reduce program that counts the average length of the words in the input files.
- **wordmedian:** A map/reduce program that counts the median length of the words in the input files.
- **wordstandarddeviation:** A map/reduce program that counts the standard deviation of the length of the words in the input files.

Some of the examples require files to be copied to or from HDFS. For those unfamiliar with basic HDFS operation, an HDFS quick start is provided in Appendix F. If you were able to complete the preceding steps, you should now have a fully functioning Apache Hadoop YARN system running in pseudo-distributed mode.

Wrap-up

With a working installation of YARN, the concepts, examples, and applications found in this book can be explored further without the need for a large production cluster. Keep in mind that many aspects of the configuration were simplified for this single-machine installation. In particular, a single workstation/server install does not have a true parallel HDFS or parallel MapReduce environment. Additional production installation scenarios are provided in Chapter 5, "Installing Apache Hadoop YARN."

This page intentionally left blank

This page intentionally left blank

Index

A

Access control, and shared clusters, 14–15
ACLs (Access Control Lists)
  administration of ResourceManager, 109
  Application ACLsManager, 119–120
  ApplicationMaster and, 139
  Capacity scheduler queues using, 47
  evolution of shared clusters, 12
  queues and, 159–160
Ad hoc clusters, 3
Admin window, Ambari dashboard, 103
Administration of YARN
  adding/decommissioning nodes, 107–108
  with Ambari, 99–103
  configuring Capacity scheduler, 108
  JVM analysis, 103–105
  managing jobs, 109–110
  monitoring basic Hadoop services, 92–95
  monitoring cluster health with Nagios, 90–92
  monitoring JVM, 95–97
  overview of, 85
  real-time monitoring with Ganglia, 97–98
  refreshing ACLs for administration of ResourceManager, 109
  refreshing superuser proxy groups mappings, 109
  refreshing user-to-groups mappings, 108
  reloading service-level authorization policy file, 109
  script-based configuration, 85–90
  setting container cores, 110
  setting container memory, 110
  setting MapReduce properties, 110–111
  tools for, 106–107
  user log management, 111–113
  using JobHistoryServer, 108
  YARN WebProxy, 108
Administration scripts
  code download, 247
  `configure-hadoop2.sh` script, 263–267
Administration Service, ResourceManager, 119
AggregatedLogDeletionService service, 111
Agility, MapReduce shared clusters issues, 18
Allocate API, 140–141, 146
Ambari, administration with, 99–103
Ambari-Agents, 73–74, 76
Ambari installation
  checking requirements, 73
  of Hadoop 2, 71–72
  installing Ambari server, 73
  installing and starting agents, 73–74
  installing HDP2.X cluster, 75–84
  performing, 72–73
  starting Ambari server, 74
Ambari-Server, 73–74
AMRMTokens SecretManager, 126
Apache
  Giraph framework, 242–243
  Hadoop. *See* MapReduce
  Hadoop YARN. *See* YARN
  Hive. *See* Hive
  Oozie worfklows, 188
  Pig. *See* Pig
  Spark framework, 244
  Storm framework, 244–245
  Tez framework, 242
Apache Software Foundation (ASF) site, 59
Application ACLsManager, 119–120, 136

Application development example, YARN
  ApplicationMaster. *See* ApplicationMaster, writing
  overview of, 191
  YARN client. *See* Client, writing YARN
APPLICATION LocalResources, 56, 132–133
ApplicationAttempts, 146–147
ApplicationClientProtocol, Client Service, 119
ApplicationConstants.Environment API, 149
ApplicationMaster
  adding local resources to container for, 235
  AMRMTokens SecretManager, 126
  cleanup on exit with FinishApplication-Request, 147
  client application life cycle, 51–53
  Client class launching, 233–234
  communication with, 149–150
  completed containers, 146
  as container 0, 49
  container specification, 42
  coordination and output commit, 146–147
  in Distributed-Shell application, 236–240
  enabling restarts, 182
  failures and recovery of, 146
  features of, 40
  information for clients, 147
  interaction with NodeManager, 145
  interaction with ResourceManager, 117, 120–121
  launching, 138–139
  launching containers, 145–146
  liveliness monitor, 121, 139–140
  monitoring MapReduce, 176–179
  overview of, 50
  requesting containers, 49
  resource requirements, 140
  ResourceRequest, 41–42
  responsibilities of, 38–39, 44, 139
  scheduling, 140–142
  scheduling protocol and locality, 142–144
  security, 147
  specifying resource timestamps to Node-Manager, 55
  submitting application to YARN, 201–205
  YARN applications formed by containers and, 115–116
ApplicationMaster Launcher, 123
ApplicationMaster, writing
  creating RMCallbackHandler, 215–217
  developing, 208–210
  initialization, 211–212
  launching container, 217–226
  overview of, 208
  registering with ResourceManager, 215
  YARN service life-cycle method invoca-tion, 214–215
ApplicationMasterProtocol, 120–121
ApplicationReport, 206–208
Applications
  Capacity scheduler limits/overriding limits on, 168–169
  interaction with ResourceManager, 120–121
  life cycle of client, 50–53
  responsibilities for writers of, 150–151
  running existing MRv1 applications with MRv2, 184–186
  unique ApplicationMaster for, 50
  YARN components and, 44–45
Applications, managing dependencies
  lifetime of LocalResources, 57
  LocalResource timestamps, 55
  LocalResource types, 55–56
  LocalResource visibilities, 56–57
  LocalResources definitions, 54–55
  overview of, 53–54
Applications status window, 175
ApplicationsManager, 122–123
ApplicationSubmissionContext, 198–202, 206–208
ApplicationSummary, 122
Architecture
  ApplicationMaster, 138–147
  Hadoop on Demand, 4–5
  NodeManager. *See* NodeManager architecture
  overview of, 115–117
  ResourceManager. *See* ResourceManager architecture

responsibilities of application writers, 150–151
shared MapReduce clusters, 9–15
YARN, 38–39
YARN containers, 148–150
ARCHIVE type
changing `LocalResourceType` to, 238
LocalResource as, 56
ASF (Apache Software Foundation) site, 59
Assign Masters window, Ambari host assignment, 79
Auditing, 14–15, 19
Authentication, 14–15, 119
Authorization, 109, 126–127
Auto-deallocation of idle clusters, 7
Automatic failover, Hadoop 2, 60
Auxiliary services, 135, 137

B

Backward compatibility. *See* Compatibility
Binary compatibility, MapReduce on YARN, 184–185
BSP (bulk-synchronous parallel model), 17

C

Callback handlers
creating RMCallbackHandler, 215–217
running ApplicationMaster, 213–214
Capacity, calculating node, 182–184
Capacity management with queues
Capacity scheduler properties, 156
enforcing fairness, 163–166
with hierarchical queues, 156–157
overview of, 160–163
Capacity scheduler
capacity management with queues, 160–163
configuration of, 108, 155–156
evolution of, 11–12
hierarchical queues, 156–159
introduction to, 153–155
limits on applications, 168–169
overview of, 47, 153
queue access control, 159–160
queues, 156
reservations, 166–167
state of queues, 167–168
user interface, 169
user limits, 163–166
`capacity-scheduler.xml`, 156
check_data_node.sh script, Nagios, 271–272
check_resource_manager.sh script, Nagios, 269–271
`chkconfig`, Ganglia, 98
CLC (container launch context)
ApplicationMaster–container manager communication, 52
containers described with, 50, 128
resource request to ResourceManager, 51
CLI (command-line interface), Client CLI in Distributed-Shell, 233–234
Client
Ambari Hadoop 2 install, 80
application life cycle, 50–53
ApplicationMaster and, 147
core functionality of YARN, 191–193
interaction with ResourceManager, 118–120
`Client` class, in Distributed-Shell application, 233–236
Client CLI, in Distributed-Shell application, 233–234
Client Service, ResourceManager, 118–119
Client, writing YARN
core functionality of, 191–193
initialization, 196–198
monitoring application, 205–208
submitting application to YARN, 198–205
Cluster menu, 46
Cluster Metrics table, 175
Clusters
Ambari Hadoop 2 install, 75
auto-deallocation of idle, 7
era of ad hoc, 3
evolution of shared. *See* Shared clusters
Ganglia monitoring Hadoop, 97–98
monitoring health with Nagios, 90–92
shortcomings of HOD, 8
Code downloads, 247
Command reference, HDFS quick reference, 279–280

Command-line
  HDFS quick reference, 281–282
  log interactions, 112
  MRv1/MRv2 script compatibility, 185
Compatibility
  MRv1 with MRv2 applications, 184–186
  MRv2, 181
  MRv2 incompatible APIs, 186
  YARN requirements for backward, 19–20
Completed containers, ApplicationMaster, 146
Completed jobs, management of, 11
Configuration
  Capacity scheduler, 155–156
  HOD, 7
  writing ApplicationMaster, 222–223
Configuration files
  Ambari installation, 72
  basic YARN administration, 106
  properties description. *See* http://yarn-book.com
  scripted Hadoop 2 install, 68–70
  YARN client constructor, 195
configure-hadoop2.sh script
  code, 263–267
  code download, 247
  script-based configuration with, 86
Constants, in Distributed-Shell application, 232–233
Container Executor, NodeManager, 136
Container ID field, ContainerTokens, 124
Container launch context. *See* CLC (container launch context)
ContainerAllocationExpirer, 123
ContainerLaunchContext
  in ApplicationMaster, 145
  ApplicationMaster setting up launch context, 237
  container environment, 148
  submitting application to YARN, 202–205
ContainerManagementProtocol, 240
ContainerManager, NodeManager
  auxiliary services, 135
  Containers Launcher, 135
  ContainersMonitor, 135–136
  LogHandler, 136
  resource localization configuration, 133–135
  resource localization process, 131–133
  resource localization service, 130–131
  RPC server, 130
Containers
  adding commands to launch containers, 239
  adding environment and file data to, 238
  adding local resources to AM container, 235
  ApplicationMaster allocating, 51–52
  ApplicationMaster monitoring, 40
  architecture of YARN, 117, 148–150
  `Client` class in Distributed-Shell application, 233–236
  coordination and output commit needed by, 146–147
  defined, 43, 148
  environment of, 148–149
  executing Distributed-Shell commands on, 229
  interaction with ApplicationMaster/NodeManager, 149–150
  launch specification API for, 42
  launching in ApplicationMaster, 145–146, 217–226
  launching in NodeManager, 137
  managing application dependencies, 53–57
  NodeManager responsibilities, 50
  overview of, 38–39, 49
  reserved, 48–49
  resource allocation model and, 50–51
  ResourceManager and, 33
  ResourceRequests and, 41–42, 142
  setting cores for, 110–111
  setting memory and cores for, 110
  storing shell script metadata in, 235
  uploading container libraries to HDFS, 234
Containers Launcher, ContainerManager, 135
ContainersMonitor, ContainerManager, 135–136

ContainerToken SecretManager, 124–126, 136
Coordination, with ApplicationMaster, 146–147
Copy files
- to HDFS, 283–284
- from HDFS, 284
- within HDFS, 285

`core-default.xml` file, 106
`core-site.xml` file
- quick-start YARN install, 24
- scripted Hadoop 2 install, 68–69

Cores, setting for containers, 110
Credentials, authentication and access control, 14–15
Customize Services window, Ambari, 81

D

DAG (directed acyclic graph), Apache Tez, 60
Dashboard, Ambari, 99–103
Data
- creating directories in YARN install for, 23
- Hadoop performance and, 35
- locality, MapReduce with YARN, 181

Data Nodes (HDFS), Ambari Hadoop 2 install, 80
DataNode service
- monitoring, 94
- quick-start YARN install, 24–25, 27

Debugging
- with `launch_container.sh` file, 230–231
- with user logs, 111

Decommissioning nodes
- HDFS, 285–286
- YARN, 107–108

Delegation tokens, ResourceManager, 119, 127
DelegationToken Renewer, 127
Delete, file/directory in HDFS, 285
DeletionService, LocalResources, 55
Dependencies, managing application, 53–57
Desktop machine, running YARN on, 21
Directed acyclic graph (DAG), Apache Tez, 60
Directories
- for data and logs in YARN install, 23
- making in HDFS quick reference, 283
- in scripted Hadoop 2 install, 63–65
- writing ApplicationMaster, 223–224

Distributed Cache, 13
Distributed-Shell application
- ApplicationMaster class, 235
- `Client` class, 233–236
- containers, 229
- `DSConstants` class, 232–233
- example, 228–229
- examples with arguments, 230–231
- final containers, 240
- main classes, 232
- running shell commands/scripts in containers, 241
- using, 227–228

Domain controller user interface, 224–225
Domain mode, configuring ApplicationMaster, 222–223
Dreadnaught, 2
Dryad on YARN, 243–244
`DSConstants` class, in Distributed-Shell application, 232–233
Dynamic information, container environment, 149
Dynamic resource requirements, ApplicationMaster, 140

E

Eclipse Memory Analyzer, JVM heat dumps, 104–105
Eden subsegment, JVM, 104
Elasticity
- Capacity scheduler support for, 154
- shortcomings of HOD, 8–9

End-user MapReduce API, 36
Environment, container, 148–149
Environment variables, 204
`Environment.JAVA_HOME.$()`, 204
EPEL (Extra Packages for Enterprise Linux)
- Nagios installation, 90
- scripted Hadoop 2 install, 60–61

Expiry time, ContainerAllocation-Expirer, 123
Expiry timestamp field, ContainerTokens, 125

F

Failures
  ApplicationMaster, 146, 182
  container, 146
Fair scheduler
  under development, 48
  evolution of shared clusters, 11–12
  overview of, 47–48
Fault tolerance, of MapReduce with YARN, 181
FIFO (first in, first out) scheduler, 46–47, 155
File-based mode, scripted Hadoop 2 install, 64
File-system URLs, container environment, 148
FILE type, 55–56
Files, listing in HDFS quick reference, 282–283
Finish API, ApplicationMaster, 147
FinishApplicationMasterRequest, 240
FinishApplicationRequest, ApplicationMaster, 147
Flexible Resource Model, requirements for YARN, 18–19
FQDN
  Ambari Hadoop 2 install issues, 83
  configuring Ambari-Agent ini file, 74
  scripted Hadoop 2 install, 61–62
  starting Ambari server, 74
Frameworks, YARN
  Apache Giraph, 242–243
  Apache Spark, 244
  Apache Storm, 244–245
  Apache Tez, 242
  distributed shell, 241
  Dryad on YARN, 243–244
  end-user interactions with, 116
  Hadoop MapReduce, 241–242
  Hamster, 245
  Hoya: HBase on YARN, 243
  REEF, 245
FSCK, performing in HDFS quick reference, 281

G

Ganglia
  managed by Ambari, 72, 99
  real-time monitoring with, 97–98
Generations, JVM processes, 103–104
GetNewApplicationResponse, 198–201
Giraph framework, 242–243
`gmetad` daemon, Ganglia, 97–98
`gmond` daemon, Ganglia, 97–98
Google's Pregel, Giraph based on, 242
Granular scheduling, Capacity scheduler supporting, 154–155
Groups, quick-start YARN install, 23

H

HA (high availability), NameNode, 60
Hadoop. *See also* Scripted Hadoop 2 uninstallation
  core components of version 1, 21
  download from website, 22–23
  evolution of, 2–3
  installation. *See* Installation, YARN
  introduction to. *See* http://yarn-book.com
  Phase 0, era of ad hoc clusters, 3
  Phase 1, Hadoop on Demand. *See* Hadoop on Demand (HOD)
  Phase 2, shared clusters. *See* Shared clusters
  Phase 3, emergence of YARN, 18–20
  review summary, 20
  scripted Hadoop 2 uninstallation, 68
  YARN. *See* YARN
`hadoop-cluster.cfg` file, monitoring, 93–95
Hadoop on Demand (HOD)
  development of, 3
  evolution of shared clusters vs., 9–14
  features and advantages of, 6–7
  HDFS in, 5
  overview of, 3–5
  shortcomings of, 7–9
hadoop-xml-conf.sh script
  code, 258–261
  XML file processing, 68, 86
`HADOOP_HOME` variable, scripted Hadoop 2 install, 63–64
`hadoop.http.staticuser.user` property, quick-start YARN install, 24
`HADOOP_VERSION` variable, scripted Hadoop 2 install, 63–64
Hamster, 245

Hard drive space, YARN installation, 22
Hardware, YARN installation, 22
`hdfs-default.xml` file, 106
HDFS (Hadoop Distributed File System)
  ad hoc clusters and, 3
  advancements in Phase 2, 10
  as core component, 21–22
  format during YARN install, 26
  in HOD, 5
  improvements to, 60
  quick reference, 279
  scripted Hadoop 2 install, 65, 67
  security credentials, 14
  shared, 4–5
  starting services during YARN install, 27
  uploading container libraries to, 234
HDFS (Hadoop Distributed File System) quick reference
  copy files from, 284
  copy files to, 283–284
  copy files within, 285
  decommissioning nodes, 285–286
  delete directory in, 285
  delete file within, 285
  general commands, 281–282
  getting status report, 280
  list files, 282–283
  make directory, 283
  performing FSCK, 281
  quick command reference, 279–280
  starting HDFS web GUI and, 280
`hdfs-site.xml` file
  quick-start YARN install, 24–25
  scripted Hadoop 2 install, 69
HDP 2.0 (Hortonworks Data Platform), Ambari Automated Install, 73, 75–84
Heap dumps, analyzing JVM, 104–105
Heap sizes, modifying JAVA for YARN install, 26
Heartbeats to ResourceManager
  ApplicationMaster container allocation, 52
  ApplicationMaster liveliness monitor, 121, 139–140
  ApplicationMaster scheduling, 140–141
  ApplicationMaster sending, 50
  NodeManager sending, 50
  with Resource Tracker Service, 121–122
Heatmaps window, Ambari dashboard, 99–100
Hierarchical queues
  Capacity scheduler, 156–159
  Fair scheduler, 49
High availability (HA), NameNode, 60
High Cluster Utilization, 8, 19
High-performance computing (HPC), Hamster, 245
Hive, 60, 187
HOD (Hadoop on Demand)
  advantages of, 6–7
  HDFS and, 5
  moving to shared clusters from, 10–15
  overview of, 3–5
  shortcomings and retirement of, 7–9
HODRings, 4
Hosts
  Ambari dashboard, 101–102
  Ambari Hadoop 2 install, 77, 79
  Nagios group definition, 90–91
  scripted Hadoop 2 install, 64
Hoya: HBase on YARN framework, 243
Httpd services, configuring Nagios, 90

I

Info operation, multiple concurrent clusters, 6
init method
  ApplicationMaster initialization, 211–212, 237
  developing ApplicationMaster, 210
  running ApplicationMaster, 213–215
  YARN client initialization, 196–198
init scripts
  monitoring Hadoop services, 92
  starting Hadoop, 71, 87
Initialization
  writing ApplicationMaster script, 211–213
  writing YARN client script, 196–198
Input/output paths, containers, 148
install-hadoop2.sh script
  code for, 249–256
  scripted Hadoop 2 install, 63
Install quick start, YARN
  configuring `core-site.xml`, 24
  configuring `hdfs-site.xml`, 24–25

Install quick start, YARN (*continued*)
configuring mapred-site.xml, 25
configuring yarn-site.xml, 25–26
creating users and groups, 23
downloading Apache Hadoop, 22–23
formatting HDFS, 26
getting started, 22
HDFS services, 27
making data and log directories, 23
minimal requirements, 22
modifying Java heap sizes, 26
overview of, 21
setting JAVA_HOME, 23
testing with sample MapReduce examples, 30–31
verify running services using web interface, 28–29
wrap-up, 31
YARN services, 28
Installation scripts
code download page for, 247
hadoop-xml-conf.sh, 258–261
install-hadoop2.sh, 249–256
uninstall-hadoop2.sh, 256–258
Installation, YARN
with Ambari. *See* Ambari installation
basics, 59–60
by downloading from ASF site, 59
improvements other than YARN, 60
overview of, 59
from scratch. *See* Install quick start, YARN
scripted. *See* Scripted Hadoop 2 installation
scripted uninstallation, 68
system preparation, 60–62
troubleshooting. *See* http://yarn-book.com
Instances, HDFS, 3
Interactive mode, scripted Hadoop 2 install, 64
Interface, Capacity scheduler, 169
IPC address, ApplicationMaster and, 139
Iptables, configuring Nagios, 90
Isolation on individual nodes, and shared clusters, 12–13

J

JAR file
testing YARN installation, 30–31
writing ApplicationMaster, 221–223
writing YARN client, 202–203
Java for YARN install, 22, 26
Java Virtual Machine (JVM)
analysis, 103–105
evolution of MapReduce framework, 15
monitoring, 95–97
JAVA_HOME
quick start YARN install, 23
scripted Hadoop 2 install, 63, 64
JBoss AS (JBoss application server) cluster. *See* Application development example, YARN
JDK (Java Development Kit)
installing Ambari server, 73
quick-start YARN install, 23
scripted Hadoop 2 install, 59, 62, 64
jhat utility, analyzing JVM heat dumps, 104–105
jmap tool, analyzing memory usage, 104
Job ACLs, 14–15
JobHistoryServer
managing completed jobs, 11
in MapReduce version 2, 182
storing MapReduce history, 108
Jobs
authentication and access control, 14–15
Capacity scheduler for, 11–12
isolation on individual nodes, 12–13
JobTracker memory management for, 10
management of completed, 11
managing MapReduce, 109
managing YARN, 109–110
recovery and upgrades, 12
scripted Hadoop 2 install, 65, 67
user log management, 111
JobToken-based authentication, 14
JobTracker
ApplicationMaster replacing in version 2, 182
Hadoop on Demand and, 4–5, 7
MapReduce shared clusters using, 9–13, 15–18
reworking in YARN, 18–19
YARN process flow, 36–38
JVM (Java Virtual Machine)
analysis, 103–105

evolution of MapReduce framework, 15
monitoring, 95–97

K

Kerberos, 14, 119
Kill events, client application life cycle, 53
`-kill`, `yarn application` command, 109–110

L

`launch_container.sh` file
in Distributed-Shell application, 230–231
writing ApplicationMaster, 224–225
Launching containers
ApplicationMaster, 145–146
NodeManager, 137
Leaf queues
defined, 157
enforcing fairness with user limits, 163–166
naming queues in Capacity scheduler, 159
scheduling, 157–158
stopping/restarting, 167–168
Libraries, uploading container libraries to HDFS, 234
Life cycle
client application, 50–53
LocalResources, 57
NodeManager overseeing container, 49
YARN service, 214–215
Limits, Capacity scheduler, 168–169
Linux file system, Hadoop 2 installation, 59
`linux-server` template, Nagios, 90
LinuxTaskController, 14
List files, HDFS quick start, 282–283
List operation, HOD, 6
`-list`, `yarn application` command, 109–110
Liveliness monitor
ApplicationMaster, 121, 139–140
NodeManager, 122
LocalCache, LocalResources, 55–56
Locality
Hadoop on Demand issues, 7–8
ResourceRequest constraints, 144
Locality Awareness
of Capacity scheduler, 155
as key requirement for YARN, 8, 19
Localization
LocalResources, 54
NodeManager, 131–133
resource localization service, 130–131
Localizer, LocalResources, 55
LocalResources
adding to AM container, 235
definitions, 54–55
initializing metadata for, 237
lifetime of, 57
localization of PRIVATE/APPLICATION resources, 132–133
localization of PUBLIC resources, 131–132
managing application dependencies, 53–54
modifying ApplicationMaster code, 235–237
managing with `Client` class, 233–234
resource localization service, 131
submitting application to YARN, 202–204
target locations of, 133
timestamps, 55
types, 55–56
visibilities, 56–57
writing ApplicationMaster, 221–222
LogHandler, ContainerManager, 136
Logs
administration and configuration, 112–113
aggregated output, 89–90
aggregation in YARN, 111
command-line utility interacting with, 112
container environment, 148
in Hadoop on Demand, 6
NodeManager overseeing, 49, 137
permissions, 113
quick-start YARN install, 23, 27
scripted Hadoop 2 install, 65
web user interface and, 112
Loss of information issue, ResourceRequests, 142–143

M

Macros, Nagios, 94–95
Main method, writing YARN client, 193–194
Map, Hadoop on Demand, 4–5
Map slots
  in earlier Hadoop versions, 50
  Fair scheduler departure from, 48
  YARN departure from, 45
`mapred-default.xml` file, 106
`mapred job` command, 109
`mapred-site.xml` file
  calculating node capacity, 182–183
  quick-start YARN install, 25
  scripted Hadoop 2 install, 69
  setting MapReduce properties, 110–111
MapReduce
  Apache Tez framework and, 60
  evolution of shared clusters. *See* Shared clusters, evolution of MapReduce
MapReduce (MRv1)
  abuse of, 17
  basic structure of, 33–34
  compatibility of MRv2 applications with, 184–186
  death of JobTracker in, 182
  evolution of, 15
  Hadoop on Demand issues, 7–9
  process flow, 35–37
  running existing code, 187–188
  shared cluster issues, 15–18
MapReduce with YARN (MRv2)
  ApplicationMaster failures and recovery, 182
  basic structure, 34–35
  calculating node capacity, 182–184
  compatibility, 181–182
  configuration file, 69–70
  debugging with user logs, 111
  defined, 21
  features, 241–242
  JobHistoryServer, 108
  LocalResource timestamps, 55
  managing jobs, 109
  need for non-MapReduce workloads, 37
  overview of, 171
  paradigm, 35
  setting properties, 110–111
  shuffle service, 137, 184
  untested features, 188–189
  user agility, 38
MapReduce with YARN (MRv2), running existing examples
  monitoring examples with web GUI, 174–179
  overview of, 171–172
  pi example, 172–174
  terasort benchmark, 180
  TestDFSIO benchmark, 180–181
  testing quick start installation, 30–31
  version 1 applications, 184–186
  version 1 existing code, 187–188
`mapreduce.framework.name` property, quick-start YARN install, 25
Master key identifier field, Container-Tokens, 125
`master_memory`, submitting application to YARN, 202
Maui, 4, 7–8
Memory
  analyzing usage on running application, 104
  Capacity scheduler for applications with high, 47
  isolation on individual nodes and, 12–13
  issues of MapReduce shared clusters, 18
  managing JobTracker, 10
  setting for containers, 110
  submitting application to YARN, 202, 206
Message Passing Interface (MPI), 245
Metadata
code for storing shell script metadata in containers, 235
for `DSConstants` class, 232
initializing for local resources, 237
Metrics dashboard, Ambari, 72
minimum-user-limit-percent property, capacity management, 164–165
Monitoring
  Ambari server, 72
  basic Hadoop services, 92–95
  cluster health with Nagios, 90–92
  JVM, 95–97
  MapReduce examples with web GUI, 174–179

real-time with Ganglia, 97–98
YARN applications, 206–208
MPI (Message Passing Interface), 245
MPICH2, 245
Multitenancy
Capacity scheduler support for, 154
Hadoop on Demand and, 3–4, 7
requirements for YARN, 19

N

Nagios
Ganglia monitoring versus, 97–98
managed by Ambari, 99
monitoring cluster health, 90–92
monitoring Hadoop services, 92–95
Nagios modules
check_data_node.sh, 271–272
check_resource_manager_old_space_pct.sh, 272–275
check_resource_manager.sh, 269–271
code download, 247
Nagios Remote Plugin Executor (NRPE), 93–95
NameNode
federation, 60
HA (high availability), 60
quick-start YARN install, 24, 26–27
scripted Hadoop 2 install, 64–65
Naming conventions
hierarchical queues, 158–159
scripted Hadoop 2 install, 64
Network partitions, application coordination issues, 147
New generation, JVM processes, 103–104
NMToken SecretManager, 126–127, 136
Node Health CheckerService component, 136
Node-level isolation, 12–13
NodeManager
Ambari Hadoop 2 install with, 80
ApplicationMaster communication pathway with, 208–210
client application life cycle and, 51–52
interaction of containers with, 149–150
interaction of nodes with Resource Manager, 121–122
launching containers, 38, 145, 220–221
liveliness monitor, 122
LocalResources, 55–56
monitoring, 94
overview of, 49–50
as per-machine slave, 38
quick-start YARN install with, 25–26
ResourceManager working with, 117
ResourceRequests and, 46
responsibilities of, 43–44
SecretManager ContainerTokens and, 124–126
user log management, 111
YARN control elements, 38–39
NodeManager architecture
Container Executor component, 136
ContainerManager component, 130–136
important functions of, 137
Node Health CheckerService component, 136
NodeStatusUpdater component, 129–130
overview, 117
overview of components, 128–129
responsibilities of, 127–128
security components, 136
Nodes
adding/decommissioning, 15, 107–108
Administration Service refreshing, 119
Ambari Hadoop 2 install, 76
calculating capacity of, 182–184
Capacity scheduler reservations on, 166–167
integrating scripts with services management, 71
interaction with Resource Manager, 121–122
scripted Hadoop 2 install, 62, 64
Nodes-list manager, 122
Nodes status window, Hadoop, 175
NodeStatusUpdater component, 129–130
Non-MapReduce workflows, 33–35, 37
NRPE (Nagios Remote Plugin Executor), 93–95
`number of containers`, ResourceRequest, 41

O

Old generation, JVM processes, 103–104
Online resources
and additional information, 277–278

Online resources (*continued*)
  Apache Giraph, 243
  Apache Spark framework, 244
  Apache Storm framework, 245
  Apache Tez, 242
  available code downloads, 247
  Capacity scheduler configuration, 108
  currently running scheduler, 46
  Dryad on YARN, 243–244
  Hadoop website, 22
  HDFS options in Hadoop 2, 60
  HDFS quick reference, 279
  Hoya: HBase on YARN framework, 243
  Java for YARN install, 23
  most recent version of Hadoop, 62
  MPICH2, 245
  Nagios, 90
  Parallel Distributed Shell, 60
  REEF framework, 245
OpenJDK path, scripted Hadoop 2 install, 63
OpenSSH package, scripted Hadoop 2 install, 61–62
Options class, command-line options, 196
`org.apache.hadoop.mapred` APIs, 184–185
`org.apache.hadoop.mapreduce` APIs, 183
Output commit, ApplicationMaster, 146–147

P

Parallel Distributed Shell, 60–62
Parallel map phase. *See also* Map slots, 35
Parent queues
  defined, 157
  naming in Capacity scheduler, 159
  scheduling, 157–158
  stopping/restarting, 167–168
Passwords
  Ambari user database, 81
  configuring secure shell without, 61, 73–74
  developing YARN ApplicationMaster, 209, 211–212
  setting Nagios, 91
  submitting application to YARN, 200, 205
  YARN client initialization, 198
  YARN client main method, 193, 195–196
Path name, Capacity scheduler queues, 156
PATTERN type, LocalResource as, 56
`pdcp` tool
  defined, 60
  installing Ambari agents, 74
  script-based configuration, 86
`pdsh` tool
  defined, 60
  installing Ambari agents, 74
  installing Ganglia, 97
  script-based configuration, 87
  scripted Hadoop 2 install, 60–62
Performance, Hadoop, 35
Permanent generation, JVM processes, 103–104
Permissions
  HDFS, 3
  log, 113
  shared cluster, 14
  writing ApplicationMaster, 223
Phases. *See* YARN, history of
pi example, 172–174
pid directories, scripted Hadoop 2 install, 65
Pig, 60, 187
Platform, YARN, 115–116
Plug-ins, Nagios, 90
Pluggable scheduler, ResourceManager, 40
Pluggable shuffle and sort, 188–189
Policies
  Capacity scheduler scheduling, 155
  reloading service-level authorization, 109
Ports, writing ApplicationMaster, 222–223
Pregel, 242
Priorities, MapReduce with YARN application, 181
Priority, ResourceRequests
  defined, 41, 142
  example, 144
  submitting application to YARN, 206
PRIVATE LocalResources, 56, 132–133
Programming Model Diversity, 17
Properties
  adding/decommissioning YARN nodes, 107–108
  ApplicationMaster, 222
  Capacity scheduler, 108
  log administration and configuration, 112–113

MapReduce, 110–111
queues in Capacity scheduler, 156
quick-start YARN install, 24, 25
refreshing ACLs for ResourceManager administration, 109
refreshing superuser proxy groups mappings, 109
refreshing user-to-groups mappings, 108
scripted Hadoop 2 install for MapReduce, 69–70
YARN WebProxy, 108
Proxy groups mappings, refreshing superuser, 109
Proxy servers, Web Application Proxy, 108
Pseudo-distributed installation, 22
PUBLIC LocalResources, 56, 131–133

Q

QJM (Quorum Journal Manager), 60
Queue access control, Capacity scheduler, 159–160
Queue ACLs, 14
Queue paths, Capacity scheduler, 158
Queues
Administration Service refreshing, 119
Capacity scheduler. *See* Capacity scheduler
controlling who can submit jobs to specific, 14
defined, 156
Fair scheduler, 48–49
FIFO scheduler, 46–47
scheduling jobs with, 11–12
submitting application to YARN, 206
Quick command reference, HDFS quick reference, 279–280

R

`radmin` utility
adding new queues at runtime, 159
warning messages when executing, 185
RAM, YARN installation requirements, 22
Recovery
ApplicationMaster, 146, 182
enabling for completed tasks, 182
evolution of shared clusters, 12
Red Hat (RPM-based installation)
defined, 60
Nagios, 90
scripted Hadoop 2, 60–62
single-node YARN server configuration, 22–23
Reduce slots
MapReduce, 35
static allocation in earlier Hadoop versions, 50
REEF (Retainable Evaluator Execution Framework), 245
RegionServers, Ambari Hadoop 2 install, 80
Registration, ApplicationMaster, 139
relaxLocality flag, ResourceRequests, 142
Reliability and Availability, YARN requirements, 17, 19
Reliability, MapReduce shared cluster issues, 16–17
Remote procedure calls. *See* RPCs (remote procedure calls)
Reservations, Capacity scheduler, 166–167
Reserved container, Fair scheduler, 48–49
Resource allocation model, 50
Resource capability, ResourceRequests, 142
Resource container, scheduler, 38–39
Resource field, ContainerTokens, 124
Resource localization service
configuring, 133–135
ContainerManager, 130–131
process of, 131–133
Resource location, ResourceRequests, 142
Resource management
client application life cycle, 50–53
Hadoop on Demand, 4–7
moving to shared clusters from, 9
YARN providing, 35–39
Resource Manager, Hadoop on Demand, 4–5, 9
`resource-name`, ResourceRequest, 41
`resource-requirement`, ResourceRequest, 41
Resource requirements, ApplicationMaster, 140
Resource scheduler, YARN as, 21
Resource Tracker Service, 121–122

Resource usage, NodeManager overseeing, 49
ResourceLocalizationService, LocalResources, 55
ResourceManager
  adding/decommissioning YARN nodes, 107–108
  ApplicationMaster's communication pathway with, 208–210
  client application life cycle, 51–53
  failures affecting cluster availability, 46
  features of, 39–40
  granting ResourceRequest, 41
  overview of, 45–46
  refreshing ACLs, 109
  registering ApplicationMaster with, 215, 237
  responsibilities of, 43–44
  scheduling containers, 49
  tasks not responsible for, 46
  YARN control elements, 38–39
ResourceManager architecture
  application interaction with, 120–121
  architectural overview, 117
  client interaction with, 118–120
  components, 117–118
  core components, 122–123
  defined, 117
  node interaction with, 121–122
  overview of components, 117–118
  security-related components, 123–127
ResourceManagerAdministrationProtocol, Administration Service, 119
ResourceRequests
  issued by ApplicationMaster, 44, 141–143
  locality constraints, 144
  loss of information issue, 142–143
  scheduling example, 143–144
  as strict or negotiable, 46
Resources
  Capacity scheduler limits/overriding limits on, 168–169
  Capacity scheduler support for, 154
  features of YARN model, 41
  issues of MapReduce shared clusters, 17–18
  submitting application to YARN, 206
Retainable Evaluator Execution Framework (REEF), 245
Review window, Ambari, 82
RingMaster, HOD architecture, 4
RMDelegationToken SecretManager, 127
ROOT queue, hierarchical
  defined, 157
  naming in Capacity scheduler, 158–159
  scheduling among queues, 157–158
  stopping/restarting, 167–168
RPC server, ContainerManager, 130
RPCs (remote procedure calls)
  ApplicationMasters service, 120
  Client Service, 119
  management of completed jobs, 11
RPM-based installation. *See* Red Hat (RPM-based installation)
RUNNING state, queues, 167–168

S

Scalability
  with ApplicationMaster, 40
  authentication and access control, 14–15
  building share compute platform with, 1
  evolution of Apache Hadoop, 2–3
  Hadoop 2 installation addressing, 37
  requirements for YARN, 19
  ResourceManager addressing, 45
  shared MapReduce cluster issues, 15–16
Scheduling
  abuse of MapReduce, 17
  among hierarchical queues, 157–158
  ApplicationMaster and, 140–144
  with Capacity scheduler. *See* Capacity scheduler
  with Fair scheduler, 47–49
  with FIFO scheduler, 46–47
  overview of, 46
  ResourceManager limited to, 38–39, 45–46
  shared clusters and, 11–12
  with YarnScheduler, 123
Script-based configuration, YARN administration, 85–90
Scripted Hadoop 2 installation
  configuration file processing, 68
  configuration file settings, 68–70
  downloading/extracting scripts, 63

of Hadoop 2, 62
JDK options, 62
providing node names, 64
running script, 64–65
setting script variables, 63
start-up scripts, 71
system preparation for, 60–62
verifying installation, 65–68
Scripted Hadoop 2 uninstallation, 68
Scripts
creating service monitoring, 92–95
downloading/installing install, 63
integrating with services management, 71
YARN installation. *See* Installation scripts
Secondary NameNode service, quick-start YARN install, 24–25, 27
SecretManager
AMRMTokens, 126
ContainerTokens, 124–126
NMTokens, 126–127
ResourceManager and, 124–126
RMDelegationToken, 127
Secure and Auditable Operation, YARN requirements, 15, 19
Security
ApplicationMaster, 147
authentication and access control, 14–15
Capacity scheduler, 154
Client Service authentication, 119
container environment, 149
evolution of shared clusters, 13–14
NodeManager, 136
ResourceManager, 124–127
Web Application Proxy in YARN addressing, 108
Service-level authorization policy file, reloading, 109
Services
Ambari Hadoop 2 install, 78, 81
evolution of shared clusters, 12
functionality in YARN, 196
of Hadoop on Demand, 6
managed by Ambari, 99
MapReduce shared cluster issues, 16–17
monitoring basic, 92–95
in quick-start YARN install, 27–28
ResourceManager web, 120
verifying with web interface, 28–29
YARN requirements for, 19
Services window, Ambari dashboard, 100–102
Shared clusters
Capacity scheduler for large, 47
Fair scheduler for large, 47–48
overview of, 9–10
Shared clusters, evolution of MapReduce
authentication and access control, 14–15
central JobTracker daemon, 10
central scheduler, 11–12
HDFS instances, 10
isolation on individual nodes, 12–13
issues of, 15–18
JobTracker memory management, 10
management of completed jobs, 11
MapReduce framework, 15
miscellaneous management features, 15
overview of, 9
recovery and upgrades, 12
security, 13–14
`-shell_args` option, adding arguments to Distributed-Shell application, 230
Shuffle service
as MapReduce auxiliary service, 137
MapReduce version 2 changes, 184
pluggable, 188–189
Single point of failure, JobTracker, 12
Slaves, Ambari Hadoop 2 install, 80
Software, distribution of Hadoop, 7
Software stack, Ambari Hadoop 2 install, 75
Source compatibility, `org.apache.hadoop.mapreduce` APIs, 183
`ssh` keys, scripted Hadoop 2 install, 61–62
`start()` method, submitting application to YARN, 198–201
Start-up scripts, 71
StartContainerRequest, 145
StartContainerResponse, 145
States, queue, 167–168
Static allocation issues, earlier Hadoop versions, 50
Static resource requirements, Application-Master, 140
Status report, HDFS quick reference, 280
`stderr` directory, in Distributed-Shell application, 229

`stdout` directory, in Distributed-Shell application, 229
Stinger Initiative Phase 3 release, 242
StopContainersResponse, 145
STOPPED state, queues, 167–168
Submitting application to YARN, 198–205
Summary window, Ambari, 83
Superuser proxy groups mappings, refreshing, 109
Survivor Space I subsegment, new generation JVM, 104
System requirements
  Ambari Hadoop 2 install, 73
  preparation for YARN Installation, 60–62
  scripted YARN installation, 59–60
  YARN install quick start, 22
  YARN redesign, 18–20

T

TaskController, 14
TaskTrackers
  Hadoop on Demand architecture, 4–5
  Hadoop on Demand data locality issues, 7–8
  health-check script in, 15
  JobTracker managing, 37
  MapReduce version 2, 36
  responsibilities of, 37
  shared MapReduce clusters, 9, 13–14, 18
Templates
  configuring Nagios, 90
  monitoring basic services, 93–95
Tenured generation, JVM processes, 103
Terasort benchmark, 180
TestDFSIO benchmark, 180–181
Testing
  scripted Hadoop 2 install, 65–66
  YARN installation, 30–31
Tez framework, 242
Timeout, killing long-running applications, 236
Timestamps, LocalResource, 55, 203
Tokens
  container environment, 149
  ContainerToken SecretManager, 124–126, 136
  DelegationToken Renewer, 127
  NMToken SecretManager, 126–127, 136
  RMDelegationToken SecretManager, 127
Tools, administrative, 106–107
Torque, 4, 7–8, 15

U

Uber Jobs, 188–189
uninstall-hadoop2.sh script, code, 256–258
Upgrades, shared clusters, 12
URL, LocalResources, 54, 56–57
Use-cases, evolution of Apache Hadoop, 3
User-limit-factor, capacity management, 166
User logs
  authentication and access control, 14
  management of, 111–113
  MapReduce shared cluster issues, 18
User-session, HOD architecture, 4–5
User-to-group mappings, refreshing, 108, 119
Users
  Capacity scheduler limits, 163–166
  creating during YARN install, 23
  HOD enabling multiple, 6
  interaction with frameworks, 116

V

Visibility, LocalResource
  APPLICATION, 56–57
  PRIVATE, 56
  PUBLIC, 56
  specifying, 57
  submitting application to YARN, 203

W

Web Application Proxy, 108
Web application, ResourceManager, 120
Web interface
  Capacity scheduler, 169
  configuring Nagios, 90
  HDFS quick reference, 280
  log aggregation and, 112
  monitoring MapReduce examples with, 174–179

scripted Hadoop 2 install, 65–66
verifying services after YARN installation, 28–29
viewing ApplicationMaster on YARN, 224–225
Web server, NodeManager security, 136
Web services, ResourceManager, 120
WebMap application, 2
WebProxy, YARN, 108
Writers of applications
example. *See* Application development example, YARN
potential multiple-writer problem, 147
responsibilities of, 150–151

X

`-XX:NewRatio`, JVM analysis, 104
`-XX:NewSize`, JVM analysis, 104

Y

YARN
High Cluster Utilization priority for, 8
Locality Awareness priority for, 8
MapReduce. *See* MapReduce
YARN and Hadoop ecosystem
beyond MapReduce, 33–35
improved utilization, 38
need for non-MapReduce workloads, 37
original MapReduce design, 35–37
overview of, 33
scalability, 37–38
user agility, 38
YARN architecture, 38–39
YARN components, 39–42
`yarn application` command, 109–110
YARN components
ApplicationMaster, 40, 50
architecture overview, 43–45
Capacity scheduler, 47
client application life cycle, 50–53
containers, 41–42, 49
Fair scheduler, 47–49
FIFO scheduler, 46–47
managing application dependencies, 53–57
NodeManager(s), 49–50
overview of, 43
relationship between application and, 44–45
resource model, 41
ResourceManager, 39–40, 45–46
ResourceRequest, 41
review summary, 57–58
`yarn-default-xml` file
client constructor, 195
client initialization, 196
schedule class, 46
YARN, history of
evolution of Apache Hadoop, 2–3
introduction to, 1–2
overview of, 1
Phase 0, era of ad hoc clusters, 3
Phase 1, Hadoop on Demand, 3–9
Phase 2, shared compute clusters, 9–18
Phase 3, emergence of YARN, 18–20
review summary, 20
`yarn rmadmin`
adding new queues at runtime, 159
configuring Capacity scheduler, 108, 156
refreshing ACLs for administration of ResourceManager, 109
refreshing superuser proxy groups mappings, 109
for YARN administrative tools, 106–107
`yarn-site.xml` file
calculating node capacity, 182–183
configuring resource localization, 133–135
configuring YARN install, 25–26
in Distributed-Shell application, 230
enabling recovery of completed tasks, 182
scripted Hadoop 2 install, 70
setting container cores, 110
setting container memory, 110
YARN client constructor, 195
YARN client initialization, 196
`yarn.acl.enable` property, 119
`yarn.admin.acl` property, 109, 120
`YarnClientApplication`, 198–202
`YarnConfiguration` class, 195–196
`yarn.log-aggregation-enable` property, 113
`yarn.nodemanager.aux-services` property, 25–26

yarn.nodemanager.log-dirs property, 112
yarn_proxy_host file, hostname, 64
YarnScheduler
  ApplicationMaster forwarding requests to, 120
  overview of, 123
  Resource Tracker Service forwarding node-heartbeat to, 122
Yet Another Resource Negotiator (YARN). *See* YARN
Young generation, JVM processes, 103–104

Z

ZKFailoverController, 60
ZooKeeper, Hadoop 2, 60