Manpreet Singh
Arshad Ali

Sams Teach Yourself

Big Data Analytics with Microsoft HDInsight® in 24 Hours

SAMS

Arshad Ali
Manpreet Singh

Sams **Teach Yourself**

Big Data Analytics with Microsoft HDInsight® in 24 Hours

SAMS 800 East 96th Street, Indianapolis, Indiana, 46240 USA

Sams Teach Yourself Big Data Analytics with Microsoft HDInsight® in 24 Hours

Copyright © 2016 by Pearson Education, Inc.

ISBN-13: 978-0-672-33727-7
ISBN-10: 0-672-33727-4

Library of Congress Control Number: 2015914167

Printed in the United States of America

First Printing November 2015

Editor-in-Chief
Greg Wiegand

Acquisitions Editor
Joan Murray

Development Editor
Sondra Scott

Managing Editor
Sandra Schroeder

Senior Project Editor
Tonya Simpson

Copy Editor
Krista Hansing Editorial Services, Inc.

Senior Indexer
Cheryl Lenser

Proofreader
Anne Goebel

Technical Editors
Shayne Burgess
Ron Abellera

Publishing Coordinator
Cindy Teeters

Cover Designer
Mark Shirar

Compositor
codeMantra

Contents at a Glance

Table of Contents

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 Adholia, 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 Abellera 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!

Reader Services

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.

We hope you enjoy reading this book and gain an understanding of and expertise on Big Data and Big Data analytics. We especially hope you learn how to leverage Microsoft HDInsight to exploit its enormous opportunities to take your organization way ahead of your competitors.

We would love to hear your feedback or suggestions for improvement. Feel free to share with us (Arshad Ali, arshad.ali@live.in, and Manpreet Singh, manpreet.singh3@hotmail.com) so that we can incorporate it into the next release. Welcome to the world of Big Data and Big Data analytics with Microsoft HDInsight!

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 books 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.
- 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.
- 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.
- 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.
- 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

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

HOUR 3

Hadoop Distributed File System Versions 1.0 and 2.0

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 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.

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.

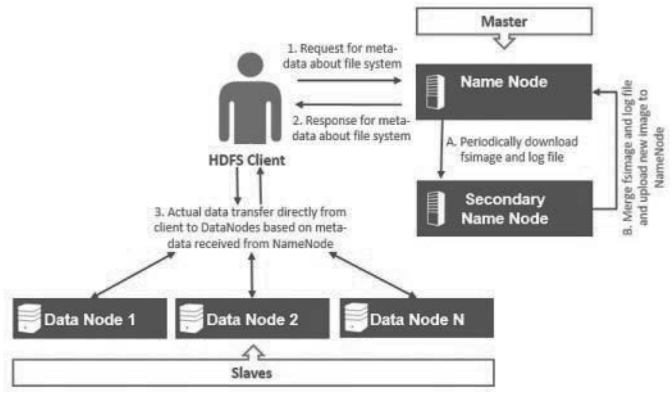

Master
1. Request for meta-data about file system
2. Response for meta-data about file system
HDFS Client
Name Node
A. Periodically download fsimage and log file
Secondary Name Node
B. Merge fsimage and log file and upload new image to NameNode
3. Actual data transfer directly from client to DataNodes based on meta-data received from NameNode
Data Node 1
Data Node 2
Data Node N
Slaves

FIGURE 3.1
How a client reads and writes to and from HDFS.

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.

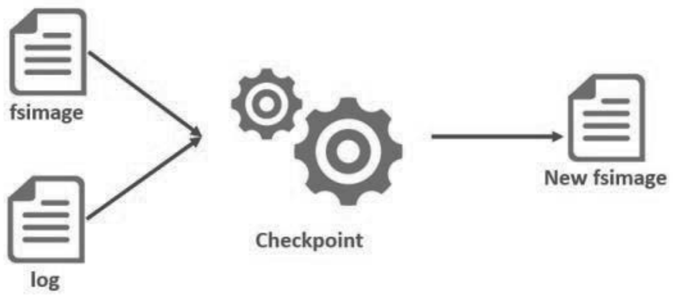


FIGURE 3.2
Checkpoint process.

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

Setting	Location	Description
`fs.checkpoint.dir`	c:\hadoop\HDFS\2nn	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.
`fs.checkpoint.edits.dir`	c:\hadoop\HDFS\2nn	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`.
`fs.checkpoint.period`	86400	The number of seconds between two periodic Checkpoints.
`fs.checkpoint.size`	2048000000	The size of the current edit log (in bytes) that triggers a periodic Checkpoint even if the `fs.checkpoint.period` hasn't expired.

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

Setting	Description
`dfs.namenode.checkpoint.dir`	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.
`dfs.namenode.checkpoint.edits.dir`	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`.
`dfs.namenode.checkpoint.period`	The number of seconds between two periodic checkpoints.

Setting	Description
`dfs.namenode.checkpoint.txns`	The secondary name node or checkpoint node will create a checkpoint of the namespace every `dfs.namenode.checkpoint.txns` transactions, regardless of whether `dfs.namenode.checkpoint.period` has expired.
`dfs.namenode.checkpoint.check.period`	The secondary name node and checkpoint node will poll the name node every `dfs.namenode.checkpoint.check.period` seconds to query the number of uncheckpointed transactions.
`dfs.namenode.checkpoint.max-retries`	The secondary name node retries failed checkpointing. If the failure occurs while loading fsimage or replaying edits, the number of retries is limited by this variable.
`dfs.namenode.num.checkpoints.retained`	The number of image checkpoint files that will be retained by the name node and secondary name node in their storage directories. All edit logs necessary to recover an up-to-date namespace from the oldest retained checkpoint will also be retained.

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.

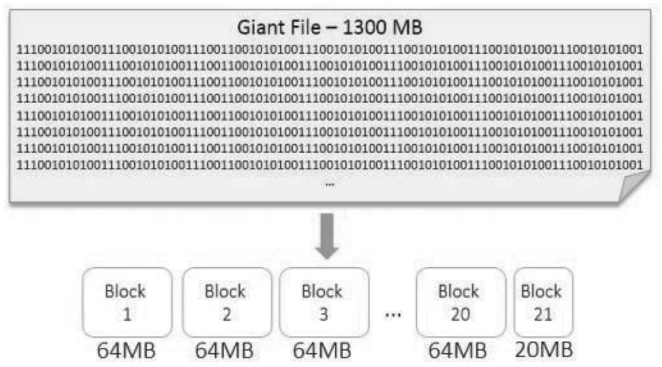


FIGURE 3.3
File split process when writing to HDFS.

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 3.3 Block Size Configuration

Name	Value	Description
`dfs.blocksize`	134217728	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.

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 3.4 Block Replication Configuration

Name	Value	Description
`dfs.replication`	3	Default block replication. The actual number of replications can be specified when the file is created. The default is used if replication is not specified in create time.
`dfs.replication.max`	512	Maximum block replication.
`dfs.namenode.replication.min`	1	Minimal block replication.

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.

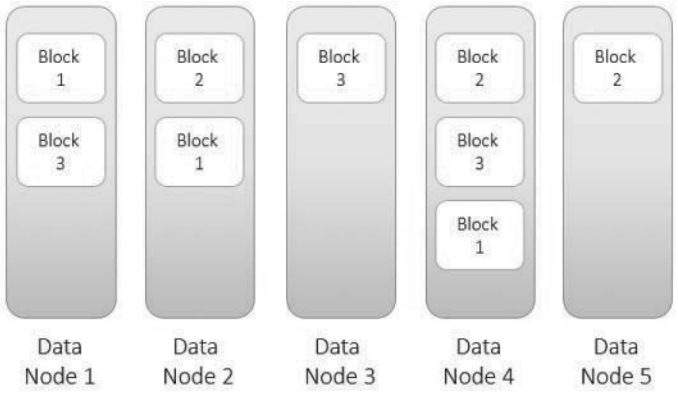


FIGURE 3.4
Data block placement on data nodes.

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).

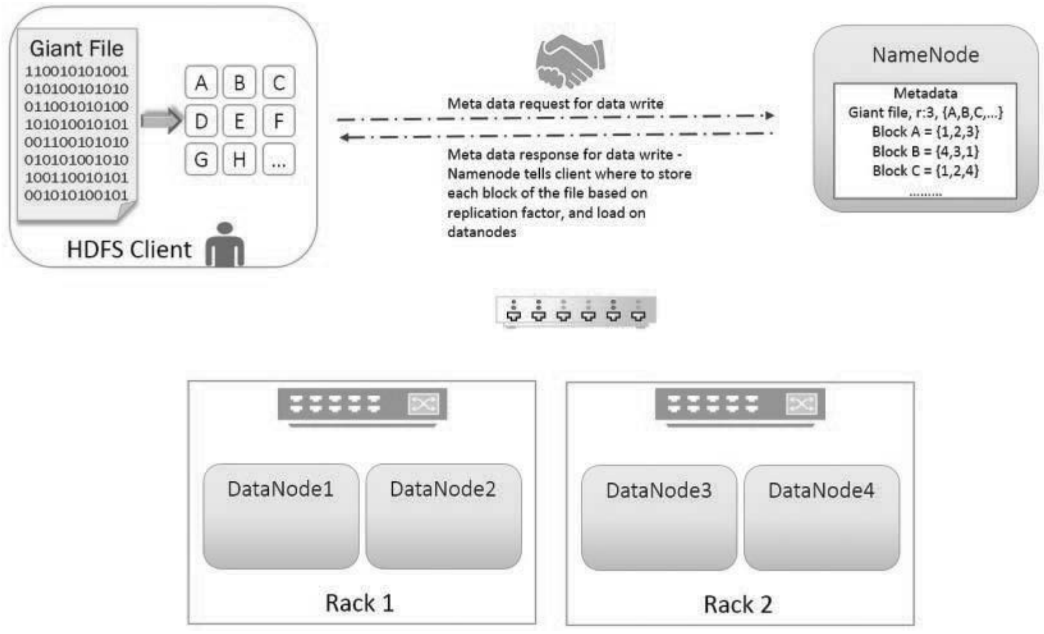


FIGURE 3.5
The client talks to the name node for metadata to specify where to place the data blocks.

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 the 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.

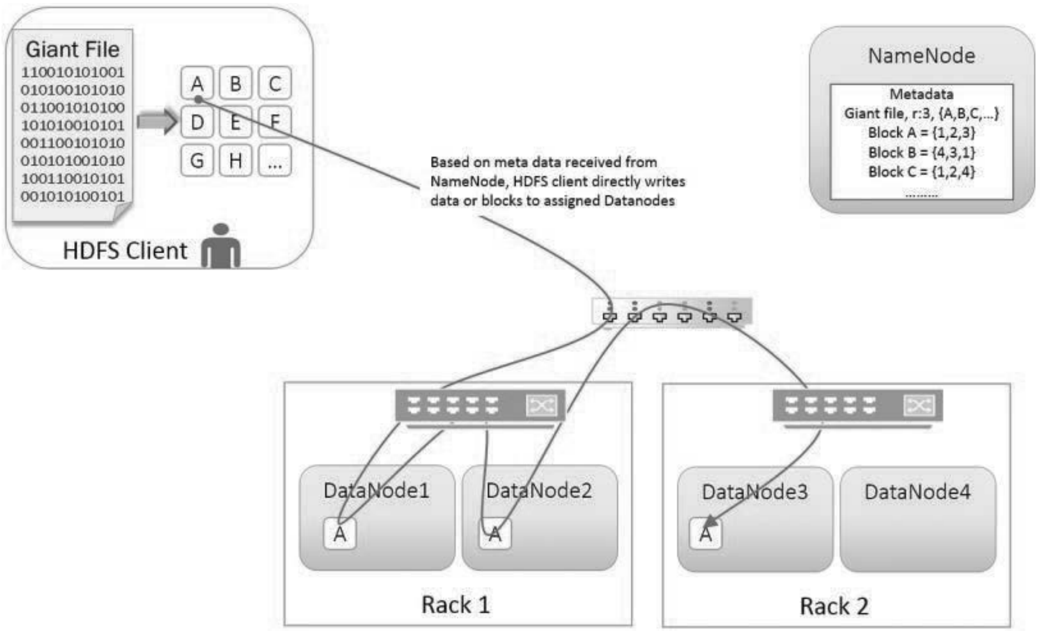


FIGURE 3.6
The client sends data blocks to identified data nodes.

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).

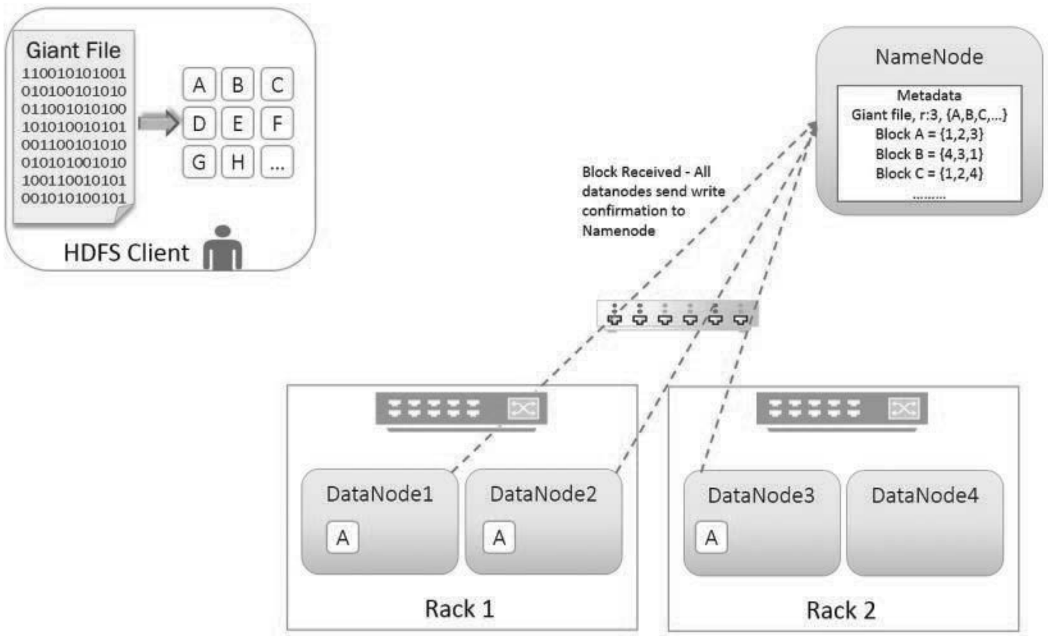


FIGURE 3.7
Data nodes update the name node about receipt of the data blocks.

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).

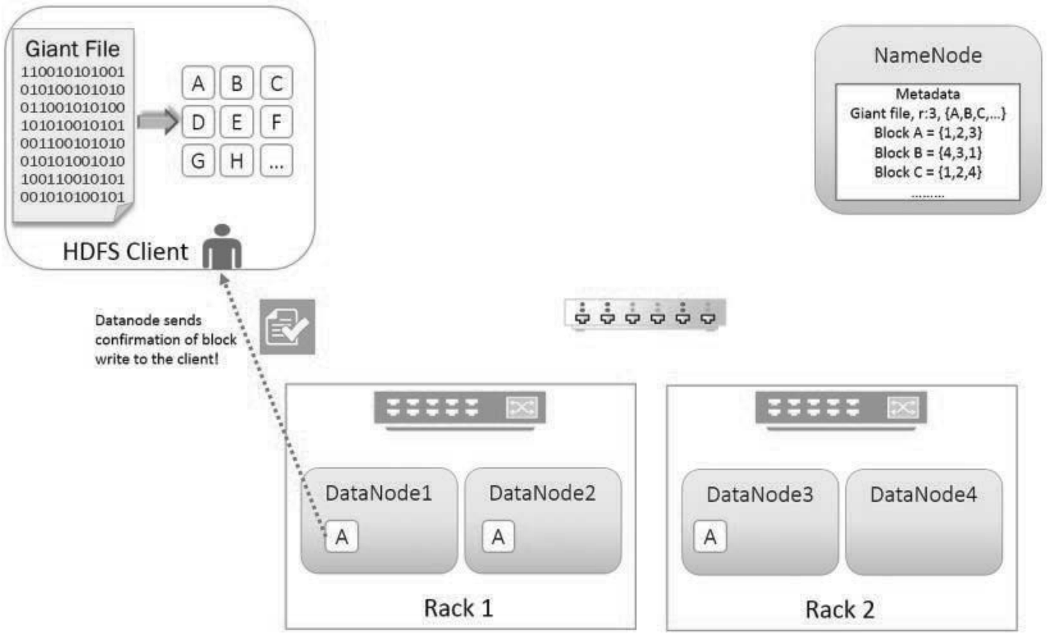


FIGURE 3.8
The first data node sends an acknowledgment back to the client.

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.

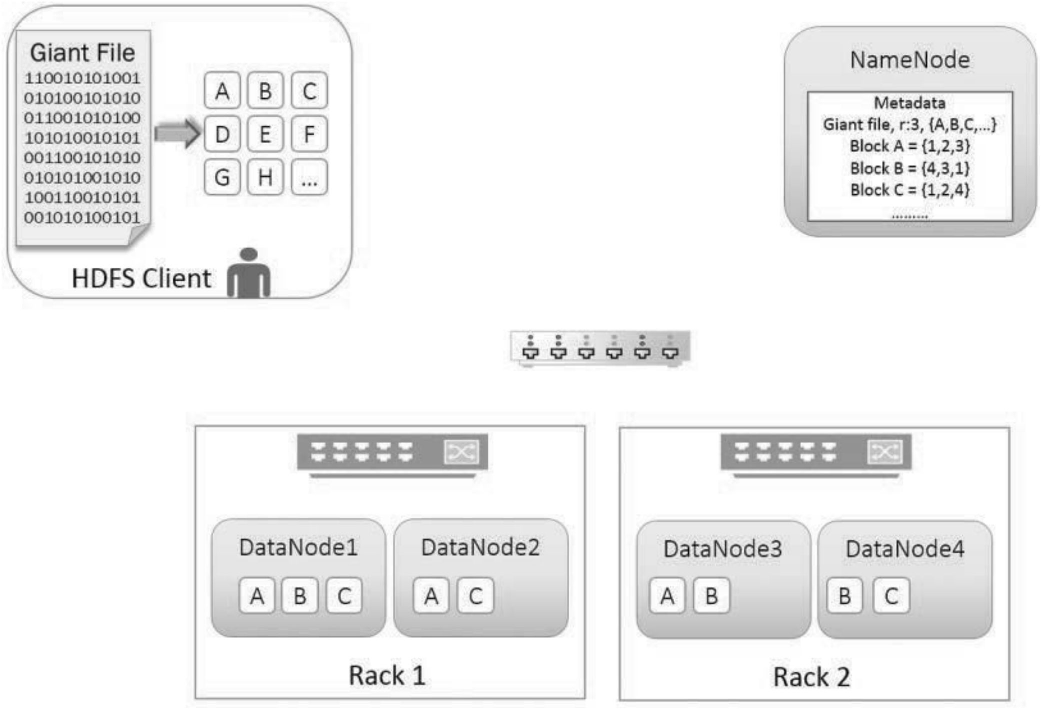


FIGURE 3.9
All data blocks are placed in a similar way.

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).

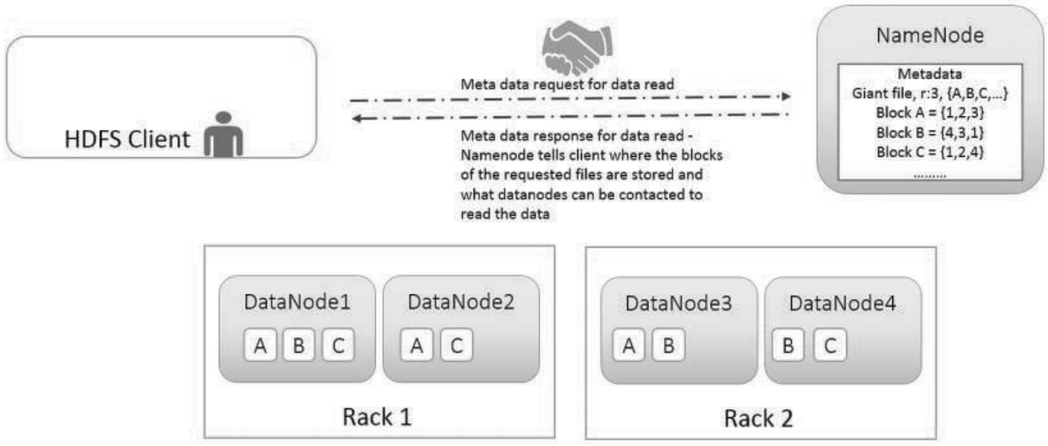

HDFS Client
Meta data request for data read
Meta data response for data read - Namenode tells client where the blocks of the requested files are stored and what datanodes can be contacted to read the data
NameNode
Metadata
Giant file, r:3, {A,B,C,...}
Block A = {1,2,3}
Block B = {4,3,1}
Block C = {1,2,4}
....
DataNode1
A B C
DataNode2
A C
Rack 1
DataNode3
A B
DataNode4
B C
Rack 2

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).

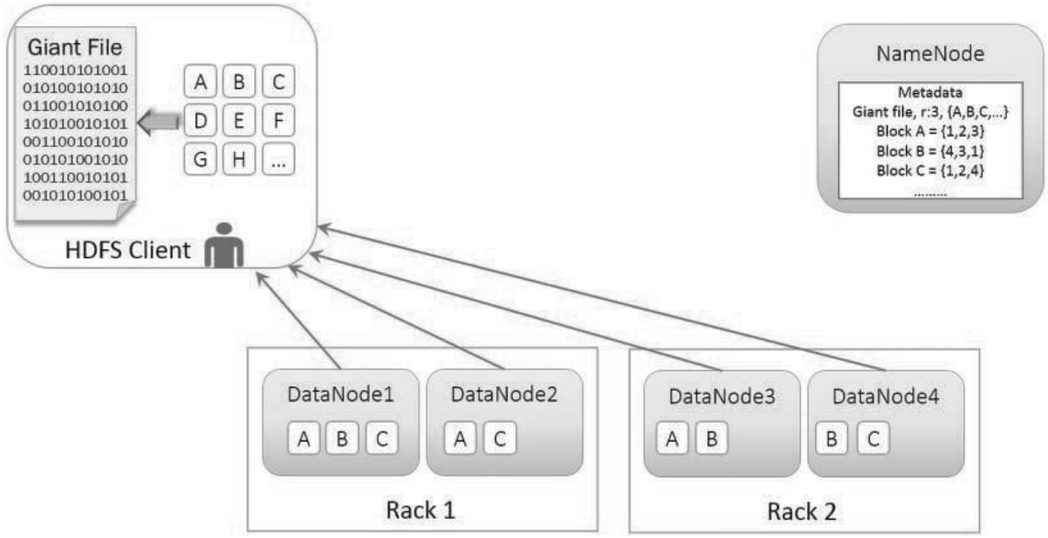


FIGURE 3.11
The client starts reading data blocks of the file from the identified data nodes.

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.

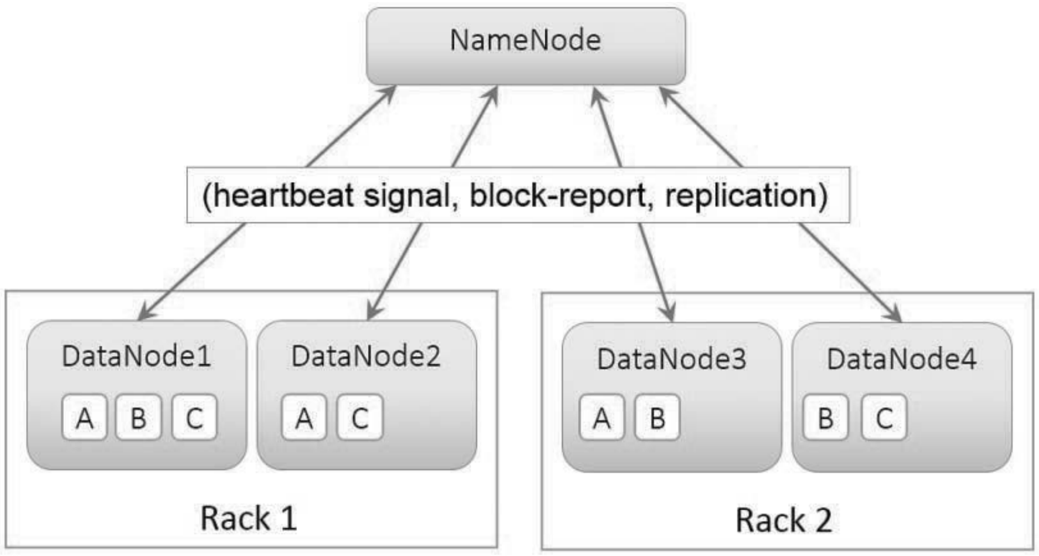


FIGURE 3.12
All data nodes periodically send heartbeat signals to the name node.

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.

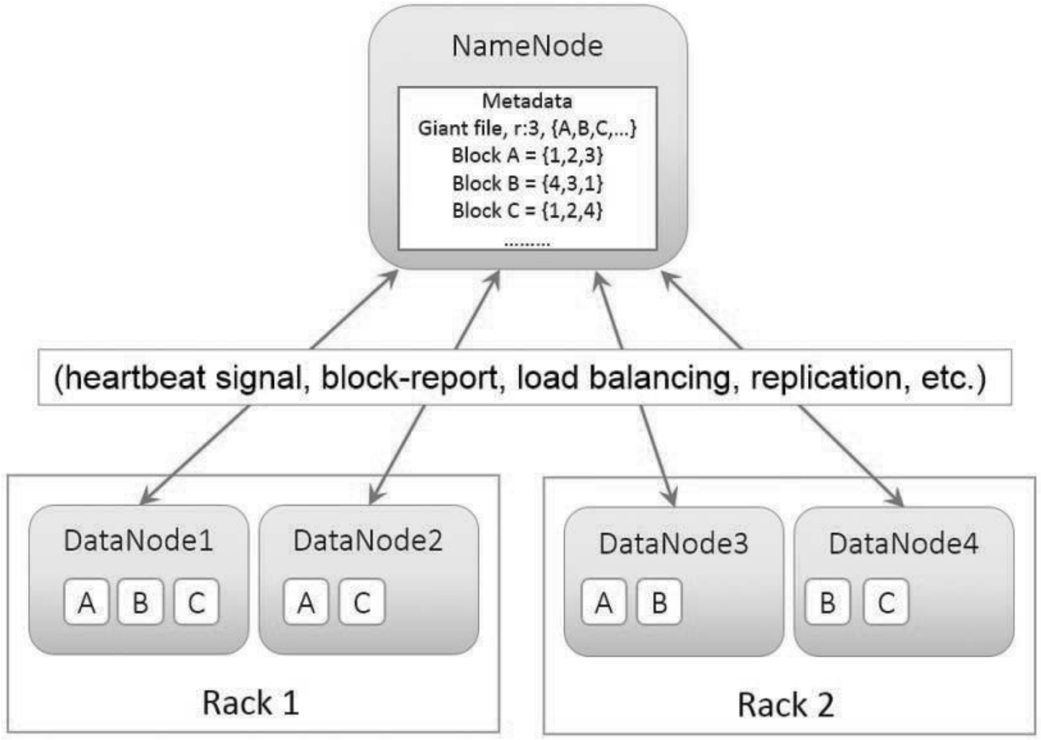


FIGURE 3.13
The name node updates its metadata based on information it receives from the data nodes.

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.

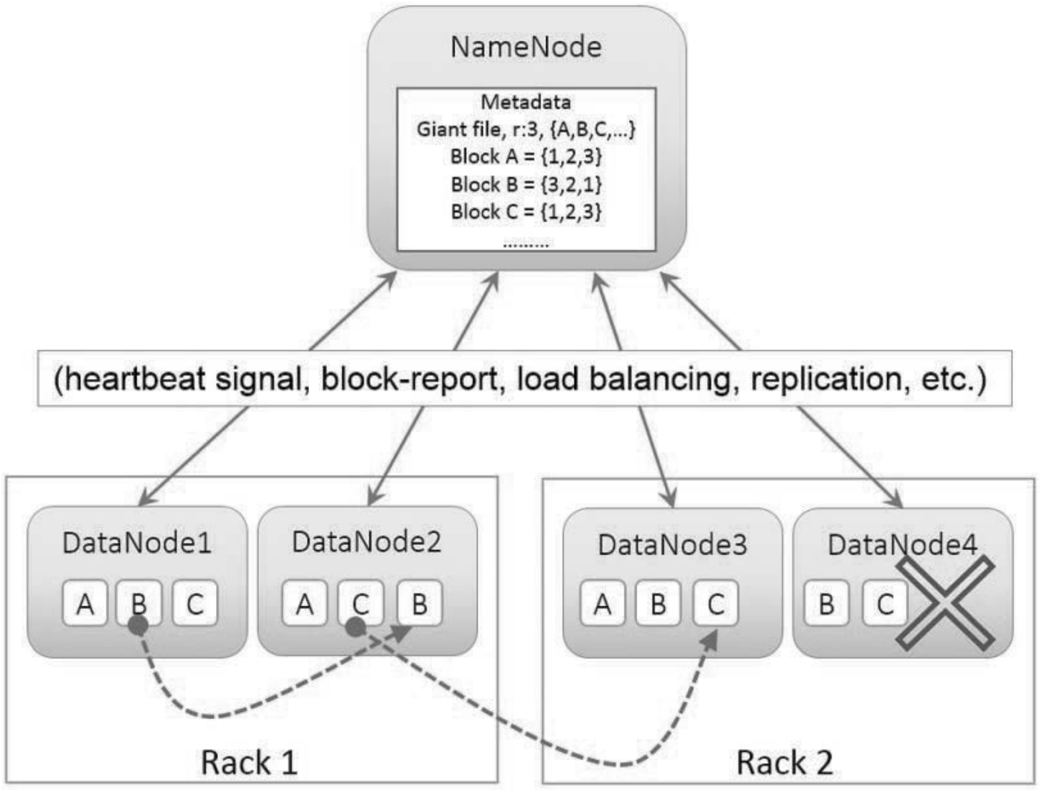


FIGURE 3.14
Handling a data node failure transparently.

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

Name	Description
`fs.trash.interval`	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.
`fs.trash.checkpoint.interval`	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.

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 that 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:

```
hadoopdn001    /hadoop/rack1
hadoopdn002    /hadoop/rack1
hadoopdn006    /hadoop/rack2
hadoopdn007    /hadoop/rack2
```

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.

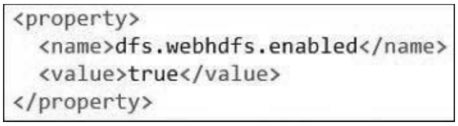
WebHDFS

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>
```

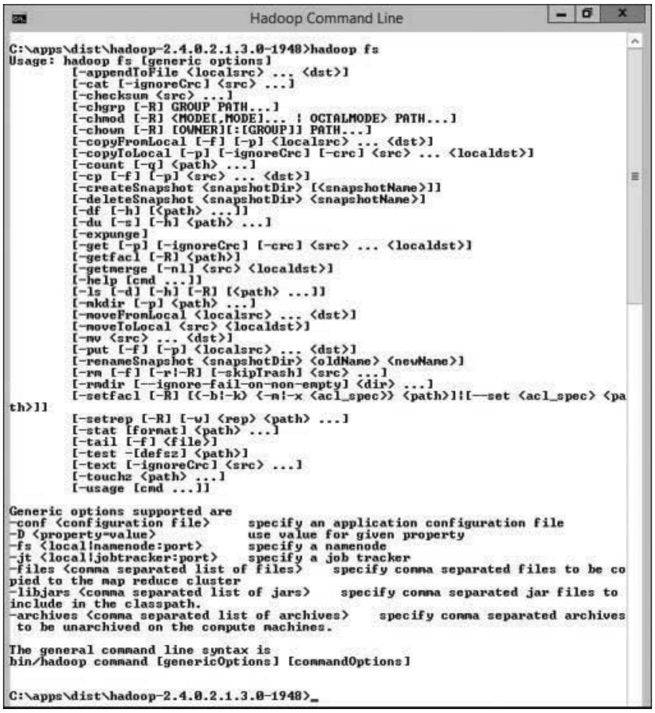
FIGURE 3.15
WebHDFS-related configuration.

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.



```
C:\apps\dist\hadoop-2.4.0.2.1.3.0-1948>hadoop fs
Usage: hadoop fs [generic options]
        [-appendToFile <localsrc> ... <dst>]
        [-cat [-ignoreCrc] <src> ...]
        [-checksum <src> ...]
        [-chgrp [-R] GROUP PATH...]
        [-chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH...]
        [-chown [-R] [OWNER][:[GROUP]] PATH...]
        [-copyFromLocal [-f] [-p] <localsrc> ... <dst>]
        [-copyToLocal [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
        [-count [-q] <path> ...]
        [-cp [-f] [-p] <src> ... <dst>]
        [-createSnapshot <snapshotDir> [<snapshotName>]]
        [-deleteSnapshot <snapshotDir> <snapshotName>]
        [-df [-h] [<path> ...]]
        [-du [-s] [-h] <path> ...]
        [-expunge]
        [-get [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
        [-getfacl [-R] <path>]
        [-getmerge [-nl] <src> <localdst>]
        [-help [cmd ...]]
        [-ls [-d] [-h] [-R] [<path> ...]]
        [-mkdir [-p] <path> ...]
        [-moveFromLocal <localsrc> ... <dst>]
        [-moveToLocal <src> <localdst>]
        [-mv <src> ... <dst>]
        [-put [-f] [-p] <localsrc> ... <dst>]
        [-renameSnapshot <snapshotDir> <oldName> <newName>]
        [-rm [-f] [-r|-R] [-skipTrash] <src> ...]
        [-rmdir [--ignore-fail-on-non-empty] <dir> ...]
        [-setfacl [-R] [{-b|-k} {-m|-x <acl_spec>} <path>]|[--set <acl_spec> <pa
th>]]
        [-setrep [-R] [-w] <rep> <path> ...]
        [-stat [format] <path> ...]
        [-tail [-f] <file>]
        [-test -[defsz] <path>]
        [-text [-ignoreCrc] <src> ...]
        [-touchz <path> ...]
        [-usage [cmd ...]]

Generic options supported are
-conf <configuration file>     specify an application configuration file
-D <property=value>            use value for given property
-fs <local|namenode:port>      specify a namenode
-jt <local|jobtracker:port>    specify a job tracker
-files <comma separated list of files>    specify comma separated files to be co
pied to the map reduce cluster
-libjars <comma separated list of jars>    specify comma separated jar files to
include in the classpath.
-archives <comma separated list of archives>    specify comma separated archives
 to be unarchived on the compute machines.

The general command line syntax is
bin/hadoop command [genericOptions] [commandOptions]

C:\apps\dist\hadoop-2.4.0.2.1.3.0-1948>_
```

FIGURE 3.16
Hadoop CLI for managing the file system.

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.

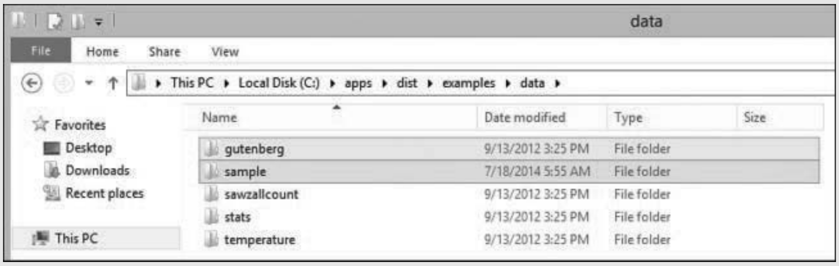
- `put`—Copies one or more files from the local file system (also reads input from stdin) to the destination file system.
- `copyFromLocal`—Copies one or more files from the local file system to the destination file system. The `-f` option overwrites the destination if it already exists.
- `get`—Copies one or more files to the local file system.
- `copyToLocal`—Copies one or more files to the local file system.

- 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.



Name	Date modified	Type	Size
gutenberg	9/13/2012 3:25 PM	File folder	
sample	7/18/2014 5:55 AM	File folder	
sawzallcount	9/13/2012 3:25 PM	File folder	
stats	9/13/2012 3:25 PM	File folder	
temperature	9/13/2012 3:25 PM	File folder	

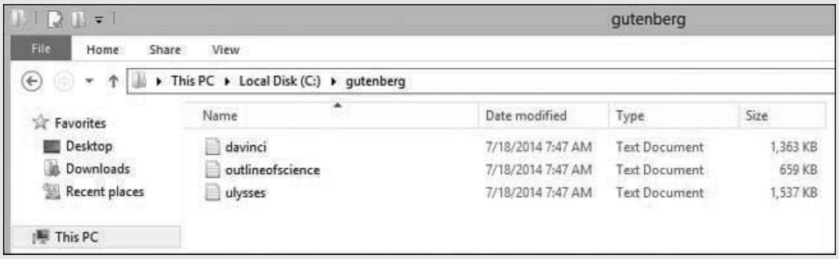
FIGURE 3.17
Source folders and files to copy to HDFS.

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
```

```
11:  hadoop fs -ls /example/sampledata
12:  @rem --- copying a file to local drive
13:  hadoop fs -copyToLocal /example/sampledata/gutenberg c:\gutenberg
```

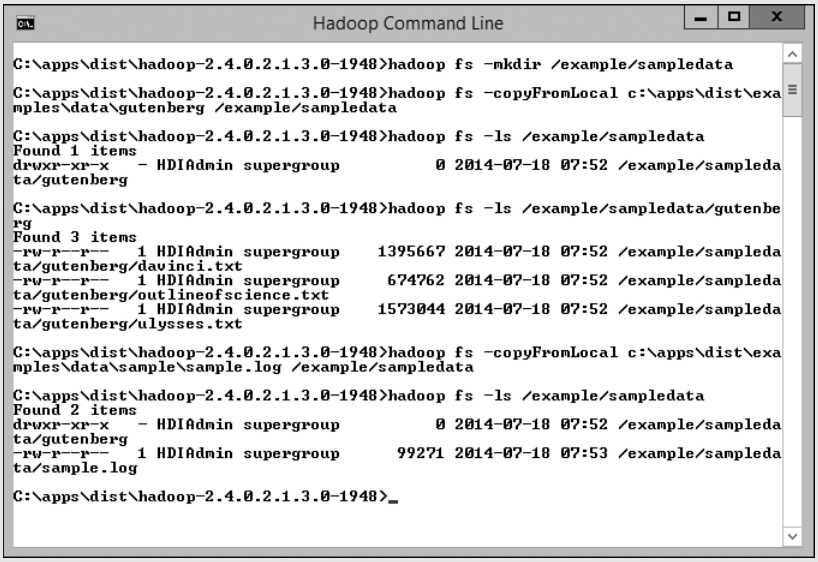
Figure 3.18 shows files copied from HDFS to the local file system using the code from line 13 of Listing 3.1.



Name	Date modified	Type	Size
davinci	7/18/2014 7:47 AM	Text Document	1,363 KB
outlineofscience	7/18/2014 7:47 AM	Text Document	659 KB
ulysses	7/18/2014 7:47 AM	Text Document	1,537 KB

FIGURE 3.18
Folder and files copied back from HDFS to the local file system.

Figure 3.19 shows the execution result of the code from Listing 3.1.



```
C:\apps\dist\hadoop-2.4.0.2.1.3.0-1948>hadoop fs -mkdir /example/sampledata

C:\apps\dist\hadoop-2.4.0.2.1.3.0-1948>hadoop fs -copyFromLocal c:\apps\dist\exa
mples\data\gutenberg /example/sampledata

C:\apps\dist\hadoop-2.4.0.2.1.3.0-1948>hadoop fs -ls /example/sampledata
Found 1 items
drwxr-xr-x   - HDIAdmin supergroup          0 2014-07-18 07:52 /example/sampleda
ta/gutenberg

C:\apps\dist\hadoop-2.4.0.2.1.3.0-1948>hadoop fs -ls /example/sampledata/gutenbe
rg
Found 3 items
-rw-r--r--   1 HDIAdmin supergroup    1395667 2014-07-18 07:52 /example/sampleda
ta/gutenberg/davinci.txt
-rw-r--r--   1 HDIAdmin supergroup     674762 2014-07-18 07:52 /example/sampleda
ta/gutenberg/outlineofscience.txt
-rw-r--r--   1 HDIAdmin supergroup    1573044 2014-07-18 07:52 /example/sampleda
ta/gutenberg/ulysses.txt

C:\apps\dist\hadoop-2.4.0.2.1.3.0-1948>hadoop fs -copyFromLocal c:\apps\dist\exa
mples\data\sample\sample.log /example/sampledata

C:\apps\dist\hadoop-2.4.0.2.1.3.0-1948>hadoop fs -ls /example/sampledata
Found 2 items
drwxr-xr-x   - HDIAdmin supergroup          0 2014-07-18 07:52 /example/sampleda
ta/gutenberg
-rw-r--r--   1 HDIAdmin supergroup      99271 2014-07-18 07:53 /example/sampleda
ta/sample.log

C:\apps\dist\hadoop-2.4.0.2.1.3.0-1948>_
```

FIGURE 3.19
HDFS command execution 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).

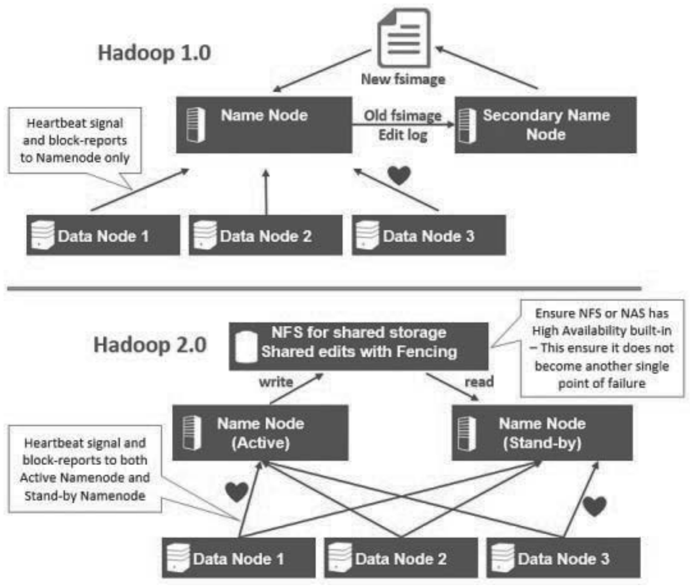

Hadoop 1.0
New fsimage
Heartbeat signal and block-reports to Namenode only
Name Node
Old fsimage
Edit log
Secondary Name Node
Data Node 1
Data Node 2
Data Node 3
Hadoop 2.0
NFS for shared storage Shared edits with Fencing
Ensure NFS or NAS has High Availability built-in – This ensure it does not become another single point of failure
write
read
Heartbeat signal and block-reports to both Active Namenode and Stand-by Namenode
Name Node (Active)
Name Node (Stand-by)
Data Node 1
Data Node 2
Data Node 3

FIGURE 3.20
HDFS high availability with shared storage.

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 synch 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.

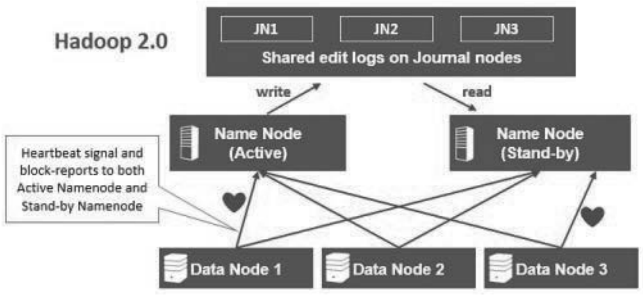

Hadoop 2.0
JN1
JN2
JN3
Shared edit logs on Journal nodes
write
read
Name Node (Active)
Name Node (Stand-by)
Heartbeat signal and block-reports to both Active Namenode and Stand-by Namenode
Data Node 1
Data Node 2
Data Node 3

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.

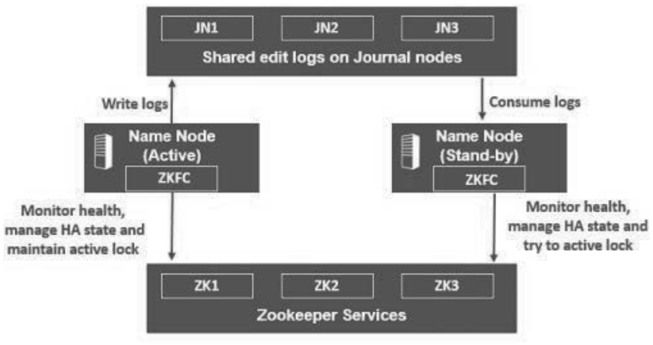

JN1
JN2
JN3
Shared edit logs on Journal nodes
Write logs
Consume logs
Name Node (Active)
ZKFC
Name Node (Stand-by)
ZKFC
Monitor health, manage HA state and maintain active lock
Monitor health, manage HA state and try to active lock
ZK1
ZK2
ZK3
Zookeeper Services

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.

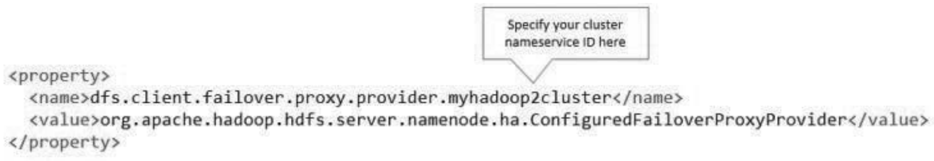
NOTE

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.

GO TO ▶ Refer to **Hour 6, “Getting Started with HDInsight, Provisioning Your HDInsight Service Cluster, and Automating HDInsight Cluster Provisioning,”** and **Hour 7, “Exploring Typical Components of HDFS Cluster,”** for more details about high availability in HDInsight.

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.



```
<property>
  <name>dfs.client.failover.proxy.provider.myhadoop2cluster</name>
  <value>org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider</value>
</property>
```

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.

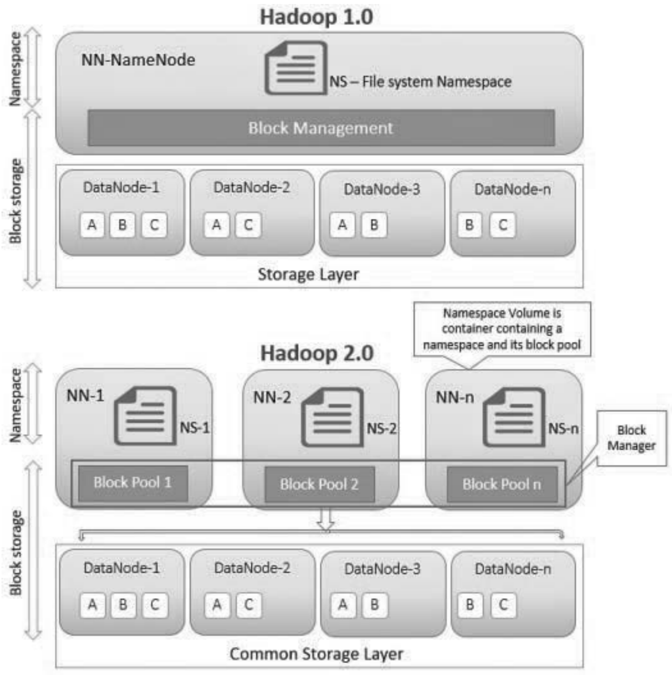


FIGURE 3.24
HDFS Federation and how it works.

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.

NOTE

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).

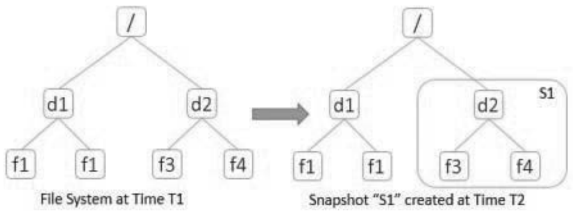


FIGURE 3.25
How HDFS Snapshot works.

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).

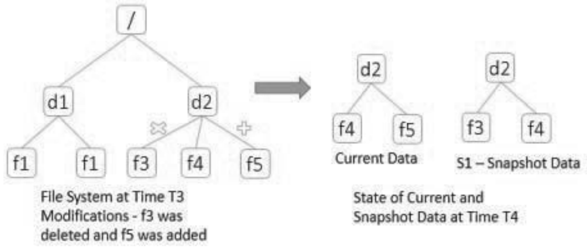


FIGURE 3.26
How changes are applied in HDFS Snapshot.

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.

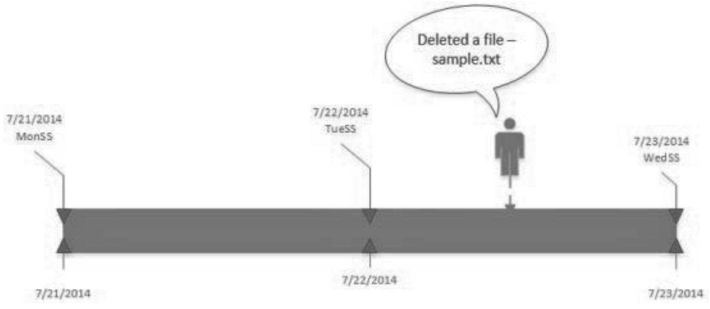


FIGURE 3.27
An example scenario for HDFS Snapshot.

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.

```
hdfs dfs -renameSnapshot <path> <oldSnapshotName> <newSnapshotName>
hadoop dfs -renameSnapshot <path> <oldSnapshotName> <newSnapshotName>
```

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 ensures that the data it receives while reading is not corrupted?
6. Is there a way to recover an accidently 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 is as a service. For example, in HDFS, we have daemons such as name node, data node, and secondary name node.

This page intentionally left blank

Index