Sams Teach Yourself

Big Data Analytics with Microsoft HDInsight®

in 24 Hours
# Contents at a Glance

Introduction ............................................................................................................. 1

**Part I: Understanding Big Data, Hadoop 1.0, and 2.0**

<table>
<thead>
<tr>
<th>HOUR</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction of Big Data, NoSQL, and Business Value Proposition</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>Introduction to Hadoop, Its Architecture, Ecosystem, and Microsoft Offerings</td>
<td>31</td>
</tr>
<tr>
<td>3</td>
<td>Hadoop Distributed File System Versions 1.0 and 2.0</td>
<td>65</td>
</tr>
<tr>
<td>4</td>
<td>The MapReduce Job Framework and Job Execution Pipeline</td>
<td>113</td>
</tr>
<tr>
<td>5</td>
<td>MapReduce—Advanced Concepts and YARN</td>
<td>145</td>
</tr>
</tbody>
</table>

**Part II: Getting Started with HDInsight and Understanding Its Different Components**

<table>
<thead>
<tr>
<th>HOUR</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Getting Started with HDInsight, Provisioning Your HDInsight Service Cluster, and Automating HDInsight Cluster Provisioning</td>
<td>179</td>
</tr>
<tr>
<td>7</td>
<td>Exploring Typical Components of HDFS Cluster</td>
<td>219</td>
</tr>
<tr>
<td>8</td>
<td>Storing Data in Microsoft Azure Storage Blob</td>
<td>233</td>
</tr>
<tr>
<td>9</td>
<td>Working with Microsoft Azure HDInsight Emulator</td>
<td>245</td>
</tr>
</tbody>
</table>

**Part III: Programming MapReduce and HDInsight Script Action**

<table>
<thead>
<tr>
<th>HOUR</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Programming MapReduce Jobs</td>
<td>255</td>
</tr>
<tr>
<td>11</td>
<td>Customizing the HDInsight Cluster with Script Action</td>
<td>277</td>
</tr>
</tbody>
</table>

**Part IV: Querying and Processing Big Data in HDInsight**

<table>
<thead>
<tr>
<th>HOUR</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Getting Started with Apache Hive and Apache Tez in HDInsight</td>
<td>287</td>
</tr>
<tr>
<td>13</td>
<td>Programming with Apache Hive, Apache Tez in HDInsight, and Apache HCatalog</td>
<td>323</td>
</tr>
<tr>
<td>14</td>
<td>Consuming HDInsight Data from Microsoft BI Tools over Hive ODBC Driver: Part 1</td>
<td>371</td>
</tr>
<tr>
<td>15</td>
<td>Consuming HDInsight Data from Microsoft BI Tools over Hive ODBC Driver: Part 2</td>
<td>385</td>
</tr>
</tbody>
</table>
16 Integrating HDInsight with SQL Server Integration Services .......... 423
17 Using Pig for Data Processing ............................................... 433
18 Using Sqoop for Data Movement Between RDBMS and HDInsight ... 445

Part V: Managing Workflow and Performing Statistical Computing
HOUR 19 Using Oozie Workflows and Job Orchestration with HDInsight ... 455
20 Performing Statistical Computing with R ........................................ 467

Part VI: Performing Interactive Analytics and Machine Learning
HOUR 21 Performing Big Data Analytics with Spark .................. 479
22 Microsoft Azure Machine Learning .......................................... 491

Part VII: Performing Real-time Analytics
HOUR 23 Performing Stream Analytics with Storm ......................... 519
24 Introduction to Apache HBase on HDInsight .......................... 535

Index ......................................................................................... 553
Table of Contents

Introduction 1

Part I: Understanding Big Data, Hadoop 1.0, and 2.0

HOUR 1: Introduction of Big Data, NoSQL, and Business Value Proposition 9
  Types of Analysis 9
  Types of Data 11
  Big Data 12
  Managing Big Data 15
  NoSQL Systems 17
  Big Data, NoSQL Systems, and the Business Value Proposition 26
  Application of Big Data and Big Data Solutions 27
  Summary 29
  Q&A 29

HOUR 2: Introduction to Hadoop, Its Architecture, Ecosystem, and Microsoft Offerings 31
  What Is Apache Hadoop? 31
  Architecture of Hadoop and Hadoop Ecosystems 33
  What’s New in Hadoop 2.0 42
  Architecture of Hadoop 2.0 43
  Tools and Technologies Needed with Big Data Analytics 46
  Major Players and Vendors for Hadoop 50
  Deployment Options for Microsoft Big Data Solutions 57
  Summary 61
  Q&A 61

HOUR 3: Hadoop Distributed File System Versions 1.0 and 2.0 65
  Introduction to HDFS 65
  HDFS Architecture 67
  Rack Awareness 88
<table>
<thead>
<tr>
<th>Chapter Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>WebHDFS</td>
<td>89</td>
</tr>
<tr>
<td>Accessing and Managing HDFS Data</td>
<td>90</td>
</tr>
<tr>
<td>What's New in HDFS 2.0</td>
<td>95</td>
</tr>
<tr>
<td>Summary</td>
<td>107</td>
</tr>
<tr>
<td>Q&amp;A</td>
<td>108</td>
</tr>
<tr>
<td><strong>HOUR 4: The MapReduce Job Framework and Job Execution Pipeline</strong></td>
<td>113</td>
</tr>
<tr>
<td>Introduction to MapReduce</td>
<td>113</td>
</tr>
<tr>
<td>MapReduce Architecture</td>
<td>116</td>
</tr>
<tr>
<td>MapReduce Job Execution Flow</td>
<td>119</td>
</tr>
<tr>
<td>Summary</td>
<td>140</td>
</tr>
<tr>
<td>Q&amp;A</td>
<td>140</td>
</tr>
<tr>
<td><strong>HOUR 5: MapReduce—Advanced Concepts and YARN</strong></td>
<td>145</td>
</tr>
<tr>
<td>DistributedCache</td>
<td>145</td>
</tr>
<tr>
<td>Hadoop Streaming</td>
<td>146</td>
</tr>
<tr>
<td>MapReduce Joins</td>
<td>148</td>
</tr>
<tr>
<td>Bloom Filter</td>
<td>149</td>
</tr>
<tr>
<td>Performance Improvement</td>
<td>150</td>
</tr>
<tr>
<td>Handling Failures</td>
<td>154</td>
</tr>
<tr>
<td>Counter</td>
<td>158</td>
</tr>
<tr>
<td>YARN</td>
<td>160</td>
</tr>
<tr>
<td>Uber-Tasking Optimization</td>
<td>170</td>
</tr>
<tr>
<td>Failures in YARN</td>
<td>171</td>
</tr>
<tr>
<td>Resource Manager High Availability and Automatic Failover in YARN</td>
<td>173</td>
</tr>
<tr>
<td>Summary</td>
<td>175</td>
</tr>
<tr>
<td>Q&amp;A</td>
<td>175</td>
</tr>
<tr>
<td><strong>Part II: Getting Started with HDInsight and Understanding Its Different Components</strong></td>
<td></td>
</tr>
<tr>
<td><strong>HOUR 6: Getting Started with HDInsight, Provisioning Your HDInsight Service Cluster, and Automating HDInsight Cluster Provisioning</strong></td>
<td>179</td>
</tr>
<tr>
<td>Introduction to Microsoft Azure</td>
<td>179</td>
</tr>
<tr>
<td>Understanding HDInsight Service</td>
<td>181</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Provisioning HDInsight on the Azure Management Portal</td>
<td>186</td>
</tr>
<tr>
<td>Automating HDInsight Provisioning with PowerShell</td>
<td>199</td>
</tr>
<tr>
<td>Managing and Monitoring HDInsight Cluster and Job Execution</td>
<td>212</td>
</tr>
<tr>
<td>Summary</td>
<td>214</td>
</tr>
<tr>
<td>Q&amp;A</td>
<td>214</td>
</tr>
<tr>
<td>Exercise</td>
<td>215</td>
</tr>
<tr>
<td><strong>HOUR 7: Exploring Typical Components of HDFS Cluster</strong></td>
<td>219</td>
</tr>
<tr>
<td>HDFS Cluster Components</td>
<td>219</td>
</tr>
<tr>
<td>HDInsight Cluster Architecture</td>
<td>223</td>
</tr>
<tr>
<td>High Availability in HDInsight</td>
<td>228</td>
</tr>
<tr>
<td>Summary</td>
<td>231</td>
</tr>
<tr>
<td>Q&amp;A</td>
<td>231</td>
</tr>
<tr>
<td><strong>HOUR 8: Storing Data in Microsoft Azure Storage Blob</strong></td>
<td>233</td>
</tr>
<tr>
<td>Understanding Storage in Microsoft Azure</td>
<td>233</td>
</tr>
<tr>
<td>Benefits of Azure Storage Blob over HDFS</td>
<td>234</td>
</tr>
<tr>
<td>Azure Storage Explorer Tools</td>
<td>235</td>
</tr>
<tr>
<td>Summary</td>
<td>243</td>
</tr>
<tr>
<td>Q&amp;A</td>
<td>243</td>
</tr>
<tr>
<td><strong>HOUR 9: Working with Microsoft Azure HDInsight Emulator</strong></td>
<td>245</td>
</tr>
<tr>
<td>Getting Started with HDInsight Emulator</td>
<td>245</td>
</tr>
<tr>
<td>Setting Up Microsoft Azure Emulator for Storage</td>
<td>251</td>
</tr>
<tr>
<td>Summary</td>
<td>253</td>
</tr>
<tr>
<td>Q&amp;A</td>
<td>254</td>
</tr>
<tr>
<td><strong>Part III: Programming MapReduce and HDInsight Script Action</strong></td>
<td></td>
</tr>
<tr>
<td><strong>HOUR 10: Programming MapReduce Jobs</strong></td>
<td>255</td>
</tr>
<tr>
<td>MapReduce Hello World!</td>
<td>255</td>
</tr>
<tr>
<td>Analyzing Flight Delays with MapReduce</td>
<td>258</td>
</tr>
<tr>
<td>Serialization Frameworks for Hadoop</td>
<td>262</td>
</tr>
<tr>
<td>Hadoop Streaming</td>
<td>270</td>
</tr>
<tr>
<td>Summary</td>
<td>276</td>
</tr>
<tr>
<td>Q&amp;A</td>
<td>276</td>
</tr>
</tbody>
</table>
HOUR 11: Customizing the HDInsight Cluster with Script Action 277
  Identifying the Need for Cluster Customization ........................................ 277
  Developing Script Action ........................................................................ 278
  Consuming Script Action ........................................................................ 280
  Running a Giraph job on a Customized HDInsight Cluster ...................... 282
  Testing Script Action with HDInsight Emulator ..................................... 285
  Summary .............................................................................................. 285
  Q&A ..................................................................................................... 285

Part IV: Querying and Processing Big Data in HDInsight

HOUR 12: Getting Started with Apache Hive and Apache Tez in HDInsight 287
  Introduction to Apache Hive ................................................................... 287
  Getting Started with Apache Hive in HDInsight ...................................... 288
  Azure HDInsight Tools for Visual Studio ................................................. 302
  Programmatically Using the HDInsight .NET SDK .................................. 311
  Introduction to Apache Tez .................................................................... 316
  Summary .............................................................................................. 321
  Q&A ..................................................................................................... 321
  Exercise ................................................................................................. 322

HOUR 13: Programming with Apache Hive, Apache Tez in HDInsight, and Apache HCatalog 323
  Programming with Hive in HDInsight ..................................................... 323
  Using Tables in Hive .............................................................................. 331
  Serialization and Deserialization ............................................................. 340
  Data Load Processes for Hive Tables ...................................................... 343
  Querying Data from Hive Tables ............................................................ 346
  Indexing in Hive .................................................................................... 363
  Apache Tez in Action ............................................................................. 365
  Apache HCatalog ................................................................................... 368
  Summary .............................................................................................. 370
  Q&A ..................................................................................................... 370
  Exercise ................................................................................................. 370
# Table of Contents

## HOUR 14: Consuming HDInsight Data from Microsoft BI Tools over Hive ODBC Driver: Part 1
- Introduction to Hive ODBC Driver ........................................................... 371
- Introduction to Microsoft Power BI ........................................................... 378
- Accessing Hive Data from Microsoft Excel ................................................ 379
- Summary .............................................................................................. 384
- Q&A ..................................................................................................... 384

## HOUR 15: Consuming HDInsight Data from Microsoft BI Tools over Hive ODBC Driver: Part 2
- Accessing Hive Data from PowerPivot ...................................................... 385
- Accessing Hive Data from SQL Server ...................................................... 409
- Accessing HDInsight Data from Power Query ........................................... 417
- Summary .............................................................................................. 421
- Q&A ..................................................................................................... 421
- Exercise ................................................................................................. 422

## HOUR 16: Integrating HDInsight with SQL Server Integration Services
- The Need for Data Movement ................................................................. 423
- Introduction to SSIS ................................................................................ 424
- Analyzing On-time Flight Departure with SSIS .......................................... 424
- Provisioning HDInsight Cluster ............................................................... 427
- Summary .............................................................................................. 431
- Q&A ..................................................................................................... 431

## HOUR 17: Using Pig for Data Processing
- Introduction to Pig Latin ......................................................................... 433
- Using Pig to Count Cancelled Flights ....................................................... 434
- Using HCatalog in a Pig Latin Script ........................................................ 438
- Submitting Pig Jobs with PowerShell ........................................................ 440
- Summary .............................................................................................. 443
- Q&A ..................................................................................................... 443

## HOUR 18: Using Sqoop for Data Movement Between RDBMS and HDInsight
- What Is Sqoop? ...................................................................................... 445
- Using Sqoop Import and Export Commands ............................................. 448
About the Authors

Arshad Ali has more than 13 years of experience in the computer industry. As a DB/DW/BI consultant in an end-to-end delivery role, he has been working on several enterprise-scale data warehousing and analytics projects for enabling and developing business intelligence and analytic solutions. He specializes in database, data warehousing, and business intelligence/analytics application design, development, and deployment at the enterprise level. He frequently works with SQL Server, Microsoft Analytics Platform System (APS, or formally known as SQL Server Parallel Data Warehouse [PDW]), HDInsight (Hadoop, Hive, Pig, HBase, and so on), SSIS, SSRS, SSAS, Service Broker, MDS, DQS, SharePoint, and PPS. In the past, he has also handled performance optimization for several projects, with significant performance gain.

Arshad is a Microsoft Certified Solutions Expert (MCSE)–SQL Server 2012 Data Platform, and Microsoft Certified IT Professional (MCITP) in Microsoft SQL Server 2008–Database Development, Data Administration, and Business Intelligence. He is also certified on ITIL 2011 foundation.

He has worked in developing applications in VB, ASP, .NET, ASP.NET, and C#. He is a Microsoft Certified Application Developer (MCAD) and Microsoft Certified Solution Developer (MCSD) for the .NET platform in Web, Windows, and Enterprise.

Arshad has presented at several technical events and has written more than 200 articles related to DB, DW, BI, and BA technologies, best practices, processes, and performance optimization techniques on SQL Server, Hadoop, and related technologies. His articles have been published on several prominent sites.

On the educational front, Arshad holds a Master in Computer Applications degree and a Master in Business Administration in IT degree.

Arshad can be reached at arshad.ali@live.in, or visit http://arshadali.blogspot.in/ to connect with him.

Manpreet Singh is a consultant and author with extensive expertise in architecture, design, and implementation of business intelligence and Big Data analytics solutions. He is passionate about enabling businesses to derive valuable insights from their data.

Manpreet has been working on Microsoft technologies for more than 8 years, with a strong focus on Microsoft Business Intelligence Stack, SharePoint BI, and Microsoft’s Big
Data Analytics Platforms (Analytics Platform System and HDInsight). He also specializes in Mobile Business Intelligence solution development and has helped businesses deliver a consolidated view of their data to their mobile workforces.

Manpreet has coauthored books and technical articles on Microsoft technologies, focusing on the development of data analytics and visualization solutions with the Microsoft BI Stack and SharePoint. He holds a degree in computer science and engineering from Panjab University, India.

Manpreet can be reached at manpreet.singh3@hotmail.com.
Dedications

Arshad:
To my parents, the late Mrs. and Mr. Md Azal Hussain, who brought me into this beautiful world and made me the person I am today. Although they couldn’t be here to see this day, I am sure they must be proud, and all I can say is, “Thanks so much—I love you both.”

And to my beautiful wife, Shazia Arshad Ali, who motivated me to take up the challenge of writing this book and who supported me throughout this journey.

And to my nephew, Gulfam Hussain, who has been very excited for me to be an author and has been following up with me on its progress regularly and supporting me, where he could, in completing this book.

Finally, I would like to dedicate this to my school teacher, Sankar Sarkar, who shaped my career with his patience and perseverance and has been truly an inspirational source.

Manpreet:
To my parents, my wife, and my daughter. And to my grandfather, Capt. Jagat Singh, who couldn’t be here to see this day.
Acknowledgments

This book would not have been possible without support from some of our special friends. First and foremost, we would like to thank Yaswant Vishwakarma, Vijay Korapadi, Avadhut Kulkarni, Kuldeep Chauhan, Rajeev Gupta, Vivek Adhola, and many others who have been inspirations and supported us in writing this book, directly or indirectly. Thanks a lot, guys—we are truly indebted to you all for all your support and the opportunity you have given us to learn and grow.

We also would like to thank the entire Pearson team, especially Mark Renfrow and Joan Murray, for taking our proposal from dream to reality. Thanks also to Shayne Burgess and Ron Abella for reading the entire draft of the book and providing very helpful feedback and suggestions.

Thanks once again—you all rock!

Arshad

Manpreet
We Want to Hear from You!

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we’re doing right, what we could do better, what areas you’d like to see us publish in, and any other words of wisdom you’re willing to pass our way.

We welcome your comments. You can email or write to let us know what you did or didn’t like about this book—as well as what we can do to make our books better.

Please note that we cannot help you with technical problems related to the topic of this book.

When you write, please be sure to include this book’s title and authors as well as your name and email address. We will carefully review your comments and share them with the authors and editors who worked on the book.

Email: consumer@samspublishing.com
Mail: Sams Publishing
     ATTN: Reader Feedback
     800 East 96th Street
     Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.
Introduction

“The information that’s stored in our databases and spreadsheets cannot speak for itself. It has important stories to tell and only we can give them a voice.” —Stephen Few

Hello, and welcome to the world of Big Data! We are your authors, Arshad Ali and Manpreet Singh. For us, it’s a good sign that you’re actually reading this introduction (so few readers of tech books do, in our experiences). Perhaps your first question is, “What’s in it for me?” We are here to give you those details with minimal fuss.

Never has there been a more exciting time in the world of data. We are seeing the convergence of significant trends that are fundamentally transforming the industry and ushering in a new era of technological innovation in areas such as social, mobility, advanced analytics, and machine learning. We are witnessing an explosion of data, with an entirely new scale and scope to gain insights from. Recent estimates say that the total amount of digital information in the world is increasing 10 times every 5 years. Eighty-five percent of this data is coming from new data sources (connected devices, sensors, RFIDs, web blogs, clickstreams, and so on), and up to 80 percent of this data is unstructured. This presents a huge opportunity for an organization: to tap into this new data to identify new opportunity and areas for innovation.

To store and get insight into this humongous volume of different varieties of data, known as Big Data, an organization needs tools and technologies. Chief among these is Hadoop, for processing and analyzing this ambient data born outside the traditional data processing platform. Hadoop is the open source implementation of the MapReduce parallel computational engine and environment, and it’s used quite widely in processing streams of data that go well beyond even the largest enterprise data sets in size. Whether it’s sensor, clickstream, social media, telemetry, location based, or other data that is generated and collected in large volumes, Hadoop is often on the scene to process and analyze it.

Analytics has been in use (mostly with organizations’ internal data) for several years now, but its use with Big Data is yielding tremendous opportunities. Organizations can now leverage data available externally in different formats, to identify new opportunities and areas of innovation by analyzing patterns, customer responses or behavior, market trends, competitors’ take, research data from governments or organizations, and more. This provides an opportunity to not only look back on the past, but also look forward to understand what might happen in the future, using predictive analytics.
In this book, we examine what constitutes Big Data and demonstrate how organizations can tap into Big Data using Hadoop. We look at some important tools and technologies in the Hadoop ecosystem and, more important, check out Microsoft’s partnership with Hortonworks/Cloudera. The Hadoop distribution for the Windows platform or on the Microsoft Azure Platform (cloud computing) is an enterprise-ready solution and can be integrated easily with Microsoft SQL Server, Microsoft Active Directory, and System Center. This makes it dramatically simpler, easier, more efficient, and more cost effective for your organization to capitalize on the opportunity Big Data brings to your business. Through deep integration with Microsoft Business Intelligence tools (PowerPivot and Power View) and EDW tools (SQL Server and SQL Server Parallel Data Warehouse), Microsoft’s Big Data solution also offers customers deep insights into their structured and unstructured data with the tools they use every day.

This book primarily focuses on the Hadoop (Hadoop 1.* and Hadoop 2.*) distribution for Azure, Microsoft HDInsight. It provides several advantages over running a Hadoop cluster over your local infrastructure. In terms of programming MapReduce jobs or Hive or PIG queries, you will see no differences; the same program will run flawlessly on either of these two Hadoop distributions (or even on other distributions), or with minimal changes, if you are using cloud platform-specific features. Moreover, integrating Hadoop and cloud computing significantly lessens the total cost ownership and delivers quick and easy setup for the Hadoop cluster. (We demonstrate how to set up a Hadoop cluster on Microsoft Azure in Hour 6, “Getting Started with HDInsight, Provisioning Your HDInsight Service Cluster, and Automating HDInsight Cluster Provisioning.”)

Consider some forecasts from notable research analysts or research organizations:

“Big Data is a Big Priority for Customers—49% of top CEOs and CIOs are currently using Big Data for customer analytics.”—McKinsey & Company, McKinsey Global Survey Results, Minding Your Digital Business, 2012

“By 2015, 4.4 million IT jobs globally will be created to support Big Data, generating 1.9 million IT jobs in the United States. Only one third of skill sets will be available by that time.”—Peter Sondergaard, Senior Vice President at Gartner and Global Head of Research

“By 2015, businesses (organizations that are able to take advantage of Big Data) that build a modern information management system will outperform their peers financially by 20 percent.”—Gartner, Mark Beyer, Information Management in the 21st Century

“By 2020, the amount of digital data produced will exceed 40 zettabytes, which is the equivalent of 5,200GB of data for every man, woman, and child on Earth.”—Digital Universe study

IDC has published an analysis predicting that the market for Big Data will grow to over $19 billion by 2015. This includes growth in partner services to $6.5 billion in 2015 and growth in software to $4.6 billion in 2015. This represents 39 percent and 34 percent compound annual growth rates, respectively.
Who Should Read This Book

What do you hope to get out of this book? As we wrote this book, we had the following audiences in mind:

- **Developers**—Developers (especially business intelligence developers) worldwide are seeing a growing need for practical, step-by-step instruction in processing Big Data and performing advanced analytics to extract actionable insights. This book was designed to meet that need. It starts at the ground level and builds from there, to make you an expert. Here you'll learn how to build the next generation of apps that include such capabilities.

- **Data scientists**—As a data scientist, you are already familiar with the processes of acquiring, transforming, and integrating data into your work and performing advanced analytics. This book introduces you to modern tools and technologies (ones that are prominent, inexpensive, flexible, and open source friendly) that you can apply while acquiring, transforming, and integrating Big Data and performing advanced analytics. By the time you complete this book, you’ll be quite comfortable with the latest tools and technologies.

- **Business decision makers**—Business decision makers around the world, from many different organizations, are looking to unlock the value of data to gain actionable insights that enable their businesses to stay ahead of competitors. This book delves into advanced analytics applications and case studies based on Big Data tools and technologies, to accelerate your business goals.

- **Students aspiring to be Big Data analysts**—As you are getting ready to transition from the academic to the corporate world, this book helps you build a foundational skill set to ace your interviews and successfully deliver Big Data projects in a timely manner. Chapters were designed to start at the ground level and gradually take you to an expert level.

Don’t worry if you don’t fit into any of these classifications. Set your sights on learning as much as you can and having fun in the process, and you’ll do fine!
How This Book Is Organized

This book begins with the premise that you can learn what Big Data is, including the real-life applications of Big Data and the prominent tools and technologies to use Big Data solutions to quickly tap into opportunity, by studying the material in 24 1-hour sessions. You might use your lunch break as your training hour, or you might study for an hour before you go to bed at night.

Whatever schedule you adopt, these are the hour-by-hour details on how we structured the content:

- Hour 1, “Introduction of Big Data, NoSQL, and Business Value Proposition,” introduces you to the world of Big Data and explains how an organization that leverages the power of Big Data analytics can both remain competitive and beat out its competitors. It explains Big Data in detail, along with its characteristics and the types of analysis (descriptive, predictive, and prescriptive) an organization does with Big Data. Finally, it sets out the business value proposition of using Big Data solutions, along with some real-life examples of Big Data solutions.

  This hour also summarizes the NoSQL technologies used to manage and process Big Data and explains how NoSQL systems differ from traditional database systems (RDBMS).

- In Hour 2, “Introduction to Hadoop, Its Architecture, Ecosystem, and Microsoft Offerings,” you look at managing Big Data with Apache Hadoop. This hour is rooted in history: It shows how Hadoop evolved from infancy to Hadoop 1.0 and then Hadoop 2.0, highlighting architectural changes from Hadoop 1.0 to Hadoop 2.0. This hour also focuses on understanding other software and components that make up the Hadoop ecosystem and looks at the components needed in different phases of Big Data analytics. Finally, it introduces you to Hadoop vendors, evaluates their offerings, and analyzes Microsoft’s deployment options for Big Data solutions.

- In Hour 3, “Hadoop Distributed File System Versions 1.0 and 2.0,” you learn about HDFS, its architecture, and how data gets stored. You also look into the processes of reading from HDFS and writing data to HDFS, as well as internal behavior to ensure fault tolerance. At the end of the hour, you take a detailed look at HDFS 2.0, which comes as a part of Hadoop 2.0, to see how it overcomes the limitations of Hadoop 1.0 and provides high-availability and scalability enhancements.

- In Hour 4, “The MapReduce Job Framework and Job Execution Pipeline,” you explore the MapReduce programming paradigm, its architecture, the components of a MapReduce job, and MapReduce job execution flow.

- Hour 5, “MapReduce—Advanced Concepts and YARN,” introduces you to advanced concepts related to MapReduce (including MapReduce Streaming, MapReduce joins, distributed caches, failures and how they are handled transparently, and performance optimization for your MapReduce jobs).
In Hadoop 2.0, YARN ushers in a major architectural change and opens a new window for scalability, performance, and multitenancy. In this hour, you learn about the YARN architecture, its components, the YARN job execution pipeline, and how failures are handled transparently.

In Hour 6, “Getting Started with HDInsight, Provisioning Your HDInsight Service Cluster, and Automating HDInsight Cluster Provisioning,” you delve into the HDInsight service. You also walk through a step-by-step process for quickly provisioning HDInsight or a Hadoop cluster on Microsoft Azure, either interactively using Azure Management Portal or automatically using PowerShell scripting.

In Hour 7, “Exploring Typical Components of HDFS Cluster,” you explore the typical components of an HDFS cluster: the name node, secondary name node, and data nodes. You also learn how HDInsight separates the storage from the cluster and relies on Azure Storage Blob instead of HDFS as the default file system for storing data. This hour provides more details on these concepts in the context of the HDInsight service.

In Hour 8, “Storing Data in Microsoft Azure Storage Blob,” shows you how HDInsight supports both the Hadoop Distributed File System (HDFS) and Azure Storage Blob for storing user data (although HDInsight relies on Azure storage blob as the default file system instead of HDFS for storing data). This hour explores Azure Storage Blob in the context of HDInsight and concludes by discussing the impact of blob storage on performance and data locality.

In Hour 9, “Working with Microsoft Azure HDInsight Emulator,” is devoted to Microsoft’s HDInsight emulator. HDInsight emulator emulates a single-node cluster and is well suited to development scenarios and experimentation. This hour focuses on setting up the HDInsight emulator and executing a MapReduce job to test its functionality.

In Hour 10, “Programming MapReduce Jobs,” expands on the content in earlier hours and provides examples and techniques for programming MapReduce programs in Java and C#. It presents a real-life scenario that analyzes flight delays with MapReduce and concludes with a discussion on serialization options for Hadoop.

In Hour 11, “Customizing the HDInsight Cluster with Script Action,” looks at the HDInsight cluster that comes preinstalled with a number of frequently used components. It also introduces customization options for the HDInsight cluster and walks you through the process for installing additional Hadoop ecosystem projects using a feature called Script Action. In addition, this hour introduces the HDInsight Script Action feature and illustrates the steps in developing and deploying a Script Action.

In Hour 12, “Getting Started with Apache Hive and Apache Tez in HDInsight,” you learn about how you can use Apache Hive. You learn different ways of writing and executing
HiveQL queries in HDInsight and see how Apache Tez significantly improves overall performance for HiveQL queries.

- In Hour 13, “Programming with Apache Hive, Apache Tez in HDInsight, and Apache HCatalog,” you extend your expertise on Apache Hive and see how you can leverage it for ad hoc queries and data analysis. You also learn about some of the important commands you will use in Apache Hive for data loading and querying. At the end this hour, you look at Apache HCatalog, which has merged with Apache Hive, and see how to leverage the Apache Tez execution engine for Hive query execution to improve the performance of your query.

- Hour 14, “Consuming HDInsight Data from Microsoft BI Tools over Hive ODBC Driver: Part 1,” shows you how to use the Microsoft Hive ODBC driver to connect and pull data from Hive tables from different Microsoft Business Intelligence (MSBI) reporting tools, for further analysis and ad hoc reporting.

- In Hour 15, “Consuming HDInsight Data from Microsoft BI Tools over Hive ODBC Driver: Part 2,” you learn to use PowerPivot to create a data model (define relationships between them, apply transformations, create calculations, and more) based on Hive tables and then use Power View and Power Map to visualize the data from different perspectives with intuitive and interactive visualization options.

- In Hour 16, “Integrating HDInsight with SQL Server Integration Services,” you see how you can use SQL Server Integration Services (SSIS) to build data integration packages to transfer data between an HDInsight cluster and a relational database management system (RDBMS) such as SQL Server.

- Hour 17, “Using Pig for Data Processing,” explores Pig Latin, a workflow-style procedural language that makes it easier to specify transformation operations on data. This hour provides an introduction to Pig for processing Big Data sets and illustrates the steps in submitting Pig jobs to the HDInsight cluster.

- Hour 18, “Using Sqoop for Data Movement Between RDBMS and HDInsight,” demonstrates how Sqoop facilitates data migration between relational databases and Hadoop. This hour introduces you to the Sqoop connector for Hadoop and illustrates its use in data migration between Hadoop and SQL Server/SQL Azure databases.

- Hour 19, “Using Oozie Workflows and Job Orchestration with HDInsight,” looks at data processing solutions that require multiple jobs chained together in particular sequence to accomplish a processing task in the form of a conditional workflow. In this hour, you learn to use Oozie, a workflow development component within the Hadoop ecosystem.

- Hour 20, “Performing Statistical Computing with R,” focuses on the R language, which is popular among data scientists for analytics and statistical computing. R was not designed to work with Big Data because it typically works by pulling data that persists elsewhere.
System Requirements

into memory. However, recent advancements have made it possible to leverage R for Big Data analytics. This hour introduces R and looks at the approaches for enabling R on Hadoop.

- Hour 21, “Performing Big Data Analytics with Spark,” introduces Spark, briefly explores the Spark programming model, and takes a look at Spark integration with SQL.

- In Hour 22, “Microsoft Azure Machine Learning,” you learn about an emerging technology known as Microsoft Azure Machine Learning (Azure ML). Azure ML is extremely simple to use and easy to implement so that analysts with various backgrounds (even nondata scientists) can leverage it for predictive analytics.

- In Hour 23, “Performing Stream Analytics with Storm,” you learn about Apache Storm and explore its use in performing real-time Stream analytics.

- Hour 24, “Introduction to Apache HBase on HDInsight,” you learn about Apache HBase, when to use it, and how you can leverage it with HDInsight service.

Conventions Used in This Book

In our experience as authors and trainers, we’ve found that many readers and students skip over this part of the book. Congratulations for reading it! Doing so will pay big dividends because you’ll understand how and why we formatted this book the way we did.

Try It Yourself

Throughout the book, you’ll find Try It Yourself exercises, which are opportunities for you to apply what you’re learning right then and there. I believe in knowledge stacking, so you can expect that later Try It Yourself exercises assume that you know how to do stuff you did in previous exercises. Therefore, your best bet is to read each chapter in sequence and work through every Try It Yourself exercise.

System Requirements

You don’t need a lot, computer wise, to perform all the Try It Yourself exercises in this book. However, if you don’t meet the necessary system requirements, you’re stuck. Make sure you have the following before you begin your work:

- A Windows-based computer—Technically, you don’t need a computer that runs only Microsoft Windows: Microsoft Azure services can be accessed and consumed using web browsers from any platform. However, if you want to use HDInsight emulator, you need to have a machine (virtual or physical) with the Microsoft Windows operating system.
An Internet connection—Microsoft HDInsight service is available on the cloud platform, so while you are working with it, you’ll be accessing the web.

An Azure subscription—You need an Azure subscription to use the platform or services available in Azure. Microsoft offers trial subscriptions of the Microsoft Azure subscription service used for learning or evaluation purposes.

Okay, that’s enough of the preliminaries. It’s time to get started on the Big Data journey and learn Big Data analytics with HDInsight. Happy reading!
This page intentionally left blank
What You’ll Learn in This Hour:

- Introduction to HDFS
- HDFS Architecture
- Rack Awareness
- WebHDFS
- Accessing and Managing HDFS Data
- What’s New in HDFS 2.0

In this hour, you take a detailed look at the Hadoop Distributed File System (HDFS), one of the core components of Hadoop for storing data in a distributed manner in the Hadoop cluster. You look at its architecture and how data gets stored. You check out its processes of reading from HDFS and writing data to HDFS, as well as its internal behavior to ensure fault tolerance. In addition, you delve into HDFS 2.0, which comes as a part of Hadoop 2.0, to see how it overcomes the limitations of Hadoop 1.0 and provides high availability and scalability enhancements.

Introduction to HDFS

The Hadoop Distributed File System (HDFS) is based on the Google File System (GFS) and written entirely in Java. Google provided only a white paper, without any implementation; however, around 90 percent of the GFS architecture has been applied in its implementation in the form of HDFS.

HDFS is a highly scalable, distributed, load-balanced, portable, and fault-tolerant (with built-in redundancy at the software level) storage component of Hadoop. In other words, HDFS is the underpinnings of the Hadoop cluster. It provides a distributed, fault-tolerant storage layer for storing Big Data in a traditional, hierarchical file organization of directories and files. HDFS has been designed to run on commodity hardware.

HDFS was originally built as a storage infrastructure for the Apache Nutch web search engine project. It was initially called the Nutch Distributed File System (NDFS).
These were the assumptions and design goals when HDFS was implemented originally:

- **Horizontal scalability**—HDFS is based on a scale-out model and can scale up to thousands of nodes, for terabytes or petabytes of data. As the load increases, you can keep increasing nodes (or data nodes) for additional storage and more processing power.

- **Fault tolerance**—HDFS assumes that failures (hardware and software) are common and transparently ensures data redundancy (by default, creating three copies of data: two copies on the same rack and one copy on a different rack so that it can survive even rack failure) to provide fault-tolerant storage. If one copy becomes inaccessible or gets corrupted, the developers and administrators don’t need to worry about it—the framework itself takes care of it transparently.

  In other words, instead of relying on hardware to deliver high availability, the framework itself was designed to detect and handle failures at the application layer. Hence, it delivers a highly reliable and available storage service with automatic recovery from failure on top of a cluster of machines, even if the machines (disk, node, or rack) are prone to failure.

- **Capability to run on commodity hardware**—HDFS has lower upfront hardware costs because it runs on commodity hardware. This differs from RDBMS systems, which rely heavily on expensive proprietary hardware with scale-up capability for more storage and processing.

- **Write once, read many times**—HDFS is based on a concept of write once, read multiple times, with an assumption that once data is written, it will not be modified. Hence, HDFS focuses on retrieving the data in the fastest possible way. HDFS was originally designed for batch processing, but with Hadoop 2.0, it can be used even for interactive queries.

- **Capacity to handle large data sets**—HDFS works for small numbers of very large files to store large data sets needed by applications targeting HDFS. Hence, HDFS has been tuned to support files of a few gigabytes to those several terabytes in size.

- **Data locality**—Every slave node in the Hadoop cluster has a data node (storage component) and a TaskTracker (processing component). When you run a query or MapReduce job, the TaskTracker normally processes data at the node where the data exists, minimizing the need for data transfer across nodes and significantly improving job performance because of data locality. This comes from the fact that moving computation near data (especially when the size of the data set is huge) is much cheaper than actually moving data near computation—it minimizes the risk of network congestion and increases the overall throughput of the system.
HDFS Architecture

HDFS Architecture

HDFS has a master and slaves architecture in which the master is called the name node and slaves are called data nodes (see Figure 3.1). An HDFS cluster consists of a single name node that manages the file system namespace (or metadata) and controls access to the files by the client applications, and multiple data nodes (in hundreds or thousands) where each data node manages file storage and storage device attached to it.

HDFS file system namespace—HDFS uses traditional hierarchical file organization, in which any user or application can create directories and recursively store files inside these directories. This enables you to create a file, delete a file, rename a file, and move a file from one directory to another one.

For example, from the following information, you can conclude that a top-level directory named user contains two subdirectories named abc and xyz. You also know that each of these subdirectories contains one file named sampleone.txt in the abc subdirectory and one file named sampletwo.txt in the xyz subdirectory. This is just an example—in practice, a directory might contain several directories, and each of these directories might contain several files.

/user/abc/sampleone.txt
/user/xyz/sampletwo.txt

Streaming access—HDFS is based on the principle of “write once, read many times.” This supports streaming access to the data, and its whole focus is on reading the data in the fastest possible way (instead of focusing on the speed of the data write). HDFS has also been designed for batch processing more than interactive querying (although this has changed in Hadoop 2.0).

In other words, in HDFS, reading the complete data set in the fastest possible way is more important than taking the time to fetch a single record from the data set.

High throughput—HDFS was designed for parallel data storage and retrieval. When you run a job, it gets broken down into smaller units called tasks. These tasks are executed on multiple nodes (or data nodes) in parallel, and final results are merged to produce the final output. Reading data from multiple nodes in parallel significantly reduces the actual time to read data.

In the next section, you explore the HDFS architecture in Hadoop 1.0 and the improvements in Hadoop 2.0.
While storing a file, HDFS internally splits it into one or more blocks (chunks of 64MB, by default, which is configurable and can be changed at cluster level or when each file is created). These blocks are stored in a set of slaves, called data nodes, to ensure that parallel writes or reads can be done even on a single file. Multiple copies of each block are stored per replication factor (which is configurable and can be changed at the cluster level, or at file creation, or even at a later stage for a stored file) for making the platform fault tolerant.

The name node is also responsible for managing file system namespace operations, including opening, closing, and renaming files and directories. The name node records any changes to the file system namespace or its properties. The name node contains information related to the replication factor of a file, along with the map of the blocks of each individual file to data nodes where those blocks exist. Data nodes are responsible for serving read and write requests from the HDFS clients and perform operations such as block creation, deletion, and replication when the name node tells them to. Data nodes store and retrieve blocks when they are told to (by the client applications or by the name node), and they report back to the name node periodically with lists of blocks that they are storing, to keep the name node up to date on the current status.

A client application talks to the name node to get metadata information about the file system. It connects data nodes directly so that they can transfer data back and forth between the client and the data nodes.

NOTE
The client communicates with the name node to get only metadata; the actual data transfer happens between the client and the data nodes. The name node is not involved in the actual data transfer.
The name node and data node are pieces of software called daemons in the Hadoop world. A typical deployment has a dedicated high-end machine that runs only the name node daemon; the other machines in the cluster run one instance of the data node daemon apiece on commodity hardware. Next are some reasons you should run a name node on a high-end machine:

- The name node is a single point of failure. Make sure it has enough processing power and storage capabilities to handle loads. You need a scaled-up machine for a name node.

**NOTE**

The word *daemon* comes from the UNIX world and refers to a process or service that runs in the background. On a Windows platform, it is generally referred to as a service.

- The name node keeps metadata related to the file system namespace in memory, for quicker response time. Hence, more memory is needed.
- The name node coordinates with hundreds or thousands of data nodes and serves the requests coming from client applications.

As discussed earlier, HDFS is based on a traditional hierarchical file organization. A user or application can create directories or subdirectories and store files inside. This means that you can create a file, delete a file, rename a file, or move a file from one directory to another.

All this information, along with information related to data nodes and blocks stored in each of the data nodes, is recorded in the file system namespace, called fsimage and stored as a file on the local host OS file system at the name node daemon. This fsimage file is not updated with every addition or removal of a block in the file system. Instead, the name node logs and maintains these add/remove operations in a separate edit log file, which exists as another file on the local host OS file system. Appending updates to a separate edit log achieves faster I/O.

A secondary name node is another daemon. Contrary to its name, the secondary name node is not a standby name node, so it is not meant as a backup in case of name node failure. The primary purpose of the secondary name node is to periodically download the name node fsimage and edit the log file from the name node, create a new fsimage by merging the older fsimage and edit the log file, and upload the new fsimage back to the name node. By periodically merging the namespace fsimage with the edit log, the secondary name node prevents the edit log from becoming too large.
NOTE

Name Node: fsimage and Edit Log File

The fsimage and the edit log file are central data structures that contain HDFS file system metadata and namespaces. Any corruption of these files can cause the HDFS cluster instance to become nonfunctional. For this reason, the name node can be configured to support maintaining multiple copies of the fsimage and edit log to another machine. This is where the secondary name node comes into play.

The name node machine is a single point of failure, so manual intervention is needed if it fails. Hadoop 1.0 does not support the automatic restart and failover of the name node, but Hadoop 2.0 does. Refer to the section “What’s New in HDFS 2.0” later in this hour, for more details on changes.

The process of generating a new fsimage from a merge operation is called the Checkpoint process (see Figure 3.2). Usually the secondary name node runs on a separate physical machine than the name node; it also requires plenty of CPU and as much as memory as the name node to perform the Checkpoint operation.

![Figure 3.2](image)

Checkpoints process.

As you can see in Table 3.1, the core-site.xml configuration file for Hadoop 1.0 contains some configuration settings related to the Checkpoint process. You can change these configuration settings to change the Hadoop behavior. See Table 3.1 for a Checkpoint-related configuration example in Hadoop 1.0.
### Table 3.1 Checkpoint-Related Configuration in Hadoop 1.0

<table>
<thead>
<tr>
<th>Setting</th>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fs.checkpoint.dir</td>
<td>c:\hadoop\HDFS\2nn</td>
<td>Determines where on the local file system the DFS secondary name node should store the temporary images to merge. If this is a comma-delimited list of directories, the image is replicated in all the directories for redundancy.</td>
</tr>
<tr>
<td>fs.checkpoint.edits.dir</td>
<td>c:\hadoop\HDFS\2nn</td>
<td>Determines where on the local file system the DFS secondary name node should store the temporary edits to merge. If this is a comma-delimited list of directories, the edits are replicated in all the directories for redundancy. The default value is the same as fs.checkpoint.dir.</td>
</tr>
<tr>
<td>fs.checkpoint.period</td>
<td>86400</td>
<td>The number of seconds between two periodic Checkpoints.</td>
</tr>
<tr>
<td>fs.checkpoint.size</td>
<td>2048000000</td>
<td>The size of the current edit log (in bytes) that triggers a periodicCheckpoint even if the fs.checkpoint.period hasn't expired.</td>
</tr>
</tbody>
</table>

Table 3.2 shows some configuration settings related to the Checkpoint process that are available in the core-site.xml configuration file for Hadoop 2.0.

### Table 3.2 Checkpoint-Related Configuration in Hadoop 2.0

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dfs.namenode.checkpoint.dir</td>
<td>Determines where on the local file system the DFS secondary name node should store the temporary images to merge. If this is a comma-delimited list of directories, the image is replicated in all the directories for redundancy.</td>
</tr>
<tr>
<td>dfs.namenode.checkpoint.edits.dir</td>
<td>Determines where on the local file system the DFS secondary name node should store the temporary edits to merge. If this is a comma-delimited list of directories, the edits are replicated in all the directories for redundancy. The default value is the same as dfs.namenode.checkpoint.dir.</td>
</tr>
<tr>
<td>dfs.namenode.checkpoint.period</td>
<td>The number of seconds between two periodic checkpoints.</td>
</tr>
</tbody>
</table>
With the secondary name node performing this task periodically, the name node can restart relatively faster. Otherwise, the name node would need to do this merge operation when it restarted.

### NOTE

**Name Node Also Does Checkpoint**

The secondary name node periodically performs a Checkpoint operation by merging the namespace fsimage with the edit log and uploading it back to the name node. But the name node also goes through a Checkpoint operation when it starts up. It applies all the transactions from the edit log to the in-memory representation of the fsimage (the name node keeps the fsimage in memory, for quicker response). The name node then writes back this new version of fsimage to the disk. At this time, it can safely truncate the edit log file because it has already been merged.

The secondary name node is also responsible for backing up the name node fsimage (a copy of the merged fsimage), which is used if the primary name node fails. However, the state of the secondary name node lags that of the primary, so if the primary name node fails, data loss might occur.

### TIP

If your configuration keeps the name node’s metadata files (fsimage and edit log) on the Network File System (NFS), you can take a copy of these files (or point back to them from the secondary name node) to the secondary name node and then run that secondary name node as the primary name node.
File Split in HDFS
As discussed earlier, HDFS works best with small numbers of very large files for storing large data sets that the applications need. As you can see in Figure 3.3, while storing files, HDFS internally splits a file content into one or more data blocks (chunks of 64MB, by default, which is configurable and can be changed when needed at the cluster instance level for all the file writes or when each specific file is created). These data blocks are stored on a set of slaves called data nodes, to ensure a parallel data read or write.

All blocks of a file are the same size except the last block, which can be either the same size or smaller. HDFS stores each file as a sequence of blocks, with each block stored as a separate file in the local file system (such as NTFS).

Cluster-wide block size is controlled by the dfs.blocksize configuration property in the hdfs-site.xml file. The dfs.blocksize configuration property applies for files that are created without a block size specification. This configuration has a default value of 64MB and usually varies from 64MB to 128MB, although many installations now use 128MB. In Hadoop 2.0, the default block is 128MB (see Table 3.3). The block size can continue to grow as transfer speeds grow with new generations of disk drives.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dfs.blocksize</td>
<td>134217728</td>
<td>The default block size for new files, in bytes. You can use the following suffix (case insensitive): k (kilo), m (mega), g (giga), t (tera), p (peta), e (exa) to specify the size (such as 128k, 512m, 1g, and so on). Or, provide the complete size in bytes, such as 134217728 for 128MB.</td>
</tr>
</tbody>
</table>
NOTE

**Impact of Changing the `dfs.blocksize`**
When you change the `dfs.blocksize` configuration setting in the `hdfs-site.xml` configuration file for an existing cluster, it does not affect the already-written files (or blocks). This new setting is effective for new files only. If you want to make it applicable to already written files (or existing files or blocks from HDFS), you must write a logic to rewrite those files to HDFS again (this automatically picks up the new configuration and splits the blocks in size, as defined in the new configuration).

**Block Placement and Replication in HDFS**
You have already seen that each file is broken in multiple data blocks. Now you can explore how these data blocks get stored. By default, each block of a file is stored three times on three different data nodes: The replication factor configuration property has a default value of 3 (see Table 3.4).

<table>
<thead>
<tr>
<th>Block Replication Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>dfs.replication</td>
</tr>
<tr>
<td>dfs.replication.max</td>
</tr>
<tr>
<td>dfs.namenode.replication.min</td>
</tr>
</tbody>
</table>

When a file is created, an application can specify the number of replicas of each block of the file that HDFS must maintain. Multiple copies or replicas of each block makes it fault tolerant: If one copy is not accessible or gets corrupted, the data can be read from the other copy. The number of copies of each block is called the *replication factor* for a file, and it applies to all blocks of a file.

While writing a file, or even for an already stored file, an application can override the default replication factor configuration and specify another replication factor for that file. For example, the replication factor can be specified at file creation time and can be even changed later, when needed.

NOTE

**Replication Factor for a File**
An application or job can also specify the number of replicas of a file that HDFS should maintain. The number of copies or replicas of each block of a file is called the *replication factor* of that file.
The name node has the responsibility of ensuring that the number of copies or replicas of each block is maintained according to the applicable replication factor for each file. If necessary, it instructs the appropriate data nodes to maintain the defined replication factor for each block of a file.

Each data node in the cluster periodically sends a heartbeat signal and a block-report to the name node. When the name node receives the heartbeat signal, it implies that the data node is active and functioning properly. A block-report from a data node contains a list of all blocks on that specific data node.

A typical Hadoop installation spans hundreds or thousands of nodes. A collection of data nodes is placed in rack together for a physical organization, so you effectively have a few dozen racks. For example, imagine that you have 100 nodes in a cluster, and each rack can hold 5 nodes. You then have 20 racks to accommodate all the 100 nodes, each containing 5 nodes.

The simplest block placement solution is to place each copy or replica of a block in a separate rack. Although this ensures that data is not lost even in case of multiple rack failures and delivers an enhanced read operation by utilizing bandwidth from all the racks, it incurs a huge performance penalty when writing data to HDFS because a write operation must transfer blocks to multiple racks. Remember also that communication between data nodes across racks is much more expensive than communication across nodes in a single rack.

The other solution is to put together all the replicas in the different data nodes of a single rack. This scenario improves the write performance, but rack failure would result in total data loss.

To take care of this situation, HDFS has a balanced default block placement policy. Its objective is to have a properly load-balanced, fast-access, fault-tolerance file system:

- The first replica is written to the data node creating the file, to improve the write performance because of the write affinity.
- The second replica is written to another data node within the same rack, to minimize the cross-rack network traffic.
- The third replica is written to a data node in a different rack, ensuring that even if a switch or rack fails, the data is not lost. (This applies only if you have configured your cluster for rack awareness as discussed in the section “Rack Awareness” later in this hour.

You can see in Figure 3.4 that this default block placement policy cuts the cross-rack write traffic. It generally improves write performance without compromising on data reliability or availability, while still maintaining read performance.
The replication factor is an important configuration to consider. The default replication factor of 3 provides an optimum solution of both write and read performance while also ensuring reliability and availability. However, sometimes you need to change the replication factor configuration property or replication factor setting for a file. For example, you need to change the replication factor configuration to 1 if you have a single-node cluster.

For other cases, consider an example. Suppose you have some large files whose loss would be acceptable (for example, a file contains data older than 5 years, and you often do analysis over the last 5 years of data). Also suppose that you can re-create these files in case of data loss). You can set its replication factor to 1 to minimize the need for storage requirement and, of course, to minimize the time taken to write it.

You can even set the replication factor to 2, which requires double the storage space but ensures availability in case a data node fails (although it might not be helpful in case of a rack failure). You can change the replication factor to 4 or higher, which will eventually improve the performance of the read operation at the cost of a more expensive write operation, and with more storage space requirement to store another copies.

NOTE

When a Name Node Starts Re-replication

A heartbeat signal from a data node guarantees that the data node is available for serving requests. When a name node realizes that a data node has died, it stops sending any new I/O request to it and concludes that the replication factor of blocks stored on that specific data node has fallen below the specified value, based on its calculation. This swings the name node into action. The name node initiates replication for these blocks to ensure that the replication factor is maintained. Re-replication might be needed for several reasons. For example, a data node itself might have
stopped working, or a replica on a data node might have been corrupted. Alternatively, a storage device might have failed on the data node, the network connecting to the data node might have gone bad, or the replication factor of a file might have increased.

When you decrease the replication factor for a file already stored in HDFS, the name node determines, on the next heartbeat signal to it, the excess replica of the blocks of that file to be removed. It transfers this information back to the appropriate data nodes, to remove corresponding blocks and free up occupied storage space.

**Writing to HDFS**

As discussed earlier, when a client or application wants to write a file to HDFS, it reaches out to the name node with details of the file. The name node responds with details based on the actual size of the file, block, and replication configuration. These details from the name node contain the number of blocks of the file, the replication factor, and data nodes where each block will be stored (see Figure 3.5).

![Figure 3.5](image)

The client talks to the name node for metadata to specify where to place the data blocks.

Based on information received from the name node, the client or application splits the files into multiple blocks and starts sending them to the first data node. Normally, the first replica is written to the data node creating the file, to improve the write performance because of the write affinity.
TIP

The client or application directly transfers the data to the first data node; the name node is not involved in the actual data transfer (data blocks don’t pass through the name node). Along with the block of data, the HDFS client, application, or API sends information related to the other data nodes where each block needs to be stored, based on the replication factor.

As you see in Figure 3.6, Block A is transferred to data node 1 along with details of the two other data nodes where this block needs to be stored. When it receives Block A from the client (assuming a replication factor of 3), data node 1 copies the same block to the second data node (in this case, data node 2 of the same rack). This involves a block transfer via the rack switch because both of these data nodes are in the same rack. When it receives Block A from data node 1, data node 2 copies the same block to the third data node (in this case, data node 3 of another rack). This involves a block transfer via an out-of-rack switch along with a rack switch because both of these data nodes are in separate racks.

FIGURE 3.6

The client sends data blocks to identified data nodes.
NOTE

**Data Flow Pipeline**

For better performance, data nodes maintain a pipeline for data transfer. Data node 1 does not need to wait for a complete block to arrive before it can start transferring to data node 2 in the flow. In fact, the data transfer from the client to data node 1 for a given block happens in smaller chunks of 4KB. When data node 1 receives the first 4KB chunk from the client, it stores this chunk in its local repository and immediately starts transferring it to data node 2 in the flow. Likewise, when data node 2 receives first 4KB chunk from data node 1, it stores this chunk in its local repository and immediately starts transferring it to data node 3. This way, all the data nodes in the flow except the last one receive data from the previous one and transfer it to the next data node in the flow, to improve on the write performance by avoiding a wait time at each stage.

When all the instructed data nodes receive a block, each one sends a write confirmation to the name node (see Figure 3.7).

---

**FIGURE 3.7**
Data nodes update the name node about receipt of the data blocks.

---

Finally, the first data node in the flow sends the confirmation of the Block A write to the client (after all the data nodes send confirmation to the name node) (see Figure 3.8).
FIGURE 3.8
The first data node sends an acknowledgment back to the client.

NOTE
For simplicity, we demonstrated how one block from the client is written to different data nodes. But the whole process is actually repeated for each block of the file, and data transfer happens in parallel for faster write of blocks.

For example, Figure 3.9 shows how data block write state should look after transferring Blocks A, B, and C, based on file system namespace metadata from the name node to the different data nodes of the cluster. This continues for all other blocks of the file.
HDFS uses several optimization techniques. One is to use client-side caching, by the HDFS client, to improve the performance of the block write operation and to minimize network congestion. The HDFS client transparently caches the file into a temporary local file. When it accumulates data as big as a defined block size, the client reaches out to the name node.

At this time, the name node responds by inserting the filename into the file system hierarchy and allocating data nodes for its storage. The client flushes the block of data from the local temporary file to the closest data node and that data node creates copies of the block to other data nodes to maintain replication factor (as instructed by the name node based on the replication factor of the file).

When all the blocks of a file are transferred to the respective data nodes, the client tells the name node that the file is closed. The name node then commits the file creation operation to a persistent store.

**CAUTION**

Remember that if the name node dies before the file is closed, the file is lost and must be resent.
NOTE

**Communication Protocols**
All communication from clients to the name node, clients to data nodes, data nodes to the name node, and name node to the data nodes happens over Transmission Control Protocol/Internet Protocol (TCP/IP). The data nodes communicate with the name node using the data node protocol with its own TCP port number (configurable). The client communicates with the name node using the client protocol with its own TCP port number (configurable). By design, the name node does not initiate a remote procedure call (RPC); it only responds to the RPC requests coming from either data nodes or clients.

**Reading from HDFS**
To read a file from the HDFS, the client or application reaches out to the name node with the name of the file and its location. The name node responds with the number of blocks of the file, data nodes where each block has been stored (see Figure 3.10).

![Figure 3.10](image)

*FIGURE 3.10*
The client talks to the name node to get metadata about the file it wants to read.

Now the client or application reaches out to the data nodes directly (without involving the name node for actual data transfer—data blocks don’t pass through the name node) to read the blocks of the files in parallel, based on information received from the name node. When the client or application receives all the blocks of the file, it combines these blocks into the form of the original file (see Figure 3.11).
FIGURE 3.11
The client starts reading data blocks of the file from the identified data nodes.

To improve the read performance, HDFS tries to reduce bandwidth consumption by satisfying a read request from a replica that is closest to the reader. It looks for a block in the same node, then another node in the same rack, and then finally another data node in another rack. If the HDFS cluster spans multiple data centers, a replica that resides in the local data center (the closest one) is preferred over any remote replica from remote data center.

NOTE
Checksum for Data Blocks
When writing blocks of a file, the HDFS client computes the checksum of each block of the file and stores these checksums in a separate, hidden file in the same HDFS file system namespace. Later, while reading the blocks, the client references these checksums to verify that these blocks have not been corrupted. (Corruption can happen because of faults in a storage device, network transmission faults, or bugs in the program.) When the client realizes that a block is corrupt, it reaches out to another data node that has the replica of the corrupt block, to get another copy of the block.

Handling Failures
On cluster startup, the name node enters into a special state called safe mode. During this time, the name node receives a heartbeat signal (implying that the data node is active and functioning properly) and a block-report from each data node (containing a list of all blocks
on that specific data node) in the cluster. Figure 3.12 shows how all the data nodes of the cluster send a periodic heartbeat signal and block-report to the name node.

**FIGURE 3.12**
All data nodes periodically send heartbeat signals to the name node.

Based on the replication factor setting, each block has a specified minimum number of replicas to be maintained. A block is considered safely replicated when the number of replicas (based on replication factor) of that block has checked in with the name node. If the name node identifies blocks with less than the minimal number of replicas to be maintained, it prepares a list.

After this process, plus an additional few seconds, the name node exits safe mode state. Now the name node replicates these blocks (which have fewer than the specified number of replicas) to other data nodes.

Now let’s examine how the name node handles a data node failure. In Figure 3.13, you can see four data nodes (two data nodes in each rack) in the cluster. These data nodes periodically send heartbeat signals (implying that a particular data node is active and functioning properly) and a block-report (containing a list of all blocks on that specific data node) to the name node.
The name node updates its metadata based on information it receives from the data nodes.

The name node thus is aware of all the active or functioning data nodes of the cluster and what block each one of them contains. You can see that the file system namespace contains the information about all the blocks from each data node (see Figure 3.13).

Now imagine that data node 4 has stopped working. In this case, data node 4 stops sending heartbeat signals to the name node. The name node concludes that data node 4 has died. After a certain period of time, the name nodes concludes that data node 4 is not in the cluster anymore and that whatever data node 4 contained should be replicated or load-balanced to the available data nodes.

As you can see in Figure 3.14, the dead data node 4 contained blocks B and C, so name node instructs other data nodes, in the cluster that contain blocks B and C, to replicate it in manner; it is load-balanced and the replication factor is maintained for that specific block. The name node then updates its file system namespace with the latest information regarding blocks and where they exist now.
Delete Files from HDFS to Decrease the Replication Factor

By default, when you delete a file or a set of files from HDFS, the file(s) get deleted permanently and there is no way to recover it. But don’t worry: HDFS has a feature called Trash that you can enable to recover your accidently deleted file(s). As you can see in Table 3.5, this feature is controlled by two configuration properties: `fs.trash.interval` and `fs.trash.checkpoint.interval` in the `core-site.xml` configuration file.
TABLE 3.5  Trash-Related Configuration

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fs.trash.interval</td>
<td>The number of minutes after which the checkpoint gets deleted. If zero, the trash feature is disabled. This option can be configured on both the server and the client. If trash is disabled on the server side, the client-side configuration is checked. If trash is enabled on the server side, the value configured on the server is used and the client configuration value is ignored.</td>
</tr>
<tr>
<td>fs.trash.checkpoint.interval</td>
<td>The number of minutes between trash checkpoints. Should be smaller than or equal to fs.trash.interval. If zero, the value is set to the value of fs.trash.interval. Each time the checkpoint process runs, it creates a new checkpoint out of the current and removes checkpoints created more than fs.trash.interval minutes ago.</td>
</tr>
</tbody>
</table>

By default, the value for fs.trash.interval is 0, which signifies that trashing is disabled. To enable it, you can set it to any numeric value greater than 0, represented in minutes. This instructs HDFS to move your deleted files to the Trash folder for that many minutes before it can permanently delete them from the system. In other words, it indicates the time interval a deleted file will be made available to the Trash folder so that the system can recover it from there, either until it crosses the fs.trash.interval or until the next trash checkpoint occurs.

By default, the value for fs.trash.checkpoint.interval is also 0. You can set it to any numeric value, but it must be smaller than or equal to the value specified for fs.trash.interval. It indicates how often the trash checkpoint operation should run. During trash checkpoint operation, it checks for all the files older than the specified fs.trash.interval and deletes them. For example, if you have set fs.trash.interval to 120 and fs.trash.checkpoint.interval to 60, the trash checkpoint operation kicks in every 60 minutes to see if any files are older than 120 minutes. If so, it deletes those files permanently from the Trash folder.

When you decrease the replication factor for a file already stored in HDFS, the name node determines, on the next heartbeat signal to it, the excess replica of the blocks of that file to be removed. It transfers this information back to the appropriate data nodes, to remove corresponding blocks and free up occupied storage space.

TIP
In both cases just mentioned, a time delay occurs between when you delete a file or decrease the replication factor for a file and when storage space is actually reclaimed.
Rack Awareness

HDFS, MapReduce, and YARN (Hadoop 2.0) are rack-aware components. This means they can learn about all the nodes of the cluster rack they belong to and act accordingly. The third block from default block placement policy (see the section “Block Placement and Replication in HDFS”) applies only if you have configured rack awareness.

If you have not configured rack awareness (in this case, all nodes of the cluster are considered to be on the same rack), or if you have a small cluster with just one rack, one replica of the block goes to the local data node and two replicas go to another two data nodes, selected randomly from the cluster.

CAUTION

This should not be an issue if you have just one rack. However, a larger cluster with multiple racks faces the possibility that all replicas of a block end up in different data nodes in the same rack. This can cause data loss if that specific rack fails.

To avoid this problem, Hadoop (HDFS, MapReduce, and YARN) supports configuring rack awareness. This ensures that the third replica is written to a data node from another rack for better reliability and availability. Even if one rack is getting down, another rack then is available to serve the requests. This also increases the utilization of network bandwidth when reading data because data comes from multiple racks with multiple network switches.

Making Clusters Rack Aware

You can make a Hadoop cluster rack aware by using a script that enables the master node to map the network topology of the cluster. It does so using the properties topology.script.file.name or net.topology.script.file.name, available in the core-site.xml configuration file.

First, you need to change this property to specify the name of the script file. Then you write the script and place it in a file at the specified location. The script should accept a list of IP addresses and return the corresponding list of rack identifiers. For example, the script takes host.foo.bar as an argument and returns /rack1 as the output.

In other words, the script should be able to accept IP addresses or DNS names and return the rack identifier; it is a one-to-one mapping between what the script takes and what it returns. For retrieving the rack identifier, the script might deduce it from the IP address or query some service (similar to the way DNS works). The simplest way is to read from a file that has the mapping from IP address or DNS name to rack identifier.
For example, imagine that you have a mapping file with the following information, where
the first column represents the IP address or DNS name of the node and the second column
represents the rack it belongs to:

```
<table>
<thead>
<tr>
<th>Node</th>
<th>Rack</th>
</tr>
</thead>
<tbody>
<tr>
<td>hadoopdn001</td>
<td>/hadoop/rack1</td>
</tr>
<tr>
<td>hadoopdn002</td>
<td>/hadoop/rack1</td>
</tr>
<tr>
<td>hadoopdn006</td>
<td>/hadoop/rack2</td>
</tr>
<tr>
<td>hadoopdn007</td>
<td>/hadoop/rack2</td>
</tr>
</tbody>
</table>
```

Given this information, you can write a script that compares the IP address or DNS name of
the node with the first column; when they match, the script returns the value from the second
column of the corresponding row.

For example, based on the previous information, if you pass `hadoopdn001`, it should return
`/hadoop/rack1`; if you pass `hadoopdn006`, it should return `/hadoop/rack2`. Likewise, if you
pass `hadoopdn101`, it should return `/default/rack` because there is no entry for `hadoopdn101`
in the file.

**NOTE**

**Default Rack**

If the value for `net.topology.script.file.name` is not configured, the default value of
`/default-rack` is returned for any IP addresses; thus, all nodes are considered to be on the
same rack.

---

**WebHDFS**

As long as an application needs to access data stored in HDFS from inside a cluster or another
machine on the network, it can use a high-performance native protocol or native Java API and
be fine. But what if an external application wants to access or manage files in the HDFS over the
Internet or HTTP or the Web?

For these kinds of requirements, an additional protocol was developed. This protocol, called
WebHDFS, is based on an industry-standard RESTful mechanism that does not require Java
binding. It works with operations such as reading files, writing to files, making directories,
changing permissions, and renaming. It defines a public HTTP REST API, which permits clients to access HDFS over the Web. Clients can use common tools such as curl/wget to access the HDFS.

WebHDFS provides web services access to data stored in HDFS. At the same time, it retains the security the native Hadoop protocol offers and uses parallelism, for better throughput.

To enable WebHDFS (REST API) in the name node and data nodes, you must set the value of dfs.webhdfs.enabled configuration property to true in hdfs-site.xml configuration file as shown in the Figure 3.15.

```
<property>
    <name>dfs.webhdfs.enabled</name>
    <value>true</value>
</property>
```

FIGURE 3.15
WebHDFS-related configuration.

Accessing and Managing HDFS Data

You can access the files or data stored in HDFS in many different ways. For example, you can use HDFS FS shell commands, leverage the Java API available in the classes of the org.apache.hadoop.fs package (http://hadoop.apache.org/docs/current/api/org/apache/hadoop/fs/package-frame.html), write a MapReduce job, or write Hive or Pig queries. You can even use a web browser to browse the files from an HDFS cluster.

HDFS Command-Line Interface

The HDFS command-line interface (CLI), called FS Shell, enables you to write shell commands to manage your files in the HDFS cluster. It is useful when you need a scripting language to interact with the stored files and data. Figure 3.16 shows the hadoop fs command and the different parameter options you can use with it.
Some commonly used parameters with the `hadoop fs` command follow:

- `mkdir`—Creates a directory based on the passed URI.

**NOTE**

The URI, or uniform resource identifier, refers to the string of characters (compact representation) that identify the name of a resource available to your application on the intranet or Internet.

- `put`—Copies one or more files from the local file system (also reads input from stdin) to the destination file system.
- `copyToLocal`—Copies one or more files from the local file system to the destination file system. The `-f` option overwrites the destination if it already exists.
- `get`—Copies one or more files to the local file system.
- `copyToLocal`—Copies one or more files to the local file system.
HOUR 3: Hadoop Distributed File System Versions 1.0 and 2.0

- `ls`—Return statistics on the file or content of a directory.
- `lsr`—Same as `ls` but recursive in nature.
- `rm`—Deletes a file or directory. The directory must be empty to drop it.
- `rmr`—Same as `rm` but recursive in nature.

### TRY IT YOURSELF

#### Getting Started with HDFS Commands

In Listing 3.1, line 2 creates a subdirectory named `sampledata` under the `example` directory. Line 4 copies a local directory named Gutenberg and its content to the `sampledata` directory on HDFS. Likewise, line 9 copies one specific file from the local file system to the HDFS folder. Figure 3.17 shows folders from the local file system used with FS Shell commands to move files and folders to the HDFS cluster.

![Source folders and files to copy to HDFS.](image)

**LISTING 3.1** Example of HDFS Command-Line Interface

```
1: @rem --- creating a directory
2: hadoop fs -mkdir /example/sampledata
3: @rem --- copying a directory
4: hadoop fs -copyFromLocal c:\apps\dist\examples\data\gutenberg /example/sampledata
5: @rem --- listing content of the directory
6: hadoop fs -ls /example/sampledata
7: hadoop fs -ls /example/sampledata/gutenberg
8: @rem --- copying a file
9: hadoop fs -copyFromLocal c:\apps\dist\examples\data\sample\sample.log /example/sampledata
10: @rem --- listing content of the directory
```
Figure 3.18 shows files copied from HDFS to the local file system using the code from line 13 of Listing 3.1.

Figure 3.19 shows the execution result of the code from Listing 3.1.
NOTE
In Hadoop 2.0, instead of using hadoop fs with the FS Shell, you use hdfs dfs. For example, if you used hadoop fs -ls /example/sampledata to list the content from the sampledata directory in earlier versions of Hadoop, you need to use hdfs dfs -ls /example/sampledata instead.

You can learn more about shell commands of Hadoop 2.0 at http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/FileSystemShell.html.

NOTE
HDFS Administration
DFSAdmin is another CLI for administering an HDFS cluster. Only HDFS administrators use DFSAdmin. For example, hadoop dfsadmin -safemode enter puts the cluster in safe mode, hadoop dfsadmin -safemode leave brings the cluster back from safe mode, and hadoop dfsadmin -report reports basic file system information and statistics, such as how much disk is available and how many data nodes are running.

On Hadoop 2.0, you must use hdfs dfsadmin -report. If you don’t, you will see the message “DEPRECATED: Use of this script to execute hdfs command is deprecated”.

Using MapReduce, Hive, Pig, or Sqoop
FS Shell commands are good as long as you want to move files or data back and forth between the local file system and HDFS. But what if you want to process the data stored in HDFS? For example, imagine that you have stored sale transactions data in HDFS and you want to know the top five states that generated most of the revenue.

This is where you need to use either MapReduce, Hive, or Pig. MapReduce requires programming skills and expertise in Java (see Hour 4, “The MapReduce Job Framework and Job Execution Pipeline”; Hour 5, “MapReduce—Advanced Concepts and YARN”; Hour 10, “Programming MapReduce Jobs”; and Hour 11, “Customizing HDInsight Cluster with Script Action”). People with a SQL background can use Hive (see Hour 12, “Getting Started with Apache Hive and Apache Tez in HDInsight,” and Hour 13, “Programming with Apache Hive, Apache Tez in HDInsight, and Apache HCatalog”). Pig helps with data flow and data transformation for data analysis (see Hour 17, “Using Pig for Data Processing”).

You can use Sqoop to move data from HDFS to any relational database store (for example, SQL Server), and vice versa. You can learn more about Sqoop in Hour 18, “Using Sqoop for Data Movement Between RDBMS and HDInsight.”

GO TO Refer to Hour 18, “Using Sqoop for Data Movement Between RDBMS and HDInsight,” for more details on Sqoop and how you can use it for data movement.
What's New in HDFS 2.0

As you learned in Hour 2, “Introduction to Hadoop, Its Architecture, Ecosystem, and Microsoft Offerings,” HDFS in Hadoop 1.0 had some limitations and lacked support for providing a highly available distributed storage system. Consider the following limitations of Hadoop 1.0, related to HDFS; Hadoop 2.0 and HDFS 2.0 have resolved them.

- **Single point of failure**—Although you can have a secondary name node in Hadoop 1.0, it’s not a standby node, so it does not provide failover capabilities. The name node still is a single point of failure.

- **Horizontal scaling performance issue**—As the number of data nodes grows beyond 4,000 the performance of the name node degrades. This sets a kind of upper limit to the number of nodes in a cluster.

In Hadoop 2.0, HDFS has undergone an overhaul. Three important features, as discussed next, have overcome the limitations of Hadoop 1.0. The new version is referred to as HDFS 2.0 in Hadoop 2.0.

**HDFS High Availability**

In the Hadoop 1.0 cluster, the name node was a single point of failure. Name node failure gravely impacted the complete cluster availability. Taking down the name node for maintenance or upgrades meant that the entire cluster was unavailable during that time. The HDFS High Availability (HA) feature introduced with Hadoop 2.0 addresses this problem.

Now you can have two name nodes in a cluster in an active-passive configuration: One node is active at a time, and the other node is in standby mode. The active and standby name nodes remain synchronized. If the active name node fails, the standby name node takes over and promotes itself to the active state.

In other words, the active name node is responsible for serving all client requests, whereas the standby name node simply acts as a passive name node—it maintains enough state to provide a fast failover to act as the active name node if the current active name node fails.

This allows a fast failover to the standby name node if an active name node crashes, or a graceful failing over to the standby name node by the Hadoop administrator for any planned maintenance.

HDFS 2.0 uses these different methods to implement high availability for name nodes. The next sections discuss them.

**Shared Storage Using NFS**

In this implementation method, the file system namespace and edit log are maintained on a shared storage device (for example, a Network File System [NFS] mount from a NAS [Network Attached Storage]). Both the active name node and the passive or standby name node have access...
to this shared storage, but only the active name node can write to it; the standby name node can only read from it, to synchronize its own copy of file system namespace (see Figure 3.20).

![HDFS high availability with shared storage.](image)

When the active name node performs any changes to the file system namespace, it persists the changes to the edit log available on the shared storage device; the standby name node constantly applies changes logged by the active name node in the edit log from the shared storage device to its own copy of the file system namespace. When a failover happens, the standby name node ensures that it has fully synchronized its file system namespace from the changes logged in the edit log before it can promote itself to the role of active name node.

The possibility of a “split-brain” scenario exists, in which both the name nodes take control of writing to the edit log on the same shared storage device at the same time. This results in data loss or corruption, or some other unexpected result. To avoid this scenario, while configuring high availability with shared storage, you can configure a fencing method for a shared storage device. Only one name node is then able to write to the edit log on the shared storage device at one time. During failover, the shared storage devices gives write access to the new active name node (earlier, the standby name node) and revokes the write access from the old active name node (now the standby name node), allowing it to only read the edit log to synchronize its copy of the file system namespace.
NOTE

Heartbeat Signal from Data Nodes
Whatever method you choose for implementing high availability for name nodes (based on shared
storage or quorum-based storage using the Quorum Journal Manager), you must configure all the
data nodes in the cluster to send heartbeat signals and block-reports to both the active name
node and the standby name node. This ensures that the standby name node also has up-to-date
information on the location of blocks in the cluster. This helps with faster failover.

Quorum-based Storage Using the Quorum Journal Manager
This is one of the preferred methods. In this high-availability implementation, which leverages the
Quorum Journal Manager (QJM), the active name node writes edit log modifications to a group
of separate daemons. These daemons, called Journal Machines or nodes, are accessible to both the
active name node (for writes) and the standby name node (for reads). In this high-availability
implementation, Journal nodes act as shared edit log storage.

NOTE

Journal Nodes and Their Count
Journal nodes can be co-located on machines with name nodes, the JobTracker, or the YARN
ResourceManager because these are relatively lightweight daemons. However, at least three Journal
node daemons must be running to ensure that file system namespace changes (edit log) are written
to a majority of these daemons. You can even run these daemons on more than three machines
(usually an odd number, such as 3, 5, or 7) to handle an increased number of failures.

Given the previous configuration, in which N is the number of these daemons running, it can survive
at max (N – 1) / 2 failures of these daemons to work normally. An odd number is chosen to ensure
that a majority can be ascertained. For example, if you have five Journal nodes, then (5 – 1)/2 = 2
nodes can be taken out because of failure, whereas the majority of the rest of 3 available nodes
can be used for decisions.

When the active name node performs any changes to the file system namespace, it persists the
change log to the majority of the Journal nodes. The active name node commits a transaction
only if it has successfully written it to a quorum of the Journal nodes. The standby or passive name
node keeps its state in sync with the active name node by consuming the changes logged by the
active name node and applying those changes to its own copy of the file system namespace.

When a failover happens, the standby name node ensures that it has fully synchronized its file
system namespace with the changes from all the Journal nodes before it can promote itself to the
role of active name node.
CAUTION

Much like the earlier implementation method (shared storage using NFS), the “split-brain” scenario is possible here, too: Both the name nodes could become active, causing data loss or corruption or an unexpected result.

To avoid this scenario, Journal nodes allow only one name node to be a writer, to ensure that only one name node can write to the edit log at one time. During failover, the Journal node gives write access to the new active name node (earlier, the standby name node) and revokes the write access from the old active name node (now the standby name node). This new standby name node now can only read the edit log to synchronize its copy of file system namespace. Figure 3.21 shows HDFS high availability with Quorum-based storage works.

**FIGURE 3.21**
HDFS high availability with Quorum-based storage.

NOTE

Do You Need a Secondary Name Node When You Set Up High Availability?

When you enable high availability for your Hadoop 2.0 cluster, the standby name node also takes charge of performing the checkpoint operation of the file system namespace. The secondary name node’s task of performing checkpoint and backup becomes redundant with the presence of a standby name node in this new configuration, so having a separate secondary name node is not required in the high availability–enabled cluster. This means you can reuse the same hardware for the secondary name node from the earlier configuration for the standby name node in the new high availability configuration. However, make sure the standby name node has hardware equivalent to that of the active name node (the same amount of physical memory because it holds the same information or file system namespace, and same number of processors because it serves the same number of client requests upon failover).
Failover Detection and Automatic Failover

Automatic failover is controlled by the `dfs.ha.automatic-failover.enabled` configuration property in the `hdfs-site.xml` configuration file. Its default value is `false`, but you can set it to `true` to enable automatic failover. When setting up automatic failover to the standby name node if the active name node fails, you must add these two components to your cluster:

- **ZooKeeper quorum**—Set up ZooKeeper quorum using the `ha.zookeeper.quorum` configuration property in the `core-site.xml` configuration file and, accordingly, three or five ZooKeeper services. ZooKeeper has light resource requirements, so it can be co-located on the active and standby name node. The recommended course is to configure the ZooKeeper nodes to store their data on separate disk drives from the HDFS metadata (file system namespace and edit log), for isolation and better performance.

- **ZooKeeper Failover Controller (ZKFC)**—ZKFC is a ZooKeeper client that monitors and manages the state of the name nodes for both the active and standby name node machines. ZKFC uses the ZooKeeper Service for coordination in determining a failure (unavailability of the active name node) and the need to fail over to the standby name node (see Figure 3.22). Because ZKFC runs on both the active and standby name nodes, a split-brain scenario can arise in which both nodes try to achieve active state at the same time. To prevent this, ZKFC tries to obtain an exclusive lock on the ZooKeeper service. The service that successfully obtains a lock is responsible for failing over to its respective name node and promoting it to active state.

![Diagram of HDFS high availability and automatic failover](image)

**FIGURE 3.22**
HDFS high availability and automatic failover.
While implementing automatic failover, you leverage ZooKeeper for fault detection and to elect a new active name node in case of earlier name node failure. Each name node in the cluster maintains a persistent session in ZooKeeper and holds the special “lock” called Znode (an active name node holds Znode). If the active name node crashes, the ZooKeeper session then expires and the lock is deleted, notifying the other standby name node that a failover should be triggered.

ZKFC periodically pings its local name node to see if the local name node is healthy. Then it checks whether any other node currently holds the special “lock” called Znode. If not, ZKFC tries to acquire the lock. When it succeeds, it has “won the election” and is responsible for running a failover to make its local name node active.

**NOTE**

Automatic failover detection and redirection can be configured for both types of high availability setup (shared storage using NFS or Quorum-based storage using Quorum Journal Manager) in a similar way, even though here we show how to set it up only for Quorum-based storage using the Quorum Journal Manager.

**Client Redirection on Failover**

As you can see in Figure 3.23, you must specify the Java class name that the HDFS client will use to determine which name node is active currently, to serve requests and connect to the active name node.

**FIGURE 3.23**

Client redirection on failover.

**NOTE**

As of this writing, ConfiguredFailoverProxyProvider is the only implementation available in Hadoop 2.0, so use it unless you have a custom one.
The session timeout, or the amount of time required to detect a failure and trigger a failover, is controlled by the `ha.zookeeper.session-timeout.ms` configuration property. Its default value is 5000 milliseconds (5 seconds). A smaller value for this property ensures quick detection of the unavailability of the active name node, but with a potential risk of triggering failover too often in case of a transient error or network blip.

**HDFS Federation**

Horizontal scaling of HDFS is rare but needed in very large implementations such as Yahoo! or Facebook. This was a challenge with single name nodes in Hadoop 1.0. The single name node had to take care of entire namespace and block storage on all the data nodes of the cluster, which doesn’t require coordination. In Hadoop 1.0, the number of files that can be managed in HDFS was limited to the amount of RAM on a machine. The throughput of read/write operations is also limited by the power of that specific machine.

HDFS has two layers (see Figure 3.24—for Hadoop 1.0, look at the top section):

- **Namespace (managed by a name node)**—The file system namespace consists of a hierarchy of files, directories, and blocks. It supports namespace-related operations such as creating, deleting, modifying, listing, opening, closing, and renaming files and directories.

- **Block storage**—Block storage has two parts. The first is Block Management, managed by a name node. The second is called Storage and is managed by data nodes of the cluster.

- **Block Management (managed by a name node)**—As the data nodes send periodic block-reports, the Block Management part manages the data node cluster membership by handling registrations, processing these block-reports received from data nodes, maintaining the locations of the blocks, and ensuring that blocks are properly replicated (deleting overly replicated blocks or creating more replicas for under-replicated blocks). This part also serves the requests for block-related operations such as create, delete, modify, and get block location.

- **Storage (managed by data nodes)**—Blocks can be stored on the local file system (each block as separate file), with read/write access to each.
In terms of layers, both Hadoop 1.0 and Hadoop 2.0 remain the same, with two layers: Namespace and Block Storage. However, the works these layers encompass have changed dramatically.

In Figure 3.24, you can see that, in Hadoop 2.0 (bottom part), HDFS Federation is a way of partitioning the file system namespace over multiple separated name nodes. Each name node manages only an independent slice of the overall file system namespace. Although these name nodes are part of a single cluster, they don’t talk to each other; they are federated and do not require any coordination with each other. A small cluster can use HDFS Federation for file system namespace isolation (for multitenants) or for use in large cluster, for horizontal scalability.

HDFS Federation horizontally scales the name node or name service using multiple federated name nodes (or namespaces). Each data node from the cluster registers itself with all the name nodes in the cluster, sends periodic heartbeat signals and block-reports, stores blocks managed by any of these name nodes, and handles commands from these name nodes. The collection of all the data nodes is used as common storage for blocks by all the name nodes. HDFS Federation also adds client-side namespaces to provide a unified view of the file system.
A Data Node Registers with Multiple Name Nodes

A data node can register itself with multiple name nodes (name spaces) in the cluster and can store the blocks for these multiple name nodes.

In Hadoop 2.0, the block management layer is divided into multiple block pools (see Figure 3.24). Each one belongs to a single namespace. The combination of each block pool with its associated namespace is called the namespace volume and is a self-contained unit of management. Whenever any name node or namespace is deleted, its related block pool is deleted, too.

NOTE

Balancer in Hadoop 2.0

In Hadoop 2.0, the block balancer has been updated to work with multiple name nodes for balancing block storage at data nodes or balancing storage at the block pool level (this includes balancing block storage at data nodes).

Using HDFS Federation offers these advantages:

- Horizontal scaling for name nodes or namespaces.
- Performance improvement by breaking the limitation of a single name node. More name nodes mean improved throughput, with more read/write operations.
- Isolation of namespaces for a multitenant environment.
- Separation of services (namespace and Block Management), for flexibility in design.
- Block pool as a generic storage service (namespace is one application to use this service.)

Horizontal scalability and isolation is fine from the cluster level implementation perspective, but what about client accessing these many namespaces? Wouldn’t it be difficult for the client to access these many namespaces at first?

HDFS Federation also adds a client-side mount table to provide an application-centric unified view of the multiple file system namespaces transparent to the client. Clients may use View File System (ViewFs), analogous to client-side mount tables in some Unix/Linux systems, to create personalized namespace views or per cluster common or global namespace views.

HDFS Snapshot

Hadoop 2.0 added a new capability to take a snapshot (read-only copy, and copy-on-write) of the file system (data blocks) stored on the data nodes. The best part of this new feature is that it has been efficiently implemented as a name node-only feature with low memory overhead,
instantaneous snapshot creation, and no adverse impact on regular HDFS operations. It does not require additional storage space because no extra copies of data are required (or created) when creating a snapshot; the snapshot just records the block list and the file size without doing any data copying upon creation.

Additional memory is needed only when changes are done for the files (that belong to a snapshot), relative to a snapshot, to capture the changes in the reverse chronological order so that the current data can be accessed directly.

For example, imagine that you have two directories, d1 (it has files f1 and f2) and d2 (it has files f3 and f4), in the root folder at time T1. At time T2, a snapshot (S1) is created for the d2 directory, which includes f3 and f4 files (see Figure 3.25).

In Figure 3.26, at time T3, file f3 was deleted and file f5 was added. At this time, space is needed to store the previous version or deleted file. At time T4, you have to look at the files under the d2 directory; you will find only f4 and f5 (current data). On the other hand, if you look at the snapshot S1, you will still find files f3 and f4 under the d2 directory; it implies that a snapshot is calculated by subtracting all the changes made after snapshot creation from the current data (for example, Snapshot data = Current Data – any modifications done after snapshot creation).
Consider these important characteristics of the HDFS snapshot feature:

- You can take a snapshot of the complete HDFS file system or of any directory or subdirectory in any subtree of the HDFS file system. To take a snapshot, you first must check that you can take a snapshot for a directory or make a directory snapshottable. Note that if any snapshot is available for the directory, the directory can be neither deleted nor renamed before all the snapshots are deleted.

- Snapshot creation is instantaneous and doesn’t require interference with other regular HDFS operations. Any changes to the file are recorded in reverse chronological order so that the current data can be accessed directly. The snapshot data is computed by subtracting the changes (which occurred since the snapshot) from the current data from the file system.

- Although there is no limit to the number of snapshottable directories you can have in the HDFS file system, for a given directory, you can keep up to 65,536 simultaneous snapshots.

- Subsequent snapshots for a directory consume storage space for only delta data (changes between data already in snapshot and the current state of the file system) after the first snapshot you take.

- As of this writing, you cannot enable snapshot (snapshottable) on nested directories. As an example, if any ancestors or descendants of a specific directory have already enabled for snapshot, you cannot enable snapshot on that specific directory.

- The .snapshot directory contains all the snapshots for a given directory (if enabled for snapshot) and is actually a hidden directory itself. Hence, you need to explicitly refer to it when referring to snapshots for that specific directory. Also, .snapshot is now a reserved word, and you cannot have a directory or a file with this name.

Consider an example in which you have configured to take a daily snapshot of your data. Suppose that a user accidently deletes a file named sample.txt on Wednesday. This file will no longer be available in the current data (see Figure 3.27). Because this file is important (maybe it got deleted accidentally), you must recover it. To do that, you can refer to the last snapshot TuesS, taken on Tuesday (assuming that the file sample.txt was already available in the file system when this snapshot was taken) to recover this file.
FIGURE 3.27
An example scenario for HDFS Snapshot.

**HDFS Snapshot Commands**

You can use either of these commands to enable snapshot for a path (the path of the directory you want to make snapshottable):

```
hdfs dfsadmin -allowSnapshot <path>
hadoop dfsadmin -allowSnapshot <path>
```

To disable to take a snapshot for a directory, you can use either of these commands:

```
hdfs dfsadmin -disallowSnapshot <path>
hadoop dfsadmin -disallowSnapshot <path>
```

When a directory has been enabled to take a snapshot, you can run either of these commands to do so. You must specify the directory path and name of the snapshot (an optional argument—if it is not specified, a default name is generated using a time stamp with the format 's'yyyyMMdd-HHmmss.SSS) you are creating. You must have owner privilege on the snapshottable directory to execute the command successfully.

```
hdfs dfs -createSnapshot <path> <snapshotName>
hadoop dfs -createSnapshot <path> <snapshotName>
```

If there is a snapshot on the directory and you want to delete it, you can use either of these commands. Again, you must have owner privilege on the snapshottable directory to execute the command successfully.

```
hdfs dfs -deleteSnapshot <path> <snapshotName>
hadoop dfs -deleteSnapshot <path> <snapshotName>
```

You can even rename the already created snapshot to another name, if needed, with the command discussed next. You need owner privilege on the snapshottable directory to execute the command successfully.
If you need to compare two snapshots and identify the differences between them, you can use either of these commands. This requires you to have read access for all files and directories in both snapshots.

hdfs snapshotDiff <path> <startingSnapshot> <endingSnapshot>
hadoop snapshotDiff <path> <startingSnapshot> <endingSnapshot>

You can use either of these commands to get a list of all the snapshottable directories where you have permission to take snapshots:

hdfs lsSnapshottableDir
hadoop lsSnapshottableDir

You can use this command to list all the snapshots for the directory specified—for example, for a directory enabled for snapshot, the path component .snapshot is used for accessing its snapshots:

hdfs dfs -ls /<path>/.snapshot

You can use this command to list all the files in the snapshot s5 for the testdir directory:

hdfs dfs -ls /testdir/.snapshot/s5

You can use all the regular commands (or native Java APIs) against snapshot. For example, you can use this command to list all the files available in the bar subdirectory under the foo directory from the s5 snapshot of the testdir directory:

hdfs dfs -ls /testdir/.snapshot/s5/foo/bar

Likewise, you can use this command to copy the file sample.txt, which is available in the bar subdirectory under the foo directory from the s5 snapshot of the testdir directory to the tmp directory:

hdfs dfs -cp /testdir/.snapshot/s5/foo/bar/sample.txt /tmp

Summary

The Hadoop Distributed File System (HDFS) is a core component and underpinning of the Hadoop cluster. HDFS is a highly scalable, distributed, load-balanced, portable, and fault-tolerant storage compound (with built-in redundancy at the software level). In this hour, we went into greater detail to understand the internal architecture of HDFS.
We looked at how data gets stored in the HDFS, how a file gets divided among one or more data blocks, how blocks are stored across multiple nodes for better performance and fault tolerance, and how replication factor is maintained. We also looked at the process of writing a file to HDFS and reading a file from HDFS, and we saw what happens behind the scenes. Then we used some commands to play around with HDFS for the storage and retrieval of files.

Finally, we looked at different limitations of HDFS in Hadoop 1.0 and how Hadoop 2.0 overcomes them. We then discussed in detail HDFS-related major features in Hadoop 2.0 in this hour.

Q&A

Q. What is HDFS, and what are the design goals?
A. HDFS is a highly scalable, distributed, load-balanced, portable, and fault-tolerant storage component of Hadoop (with built-in redundancy at the software level).

When HDFS was implemented originally, certain assumptions and design goals were discussed:

- **Horizontal scalability**—Based on the scale-out model. HDFS can run on thousands of nodes.

- **Fault tolerance**—Keeps multiple copies of data to recover from failure.

- **Capability to run on commodity hardware**—Designed to run on commodity hardware.

- **Write once and read many times**—Based on a concept of write once, read multiple times, with an assumption that once data is written, it will not be modified. Its focus is thus retrieving the data in the fastest possible way.

- **Capability to handle large data sets and streaming data access**—Targeted to small numbers of very large files for the storage of large data sets.

- **Data locality**—Every slave node in the Hadoop cluster has a data node (storage component) and a JobTracker (processing component). Processing is done where data exists, to avoid data movement across nodes of the cluster.

- **High throughput**—Designed for parallel data storage and retrieval.

- **HDFS file system namespace**—Uses a traditional hierarchical file organization in which any user or application can create directories and recursively store files inside them.

Q. In terms of storage, what does a name node contain and what do data nodes contain?
A. HDFS stores and maintains file system metadata and application data separately. The name node (master of HDFS) contains the metadata related to the file system (information about each file, as well as the history of changes to the file metadata). Data nodes (slaves of HDFS) contain application data in a partitioned manner for parallel writes and reads.
The name node contains an entire metadata called namespace (a hierarchy of files and directories) in physical memory, for quicker response to client requests. This is called the fsimage. Any changes into a transactional file is called an edit log. For persistence, both of these files are written to host OS drives. The name node simultaneously responds to the multiple client requests (in a multithreaded system) and provides information to the client to connect to data nodes to write or read the data. While writing, a file is broken down into multiple chunks of 64MB (by default, called blocks). Each block is stored as a separate file on data nodes. Based on the replication factor of a file, multiple copies or replicas of each block are stored for fault tolerance.

Q. What is the default data block placement policy?
A. By default, three copies, or replicas, of each block are placed, per the default block placement policy mentioned next. The objective is a properly load-balanced, fast-access, fault-tolerant file system:
   ▶ The first replica is written to the data node creating the file.
   ▶ The second replica is written to another data node within the same rack.
   ▶ The third replica is written to a data node in a different rack.

Q. What is the replication pipeline? What is its significance?
A. Data nodes maintain a pipeline for data transfer. Having said that, data node 1 does not need to wait for a complete block to arrive before it can start transferring it to data node 2 in the flow. In fact, the data transfer from the client to data node 1 for a given block happens in smaller chunks of 4KB. When data node 1 receives the first 4KB chunk from the client, it stores this chunk in its local repository and immediately starts transferring it to data node 2 in the flow. Likewise, when data node 2 receives the first 4KB chunk from data node 1, it stores this chunk in its local repository and immediately starts transferring it to data node 3, and so on. This way, all the data nodes in the flow (except the last one) receive data from the previous data node and, at the same time, transfer it to the next data node in the flow, to improve the write performance by avoiding a wait at each stage.

Q. What is client-side caching, and what is its significance when writing data to HDFS?
A. HDFS uses several optimization techniques. One is to use client-side caching, by the HDFS client, to improve the performance of the block write operation and to minimize network congestion. The HDFS client transparently caches the file into a temporary local file; when it accumulates enough data for a block size, the client reaches out to the name node. At this time, the name node responds by inserting the filename into the file system hierarchy and allocating data nodes for its storage. The client then flushes the block of data from the local, temporary file to the closest data node, and that data node transfers the block to other data nodes (as instructed by the name node, based on the replication factor of the file). This client-side caching avoids continuous use of the network and minimizes the risk of network congestion.
Q. How can you enable rack awareness in Hadoop?

A. You can make the Hadoop cluster rack aware by using a script that enables the master node to map the network topology of the cluster using the properties topology.script.file.name or net.topology.script.file.name, available in the core-site.xml configuration file. First, you must change this property to specify the name of the script file. Then you must write the script and place it in the file at the specified location. The script should accept a list of IP addresses and return the corresponding list of rack identifiers. For example, the script would take host.foo.bar as an argument and return /rack1 as the output.

**Quiz**

1. What is the data block replication factor?
2. What is block size, and how is it controlled?
3. What is a checkpoint, and who performs this operation?
4. How does a name node ensure that all the data nodes are functioning properly?
5. How does a client ensure that the data it receives while reading is not corrupted?
6. Is there a way to recover an accidentally deleted file from HDFS?
7. How can you access and manage files in HDFS?
8. What two issues does HDFS encounter in Hadoop 1.0?
9. What is a daemon?

**Answers**

1. An application or a job can specify the number of replicas of a file that HDFS should maintain. The number of copies or replicas of each block of a file is called the replication factor of that file. The replication factor is configurable and can be changed at the cluster level or for each file when it is created, or even later for a stored file.

2. When a client writes a file to a data node, it splits the file into multiple chunks, called blocks. This data partitioning helps in parallel data writes and reads. Block size is controlled by the dfs.blocksize configuration property in the hdfs-site.xml file and applies for files that are created without a block size specification. When creating a file, the client can also specify a block size specification to override the cluster-wide configuration.

3. The process of generating a new fsimage by merging transactional records from the edit log to the current fsimage is called checkpoint. The secondary name node periodically performs a checkpoint by downloading fsimage and the edit log file from the name node and then uploading the new fsimage back to the name node. The name node performs a checkpoint upon restart (not periodically, though—only on name node start-up).
4. Each data node in the cluster periodically sends heartbeat signals and a block-report to the name node. Receipt of a heartbeat signal implies that the data node is active and functioning properly. A block-report from a data node contains a list of all blocks on that specific data node.

5. When writing blocks of a file, an HDFS client computes the checksum of each block of the file and stores these checksums in a separate hidden file in the same HDFS file system namespace. Later, while reading the blocks, the client references these checksums to verify that these blocks were not corrupted (corruption might happen because of faults in a storage device, network transmission faults, or bugs in the program). When the client realizes that a block is corrupted, it reaches out to another data node that has the replica of the corrupted block, to get another copy of the block.

6. By default, no—but you can change this default behavior. You can enable the Trash feature of HDFS using two configuration properties: `fs.trash.interval` and `fs.trash.checkpoint.interval` in the `core-site.xml` configuration file. After enabling it, if you delete a file, it gets moved to the Trash folder and stays there, per the settings. If you happen to recover the file from there before it gets deleted, you are good; otherwise, you will lose the file.

7. You can access the files and data stored in HDFS in many different ways. For example, you can use HDFS FS Shell commands, leverage the Java API available in the classes of the `org.apache.hadoop.fs` package, write a MapReduce job, or write Hive or Pig queries. In addition, you can even use a web browser to browse the files from an HDFS cluster.

8. First, the name node in Hadoop 1.0 is a single point of failure. You can configure a secondary name node, but it’s not an active-passing configuration. The secondary name node thus cannot be used for failure, in case the name node fails. Second, as the number of data nodes grows beyond 4,000, the performance of the name node degrades, setting a kind of upper limit to the number of nodes in a cluster.

9. The word *daemon* comes from the UNIX world. It refers to a process or service that runs in the background. On a Windows platform, we generally refer to it as a service. For example, in HDFS, we have daemons such as name node, data node, and secondary name node.
adding
  Slicer controls, 401
airline departures, analyzing, 360-361
algorithms
  built-in machine learning, 509-512
  compression-decompression, 151
  regression, 492
ALTER DATABASE command, 328
ALTER INDEX command, 364
ALTER TABLE command, 337
Amazon Dynamo, 20
Amazon Elastic MapReduce (EMR), 51-52
Ambari, 50-51
analysis
  airline departures, 360-361
canceled airline flights, 362-363
datasets, 358-360
diagnostic (descriptive), 10, 492
flight delays, 256-261
PowerPivot
  data models, 398
  in Excel, 399
predictive, 11, 492
prescriptive, 11
speed limit violations, 524-532
Analytics Platform System (APS), 58
Apache Ambari, 50-51

A
access control lists (ACLs), 154
accessing
  HDInsight data from Power Query, 417-421
Hive data
  from Excel, 379-384
  from PowerPivot, 385-406
  from SQL Server Analysis Services (SSAS), 413-415
  from SQL Server Reporting Services (SSRS), 415-417
  from SQL Servers, 406-417
  PowerPivot from Excel, 399
accounts, creating storage, 192
actions (Oozie), 456
Active Directory, 49
add method, 150
Add-AzureAccount cmdlet, 441
Add-AzureHdInsightScriptAction
PowerShell cmdlet, 281
add-ins, 386

32-bit Hive ODBC drivers
  comparing, 372
  configuring, 373-377
64-bit Hive ODBC drivers
  comparing, 372
  configuring, 373-377

Index
Apache Cassandra, 22
Apache Flume. See Flume
Apache Hadoop. See Hadoop
Apache HBase. See HBase
Apache HCatalog. See HCatalog
Apache Hive. See Hive
Apache Mahout. See Mahout
Apache Nutch. 65
Apache Oozie. See Oozie
Apache Pig. See Pig
Apache Spark. See Spark
Apache Sqoop. See Sqoop
Apache Storm. See Storm
Apache Tez. See Tez
application life-cycle management, 160. See also YARN
Application Manager, 163
Application Master, 166
communication with, 169
failures, 171-172
uber-tasking optimization, 170
applications
Apache Storm, creating, 524-529
Hive, creating, 310
Tez, 317
APS (Analytics Platform System), 58
architecture
Hadoop, 33
Hadoop 2.0, 43-44
HBase, 538-539
HDFS (Hadoop Distributed File System), 33-35
block placement and replication, 74-77
deleting files, 86-87
file split, 73-74
handling failure, 83-85
name nodes, 67-72
reading from, 82-83
writing to, 77-82
HDFS (Hadoop Distributed File System), 33-35
application life-cycle management, 160. See also YARN
Application Manager, 163
Application Master, 166
communication with, 169
failures, 171-172
uber-tasking optimization, 170
applications
Apache Storm, creating, 524-529
Hive, creating, 310
Tez, 317
APS (Analytics Platform System), 58
architecture
Hadoop, 33
Hadoop 2.0, 43-44
HBase, 538-539
HDFS (Hadoop Distributed File System), 33-35
block placement and replication, 74-77
deleting files, 86-87
file split, 73-74
handling failure, 83-85
name nodes, 67-72
reading from, 82-83
writing to, 77-82
HDInsight
clusters, 223-225
storage, 240-241
MapReduce, 35-36, 116-119
Storm, 520-522
ASV (Azure Storage Vault), 241
atomicity, consistency, isolation, and durability (ACID), 344
attributes, hiding, 399
authentication, 376
automatic failover, 99-100, 173-175
automating HDInsight, 199-212
availability (HA), 95-101
automatic failover, 99-100
failover detection, 99-100
QJM (Quorum Journal Manager), 97-98
shared storage using NFS, 95-97
in YARN, 173-175
Avro, 263-270
Hadoop streaming, 273
AZCopy, 237
Azure. See Microsoft Azure
Azure Flat Network Storage, 242
Azure Management Portal, 280-281, 499
Azure ML (Microsoft Azure Machine Learning), 48, 56-57, 491
benefits of, 494-495
built-in machine learning algorithms, 509-512
data, reading, 492-502
Descriptive Statistics module, 505
history of machine learning, 491-493
overview of, 493-495
predictive models, 506-512
processes to build solutions, 497-498
web services, publishing as, 512-514
workflow, 498-505
workspaces, 495-497
Azure ML Studio, 496-497
Azure PowerShell, 237-239
automation setup, 426-427
Azure storage, 47
Azure Storage Blob. See WASB
Azure Storage Explorer, 235-237
Azure Storage Vault (ASV), 241
Azure virtual networks, 545-546
Azure VNet, 49

B
bad records, handling, 157
base classes, extending, 130
Big Data
business value proposition, 26
characteristics of, 12-13
variability, 14
variety, 13
velocity, 14
volume, 13
data sources. See data sources
Hadoop versus, 14-15
clusters

importance of managing, 15-16
phases of analytics
data acquisition, 46-47
data analysis, 47-48
data management, 49
data storage, 47
data visualization, 48-49
development and monitoring, 49
usage scenarios, 27-29
BISM (Business Intelligence Semantic Model), 413
bitmap indexes, 364
blacklisting
customizing, 155
Hadoop, 190
HDFS clusters. See also HDFS clusters
HDInsight, 55-56
Hadoop clusters on, 55-56
Microsoft Big Data solutions, 59-60
Power BI, 378-379
Cloudera, 50
Manager, 50
Support, 50
CloudXplorer, 324
Cluster Dashboards (Apache Hive), 299-301
clusters
blacklisting, 155
Hadoop, customizing, 190
HDFS clusters
HDInsight. See HDFS clusters
HDInsight, 378-379
accessing hive-based data from, 379-384
architecture, 223-225
connecting from Visual Studio, 303
deployment, 182-186
managing, 212-214
provisioning, 203-206
Tez, 318
codecs, 151
column families (HBase), 536, 541-544
column-store databases, 21-22, 536
columns
in Hive tables, 338
CombineFileInputFormat class, 123
Combiners (MapReduce), 114, 132-134
command line
Hadoop for Azure Storage Blob, 239
HDFS (Hadoop Distributed File System), 90-94
MapReduce, 139
command-line interfaces (CLIs), 139
Hive, 290-295
comma-separated value (CSV) files, 256, 324
COMMENT clause, 340
commodity hardware, HDFS and, 66
communication
with Application Master, 169
counters for, 158
compact indexes, 363
compaction of HBase tables, 544-545
components
Hadoop folders, 291
of YARN, 162-166
compression (MapReduce), 150-151
configure() function, 129
configuring
32-bit Hive ODBC drivers, 377-378
64-bit Hive ODBC drivers, 373-377
Azure ML (Microsoft Azure Machine Learning), 495-496
default file system, 241-242
HDIInsight, verifying, 196-198
Hive ODBC data source, 429-430
Microsoft Azure storage emulator, 251-253
Microsoft Hive ODBC Drivers, 372-378
remote desktop connections, 193-195
connections
to existing connections, 392
HDIInsight, 389
Microsoft Azure, 200
remote desktop, configuring, 193-195
contains method, 150
Context object (MapReduce), 139-140
Control Flow (SSIS), 424
coordinating Oozie workflows, 464-465
copyFromLocal parameter, 91
copyToLocal parameter, 91
cost-based optimization (CBO) engines, 319
costs
effectiveness of Hadoop, 32
of Microsoft Azure, 180
CouchDB, 23-24
counters (MapReduce), 158-159
CRAN (Comprehensive R Archive Network), 468
CREATE DATABASE command, 328
Create New Data Source Wizard, 374, 378
CREATE TABLE AS SELECT (CTAS) command, 341
CREATE TABLE command, 338
CREATE TABLE LIKE command, 341
CREATE VIEW command, 342
credit scoring and analysis, 27
crime data analysis scenario (Spark SQL), 486-489
cross-selling recommendations, 27
customer churn analysis, 27
customizing
counters, 159
HDIInsight, 190
HDIInsight clusters
consuming Script Actions, 280-282
identifying need for, 277-278
running Giraph job, 282-284
script development, 278-280
testing Script Action, 285
partitioning, 129
Cutting, Doug, 31, 263
daemons, 69
single nodes running on, 119
data
Big Data. See Big Data
semi-structured data, 12
Driver class (MapReduce), 134-138
DROP DATABASE command, 329
dynamic partition inserts, 355-358

E
ecosystems for Hadoop
Flume, 39-40
HBase, 39
HCatalog, 37
Hive, 36-37
Mahout, 39
MSBI (Microsoft Business Intelligence), 42
Oozie, 40
Pegasus, 41
Pig, 37-38
RHadoop, 41
Sqoop, 38
edit log file, 70
EMR (Amazon Elastic MapReduce), 51-52
emulators (HDInsight), 245-246
HBase and, 550
installation, verifying, 248-249
installing, 246-248
MapReduce job, 249-250
Microsoft Azure storage
emulator, 251-253
Script Action, testing, 285
enabling R language
on Hadoop, 471-472
on HDInsight, 472-476
Enterprise Agreement (EA) model, 180
equi-joins, 347

ETL (extraction, transformation, and loading), 417
workflow packages, creating, 424
Excel, 48
Hive, accessing data from, 379-384
Power Query, launching, 417-421
reporting, 399-402
visualization, 399-402
executing
Apache Hive (HDInsight), 288-301
Hive queries, 288, 313-314, 428
HiveQL queries, 294
Oozie workflows, 461
Pig jobs, 441-442
PowerShell scripts, 200
SSIS packages, 430-431
Tez, 318, 365-368
exercises
Hadoop command line, 239
HDFS commands, 92-93
HDInsight name node services, 225-227
importing data from SQL Server to Sqoop, 448-451
installing HDInsight emulator, 246-248
running MapReduce job, 250
experiments
Azure ML (Microsoft Azure Machine Learning), 495-496
creating, 501
exporting Sqoop
data with PowerShell, 451-452
exercise, 448-451
export process, 448
from HDFS with, 447-448
extending base classes, 130
external data, loading in R, 468-469
external tables, programming, 335-338
extraction, transformation, and loading (ETL), 417

F
failover
automatic (YARN), 173-175
detection, 99-100
client redirection on, 100
with HBase, 539
with ZooKeeper, 228-230
failure, handling in HDFS, 83-85
Fair Scheduler, 152-153
Fast SQLPrepare, 376
fault tolerance
of Hadoop, 32
of HDFS, 66
MapReduce, 115
Spark, 482-483
federation, HDFS (Hadoop Distributed File System), 44-45, 101-103
fields grouping, 521
file system namespace of HDFS, 67
FileInputFormat class, 123
files
Avro, 263
comma-separated value (CSV), 256, 324
Optimized Row Columnar (ORC), 320
split in HDFS, 73-74
storage (Microsoft Azure), 234
filtering Pig relations, 436-437
first in, first out (FIFO) method, 152, 154
flexibility of Hadoop, 32
flight delays, analyzing, 256-261
flight departure analysis scenario (Oozie), 457-463
defining workflows, 459-461
executing workflows, 461
monitoring workflows, 461
prerequisites, 458-459
querying results, 462-463
flight departure analysis scenario (SSIS), 424-427
package variables, 426
prerequisites, 425-426
Windows Azure PowerShell automation, 426-427
Flume, 39-40, 46
folders, Hadoop components, 291
fraud detection and prevention, 27
FROM clause, 347
From Data Connection Wizard, 379
FS Shell, 90-94
fsimage files, 69-70, 221-222
Checkpoint process, 70-72
functions (MapReduce), 113

G
get parameter, 91
Get-AzureHDInsightJobOutput cmdlet, 442, 452
getPartition() function, 129
GFS (Google File System), 65
ggplot2, 470
Giraph
running jobs, 282-284
Script Action, installing with, 278-280
Google BigTable, 536
Google File System (GFS), 65
graphs
databases, 24
mining, 41
grouping
Pig relations, 437
Storm, 521
Grunt, running code in, 436

H
HA (High Availability), 95-101
automatic failover, 99-100
failover detection, 99-100
QJM (Quorum Journal Manager), 97-98
Resource Manager, 173-175
shared storage using NFS, 95-97
Hadoop
architecture, 33
Avro, 263-270
Big Data versus, 14-15
characteristics of, 32
clusters, customizing, 190
command line, for Azure Storage Blob, 239
components folders, 291
compression-decompression, 150
ecosystems
Flume, 39-40
HBase, 39
HCatalog, 37
Hive, 36-37
Mahout, 39
MSBI (Microsoft Business Intelligence), 42
Oozie, 40
Pegasus, 41
Pig, 37-38
RHadoop, 41
Sqoop, 38
enabling R on, 471-472
explained, 31-32
licensing, 50
rack awareness, 88-89
serialization frameworks, 262-270
streaming, 146-148, 270-275
vendors
Amazon Elastic MapReduce (EMR), 51-52
Cloudera, 50
Hortonworks, 50-51
MapR Technologies, 51
Microsoft Big Data solutions, 52-57
Hadoop 1.0
Checkpoint-related configuration files, 71
limitations, 42-43
Hadoop 2.0
architecture, 43-44
Checkpoint-related configuration files, 71
hdfs dfs command, 94
HDFS new features, 44-46
Hadoop Distributed File System
See HDFS
hadoop fs command parameters, 91-92
Hadoop MapReduce. See MapReduce
handling
bad records, 157
features (MapReduce), 154-158
HBase, 22, 39, 47, 535-537
architecture, 538-539
HDInsight cluster creation, 545-551
  Azure management portal, 546-547
  Azure virtual networks, 545-546
  PowerShell, 547-550
  verifying, 550-551
success stories, 538
tables
  compaction, 544-545
  creating, 539
  data distribution and storage, 541-544
  reading from, 541
  writing to, 540-541
  when to use, 537-538
HBase Master, 538-539
HBase Region Servers, 538-539
HCatalog, 37, 49, 368-369
  in Pig Latin scripts, 438-440
HDFS (Hadoop Distributed File System)
architecture, 33-35
  block placement and replication, 74-77
  deleting files, 86-87
  file split, 73-74
  handling failure, 83-85
  name nodes, 67-72
  reading from, 82-83
  writing to, 77-82
  blob storage versus, 234-235, 240-241
  command line, 90-94
  data processing tools, 94
  default file system, 94
  design goals, 66-67
  exporting data with Sqoop, 447-448
  Federation, 101-103
  HA (High Availability), 95-101
    automatic failover, 99-100
    failover detection, 99-100
    QJM (Quorum Journal Manager), 97-98
  shared storage using NFS, 95-97
Hadoop 1.0 versus Hadoop 2.0, 43
Hadoop 2.0 features, 44-46
HDInsight support, 224
  new features in, 95-107
  origin of, 65
  rack awareness, 88-89
  snapshot, 103-107
WebHDFS, 89-90
HDFS (Hadoop Distributed File System) clusters
components, 219
  name nodes, 220
  secondary name nodes, 221-222
  standby name nodes, 222
  rack awareness, 88-89
hdfs dfs command, 94
HDInsight, 52, 58
  Apache Hive, executing, 288-301
  Apache Tez in, 287
  benefits, 59
  blob storage
    for HBase clusters, 539
    HDFS versus, 234-235
  cluster customization
    consuming Script Actions, 280-282
    identifying need for, 277-278
    running Giraph job, 282-284
    script development, 278-280
    testing Script Action, 285
cluster provisioning, 427-431
cluster customization
  consuming Script Actions, 280-282
  identifying need for, 277-278
  running Giraph job, 282-284
  script development, 278-280
  testing Script Action, 285
clusters
  accessing hive-based data from, 379-384
  architecture, 223-225
  connecting from Visual Studio, 303
  creating with HBase, 545-551
  deployment, 182-186
  provisioning, 203-206
  viewing Hive Jobs, 306-308
  configuring, verifying, 196-198
  connections, 389
  customizing, 190
data integration with SQL Server, importance of, 423
Hive Job Browser window, 307
HiveQL
Pig Latin versus, 38 queries, executing, 294
horizontal scalability, 101 of Hadoop, 44-45 of HDFS, 66
Hortonworks, 50-51
Hortonworks Data Platform (HDP) on Windows, 52
on virtual machines, 60

I
IaaS (Infrastructure as a Service), 179
IDEs (integrated development environments), Azure ML Studio, 496-497
IF EXISTS clause, 337
IF NOT EXISTS clause, 337
IF NOT EXISTS keyword, 328
Import Wizard, 388
Import-AzurePublishSettingsFile cmdlet, 441
importing
data from ODBC, 389
to HDInsight clusters with Sqoop, 445-447
to Hive with Sqoop, 447
publish settings, 426-427
Sqoop
data with PowerShell, 451-452
exercise, 448-451
tables, 389
indexes
in Hive, 363-365
InputFormat class (MapReduce), 121
InputSplit interface, 121
installing
Giraph with Script Action, 278-280
HDInsight, 196, 246-248
running MapReduce job, 249-250
verifying installation, 248-249
.NET clients, 311
R language
on HDInsight, 472-473
on Windows, 468
Spark on HDInsight, 480-481
integration of data. See data integration; SSIS (SQL Server Integration Services)
interfaces
command-line interfaces (CLIs), 139, 290-295
InputSplit, 121
RecordReader, 122
Tool (MapReduce), 139
WritableComparable<T>, 127
internal tables, programming, 331-335

J
Java, reading using Avro, 264-270
Java Virtual Machines (JVMs), 116, 124, 147
MapReduce, 151-152
Job History Server, 165, 225
job.properties file, creating, 461
jobs
completion of, 315
execution flow in YARN, 167-169
Hive, viewing, 306-308
MapReduce
execution flow, 119-140
programming, 255
request/response flow, 117-118
Pig, creating definitions, 441
scheduling, 152
JobTrackers, 35, 225
failure, 154
MapReduce, 115-116
joins
equi-joins, 347
MapReduce, 148-149
map-side, 148
Pig relations, 437
reduce-side, 149
Journal nodes, 97

K
Karmasphere Analyst, 52
key-value store databases, 20-21
blob storage, 239
Kimball, Ralph, 391
Klout, 415

L
layers
Data Access, 414
Data Model, 415
HDFS (Hadoop Distributed File System), 101
licensing Hadoop, 50
life science research, 28
linking SQL Server, 406-412
listings
    defining job properties, 461
    defining Oozie time-based coordinator job, 464
    defining Oozie workflows, 460, 463
    defining Pig relations, 436
    defining properties of Oozie time-based coordinator job, 465
    Giraph
        installation script, 279
        job submission, 284
    HCatalog usage with Pig queries, 440
    HDFS command-line interface example, 92
    Pig
        job definition creation, 441
        job execution submission, 442
    provisioning HDInsight with HBase cluster using PowerShell scripting, 547
    rmr package usage, 474
lists (R), 468
load balancing, HBase regions, 543
load processes, Hive tables, 343-346
loading
    data into PowerPivot, 391-397
    external data in R, 468-469
local mode (Pig Latin), 434
LOCATION clause, 339
log mining with Spark, 483-485
ls parameter, 92
lsr parameter, 92

M

machine learning. See also Azure ML
    built-in machine learning algorithms, 509-512
    Descriptive Statistics module, 505
    history of, 491-493
    overview of, 493-495
    predictive models, 506-512
    processes to build solutions, 497-498
    web services, publishing as, 512-514
    workflow, 498-505
    workspaces, 495
Mahout, 39, 48
management
    Big Data, importance of, 15-16
    blocks (HDFS), 101
    HDInsight, 212-214
Management Portal (Azure), provisioning HDInsight, 186-198
many-to-many relationships, 397
map function, 125
Map function (MapReduce), 113
Mapper class (MapReduce), 124-128
Mapper function, 256
MapR Technologies, 51
MapReduce, 94
    Application Master, 225
    architecture, 35-36, 116-119
    bad records, handling, 157
    Bloom Filter objects, 149-150
    C#, programming, 270-273
    Capacity Scheduler, 153-154
Combiners, 132-134
    compression, 150-151
Context object, 139-140
counters, 158-159
DistributedCache, 145-146
Driver class, 134-138
Fair Scheduler, 152-153
flight delays, analyzing, 256-261
Hadoop streaming, 146-148
handling features, 154-158
Hello World, 255-256
InputFormat class, 121
Java Virtual Machines (JVMs), 151-152
jobs
    execution flow, 119-140
    programming, 255
    request/response flow, 117-118
    running, 249-250, 472-476
    scheduling, 152
JobTracker failure, 154
joins, 148-149
Mapper class, 124-128
    mode (Pig Latin), 434
    multiple input and output formats, 124
    number of Mappers, 127-128
    optimizing, 150-154
OutputFormat class, 123
    overview of, 113-116
    partitioning, 128-130
Reducer class, 130-132
Spark versus, 479, 481-483
    speculative execution, 156-157
    task failures, 155-156
TaskTrackers
- data node co-location, 118-119
- failure, 154-155
Tez. See Tez
Tool interface, 139
YARN
- comparing, 45-46
- components of, 162-166
- failures in, 171-173
- Node Manager, 166-167
- overview of, 160-169
- uber-tasking optimization, 170-171
map-side joins, 148
matrix (R), 468
maxbytes property, 170
maxmaps property, 170
maxreduces property, 170
MDX (Multi-Dimensional Expression), 414
membershipTest method, 150
metadata
- layers (HCatalog), 368-369
- storage in name nodes, 221-222
metastores (Apache Hive), 289
methods, HDInsightUtilities module, 278
Microsoft Analytics Platform System (APS), 58
Microsoft Azure. See also HDInsight
- adding accounts to PowerShell, 441
- blob storage
  - AZCopy, 237
  - Azure PowerShell, 237-239
  - Azure Storage Explorer, 235-237
- data locality, 242-243
- Hadoop command line, 239
- for HBase clusters, 539
- HDFS versus, 234-235
- HDInsight storage architecture, 240-241
- performance, 242-243
Management Portal
- HDInsight cluster creation with HBase, 546-547
- provisioning HDInsight, 186-198
- overview of, 179-181
- publish settings, downloading and importing, 426-427
- storage emulator, configuring, 251-253
- storage services, 181, 233-234
- table storage, 20-21
- virtual machines, 60
Microsoft Azure Machine Learning.
- See Azure ML
Microsoft Big Data solutions, 52-57
- cloud services, 59-60
- on-premises, 58
Microsoft Business Intelligence (MSBI), 42, 54, 371
Microsoft Hive ODBC Drivers, 371-378
- 32-bit/64-bit IDBC drivers, comparing, 372
Microsoft OLE DB Provider for ODBC (MSDASQL), 409
Microsoft Power BI, overview of, 378-379
Microsoft SQL Server Parallel Data Warehouse (PDW), 58
mkdir parameter, 91
models
- dimensional modeling (DM), 391
- Enterprise Agreement (EA), 180
- pay as you go (PAYG), 180
- predictive, machine learning, 506-512
modules
- Azure ML (Microsoft Azure Machine Learning), 495
- Descriptive Statistics, 505
MongoDB, 23
monitoring
- HDInsight, 212-214
- Oozie workflows, 461
moving map-outputs, 129
MSBI (Microsoft Business Intelligence), 42, 54, 371
MSDASQL (Microsoft OLE DB Provider for ODBC), 409
Multi-Dimensional Expression (MDX), 414
- multiple input and output formats (MapReduce), 124
- multitenancy, 161, 166
- YARN, 45
name nodes, 220
- block placement and replication, 74-77
- explained, 33-34
- handling failure, 83-85
- HDFS architecture, 67-72
- HDInsight services, 225-227
- high availability, 222
- metadata storage, 221-222
- writing to HDFS, 77-82
namespaces, 101
naming Data Source Names (DSNs), 375
NAS (Network Attached Storage), 96
NDFS (Nutch Distributed File System), 65
.NET SDKs, 311-316
Network Attached Storage. See NAS
network connectivity (Sqoop), 446
Network File System. See NFS
New HDInsight Cluster Wizard, 187
New-AzureHDInsightCluster cmdlet, 427
New-AzureHDInsightHiveJob Definition cmdlet, 428
New-AzureHDInsightPigJob Definition cmdlet, 440
New-AzureHDInsightSqoopJob Definition cmdlet, 451
NFS (Network File System), shared storage using, 95-97
Nimbus, 520
Node Manager, 225
failures, 172
Node Manager (YARN), 166-167
nodes
clusters (Tez), 318
head nodes, 224
name nodes, 220
HDInsight services, 225-227
high availability, 222
metadata storage, 221-222
secondary name nodes, 221-222
standby name nodes, 222
NoSQL
benefits, 24-25
business value proposition, 26
limitations, 25
RDBMS versus, 17-18
types of technologies, 19
column-store databases, 21-22
document-oriented databases, 22-24
graph databases, 24
key-value store databases, 20-21
usage scenarios, 27-29
number of Mappers, 127-128
Nutch Distributed File System (NDFS), 65
on-time flight departure analysis scenario (SSIS), 424-427
package variables, 426
prerequisites, 425-426
Windows Azure PowerShell automation, 426-427
Oozie, 40, 49, 431, 455-456
flight departure analysis scenario, 457-463
defining workflows, 459-461
executing workflows, 461
monitoring workflows, 461
prerequisites, 458-459
querying results, 462-463
SSIS versus, 465
workflows, 456
coordinating, 464-465
submitting with HDInsight .NET SDK, 463-464
Optimized Row Columnar (ORC) files, 320
optimizing
compression, 150-151
MapReduce, 150-154
uber-tasking optimization (YARN), 170-171
output of Pig jobs, 442-443
OutputFormat class (MapReduce), 123
O
objects
Bloom Filter, 149-150
Context (MapReduce), 139-140
ODBC Data Source Administrator (64-Bit) dialog box, 373
OLTP (OnLine Transaction Processing), 19. See also HBase
OnLine Transaction Processing. See OLTP
on-premises deployment, 58
on-time flight departure analysis scenario (Oozie), 457-463
defining workflows, 459-461
executing workflows, 461
monitoring workflows, 461
prerequisites, 458-459
querying results, 462-463
P
PaaS (Platform as a Service), 179
parallelism
MapReduce, 115
in Pig Latin scripts, 440
in Sqoop, 446
PARTITIONED BY clause, 340
partitioning
dynamic partition inserts, 355-358
MapReduce, 128-130
switching/swapping, 354
tables, 350-354
pass-through modes, 414
pay as you go (PAYG) model, 180
PDW (Microsoft SQL Server Parallel Data Warehouse), 58
Pegasus, 41, 48
performance, 161
blob storage, 242-243
MapReduce, 150-154
Tez, 319
per-MapReduce job blacklisting, 155
Pig, 37-38, 47, 94
canceled flights scenario, 434-438
defining relations, 435-436
filtering relations, 436-437
grouping relations, 437
joining relations, 437
saving relations to file system, 438
summarizing relations, 437
uploading data, 434-435
HCatalog, 438-440
joins, 148
Pig Latin scripting language, 433-434
submitting jobs with PowerShell, 440-443
adding Azure accounts, 441
creating job definitions, 441
execution of jobs, 441-442
generating job output, 442-443
Pig Latin scripting language, 37-38, 433-434
HCatalog usage, 438-440
HiveQL versus, 38
plymr package, 41
PolyBase, 58
Power BI, 48, 378-379
Power Map, 54
reporting, 405-406
visualization, 405-406
Power Query, 54
Excel, launching, 417-421
HDInsight, accessing, 417-421
Power View, 54
reporting, 403-404
visualization, 403-404
PowerPivot, 48, 54
data visualization phase, 390-399
enabling, 386-390
Hive, accessing, 385-406
reporting, 390-399
PowerShell, 237-239
Apache Hive, scripting, 295-299
automation setup, 426-427
HDInsight
automating provisioning, 199-212
cluster creation with HBase, 547-550
verifying setup, 207-212
HDInsight cluster provisioning, 427-431
configuring Hive ODBC data source, 429-430
executing Hive queries, 428
executing SSIS package, 430-431
Script Action usage, 281
Sqoop and, 451-452
submitting Pig jobs, 440-443
adding Azure accounts, 441
creating job definitions, 441
execution of jobs, 441-442
generating job output, 442-443
predictive analysis, 11, 492. See also Azure ML
predictive models (Azure ML), 506-512
prescriptive analysis, 11
pricing plans of Microsoft Azure, 180
private distributed cache files, 146
programming
HDInsight .NET SDKs, 311-316
Hive
in HDInsight, 323-331
tables, 331-340
MapReduce
in C#, 270-273
Hello World, 255-256
jobs, 255
models for Spark, 481-483
Serializer and Deserializer (SerDe), 340-343
properties
Hive job, viewing, 308
viewing, 304-306
provisioning HDInsight, 186-198, 427-431
automating PowerShell, 199-212
clusters, 203-206
Hive
ODBC data source, configuring, 429-430
queries, executing, 428
SSIS package, executing, 430-431
Storm clusters, 521
public distributed cache files, 146
publish settings, downloading and importing, 426-427
publishing Azure ML as web services, 512-514
put parameter, 91

Q
QJM (Quorum Journal Manager), storage using, 97-98
queries
data analysis, 348-354
Hive
executing, 288, 313-314
tables, 346-363
writing, 309
HiveQL, executing, 294
specifying, 393
Query Wizard, 382
queue storage (Microsoft Azure), 234
Quorum Journal Manager. See QJM
quorum-based storage, 228

R
R language, 41, 467
data analysis, 470
data types, 468-469
enabling
on Hadoop, 471-472
on HDInsight, 472-476
installing
on HDInsight, 472-473
on Windows, 468
rack awareness, 88-89
RDBMS (Relational Database Management System), 288
Hive, comparing, 327-328
indexes in, 365
NoSQL versus, 17-18
RDDs (resilient distributed datasets), 482-483
reading
data, Azure ML, 492-502
from HBase tables, 541
from HDFS, 82-83
RecordReader interface, 122
records, handling bad, 157
reduce function, 130
Reduce function (MapReduce), 114
Reducer class (MapReduce), 130-132
reduce-side joins, 149
refreshing PowerPivot models, 394
regions (HBase), 539
load balancing, 543
regression algorithms, 492
Relational Database Management System. See RDBMS
relations (Pig)

defining, 435-436
filtering, 436-437
joining, 437
putting to file system, 438
summarizing, 437
relationships, many-to-many, 397
remote desktop connections, enabling HDInsight, 193-195
Remove-AzureHDInsightCluster cmdlet, 428
replication
deleting files and, 86-87
HDFS blocks, 74-77
reporting
Excel, 399-402
Power Map, 405-406
Power View, 403-404
PowerPivot, 390-399
requests, job request/response flow (MapReduce), 117-118
Research and Innovative Technology Administration (RITA), 323
resilient distributed datasets (RDDs), 482-483
resource management, 160. See also YARN
Resource Manager, 163, 225
failures, 172
HA (High Availability), 173-175
responses, job request/response flow (MapReduce), 117-118
reuse of JVMs in YARN, 152
RHadoop, 41, 48-49
rbase package, 41, 472
rhdfs package, 41, 472
rm parameter, 92
rmr package, 41, 472-476
rmr parameter, 92
ROW FORMAT DELIMITED FIELDS TERMINATED BY clause, 338
row-store databases, column-store databases versus, 536
rows, HBase data distribution and storage, 541-544
run function, 125, 130
running Giraph jobs, 282-284
MapReduce jobs, 249-250

S
S language, 467
safe mode (HDFS), 83-85
saving Pig relations to file systems, 438
Scala, 484
scalability, 161
  of Hadoop, 32, 44-45
  of HDFS, 66
  MapReduce, 115
  of YARN, 45
Scheduler (Resource Manager), 163
scheduling
  Capacity Scheduler, 153-154
  Fair Scheduler, 152-153
  jobs, 152
schema-first systems, 17
schema-later systems, 18
schemas
  databases, comparing, 328-331, 411
  on read approach, 288
SCP (Stream Computing Platform), 522-523
Script Action
  with Azure Management Portal, 280-281
  with HDInsight .NET SDK, 282
  running Giraph job, 282-284
  script development, 278-280
  testing, 285
  version requirements, 278
  with Windows PowerShell, 281
  scripting PowerShell (Apache Hive), 295-299
secondary name nodes, 221-222
  Checkpoint process, 72
  explained, 34, 69
  standby name nodes versus, 222
secure gateway nodes, 183
segregation of duties, 161
SELECT command, 342
semi-structured data, 12
semi-structured scalar data, serialization, 262
serialization frameworks (Hadoop), 262-270
Serializer and Deserializer (SerDe), 340-343
Server Explorer, 304
servers, linking SQL Server, 406-412
service-level agreements (SLAs), 153
services
  HDInsight. See also HDInsight name node, 225-227
  overview of, 181-186
  Microsoft Azure, 180
  Storage, 181
setup function, 125, 130
  HDInsight configuration, verifying, 196-198
shared storage using NFS (Network File System), 95-97
SharePoint, 48
  Power View in, 405
Shark, 485
SHOW DATABASES command, 328
show tables command, 291
shuffle grouping, 521
shuffling map-outputs, 129
signals, heartbeat, 97, 154
single node clusters, 36
Sirosh, Joseph, 56, 493
Slicer controls, adding, 401
snapshots, 45
  HDFS (Hadoop Distributed File System), 103-107
social network analysis, 27
sources of data. See data sources
Spark, 479-480
  HDInsight, installing on, 480-481
  log mining, 483-485
  MapReduce versus, 479, 481-483
  programming model, 481-483
Spark SQL, 485
  crime data analysis scenario, 486-489
  Hive versus, 486
speculative execution (MapReduce), 156-157
speed limit violations, analyzing, 524-532
spouts, 520
SQL (Structured Query Language), 183. See also NoSQL; SQL Servers
  crime data analysis scenario, 486-489
swapping partitioning 569

Hive versus, 486
Spark SQL, 485
SQL Azure tables, creating, 530-531
SQL Servers
Analysis Services (SSAS), 48, 54, 413-415
data integration with HDInsight, importance of, 423
Hive, accessing, 406-417
importing data
to HDInsight clusters, 445-447
to Sqoop, 448-451
Integration Services. See SSIS (SQL Server Integration Services)
linking, 406-412
Management Studio (SSMS), 302, 410
parallel data warehouse, 47
RDBMS, 47
Reporting Services (SSRS), 49, 54, 415-417
StreamInsight, 46, 48
Sqoop, 38, 46, 94, 445
exporting data
exercise, 448-451
export process, 448
from HDFS, 447-448
with PowerShell, 451-452
importing data
exercise, 448-451
to HDInsight clusters, 445-447
to Hive, 447
with PowerShell, 451-452
network connectivity, 446
SSAS (SQL Server Analysis Services), 48, 413-415
SSIS (SQL Server Integration Services), 47, 54, 423
ETL workflow packages, creating, 424
flight departure analysis scenario, 424-427
package variables, 426
prerequisites, 425-426
Windows Azure PowerShell automation, 426-427
HDInsight clusters, provisioning, 427-431
Oozie versus, 465
SSMS (SQL Server Management Studio), 410
SSRS (SQL Server Reporting Services), 415-417
standard input stream (STDIN), 146
standard output stream (STDOUT), 146
standby name nodes, 222
secondary name nodes versus, 222
Start-AzureHDInsightJob cmdlet, 441, 452
statistics
Descriptive Statistics module, 505
storage
accounts, creating, 192
blocks (HDFS), 101
emulator (Microsoft Azure), configuring, 251-253
NAS (Network Attached Storage), 96
QJM (Quorum Journal Manager), 97-98
services, 181, 233-234.
See also specific types of storage (e.g. blob storage, table storage)
shared using NFS, 95-97
STORED AS clause, 338
Storm, 519-522
applications, creating, 524-529
architecture, 520-522
HDInsight, provisioning clusters, 521
SCP (Stream Computing Platform), 522-523
speed limit violations, analyzing, 524-532
SQL Azure tables, creating, 530-531
topologies
creating, 529-530
submitting, 531
stream analytics, 519. See also Storm
SCP (Stream Computing Platform), 522-523
speed limit violations, analyzing, 524-532
Stream Computing Platform (SCP), 522-523
streaming
access in HDFS, 67
Hadoop, 146-148, 270-275
structured data, 11
submitting topologies (HDInsight), 531
subscriptions, Microsoft Azure, 180, 201
summarizing Pig relations, 437
Supervisor, 520
swapping partitioning, 354
switching partitioning, 354
System Center Operations Manager, 49

T
Table Import Wizard, 389, 391

- tables
  - columns (in Hive), 338
  - CREATE TABLE command, 338
  - external tables, programming, 335-338
  - HBase
    - compaction, 544-545
    - creating, 539
    - data distribution and storage, 541-544
    - reading from, 541
    - writing to, 540-541
  - Hive
    - creating, 293, 309-321
    - data load processes for, 343-346
    - programming, 331-340
    - queries, 346-363
  - importing, 389
  - internal tables, programming, 331-335
  - partitioning, 350-354
  - properties, viewing, 304-306
  - SQL Azure tables, creating, 530-531
  - storage (Microsoft Azure), 20-21, 234
  - temporary, 342
  - views, creating, 342-343
- tabular data, serialization, 262

- targeted campaigning, 27
- task failures
  - MapReduce, 155-156
  - YARN, 171
- Task Tracker, 225
  - data node co-location, 118-119
- tasks (SSIS), 424
- TaskTrackers, 35
  - failure, 154-155
  - MapReduce, 115-116
- TCP/IP, 82
  - Team Foundation Server, 49
- temporary tables, 342
- testing
  - requests/responses, 513
  - Script Action, 285
- TextInputFormat class, 122
- Tez
  - executing, 365-368
  - execution engines, 320
  - overview of, 287, 316-320
- throughput of HDFS, 67
- Timeline Server, 165
- Tool interface (MapReduce), 139
- tools
  - CloudXplorer, 324
  - ETL (extraction, transformation, and loading), 417
  - for Visual Studio (HDInsight), 302-311
- topologies, Apache Storm, 520
  - creating, 529-530
  - submitting, 531
- Trash folder (HDFS), 86-87
- TRUNCATE TABLE command, 331
- Try It Yourself. See exercises

U
uber-tasking optimization (YARN), 170-171
uniform resource identifier (URI), 91
unstructured data, 11-12
uploading
  - to default file system, 239
  - to HDInsight clusters, 434-435
URI (uniform resource identifier), 91
usage scenarios for Big Data, 27-29
user-defined aggregating functions (UDAFs), 345
user-defined functions (UDFs), 345
user-defined table-generating functions (UDTFs), 345

V
variability characteristics of Big Data, 14
variables (SSIS), 424
  - flight departure analysis scenario, 426
variety characteristics of Big Data, 13
vectors (R), 468
velocity characteristics of Big Data, 13

vendors for Hadoop
Amazon Elastic MapReduce (EMR), 51-52
Cloudera, 50
Hortonworks, 50-51
MapR Technologies, 51
Microsoft Big Data solutions, 52-57

verifying
HDInsight
cluster creation, 550-551
configuration, 196-198
connections, 389
emulator installation, 248-249
PowerShell setup, 207-212
relationships (Azure/PowerShell), 202

versioning in HBase, 536

viewing
Azure ML, 499
Hive jobs, 306-308
properties, 304-306
views, creating tables, 342-343
virtual machines in Microsoft Azure, 60
Virtual Network (VNET), 189
virtual networks in Azure, 545-546
Visual Studio, 49
HDInsight tools for, 302-311

visualization
Azure ML, 505
Excel, 399-402
Power Map, 405-406
Power View, 403-404
PowerPivot, 390-399

volume characteristics of Big Data, 13

W

Wait-AzureHDInsightJob cmdlet, 442
WASB (Windows Azure Storage—Blob), 183-184, 240-241, 300
weather forecasting, 28
web services, publishing as (Azure ML), 512-514
WebHDFS, 89-90
WHERE clause, 347
Windows
Hadoop on, 52-57
installing R, 468
windows, Hive Job Browser, 307
Windows Azure Storage—Blob (WASB), 240-241
Windows PowerShell, 237-239
automation setup, 426-427
HDInsight cluster creation with HBase, 547-550
HDInsight cluster provisioning, 427-431
configuring Hive ODBC data source, 429-430
executing Hive queries, 428
executing SSIS package, 430-431
Script Action usage, 281
Sqoop and, 451-452
submitting Pig jobs, 440-443
adding Azure accounts, 441
creating job definitions, 441
execution of jobs, 441-442
getting job output, 442-443
WITH clause, 347

wizards
Create New Data Source Wizard, 374, 378
From Data Connection Wizard, 379
HDInsight Provision Wizard, 189
Import Wizard, 388
New HDInsight Cluster Wizard, 187
Query Wizard, 382
Table Import Wizard, 389, 391

word counts, 126, 131, 135

worker nodes, 183

workflows
Azure ML, 498-505
coordinating, 464-465
flight departure analysis scenario, 457-463
defining workflows, 459-461
executing workflows, 461
monitoring workflows, 461
prerequisites, 458-459
querying results, 462-463
Oozie, 456
submitting with HDInsight .NET SDK, 463-464

workspaces (Azure ML), 495-497

WritableComparableT interface, 127

write once, read many, 66

writing
to HBase tables, 540-541
to HDFS, 77-82
Hive queries, 309
Y

YARN
  automatic failover, 173-175
  components of, 162-166
  failures in, 171-173
  job execution flow in, 167-169
  MapReduce versus, 45-46
  Node Manager, 166-167
  overview of, 160-169
  reuse of JVMs in, 152
  uber-tasking optimization, 170-171

Z

ZKFC (ZooKeeper Failover Controller), 99, 228-230
ZooKeeper
  HBase and, 538-539
  nodes, 183
  quorums, configuring, 99