Community Experience Distilled

Hadoop Backup and Recovery Solutions

Learn the best strategies for data recovery from Hadoop backup clusters and troubleshoot problems

KNOWARTH
DELIVERING EXCELLENCE

Gaurav Barot Chintan Mehta
Amij Patel

[PACKT] open source* community experience distilled
PUBLISHING

Hadoop Backup and Recovery Solutions

Learn the best strategies for data recovery from Hadoop backup clusters and troubleshoot problems

Gaurav Barot

Chintan Mehta

Amij Patel

[PACKT] PUBLISHING open source
community experience distilled

BIRMINGHAM - MUMBAI

Hadoop Backup and Recovery Solutions

First published: July 2015

Production reference: 1220715

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78328-904-2

`www.packtpub.com`

Credits

Authors
Gaurav Barot
Chintan Mehta
Amij Patel

Reviewers
Skanda Bhargav
Venkat Krishnan
Stefan Matheis
Manjeet Singh Sawhney

Commissioning Editor
Anthony Lowe

Acquisition Editor
Harsha Bharwani

Content Development Editor
Akashdeep Kundu

Technical Editor
Shiny Poojary

Copy Editors
Tani Kothari
Kausambhi Majumdar
Vikrant Phadke

Project Coordinator
Milton Dsouza

Proofreader
Safis Editing

Indexer
Tejal Soni

Graphics
Jason Monteiro

Production Coordinator
Aparna Bhagat

Cover Work
Aparna Bhagat

About the Authors

Gaurav Barot is an experienced software architect and PMP-certified project manager with more than 12 years of experience. He has a unique combination of experience in enterprise resource planning, sales, education, and technology. He has served as an enterprise architect and project leader in projects in various domains, including healthcare, risk, insurance, media, and so on for customers in the UK, USA, Singapore, and India.

Gaurav holds a bachelor's degree in IT engineering from Sardar Patel University, and has completed his post graduation in IT from Deakin University Melbourne.

I would like to thank all my team members and fellow coauthors at KNOWARTH Technologies. I sincerely thank and appreciate the entire team at Packt Publishing for providing this opportunity. Thanks a lot to Akashdeep Kundu for his continuous support and patience throughout this project.

Last but not least, I would like to thank my parents and my two younger sisters, Kinjal and Yogini, for their love and encouragement. A special thanks to my wife, Kruti, and my lovely daughter, Twisha. Both of them have been tolerant and understanding during all the time I spent on my computer while working on this book.

Chintan Mehta is a cofounder of KNOWARTH Technologies (`www.knowarth.com`) and heads the cloud/RIMS department. He has rich, progressive experience in the AWS cloud, DevOps, RIMS, and server administration on open source technologies.

Chintan's vital roles during his career in infrastructure and operations have included requirement analysis, architecture design, security design, high availability and disaster recovery planning, automated monitoring, automated deployment, build processes to help customers, performance tuning, infrastructure setup and deployment, and application setup and deployment. He has done all these along with setting up various offices in different locations with fantastic sole ownership to achieve operation readiness for the organizations he has been associated with.

He headed and managed cloud service practices with his previous employer, and received multiple awards in recognition of the very valuable contribution he made to the business. He was involved in creating solutions and consulting for building SaaS, IaaS, and PaaS services on the cloud. Chintan also led the ISO 27001:2005 implementation team as a joint management representative, and has reviewed *Liferay Portal Performance Best Practices*, *Packt Publishing*. He completed his diploma in computer hardware and network certification from a reputed institute in India.

I have relied on many people, both directly and indirectly, in writing this book.

First of all, I would like to thank my coauthors and the great team at Packt Publishing for this effort. I would especially like to thank my wonderful wife, Mittal, and my sweet son, Devam, for putting up with the long days, nights, and weekends when I was camping in front of my laptop.

Many people have inspired me, made contributions to this book, and provided comments, edits, insights, and ideas—specifically Parth Ghiya, Chintan Gajjar, and Nilesh Jain. Special thanks go to Samir Bhatt, who got me started on writing this book; there were several things that could have interfered with my book. I also want to thank all the reviewers of this book.

Last but not least, I want to thank my parents and all my friends, family, and colleagues for supporting me throughout the writing of this book.

Amij Patel is a cofounder of KNOWARTH Technologies (`www.knowarth.com`) and leads mobile, UI/UX, and e-commerce vertical. He is an out-of-the-box thinker with a proven track record of designing and delivering the best design solutions for enterprise applications and products.

He has a lot of experience in the Web, portals, e-commerce, rich Internet applications, user interfaces, big data, and open source technologies. His passion is to make applications and products interactive and user friendly using the latest technologies. Amij has a unique ability—he can deliver or execute on any layer and technology from the stack.

Throughout his career, he has been honored with awards for making valuable contributions to businesses and delivering excellence through different roles, such as a practice leader, architect, and team leader. He is a cofounder of various community groups, such as Ahmedabad JS and the Liferay UI developers' group. These are focused on sharing knowledge of UI technologies and upcoming trends with the broader community. Amij is respected as motivational, the one who leads by example, a change agent, and a proponent of empowerment and accountability.

I would like to thank my coauthors, reviewers, and the Packt Publishing team for helping me at all the stages of this project. I would also like to thank my parents and family, especially my wife, Nehal Patel. She supported me when we were expecting a baby and I was writing this book. Then, I would like to thank my children, Urv and Urja, for adjusting to my late nights and weekends when I was busy writing this book.

My special thanks to Parth Ghiya, who helped me a lot with brainstorming sessions, practical examples, and ideas to make the chapters more interactive. Also, I would like to thank Chintan Gajjar and Nilesh Jain for helping with the graphics and images as and when required.

About the Reviewers

Skanda Bhargav is an engineering graduate from Visvesvaraya Technological University (VTU) in Belgaum, Karnataka, India. He did his major in computer science engineering. He is a Cloudera-certified developer in Apache Hadoop. His interests are big data and Hadoop. He has been a reviewer of the following books and videos, all by Packt Publishing:

- *Building Hadoop Clusters [Video]*
- *Hadoop Cluster Deployment*
- *Instant MapReduce Patterns – Hadoop Essentials How-to*
- *Cloudera Administration Handbook*
- *Hadoop MapReduce v2 Cookbook – Second Edition*

I would like to thank my family for their immense support and faith in me throughout my learning stage. My friends have brought the confidence in me to a level that makes me bring the best out of myself. I am happy that God has blessed me with such wonderful people around me, without whom, my success as it is today would not have been possible.

Venkat Krishnan is a programming expert who has spent 18 years in IT training and consulting in the areas of Java, Enterprise J2EE, Spring, Hibernate, web services, and Android. He spent 5 years in Hadoop training, consulting, and assisting organizations such as JPMC and the Microsoft India Development Center in the inception, incubation, and growth of big data innovations.

Venkat has an MBA degree. He has mentored more than 5,000 participants in the area of big data on the Hadoop platform in Linux and Windows, and more than 10,000 participants in Java, J2EE, Spring, and Android. He has also mentored participants in Amdocs, Fidelity, Wells Fargo, TCS, HCL, Accenture, and other organizations in Hadoop with its ecosystem components, such as Hike, HBase, Pig, and Sqoop. Venkat has provided training for associates across the globe, in countries such as Japan, Australia, Europe, South Africa, USA, Mexico, Dubai, Oman, and others.

I offer my humble thanks to my parents and all my teachers who have helped me learn and be open to new upcoming technologies. Also, I want to thank my wife, Raji, and my children, Disha and Dhruv, for supporting me in all of my new endeavors. In the ever-evolving world of technology, I consider myself lucky to be able to share my knowledge with the brilliant kids of tomorrow. Thanks to the Packt Publishing team for giving me the opportunity to review the work of some eminent people.

Stefan Matheis is the CTO of Kellerkinder GmbH, based near Mannheim, Germany. Kellerkinder offers technical support for various projects based on PHP, as well as consulting and workshops for Apache Solr.

His passion includes working in API development, natural language processing, graph databases, and infrastructure management. He has been an Apache Lucene/Solr committer since 2012, as well as a member of the project management committee.

Stefan is also a speaker at various conferences, the first of which was the Lucene/Solr Revolution in Dublin, Ireland. The admin UI that is shipped with all releases since Solr 4.0 is what he is known for in the Solr community and among its users. He has reviewed a few books, one of which is *Solr Cookbook – Third Edition*, *Packt Publishing*.

He can be contacted at `stefan@kellerkinder.de`.

Manjeet Singh Sawhney currently works for a large IT consultancy in London, UK, as a Principal Consultant - Information / Data Architect. Previously, he worked for global organizations in various roles, including Java development, technical solutions consulting, and data management consulting. During his postgraduate studies, he also worked as a Student Tutor for one of the top 100 universities in the world, where he was teaching Java to undergraduate students and was involved in marking exams and evaluating project assignments. Manjeet acquired his professional experience by working on several mission-critical projects, serving clients in the financial services, telecommunications, manufacturing, retail, and public sectors.

I am very thankful to my parents; my wife, Jaspal; my son, Kohinoor; and my daughter, Prabhnoor, for their encouragement and patience, as reviewing this book took some of the evenings and weekends that I would have spent with my family.

Table of Contents

Preface

Hadoop is an open source technology that enables the distributed processing of large datasets across clusters of community servers. It is designed to scale up from a single server to thousands of machines, with a very high degree of fault tolerance. Rather than relying on high-end hardware, the resiliency of these clusters comes from their ability to detect and handle failures at the application level. Hadoop enables computing solutions that are scalable, cost effective, fault tolerant, and flexible.

Hadoop has already been accepted by many organizations for enterprise implementations. There is a good amount of information available that focuses on Hadoop setup and development. However, structured information for a good strategy of backing up and recovering Hadoop is not available. This book covers backup and recovery solutions for Hadoop, which is very much needed information.

The information provided in this book is not easily available in a structured way. It also targets the fundamentals of backup and recovery and common issues you can come across in Hadoop.

There are three major items that this book covers:

- Hadoop backup and recovery fundamentals
- Hadoop backup needs and the implementation strategy
- Troubleshooting problems

What this book covers

Chapter 1, Knowing Hadoop and Clustering Basics, explains the basics of Hadoop administration, HDFS, and a cluster setup, which are the prerequisites for backup and recovery. It also explains the HDFS design and daemons, and Hadoop cluster best practices.

Chapter 2, Understanding Hadoop Backup and Recovery Needs, explains the importance of backup and recovery in Hadoop. This chapter also talks about the areas that should be considered during backup. It helps you understand the principles and requirements of backup and recovery.

Chapter 3, Determining Backup Strategies, discusses common failure points in Hadoop. This helps the user determine common areas that need to be protected. It also talks about the importance of the backup strategy and backing up Hive metadata.

Chapter 4, Backing Up Hadoop, introduces HBase and talks about its importance. This chapter lets you know the different ways of taking a backup. It also provides a comparison of different backup styles.

Chapter 5, Determining Recovery Strategy, talks about defining a recovery strategy for various causes of failure when Hadoop is considered for a business-critical high-scale implementation.

Chapter 6, Recovering Hadoop Data, helps you with recovery in scenarios covering fail-over, corruption, working drives, the NameNode, tables, and metadata.

Chapter 7, Monitoring, provides an overview of the importance of monitoring. This chapter introduces the concept of Hadoop matrix. It also talks about the different areas for monitoring node health in Hadoop.

Chapter 8, Troubleshooting, helps you understand troubleshooting strategies. It talks about common failure points and techniques to resolve failures.

What you need for this book

You will need access to a single-node setup, and a fully distributed and properly configured cluster. Tools such as Java JDK are an integral part of this.

When you work through the later part of this book, you will need Apache ZooKeeper and Ganglia installed to verify and configure the various steps with regard to backup, recovery, testing, and more.

Who this book is for

If you are a Hadoop administrator and are looking to get a good grounding in how to back up large-scale data and manage Hadoop clusters, then this book is for you.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "You can find the `hdfs-default.xml` file in the `hadoop/conf` directory."

A block of code is set as follows:

```
<property>
<name>dfs.block.size<name>
<value>134217728<value>
<description>Block size<description>
<property>
```

Any command-line input or output is written as follows:

```
hadoop distcp hdfs://node1:8020/src/d1 \
hdfs://node1:8020/src/d2 \
hdfs://node2.8020dest/node
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "In Cloudera Manager Server, we can use the **Add Peer** link available on the **Replication** page."

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1 Knowing Hadoop and Clustering Basics

Today, we are living in the age of data. People are generating data in different ways: they take pictures, send e-mails, upload images, write blogs, comment on someone's blog or picture, change their status on social networking sites, tweet on Twitter, update details on LinkedIn, and so on. Just a couple of decades ago, a general belief was that 1 GB of disk storage would be more than enough for a personal computer. And, nowadays, we use hard disks having capacity in terabytes. Data size has not only grown in personal space but also in professional services, where people have to deal with a humongous amount of data. Think of the data managed by players such as Google, Facebook, New York Stock Exchange, Amazon, and many others. The list is endless. This situation challenged the way we were storing, managing, and processing data traditionally. New technologies and platforms have emerged, which provide us with solutions to these challenges. But again, do you think a solution providing smart data storage and retrieval will be enough? Would you like to be in a situation where you log in to your Facebook or Instagram account and find that all your data was lost due to a hardware failure? Absolutely not!

It is important to not only store data but also process the data in order to generate information that is necessary. And it is equally important to back up and recover the data in the case of any type of failure. We need to have a sound plan and policy defined and implemented for data backup and recovery in the case of any failure.

Apache Hadoop is a platform that provides practical, cost-effective, and scalable infrastructure to store, manage, and process data. This book focuses on understanding backup and recovery needs and defining and determining backup and recovery strategies, as well as implementing them in order to ensure data and metadata backup and recovery after the occurrence of a failure in Hadoop. To begin with, in this chapter, we will discuss basic but important concepts of Hadoop, HDFS, daemons, and clustering. So, fasten your seatbelts and get ready to fly with me in the Hadoop territory!

Understanding the need for Hadoop

As mentioned at the beginning of the chapter, the enormous data growth first became a challenge for big players such as Yahoo!, Google, Amazon, and Facebook. They had to not only store this data but also process it effectively. When their existing tools and technologies were not enough to process this huge set of data, Google introduced the world to MapReduce in 2004. Prior to publishing MapReduce in 2004, Google also published a paper in 2003 on **Google File System** (**GFS**), describing pragmatic, scalable, and distributed file systems optimized to store huge datasets. It was built to support large-scale, data-intensive, and distributed processing applications.

Both these systems were able to solve a major problem that many companies were facing at that time: to manage large datasets in an effective manner.

Doug Cutting, who developed Apache Lucene, grabbed the opportunity and led an initiative to develop an open source version of MapReduce called Hadoop. Right after this, Yahoo! and many others supported the initiative and efforts. Very soon, Hadoop was accepted as an open source framework for processing huge datasets in a distributed environment. The good part is that it was running on a cluster of commodity computers and hardware. By 2008, big organizations such as Yahoo!, Facebook, New York Times, and many others started using Hadoop.

There is a common misconception that Hadoop is an acronym of a long name. However, that's not the case; it's a made-up name and not an acronym. Let's see what Doug Cutting has to say about the name:

"The name my kid gave a stuffed yellow elephant. Short, relatively easy to spell and pronounce, meaningless, and not used elsewhere—these are my naming criteria. Kids are good at generating such names. Google is a kid's term."

By now, you must be thinking that knowing the history of Hadoop is fine but what do you do with it? What is the actual use of Hadoop? What are the scenarios where Hadoop will be beneficial compared to other technologies? Well, if you are thinking of all these questions, then your neurons are fired up and you've started getting into the Hadoop space. (By the way, if you are not thinking about these questions, then you might already know what Hadoop does and how it performs those actions or you need a cup of strong coffee!)

For the next few minutes, assume that you are the owner of an online store. You have thousands of products listed on your store and they range from apparels to electronic items and home appliances to sports products. What items will you display on the home page? Also, once a user has an account on your store, what kind of products will you display to the user? Well, here are the choices:

- Display the same products to all the users on the home page
- Display different products to users accessing the home page based on the country and region they access the page from. (Your store is very popular and has visitors from around the globe, remember!)
- Display the same products to the users after they log in. (Well, you must be thinking "do I really want to do this?" If so, it's time to start your own e-commerce store.)
- Display products to users based on their purchase history, the products they normally search for, and the products that other users bought, who had a similar purchase history as this user.

Here, the first and third options will be very easy to implement but they won't add much value for the users, will they? On the other hand, options 2 and 4 will give users the feeling that someone is taking care of their needs and there will be higher chances of the products being sold. But the solution needs algorithms to be written in such a way that they analyze the data and give quality results. Having relevant data displayed on the store will result in happy and satisfied customers, revisiting members, and most importantly, more money!

You must be thinking: what relates Hadoop to the online store?

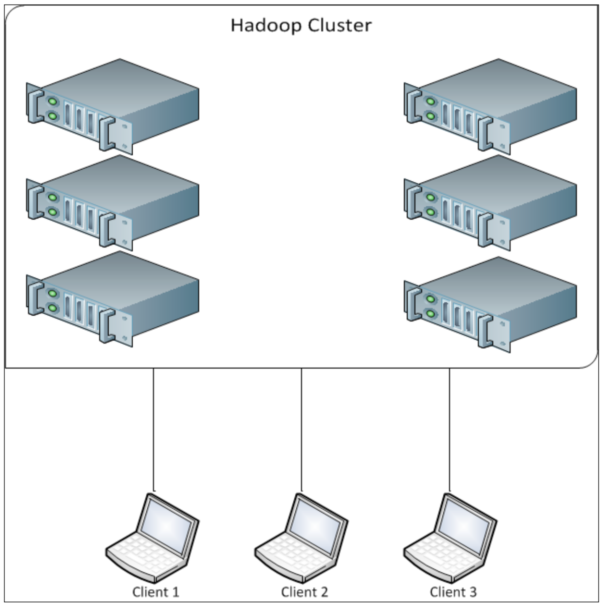
Apache Hadoop provides tools that solve a big problem mentioned earlier: managing large datasets and processing them effectively. Hadoop provides linear scalability by implementing a cluster of low-cost commodity hardware. The main principle behind this design is to bring computing logic to data rather than bringing huge amounts of data to computing logic, which can be time-consuming and inefficient in the case of huge datasets. Let's understand this with an example.

Hadoop uses a cluster for data storage and computation purposes. In Hadoop, parallel computation is performed by MapReduce. A developer develops or provides the computation logic for data processing. Hadoop runs the computation logic on the machines or nodes where the data exists rather than sending data to the machines where the computation logic is built-in. This results in performance improvement, as you are no longer sending huge amounts of data across the network but sending the processing/computation logic, which is less in size compared to the nodes where the data is stored.

To put all of this in a nutshell, Hadoop is an open source framework used to run and write distributed applications for processing huge amounts of data. However, there are a few fundamental differences between Hadoop and traditional distributed frameworks, as follows:

- **Low cost per byte**: Hadoop's HDFS uses low-cost hardware storage and shares the cost of the network and computers it runs on with MapReduce. HDFS is open source software, which again reduces the cost of ownership. This cost advantage lets organizations store more data per dollar than traditional distributed systems.
- **High data reliability**: As Hadoop's distributed system is running on commodity hardware, there can be very high chances of device failure and data loss. Hadoop has been architected keeping in mind this common but critical issue. Hadoop has been tested and has proven itself in multiple use cases and cluster sizes against such failures.
- **High throughput**: Hadoop provides large throughput access to application data and is suitable for applications with large datasets.
- **Scalable**: We can add more nodes to a Hadoop cluster to increase linear scalability very easily by adding more nodes in the Hadoop cluster.

The following image represents a Hadoop cluster that can be accessed by multiple clients. As demonstrated, a Hadoop cluster is a set of commodity machines. Data storage and data computing occur in this set of machines by HDFS and MapReduce, respectively. Different clients can submit a job for processing to this cluster, and the machines in the cluster jointly execute the job.



There are many projects developed by the Apache Software Foundation, which make the job of HDFS and MapReduce easier. Let's touch base on some of those projects.

Apache Hive

Hive is a data warehouse infrastructure built on Hadoop that uses its storage and execution model. It was initially developed by Facebook. It provides a query language that is similar to SQL and is known as the **Hive query language** (**HQL**). Using this language, we can analyze large datasets stored in file systems, such as HDFS. Hive also provides an indexing feature. It is designed to support ad hoc queries and easy data summarization as well as to analyze large volumes of data. You can get the full definition of the language at `https://cwiki.apache.org/confluence/display/Hive/LanguageManual`.

Apache Pig

The native language of Hadoop is Java. You can also write MapReduce in a scripting language such as Python. As you might already know, the MapReduce model allows you to translate your program execution in a series of map and reduce stages and to configure it properly. Pig is an abstraction layer built on top of Hadoop, which simplifies the authoring of MapReduce jobs. Instead of writing code in Java, developers write data processing jobs in scripting languages. Pig can execute a series of MapReduce jobs using Pig Scripts.

Apache HBase

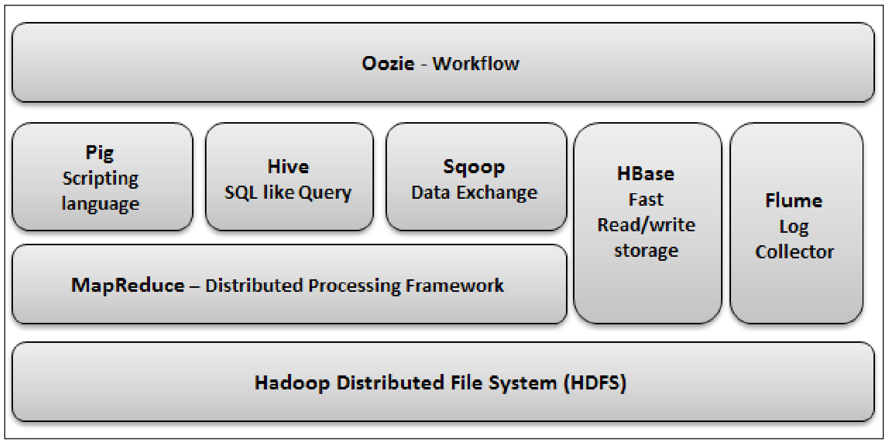
Apache HBase is an open source implementation of Google's Big Table. It was developed as part of the Hadoop project and runs on top of HDFS. It provides a fault-tolerant way to store large quantities of sparse data. HBase is a nonrelational, column-oriented, and multidimensional database, which uses HDFS for storage. It represents a flexible data model with scale-out properties and a simple API. Many organizations use HBase, some of which are Facebook, Twitter, Mozilla, Meetup, Yahoo!.

Apache HCatalog

HCatalog is a metadata and table storage management service for HDFS. HCatalog depends on the Hive metastore. It provides a shared schema and data types for Hadoop tools. It solves the problem of tools not agreeing on the schema, data types, and how the data is stored. It enables interoperability across HBase, Hive, Pig, and MapReduce by providing one consistent data model and a shared schema for these tools. HCatalog achieves this by providing table abstraction. HCatalog's goal is to simplify the user's interaction with HDFS data and enable data sharing between tools and execution platforms.

There are other Hadoop projects such as Sqoop, Flume, Oozie, Whirr, and ZooKeeper, which are part of the Hadoop ecosystem.

The following image gives an overview of the Hadoop ecosystem:


Oozie - Workflow
Pig Scripting language
Hive SQL like Query
Sqoop Data Exchange
HBase Fast Read/write storage
Flume Log Collector
MapReduce – Distributed Processing Framework
Hadoop Distributed File System (HDFS)

Understanding HDFS design

So far, in this chapter, we have referred to HDFS many times. It's now time to take a closer look at HDFS. This section talks about HDFS basics, design, and its daemons, such as NameNode and DataNode.

The **Hadoop Distributed File System** (**HDFS**) has been built to support high throughput, streaming reads and writes of huge files. There can be an argument from traditional SAN or NAS lovers that the same functions can be performed by SAN or NAS as well. They also offer centralized, low-latency access to large file systems. So, what's the purpose of having HDFS?

Let's talk about Facebook (that's what you like, isn't it?). There are hundreds of thousands of users using their laptops, desktops, tablets, or smart phones, trying to pull humongous amount of data together from the centralized Facebook server. Do you think traditional SAN or NAS will work effectively in this scenario? Well, your answer is correct. Absolutely not!

HDFS has been designed to handle these kinds of situations and requirements. The following are the goals of HDFS:

- To store multimillion files, where each file cannot be more than 1 GB and the overall file storage crosses petabytes.
- To create clusters using commodity hardware rather than using RAID to achieve the large storage mentioned in the previous point. Here, high availability and throughput is achieved through application-level replication.
- To provide robustness by gracefully handling hardware failure.

HDFS has many similarities to a traditional file system, such as SAN or GFS. The following are some of them:

- Files are stored in blocks. Metadata is available, which keeps track of filenames to block mapping.
- It also supports the directory tree structure, as a traditional file system.
- It works on the permission model. You can give different access rights on the file to different users.

However, along with the similarities, there are differences as well. Here are some of the differences:

- **Very low storage cost per byte**: HDFS uses commodity storage and shares the cost of the network it runs on with other systems of the Hadoop stack. Also, being open source, it reduces the total cost of ownership. Due to this cost benefit, it allows an organization to store the same amount of data at a very low cost compared to traditional NAS or SAN systems.
- **Block size for the data**: Traditional file systems generally use around 8 KB block size for the data, whereas HDFS uses larger block size of data. By default, the block size in HDFS is 64 MB, but based on the requirement, the admin can raise it to 1 GB or higher. Having a larger block size ensures that the data can be read and written in large sequential operations. It can improve performance as it minimizes drive seek operations, especially when performing large I/O streaming operations.
- **Data protection mechanisms**: Traditional file systems use specialized data storage for data protection, while HDFS replicates each block to multiple machines in the cluster. By default, it replicates the data block to three nodes. This ensures data reliability and high availability. You will see how this can be achieved and how Hadoop retrieves data in the case of failure, later in this book.

HDFS runs on commodity hardware. The cluster consists of different nodes. Each node stores a subset of data, and the data is aggregated to make a complete file system. The data is also replicated on three different nodes to provide fault tolerance. Since it works on the principle of moving computation to large data for processing, it provides high throughput and fits best for the applications where huge datasets are involved. HDFS has been built keeping the following goals in mind:

- Handling hardware failure elegantly.
- Streaming data access to the dataset.
- Handling large datasets.
- A simple coherency model with a principle of the write-once-read-many access model.
- Move computation to data rather than moving data to computation for processing. This is very helpful in the case of large datasets.

Getting familiar with HDFS daemons

We have understood the design of HDFS. Now, let's talk about the daemons that play an important role in the overall HDFS architecture. Let's understand the function of the daemons with the following small conversation; let the members involved in the conversation introduce themselves:

User	I'm the one who needs data to be read and written. But, I'm not a daemon.
Client	Hi, there! People sit in front of me and ask me to read and write data. I'm also not an HDFS daemon.
NameNode	There is only one of my kinds here. I am the coordinator here. (Really? What about the secondary NameNode? Don't worry, we will cover this at the end of this section.)
DataNode	We are the ones who actually store your data. We live in a group. Sometimes, we are in an army of thousands!

Now, let's see different scenarios in order to understand the function of each of these.

Scenario 1 – writing data to the HDFS cluster

Here, a user is trying to write data into the Hadoop cluster. See how the client, NameNode, and DataNode are involved in writing the data:

User	"Hello Mr. Client, I want to write 2 GB of data. Can you please do it for me?"
Client	"Sure, Mr. User. I'll do it for you."
User	"And yes, please divide the data into 512 MB blocks and replicate it at three different locations, as you usually do."
Client	"With pleasure, sir."
Client (thinking)	Hmm, now I have to divide this big file into smaller blocks, as asked by Mr. User. But to write it to three different locations, I have to ask Mr. NameNode. He is such a knowledgeable person.
Client	"Hey, Mr. NameNode. I need your help. I have this big file, which I have divided into smaller blocks. But now I need to write it to three different locations. Can you please provide me the details of these locations?"
NameNode	"Sure, buddy. That's what I do. First of all, let me find three DataNodes for you."

NameNode (after finding DataNodes)	"Here you go, my friend! Take the addresses of the DataNodes where you can store information. I have also sorted them based on the increase in distance from you."
Client	(Such a gentleman.) "Thanks a lot, Mr. NameNode. I appreciate all the efforts you've put in."
Client	"Hello Mr. DataNode1. Can you please write this block of data on your disk? And yes, please take the list of the nodes as well. Please ask them to replicate the data."
DataNode1	"With pleasure, Mr. Client. Let me start storing the data and while I'm doing that, I will also forward the data to the next DataNode."
DataNode2	"Let me follow you, dear friend. I've started storing the data and forwarded it to the next node."
DataNode3	"I'm the last guy who's storing the data. Now, the replication of data is completed."
All DataNodes	"Mr. NameNode, we have completed the job. Data has been written and replicated successfully."
NameNode	"Mr. Client, your block has been successfully stored and replicated. Please repeat the same with the rest of the blocks too."
Client (after repeating it for all the blocks)	"All the blocks are written, Mr. NameNode. Please close the file. I truly appreciate your help."
NameNode	"OK. The case is closed from my end. Now, let me store all the metadata on my hard disk."

As you have seen from the conversation in the preceding table, the following are the observations:

- The client is responsible for dividing the files into smaller chunks or blocks
- NameNode keeps the address of each DataNode and coordinates the data writing and replication process. It also stores the metadata of the file. There is only one NameNode per HDFS cluster.
- The DataNode stores the blocks of the data and takes care of the replication.

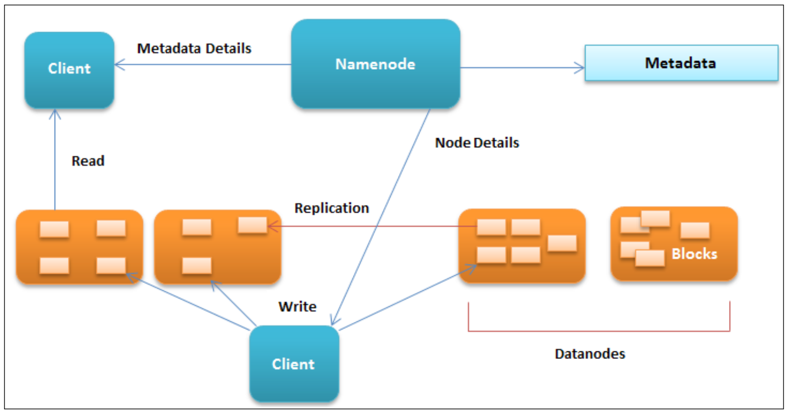
Since we started the discussion with the writing of data, it cannot be completed without discussing the reading process. We have the same members involved in this conversation too.

Scenario 2 – reading data from the HDFS cluster

You have seen how data is written. Now, let's talk about a scenario when the user wants to read data. It will be good to understand the role of the client, NameNode, and DataNode in this scenario.

User	"Hello, Mr. Client. Do you remember me? I asked you to store some data earlier."
Client	"Certainly, I do remember you, Mr. User."
User	"Good to know that. Now, I need the same data back. Can you please read this file for me?"
Client	"Certainly, I'll do it for you."
Client	"Hi, Mr. NameNode. Can you please provide the details of this file?"
NameNode	"Sure, Mr. Client. I have stored the metadata for this file. Let me retrieve the data for you. Here, you go. You will need these two things to get the file back: • A list of all the blocks for this file • A list of all the DataNodes for each block Use this information to download the blocks."
Client	"Thanks, Mr. NameNode. You're always helpful."
Client (to the nearest DataNode)	"Please give me block 1." (The process gets repeated until all the blocks of the files have been retrieved.)
Client (after retrieving all the blocks for the file)	"Hi, Mr. User. Here's the file you needed."
User	"Thanks, Mr. Client."

The entire read and write process is displayed in the following image:



Now, you may wonder what happens when any of the DataNodes is down or the data is corrupted. This is handled by HDFS. It detects various types of faults and handles them elegantly. We will discuss fault tolerance in the later chapters of this book.

Now, you have understood the concepts of NameNode and DataNode. Let's do a quick recap of the same.

HDFS, having the master/slave architecture, consists of a single NameNode, which is like a master server managing the file system namespace and regulates access to the files by the client. The DataNode is like a slave and there is one DataNode per node in the cluster. It is responsible for the storage of data in its local storage. It is also responsible for serving read and write requests by the client. A DataNode can also perform block creation, deletion, and replication based on the command received from the NameNode. The NameNode is responsible for file operations such as opening, closing, and renaming files.

You may have also heard the name of another daemon in HDFS called the secondary NameNode. From the name, it seems that it will be a backup NameNode, which will take charge in the case of a NameNode failure. But that's not the case. The secondary NameNode is responsible for internal housekeeping, which will ensure that the NameNode will not run out of memory and the startup time of the NameNode is faster. It periodically reads the file system changes log and applies them to the `fsimage` file, thus bringing the NameNode up to date while it reads the file during startup.

Many modern setups use the HDFS high availability feature in which the secondary NameNode is not used. It uses an active and a standby NameNode.

Understanding the basics of Hadoop cluster

Until now, in this chapter, we have discussed the different individual components of Hadoop. In this section, we will discuss how those components come together to build the Hadoop cluster.

There are two main components of Hadoop:

- **HDFS**: This is the storage component of Hadoop, which is optimized for high throughput. It works best while reading and writing large files. We have also discussed the daemons of HDFS—NameNode, DataNode, and the secondary NameNode.
- **MapReduce**: This is a batch-based, distributed computing framework, which allows you to do parallel processing over large amounts of raw data. It allows you to process the data in the storage asset itself, which reduces the distance over which the data needs to be transmitted for processing. MapReduce is a two-step process, as follows:
    - **The map stage**: Here, the master node takes the input and divides the input into smaller chunks (subproblems), which are similar in nature. These chunks (subproblems) are distributed to worker nodes. Each worker node can also divide the subproblem and distribute it to other nodes by following the same process. This creates a tree structure. At the time of processing, each worker processes the subproblem and passes the result back to the parent node. The master node receives a final input collected from all the nodes.

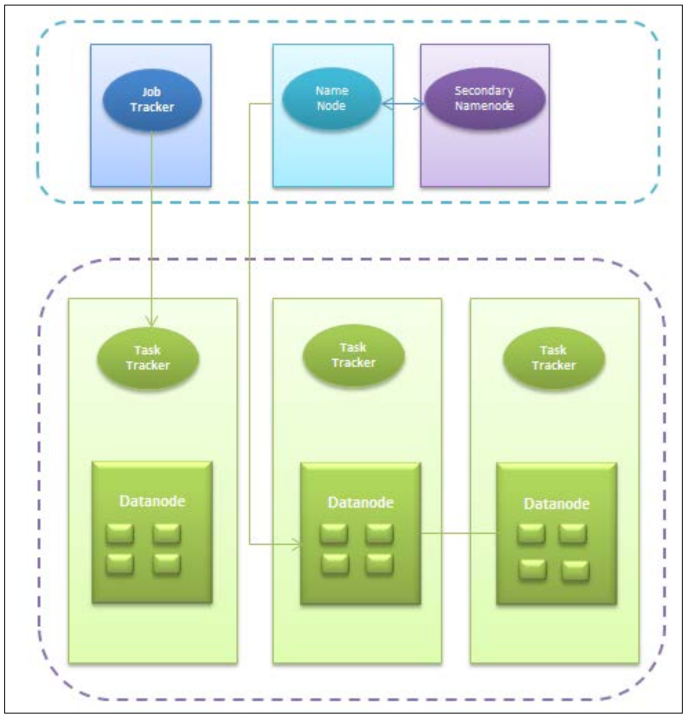
    - **The reduce stage**: The master node collects all the answers from the worker nodes to all the subproblems and combines them to form a meaningful output. This is the output to the problem, which originally needed to be solved.

Along with these components, a fully configured Hadoop cluster runs a set of daemons on different servers in the network. All these daemons perform specific functions. As discussed in the previous section, some of them exist only on one server, while some can be present on multiple servers. We have discussed NameNode, DataNode, and the secondary NameNode. On top of these three daemons, we have other two daemons that are required in the Hadoop cluster, as follows:

- **JobTracker**: This works as a connection between your application and Hadoop. The client machine submits the job to the JobTracker. The JobTracker gets information about the DataNodes that contain the blocks of the file to be processed from the NameNode. After getting the information about the DataNodes, the JobTracker provides the code for map computation to TaskTrackers, which are running on the same server as DataNodes. All the TaskTrackers run the code and store the data on their local server. After completing the map process, the JobTracker starts the reduce tasks (which run on the same node), where it gets all the intermediate results from those DataNodes. After getting all the data, HDFS does the final computation, where it combines the intermediate results and provides the final result to the client. The final results are written to HDFS.
- **TaskTracker**: As discussed earlier, the JobTracker manages the execution of individual tasks on each slave node. In the Hadoop cluster, each slave node runs a DataNode and a TaskTracker daemon. The TaskTracker communicates and receives instructions from the JobTracker. When the TaskTracker receives instructions from the JobTracker for computation, it executes the map process, as discussed in the previous point. The TaskTracker runs both map and reduce jobs. During this process, the TaskTracker also monitors the task's progress and provides heartbeats and the task status back to the JobTracker. If the task fails or the node goes down, the JobTracker sends the job to another TaskTracker by consulting the NameNode.

> Here, JobTracker and TaskTracker are used in the Hadoop cluster. However, in the case of YARN, different daemons are used, such as ResourceManager and NodeManager.

The following image displays all the components of the Hadoop cluster:



The preceding image is a very simple representation of the components in the Hadoop cluster. These are only those components that we have discussed so far. A Hadoop cluster can have other components, such as HiveServer, HBase Master, Zookeeper, Oozie, and so on.

Hadoop clusters are used to increase the speed of data analysis of your application. If the data grows in such a way that it supersedes the processing power of the cluster, we can add cluster nodes to increase throughput. This way, the Hadoop cluster provides high scalability. Also, each piece of data is copied onto other nodes of the clusters. So, in the case of failure of any node, the data is still available.

While planning the capacity of a Hadoop cluster, one has to carefully go through the following options. An appropriate choice of each option is very important for a good cluster design:

- Pick the right distribution version of Hadoop
- Designing Hadoop for high availability of the network and NameNode
- Select appropriate hardware for your master—NameNode, the secondary NameNode, and JobTracker
- Select appropriate hardware for your worker—for data storage and computation
- Do proper cluster sizing, considering the number of requests to HBase, number of MapReduce jobs in parallel, size of incoming data per day, and so on
- Operating system selection

We have to determine the required components in the Hadoop cluster based on our requirements. Once the components are determined, we have to determine base infrastructure requirements. Based on the infrastructure requirements, we have to perform the hardware sizing exercise. After the sizing, we need to ensure load balancing and proper separation of racks/locations. Last but not least, we have to choose the right monitoring solution to look at what our cluster is actually doing.

We will discuss these points in later chapters of this book.

Summary

In this chapter, we discussed the Hadoop basics, the practical use of Hadoop, and the different projects of Hadoop. We also discussed the two major components of Hadoop: HDFS and MapReduce. We had a look at the daemons involved in the Hadoop cluster—NameNode, the secondary NameNode, DataNode, JobTracker, and TaskTracker—along with their functions. We also discussed the steps that one has to follow when designing the Hadoop cluster.

In the next chapter, we will talk about the essentials of backup and recovery of the Hadoop implementation.

2
Understanding Hadoop Backup and Recovery Needs

In *Chapter 1*, *Knowing Hadoop and Clustering Basics*, we discussed the basics of Hadoop administration, HDFS, and cluster setups, which are prerequisites for backup and recovery. In this chapter, we will discuss backup and recovery needs. This chapter assumes that you have gone through the previous chapter and are familiar with the concepts of NameNode, DataNode, and how communication occurs between them. Furthermore, this chapter assumes that you have Hadoop set up either in your single node / multinode cluster and you have familiarity of running commands of Hadoop.

In the present age of information explosion, data is the backbone of business organizations of all sizes. We need a complete data backup and recovery system and a strategy to ensure that critical data is available and accessible when the organizations need it. Data must be protected against loss, damage, theft, and unauthorized changes. If disaster strikes, data recovery must be swift and smooth so that business does not get impacted. Every organization has its own data backup and recovery needs, and priorities based on the applications and systems they are using. Today's IT organizations face the challenge of implementing reliable backup and recovery solutions in the most efficient, cost-effective manner. To meet this challenge, we need to carefully define our business requirements and recovery objectives before deciding on the right backup and recovery strategies or technologies to deploy.

Before jumping onto the implementation approach, we first need to know about the backup and recovery strategies and how to efficiently plan them.

Understanding the backup and recovery philosophies

Backup and recovery is becoming more challenging and complicated, especially with the explosion of data growth and increasing need for data security today. Imagine big players such as Facebook, Yahoo! (the first to implement Hadoop), eBay, and more; how challenging it will be for them to handle unprecedented volumes and velocities of unstructured data, something which traditional relational databases can't handle and deliver. To emphasize the importance of backup, let's take a look at a study conducted in 2009. This was the time when Hadoop was evolving and a handful of bugs still existed in Hadoop. Yahoo! had about 20,000 nodes running Apache Hadoop in 10 different clusters.

HDFS lost only 650 blocks, out of 329 million total blocks. Now hold on a second. These blocks were lost due to the bugs found in the Hadoop package. So, imagine what the scenario would be now. I am sure you will bet on losing hardly a block. Being a backup manager, your utmost target is to think, make, strategize, and execute a foolproof backup strategy capable of retrieving data after any disaster. Solely speaking, the plan of the strategy is to protect the files in HDFS against disastrous situations and revamp the files back to their normal state, just like James Bond resurrects after so many blows and probably death-like situations.

Coming back to the backup manager's role, the following are the activities of this role:

- Testing out various case scenarios to forestall any threats, if any, in the future
- Building a stable recovery point and setup for backup and recovery situations
- Preplanning and daily organization of the backup schedule
- Constantly supervising the backup and recovery process and avoiding threats, if any
- Repairing and constructing solutions for backup processes
- The ability to reheal, that is, recover from data threats, if they arise (the resurrection power)
- Data protection is one of the activities and it includes the tasks of maintaining data replicas for long-term storage
- Resettling data from one destination to another

Basically, backup and recovery strategies should cover all the areas mentioned here. For any system data, application, or configuration, transaction logs are mission critical, though it depends on the datasets, configurations, and applications that are used to design the backup and recovery strategies. You saw Hadoop basics in *Chapter 1, Knowing Hadoop and Clustering Basics*, so let's see how we should plan backup and recovery strategies for Hadoop.

Hadoop is all about big data processing. After gathering some exabytes for data processing, the following are the obvious questions that we may come up with:

- What's the best way to back up data?
- Do we really need to take a backup of these large chunks of data?
- Where will we find more storage space if the current storage space runs out?
- Will we have to maintain distributed systems?
- What if our backup storage unit gets corrupted?

The answer to the preceding questions depends on the situation you may be facing; let's see a few situations.

One of the situations is where you may be dealing with a plethora of data. Hadoop is used for fact-finding semantics and data is in abundance. Here, the span of data is short; it is short lived and important sources of the data are already backed up. Such is the scenario wherein the policy of not backing up data at all is feasible, as there are already three copies (replicas) in our DataNodes (HDFS). Moreover, since Hadoop is still vulnerable to human error, a backup of configuration files and NameNode metadata (`dfs.name.dir`) should be created.

You may find yourself facing a situation where the data center on which Hadoop runs crashes and the data is not available as of now; this results in a failure to connect with mission-critical data. A possible solution here is to back up Hadoop, like any other cluster (the Hadoop command is `Hadoop`).

Replication of data using DistCp

To replicate data, the `distcp` command writes data to two different clusters. Let's look at the `distcp` command with a few examples or options.

DistCp is a handy tool used for large inter/intra cluster copying. It basically expands a list of files to input in order to map tasks, each of which will copy files that are specified in the source list.

Let's understand how to use `distcp` with some of the basic examples. The most common use case of `distcp` is intercluster copying. Let's see an example:

```
bash$ hadoop distcp2 hdfs://ka-16:8020/parth/ghiya \hdfs://ka-001:8020/
knowarth/parth
```

This command will expand the namespace under `/parth/ghiya` on the `ka-16` NameNode into the temporary file, get its content, divide them among a set of map tasks, and start copying the process on each TaskTracker from `ka-16` to `ka-001`.

The command used for copying can be generalized as follows:

```
hadoop distcp2 hftp://namenode-location:50070/basePath hdfs://namenode-
location
```

[Here, `hftp://namenode-location:50070/basePath` is the source and `hdfs://namenode-location` is the destination.]

In the preceding command, `namenode-location` refers to the hostname and `50070` is the NameNode's HTTP server post.

Updating and overwriting using DistCp

The `-update` option is used when we want to copy files from the source that don't exist on the target or have some different contents, which we do not want to erase. The `-overwrite` option overwrites the target files even if they exist at the source.

The files can be invoked by simply adding `-update` and `-overwrite`.

[In the example, we used `distcp2`, which is an advanced version of DistCp. The process will go smoothly even if we use the `distcp` command.]

Now, let's look at two versions of DistCp, the legacy DistCp or just DistCp and the new DistCp or the DistCp2:

- During the intercluster copy process, files that were skipped during the copy process have all their file attributes (permissions, owner group information, and so on) unchanged when we copy using legacy DistCp or just DistCp. This, however, is not the case in new DistCp. These values are now updated even if a file is skipped.
- Empty root directories among the source inputs were not created in the target folder in legacy DistCp, which is not the case anymore in the new DistCp.

There is a common misconception that Hadoop protects data loss; therefore, we don't need to back up the data in the Hadoop cluster. Since Hadoop replicates data three times by default, this sounds like a safe statement; however, it is not 100 percent safe. While Hadoop protects from hardware failure on the DataNodes—meaning that if one entire node goes down, you will not lose any data—there are other ways in which data loss may occur. Data loss may occur due to various reasons, such as Hadoop being highly susceptible to human errors, corrupted data writes, accidental deletions, rack failures, and many such instances. Any of these reasons are likely to cause data loss.

Consider an example where a corrupt application can destroy all data replications. During the process, it will attempt to compute each replication and on not finding a possible match, it will delete the replica. User deletions are another example of how data can be lost, as Hadoop's trash mechanism is not enabled by default.

Also, one of the most complicated and expensive-to-implement aspects of protecting data in Hadoop is the disaster recovery plan. There are many different approaches to this, and determining which approach is right requires a balance between cost, complexity, and recovery time.

A real-life scenario can be Facebook. The data that Facebook holds increases exponentially from 15 TB to 30 PB, that is, 3,000 times the Library of Congress. With increasing data, the problem faced was physical movement of the machines to the new data center, which required man power. Plus, it also impacted services for a period of time. Data availability in a short period of time is a requirement for any service; that's when Facebook started exploring Hadoop. To conquer the problem while dealing with such large repositories of data is yet another headache.

The reason why Hadoop was invented was to keep the data bound to neighborhoods on commodity servers and reasonable local storage, and to provide maximum availability to data within the neighborhood. So, a data plan is incomplete without data backup and recovery planning. A big data execution using Hadoop states a situation wherein the focus on the potential to recover from a crisis is mandatory.

We will have more details on backup and recovery using Hadoop in the next chapters; let's come back to the back up and recovery philosophy.

The backup philosophy

We need to determine whether Hadoop, the processes and applications that run on top of it (Pig, Hive, HDFS, and more), and specifically the data stored in HDFS are mission critical. If the data center where Hadoop is running disappeared, will the business stop?

Some of the key points that have to be taken into consideration have been explained in the sections that follow; by combining these points, we will arrive at the core of the backup philosophy.

Changes since the last backup

Considering the backup philosophy that we need to construct, the first thing we are going to look at are changes. We have a sound application running and then we add some changes. In case our system crashes and we need to go back to our last safe state, our backup strategy should have a clause of the changes that have been made. These changes can be either database changes or configuration changes.

Our clause should include the following points in order to construct a sound backup strategy:

- Changes we made since our last backup
- The count of files changed
- Ensure that our changes are tracked
- The possibility of bugs in user applications since the last change implemented, which may cause hindrance and it may be necessary to go back to the last safe state
- After applying new changes to the last backup, if the application doesn't work as expected, then high priority should be given to the activity of taking the application back to its last safe state or backup. This ensures that the user is not interrupted while using the application or product.

The rate of new data arrival

The next thing we are going to look at is how many changes we are dealing with. Is our application being updated so much that we are not able to decide what the last stable version was? Data is produced at a surpassing rate. Consider Facebook, which alone produces 250 TB of data a day. Data production occurs at an exponential rate. Soon, terms such as zettabytes will come upon a common place.

Our clause should include the following points in order to construct a sound backup:

- The rate at which new data is arriving
- The need for backing up each and every change
- The time factor involved in backup between two changes
- Policies to have a reserve backup storage

The size of the cluster

The size of a cluster is yet another important factor, wherein we will have to select cluster size such that it will allow us to optimize the environment for our purpose with exceptional results. Recalling the Yahoo! example, Yahoo! has 10 clusters all over the world, covering 20,000 nodes. Also, Yahoo! has the maximum number of nodes in its large clusters.

Our clause should include the following points in order to construct a sound backup:

- Selecting the right resource, which will allow us to optimize our environment. The selection of the right resources will vary as per need. Say, for instance, users with I/O-intensive workloads will go for more spindles per core. A Hadoop cluster contains four types of roles, that is, NameNode, JobTracker, TaskTracker, and DataNode.
- Handling the complexities of optimizing a distributed data center.

Priority of the datasets

The next thing we are going to look at are the new datasets, which are arriving. With the increase in the rate of new data arrivals, we always face a dilemma of what to backup. Are we tracking all the changes in the backup? Now, if are we backing up all the changes, will our performance be compromised?

Our clause should include the following points in order to construct a sound backup:

- Making the right backup of the dataset
- Taking backups at a rate that will not compromise performance

Selecting the datasets or parts of datasets

The next thing we are going to look at is what exactly is backed up. When we deal with large chunks of data, there's always a thought in our mind: Did we miss anything while selecting the datasets or parts of datasets that have not been backed up yet?

Our clause should include the following points in order to construct a sound backup:

- Backup of necessary configuration files
- Backup of files and application changes

The timelines of data backups

With such a huge amount of data collected daily (Facebook), the time interval between backups is yet another important factor. Do we back up our data daily? In two days? In three days? Should we backup small chunks of data daily, or should we back up larger chunks at a later period?

Our clause should include the following points in order to construct a sound backup:

- Dealing with any impacts if the time interval between two backups is large
- Monitoring a timely backup strategy and going through it

The frequency of data backups depends on various aspects. Firstly, it depends on the application and usage. If it is I/O intensive, we may need more backups, as each dataset is not worth losing. If it is not so I/O intensive, we may keep the frequency low.

We can determine the timeliness of data backups from the following points:

- The amount of data that we need to backup
- The rate at which new updates are coming
- Determining the window of possible data loss and making it as low as possible
- Critical datasets that need to be backed up
- Configuration and permission files that need to be backed up

Reducing the window of possible data loss

The next thing we are going to look at is how to minimize the window of possible data loss. If our backup frequency is great then what are the chances of data loss? What's our chance of recovering the latest files?

Our clause should include the following points in order to construct a sound backup:

- The potential to recover latest files in the case of a disaster
- Having a low data-loss probability

Backup consistency

The next thing we are going to look at is backup consistency. The probability of invalid backups should be less or even better zero. This is because if invalid backups are not tracked, then copies of invalid backups will be made further, which will again disrupt our backup process.

Our clause should include the following points in order to construct a sound backup:

- Avoid copying data when it's being changed
- Possibly, construct a shell script, which takes timely backups
- Ensure that the shell script is bug-free

Avoiding invalid backups

We are going to continue the discussion on invalid backups. As you saw, HDFS makes three copies of our backup for the recovery process. What if the original backup was flawed with errors or bugs? The three copies will be corrupted copies; now, when we recover these flawed copies, the result indeed will be a catastrophe.

Our clause should include the following points in order to construct a sound backup:

- Avoid having a long backup frequency
- Have the right backup process, and probably having an automated shell script
- Track unnecessary backups

If our backup clause covers all the preceding mentioned points, we surely are on the way to making a good backup strategy. A good backup policy basically covers all these points; so, if a disaster occurs, it always aims to go to the last stable state. That's all about backups. Moving on, let's say a disaster occurs and we need to go to the last stable state. Let's have a look at the recovery philosophy and all the points that make a sound recovery strategy.

The recovery philosophy

After a deadly storm, we always try to recover from the after-effects of the storm. Similarly, after a disaster, we try to recover from the effects of the disaster. In just one moment, storage capacity which was a boon turns into a curse and just another expensive, useless thing. Starting off with the best question, what will be the best recovery philosophy? Well, it's obvious that the best philosophy will be one wherein we may never have to perform recovery at all. Also, there may be scenarios where we may need to do a manual recovery.

Let's look at the possible levels of recovery before moving on to recovery in Hadoop:

- Recovery to the flawless state
- Recovery to the last supervised state
- Recovery to a possible past state

- Recovery to a sound state
- Recovery to a stable state

So, obviously we want our recovery state to be flawless. But if it's not achieved, we are willing to compromise a little and allow the recovery to go to a possible past state we are aware of. Now, if that's not possible, again we are ready to compromise a little and allow it to go to the last possible sound state. That's how we deal with recovery: first aim for the best, and if not, then compromise a little.

Just like the saying goes, "The bigger the storm, more is the work we have to do to recover," here also we can say "The bigger the disaster, more intense is the recovery plan we have to take."

So, the recovery philosophy that we construct should cover the following points:

- An automation system setup that detects a crash and restores the system to the last working state, where the application runs as per expected behavior.
- The ability to track modified files and copy them.
- Track the sequences on files, just like an auditor trails his audits.
- Merge the files that are copied separately.
- Multiple version copies to maintain a version control.
- Should be able to treat the updates without impacting the application's security and protection.
- Delete the original copy only after carefully inspecting the changed copy.
- Treat new updates but first make sure they are fully functional and will not hinder anything else. If they hinder, then there should be a clause to go to the last safe state.

Coming back to recovery in Hadoop, the first question we may think of is what happens when the NameNode goes down? When the NameNode goes down, so does the metadata file (the file that stores data about file owners and file permissions, where the file is stored on DataNodes and more), and there will be no one present to route our read/write file request to the DataNode.

Our goal will be to recover the metadata file. HDFS provides an efficient way to handle NameNode failures. There are basically two places where we can find metadata. First, `fsimage` and second, the edit logs.

Our clause should include the following points:

- Maintain three copies of the NameNode.
- When we try to recover, we get four options, namely, continue, stop, quit, and always. Choose wisely.
- Give preference to save the safe part of the backups. If there is an **ABORT!** error, save the safe state.

Hadoop provides four recovery modes based on the four options it provides (continue, stop, quit, and always):

- **Continue**: This allows you to continue over the bad parts. This option will let you cross over a few stray blocks and continue over to try to produce a full recovery mode. This can be the **Prompt when found error** mode.
- **Stop**: This allows you to stop the recovery process and make an image file of the copy. Now, the part that we stopped won't be recovered, because we are not allowing it to. In this case, we can say that we are having the **safe-recovery** mode.
- **Quit**: This exits the recovery process without making a backup at all. In this, we can say that we are having the **no-recovery** mode.
- **Always**: This is one step further than continue. Always selects continue by default and thus avoids stray blogs found further. This can be the **prompt only once** mode.

We will look at these in further discussions. Now, you may think that the backup and recovery philosophy is cool, but wasn't Hadoop designed to handle these failures? Well, of course, it was invented for this purpose but there's always the possibility of a mashup at some level. Are we overconfident and not ready to take precaution, which can protect us, and are we just entrusting our data blindly with Hadoop? No, certainly we aren't. We are going to take every possible preventive step from our side. In the next topic, we look at the very same topic as to why we need preventive measures to back up Hadoop.

Knowing the necessity of backing up Hadoop

Change is the fundamental law of nature. There may come a time when Hadoop may be upgraded on the present cluster, as we see many system upgrades everywhere. As no upgrade is bug free, there is a probability that existing applications may not work the way they used to. There may be scenarios where we don't want to lose any data, let alone start HDFS from scratch.

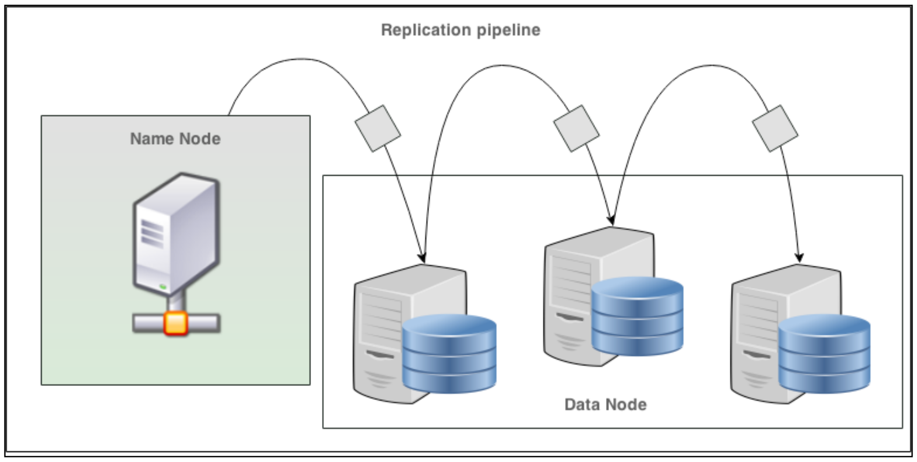
This is a scenario where backup is useful, so a user can go back to a point in time. Looking at the HDFS replication process, the NameNode handles the client request to write a file on a DataNode. The DataNode then replicates the block and writes the block to another DataNode. This DataNode repeats the same process. Thus, we have three copies of the same block. Now, how these DataNodes are selected for placing copies of blocks is another issue, which we are going to cover later in *Rack awareness*. You will see how to place these copies efficiently so as to handle situations such as hardware failure. But the bottom line is when our DataNode is down there's no need to panic; we still have a copy on a different DataNode.

Now, this approach gives us various advantages such as:

- **Security**: This ensures that blocks are stored on two different DataNodes
- **High write capacity**: This writes only on a single DataNode; the replication factor is handled by the DataNode
- **Read options**: This denotes better options from where to read; the NameNode maintains records of all the locations of the copies and the distance from the NameNode
- **Block circulation**: The client writes only a single block; others are handled through the replication pipeline

[During the write operation on a DataNode, it receives data from the client as well as passes data to the next DataNode simultaneously; thus, our performance factor is not compromised. Data never passes through the NameNode. The NameNode takes the client's request to write data on a DataNode and processes the request by deciding on the division of files into blocks and the replication factor.]

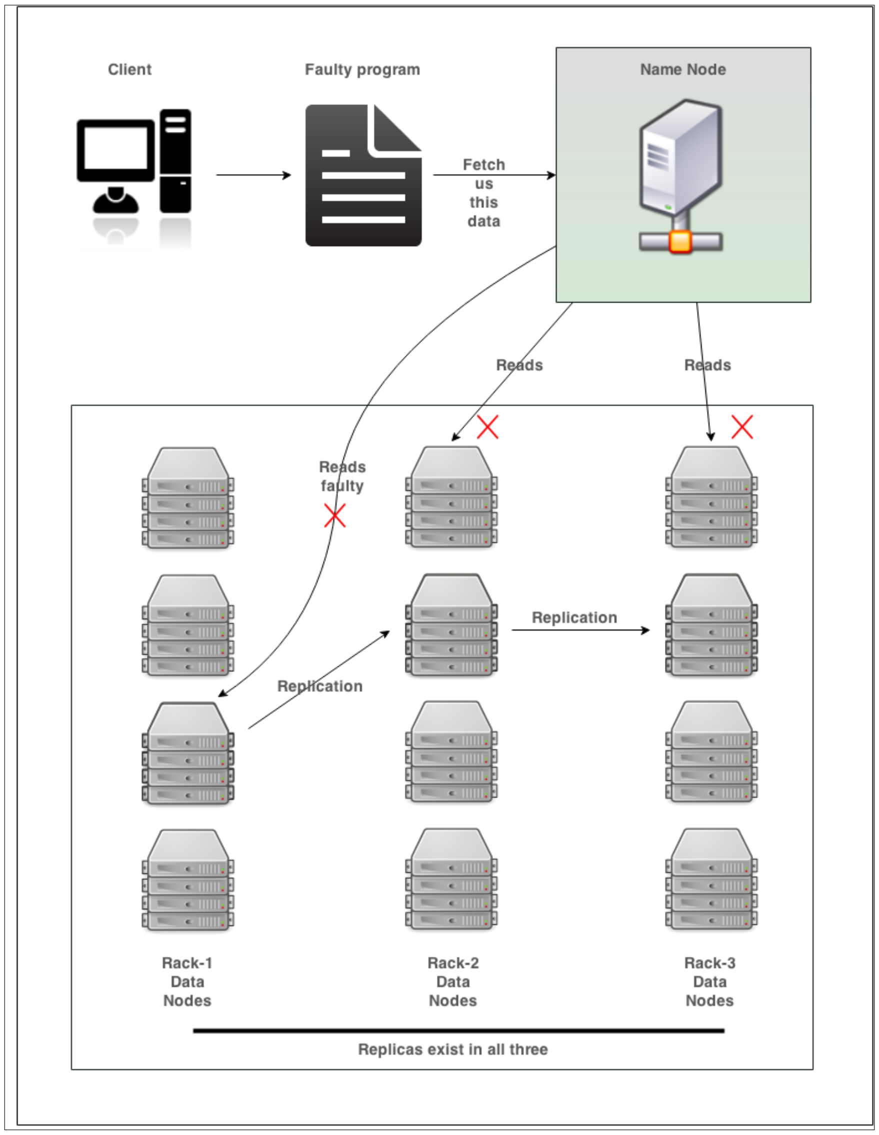
The following figure shows the replication pipeline, wherein a block of the file is written and three different copies are made at different DataNode locations:



After hearing such a foolproof plan and seeing so many advantages, we again arrive at the same question: is there a need for backup in Hadoop? Of course there is. There often exists a common mistaken belief that Hadoop shelters you against data loss, which gives you the freedom to not take backups in your Hadoop cluster. Hadoop, by convention, has a facility to replicate your data three times by default. Although reassuring, the statement is not safe and does not guarantee foolproof protection against data loss. Hadoop gives you the power to protect your data over hardware failures; the scenario wherein one disk, cluster, node, or region may go down, data will still be preserved for you. However, there are many scenarios where data loss may occur.

Consider an example where a classic human-prone error can be the storage locations that the user provides during operations in Hive. If the user provides a location wherein data already exists and they perform a query on the same table, the entire existing data will be deleted, be it of size 1 GB or 1 TB.

In the following figure, the client gives a read operation but we have a faulty program. Going through the process, the NameNode is going to see its metadata file for the location of the DataNode containing the block. But when it reads from the DataNode, it's not going to match the requirements, so the NameNode will classify that block as an under replicated block and move on to the next copy of the block. Oops, again we will have the same situation. This way, all the safe copies of the block will be transferred to under replicated blocks, thereby HDFS fails and we need some other backup strategy:



When copies do not match the way NameNode explains, it discards the copy and replaces it with a fresh copy that it has.

HDFS replicas are not your one-stop solution for protection against data loss.

Determining backup areas – what should I back up?

After having a look at the philosophy behind backup and recovery, we now will go through the various areas that should be backed up. The important areas basically cover the metadata file, application, and configuration file. Now, with each area, some important factors are associated, which need to be taken care of. Let's start our journey with datasets.

Datasets

Datasets are basically collections of data. Their size may range from a few terabytes to some petabytes. Let's take a look at the Facebook example; in one single day, about 250 TB of data is produced. For players such as Amazon and Facebook, data is everything, as based on the data analysis and computations they do, they provide good results. Also, associated with the datasets we have the metadata file. Without the metadata file, we won't know which dataset is stored on which DataNode. Metadata is just like the human brain, routing each dataset to a specific DataNode. Metadata stores information as well. Hence, it is one of the most important factors to be considered while backup. More emphasis should be laid on it any day, rather than backing up raw data.

Starting with the metadata file, the metadata files usually contain the following:

- Block size
- The replication factor
- A list of all the blocks for a file
- A list of DataNodes for each block (sorted by distance)
- The ACK package
- The checksums
- The number of under replicated blocks

Going through each one of them in detail will give us a better understanding.

Block size – a large file divided into blocks

In block size, we generally see how large files are subdivided into small blocks. Now, each of these small blocks is distributed within subsequent DataNodes. The details of the DataNodes, that is, the default size of the block to be sent in DataNodes is generally 64 or 128 MB. Files larger than block size are divided into parts. So, if the default size is 64 MB and your file size is 100 MB, the other 36 MB will be distributed to another DataNode. The metadata file keeps a track of these distributions.

Now, you can decide what size of block you want to store. The default value that can be overridden is 64 MB / 128 MB. Just go on to the `hdfs-default.xml` file.

You can find the `hdfs-default.xml` file at the `hadoop/conf` directory.

Find the `dfs.block` property size and change it as per your need. You can use the `(K(kilo),m(mega),g(giga),t(tera),p(peta),e(exa))` suffix, or you can write the corresponding byte value:

```
<property>
<name>dfs.block.size<name>
<value>134217728<value>
<description>Block size<description>
<property>
```

Replication factor

The replication factor basically covers the number of copies to be made. By default, the replication factor is three; this means it will keep three copies on different DataNodes. Now, where exactly we keep these copies is another topic that we are going to see later. The bottom line is that the replication factor is defined here, which tells you how many replicas to keep. The replicas are distributed to different DataNodes.

Now, if you want to change the default replication factor, go to `hdfs-site.xml`, look for the `dfs.replication` property, and change it to your desired value. If you keep the property as five, then there will be four replicas and one original block, as shown here:

```
<property>
<name>dfs.replication<name>
<value>3<value>
<description>Block Replication<description>
<property>
```

If you want to change it on a file basis, you can do so with the help of the following command:

```
hadoop fs -setrep -w 3 /location/to/file/
```

A list of all the blocks of a file

Now, when we are dealing with a file whose size is greater than the default block size defined in `hdfs-default.xml`, it's obvious that it will be divided into separate blocks. So, we need a place where we can hold information such as how it is divided, the number of blocks involved in the file, where the blocks are stored, and more. All this information is stored in the NameNode's metadata file.

A list of DataNodes for each block – sorted by distance

So, going ahead in the picture, it's obvious that we need the location of the DataNode where our blocks are going to be placed. The metadata file contains this information as well. Now, it has a distance parameter which defines how far the DataNode is from the NameNode. In metadata, the list of NameNodes is placed based on this very parameter. Simply put, when the file size exceeds the default block size, the file is divided into blocks, which are placed on other DataNodes. The NameNode decides where to keep the other blocks and maintains their location in its metadata file.

The ACK package

The ACK package stands for the acknowledgement package. When a file is sent to DataNodes, it sends an acknowledgement package back to the NameNode as a confirmation that it received the file. If an acknowledgement package is not received within a standard time, that is, generally 10 minutes, the NameNode declares the DataNode as dead and moves to the next DataNode.

The checksums

So, when we are talking about dividing the files into blocks, we generally think of how those files will be merged again and in the same sequence. That's where a checksum is associated. With each block of file, the NameNode appends a number to the block and keeps track of the number in the checksum. This information is stored in the metadata.

The number of under-replicated blocks

The number of under replicated blocks basically tells us about how many blocks failed to replicate at a proper level.

To fix these problems, try one of the following solutions:

- First, note that the system will always match the replication factor; so, if you have data that is under replicated, it should automatically adjust as it will replicate the block to other DataNodes.
- Run a balancer. It's a cluster balancing tool.
- Just set a bigger replication factor on the file, which is under replicated.

A typical structure of a small clustered setup is as follows:

Configure capacity	223.5 GB
DFS used	38.52 GB
Non DFS used	45.35 GB
DFS remaining	148.62 GB
DFS used percent	16.57 percent
DFS remaining percent	0.26 percent
Number of live nodes	4
Number of dead nodes	0
Number of decommissioning nodes	1
Number of under replicated blocks	5

Let's take a look at the basic components found in a typical metadata file:

- **Configure capacity**: This denotes the capacity of the NameNode.
- **DFS used**: DFS is generally the distributed filesystem used. It tells you about the files stored in `dfs.directories`.
- **Non DFS used**: Simply speaking, this includes all those files that are stored locally. For example, the log files. All those files that are not in the `dfs.directories` files.
- **DFS used percent**: This just gives a percentage copy of the DFS used.
- **DFS remaining percent**: This just gives a percentage copy of how much storage is remaining.

- **Live nodes**: This denotes the number of nodes that are alive and continuously send an acknowledgement packet/heartbeat. The NameNode declares those nodes as dead who don't send an acknowledgement package back.
- **Dead nodes**: This denotes the number of nodes that failed to send an acknowledgement packet or a heartbeat back. So, NameNode tracks all those nodes that have either crashed or suffered from massive failures. NameNode then disconnects them from the loop.
- **Decommissioning nodes**: This denotes the number of nodes that we have removed from the Hadoop cluster.
- **Number of under replicated blocks**: This talks about the blocks of data that have not been replicated properly either due to non-availability of the DataNodes or due to some human-prone errors.

If you want a report in your cluster, just enter the following command:

```
hadoop dfsadmin -report
```

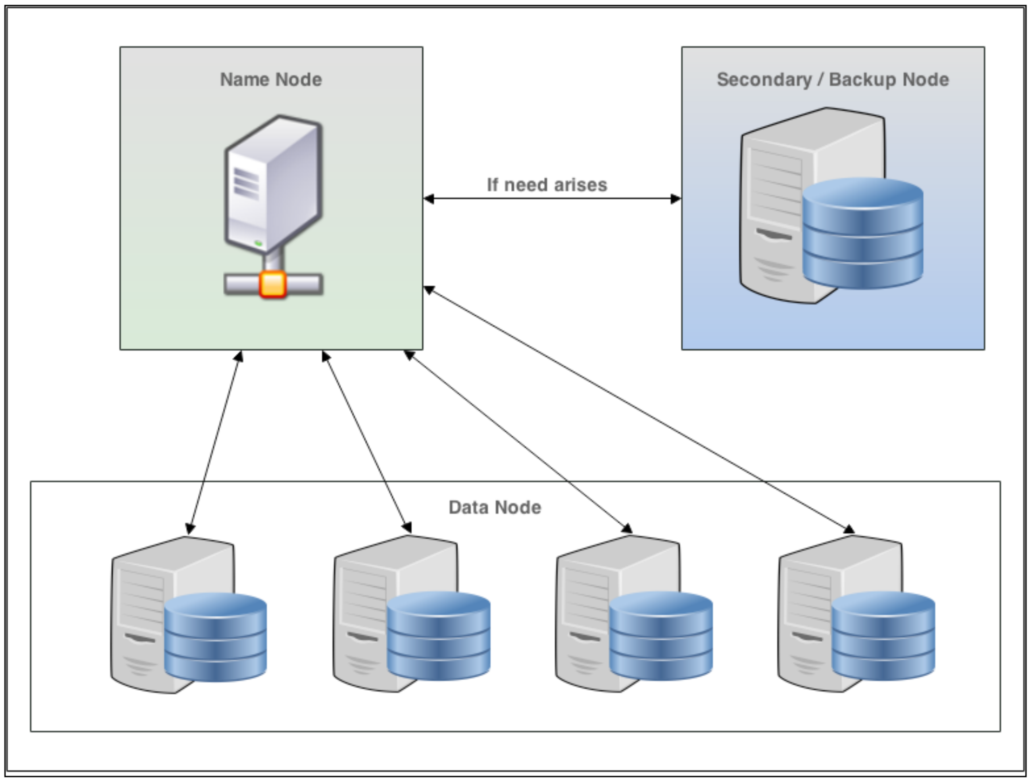
The output will be similar to the following:

```
Configured Capacity: 422797230080 (393.76 GB)
Present Capacity: 399233617920 (371.82 GB)
DFS Remaining: 388122796032 (361.47 GB)
DFS Used: 11110821888 (10.35 GB)
DFS Used%: 2.78%
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0
-------------------------------------------------
Datanodes available: 5 (5 total, 0 dead)
Name: 10.145.223.184:50010
Decommission Status : Normal
Configured Capacity: 84559446016 (78.75 GB)
DFS Used: 2328719360 (2.17 GB)
Non DFS Used: 4728565760 (4.4 GB)
DFS Remaining: 77502160896(72.18 GB)
DFS Used%: 2.75%
DFS Remaining%: 91.65%
Last contact: Thu Feb 28 20:30:11 EST 2013
```

Hence, protecting your metadata is a more important task because if the NameNode's metadata is lost, the entire filesystem is rendered unusable. To simplify this further, you will have the body but no brain.

> Here's a piece of advice: always make multiple copies of different stages of these files, which will result in viable protection against data corruption residing in the copies itself or in the live files running on the NameNode.

So, continuing our discussion on metadata, the question of how we are going to do the NameNode recovery arises. Let's look at the basic architecture:



The figure shows the involvement of the secondary NameNode, which we will understand first.

The secondary NameNode

To understand the secondary NameNode, we will first have to understand what happens when the NameNode starts up, which will lead us to the need for a secondary NameNode. As you know, the NameNode stores metadata. Simply saying, the NameNode maintains a log of the changes to the filesystems and it stores the log in `edits`. When we start the NameNode, it does the following two things:

- Reads the current HDFS state from `fsimage` (the maintained image file of data)
- Applies edits from the edit logs file

Now, it writes the HDFS state into `fsimage` and starts the operation again, nullifying the contents of the edit log file.

You can clearly see that we have a problem here. NameNode only merges `fsimage` and `edits` when it starts up. So, what happens if we make huge changes after the NameNode starts? Our changes will be maintained in the edit logs, but clearly, its size will be huge, so our performance is compromised on the startup of the next NameNode.

We need something, which within a regular time frame (a fixed time interval), merges `fsimage` and edit logs and thus keeps the size of the edit logs within a particular limit. That's where the secondary NameNode comes into the picture, and this is the core functionality of the secondary NameNode. The secondary NameNode usually runs on a different machine than the primary NameNode. Two factors are defined here:

- `fs.checkpoint.period`: This gives the period after which `fsimage` and edit logs are merged. This is the regular time interval we were talking about in the preceding paragraph.
- `fs.checkpoint.size`: This is the maximum size of the edit logs beyond which it won't allow the edit logs to go.

> Don't go by its name and assume that it's a good standby for the NameNode. The secondary NameNode only compacts edit logs. It doesn't have an exact copy of the metadata. It periodically copies `fsimage` and `EditLog` from the NameNode and merges `fsimage` with the log file. It provides a high availability NameNode. Its purpose is to have a checkpoint in HDFS.

Coming back to the question of how the NameNode will be recovered, the secondary NameNode maintains a copy of the NameNode metadata. Recovering this file is beneficial so that it can be used when we move from one node to another, to make it the NameNode.

Now, looking at the point of when the need for NameNode recovery arises, it is obviously when the NameNode crashes. The NameNode crashes due to the following reasons:

- Hardware failures
- Misconfiguration
- Network issues

Let's continue with the example of Yahoo!. They noted that in a period of 3 years, out of the 15 NameNodes that failed, only three were due to hardware failures. So, be on a red alert for misconfiguration files.

[NameNodes are very less likely to fail because of hardware failures.]

Coming back to recovery, three options are available.

Fixing the disk that has been corrupted or repairing it

A simple `fsck` command examines all the files, tells us about the files that contain corrupted blocks, and tells us how to fix them. However, it just operates on data and not on metadata.

[The `fsck` command works only on data and not on metadata.]

Recovering the edit log

We can recover a partial or corrupted edit log. This is a manual NameNode recovery, which is an offline process. An administrator can recover corrupted edit logs via this process. Through the corrupted edit log, we can clearly find the corrupted filesystems.

Recovering the state from the secondary NameNode

It is a script that periodically saves the secondary NameNode's previous checkpoint location to a different location. The previous checkpoint can be set by the `fs.checkpoint.dir` property. While saving the file to a different location, the file should not break. Two different calls to the `/getimage` servlet are included in NameNode's web server:

- The first call uses `getimage=1` and gets `fsfileimage`
- The second uses `getedit=1` and gets the `edits` file

In both the situations, we get data from a consistent point on the disk.

After ensuring that we have our brain safe and sound, we will move on to when we need backup and recovery plans. During the following classes of problems, we need to plan and formulate a backup and recovery strategy.

Active and passive nodes in second generation Hadoop

What we saw until now was first generation Hadoop architecture. In generation 1 Hadoop, the NameNode is a single point of failure. So, if the NameNode is gone, there's no way we can restore everything quickly. In generation 2, we now have passive and active NameNodes kind of structure. If the active NameNode fails, the passive NameNode takes over.

Later, in the upcoming chapters, you will learn about the ZooKeeper service, which acts as a coordinator between the active and secondary NameNode and ensures that the secondary NameNode is always in sync with the active NameNode. So, if the active NameNode fails, the secondary NameNode starts taking over as the primary NameNode.

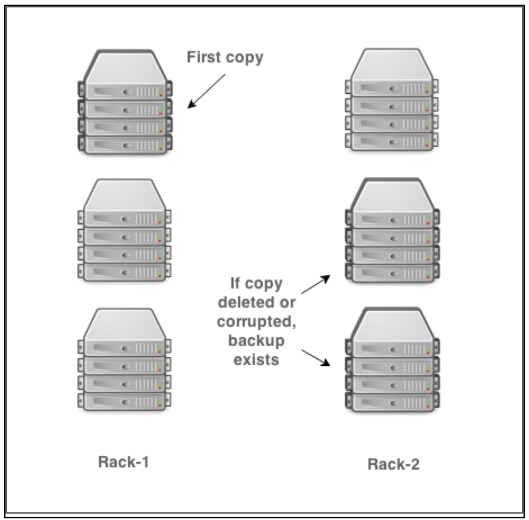
Hardware failure

Hadoop is designed on the following principle:

"*Make the software so acute that it gets the power to deal with hardware failure.*"

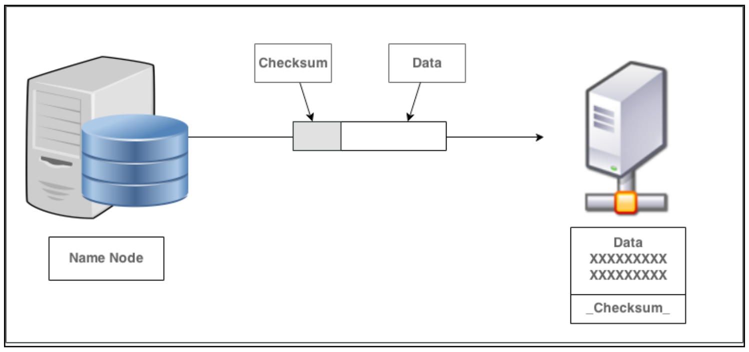
The idea is to implement the principle of replication: make multiple copies and save them on various machines. The standard is to make three copies; so, when a node fails, even if the data on the node will not be available, we will still have access to the data.

We will look at the possible three hardware failure situations in the sections that follow. Have a look at the following screenshot:



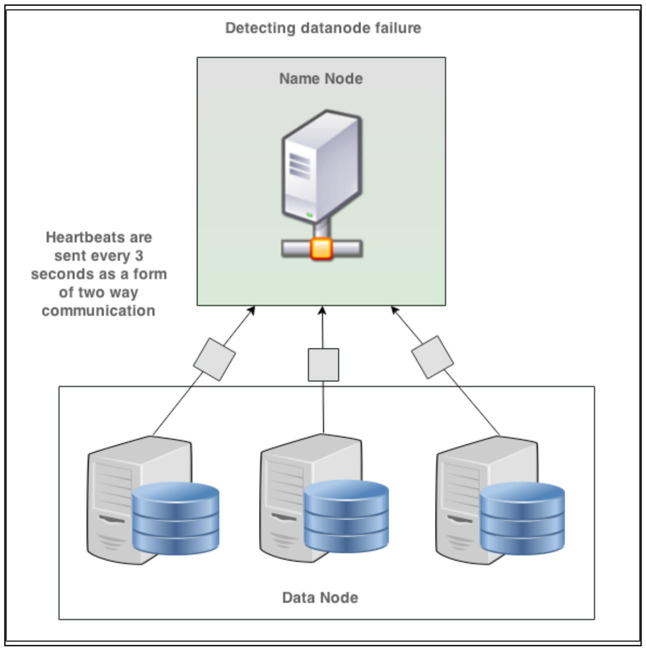
Data corruption on disk

A basic principle of checksums is used to detect data corruption. Corruption may occur during any stage (transmission, read, or write). In Hadoop, DataNodes verify checksums and if the need arises, prevent the data blocks and ask the blocks to be resent. The checksums are stored further, along with blocks, for verifying the process of integration. Failures are reported, which starts the re-replication of healthy replicas. If a corrupt block is found, the NameNode re-replicates the block from its uncorrupted replica and sets the corrupted block to be trashed out. Have a look at the following screenshot:



Disk/node failure

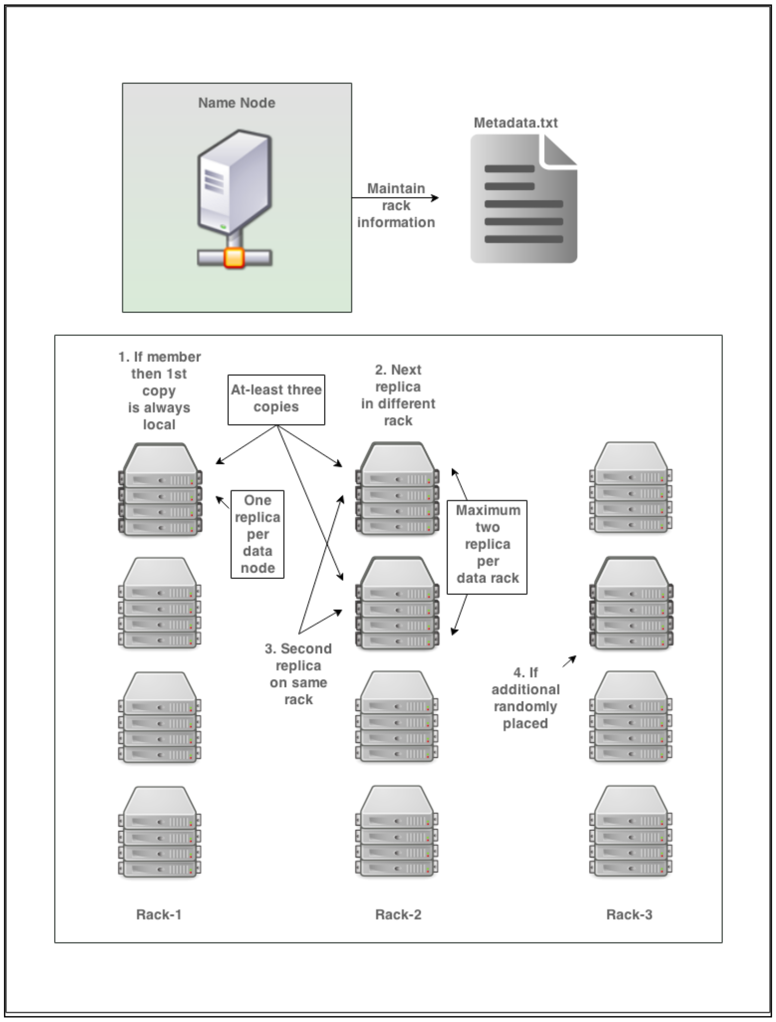
The NameNode always listens for the heartbeats sent out by the DataNodes. If the DataNode's heartbeats are not received by the NameNode, then after a certain span of time (10 minutes), it assumes that the node is dead. It removes the dead node from the list, leading to the replication of blocks that were on the failed node using the replicas present on other nodes. Thus, the process of synchronous replication is beneficial. The first two replicas are always on different hosts. The process follows from the NameNode to DataNode 1 to DataNode 2, that is, the replication pipeline, as shown here:


Detecting datanode failure
Name Node
Heartbeats are sent every 3 seconds as a form of two way communication
Data Node

Rack failure

Throughout the chapter, we came across the replication factor several times, where I mentioned that we will cover how the blocks are placed on different racks. Well, here it is, brace yourselves. We will look at what a rack is, how blocks are placed among racks, and how to recover from rack failures.

Coming back to the Yahoo! example, for such a large enterprise, it is of utmost necessity to have zero downtime. Hence, they divide racks in to two parts: operational and nonoperational. It is of utmost importance for them that their operational racks never go down:


Name Node
Maintain rack information
Metadata.txt
1. If member then 1st copy is always local
At-least three copies
2. Next replica in different rack
One replica per data node
Maximum two replica per data rack
3. Second replica on same rack
4. If additional randomly placed
Rack-1
Rack-2
Rack-3

Starting off, a rack is basically a collection of DataNodes within the same vicinity. So, when a DataNode goes down, it's fine since we have a copy, but what happens when the entire rack goes down? The copy of the block at the distinct location in the same rack will also be gone, so it's wise to place a block or a copy of the block on a different rack. Block replicas are written to separate DataNodes (to handle DataNode failures) or written in different racks (to handle rack failure).

While replicating through HDFS, the following points should be implemented, or to put this in better words, the following points constitute the rack management policy:

- Configure at least three replicas and provide rack information (`topology.node.mapping.impl` or `topology.script.file.name` so as to get its rack ID)
- The third replica should always be in a different rack
- The third copy is of utmost importance, as it allows a time window to exist between failure and detection
- If a person who wants to write data is a member of a cluster, then it's always selected as the place for the first replica
- For the next replica location, pick a different rack than the first one
- There should be one replica per DataNode
- There should be a maximum of two replicas per rack

The following are the advantages of our rack policy:

- Our replicas are not evenly distributed across the racks. We have placed one third on one rack, two thirds on another rack, and the remaining one third is evenly distributed somewhere else. Hence, it gives protection against node failure / rack failures.
- It improves the write performance incredibly, as data is being written on the DataNodes as well as sent to other DataNodes for replication. Subsequent copies are written by the DataNode. Thus, write performance is enhanced without compromising read performance or data accuracy.

When files are written from the NameNode to the DataNode, a pipeline is formed with the objective of creating replica copies in a sequence. Data is passed through packets (depending on the file size; if it exceeds the default size, the file is divided into blocks). With each block sent, the NameNode performs the process of a checksum and heartbeat. If the DataNode fails to pass those tests, the NameNode will make its copy again to cover up for the missing copy. Upcoming blocks will be written using a new pipeline.

Now, we will look at another class, that is, software failure. We will walk through possible software failure causes. As a general law of nature, "no system is ever going to be without a bug." Recalling a very famous quote, "If debugging is the process of removing software bugs, then programming is the process of putting them in." Well, the bottom line is that you will always have to struggle with software bugs.

Software failure

Hadoop provides large data guards that are capable of doing recovery from any kinds of disasters. It is possible to have software bugs that affect all our replicas but again it's a rare case scenario. Like any development, the best we can do when dealing with software bugs is the following:

- Unit testing of the module
- Code review (by fellow peers as well as tools, if available)
- System testing

During the following classes of problems, we need to plan and formulate a backup and recovery strategy:

- **Accidental or malicious data deletions**: The most classic human-prone error is when something is deleted by accident. We have a common solution: recycle bin / trash. So, it is obvious logic that if the trash system is enabled (by default, it is not), files that are removed from the Hadoop filesystem are moved into the hidden trash directory instead of being deleted immediately. If the trash server is enabled, files are moved to the trash, whereby the possibility of recovery is still present. Note the following points about the trash system available in Hadoop:
    - Enable `fs.trash.interval` to set the trash interval.
    - The trash server only works through the command line, that is, the `fs` shell. Programmatic deletes will not activate the trash mechanism.
    - The trash mechanism is not always foolproof. Trash is only supported by command-line tools; so if someone deleted a folder using Hadoop, the API's data will not be recoverable using the trash mechanism.

> The trash system is not enabled by default. The files in trash are removed periodically. Always let the system delete data instead of immediately deleting the data. The trash deletion period is configurable.

- **Permissions facility**: Let's forget Hadoop for a minute and go to applications such as Liferay. It has a very organized structure wherein someone assumes the role of the admin, group owner, and so on. However, the fascinating thing is the permissions-based structure. A group owner is not able to have admin rights. Well, the idea is similar in Hadoop where clusters are shared. A permissions-based structure is there in HDFS, where users have access to the data, but there's also a restriction on the changes made by the user at the Hadoop directory level. The following are the best practices adopted generally:
    - To prevent data loss on a cluster shared by all, apply the principle of minimum access.
    - Users should only have access to the data they are supposed to access and limit the number of files/bytes of files rooted at the directory level.
    - Set apart different processes in the data pipeline using the permission control on files and directories. For example, categorizing the data into read-only / can't delete files / access / no-access modes.
    - There may be different downstream processes; ensure that they have different user names and corresponding user roles have been assigned.
    - Carefully allocate permissions on users for the data they produce and the data they consume.
    - Have quota management by restricting the number of files / bytes of files.

Well, that pretty much concludes datasets. We looked at major reasons, possible ways, and solutions provided. Moving on to the next item, "applications", which need to be in our plan.

Applications

We come across various applications in Hadoop, some custom made and some available in Hadoop, out of the box. Say, for example, MapReduce, JobTracker, and many more; when we talk about backup, we need to talk about these too. The applications that need to be backed up include the following:

- System applications (JobTracker, NameNode, region servers, and more)
- User applications (custom applications written to smoothen various processes in Hadoop, such as various MapReduce programs)

You have already seen how to deal with the backup of NameNodes. Now, looking at JobTracker, it clearly is a single point of failure. If JobTracker goes down, then again all the running nodes are down and we are stuck again at a deadlock.

Configurations

Now, let's look at the third part, that is, the configuration file, which basically maintains a list of the permissions given to users and stores them in a special parameter called the access control list.

We recently spoke about the least permissions privilege policy, only granting access that the user needs. For example, an editor can only edit and not delete a file. Now, this is maintained in Hadoop by a mechanism called service level authorization. It ensures that all the users associated with the Hadoop service have the necessary, preconfigured permissions and are authorized to use the application.

The `hadoop-policy.xml` file is used to define access control lists for various Hadoop clients, such as HBase. So, taking a backup of this file is extremely necessary as it provides information regarding the rights of the users.

By default, service level authorization is disabled for Hadoop. To enable it, set the configuration property `hadoop.security.authorization` to `true` in the `$HADOOP_CONF_DIR/core-site.xml` directory.

For a parameter in service level authorization, the corresponding knob contains the corresponding access control list. Let's see a few examples, as follows:

- Consider a basic configuration file that we want to back up. The scenario is "allow any user to talk to the HDFS cluster as a DFS client":

  ```
  <property>
          <name>security.client.protocol.acl</name>
         <value>*</value>
  </property>
  ```

  This protocol says that all users are included here.

- Allowing only a few users to submit jobs to the MapReduce cluster. It is done using the following script:

  ```
  <property>
       <name>security.job.submission.protocol.acl</name>
       <value>parth-ghiya mapreduce</value>
  </property>
  ```

  Taking a backup of this file is thus quite important.

We saw the basic three components that have an impact on an application running in Hadoop.

Let's recall all those three components:

- **Datasets**: These are collections of data, be it distributed over the cluster or in any other form.
- **Applications**: Various custom applications are written to smoothen the Hadoop process, such as various MapReducer programs and in-built applications, such as JobTrackers, and more.
- **Configurations**: This is the part where permissions, necessary preconfigured settings, and everything else is handled. It describes various Hadoop services and their corresponding knobs values. We also define various access control lists here.

Each of the three components have an equal emphasis and can have a severe impact. You saw what each component covers and how it can impact the overall system.

Moving on to the next topic, when will we need backup? The answer is in scenarios where we are stuck with a problem, we want to go to the previous state, or when we want to migrate data from a small cluster to a bigger cluster. But is backup my one-stop solution? Will we be able to recover the data fully if we have backups? In the next topic, we are going to see all these topics and cover these in detail.

Is taking backup enough?

Well, it's obvious. With the data explosion that is going on at the present moment, sooner or later, there is going to come a moment when we may be facing various tasks such as:

- Moving the data center from one place to another
- Making sure that the downtime for the application is almost zero
- Making sure that critical datasets are not lost during the data migration process or the recovery process

So, a mere backup plan won't be much help unless we formulate the corresponding recovery plan to come out of the disaster. During disaster or data migration processes, each and every moment is useful, because if the application is down for some uncounted time, we are going to suffer terribly. Take the example of Amazon; going back a few years, Amazon was down for 2 hours in June 2008.

During this time, studies shows that they might have lost $30,000 a minute, that is, $36,00,000 in 2 hours. Imagine the impact it must have had on its annual revenue generated. So, to avoid situations like these, we need a good disaster recovery plan. The next topic lays deep emphasis on the disaster recovery plan. It tells us what a disaster recovery plan should constitute and the aspects that it should focus on.

Also, we can see one example of how Facebook handled its data migration process (transferring about 30 petabytes of data, which is approximately 30,000 terabytes of data). The application downtime was almost zero. So, hold on to your seats and control your excitement; we are going to arrive there very soon.

Understanding the disaster recovery principle

To start off, let's first understand what exactly a disaster is.

Knowing a disaster

Traditionally, a disaster is "any such condition wherein a person may not get their hands on a required feature or a resource for a significant period of time." The condition can cause the unavailability of resources/data. The user suffers scenarios such as a sustainable amount of damage or inability to use the application in immediate need, or the application fails to fulfill the purpose he needs.

In simple words, say you log in to your Facebook account and you are not able to use it for quite some time; perhaps, you will never use Facebook again and move on to Twitter. (Facebook ensures that this never happens, thus effectively handling disaster recovery situations.) For big players such as Amazon, Facebook, and Twitter, user data is mission critical. Loss of this data will lead to non-use of the application.

The disaster recovery principle will address the following points:

- **The situations you encounter are dissimilar**: This is the most important point, which talks about a simple thing. The situation you may be facing will not be the same as the situation some other people will be facing. Each of the disaster situations is unique. Gather your needs and requirements; assess the thump of losing the application and the data associated with it. Forestall it and take appropriate precautions. Let's see an example depicting how situations may vary; say, we have a bank application, so our needs will be high availability time and the withholding of data in case of impersonation or fraudulent access. Moving on to a social media application, here definitely we don't need the "withholding of data" factor.

- **Independent from service or maintenance**: In the case of disaster, don't abandon your existing state and switch to some other framework. Just follow the routine principles of data recovery and adhere to some standards. Understand and forestall the impacts of data loss and take preventive measures accordingly in advance.
- **Division of data into critical/noncritical**: This factor basically involves making our critical data stand apart from our noncritical data. In the case of disasters, we include a clause that says data which we classified as critical should be recovered at any cost. Failure to recover this data will result in a serious breakdown. We must determine the data that is critical to us, where we cannot compromise at all.
- **Considering the significance window**: This factor basically talks about the data we are going to store and its significance. So, it's as simple as whether the data that we are going to store in abundance is of any significance. Are we going to use it further? Let's see an example; storing facts and statistics involved in various sports is a part of significant data, because we can use it in the future as a comparison tool. A similar example will be weather data and reports, but data such as how many people went to a particular social media gathering is of no viable value. Taking a backup of such data will be just a waste of our precious resources.
- **Right storage format**: Now, this is purely situation specific. These questions are discussed here: what are we going to use? Cloud storage, on-premise storage, or data disks? What are the risks associated with each of the storage forms? What will be the impact of each medium? How much recovery time we will have? Based on the situation, we should select wisely. If we use cloud storage, how different is my disaster recovery strategy going to be?
- **Right format**: Choosing the right format of data is of utmost importance. Is the data raw, that is, is the data not processed? Do we need raw data? Is our data aggregated (derived from various sources) or is my data standalone/ nonaggregated? What kind of data format do I basically need? Let's see an example: we collect a record each second during the whole year, now that's 31,536,000 entries in a year. But what if we summarize the data into hours/ weeks/months? We will have a major reduction by more than 90 percent for sure (that's how summarized data is useful). Now, what if we divide the data into minimum values / maximum values / mean / median / mode? This will give a reduction by more than 95 percent. So, the question is, do we really need more than 31 million records when we can get a summary from a smaller number of records? It's pretty simple that we don't.

Now that we have pretty much looked at what ideally a good disaster recovery plan covers, let's look at how Facebook implemented the data migration process during March 2011. Let's see how Facebook handled the disaster when it had to obtain a new data center and shift data there. They called the project "Moving an Elephant."

Within the last 5 years, Facebook has moved its data center because its data had grown up to 30 PB (30 petabytes can be 3,000 times the library of Congress or, better still, 30 times all the data used to render high-end graphic movies, such as *Avatar*). The noteworthy part about the movement was the disaster recovery solution they came up with. The solution provided the downtime of the application to be nearly zero and considering the huge amount of data, this looked nearly impossible.

The solution provided a way for the application to be in the running state despite the data being moved from one center to another, because users/analysts depend on Facebook 24 / 7 / 365, all the time. With increasing data, the company plans to make data centers at two other locations. To efficiently manage such data at different centers and to effectively recover from disaster (if it occurs) is what we are going to look at here.

Yang, the person responsible for the operation, quoted, "With replication deployed, operations could be switched over to the replica cluster with relatively little work in case of a disaster. The replication system could increase the appeal of using Hadoop for high-reliability enterprise applications." Another important statement was the involvement of Hive in the disaster recovery process.

Let's go through their disaster recovery journey. They thought of the following solutions:

- Just simply move all the machines physically from one place to another. But soon, they realized that this was not at all a good option. There are too many people depending on Facebook for many different purposes, and they depend on Facebook 24/7 and on all days. So, this was a poor option as the downtime would be too much and ultimately the company would face a breakdown.
- Just simply mirror the data. It will reflect the data from the old cluster to the new cluster with a larger capacity. When switch-over time occurs, we will simply redirect or reroute everything to the new cluster. But here's the catch: they were dealing with a live system where files were constantly added, removed, and modified. As replication minimizes downtime, this was the approach they decided to go with.

How did Facebook undertake its disaster recovery plan? The replication process was executed in three core steps:

1. First, a bulk copy was made, which transferred most of the data from the source destination to the larger cluster. Most of the copies were done through the `distcp` command (the `distcp` command prebundled with Hadoop, which uses MapReduce to copy files in parallel). Well, this was a child's play.
2. After the bulk copy was made, changes in the file that occurred after the bulk copy was made were detected through a Hive plugin, which made those changes in an audit log. The replication constantly pulled the audit log and copied those changes to the new cluster.
3. When migration time came, they shut down the JobTracker, so no new files could be created. Replication was allowed to be caught up and then both the clusters were identical. Now, it was just child's play. Change the DNS, start the JobTracker, and get the system up and running.

The bottom line is that Facebook handled disaster recovery situations efficiently through the replication process.

Now that we have understood what a disaster is, what constitutes a disaster recovery policy? We even went through an example of how Facebook devised its recovery plan and successfully implemented it; we will go through the needs for recovery.

The need for recovery

Now, we need to decide up to what level we want to recover. Like you saw earlier, we have four modes available, which recover either to a safe copy, the last possible state, or no copy at all. Based on your needs decided in the disaster recovery plan we defined earlier, you need to take appropriate steps based on that.

We need to look at the following factors:

- The performance impact (is it compromised?)
- How large is the data footprint that my recovery method leaves?
- What is the application downtime?
- Is there just one backup or are there incremental backups?
- Is it easy to implement?
- What is the average recovery time that the method provides?

Based on the preceding aspects, we will decide which modes of recovery we need to implement. The following methods are available in Hadoop:

- **Snapshots**: Snapshots simply capture a moment in time and allow you to go back to the possible recovery state.
- **Replication**: This involves copying data from one cluster and moving it to another cluster, out of the vicinity of the first cluster, so that if one cluster is faulty, it doesn't have an impact on the other.
- **Manual recovery**: Probably, the most brutal one is moving data manually from one cluster to another. Clearly, its downsides are large footprints and large application downtime.
- **API**: There's always a custom development using the public API available. We will move on to the recovery areas in Hadoop.

Understanding recovery areas

Recovering data after some sort of disaster needs a well-defined business disaster recovery plan. So, the first step is to decide our business requirements, which will define the need for data availability, precision in data, and requirements for the uptime and downtime of the application. Any disaster recovery policy should basically cover areas as per requirements in the disaster recovery principal. Recovery areas define those portions without which an application won't be able to come back to its normal state. If you are armed and fed with proper information, you will be able to decide the priority of which areas need to be recovered.

Recovery areas cover the following core components:

- Datasets
- NameNodes
- Applications
- Database sets in HBase

Let's go back to the Facebook example. Facebook uses a customized version of MySQL for its home page and other interests. But when it comes to Facebook Messenger, Facebook uses the NoSQL database provided by Hadoop. Now, looking from that point of view, Facebook will have both those things in recovery areas and will need different steps to recover each of these areas.

Summary

In this chapter, we went through the backup and recovery philosophy and what all points a good backup philosophy should have. We went through what a recovery philosophy constitutes. We saw the modes available for recovery in Hadoop. Then, we looked at why backup is important even though HDFS provides the replication process.

Next, we moved on to the key areas that require backup. We looked at datasets, which generally constitute blocks of the file evenly distributed across the DataNodes. We looked at the metadata file and what it contains. You saw how to back up the metadata file. We went through the scenarios when we will need a backup of the datasets. Then, we went through backup needs in the application and the configuration file. Finally, we looked at the disaster recovery philosophy.

We looked at what is a disaster and what all factors we should think of when we determine a disaster recovery policy. Then, you saw how Facebook carried out its disaster recovery philosophy successfully. Lastly, we looked at the recovery needs and areas. Quite a journey, wasn't it? Well, hold on tight. These are just your first steps into **Hadoop User Group** (**HUG**).

3
Determining Backup Strategies

In *Chapter 2*, *Understanding Hadoop Backup and Recovery Needs*, we understood the philosophy for backup and recovery. This chapter assumes you have gone through the previous chapter and are familiar with concepts of NameNode, DataNode, and how communication occurs between NameNodes and DataNodes.

In this chapter, we would learn which areas of Hadoop need to be protected while running Hadoop. We would also learn what the common failure types are, which includes hardware failures, user application failures, and primary site failures. If we have a good understanding of failure types, we would define a backup strategy. We would learn the need of backup for Hive metadata.

Before we start doing backup, defining backup strategy is an essential part to consider before implementing it. Here, we would cover topics to define backup strategy for various possible failures of Hadoop.

Knowing the areas to be protected

While using Hadoop, we would be storing a lot of data. Having said that, most of us will have questions, such as how are we going to do backup? What is our backup strategy? What is going to be our recovery plan? What is our business continuity plan? Let's overcome some of these questions in this chapter.

While working on such a large repository of data, it becomes quite a difficult problem to overcome issues. However, proper backup strategy would help to address most of the known scenarios. Having lot of new data inserted in a large cluster makes it more difficult to nail down what has changed since the last backup. We should take care of a few key concerns when we define backup strategy.

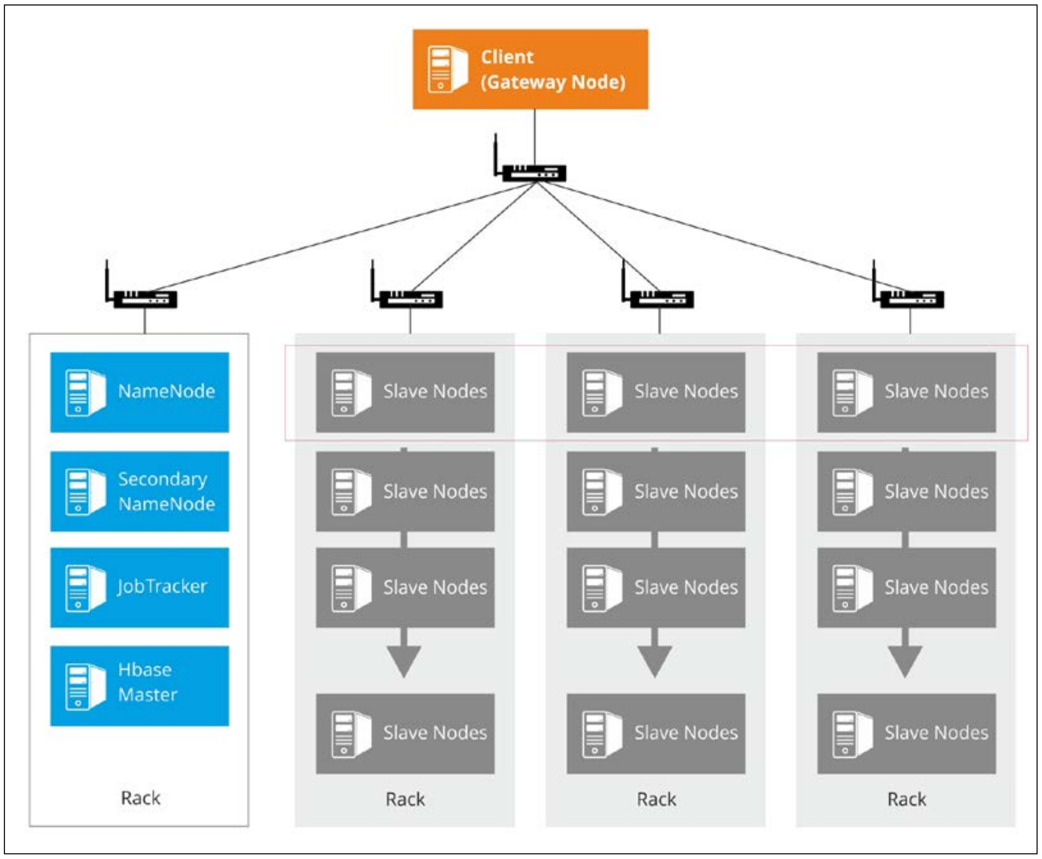
All backup solutions need to deal explicitly with a few key concerns. Selecting the data that should be backed up is a two dimensional problem, as both the critical datasets, that is, subset of the data within each dataset that has yet not been backed up, are as important as the first one.

Defining timeliness of backup is an equally important question that has to be addressed. Increasing the frequency of a backup may not be a feasible option as it has its own overheads. Backup frequency might be lesser, which fetches large chunks of data; that can incur a bigger window for loss of data. Since we have backups, one of the most difficult problems that must be never overlooked is consistency of backup as it may jeopardize data across clusters.

Consistency of backup also includes how we tackle copying of data while it's continuously getting updated that can possibly result in an invalid backup. Hence it would be a good reason to have knowledge of underlying filesystem on which the application functions become helpful. One of the examples, which those who have experience of relational databases would be aware of, is the problems of copying data simply from a running system.

Let's put down in simple terms what backup does; it's actually a batch operation execution at a specific point of time that copies updated data to other location that consists of total data and helps to identify success or failure of the operation by maintaining relevant information. While we do the batch operations, it depends on the filesystem and features provided by the application to determine whether applications can be running or not during batch operation to backup data.

In today's world, many enterprise filesystems support advance features such as decoupling data from time to time to a specified location, and to minimize the window of the required time feature, for example, snapshot is used to get reliable capture of the data lying on the filesystem:



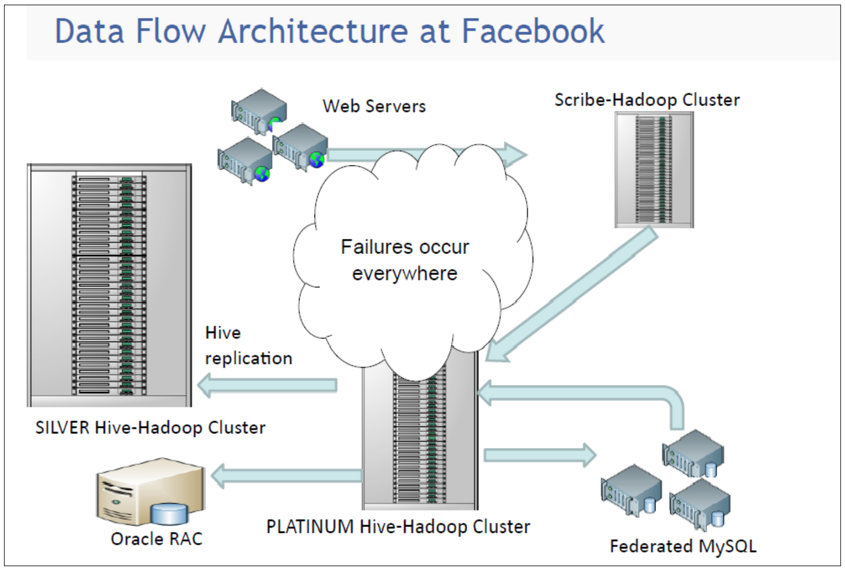
Normally, medium to large Hadoop cluster consists of a two- or three-level architecture built with rack-mounted servers as shown in the preceding image.

Hadoop has two primary approaches for backup:

- First is the distributed copy tool, commonly known in short as DistCp. The DistCp tool copies HDFS data, either in parallel to another location of the same cluster or it can also copy between clusters.
- Another approach for backup can be in architecture design to write data into different clusters from the application itself.

Having said that, we conclude that each of the approach has its pros and cons. This lends them to deal with various situations.

One question that always comes to mind is, where would failure occur in a Hadoop cluster? It would help to create backup strategy around probable failures. Take a look at the following figure where the Facebook team has explained data flow briefly, which is a good reference point to understand areas where failure might occur. I came across this presentation on the Facebook engineering blog for Facebook's data flow architecture; my eyes caught one of its slides where it says **Failures occur everywhere**:


Data Flow Architecture at Facebook
Web Servers
Scribe-Hadoop Cluster
Failures occur everywhere
Hive replication
SILVER Hive-Hadoop Cluster
Oracle RAC
PLATINUM Hive-Hadoop Cluster
Federated MySQL

As we have now come to know that when it comes to backup, there is no specific area to be protected and we have to look at all the angles for backup to ensure that we have business continuity with reliable data in place available all the time. Let's now understand in detail what the areas in Hadoop where failures, such as hardware failures, application failures, and primary site failures, occur.

Understanding the common failure types

Sometimes we don't even notice that the chance of facing data loss results from subsequent failures caused by a root cause such as the failed component. Most people don't even consider this until they have experienced it.

For instance, I worked on relational database that had two slaves and two disaster recovery sites replication being done. One of the queries that was supposed to be executed to query report was executed in such a way that it updated one of my core data table; no one noticed this until the whole system along with the two slaves and two **disaster recovery** (**DR**) sites became useless because of data corruption. This is when we came up with delay replication for our relation database replication.

It doesn't happen that frequently but once in a while if such a scenario happens, it ruins the data we would deal with. More frequent causes can be network switch failure, air conditioning failure, power failure, or equipment burns out. We will understand such common failure types in this section, based on which we would also learn what should be considered in the backup strategy of Hadoop.

We are going to talk about common failure types here, for example, a failed host. However, remember that we should not oversee scenarios that are not likely to happen frequently, especially when we are managing Hadoop clusters and its data on a larger scale.

Hardware failure

Hadoop's framework has been fundamentally designed with the assumption that hardware failures are common in real world scenario, hence they should be handled automatically within Hadoop's framework. This really helps us to get out of an overhead burden of one of the areas to be taken care of during failures, right? Before driving to conclusion, let's discuss in detail the consequences of hardware failures.

It is suggested to keep all Hadoop-compatible filesystems with location awareness for effective scheduling of work such as name of rack and, to be precise, name of the network switch where the worker node would be residing. Hadoop applications run on the node using this information about where the data is and if this fails, it would run on the same switch or rack that helps to reduce backbone traffic as well.

While replicating data, HDFS uses this method to segregate data on different racks with different copies. Hardware failure being handled automatically, which is considered in Hadoop's fundamental design, helps to achieve better uptime and reliability of data in events such as failure of switch or rack power outage. Data would be still readable in such events that have location awareness configured in Hadoop.

It is said that hardware failure is not an exception, it's a standard. When having large number of components playing a role in HDFS, there are obvious chances that some of the components are always non-functional. Therefore, quick and automatic recovery with detection of faults is core architectural goal of HDFS.

Hadoop detects DataNode failures with the help of timeouts and connection errors. If a timeout expires or connection established is broken, all the operations of read or write are switched to new source or destination nodes obtained from the NameNode.

Host failure

Failure of a host is one of the common failure types. It could have failed due to any of the common reasons, such as:

- Failed CPU
- Fans not working
- Power supply failure

The problems mentioned here can cause failures in processes running on the host of Hadoop. Host failure can also occur because of OS issues like kernel panic, I/O locks, failed patches, and so on, and can also result in a similar situation to be addressed by Hadoop.

We can expect Hadoop to take care of host failures, such as when the host is unreachable, there is OS crash or hardware failure, or even a reboot, as it is one of the core fundamentals behind the design of Hadoop's framework. When Hadoop hosts appear to be functioning properly there might be underlying problems, such as a bad sector on the hard disk, data corruption, or faulty memory.

Hadoop's status reports for corrupted blocks help to address such misleading problems. The following is an example of Hadoop status report; key parts to look for are the lines reporting on blocks, live nodes, and the last contact time for each node:

```
bash$ hadoop dfsadmin -report
Configured Capacity: 13723995724736 (12.28 TB)
Present Capacity: 13731765356416 (12.29 TB)
DFS Remaining: 4079794008277 (3.61 TB)
DFS Used: 9651980438139 (8.68 TB)
DFS Used%: 71.29%
Under replicated blocks: 19
Blocks with corrupt replicas: 35
Missing blocks: 0
-------------------------------------------------
Datanodes available: 1 (1 total, 0 dead)

Name: 10.20.20.25:50010
Decommission Status : Normal
Configured Capacity: 11546003135 (10.75 GB)
DFS Used: 182736656 (175.26 MB)
```

```
Non DFS Used: 7445455807 (6.53 GB)
DFS Remaining: 3914760672(3.55 GB)
DFS Used%: 1.68%
DFS Remaining%: 34.91%
Last contact: Sat May 15 12:39:15 IST 2014
```

Hosts tend to fail, but without having redundancy of host or components within host, any single component failure in host may risk the entire host or sometimes a cluster. Hadoop's framework deals with hardware failure such as switching to slave hosts without a downtime or sometimes even failures like corruption. However, failures occurring on master host or connectivity issue between master servers to majority of slave servers can also cause significant downtime.

Using commodity hardware

If we go back to our earlier days of distributed computing, hardware failures were not tolerated well. They were considered as an exception. Hadoop provides the ability to tackle hardware failures as it is designed to be fault tolerant. It was cumbersome to manage during the old days where such failures were handled using high quality components and by having backup systems in places.

Similar to what we had earlier for relational database, we had two slave and two DR sites in place, which become backup systems for us and even backups to the backup systems were implemented. So we tend to design our systems to overcome hardware failures, and keep the system up. Hadoop has the flexibility to run on commodity hardware. But the initial questions that come to our mind are: what about high availability? How would I manage my clusters if they fail? What if I need to move to a bigger cluster? That's right; we need to be prepared for this. How? Let's see.

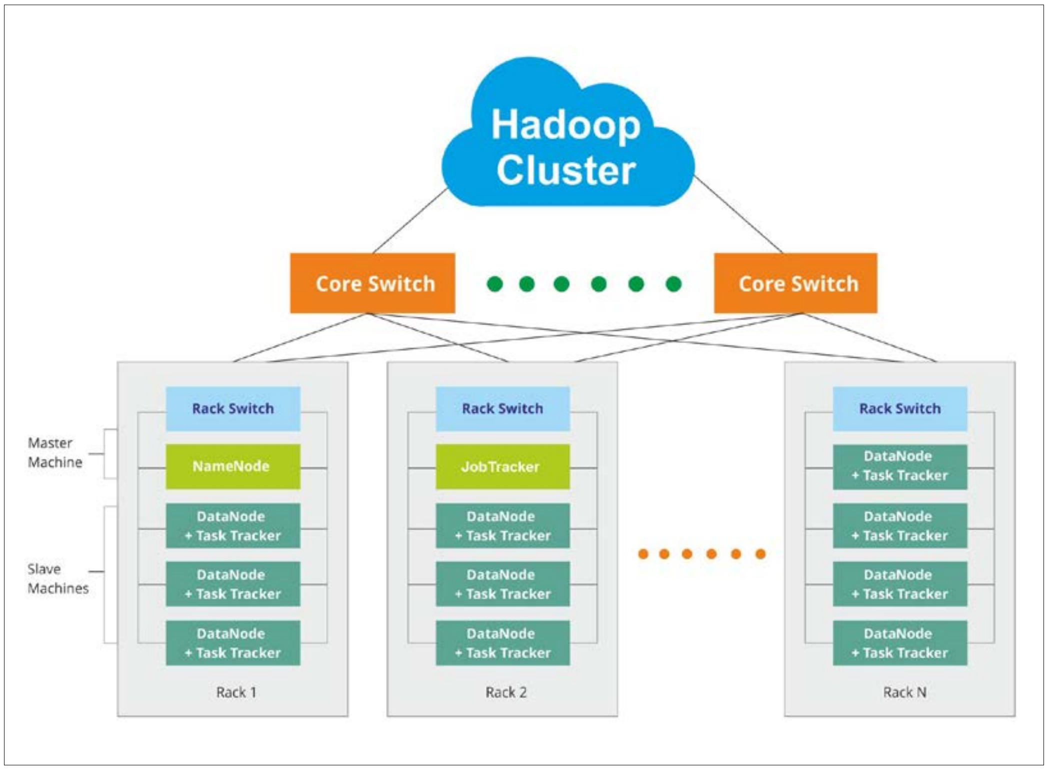
We have been relying on hardware components so far, let's change our mindset and make our application smart enough to detect hardware failure and rectify this automatically. This means no human intervention is required, right? Yes, that's correct.

Hadoop solution is intelligent enough to deal with hardware failure.

Hardware failures may lead to loss of data

As we have now understood that node failure is very much expected; having a bunch of machines to store data across the network and that data being spread across a board of nodes in the network, what would happen when a node fails. Data would be lost or would it be unavailable? We need to prevent this, but how?

One of the simplest approaches is to store data in different machines with multiple copies. In backup terminology, it's called replication. HDFS is strong enough to function even if one of your nodes fails. This is done by duplicating data across the nodes. In a scenario where data segment #2 is getting replicated three times on DataNodes A, B, and C, and DataNode A fails. In this case, data would still be accessible from nodes B and C. This makes HDFS robust for hardware failures. The following image will help us to understand in a much better way to approach this:



One of the key features of Hadoop clusters is to purchase quality commodity equipment. Most Hadoop buyers are cost conscious and as clusters grow, cost can be a major issue. We should think about the whole system, including network, power, and the extra components included in many high-end systems when we calculate cost. The following are a few of them:

- **CPU**: We would recommend using medium clock speed CPU and no more than two sockets. Most of the workloads and extra power per node are not cost-effective solution.

- **Power**: It is worth the effort to understand the usage of power the systems will occupy and not just buy fastest and biggest nodes available when we design Hadoop clusters. It's common nowadays that machines in data centers and cloud are designed in such a way that reduces power and cost, and excludes a lot of detail to ramp up traditional server. Power is considered to be a major factor when we design Hadoop clusters. It would be worth looking at cloud servers if we are buying large volumes.
- **RAM**: You need to consider the amount of RAM to keep the processors busy. Having around 36 GB of RAM is a good number, considering the commodity prices. Hadoop framework will have lots of RAM (~6 GB) and can still have enough room to run rest of the resources. Rest of the resources are easy to identify as you go along, such as running parallel processes and even caching data on disk that would help to improve performance. Use error-correcting SATA drives and RAM with good **mean time between failures** (**MTBF**) numbers.
- **Disk**: Running Hadoop without high-capacity drives is usually a no-no situation. Normally you get 24 or 36 TB/node in 12-drive systems. It is not practical to put this amount of storage in a node, because hard drive failures are a common scenario. In such large clusters, replicating this amount of data could bog down the network in such a way that it would disrupt availability of clusters and can cause huge delay in jobs. However, the good part is Hadoop, lately has been engineered to handle more smartly than ever. It would allow serving from the remaining available hard drives. Now we should expect to see a lot from 12+ drive systems. So the thumb rule is to add more disks for storage and not for seeks. We can drop off disk count to 4 or 6 per node in case the workload required to run doesn't need large amount of storage, this would surely help.
- **Network**: Hadoop workloads vary a lot and to have enough network capacity to run all nodes in clusters with reasonable speeds is a must. It is fine to have at least 1 GB bandwidth for smaller clusters, which nowadays is pretty easy by connecting all of them to good switches. However, large clusters such as those in data centers like Facebook or Yahoo! have around 2 * 10 GB per 20 node rack with rack nodes connected by two 1 GB links individually. Networking includes core switches, rack switches, network cards, cables, and so on. Having redundancy at each component needs to be in place. Key part is to understand your workload requirements to effectively plan network and its redundancy.

User application failure

In a real-world scenario, there are high chances of processes getting crashed, buggy code, and processes running slow either because of buggy code or server issues. Hadoop is capable of taking care of such scenarios and ensuring that tasks are completed.

Software causing task failure

It is said to be fairly rare to see Hadoop processes crash or fail unexpectedly. Hence, what we are more likely to see is failure caused by tasks where faults in either map or reducing tasks are being executed in the clusters. Let's have a look at such scenarios and how to handle them effectively.

Failure of slow-running tasks

Let's first look at what happens if task that is running appears to hang up or stops making development while running.

How Hadoop handles slow-running tasks

Hadoop performs a very well balanced act here. Hadoop wants to dismiss tasks that are running slowly or have got stuck. It might happen that for obvious reason tasks run slowly; one of the causes can be fetching information or relying on external resources for completing its execution.

In such cases, Hadoop looks for indication of progress from running tasks to decide how long it has been in hang/quiet/idle state. In general, it could be because of preparing results, writing values to the counters, or explicit reporting is in progress.

Hadoop provides the progressable interface, the method of which is stated in the following code snippet. This helps Hadoop to take care of identifying slow running tasks:

```
Public void progress();
```

Context class implements this interface, so to identify status of the process as running, the mapper or reducer can call the context to keep the process running.

Speculative execution

The MapReduce job, typically, will consist of many separate tasks that are running and task executions will be reduced.

When MapReduce jobs are run across clusters, there happens to be a big risk that a misconfigured or an unhealthy node will cause jobs to run significantly slower as compared to rest of the MapReduce jobs.

Hadoop has its sophisticated way to address this, it assigns duplicate maps or reduces tasks across the cluster when it is towards the end of a map or reduce phase. If a particular drive is taking a long time to complete a task, Hadoop will create a duplicate task on another disk. Disks that finish the task first are retained and disks that do not finish first are killed.

Hadoop's handling of failing tasks

A task doesn't simply hang; sometimes they'll throw exceptions, terminate, or stop executing silently than the ones mentioned previously.

Let's see how Hadoop responds to task failures, we need to set the following configuration properties in `mapred-site.xml` to take care of this:

- `mapred.map.max.attempts`: A given map task will be retried this many times before causing the job to fail
- `mapred.reduce.max.attempts`: A given reduce task will be retried these many times before causing the job to fail
- `mapred.max.tracker.failures`: The job will fail if this many individual task failures are recorded

[Default value for all the preceding properties is 4.]

It does not make sense to set value of `mapred.tracker.max.failures` lesser than the other two properties, as it would overrule rest of the two properties.

Setting numbers of these would depend on the nature of data we carry and the jobs running. If we have jobs that are fetched from external resources that may occasionally have temporary errors, in such scenarios increasing the number of repeat failures of a task can come in handy to take care of failed tasks. However, if the task is data sensitive, then putting low values for the maximum attempts would be beneficial as once the task fails, it would retry to do it again, which can be consuming unwanted resources. However, a value higher than 1 does makes sense when we have a large complex cluster environment where we do expect various types of temporary failures.

Task failure due to data

Last but not least, task failure can be because of data itself. This means tasks crash because the record provided while running the task had corrupted data, used wrong data format or types, or a wide variety of such problems occurring because of it. To keep it short, cases where the data received deviates from the expected one.

DataNodes does checksum as they are responsible for verifying the data they receive before storing the data. This applies to data that they receive from clients and from other DataNodes during replication process happening constantly.

When clients read data from DataNodes, checksums do get in the picture as well, comparing them with the ones stored at the DataNode. Each DataNode keeps a persistent log of checksum verifications, so it knows the last time each of its blocks was verified. When a client successfully verifies a block, it sends request to the DataNode to update the logs. Keeping all these statistics is valuable in detecting bad hard drives.

Data loss or corruption

For data loss or corruption, it is strongly suggested that the underlying disk storages being used are highly reliable such as RAID 1 or RAID 5 disk array, with hot spares for the NamesNode's filesystem image and edit log. Data loss or severe failures because of user errors cannot be considered fool-proof because they have such things in place.

The NameNode configuration will accept a comma-separated list of directories and will maintain copies of data in the full set of directories. Having extra precautionary measures of redundancy provides a current time backup of the filesystem metadata. Secondary NameNode provides a delayed replication of the NameNode data.

If one of the filesystem images or the edit log out of multiple directories is damaged then deleting that directory's content and restarting NameNode brings it back to the show. However, it is advisable to avoid restarting NameNode as the cluster wouldn't be operational during restarts.

If we notice that the directory itself is not available, it is suggested to remove it from the list in the configuration and restart the NameNode.

If all the directories are unavailable or have dubious existence, perform the following:

1. Archive the data first.
2. Wipe all of the directories listed in `dfs.name.dir`.
3. Copy the contents of `fs.checkpoint.dir` from the secondary NameNode to `fs.checkpoint.dir` on the primary NameNode machine.
4. Run the following NameNode command:

   ```
   hadoop namenode -importCheckpoint
   ```

If there is no good copy of the NameNode data, the secondary NameNode image may be imported. The secondary NameNode takes periodic snapshots at `fs.checkpoint.period` intervals, so it is not as current as the NameNode data.

You may simply copy the filesystem image from the secondary NameNode to a filesystem image directory on the NameNode, and then restart.

As the imported data is older than the current state of the HDFS filesystem, the NameNode may spend significant time in safe mode as it brings the HDFS block store into consistency with restored snapshot.

No live node contains block errors

Generally, if you notice that no live node contains block errors, it would reside in your log files for your applications. It means that HDFS was unable to find a block of a requested file from the client code in your application that interfaces with HDFS. It would specifically occur when the client code receives a list of the DataNodes and blocks ID pairs from the NameNode, but it is unable to retrieve the requested block from any of the DataNodes:

```
[admin@BD1-root]$ hadoop fs -cat output/d*/part-*
29/10/14 15:34:09 INFO hdfs.DFSClient: No node available for block: blk_-
5883967349607013522_1099 file=/user/gpadmin/output/d15795/part-00000
29/10/14 15:34:09 INFO hdfs.DFSClient: Could not obtain block blk_-
5883967349607013522_1099 from any node:  java.io.IOException: No live
nodes contain current block
```

This may be because there are no more file descriptors available, there is a DNS resolution failure, there is a network problem, or all of the DataNodes in question are actually unavailable.

Bad data handling – through code

It is suggested to write mappers and reducers for bad data handling that deals with data defensively. If a value received by the mapper is expected to be a comma-separated list of values, first validation of the number of items should be implemented before processing the data. If the first value should be a string representation of an integer, ensure that the conversion into a numerical type has strong error handling and default behavior.

But there will be always some unexpected data input pushed, no matter how careful we were while pushing it. We shall consider accepting values in a different Unicode character set, but what about null values, unwanted terminated strings, multiple character sets, and many more?

If the data input to your jobs is something you generate and/or control, these possibilities is of least concern. However, if you are processing data received from external sources, there will always be a room for surprises.

Hadoop's skip mode

One of the alternatives is to configure Hadoop to approach task failures differently. Before considering failed task as a major event, Hadoop can instead help to identify which data or records were the root cause of the task failure and exclude them from further task executions. This is also known as skip mode. This comes in handy when we are experiencing various types of data issues where fixing it by coding is not practical or necessary. It also helps when you are using third-party libraries within the job where you may not have the source code to fix for seamless execution.

Skip mode is currently available only for jobs written to the pre 0.20 version of API, which is another consideration.

Since Hadoop 0.22, secondary NameNode was replaced with two new components, BackupNode and CheckpointNode. The latter of these is effectively a renamed secondary NameNode; it is responsible for updating the `fsimage` file at regular checkpoints to decrease the NameNode start-up time.

The BackupNode, however, is a step closer to the goal of a fully functional hot backup for the NameNode. BackupNode constantly receives a stream of filesystem updates from the NameNode and keeps its in-memory state up to date at any given point of time, with the current state held in the master NameNode. In case NameNode crashes, the BackupNode is much more capable to itself act as a new NameNode. Unfortunately, this process isn't automatic; it needs to restart the cluster manually, but obviously would be of great relief in case of NameNode failure situations.

Handling skip mode in Hadoop

Just add the following property in `mapred-site.xml` to handle the skip mode. Hadoop will go in to skip mode if the same entity fails for the preconfigured value. The property is as follows:

```
mapred.skip.attempts.to.start.skipping=2
```

The default value here is `2` for the property.

What happens when we enter in to the skip mode?

We have the power to configure Hadoop when we enter in the skip mode. We can control whether to remove the entire record, or just the bad record. This will show how long Hadoop will proceed with the goal of narrowing the content.

Just add the following lines of code:

```
mapred.skip.map.max.skip.records=1
#only skip the bad record

mapred.skip.map.max.skip.records=0
#don't go into skip mode

mapred.skip.map.max.skip.records=Long.MAX_VALUE
#don't try to narrow
```

Learning a way to define the backup strategy

We have learned a lot of ways to understand causes of destruction in this chapter so far, and I hope you never have a day where you notice this much failure in a Hadoop cluster in a short time. But, in case you do, it will really help from the areas that we have covered so far.

Generally component failures are not that would bring you in to panic situations, especially when we have large clusters, component failures or host failures will be a routine job and for which Hadoop design is engineered to deal with such situations. HDFS comes with its responsibility to store the data managing replication of each dataset and replicating copies to be made when DataNode processes die.

Why do I need a strategy?

However, assuming that the concept of Hadoop will protect you from data loss, and therefore thinking that backing up of data is not essentially a big risk, though Hadoop replicating data by default to three times brings us in a safe situation. Yes, it protects from hardware failures on the DataNodes as it switches to either of the other nodes, but there are other causes of loss of data too that cannot be ignored.

Data loss can occur because of a human error, too. Some developer might create a Hive table with wrong location of the table that might cause all critical data that are lying to vanish. We should ensure that Hadoop's trash mechanism is configured, so such mistakes are detected quickly and rectified by recovering data and updating the developer's mistake. Hadoop trash is only supported by the command line tools. So, if APIs of Hadoop, such as Hue are used to delete, data wouldn't be recoverable from trash. In addition to this, trash stores data for the amount of time that is preset. In case such issues are not detected within the stipulated time, data would be deleted permanently and that calls for better data security, retention, and disaster recovery to be in place.

When it comes to data security, Hadoop follows the Unix POSIX model. Having said that, processes that load the critical data need to have privileges to write on it. Our intention is to minimize the risk factor because of user error that can cause loss of critical data.

Hadoop disaster recovery plan is one of the most complicated and expensive elements to protect data. Some of the best approaches have balanced cost, complexity, and recovery time while planning for disaster recovery. Having two Hadoop clusters seems to be excellent disaster recovery plan in Hadoop; we need to make sure that data insertion happens to both Hadoop clusters simultaneously.

We have come across implementations where data written on Hadoop is also inserted to other non-Hadoop locations. This allows us, if data in Hadoop is lost or corrupted, to load data from this other location into Hadoop.

Let's keep things ready before the catastrophe strikes.

There are several things to do in advance that would bring us in a better position when catastrophe strikes. We have listed down a few notes, which can help us to overcome disasters in Hadoop, such as data corruption, host corruption, rack failures, and so on:

- NameNode should be configured to write `fsimage` at multiple locations.
- In case of correlation failures with a cluster, you should know how quickly you can repair or replace the host failure. Identify hosts which will be the location of the interim NameNodes during failure.
- Before NameNode is identified, it needs to be ensured that the right hardware is in place that would take up the task of becoming a NameNode, the host is currently being used for DataNode and TaskTracker, and that the reduction in cluster performance due to the loss of these workers won't be too great.
- Core configuration files such as the `core-site.xml` and `hdfs-site.xml` files should be placed ideally in a shared location like NFS, and they should be updated to point to the new host. Making any changes to the configuration files should be replicated to rest of the copies as well.
- Make sure you keep files in the slave NameNodes updated and copy slave files from the NameNode into new hosts.

What should be considered in a strategy?

Backup strategy is essential, but what should be considered in it? With large sets of data volumes that Hadoop is capable of storing, however, to come up with where to back up its location becomes a challenging task. First thing to do is data prioritization, that is, data that cannot be regenerated and is critical should have higher priority than data that can be easy to regenerate or which has less impact and less priority. You can ignore to back up data that has least priority.

It is good to have a policy for user's directories in HDFS along with quotas that can be backed up on frequent basis. Make it transparent to users about the policy in place, so they know what is supposed to be expected out of it.

Filesystem check (fsck)

It is suggested to run the `fsck` tool frequently on the whole filesystem that becomes a major proactive action item to look out for missing or corrupted blocks.

The `fsck` tool helps to determine which files are having issues. When `fsck` is executed, we would get list of all files that are missing/corrupted/under-replicated blocks. We can ignore under-replicated blocks, as follows:

```
hadoop fsck / | egrep -v '^\.+$' | grep -v replica
```

> Running the `fsck` command on a large HDFS cluster is cumbersome as listing of every file on the cluster won't be of much help. To list only missing or corrupted files use the command.

After identifying the file that is corrupted, we should identify where the corrupted files are residing. It may be the case where files larger than standard size are divided in to blocks of different nodes. With the help of the following command, we can identify where the block resides:

```
hadoop fsck /path/to/corrupt/file -locations -blocks -files
```

As we have now identified required details for a corrupted file, let us nail down the problem on those nodes to bring it back on track and make it as healthy as it had earlier been.

Keep a note of following issues on these machines:

- Do we have filesystem errors?
- Are any mount points missing?
- Is DataNode running properly?
- Have the filesystems been reformatted or provisioned?

When you are done healing all unhealthy blocks and don't have further room to heal rest of the blocks, you can use the following command to have HDFS filesystem back in healthy state and start tracking few more new errors:

```
hadoop fs -rm /path/to/file/with/permanently/missing/blocks
```

Filesystem balancer

Run the balancer tool regularly to keep the filesystem DataNodes balanced evenly. Not aware of balancer tool? Let's first understand what the balancer tool does for us.

When size of a file increases it is generally distributed across various nodes. After certain period, the scattering of blocks over different DataNodes can become unbalanced. Now an unbalanced cluster can affect performance on MapReduce that will cause problems in highly available and utilized DataNodes. So it is better to avoid these situations and run the balancer tool on a timely basis.

But how does it actually work on Hadoop?

Whenever you start balancer tool (which is generally a Hadoop daemon) the following sequence occurs:

1. Redistribution of blocks by shifting them from excessively and over utilized DataNodes to underutilized and less utilized DataNodes.
2. Redistribution of blocks occurs according to the same rack awareness policy that we defined earlier in *Chapter 2*, *Understanding Hadoop Backup and Recovery Needs*. We ensured that the block is safe when something fails, be it a node or the entire rack.
3. It tries to balance the cluster, so it will carry on the redistribution process until it suffices the following condition. The difference between the space capacity of the cluster (which is defined as the ratio of used space on the node to the total capacity of the node) and the utilization of the cluster (which is defined as the ratio of used space on the cluster to the total capacity on the cluster) should no longer be more than a given threshold percentage in order for a improved performance.

You may specify the threshold argument if you want to define your own balance ratio using the following command where we have taken an argument of 25 percent:

```
$ hdfs balancer -threshold 25
```

The threshold argument generally specifies the percentage that defines what it means for the cluster to be steady defined. The default value for the flag is 10 percent.

The balancer runs until the cluster goes to a balanced state. Once the balance state is achieved, it cannot move further blocks or it loses contact with NameNode. It creates log file in the standard log directory where each and every block/file redistributed to a different node is recorded.

The process of balancing basically runs as a background process without consuming much of the cluster resources or hampering other clients that are using the cluster. So obviously it will limit a bandwidth to a very modest rate generally around 1 MBps. Depending on your cluster hardware, you can play with the speed by setting the `dfs.balance.banddwidthPerSec` property in `hdfs-site.xml`. By default the value is 1048576 bytes per second.

Upgrading your Hadoop cluster

Upgrading an HDFS and MapReduce cluster requires careful planning. If the layout version of the filesystem has changed, then HDFS upgrades will take care of migrating the filesystem data and metadata to a new version that is compatible to the new format.

Having considered the migration of data, there would be always chances of data loss, so a backup of your data along with metadata should be taken care. Before any upgrades, it is very much essential to have a trial run on a similar or smaller cluster with the actual data. It would help to target any errors while upgrading or any other problems before executing on the production cluster.

If filesystem layout has not changed, then upgrade is almost as straightforward as installing newer version of HDFS and MapReduce on the clusters and halting old daemons. Start new daemons along with updated configuration files after pointing to the client to use new libraries. What if it still fails? No issues. As this process is reversible, so rolling back after an upgrade is also straightforward.

Once successful upgrade is done and verified, don't forget to execute a couple of clean-up steps as follows:

1. Remove the old installation and configuration files from the cluster.
2. Fix any deprecation warnings in your code and configuration.

You can keep only the previous version of the filesystem; you can't roll back several versions. Therefore to carry out another upgrade to HDFS data and metadata, you will need to delete the previous version by a process called finalizing the upgrade. Once an upgrade is finalized, there is no procedure to roll back to a previous version.

Designing network layout and rack awareness

Surviving from entire rack failure or its components is essential while planning. Hence designing cluster's network topology is the key part while planning of HDFS clusters, as it also impacts performance and reliability. Having NameNode's backup server in a separate rack from the primary NameNode is must. This is a basic step to avoid failure of cluster when one of the racks is down.

Having more than one rack, managing the location of both replicas and tasks becomes complex. To reduce the impact of data loss, replicated blocks should be placed in separate racks. If client inserts data in Hadoop cluster, first replication is done on the DataNode where the client resides, otherwise replication is done randomly among the cluster. It is good to put other replicas on a random rack other than the rack where client resides in the DataNode.

We have to provide an executable script to Hadoop that would know the location of each node by mapping IP addresses into relevant rack names. This network topology script must reside in the master node and location must be specified in the `topology.script.file.name` property in `core-site.xml`. The `topology.script.number.args` property controls the maximum number of IP addresses that Hadoop will ask for at any time. It's convenient to simplify your script by setting that value to `1`.

Most important areas to consider while defining a backup strategy

Disaster recovery planning can be considered primarily no different from planning for IT disaster recovery. The criticality or benefit of Big Data business application is directly proportional to planning of big data disaster recovery. Let's go through the following considerations, which are the most important areas to drive the discussion while planning and designing cost effective disaster recovery solution:

- Determining the actual requirements.
- Choosing data processing and data analysis/visualization processing.
- Considering data sizing.
- Is the data large enough to process though Hadoop?
- What is the size of data you generate on a regular basis?

- What is the aggregate size of data per time?
- Are you moving traditional data storage/processing to Hadoop or starting new?
- What ability and capacity do you have to manage the data as it grows?
- Is the data already stored somewhere else?
- On-premise storage or on-cloud with a specific storage vendor like AWS/Google?
- Would you like to move your data to the current location or to a new location?
- Data storage process will start once the cluster is ready.
- Considering data type and processing time.
- File storage method. Would it be compressed / un-compressed?
- Is there any time requirement to process the data in a limited time such as output of one MapReduce job is input of second MapReduce job run every 30 minutes?
- Considering Hadoop cluster:
    - On-premises
    - Cloud
    - Third party hosting
- Do you have IT to manage the cluster?
- Are you a significantly small team with limited time and resources?
- What if you want to utilize the existing infrastructure?
- Hadoop distribution:
    - Build your own Hadoop
    - Choose one from the available Hadoop distribution
    - Cloudera, Hortonwork, MapReduce
    - What about cloud-based Hadoop?
    - Amazon EMR service
    - Hadoop and Azure
    - Others

- Consideration of using EMR:
    - Immediately available after 15-20 minutes of configuration.
    - No hardware of any type needed, just your machine.
    - Pay as you go, so you may end up paying a few dollars to run the POC.
    - If data is already in Amazon object storage (S3), EMR cloud can be a very good choice.
    - If result set is large and another application is using EMR, you need to copy it locally.
    - If you have to copy of large amount of data to Amazon EMR, then this may not be good option.
- Consideration of choosing data processing app:
    - Do you have in-house talent for data processing? Who is going to write MapReduce jobs?
    - Is your data transferred to Hive table processed though SQL?
    - Is your data transferred to pig tables for processing?
    - Are you going to hire an elite programmer to write MapReduce jobs?
    - Do you understand your data well?
    - What is your timeline to try and accept a POC?
    - Are you going to choose third-party vendor?
    - Do you have the security requirement for vendors?
    - What if the full application is already available?
    - Any limitation with the result processing such as time limit?
- Consideration of choosing BI vendor:
    - Do you already have a BI application that you would want to use with processed results?
    - Do you have a limitation within BI, related to data access and security?
    - Would you like to have a single vendor for data processing, data analysis, and visualization, or otherwise?
    - Financial and technical limitation in choosing a vendor.
    - Most of BI vendors access Hadoop data through connector, so make sure this is something acceptable for your requirement.

    - Only few vendors have BI application running natively on Hadoop that uses MapReduce framework for data processing, so make sure who you choose to meet your needs.

- Consideration of running Hadoop cluster successfully:
    - It is very important to run your Hadoop cluster properly to get ROI.
    - Various jobs need tweaking for best results and it takes good amount of time to find optimum settings.
    - Cluster setting can be done globally or specific to individual jobs.
    - Various applications submitting MapReduce jobs can use scheduler to scheduler jobs during a specific time.
    - Some jobs may require more resources than others, so such jobs need rescheduling according to cluster health.
    - Depending on resource utilization, it is best to schedule jobs at a time when resources are available.
    - Data security is very important so deploy appropriate security mechanism to meet your needs.
    - Test various third party apps properly, otherwise one bad application may make the overall cluster unstable.
    - If possible, deploy third-party application outside the NameNode and provide SSH access to Hadoop node, this way you can minimize additional stress on Hadoop cluster.
    - Install Hadoop monitoring application to keep monitoring Hadoop cluster health.
    - Keep checking your cluster for additional storage and other resource requirements.
    - If a particular node is weaker than other nodes, find a replacement otherwise it will slow down the other nodes.

While it is easier to set up and turn on your Hadoop cluster, the cost to keep Hadoop cluster up and running is considerable, so what you choose is very important. Each enterprise has unique requirements. Choose what is best for you, based on your needs and existing or new incoming data requirements.

Understanding the need for backing up Hive metadata

So far what we have understood is replication of data in Hadoop, however, we have yet not discussed how to protect Hadoop metadata including Hive schemas, NameNode files, and so on. To protect metadata is also equally important as protecting your raw data.

What is Hive?

As Hadoop is built to store large amount of data, Hadoop cluster consists of a variety of pools of data that have multiple formats and are pushed from various sources. Hive allows you to explore and structure this data, analyze it, and then turn it into business insight.

The Apache Hive data warehouse software enables us to query and manage large datasets in distributed storage. Hive tables are similar to the one in a relational database that are made up of partitions. We can use `HiveQL` to access data. Tables are serialized and have the corresponding `HDFS` directory within a database.

Hive replication

Hive replication helps to back up and synchronize Hive metastore along with the data in clusters. In Cloudera Manager Server, we can use the **Add Peer** link available on the **Replication** page. Once peer relationship setup is done, we can configure Hive metastore replication by carrying out the following steps:

1. Click on the **Services** tab, and go to Hive service where replication of data needs to be done.
2. Click on the **Replication** tab.
3. Select the Hive service as source of the data. The **Create Replication** screen will pop up.
4. Keep **Replicate All** checked for replication of all Hive metastore databases.
5. Select the destination target.
6. Select scheduler. We can have it immediately, once or recurring.
7. We should uncheck **Replicate HDFS Files**, which would replicate only Hive metadata not data files.
8. Click on **Save Schedule**.

Summary

In this chapter, we went through how we can define a backup strategy of Hadoop and the areas that are needed to be protected. We also understood what the common failure types are when we have Hadoop clusters. Then we looked at some of these to protect us from failures, starting with a component failure to even rack or primary site failure.

We learned one important aspect for defining a backup strategy is that failure occurs everywhere, so each and every component whether it is small or big, should be considered while defining the backup strategy. After we understood the common failure types, it was time to know how to define a backup strategy and the things that should be covered when we think of Hadoop for backup strategy. Having a big list confuses us, so we also learned what is most important to you when you create a new cluster.

Finally, after discussing so much about data we covered the important aspect of Hadoop that is its metadata. We understood what is Hive, how it works and its replication. It's time now to have look at how we back up and the different ways of backup in next chapter. Along with backup, we would also learn about HBase and its importance.

4
Backing Up Hadoop

In the previous chapters, we learnt about the importance of backup and recovery in Hadoop. We have also discussed various areas that we need to consider for the backup—dataset, application, and configuration. In *Chapter 3*, *Determining Backup Strategies*, you learnt about common failure points in Hadoop, such as hardware failure and user application failure. We also learnt a way to define a backup strategy considering the most important needs of our application.

Now, in this chapter, we will discuss the different ways of taking backup in Hadoop. We will be introduced to HBase and learn about the importance of HBase. There are various ways to take backup in HBase, namely:

- Snapshot
- Distributed copy
- Replication
- Export
- Copy table
- HTable API
- Offline or manual backup

We will discuss all the backup styles and will compare all the styles in this chapter.

Data backup in Hadoop

As we have been discussing in this book, Hadoop is used to store a humongous amount of data. The data size would easily go into petabytes. Just consider Facebook. It stores the data of millions of users. It stores not only the status updates or comments that users post, but also the pictures, videos, and other large files. When we have this huge amount of data, one obvious question comes into mind, "how can we back up all the data?" This is one of the important areas to take care of when working with such large amounts of data.

Let's take an example of Facebook, where this huge amount of data is constantly being pushed to the servers by various users. Solving a simple question such as knowing what has changed since the last backup could be very difficult if it is not taken care of properly, particularly when there is a large cluster implemented.

Also, when we are selecting the data to be backed up, it is very important to consider the following points:

- Selecting a required dataset for the backup. If there is any set of data within the dataset that has not been backed up.
- Determining the frequency of the backup. It is easy to decide to take backup less frequently. However, this can increase the chances of data loss. On the other hand, it would not be practical to take backup very frequently as it increases the overhead on the application and impacts the application performance. So, it is very important to select the right frequency of backup. Also, once the data grows beyond a certain limit, we may have to determine a data purging strategy to limit the storage cost as well.
- As we discussed, in the case of Facebook, the data flows very fast and changes very frequently. Data consistency will be an important factor in this scenario. If we try to copy the data when it is being changed, it can result in invalid backup.

So, it is very important to know about the application that we are trying to back up the data for. If we see entire backup process from a layman's perspective, it is about executing a batch operation at a certain point in time, which copies a dataset and places it into a secondary location and provides feedback—success or failure. However, it is important to have your application up and running during this process.

There are two approaches for backing up Hadoop:

- The first approach is by using the **distributed copy** (**DistCp**) tool. It is used for copying data within the same cluster or between two different clusters. DistCp uses MapReduce for distribution, error handling and recovery, and reporting.
- Copying the incoming data to two different clusters. This is an architectural approach where we create two different clusters. One cluster will be used as the primary cluster, and the other will be used as the secondary cluster. Whenever there is any incoming data, it will also be written to the secondary cluster once the writing data operation on the primary cluster completes.

Now, let's understand both the approaches in detail.

Distributed copy

As discussed in the previous section, DistCp uses MapReduce to copy data from one cluster to another cluster. In this section, we will understand the basics of distributed copy and some of the options available to perform this task. In Apache Hadoop 1.x and CDH3, `distcp` is a subcommand of the `hadoop` command, whereas later versions use it as a subcommand of `mapred`.

The distributed copy command (`distcp`) is very simple. We have to define the command along with the option, source URL, and destination URL, as follows:

```
hadoop distcp hdfs://node1:8020/src/node \ hdfs://node2:8020/dest/node
```

This command will expand the namespace under `/src/node` on Node 1 under a temporary file, partition its contents among a set of map tasks, and start a copy on each TaskTracker from Node 1 to Node 2. Notice that DistCp requires the absolute path.

It is also possible to have multiple source folders in distributed copy. See the following example:

```
hadoop distcp hdfs://node1:8020/src/d1 \
hdfs://node1:8020/src/d2 \
hdfs://node2.8020dest/node
```

In this example, if there is any collision in the two defined sources, DistCp will abort the copy operation with an error message. However, if there is any collision at the destination, it will be resolved on the basis of the options that we specify in the command. We will see the options later in this section. By default, all the existing files will be skipped. The user will get a report of the skipped files at the end of each job.

The following are the different options available while using the `distcp` command:

Option	Description
`-p [rbugp]`	Preserve status: • `r`: Replication number • `b`: Block size • `u`: User • `g`: Group • `p`: Permission
`-i`	Ignore failures
`-log <logdir>`	Writes log to `<logdir>`

Option	**Description**
`-m <num_maps>`	Maximum number of simultaneous copies
`-overwrite`	Overwrite destination
`-update`	Overwrite if the `src` size is different from the `dest` size
`-f <urilist_uri>`	Use `list <urilist_uri>` as the `src` list

If we see the last option, it talks about using the list as a URI list. The following is an example for this:

```
hadoop distcp -f hdfs://node1:8020/srclist \
hdfs://node2:8020/dest/node
```

Here, the `srclist` contains the following:

```
hdfs://node1:8020/src/d1
hdfs://node1:8020/src/d2
```

Using `distcp`, you can copy data in one of the following manner:

- Between two different clusters
- Within the same clusters but under a different path

If we are using the `hdfs://` schema to connect to clusters, all the clusters must use the same version of Hadoop. If not, we have to use `webhdfs://`, which uses an HTTP-based, version-agnostic protocol to transfer data. However, the use of `webhdfs://` may decrease the speed of your data transfer. We need a certain configuration to use `webhdfs://`.

Architectural approach to backup

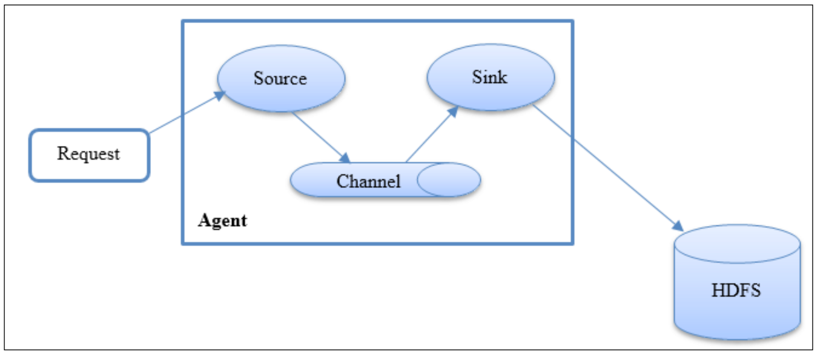
In the previous section, we have seen an approach to take backup explicitly. This is a traditional approach where one specifies the backup interval and then takes the backup by defining the data to be backed up. This approach has its own benefits and drawbacks. In this model, one master storage contains all the data, and then a batch process is run to move the data to the slave storage. This is an approach that any developer would love as the backup is outside of the development domain and the developer doesn't have to think a lot about it. However, the same scenario would cause a lot of questions for the DB support staff as they will be thinking about data consistency, right frequency to take the backup, and so on.

So another approach to take the backup is to take care of it at the application level itself. At this level, consistency requirements are known. By taking care of the backup at an application level, we are bringing the issue to everyone's notice, including the development team and an admin, and will deal with it more explicitly.

Here, we are talking about an application taking care of inserting the data into the two different clusters (or locations) defined during the architecture design. This will eliminate the need to take the backup manually at certain intervals. It is very easy to achieve this by having two different write operations to the defined clusters. However, we can also have an even more effective setup to achieve this.

The approach could be a service-based approach, where we have a service layer responsible for writing the data to two different clusters. Here, the application will only interface with the service layer and once the data is placed on the layer (in a queue-like fashion), the service layer will take care of writing the data to both the clusters using an event. This could be an asynchronous operation that will prevent from the event of blocking. This approach will decouple our application from the data write operations. We can also have a ping-back mechanism where the service will notify the application about the success or failure of writing the data to the clusters. However, this feedback mechanism may come at the cost of the speed or performance.

If you are a seasoned programmer or an architect, it would be easy for you to build this kind of system from scratch. However, Apache Flume in Hadoop ecosystem also provides exactly the same functionality. Flume is a distributed, reliable, and available service for collecting, aggregating, and moving a large amount of data. We can configure Flume to listen to multiple data sources like writing received events to multiple channels. The following simple diagram shows the mechanism of Flume:



Here, Flume considers each request to be a web server request. Once it receives a request, it is sent to a channel. Channels are nothing but queues that store data reliably before we write it to the disk. Sink takes the events from channel and then performs the required operation, for example, writing data to HDFS. Flume has the ability to send the duplicate events to separate channels. Here, each channel may be responsible for writing to a separate cluster. This can help us take care of the backup problem that we have been discussing. Flume is more suitable for writing log files or syslog events. However, when we have to upload the plain files, we can still use the parallel writing technique that we discussed at the beginning of this section.

When we use these techniques, we only have to worry about backing up the data generated as a result of the MapReduce job. This can be backed up by using a traditional method, such as using distributed copy.

Using this technique has its own benefits, such as always getting the fresh data, reducing the time window between backup operations. However, it also has downsides. This entire process needs to be taken care of at the application level, which can increase the development time and cost. Further, it may need additional hardware for setting up the queuing mechanism or Flume.

HBase

So far in this chapter, we have discussed the various ways to back up data in Hadoop. Now, we will discuss HBase and various ways to back up data in HBase. So, let's start with an introduction to HBase.

HBase history

The HBase project was begun by Powerset to process massive amounts of data for natural language search. It was started after Google published a paper on Bigtable (`http://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf`) in the year 2006. HBase development started as an open source implementation of Google's Bigtable at the end of the year 2006. It was developed by keeping the following aspects in mind:

- Lots of semi-structured data
- Commodity hardware
- Horizontal scalability
- Tight interaction with MapReduce

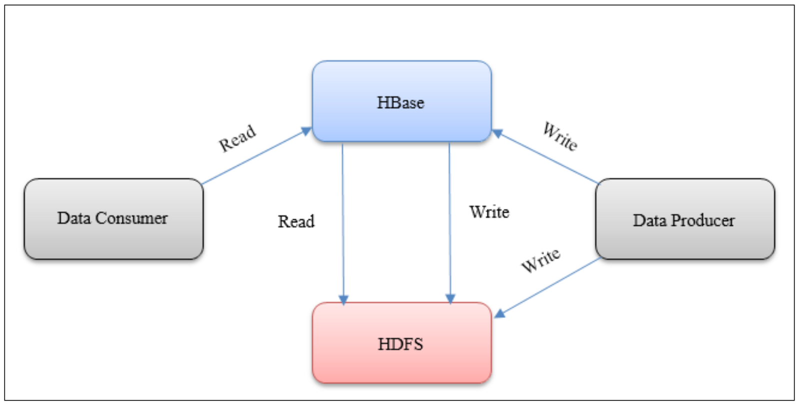
HBase was developed as part of the Hadoop project and runs on top of HDFS. The first usable version of HBase was released at the end of 2007. In 2008, Hadoop became a top-level project in Apache, and HBase became its subproject.

HBase introduction

Apache HBase is an open source distributed sorted map and non-relational database that runs on top of HDFS and is written in Java. It is column-oriented and multi-dimensional, and provides high availability and high performance. HBase provides fault-tolerant storage and quick access to a huge amount of sparse data. Let's understand some of the very common keywords associated with the HBase definition:

- **Sorted map**: HBase maintains maps of key-value pairs. This mapping is also known as cells. We can use a key to find a value in HBase. Here, each cell is sorted by key. This is useful because we can conduct a search based on the key but also search values between two keys.
- **Multidimensional**: In HBase, each key has a structure. Each key has a row key, column family, and timestamp. So, the actual key-value pair mapping in HBase will be like (row key, column family, column, timestamp) value.
- **Sparse data**: It means a small amount of information caught within a large collection of unimportant data.
- **Distributed**: The data in HBase can be spread over multiple machines (as it runs on HDFS) and reach millions of cells.
- **Consistent**: HBase ensures that all the changes within the same row key are atomic and a user always gets the last written or committed values.

HBase adds transactional capabilities to Hadoop. It allows users to conduct CRUD operations in the database. It provides real-time random read and write access to the data stored in HDFS. See the following image for more understanding:



HBase is good at serving a large amount of data. It should be used when fast, real time, and random access to data is required. You should use HBase when you have a large amount of unstructured data and want to have high scalability. It is also a good solution when you have lots of version data and you want to store all of them.

On the other hand, it would not be a good choice if you have only a few thousand or hundred thousand records. Further, if your requirements include the use of RDBMS, you should not use HBase.

Some of the main features of HBase are as follows:

- Linear and modular scalability
- Fault-tolerant storage for a huge amount of data
- Flexible data model
- Support for atomic operations
- Atomic sharding
- Easy-to-use Java API for client access
- Replication across the data center
- Strictly consistent read and writes
- Block cache and bloom filters for real-time queries

We have been mentioning that HBase runs on top of HDFS. So, a very common question may arise, "what is the difference between HBase and HDFS?" Let's understand this by looking at the following table:

HDFS	**HBase**
HDFS is a distributed file system well suited for the storage of large files. It is not a general-purpose filesystem and does not provide fast individual record lookups in files.	HBase is built on top of HDFS and provides fast record lookups for large tables. It internally puts the data in indexed StoreFiles available in HDFS for fast lookups.
It is suitable for high latency operations batch processing.	It is built for low latency operations.
The data here is mainly accessed through MapReduce.	It provides access to single rows from billions of records.
It has been designed for batch processing, so it does not have the concept of random read and writes.	It supports random read and writes.

Understanding the HBase data model

We have understood the basics of HBase and the scenarios when we should use it. Now, in this section, we will discuss the data model of HBase. It will help us to understand the various components involved in the HBase data model, the structure of how data is stored in HBase, and a way to retrieve the data from HBase. So, first, let's begin with the data model of HBase.

As discussed in the previous section, HBase is multidimensional and has various components like `rowkey`, `Column Family`, `Column Qualifier`, `Timestamp`, and `Value`. The following image illustrates each component. We will learn about all these components in this section:

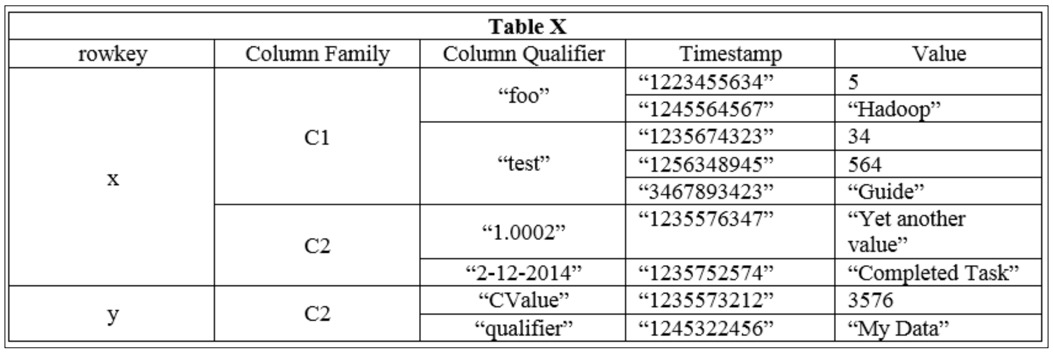


Table X

rowkey	Column Family	Column Qualifier	Timestamp	Value
x	C1	“foo”	“1223455634”	5
			“1245564567”	“Hadoop”
		“test”	“1235674323”	34
			“1256348945”	564
			“3467893423”	“Guide”
	C2	“1.0002”	“1235576347”	“Yet another value”
		“2-12-2014”	“1235752574”	“Completed Task”
y	C2	“CValue”	“1235573212”	3576
		“qualifier”	“1245322456”	“My Data”

The HBase data model has been designed to store structured and semi-structured data. As you can see in the previous table, the data varies in type, size of the fields, and the number of columns. Let's understand different logical components of the HBase data model:

- **Tables**: All the data in HBase is stored in tables. The tables behave like a logical collection of rows. You can give a string value to define a table name.
- **Rows**: Data is stored in a row within a table. Each row has a unique identifier called row key. Row keys do not have a data type in HBase. They are treated as a byte array.
- **Column family**: In HBase, columns are grouped into column families. All the column members of a column family have the same prefix, for example, in our example both columns `c1:foo` and `c1:test` are members of column family `c1`. All the columns in a column family are stored in HFile. It is required to declare the column families upfront in the schema definition. However, the columns can be defined on the fly. We use string to define a column family.

- **Column qualifier**: In each column family, data is addressed in a column identifier. It is also known as a column. Unlike `Column Family`, we do not need to define a column qualifier in advance. Columns do not have a data type and are treated as a byte array.
- **Cell**: A cell in HBase is a combination of row key, column family, column, and timestamp. It also works as a key in the HBase table. The data stored for each cell is considered a value of the key. The cell is also treated as a byte array.
- **Timestamp**: It is also known as a version in HBase. Whenever any value is written in HBase, it is stored with the version number. By default, the version number is the timestamp of when the cell was written. If we do not specify any timestamp while storing the value, it uses the current timestamp. We can store multiple versions of data by using this facility. If we do not specify any version at the time of retrieval, it returns the latest value.

To put the data model discussion in a nutshell, remember the following: HBase is a column-oriented database. The data is stored in tables, and the primary key is row key. Each row in HBase can have many columns. A cell consists of row key, column family, column, and timestamp. Each value in the HBase table has a timestamp.

Accessing HBase data

There are four primary data operations that we can execute to access data in HBase—`Get`, `Put`, `Scan`, and `Delete`:

- `Get`: This returns an attribute for a specified row.
- `Put`: This is used to either add a new row in the table or update an existing row.
- `Scan`: This allows you to iterate over multiple rows for a defined attribute. Using scan, you can match columns with filters and other attributes, such as the start and end time.
- `Delete`: This is used to remove a row from a table.

Now, we have understood about HBase and the various operations we can perform using HBase. As discussed, HBase is used when there is a huge amount of data involved in an existing cluster and we need fast random reads and writes. Now, let's go back to our initial discussion about various ways of backing up the data and how efficiently we can back up the data, particularly when there is huge dataset involved. In the next section, we will talk about different approaches to backing up data in HBase.

Approaches to backing up HBase

In today's world, many enterprises have adopted HBase in their critical business applications; some of the examples are Facebook, Twitter, Yahoo!, Mozilla, and Adobe. These enterprises are dealing with a large amount of data, and it is a prime need for them to protect the data by defining a robust backup and disaster recovery strategy for the data stored in their HBase cluster. It may seem very difficult to take a backup of petabytes of data, but the HBase and Hadoop ecosystem provide a mechanism to accomplish the backup and restore.

In this section, we will learn about various mechanisms available for backing up the data stored in HBase. We will be discussing the following approaches for the HBase backup:

- Snapshots
- Replication
- Export
- Copy table
- HTable API
- Offline backup

Snapshots

HBase snapshot is one of the simpler methods of taking HBase backup. It requires no downtime and leaves the smallest data footprint. Snapshot is simply a point-in-time image of your table or file system. It creates an equivalent of Unix hard links to your table's storage files in HDFS. The snapshot completes in a fraction of a second and leaves no performance overhead on the system. It is a set of metadata information that allows an admin to go back to a previous version of the table. Snapshot does not take a copy of the table; it only keeps the names of the files. No data is being copied while taking snapshots.

If you restore to the snapshot, you will be going back to the state when you took the snapshot. Any data added or updated after taking the snapshot will be lost if you restore back to the stage when you took the snapshot.

Operations involved in snapshots

The following operations are involved in taking a snapshot:

1. **Take a snapshot**: Take a snapshot of a specified table.
2. **Clone a snapshot**: It creates a new table using the same schema and with the same data present at the time of taking the snapshot.

3. **Restore a snapshot**: It brings the table schema and data back to the state when the snapshot was taken. If you have made any changes in the table or data since the snapshot was taken, all those changes will be discarded.
4. **Delete a snapshot**: It removes the snapshot from the system.
5. **Export a snapshot**: It copies the snapshot data to another cluster.

The following are two ways to take a snapshot:

- **Offline snapshot**: This is the simplest way to take a snapshot. While taking a snapshot, a table is disabled. Once a table is disabled, we cannot read or write data from or to the table. We have to simply gather the metadata of the table and store it.
- **Online snapshot**: In this case, the tables are not disabled. Tables are enabled, and each region server is handling a `put` and `get` request. Here, a master receives a request for a snapshot and it asks each region server to take a snapshot of the region that it is responsible for.

In case of an online snapshot, the communication between the master and the region servers takes place via ZooKeeper. The master creates a `znode`, meaning it prepares the snapshot. Each region server will process the request and then prepare the snapshot for the table that it is responsible for. Once they have prepared a snapshot for the table, they add a subnode to `znode`, notifying that they have completed the request.

Once all the region servers have reported back their status, the master creates another node to notify **Commit Snapshot**. Each region server commits the snapshot, and then, the master marks it as complete.

Snapshot operation commands

The following are the different operations involved in the snapshot process. All the operations are explained with the respective commands:

- **Create snapshot**: Creating a snapshot of a table is very simple. It can be done using the following command:

  ```
  hbase (main):001:0> snapshot 'TableName', 'SnapshotName'
  ```

  Once you execute this command, it will create small files in the HDFS, which contains necessary information to restore your snapshot. Again, restoring the snapshot is very simple and we can do so by the following steps:

  - Disable table

    - Restore snapshot
    - Enable table

  As you can see in the following operations, we have to disable a table while restoring a snapshot:

  ```
  hbase (main):002:0> disable 'TableName'
  hbase (main):003:0> restore_snapshot 'SnapshotName'
  hbase (main):004:0> enable 'TableName'
  ```

  It will cause a small outage while restoring the snapshot.

- **Remove snapshot**: To remove a snapshot, we need to run the following command:

  ```
  hbase (main):005:0> delete_snapshot 'SnapshotName'
  ```

- **Clone snapshot**: To create a snapshot from a specified snapshot, we need to use the `clone_snapshot` command. In this command, data is not copied, so we don't have to utilize double space for the same set of data:

  ```
  hbase (main):006:0> clone_snapshot 'SnapshotName', 'NewTableName'
  ```

- **Export snapshot**: To export an existing snapshot to another cluster, we use the `ExportSnapshot` command. This works at the HDFS level and we have to define an HDFS location:

  ```
  hbase org.apache.hadoop.hbase.snapshot.ExportSnapshot -snapshot
  SnapshotName -copy-to hdfs://node1:8082/hbase
  ```

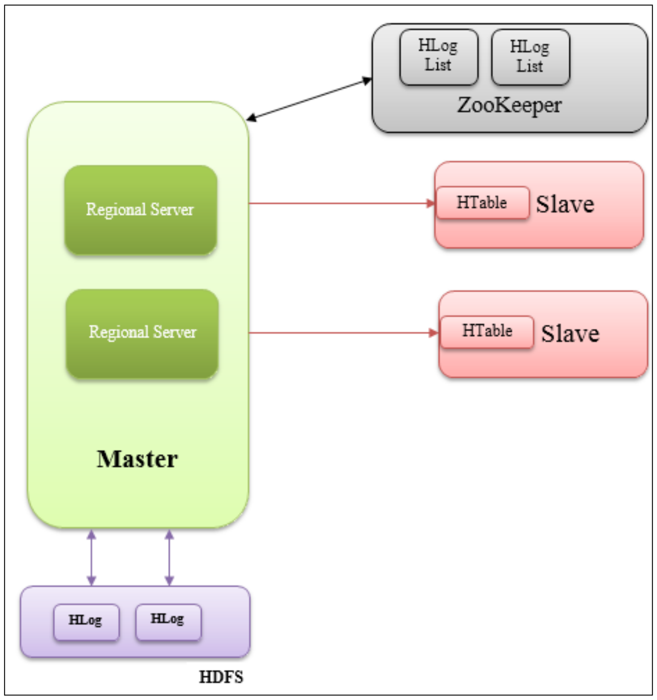
The snapshots create a full image of the table all the time. No incremental snapshot functionality is available at this moment.

HBase replication

HBase replication is also a low overhead backup tool. Using HBase replication, we can copy data from one HBase cluster to a different HBase cluster. Here, transactions from one cluster are pushed into the second cluster. The originating cluster is called the master, and the receiving cluster is called the slave.

All the transactions pushed by the servers are asynchronous. The clusters can be geographically distant. The data inserted in the master will not be immediately synced with the slave, so the data won't be immediately consistent between the master and the slave. Having asynchronous transactions ensures minimal overhead on the master. Also, all the transactions are shipped in batches, which will ensure higher throughput.

The replication has been shown in the next image:



HBase uses a **Write Ahead Log** (**WAL**) per region server to ensure that the data is not lost before it is synced to. The WAL is not there to read or write. It is used in case the regional server crashes before its in-memory state (MemStore) is written to HFiles. In case of a server crash, WAL is replayed by a log splitting process to restore data in WALs.

As you see, HBase also uses ZooKeeper in the replication process. HBase replication maintains its state in ZooKeeper. ZooKeeper is used to manage major replication-related activities like slave cluster registration and handling the failover of regional servers. ZooKeeper has two child nodes—Peers znode and RS znode:

- **Peers znode**: This contains a list of all the peer replication clusters and the status of all the clusters.
- **RS znode**: This contains a list of WAL logs that need to be replicated.

HBase replication supports replication across data centers. It is used in scenarios like Disaster Recovery. Here, the slave cluster will serve real-time traffic if the master cluster is down. Since the slave node has all the data received from the master, it can serve all the user requests in case of a master cluster failure. Here, HBase replication doesn't take care of automatic switching from the master cluster to the slave in case of a master cluster failure. It has to be taken care of by an individual user or the application.

Modes of replication

There are three types of replication modes in HBase. They are as follows:

- **Master-slave replication**: This replication is similar to any master-slave setup we see in other applications. There is a master cluster that pushes all the transactions to the slave cluster. The slave cluster receives the transactions and stores them along with the other tables or data that it has. Here, the communication flow is unidirectional, that is, from the master cluster to the slave cluster.
- **Master-master replication**: This is also similar to the one mentioned here with a slight difference. In this replication type, both the clusters can behave like master clusters and slave clusters. The communication flow in this type of replication is bidirectional. Here, both the clusters are assigned a cluster ID, which is used to avoid any conflicts in the communication.
- **Cyclic**: The cyclic replication is a combination of master-slave and master-master replication. Here, we have more than two nodes in which two nodes can work as master-master and the other two nodes can work as master-slave. This set up also uses the cluster ID for communication.

Export

HBase has a utility called Export, which is used to export the data of the HBase table to plain sequence files in the HDFS folder. When we use the `Export` command to export data, it first creates a MapReduce job, which will make a call to the HBase cluster. Each call will get data from each row of the specified table, and then write the data into the HDFS folder that we have specified.

The following is a command to export the HBase table data to the local filesystem:

```
hbase org.apache.hadoop.hbase.mapreduce.Export "Table Name" "/Local/
Directory/Path"
```

If we want to copy the HBase table data into HDFS, we can use the following command:

```
hbase org.apache.hadoop.hbase.mapreduce.Export "Table Name" "hdfs://
nameNode/path"
```

> Please note that this approach will need a reliable and fast network connection to the remote cluster.

Once the data is exported, it can be copied to any other location. We can also import from a local dump to an existing HBase table by using the following command:

```
hbase org.apache.hadoop.hbase.mapreduce.import "Table Name" "/Local/
Directory/Path"
```

The `Export` process will use MapReduce and then HBase client APIs, so there could be a performance impact on the application. However, `Export` provides a feature to filter the data by version or date range. It can be useful to take incremental backups, which is usually not possible with `CopyDisk` or HBase replication. We can use the following command to apply filters in `Export`:

```
hbase org.apache.hadoop.hbase.mapreduce.Export "Table Name"
"/Local/Directory/Path" [<versions> [<starttime> <endtime>]]
```

The copy table

The copy table is an option that is very similar to the `Export` option that we discussed in the previous section. It also creates a MapReduce job from an HBase client API that reads the data from a source table. However, the major difference is that Export writes the data to an existing folder, whereas the copy table will directly write the data into an existing table in HBase. The copy table can copy a part of the table or an entire table to a table in the same cluster or a remote cluster.

> The copy table functionality copies each table data row by row and writes into the destination table in the same manner. This can cause a significant performance overhead. If your table is very large, it can cause the memory store of the destination region server to fill up very quickly.

Whenever we use a copy table, we need to ensure that both the clusters are online when we are working on a multi-cluster setup and the target table has to be present.

The following is the syntax for the copy table functionality:

```
CopyTable [general options] [--starttime=X] [--endtime=Y] [--new.
name=NEW] [--peer.adr=ADR] <tablename>
```

The options for the copy table are as follows:

Option	Description
`rs.class`	This indicates the `hbase.regionserver.class` of the peer cluster. It needs to be used if we are copying the table to a cluster, which is different from the current cluster.
`rs.impl`	This is the `hbase.regionserver.impl` of the peer cluster.
`Startrow`	Start row from which we need to start the copy.
`stoprow`	Stop row where we need to stop the copy.
`starttime`	Beginning of the time range. If we don't give the time range, it will consider the start time to be forever.
`endtime`	End time. It needs not to be specified if we haven't mentioned the start time.
`versions`	Number of cell versions to copy.
`new.name`	Name of the destination table.
`peer.adr`	Address of the peer cluster.
`Tablename`	Name of the table to be copied.

To copy an `srcTable` into `destTable` in the same cluster, we can use the following:

```
hbase org.apache.hadoop.hbase.mapreduce.CopyTable --new.name=destTable
srcTable
```

As is evident from the narration and the options available in the copy table, it can be used in the following scenarios:

- Creating an internal copy of a table. It would be similar to a raw snapshot mechanism where we can create a physical copy of a certain table.
- In case of taking a backup of a remote HBase cluster instance.
- Creating incremental HBase table copies. We can specify the start time and the end time when using the copy table to take incremental backup.
- Copying data from specific column families.

[As discussed earlier, the copy table uses MapReduce over HBase. It also has performance overheads due to the way it copies the table. So, this approach may not be suitable if the dataset to be copied is very large.]

HTable API

All the options that we have mentioned so far are available as HBase tools for the backup HBase data. However, just like any other application, you can also develop your own custom application using an HTable public API and the database queries. You can find more information about the HTable API at `http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/HTable.html`.

MapReduce also provides batch processing in the application framework. We can write a MapReduce job to take advantage of this feature.

You need to have a sound understanding of Hadoop development in order to use this option for Backup. In case it is not used in a proper way, it can cause serious performance issues in the application.

Offline backup

This is a most crude way to take a backup of HBase. Here, we cleanly shut down the HBase cluster and manually copy all the data and folders available in `/hbase` into our HDFS cluster. Since the HBase cluster is down, we will always get the most accurate copy of the data as no write would be possible at this time. However, this is one of the most troublesome methods of taking backup as it would not be possible to take incremental backups in this scenario. Thus, this is not an advisable approach to take backup.

Comparing backup options

So far in this chapter, we have discussed about all the backup options and the pros and cons of each option. Now, it is time to compare all these options from various aspects.

Aspects	**Snapshot**	**Replication**	**Export**	**Copy Table**	**HBase API**	**Offline/ Manual**
Performance impact	Less	Less	High	High	Medium	Depending on code
Data footprint	Smallest	Big	Big	Big	Big	Big
Cluster downtime	Less	None	None	None	None	High
Incremental backup	Not possible	Basic	Yes	Yes	Yes	Not possible
Implementation complexity	Easy	Medium	Easy	Easy	Complex	Medium

On the basis of our needs, we can choose the right backup option:

- Snapshot should be used when we have a requirement to keep an offsite backup of our data. It has the minimum performance impact and the smallest data footprint. However, if we have a requirement of taking incremental backup of data, then snapshot would not be the right choice.
- If we are looking at a robust, fault-tolerant option for data backup, then replication would be an ideal candidate. However, it does not provide just-in-time data consistency between the master and the slave cluster data. So, if we need a just-in-time data update in the master and the slave, then we should avoid using replication.
- If we are looking at a backup option that can provide accurate incremental backup and is easy to implement, then `Export` would be a good choice. However, it has a performance impact as it uses MapReduce and HBase client APIs. Further, it would not be an ideal option if we want data to be directly copied into a remote cluster table.
- If we need to copy the table data into another table of the same cluster or a remote cluster, then copy table is the right fit. It copies all the table data into the remote table. Since this option reads individual rows and writes into the destination table, it may not be suitable for a situation where we need to move a high volume of data or large datasets.
- If you have very good knowledge of the Hadoop API, then you can choose to use the HBase API to write your own backup process. It gives you the flexibility and freedom to choose your own implementation. However, it may impact the performance if you do not follow an optimal design or the best coding practices. We should avoid using this option if we are not familiar with the Hadoop API.
- Offline/manual backup ensures that we get the most up-to-date data as the cluster will be stopped when we are taking the backup. However, it can have a huge downtime of the cluster. Ideally, this should be the last option of taking a backup as it has a huge downtime and a high data footprint. Further, it does not support incremental backup.

Summary

In this chapter, we have learnt about backing up data in Hadoop. We discussed approaches of data backup in Hadoop—distributed copy and programming approach. We also became familiar with HBase and the use of HBase. We discussed various approaches of backing up data in HBase. Finally, we concluded the chapter with the comparison of all the approaches and pointers of when to use which approach. Once we have learnt about the backup options, next comes the recovery plan. The next chapter will discuss how to determine the recovery strategy.

5 Determining Recovery Strategy

Before we move ahead with this chapter, let's take a quick recap of all the things we went through till now. Let's quickly recall all our stops in our journey so far.

Chapter 1, *Knowing Hadoop and Clustering Basics*, laid the foundation of our journey. It explained about Hadoop administration, HDFS daemons, and HDFS and cluster setup and talked about the Hadoop best practices. It explained why Hadoop was needed and provided details about Apache Hive, Apache Pig, HBase, HDFS design, and more.

In *Chapter 2*, *Understanding Hadoop Backup and Recovery Needs*, we understood the philosophy behind backup and recovery. We noted all the points that we should take care of while forming a backup strategy and all the things that would be covered in it. We noted why backup in Hadoop was necessary and we went through the three areas that need to be taken care of (datasets, applications, and configuration).

Chapter 3, *Determining Backup Strategies*, was all about backup strategies. We saw all possible common failure types that could cause a disaster. We understood that hardware failure may lead to data loss. We came across situations wherein we had to deal with corrupt files that can cause further issues in the replication. Finally, we saw all the things that should be included in our backup strategy.

Finally, *Chapter 4*, *Backing Up Hadoop*, talked about various methods to take a backup in HBase. We finally saw data backup techniques in Hadoop. We saw all the possible options that can be run with the `distcp` command. We learned about HBase, its data model, and its origin and features, and saw various ways of backing HBase tables such as snapshots, replication, and export. Finally, we compared all these methods to help us to choose the right backup plan.

This chapter is going to talk about how to define your recovery strategy for various causes of failure when Hadoop is considered a critical solution.

We are going to see the various recovery options available to us. Hadoop introduced the concept of high availability in version 2.0. With HA available to us now, we no longer have only one NameNode running in HDFS. Earlier even though HDFS was efficient, it had a well-known single point of failure. This single failure could impact HDFS's availability. However, now with HA available, we can have more than one NameNode; of these, one node will be running in an active state, while the other will be in a passive state, so when the active node goes down, the other node will be ready to take over as NameNode.

The concept of recovery is very similar to data recovery in normal terms. Earlier, we learned about commissioning/decommissioning a node. The process of recovery is as simple as decommissioning a faulty node, taking the backup node and commissioning the active and working node. Although the process sounds too simple, it becomes fairly complicated when your time, resources, and money are involved.

Of course, everything is not as easy as that. If we are going to depend on a single solution there is going to be either recovery or no recovery at all. Anyone can say that this is not a good idea at all; it is much nicer to have a series of backup solutions so as to minimize the downtime to a very minimum value.

Knowing the key considerations of recovery strategy

When Hadoop is considered a critical solution, disaster recovery and backup need careful, thorough, and supervised planning. However, setting up a recovery plan can be complicated and confusing. Let's walk through some of the confusions and key considerations to be taken care of. As one rotten apple spoils the other apples, one corrupt copy will eventually spread out in the system and ultimately, decrease the chances of disaster recovery. This is the penultimate principle of disaster recovery.

As long as an error doesn't bound to or corrupt good data, you can secure whatever went amiss in your system.

- **Backup versus recovery**: Their purpose may sound similar, but there is a wide difference between them: backup is one of the strategies of protecting data assets to safeguard the data; safeguarding the data is just a part of the disaster recovery process. In our context, we don't have the option of replacement. There is either redundant data or no data at all. Unfortunately, it's not like when a piece is damaged, you go out and buy a new one. Today, in the era of the Internet, the World Wide Web and 24/7 operations, there are people who use this strategy (turn off the system, replace the corrupt system with a new system, and turn on the new system again).
- **Improper archive versus proper archive**: For most of the companies, a simple plan to take a backup may work. For them, taking a backup of the snapshots, would serve as an archive. However, the basic functionality of an archive may fail, which is generally termed as "improper archive". Archiving includes fetching the required results by finding and searching among all the data you may have. This would be hard to do in a backup. Hence, this type is generally termed as proper archive. This may not be the solution in today's world, where data flows in petabytes. We saw some of the ways to create our archives in an earlier chapter.

Now, moving towards the bigger term disaster recovery, the term disaster recovery is very much similar to any other recovery. Here, we add disaster because we are dealing with a huge amount of data and losing data would eventually mean losing customers and business. Disaster recovery is basically being alert about major disasters in your data centers like a tornado. Since we are dealing with a huge amount of data, we need to keep an eye open on all our data centers and the ability to get the business up and running when something disastrous happens. For this, we need something like `rsync`. For those not familiar with Unix, background `rsync` is a powerful tool to mirror data on a single host/multiple hosts. Rsync not only provides duplicate folders and files but also provides backup and restore options. Now, this kind of option is going to help a lot, as you will be able to get up and run immediately as your backup is ready to go.

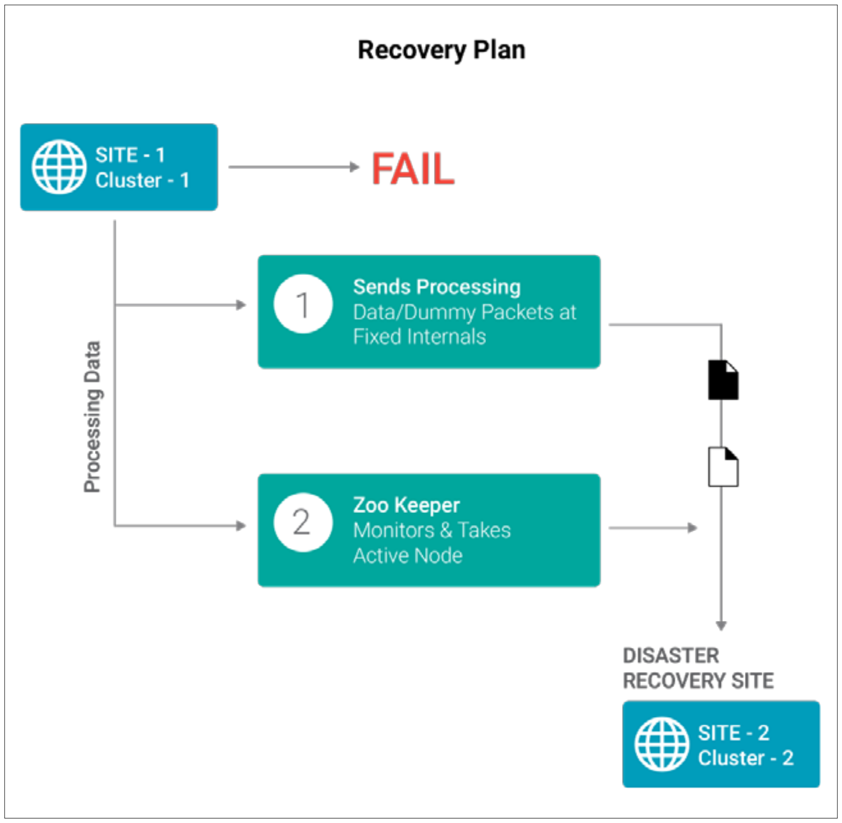
Let us now discuss the various options for the making of a disaster recovery site for a Hadoop-implemented data center. Naturally, the disaster recovery site would be at a distance of more than 500 miles. Now, obviously, there won't be online replication as that would need infinite bandwidth between the two sites. Therefore, we are in dire need of a solution here.

Keeping the problem in mind, let's walk towards working a solution and formulating the required steps needed to formulate our recovery plan.

Our plan should constitute the following points:

- A mechanism to run on the replication server just like a cron job. The mechanism runs daily and keeps the replication cluster and the production cluster in sync.
- A process that continually monitors the state of the server and immediately triggers the alarm when one of the DataNodes has failed and it is time to start the recovery process.
- A system that can change the configuration settings and make the replication cluster a production cluster if required.

Well, on the basis of all these assumptions, let us walk through two different approaches with the help of the following figure:



The situation remains the same: two different clusters (production and replication) located at a far distance. Say if the site fails, travelling to 500 miles and bringing back the replication cluster would be remorseful, because till then, the site would be down. Now, we will formulate two solutions based on the preceding mentioned three points:

- **Daily process/mechanisms**: So, in our best possible scenario, both the clusters would be running separate instances of Hadoop. The production server with the purpose of processing data, while the replication server with the objective of running Hadoop daemons such as JobTracker, TaskTracker, NameNode, secondary NameNode, and DataNode. It would be our secondary site with the primary goal to capture data from the first site and keep it on HDFS. So, when the first site fails, we can process the data available in HDFS.
- **Monitoring tool**: We need tools to continuously monitor and envision our environment. So, in case a disaster strikes, we will able to restore the system and get it up and running to its last safe state. This is the basic principle of the monitoring tool.
    - So, for communication between the primary and the replication cluster, one possible way is that the primary cluster constantly sends out packets of data to the replication cluster, and that's what we call the heartbeats packets. Now, if there are no data packets to be sent, it will send dummy packets, just for the second cluster to know "Hey, I am still alive..!"
    - We talked about only two clusters at this time, but what if there are more than two? The primary cluster has to send packets everywhere. Oh Boy! What a mess indeed. Fortunately, Apache ZooKeeper provides us with a centralized service for providing distributed synchronization and maintaining configuration information, that is, the ZooKeeper. ZooKeeper monitors and decides whom to make the active node in case the primary cluster fails.

- **Recovery process**: Until now, we are on the right track; a disaster occurred and we have our backup safe and sound. Now, it's all about recovering the data, and that's what this chapter is all about. We will broadly classify our recovery processes into three main portions.
    - **Disaster failure at data center**: We saw many causes of failures in the earlier chapters. Be it node failures, hardware failures, or others. Most of these fall into this category when we deal with recovery. The idea is fairly simple. Have a backup of changes/entire files away from the active rack. In case of failure, just decommission the fail node and commission the backup node. Now, there can be various approaches to this. In manual configuration, we need to manually configure everything; and in automatic configuration, we need to automatically configure the same, so as not to worry about manual setup, downtime, and more.
    - **Point-in-time copy for auditing**: It may so happen that we need to monitor the entire system or, in simple words, who did what inside the system, so that in case of failures, we can identify the source and recover the data from it by using this audit information.
    - **Accidental deletions**: The last and probably the most common problem faced. What happens when we accidently delete something? Is there a provision like recycle bin available, from which we may be able to recover. That's where we will learn about the Hadoop trash mechanism.

Disaster failure at data centers

Failure may occur anytime. Your JVM may crash or your NameNode may become healthy (behave abruptly) or may even crash. There has to be a failover option configured so as to easily configure other nodes to take place of the failover NameNode. Let's look at how different Hadoop components handle disaster failures.

How HDFS handles failures at data centers

Hadoop has been designed such that it can diminish the impact of hardware failures on the overall system availability. The entire Hadoop stack that exists today has high availability. High availability means that it does not have a single point of failure within the system that can bring down the entire site or make the site unavailable. Now, HDFS stores its metadata in two main places, namely the `fsimage` and the edit log.

Now, you do have the option of taking backup of your edit log, going to the kernel, and entering the following command:

```
./bin/hadoop namenode-recover
```

Then, after a lot of efforts, you can find out the full piece or a bit of the corrupt log. However, then, you are not using high availability at all. You have a secondary node and even a backup node all set up and waiting and you are not using it at all. It's like when it's raining, you don't take shelter under some shade but take an umbrella and stand outside. We are going to look over at the automatic failover introduced in Hadoop 2.0.

Before Hadoop 2.0, there were no provisions at all for HA; there was only one NameNode. So, there was no way ZooKeeper could be integrated, but then, HA was introduced and the failover configuration via ZooKeeper (ZKFC) was developed.

Automatic failover configuration

By default, NameNode HA needs a manual failover to be performed by a person outside a system. Enabling an automatic failover requires the need of two new components:

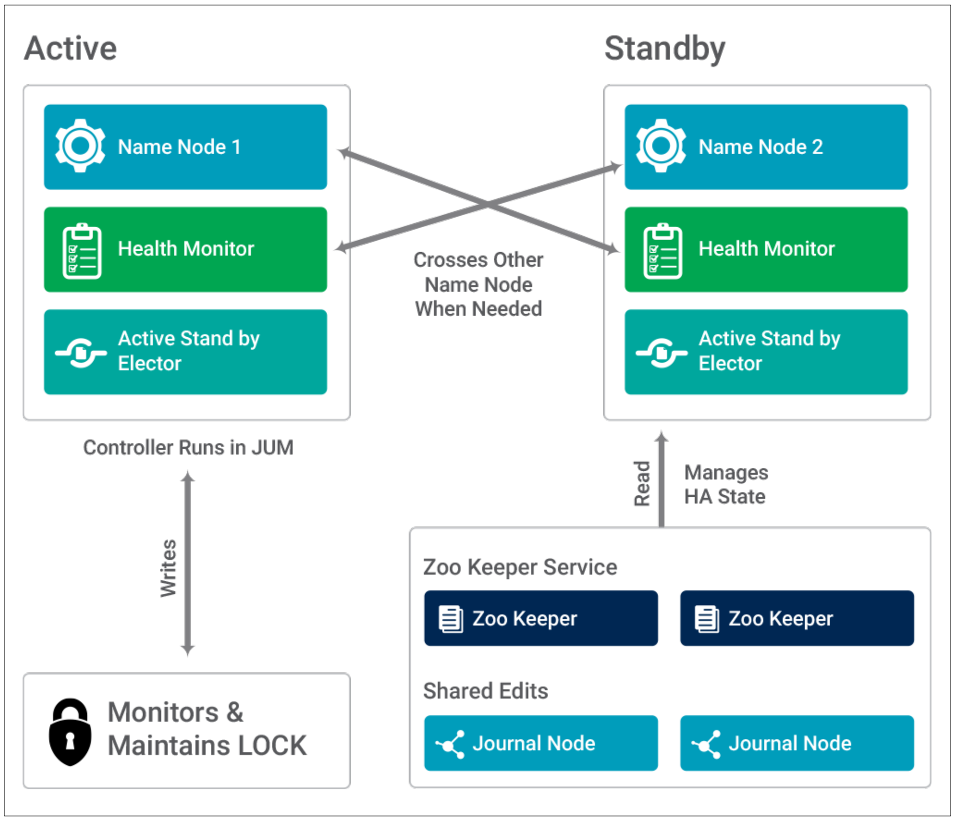
- **Apache ZooKeeper**: Service provider for distributed locking, coordination, and configuration
- **ZooKeeper Failover Controller**: Someone to monitor the nodes

What makes **ZooKeeper** (**ZK**) most attractive is that it satisfies all requirements of disaster recovery with no external configuration at all. Using ZK as a disaster recovery solution involves a successful establishment of the following three facts:

- **Failure detector**: The active NameNode creates a very short lasting node in ZK. If the active NameNode fails, this very short lasting node should be automatically deleted after a configured timeout period.
- **Active node locator**: The ability to write a small amount of information into ZK so that clients or other services can easily find the active node authoritatively.
- **Mutual exclusion of active state**: We use the HA at our disposal and ensure that at a time only one node is active at most.

How automatic failover configuration works

The ZooKeeper failover controller system is a ZooKeeper client that detects the states of the NameNodes. All the machines that run the NameNode services also run the ZKFC as the main goal of ZKFC is to monitor the health of the NameNode:



Now, three processes constitute the ZKFC:

- **Health monitoring**: ZKFC repeatedly pings its local NameNode with a health check command, so as to know if the NameNode is alive or not.
- **ZooKeeper session administration**: After checking whether the local NameNode is healthy, ZooKeeper starts its service and holds a session open in ZooKeeper. Now, based on whether the node is active or not, it will hold a special lock named `zlock`. If the node is active, it holds a special lock `znode`. This lock uses ZooKeeper's support for short-lived nodes. If the session expires, the lock will be automatically deleted.

- **ZooKeeper-based ballot**: If the local NameNode is healthy and is the only node to be holding `zlock`, ZKFC will try to acquire the lock, and if it succeeds, it has won the ballot, which will conclude in a failover process to run to make its local node active.

How to configure automatic failover

To configure an automatic failover, you need to change your `hdfs-site.xml` file, which basically is for saying "Hey dude, enable my automatic failover." The second change is in `core-site.xml`, which basically is for saying "These are the host ports running the ZooKeeper service; please note." The following are the steps:

1. Initializing the parameters:
    - Changes in `hdfs-site.xml`:

      ```
      <property>
      <name>dfs.ha.automatic-failover.enabled</name>
      <value>true</value>
      </property>
      ```

    - Changes in `core-site.xml`:

      ```
      <property>
      <name>ha.zookeeper.quorum</name>    <value>zk1.example1.
      com:2181,zk2.example2.com:2181,zk3.example3.com:2181</value>
      </property>
      ```

2. Loading HA state in ZooKeeper is the time to initialize the required state in ZooKeeper. Just log on to one of the NameNode hosts, and run the following command:

   ```
   $ hdfs zkfc -formatZK
   ```

   This will create a short lasting node (ephemeral node/znode), inside of which the automatic failover system will store its data. The created node is termed as an ephemeral node because it would be active only until the session that created the node is active, after which it would be deleted. So, we won't have any child nodes attached to an ephemeral node, because when attached to an ephemeral node, it won't last very long.

3. If ZooKeeper is not running, start it by executing the following command:

   ```
   su - zookeeper -c "export  ZOOCFGDIR=/etc/zookeeper/conf ; export
   ZOOCFG=zoo.cfg ; source /etc/zookeeper/conf/zookeeper-env.sh ; /
   usr/lib/zookeeper/bin/zkServer.sh start"
   ```

   After executing the preceding command, start the NameNodes and DataNodes.

4. Now, since the automatic failover configuration has been configured, the ZKFC daemon will automatically start on any host that runs a NameNode. Further, when NameNode starts automatically, one of the NameNodes will be made active and the others would be made passive. On whichever node the ZKFC is running, it will cause that node to become active. If ZKFC is not started by default, then run the following command:

   ```
   /usr/lib/hadoop/sbin/hadoop-daemon.sh start zkfc
   ```

5. Hurray! You are done. Go forward, kill one of your nodes, and see the magic for yourself.

It was quite easy to configure the automatic failover. Let's look at some of the troubleshooting we may come across during the configuration:

- **What will happen when ZooKeeper goes down? Am I doomed?**: It's obvious, no ZooKeeper, no automatic failover. No magic. Of course, you are not doomed. However, HDFS will continue to run without any impact. When ZooKeeper is up and running again, HDFS will reconnect with it without any problems.
- **Which node is active?**: Whichever node is started first, it will become primary or active, and if the active node is down, then `znode` associated with it will also be removed.
- **Can manual failover be configured?**: Yes, definitely, run the same `hdfs haadmin` command. That's the power of HA.
- **Any additional monitoring? What additional monitoring do we need?**: We need to continuously monitor ZKFC because if ZKFC fails, we won't be able to set up the automatic failover configuration.

Now, let's look at how to monitor a HA cluster.

We can monitor the HA cluster with the help of the following command:

```
Usage: DFSHAAdmin <options>
```

The following are the options:

```
[-ns <nameserviceId>]

[-transitionToActive <serviceId>]

[-transitionToStandby <serviceId>]

[-failover [--forcefence]

[--forceactive] <serviceId> <serviceId>]
```

```
[-getServiceState <serviceId>]
[-checkHealth <serviceId>]
[-help <command>]
```

Let's discuss the important options now.

The transitionToActive and transitionToStandBy commands

It basically executes a transition or changes the state of the given NameNode from/to active or standby as per the name suggestion. These commands will not attempt any fencing; hence, it should be rarely used.

Failover

It basically initiates a failover between two NameNodes.

The following two scenarios may occur:

- If the first NameNode is in the standby state, this command transitions the second NameNode to the active state without any error.
- If the first NameNode is in the active state, obviously, an attempt will be made to transition it to the standby state. If the primary NameNode's state remains unchanged, the fencing method will be attempted until one succeeds, and if it still remains unchanged, then an error will be returned.

Use the failover command, which will cause one node to be in the standby mode and another in the active mode.

The getServiceState command

This command is helpful to know whether the provided NameNode is in the active mode or the standby mode.

It prints out either `StandBy` or `active` to `STDOUT` statements.

The checkHealth command

This command is useful to print out the health of the node. It prints out zero or non-zero values based on whether all the internal processes are running as per expectation or not.

How HBase handles failures at data centers

Be it any plan, your business requirements are going to decide the way ahead. Once you have set up backups of your choice.

The following types of restoring take place on the basis of the type of recovery:

- **Failover to backup cluster**: If you have replicated your HBase data to a backup cluster already, this is child's play now. All you have to do is change your settings and point users and the end user programs to the new DNS.
- **Import table / restore a snapshot**: Import is a utility that will load data that has been exported to other forms previously, back into HBase. For example, to import 0.94 exported files in a 0.96 cluster or onwards, you need to set the `hbase.import.version` system property when running the `import` command as follows:

  ```
  $ bin/hbase -Dhbase.import.version=0.94
  org.apache.hadoop.hbase.mapreduce.Import <tablename> <inputdir>
  ```

- **Point the HBase root folder to backup location**: One of the not-so-popular ways in which you can recover yourself from a disaster is to simply change the location of your HBase folder. You can do so by pointing `hbase.root.dir` (this is situated in `hbase-site.xml`) to the new HBase folder. Now, you may think if we are going to copy the entire data structure to your production cluster, then definitely, the purpose of snapshots would be defeated. Moreover, you are correct; this is also one of the unfavorable ways. One more reason can be that your `.meta` is out of sync.

Restoring a point-in time copy for auditing

Because of the ever-changing business requirements, for the accomplishment of various business goals, one common use case is to monitor the disaster recovery capacity. That's where the feature of auditing comes in handy.

In Hadoop, HDFS provides an exceptional feature for logging and noting all filesystem access requests. Audit logging is done using `log4j` logging at the information level. It is disabled by default.

Looking over the configuration file in `log4j.properties`, find the following line of code:

```
log4j.logger.org.apache.hadoop.hdfs.server.namenode.FSNameSystem.
audit=WARN
```

Replace the word `WARN` with `INFO`. The result will be a log written to NameNode's log for every HDFS event. Audit events are emitted as a set of key-value pairs:

Key	**Value**
`ugi`	`<user>,<group>[,<group>]*`
`Ip`	`<client ip address>`
`cmd`	`(open|create|delete|rename|mkdirs|listStatus|setReplication|setOwner|setPermission)`
`src`	`<path>`
`dst`	`(<path>|"null")`
`perm`	`(<user>:<group>:<perm mask>|"null")`

An example of one line of audit log is as follows:

```
<log4j header> ugi=parth-ghiya,users,staff ip=/127.0.0.10 cmd=mkdirs
src=/foo/bar dst=null perm=parth-ghiya:staff:rwxr-xr-x
```

This line basically says that `parth-ghiya` belongs to the `staff` group, logged in from address `127.0.0.10` with the `mkdir` command on the following source (making a new folder at source) with permission to read, write, and edit.

[Configure `log4j` so that the audit log is written to a separate file and does not collide with NameNode's other log entries.]

So, the supported systems that provide the audit log functionality as of December 2014 are as follows:

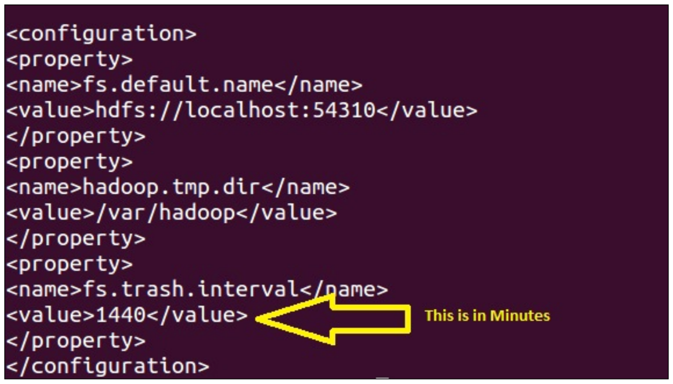
- HDFS
- HBase
- Hive
- Impala
- Oozie

Restoring a data copy due to user error or accidental deletion

Humans are prone to mistakes; there may be a time where you may delete necessary files from Hadoop. That's where Hadoop's trash mechanism serves as your defense line.

This trash mechanism in Hadoop has to be enabled in HDFS by using the following method. Go to your `core-default.xml` and look for the `fs.trash.interval` property. By default, its value is zero, which means that it won't keep values in the trash folder, that is, no backup for us. We can change its value and keep its new value as the number of minutes after which the backups get deleted. If the value stays at zero, it won't back up the data and the trash mechanism won't be enabled. Perform the following steps:

1. Set this property in `/conf/core-site.xml` as seen in the following figure:



```
<configuration>
<property>
<name>fs.default.name</name>
<value>hdfs://localhost:54310</value>
</property>
<property>
<name>hadoop.tmp.dir</name>
<value>/var/hadoop</value>
</property>
<property>
<name>fs.trash.interval</name>
<value>1440</value>
</property>
</configuration>
```

Have a look at the following code snippet:

```
<property>
  <name>fs.trash.interval</name>
  <value>60</value>
  <description>Number of minutes after which the checkpoint gets
deleted. If zero, the trash feature is disabled.
  </description>
</property>
```

One more configuration property is `fs.trash.checkpoint.interval`. It specifies the time period or the checkpoint interval. It checks the trash folder at the time specified here, after which it deletes all files that are older than that time period. For example, if you set the interval to 1 hour; then, after each hour, a check is performed to see whether any file's trash interval time is greater than 1 hour; if so, this file is deleted.

2. Run the `syncconf.sh` script; it will synchronize the configurations:

```
syncconf.sh hadoop
```

3. Restart your Hadoop server to let the changes take place:

```
stop.sh hadoop
start.sh hadoop
```

4. The trash folder, by default, is `/user/`; this is how you can recover a deleted item.

Defining recovery strategy

A disaster recovery plan goes further than just the technology, and the needs for disaster recovery can greatly influence the solution. Each problem here has its own unique solution. Let's look at an example. In case of a fire, one common approach is to power down the main frame and the other systems attached to it, and then, turn on the sprinklers. Then, the solution is to disassemble the components and then, manually dry them with a hair dryer. One Google hit and you will get scores of results suggesting various methods and tools. The larger the number of results, the greater is the confusion. So let's start defining our strategy and all the points that we should keep in mind.

So, starting with the recovery strategy, the best recovery strategy would be one in which you never have to do anything, of course. In the case of a disaster, the system is triggered and the recovery process starts on its own.

So, at its core, in simple words, our strategy should have the following three components:

- Centralized configuration (one-stop access for everything: backups, log files, and so on)
- Monitoring (observe the health and status of the nodes and track replication)
- Alerting (something to trigger that a disaster has occurred)

Before walking through the strategy, let's walk through two important terms:

- **Recovery Point Objective** (**RPO**): RPO is the final limit of data that a business can lose before a disaster can cause extreme problems or may completely shut down a firm. It's basically geared mostly towards data backup. Any critical solution relying solely on data is extremely vulnerable during a shutdown. Take the example of an e-commerce website. Now, say disaster strikes and the whole inventory and billing data disappears or becomes out of date and it's a complete mess! With respect to the business continuity, RPO is the time that a business can afford to lose data, before serious consequences are seen on their business.
- **Recovery time objective** (**RTO**): RTO, as the name suggests, is much simpler and simply suggests the target time for the resumption of activities after which a disaster has stuck. The larger the RTO, lesser is the money that it needs. Take a few steps back and recall snapshots, they are point-in-time copies of filesystem. How dramatically would they reduce the recovery time and how much money would be saved?

Now that you been through both of the concepts, let's walk through the formulation of a more practical recovery strategy of ours.

Centralized configuration

It helps to define the backup and disaster recovery policies and configuration and apply them across services. Policies can be enacted to be run on a single scheduled job or as a recurring scheduled job. It could be like the way you run a cron job for example.

Monitoring

Monitoring provides the option to monitor the nodes and track their health, check the replication process, and observe which nodes have failed. It should basically track all the causes of hardware failures.

Alerting

Alerting should give user notifications regarding replication job failures or other problems in the system.

Thus far, our recovery strategy sounds pretty good. We have a centralized configuration, more than one NameNode, only one node in the active mode, and the monitoring and alerting system setup, so if one NameNode is down, others can take its place.

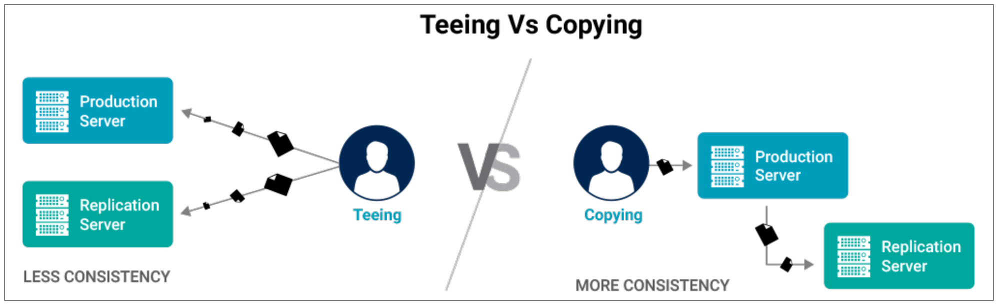
Now, a major question: How do we distribute data among all of the clusters? There are generally two processes for this: teeing and copying. Let's look at them in detail now.

Teeing versus copying

Now, in the case of a disaster recovery strategy, there are two different processes that can be used, namely teeing and copying.

Let us understand the difference between them by seeing a small conversation between them, where they start by introducing themselves.

Teeing	"Hi! During my process, data sent during the injest phase is sent to both the production and replication clusters. The production cluster and the replication cluster are in constant sync."
Copying	"Hello! During my process, data is just sent to the production cluster. Taking it to the replication cluster is a completely different step for me. I just have to run this different step, and not worry about whether the data has reached both clusters or not."
Teeing	"Ahaa! You see Mr. Copying, due to my combined first step, I have the minimal time delay between my replication and the production server. So, if one fine day Mr. Ghiya's site goes down, I just have to point to the replication server using DNS techniques and my task is done."
Copying	"There you see my friend, when you say you do both things in one step, you have to execute the processing step twice in both clusters. I have the added advantage of executing a processing step only once. Once this is done, I have consistent data between both sites, but for you, there is no consistency between sites."
Teeing	You see the negative side, my friend. My use case is not for the performance at all. For users who need more data availability rather than performance on critical processed data, I am their best buddy.
Teeing	Further, you missed something. Due to incremental copies, you will have a time delay for recovery objectives. Now, this is not going to bring a smile on Mr. Ghiya's face.



So, we saw both the processes here, each with it pros and cons. An obvious question might be guidance. Let's look at a quick comparison between them:

Teeing	**Copying**
Sends data during the ingest phase to the production and replication clusters at the same time.	Data is copied from a production server to a replication server as a separate task after processing.
The time delay is minimal between clusters as both the production and the replication clusters are injected at the same time.	For incremental copies to be maintained, objectives to do recovery point objectives could be more.
Thus, the bandwidth required could be larger.	For a one-time process, initially, a greater bandwidth could be required.
Requires re-processing of data on both sides, that is, at the replication and the production clusters.	One-time processing is required, and then, we have processed data.
No consistency between two sites: which site contains the processed data, and which one does not?	Always consistent data at both sites.

Well, the answer remains the same: the choice between the two is situation-specific, but generally, use copying over teeing. Further, you need to look at the following points to make your recovery strategy better:

- Are you sending uncompressed data? Is your data compressed?
- Are you doing something without forgetting the bandwidth? Do you know your bandwidth needs?
- Are you aware about your network security setup? Do you have a system setup to detect a failure as soon as possible?
- Have you left any loopholes? Have you configured your security properly?
- Are you aware of version copying? How will you take care of cross version copying?
- Are you maintaining the daily ingest rate records?
- Have you set up an authentication protocol such as Kerberos?

Summary

In this chapter, we learned about the recovery process and strategies. We got an enhancing vision of what to do after taking a Hadoop backup. We saw different situations that can cause a failure and how these are handled by Hadoop. We got an introduction to the automatic recovery process introduced in Hadoop 2.0. We looked at the ZooKeeper architecture for failover configurations. Then, we saw a detailed introduction to the recovery strategy, the process, the components involved, and more. The next chapter will give more insights on a practical implementation of recovery strategies.

6 Recovering Hadoop Data

In the previous chapter, we learnt about defining a strategy for recovery in Hadoop. We also learnt about key considerations to be taken care of while defining the strategy for Hadoop data. *Chapter 5*, *Determining Recovery Strategy*, helped us to gain much better insight on what to do after taking a Hadoop backup. Various types of situations that might cause a failure and overcoming such failures were explained. Along with this, we looked at the ZooKeeper architecture and automatic recovery process. We covered the detailed recovery strategy, the process, the components involved, and much more.

As we have now learnt about defining the recovery strategies of Hadoop, let us now dive deep in to how we can recover data in scenarios such as a failover and corruption of working drives, NameNodes, tables, and metadata. We will cover these scenarios under the specific conditions mentioned as follows:

- Failover to backup cluster
- Importing table or restoring a snapshot
- Pointing the HBase root folder to the backup location
- Locating and repairing corruption
- Recovering a drive to a working state
- Lost files
- The recovery of NameNode

Let us now not wait anymore to discuss in detail how to recover Hadoop data in critical situations.

Failover to backup cluster

Since we have understood by now that a manual failover mechanism is incapable of triggering an automatic failover where NameNode failures are detected, in such cases, an automatic failover mechanism ensures providing a hot backup in the event of a failover. This was taken care of with ZooKeeper. To configure an automatic failover, we need to have nodes in an odd number quorum, to start with at least three of them that would keep most of the ZooKeeper servers running in case of any NameNode failures. We also have to take care of keeping ZooKeeper grouped in separate network racks as Hadoop stores enormous data that would easily reach to petabytes, which we can imagine for Facebook. Facebook has millions of users who have data; it consists of comments, user posts, status, pictures, and videos. How has Facebook been managing backup for all these and not just taking backups as a huge amount of data is constantly updated on its servers by millions of users? Addressing a common question like what has been changed after the latest backup has also become a tedious job if it is not addressed in a systematic manner, particularly when a huge cluster implementation has been done.

Let us start configuring the fully distributed mode of the Hadoop cluster setup with an automatic failover. We installed Hadoop 2.6 in all nodes; it consists of two DataNodes, three NameNodes, and one client machine. The setup of a fully distributed Hadoop cluster has been achieved using ZooKeeper for an automatic failover. The following table describes the hostname and the IP address of the configured machines in our scenario:

namenode1	**ha-namenode01**	**172.16.1.101**
namenode2	**ha-namenode02**	**172.16.1.102**
namenode3	**ha-namenode03**	**172.16.1.103**
datanode1	**ha-datanode01**	**172.16.1.104**
datanode1	**ha-datanode02**	**172.16.1.105**
client	**ha-client**	**172.16.1.106**

Installation and configuration

Here, we will cover user settings, password-less SSH configuration, installation of Java, and finally Hadoop.

Let us download Apache Hadoop and Oracle JDK 8 before we move to the rest of the configuration.

The following configuration needs to be done in all nodes:

- Deactivate the firewall on all nodes
- Deactivate SELinux on all nodes
- Update the `/etc/hosts` file in all nodes for respective hostname and their IP addresses
- Oracle JDK is recommended to be used

The user and group settings

Let us configure `ha-namenode01`, `ha-namenode02`, `ha-namenode03`, `ha-datanode01`, `ha-datanode02`, and `ha-client`, which are part of our cluster.

Let us create a group called `hadoop` and create a user `hadmin` to perform tasks related to all Hadoop administrative tasks:

```
# groupadd hadoop
# useradd -m -d /home/hadmin -g hadoop hadmin
# passwd hadmin
```

Java installation

Java installation needs to be carried out in all the nodes. It is preferable to use Oracle's JDK:

- `ha-namenode01`
- `ha-namenode02`
- `ha-namenode03`
- `ha-datanode01`
- `ha-datanode02`
- `ha-client`

Password-less SSH configuration

To start HDFS and MapReduce daemons, we will need a password-less SSH environment in all nodes.

Let us configure the NameNodes used in our environment, that is, `ha-namenode01`, `ha-namenode02`, and `ha-namenode03`:

```
hadmin@ha-namenode01:~$ ssh-keygen -t rsa
hadmin@ha-namenode01:~$ ssh-copy-id hadmin@ha-namenode01
```

```
hadmin@ha-namenode01:~$ ssh-copy-id hadmin@ha-namenode02
hadmin@ha-namenode01:~$ ssh-copy-id hadmin@ha-namenode03
hadmin@ha-namenode01:~$ ssh-copy-id hadmin@ha-datanode01
hadmin@ha-namenode01:~$ ssh-copy-id hadmin@ha-datanode02
```

Let us now test password-less SSH from `ha-namenode01` to `ha-namenode02`, `ha-namenode03`, `ha-datanode01`, and `ha-datanode02`:

```
hadmin@ha-namenode01:~$ ssh ha-namenode02
hadmin@ha-namenode01:~$ ssh ha-namenode03
hadmin@ha-namenode01:~$ ssh ha-datanode01
hadmin@ha-namenode01:~$ ssh ha-datanode02
```

ZooKeeper installation

Let us now configure the NameNodes used in our environment, that is, `ha-namenode01`, `ha-namenode02`, and `ha-namenode03`.

Download ZooKeeper from `http://apache.spinellicreations.com/zookeeper/zookeeper-3.4.6/` and extract the ZooKeeper tarball in the `/opt` system folder on all three NameNodes. Set the permissions for the `hadmin` user.

Let us configure NameNodes used in our environment, that is, `ha-namenode01`, `ha-namenode02`, and `ha-namenode03`:

```
root~# chown -R hadmin:hadoop /opt/zookeeper-3.4.6
```

Let us now create a `zoo.cfg` file, which would reside in the `conf` folder and do the required modifications as follows:

```
hadmin~$ vi /opt/zookeeper-3.4.6/conf/zoo.cfg
tickTime=2000
dataDir=/opt/ZooData
clientPort=2181
initLimit=5
syncLimit=2
server.1=ha-namenode01:2888:3888
server.2=ha-namenode02:2888:3888
server.3=ha-namenode03:2888:3888
```

Add the executable path to the `bashrc` file of all the NameNodes:

```
hadmin~$ vi ~/.bashrc
export PATH=$PATH:/opt/zookeeper-3.4.6/bin
```

`ticktime` determines the time in milliseconds to determine heartbeats for ZooKeeper.

`dataDir` refers to the folder where ZooKeeper would store data (in-memory data snapshots and transaction logs for data updates).

`clientPort` is the port number for the client connections.

`initLimit` provides the maximum time duration used by the ZooKeeper server in the quorum to connect to the master.

`syncLimit` is the maximum time for a server to be outdated from a master.

`server.n` provides a list of servers, which runs the ZooKeeper service and the first port is used by the slaves to connect to the master and the second port is used by the ZooKeeper servers while the master is selected.

We would have to create a `myid` file in the `data` folder, which is often missed. ZooKeeper leverages the `myid` file to get identified in the cluster. Let us create a `Data` folder with the required `hadmin` privileges in `/opt` as stated in the `/opt/zookeeper-3.4.6/conf/zoo.cfg` file and create a file named `myid` in it. Add `1` to the `myid` file in the `ha-namenode01` NameNode, `2` to the `myid` file in the `ha-namenode02` NameNode, and `3` to the `myid` file in the `ha-namenode03` NameNode.

Let us now configure the NameNodes used in our environment, that is, `ha-namenode01`, `ha-namenode02`, and `ha-namenode03`:

```
root:~# mkdir -p /opt/ZooData
root:~# chown -R hadmin:hadoop /opt/ZooData/
Location: ha-namenode01
hadmin@ha-namenode01~$ vi /opt/ZooData/myid
1
Location: ha-namenode02
hadmin@ha-namenode02~$ vi /opt/ZooData/myid
2
Location: ha-namenode03
hadmin@ha-namenode03~$ vi /opt/ZooData/myid
3
```

We can verify the same with the following command in all the nodes. Each NameNode can be checked with the relevant `myid` value specified:

```
hadmin@ha-namenode01~$ cat /opt/ZooData/myid
```

The `myid` file consists of a unique number between `1` and `255`; `myid` represents the server ID for ZooKeeper.

To start ZooKeeper, use the following command; this would start ZooKeeper, which we need to execute on all three NameNodes:

```
hadmin~$ zkServer.sh start
```

Hadoop installation

Let us configure the NameNodes, DataNodes, and client used in our environment, that is, `ha-namenode01`, `ha-namenode02`, `ha-namenode03`, `ha-datanode01`, `ha-datanode02`, and `ha-client`.

Let us extract the Hadoop tar file in the `/opt` folder and change the ownership of this folder to the `hadmin` user:

```
root:~# cd /opt
root:~# tar -xzvf hadoop-2.6.0.tar.gz
root:~# chown -R hadmin hadoop-2.6.0/
```

Now let us login as the `hadmin` user and set the environment variables in the `.bashrc` file:

```
hadmin:~$ vi ~/.bashrc
###Configuration for Java - JAVA_HOME to be the root of your Java
installation###
JAVA_HOME=/opt/jdk1.7.0_30/
export PATH=$PATH:$JAVA_HOME/bin

###Configuration for Hadoop###
HADOOP_PREFIX=/opt/hadoop-2.6.0/
export PATH=$PATH:$HADOOP_PREFIX/bin:$HADOOP_PREFIX/sbin
```

The test installation of Hadoop

The following command will help to verify a successful Hadoop installation:

```
hadmin:~$ hadoop version
Hadoop 2.6.0
Subversion https://git-wip-us.apache.org/repos/asf/hadoop.git -r
e34965099ecb8d220gba99dc54d4c99c8f9d33bb1
Compiled by jenkins on 2012-12-13T21:20Z
Compiled with protoc 2.4.0
From source with checksum 18e43457c8f927c9695f1e9522849d6a
```

Hadoop configuration for an automatic failover

To configure Hadoop with an automatic failover cluster, we need to make changes in the configuration files that reside in the `/opt/hadoop-2.6.0/etc/hadoop/ folder`.

Let us configure the NameNodes, DataNodes, and client used in our environment, that is, `ha-namenode01`, `ha-namenode02`, `ha-namenode03`, `ha-datanode01`, `ha-datanode02`, and `ha-client`.

The following changes needs to be done in the `hadoop-env.sh` file:

```
hadmin:~$ vi /opt/hadoop-2.6.0/etc/hadoop/hadoop-env.sh
export JAVA_HOME=/opt/jdk1.7.0_30
export HADOOP_LOG_DIR=/var/log/hadoop/
```

Let us create a folder for logs as specified in the `hadoop-env.sh` file with the required `hadmin` user permissions:

```
root:~# mkdir /var/log/hadoop
root:~# chown -R hadmin:hadoop /var/log/hadoop
```

Let us configure the NameNodes, DataNodes, and client used in our environment, that is, `ha-namenode01`, `ha-namenode02`, `ha-namenode03`, `ha-datanode01`, `ha-datanode02`, and `ha-client`:

```
hadmin:~$ vi /opt/hadoop-2.6.0/etc/hadoop/core-site.xml
```

The following changes need to be made in the `core-site.xml` file:

```
<configuration>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://haauto/</value>
  </property>
</configuration>
```

The following changes need to be made in the `Hdfs-site.xml` file:

```
<configuration>
  <property>
  #Replication factor is set to '2' as we have two data nodes.
    <name>dfs.replication</name>
    <value>2</value>
  </property>
  <property>
    <name>dfs.name.dir</name>
    <value>file:///hdfs/name</value>
  </property>
  <property>
    <name>dfs.data.dir</name>
    <value>file:///hdfs/data</value>
  </property>
  <property>
    <name>dfs.permissions</name>
    <value>false</value>
  </property>
  <property>
    <name>dfs.nameservices</name>
    <value>haauto</value>
  </property>
  <property>
    <name>dfs.ha.namenodes.haauto</name>
    <value>namenode01,namenode02</value>
  </property>
  <property>
    <name>dfs.namenode.rpc-address.haauto.namenode01</name>
    <value>ha-namenode01:8020</value>
  </property>
  <property>
    <name>dfs.namenode.http-address.haauto.namenode01</name>
    <value>ha-namenode01:50070</value>
  </property>
  <property>
    <name>dfs.namenode.rpc-address.haauto.namenode02</name>
    <value>ha-namenode02:8020</value>
  </property>
  <property>
```

```
      <name>dfs.namenode.http-address.haauto.namenode02</name>
      <value>ha-namenode02:50070</value>
    </property>
    <property>
      <name>dfs.namenode.shared.edits.dir</name>
      <value>qjournal://ha-namenode01:8485;
        ha-namenode02:8485/haauto</value>
    </property>
    <property>
      <name>dfs.ha.fencing.methods</name>
      <value>sshfence</value>
    </property>
    <property>
      <name>dfs.ha.fencing.ssh.private-key-files</name>
      <value>/home/hadmin/.ssh/id_rsa</value>
    </property>
    <property>
      <name>dfs.ha.automatic-failover.enabled.haauto</name>
      <value>true</value>
    </property>
    <property>
      <name>ha.zookeeper.quorum</name>
      <value>ha-namenode01:2181,ha-namenode02:2181,
        ha-namenode03:2181</value>
    </property>
  </configuration>
```

Let us quickly go through the following checklist:

1. Create the `/hdfs/name` folder in all the NameNodes with the required `hadmin` privileges:

   ```
   root:~# mkdir -p /hdfs/name
   root:~# chown -R hadmin:hadoop /hdfs/name
   ```

2. Create the `/hdfs/data` folder in all DataNodes with the required `hadmin` permissions:

   ```
   root:~# mkdir -p /hdfs/data
   root:~# chown -R hadmin:hadoop /hdfs/data
   ```

3. In the `ha-client` host, add the following property to the `hdfs-site.xml` file: the `dfs.client.failover.proxy.provider.haauto` variable specifies the Java class name used by clients to determine the NameNodes active in a cluster:

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

4. Enable an automatic failover for the `nameservice` ID `haauto` by setting the property `dfs.ha.automatic-failover.enabled.haauto` to `true`.

The following changes need to be done in the `Slaves` file. This file contains the hostnames of the DataNodes.

Let us configure the NameNodes used in our environment, that is, `ha-namenode01`, `ha-namenode02`, and `ha-namenode03`:

```
hadmin@hanamenode01:~$ vi /opt/hadoop2.6.0/etc/hadoop/slaves
ha-datanode01
ha-datanode02
```

Let us now initialize and start an automatic failover on Hadoop cluster that we have configured.

Preparing for the HA state in ZooKeeper

ZooKeeper needs to be initialized by running the following command from any one of the NameNodes. Here the location is `ha-namenode01`:

```
hadmin@ha-namenode01:~$ hdfs zkfc -formatZK
```

Formatting and starting NameNodes

NameNodes need to be formatted to start using the HDFS filesystem.

Let us configure the NameNode used in our environment, that is, `ha-namenode01`:

```
hadmin@ha-namenode01:~$ hadoop namenode -format
hadmin@ha-namenode01:~$ hadoop-daemon.sh start namenode
```

Let us configure the NameNode used in our environment, that is, `ha-namenode02`:

```
hadmin@ha-namenode02:~$ hadoop namenode -bootstrapStandby
hadmin@ha-namenode02:~$ hadoop-daemon.sh start namenode
```

By default, both NameNodes configured will be in the `standby` state.

Starting the ZKFC services

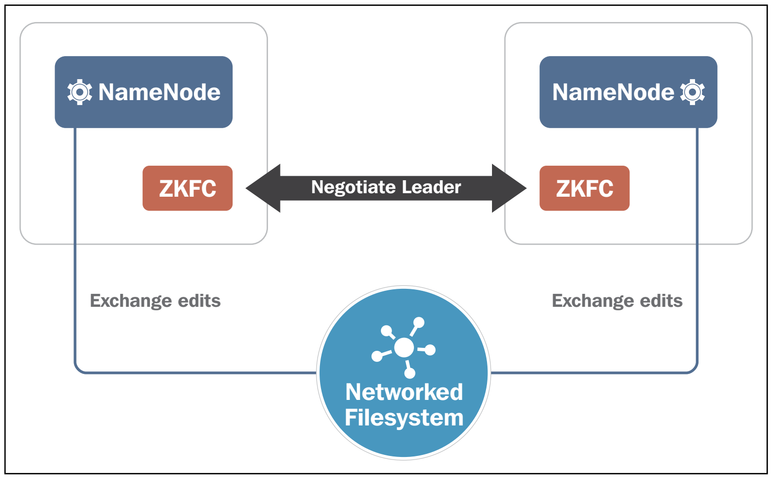
The **ZooKeeper Failover Controller** (**ZKFC**) service has to be started to make any one NameNode as `active`. Let us run the following command in both the `NameNodes`:

```
hadmin@ha-namenode01:~$ hadoop-daemon.sh start zkfc
hadmin@ha-namenode02:~$ hadoop-daemon.sh start zkfc
```

Once the ZKFC service is started, we can verify with the help of the following command that one of the NameNodes is active:

```
hadmin@ha-namenode01:~$ hdfs haadmin -getServiceState namenode01
namenode01 | active
hadmin@ha-namenode01:~$ hdfs haadmin -getServiceState namenode02
namenode02 | active
```

With the help of the following image, we can understand what we have done so far for ZooKeeper:



Starting DataNodes

Let us start the DataNodes with the help of the following command from one of the NameNodes:

```
hadmin@ha-namenode01:~$ hadoop-daemons.sh start datanode
```

Verifying an automatic failover

Let us verify an automatic failover; we can do so by using the command line or from the NameNode web interface.

The following command will help us to verify from the command line. It can be either active or on standby:

```
hadmin@ha-namenode01:~$ hdfs haadmin -getServiceState namenode01
namenode01 | active
hadmin@ha-namenode01:~$ hdfs haadmin -getServiceState namenode02
namenode01 | active
```

Once we verify an automatic failover and the NameNode active state, we can fail the active NameNode by running the `jps` command and killing the NameNode daemon. We will be able to see other NameNodes become active automatically. That's it!

Now let us move on to scenarios where we have to restore from a snapshot or an import table.

Importing a table or restoring a snapshot

The corruption of the file system might be because of multiple reasons, such as software upgrades corrupting the filesystem, human errors or, bugs in the application. With the help of snapshots in HDFS, we can reduce the probable damage to the data in the system during such scenarios.

The snapshot mechanism helps to preserve the current state of the filesystem and enables administrators to roll back the namespace and storage states in the working condition.

HDFS can have only one existence of a snapshot with an optional configuration with the administrator to enable it during startup. If a snapshot is triggered, NameNode refers to the checkpoint and the journal file and merges them in the memory. It would now write a new checkpoint and an empty journal on to a new location, so the old checkpoint and journal remain unaffected.

During the handshake, NameNode pushes DataNodes to check whether a snapshot is to be created or not. A local snapshot in DataNode cannot be generated by copying the folder storing data files as it would obviously double up the storage capacity requirement on the cluster. As an alternative, each DataNode creates a copy of the data folder and have hard linked the existing block files onto it. When DataNode removes a block, only a hard link would be removed and block modifications are made while appending with the help of a copy-on-write technique. Hence, the initial block replicas are intact in their original storage folder.

NameNode recovers the checkpoint that was saved while the snapshot was last created. However, DataNodes restore to older folders, which were renamed and run as a background process to remove the block replicas created after the snapshot was taken. However, once a rollback is done, there is no facility to roll forward.

While developing a system, the format of the NameNode checkpoint files and journals or the representation of the block replica files of DataNodes may change. The layout version identifies the format of the data representation and is persistently stored in the DataNode and NameNode storage folders. During startup, each of the nodes compares the version of the layout of the software currently being stored in its data folders and converts it automatically to the new format from the old format. For conversion, it is mandatory to create a snapshot when the system is restarted with a new version of the software layout.

Let us now quickly go through the commonly used commands for importing a table.

Import a table from the local environment into the existing HBase table:

```
bin/hbase org.apache.hadoop.hbase.mapreduce.Driver import table_name /
datafolder
```

It will be good to compare the number of rows along with its count before export and after the import is completed:

```
bin/hbase org.apache.hadoop.hbase.mapreduce.Driver rowcounter table_name
```

The number of rows is also visible in the HDFS counter called rows; have a look at the following sample output:

```
mapred.JobClient:   org.apache.hadoop.hbase.mapreduce.RowCounter$RowCount
erMapper$Counters
mapred.JobClient:     ROWS=422579
```

Pointing the HBase root folder to the backup location

HBase stores all data in the folder mentioned by the `hbase.rootdir` configuration property. This is where everything is preserved. However, copying this folder isn't a viable solution for multiple reasons, particularly in a large cluster and for running an HDFS cluster.

While the HBase cluster is running, there are various things going on simultaneously such as region splits, compactions, and MemStore flush. These would keep on changing the stored data, which makes copying the HBase root folder a fruitless effort. One of the other factors to be considered is while the HBase cluster is running, MemStore stores the data that isn't flushed.

However, if we stop the HBase cluster in a clean way, the MemStore data gets cleared and doesn't get altered by any other process. Copying the entire root folder data could be a good point-in-time backup alternative during this time. Unfortunately, it doesn't help during an incremental backup, which is a current challenge; hence, at the end of the day, this approach is not practical. Restoring to the root folder from a backup location is kind of starting HBase normally with the root folder pointed to the backup location.

Replace `hbase.rootdir` in `hbase-site.xml` with a new location of the folder from the backup location where we need HBase to store data:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <name>hbase.rootdir</name>
    <value>file:///home/hduser/HBASE/hbase</value>
  </property>
  <property>
    <name>hbase.zookeeper.property.dataDir</name>
    <value>/home/hduser/HBASE/zookeeper</value>
  </property>
</configuration>
```

Let us now move to understand what we shall do in the case of corruptions and how we find them and repair them.

Locating and repairing corruptions

HDFS supports the `fsck` command, which helps to check for various discrepancies. It helps to identify problems related to files like under-replicated blocks or missing blocks. What we would have seen so far in the traditional `fsck` command of a native filesystem is that it can fix the detected errors, which isn't the case in HDFS `fsck`. In general, NameNode identifies and resolves most of the failures.

The `fsck` command by default doesn't check for open files but provides an option to check for all files and report it accordingly. As the Hadoop `fsck` command is not part of the Hadoop shell command list, it can be executed as `bin/hadoop/fsck`. The `fsck` command can be executed to check for the whole file system or some part of it.

Hadoop's `fsck` utility command output will show the `CORRUPT` status when the file is found to be corrupted. The following command will check the complete filesystem namespace:

```
# hadoop fsck /
```

The following command will show details in the verbose mode, which we have nailed down to a meaningful output. It ignores the empty lines; however, dots and lines will be brief about replication:

```
hadoop fsck / | egrep -v '^\.+$' | grep -v replica
```

Using the `-files`, `-locations`, and `-blocks` parameters available in `fsck`, we can also find out which blocks are missing and where they're located:

```
hadoop fsck /path/to/corrupted/file -files -locations -blocks
```

Once we identify the corruption, we can find the file with the following command, which shows us `MISSING` where the possibility from all sources for the block are missing:

```
#hadoop fsck /corrupt/file -locations -blocks -files

/home/chintan/cm.iso 4513401622 bytes, 53 block(s):  OK

0. blk_-6574899661619162507_21831 len=134317738 repl=3
[172.16.1.101:50010, 172.16.1.102:50010, 172.16.1.103:50010]

1. blk_-8503098334894134725_21831 len=134317738 repl=3 MISSING!

2…
```

With the help of the output of this command, we can determine where the blocks for the file reside. The file might be distributed in multiple blocks if it is of a large size.

We can use the block numbers reported to check the DataNode and NameNode logs to find the system or systems in which the blocks reside. We can then also check for errors in the filesystem in the reported machines, such as missing mount points, DataNode might not be running, or the filesystem is provisioned/reformatted with the help of this file or the data that is corrupted and bring that file or data back online to restore it to a healthy state.

Once we have identified and repaired the corrupted blocks, and if there is no scope to recover any further blocks, we can use the following command to restore the filesystem to the healthy state:

```
hadoop fs -rm /file/with/corrupted/blocks
```

As we have now understood how to take care of recovering from the corruption of the Hadoop filesystem, let us get to the next scenario where we will recover a working drive.

Recovering a drive from the working state

In this situation, what would you do when a DataNode that contains multiple data disks is corrupted, but the rest of the disks are in a good condition? Let us recover from such a situation. We can recover the blocks after remounting the working data disks on to another DataNode. If there is space available, we can simply attach the disk to another DataNode and copy the data from the version file to the corresponding folder.

In case of a space crunch on another DataNode, we can attach the working disk temporarily and retire another DataNode to verify whether all blocks are copied somewhere in the Hadoop cluster. The following is the procedure to achieve it:

1. Stop the DataNode process with the following command:
   ```
   service hadoop stop
   ```
2. Attach the working drive to another DataNode.
3. Mount the disk temporarily onto another location, such as `/mnt/workingdrive`.

4. Copy the version file from another DataNode partition, such as `/hadoop/data/current/version` and overwrite it with the one that is temporarily mounted at `/mnt/workindrive/hadoop/data/current/version`. This is must as each DataNode service does tagging on directory which is identified from the VERSION file. The `VERSION` file contains unique ID for the DataNode that created the directory. As we overwrite the version file, the DataNode will act as if the folder were its own property. The following is an example for the same:

   ```
   cp /hadoop/data/current/VERSION /mnt/workindrive/hadoop/data/
   current/VERSION
   ```

5. The temporary folder needs to be updated in `/etc/sysconfig/hadoop` to list the DataNode partitions in `HADOOP_DATA`:

   ```
   HADOOP_DATA=/hdfs/data
   ```

6. Let us now start the service with the following command, which regenerates the Hadoop configuration files in `/etc/hadoop`:

   ```
   hadoop-firstboot start
   ```

7. Start the DataNode process once again with the following command:

   ```
   service hadoop start
   ```

8. Decommission the DataNode. Edit the `$HADOOP_CONF_DIR/dfs.exclude` file and add the list of DataNode hostnames in NameNode. After this, we can proceed to update NameNode, which forces the NameNode to re-read the excluded file and start the decommissioning process:

   ```
   hadoop dfsadmin -refreshNodes
   ```

9. Once the decommissioning process is completed, we can stop the Hadoop service with the following command to process further for recovery:

   ```
   service hadoop stop
   ```

10. Let us now remove the temporary folder configuration from `/etc/sysconfig/hadoop` and re-execute the service with the help of the following command:

    ```
    hadoop-firstboot stop
    ```

11. Recommission the DataNode by eliminating the relevant entry from `hosts_exclude` and executing the `hadoop dfsadmin` command mentioned next:

    ```
    dfsadmin -refreshNodes
    ```

12. Almost completed now, let us now remove the disk from the DataNode and restart the DataNode service with the help of the following command:

    ```
    service hadoop start
    ```

One of the important considerations here is that the version file contains a unique ID for a DataNode that will be created along with the folder. We need to ensure that the version file on a healthy DataNode does not get removed, and the version file on a working disk is updated before we add a new DataNode.

Let us now move to understand a scenario where files are not recoverable.

Lost files

What would we do if there was no way left to recover a complete file with all the possibilities? We can use the following command with the `-move` parameter, which moves parts of the file to `/lost+found/`:

```
# hadoop fsck -move
```

A better workaround is to have more replicas; it significantly reduces the probability of losing data from the replicas of the file block. As we know that having more replicas is prevented, but having only one set of replicas for the data can only help in saving storage requirements.

As we have now gone through almost all types of data recovery, let us quickly move to the last section of this chapter, which will help us to recover from a NameNode failure.

The recovery of NameNode

NameNode is the one of the most important parts of a Hadoop service as it contains the information of all locations of data blocks in the HDFS cluster; it also maintains the state of the distributed file system. When we come across scenario where NameNode fails, previous checking generated by the secondary NameNode comes to rescue. The checkpoint process takes care of periodic checks on the state of the system. However, having said that, the secondary NameNode cannot be considered a backup alternative for NameNode. The data might be unquestionably old while recovering from the secondary NameNode checkpoint. Although, recovering from the disaster of a NameNode failure containing the older state of the filesystem is way better than recovering nothing at all. Let's now go through the recovery process.

Let us consider that the NameNode service has failed and the secondary NameNode is already in place on a different system. The `fs.checkpoint.dir` property should be set in the `core-default.xml` file. This property helps the secondary NameNode to know where to store checkpoints on the local filesystem of NameNode.

The following steps need to be carried out to recover from a NameNode failure during a disaster:

1. Let us first stop the secondary NameNode:

   ```
   $ cd /path/to/hadoop
   $ bin/hadoop-daemon.sh stop secondarynamenode
   ```

2. A new NameNode needs to be brought up, which would consider itself as the new NameNode. Considering all prerequisites such as the installation and configuration of Hadoop, NameNode configuration, and password-less SSH login are already in place. On top of it, we should configure the same IP address and hostname as those of the failed NameNode.
3. Copy all the data contents of `fs.checkpoint.dir` on the secondary NameNode to the `dfs.name.dir` folder into the new NameNode system.
4. It's time to start a new NameNode system service with the help of the following command:

   ```
   hadmin@ha-namenode01:~$ bin/hadoop-daemon.sh start namenode
   ```

5. Let us also start the secondary NameNode system service on the secondary NameNode system with the help of the following command:

   ```
   hadmin@ha-namenode02:~$ bin/hadoop-daemon.sh start
   secondarynamenode
   ```

6. Time to verify whether NameNode is successfully started by looking at the NameNode status page at `http://ha-namenode01:50070/`.

What did we do just now?

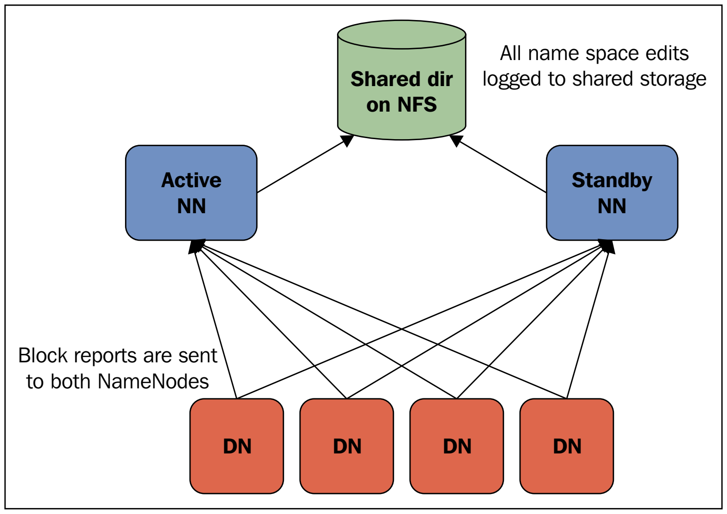
We first stopped the service of the secondary NameNode. Next, we leveraged a system that was a replica in terms of the installation and configuration of NameNode, which had failed. Then, we copied all data for the checkpoint and edited files in the newly configured NameNode. These quick steps helped us to recover the status of the file system, metadata, and edits, from the last checkpoint. After all the relevant configuration and data copying was done, we restarted the new NameNode and the secondary NameNode machine.

Wait! We're not done yet. There is something more to learn.

Recovering data that is older than a certain time might be unacceptable for heavy workloads. What shall we do in such a case? Here is the answer.

One of the alternatives is to set up offsite storage where NameNode can write its image and edit files. By using this method, if we come across hardware failures for the NameNode, we can recover the almost fresh filesystem status without it. This way, if there is a hardware failure of the NameNode, you can recover the latest filesystem without restoring the old data from the secondary NameNode snapshot.

The following image will help you to understand it in a much better way where **NN** represents **NameNode** and **DN** denotes **DataNode**:


Shared dir on NFS
All name space edits logged to shared storage
Active NN
Standby NN
Block reports are sent to both NameNodes
DN
DN
DN
DN

The first step is to have a new machine setup, which can hold the NameNode image and edit file backup. Once we are done with it, we need to mount the backup system on the NameNode server. At the end, we would have to modify the `hdfs-site.xml` file on the server that has NameNode running, which would write to a local filesystem and a backup system-mounted drive:

```
$ cd /path/to/hadoop
$ vi conf/hdfs-site.xml
```

The following modifications have to be made:

```
<property>
<name>dfs.name.dir</name>
<value>/path/to/hadoop/cache/hadoop/dfs, /backup/directory</value>
</property>
```

The NameNode will now start writing all filesystem metadata to the paths mentioned in the value.

Summary

In this chapter, we covered some of the key areas of recovering Hadoop data in various scenarios. A failover on a backup cluster with the detailed steps was one of the key factors. We also covered learning with point-in-time data with the help of snapshot recovery. We dealt with data corruption issues and dealing with files that are not recoverable. Then, we dealt with the recovery from a NameNode failure. These will help us to deal with critical situations. In the next chapter, we will learn how to prevent such issues with the help of monitoring.

7
Monitoring

In the previous chapter, we learnt how to recover from failures. We went through various scenarios detailing the recovery steps along with the setup of failover to backup cluster. We overcame situations of failure such as locating and repairing corruption, lost files, recovering NameNode, and much more. These steps would surely be helpful when things fail.

However, as we progress we surely don't want to be in situations where we need to recover from failures, so let's do monitoring of areas that would help to take proactive actions for failures. Monitoring of multiple areas needs to be covered using methods such as system monitoring, network, clustering, and processes of Hadoop along with dependent services that include MapReduce. In fact, having proper logging mechanism helps a lot in case of failures, this might play a major role in finding the root cause of the failures.

In this chapter, we will cover a few important topics of monitoring, such as:

- Overview of monitoring
- Hadoop metrics
- Node health monitoring
- Cluster monitoring
- Loggings

Before we move on to the details, let's have an overview of monitoring with respect to Hadoop and its importance.

Monitoring overview

Monitoring is an important part of system administration. In this section, we will look at the monitoring facilities in Hadoop and how they can be hooked to external monitoring tools.

Not having an effective system to monitor performance and state of large mission-critical systems is very unlikely. Almost every organization has monitoring tools to dive deep in to data centers and infrastructure. No system, including Hadoop, is designed to run by on its own, however, integration of data, development, and production would be prioritized as compared to integrating monitoring systems that would be the key player once the system is in production.

Monitoring gives a lot of information such as systems, network, applications, and much more with regards to Hadoop; this can help us to know whether we are getting the utmost service response at different levels of a cluster. It might help to monitor few key components such as NameNodes and JobTracker that cannot be ignored when we think about Hadoop monitoring.

Taking proactive measures to have additional systems in the cluster for TaskTrackers and DataNodes, as they tend to fail especially in large clusters with a few unhealthy nodes, can be easily managed. Monitoring can be done via multiple ways; this can be done with the help of a third-party monitoring tool, applications inbuilt commands, or automated scripts, which might help to identify Hadoop clusters' health or their service level.

Monitoring can be bifurcated into two primary components, that is, collection of metrics and subsequent data ingestion. Hadoop would be an addition to the monitoring system to do its job. Subsequent data ingestion would help the monitoring system to provide a dashboard of the metrics collected and alert us in case of probable risks or during actual failures in Hadoop.

Most of the monitoring systems have capabilities to represent data, as expected from monitoring, and sometimes even more that helps to drill down the root cause of the issue or trends in the data center. As we now know that health monitoring is a must, which might consist of and not limited to system monitoring and service health. The other aspect is to set performance metrics that help the system administrator in response time, usage pattern, and lot more. Here, we would be focusing predominantly on monitoring health of Hadoop and dependent systems along with services.

Distributed systems such as Hadoop always have to face challenges due to interaction with multiple services and do collaboration of metrics collected. Let's take one scenario where we generally tend to monitor HDFS daemons; whether they are running or not, if these daemons are using system resources optimally, whether they are responding and processing as they are supposed to do.

Even if we monitor all of these, still we won't know clearly whether our HDFS is fully functional or not. Along with this, if we know that DataNodes are healthy and they are able to reach NameNode, or state of blocks being distributed, then this gives us more confidence that our HDFS is fully functional. Having excessive or too little information flown to the dashboard cannot be relied upon when we depend on the monitoring system to help us get the complete state of health and respective metrics, such as performance or collaborative dashboard.

HDFS monitoring as well has its own challenges to deal with. For instance, if we want to have a holographic view of MapReduce performance on top of HDFS where performance metrics and thresholds are united integrally to HDFS, it becomes difficult to find the root cause of the issue within systems and services borders. Many of the tools such as Ganglia, Dynatrace, and many more are good in finding these problems; however, they require specific details and planning when it comes to distributed systems. Detailing would determine whether the system administrator's life would be easy or not.

As we have now understood the overview of monitoring and its aspects with regards to Hadoop, let's detail it further to know what Hadoop metrics are, relevant configuration, and using current monitoring services that are used in general.

Metrics of Hadoop

HDFS and daemons of MapReduce gather relevant events and capacities that are mutually known as metrics. If we consider DataNode, it would gather metrics such as bytes written, requests from client, sum of blocks that are replicated and so on, would be part of the metrics.

Metrics are a part of the context. Hadoop uses `mapred`, `jvm`, `dfs`, and `rpc` as contexts. Context can be defined as a part of publication; we have the leverage to select which context is to be published. For instance, we can opt to publish the `rpc` context, but not to publish the `mapred` context. Configuration file for metrics is `hadoopmetrics.properties`, contexts would be configured by default to not publish relevant metrics. Following is the glimpse for the default configuration of `hadoopmetrics.properties`:

```
dfs.class=org.apache.hadoop.metrics.spi.NullContext
mapred.class=org.apache.hadoop.metrics.spi.NullContext
```

```
jvm.class=org.apache.hadoop.metrics.spi.NullContext
rpc.class=org.apache.hadoop.metrics.spi.NullContext
```

Over here, we can notice that each line in the configuration specifies the class that handles metrics for the context. The class used should be part of the `MetricsContext` interface and as mentioned in the configuration file, `NullContext` neither pushes updates nor publishes metrics. Rest of the implementation of `MetricsContext` have been covered in the upcoming sections.

Hadoop's raw metrics that are collected can be viewed by accessing to the metrics' web URL. It obviously comes handy. If we want to check JobTracker metrics, it can be checked at `http://jobtrackerhost:50030/metrics` in plain text. If we want to view in JSON format, then we can use JSON switch as `http://jobtrackerhost:50030/metrics?format=json`.

FileContext

`FileContext` takes care of writing metrics to a file. It consists of two properties for configuration.

The `fileName` property is used to specify name of the file to which metrics are pushed, and the other property is a period that is used to specify time interval in seconds for file updates. Both of these properties are optional, and by default metric would be written with an interval of five seconds to standard output. Configuration properties are applied to context name that is specified by updating the property name to its context. For instance, if we want to push the `jvm` context to the log file, we can modify the configuration as mentioned here:

```
jvm.class=org.apache.hadoop.metrics.file.FileContext
jvm.fileName=/opt/metrics_jvm.log
```

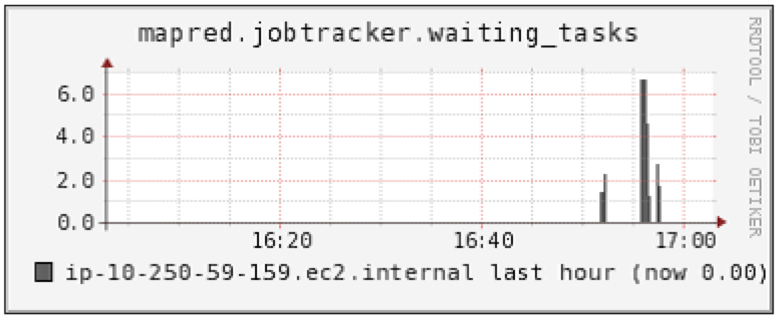
Let's understand what we have done here. First we changed the `jvm` context to use `FileContext`, and in the other line we have configured the property to use the `jvm` context's filename in one of the files.

`FileContext` comes handy to debug local systems, however, when it comes to a large cluster, it's not suitable, as out files would be spread among the cluster and this becomes very difficult to analyze.

GangliaContext

Ganglia (`http://ganglia.info/`) is a scalable open source distributed monitoring system ideal for large clusters and grids. Ganglia has been designed to achieve very low resources overheads and higher concurrency. Ganglia also collects metrics of memory and CPU usage with the help of `GangliaContext` that can be pushed by Hadoop metrics.

`GangliaContext` requires only one property—servers that allows space and/or comma separated list of servers of Ganglia host-port sets. For instance, the kind of metrics we can get out of Ganglia can be seen in the following image that helps us to understand how the tasks in a queue of JobTrackers differ over a period of time:



NullContextWithUpdateThread

`FileContext` and `GangliaContext` both push their metrics to an external monitoring system. When it comes to JMX or some other monitoring systems, they pull metrics from Hadoop. `NullContextWithUpdateThread` is meant for this purpose. As discussed earlier, in case of `NullContext`, it doesn't push metrics, however, it runs periodically to push metrics that are stored in memory.

All the implementation part of `MetricsContext`, apart from `NullContext`, periodically by default pushes metrics in five seconds. And `NullContextWithUpdateThread` also leverages the period property to push the metrics. If we were consuming GangliaContext, then fetching of metrics periodically wouldn't come into place.

CompositeContext

`CompositeContext` would allow pushing of metrics to multiple contexts for the same set of metrics, for example, `GangaliContext` and `FileContext`. Let's look at an example mentioned in the following for better understanding.

`CompositeContext` allows you to output the same set of metrics to multiple contexts, such as a `FileContext` and `GangliaContext`. The configuration is slightly tricky and is best shown by this example:

```
jvm.class=org.apache.hadoop.metrics.spi.CompositeContext
jvm.arity=2
jvm.sub1.class=org.apache.hadoop.metrics.ganglia.GangliaContext
jvm.sub2.class=org.apache.hadoop.metrics.file.FileContext
jvm.fileName=/opt/metrics_jvm.log
jvm.servers=172.16.20.101:8649
```

The `arity` property is supposed to have a number of subcontexts specified; in our example, we have two. Names of the property for each of them are modified with a specific subcontext number, that is, `jvm.sub1.class` and `jvm.sub2.class`.

Java Management Extension

Java Management Extension (**JMX**) is a Java API standard to monitor and manage the applications. Hadoop exposes several metrics to JMX with the help of **managed beans** (**Mbeans**). Mbeans of the `rpc` and `dfs` contexts publishes metrics but not of the `mapred` context or `jvm` context. JVM, however, itself published JVM metrics. JDK, by default, has JConsole tools to view MBeans in a running JVM. It is very helpful to figure out Hadoop metrics details.

Monitoring tools such as Nagios have the ability to query MBeans with the help of JMX, with this it becomes easy to monitor Hadoop cluster from the various monitoring systems that are available. Only thing we need to take care of is to allow remote JMX connections with required security such as SSL connections, SSL client authentication, and password authentication.

Changes required to allow remote JMX connections are to be configured in the `hadoop-env.sh` file. Following configuration has a similar example that has password authentication enabled for now:

```
export HADOOP_NN_OPTS="-Dcom.sun.management.jmxremote -Dcom.sun.
management.jmxremote.password.file=$CONF_ _DIR/jmxpasword.remote
-Dcom.sun.management.jmxremote.ssl=false -Dcom.sun.management.
jmxremote.port=9005 $HADOOP_NN_OPTS"
```

The `jmxpassword.remote` file consists of passwords and relevant users in the format of plain text. With the help of the preceding configuration, we can browse Mbeans of a remote NameNode via remote JMX connection. We can leverage tools available for JMX to pull the values of the MBeans attribute.

Following example of the `check_jmx` plugin that can be integrated with Nagios will show us output of under-replicated blocks:

```
./check_jmx -U service:jmx:rmi:///jndi/rmi://NameNodeHost:9005/
jmxrmi –O hadoop:service=NameNode,name=FSNamesystemState -A
UnderReplicatedBlocks -w 200 -c 2000 –username jmxmonitor -password
jmxpassword
JMX OK - UnderReplicatedBlocks is 0
```

The `check_jmx` plugin will establish a JMX RMI connection with a NameNode host on port 9005 with the credentials provided. It would then read the attribute of `UnderReplicatedBlocks` for the object specified that in our case is `hadoop:service=NameNode,name=FSNamesystemState` and show the value in the console. Two additional switches, `-w` and `-c` , are meant for warning and critical values; right values of warning and critical are generally determined later, once we have the cluster in a fully operational state for some time.

Use of Ganglia along with alerting systems is pretty common in Hadoop cluster. Ganglia is designed to collect various metrics and process them for graphical view, as compared to other monitoring systems, such as Nagios that are ideal to shoot alerts for thresholds defined as per requirement of the service level of each of the metrics in Hadoop cluster.

So far we have understood the aspects of monitoring and various metrics in Hadoop that play a very important role for any successful and minimal downtime cluster. Let's now get in to the details of node health monitoring in the following topics.

Monitoring node health

Once Hadoop is configured with the monitoring system, our next course of action would be to decide which key metrics should be monitored and which to exclude. And to be precise, which key metrics play an important role to know that our Hadoop cluster's health when viewed holistically. When it comes to distributed systems such as Hadoop, dependency of services makes things more complicated as relationship of dependencies within the services has to be taken care of. For example, if HDFS isn't healthy, then alert from MapReduce is not viable to know which services are creating issues.

Selecting metrics is one part to be taken care of, however, another important factor is to set right thresholds for alerts. One of the common problems we come across is, on any system as soon as metrics are collected, they get outdated instantly. A disk consisting of metadata for the NameNode is 95 percent used out of 2 TB, this means that we still have around 102 GB of space available.

To keep an eye on total disk size is obviously a factor to be considered, however, if we aren't aware of growth rate that might be on an average of 2 GB per day, then having 102 GB should be on the safer side and relevant alerts shouldn't be enforced. When we calculate growth rate, we also need to consider archived data. Relevant archival of data along with log rotation needs to be planned, so that use of resources are done optimally for better performance and maintenance.

As there would be various scenarios that need to be thought through, we would prefer to follow few simple guidelines, as follows:

- Thresholds are meant to change. Usage of resources and cluster would vary and accordingly thresholds should also be up to date.
- Initially having generic thresholds are good to start with, however, after a certain time we should revamp them.
- Relevant alerts play key factor in any monitoring system. Unwanted alerts create negative impression on the systems team regarding the importance of the incident that results in weak monitoring and response from the systems team.

Hadoop host monitoring

Here we will see which parameters we need to cover for Hadoop cluster host monitoring. Common monitoring parameters such as memory utilization, disk capacity, network throughput and CPU utilization are essentials of any monitoring systems. Almost all Hadoop services use local disks almost everywhere. Many daemons have not been considered to handle draining capacity for storage.

For instance, NameNode corruption of metadata when storage capacity reaches its peak where edit log gets stored has been considered a minor issue. There is no doubt that Hadoop core engine has gone through lot of improvements to handle such scenarios, however, neither has this been eliminated completely nor would we prefer to take a chance, would you?

DataNode directories are the heart of Hadoop when we talk about data, as it stores large sets of data always. If we come across disk capacity reaching to its peak level, it would stop storing further blocks in it. When not taken care of at the right moment and situation goes beyond our control where all NameNode systems disk capacity is full, NameNode will immediately halt pushing requests for new blocks on the DataNode resulting in the DataNode to be considered as read-only.

If we plan to have MapReduce data and DataNode on the same system, we should take care of `dfs.datanode.du` that is reserved to store temporary data required to run tasks. On the other hand, MapReduce data utilization would be hard to predict resulting in tackle threshold alerts on estimates. Data residing on these directories would be cleaned up once jobs are completed, whether it fails to complete or completes successfully. Volumes that are backed up with `mapred.local.dir` would be considered as unavailable if disk capacity is at its full capacity or there is drive failure, TaskTracker would be on standby for the system. Jobs assigned during this period to TaskTracker will be ultimately barred by it.

Memory utilization has two aspects; virtual and physical memory. Heap memory allocation has to be distributed properly so it doesn't sum up to cause over burn of the total physical memory in the system. Considering that we have configured heap memory size properly in Hadoop cluster, we shouldn't come across memory swapping. If we notice memory swapping in the system, it can directly affect performance that will cause failures of jobs, processes, and much more.

CPU utilization is one of the other most commonly used metrics to monitor systems. However, there is a misconception when the CPU load average is monitored while using Linux systems. Most of the people have the feeling that load average is: numbers representing an average gradually over a period of time, that is, three numbers being represented in an average duration in minutes of one, five, and fifteen, and lower the number better it is. If it's on higher side, it tends to be an overhead on CPU utilization resulting in higher load average numbers. So next question that comes to our mind is which number should we observe, is it one, five, or fifteen minutes? Ideally, looking at five or fifteen minutes average is good.

To dive deeper, let's also understand what these numbers represent when we have a multi-core CPU or multiple CPUs in a system. In this case, load average interpretation would change. For instance, if we have a load average of 2 on a single CPU, it means our system is fully loaded that means one of the processes was utilizing the entire capacity and the other one was waiting. However, on a system with two CPUs, two different processes used different CPUs during this period. While considering the same scenario with four CPUs, two processes were utilizing two CPUs capacities while other two CPUs were not utilized at all.

To understand the load average number, you need to know how many CPUs your system has. A load average of 6.03 would indicate that a system with a single CPU was massively overloaded, but it would be fine on a computer with 8 CPUs. To conclude, on load average on a large cluster of Hadoop, it's not preferable to set thresholds as this may give us false alarms on system utilization.

Network utilization is almost same as compared to CPU utilization when we talk about monitoring. Network utilization monitoring gives us a better insight into how data is being pushed and pulled out of the system, though it does not give a specified limit to which we can put a threshold for us to get an alert. Network performance degradation, such as high latency time, may cause some jobs to fail that preliminarily affects HBase, where specific set of servers of a section having high latency or unreachable for certain period from ZooKeeper quorum, might also quickly cause major failure due to dependencies. For this, RPC metrics is useful for providing a much better view of each daemon's latency, so this would be ideal to monitor.

Hadoop host monitoring is certainly important to determine some of the aspects to quantify health. But as said earlier, being a distributed system monitoring of a Hadoop process cannot be relied upon to judge whether cluster on a whole is healthy or not. It's likely for a process to be in a running state but not communicating with other systems in the cluster for any reason; hence, a false alarm for us.

Hadoop process monitoring

As Hadoop processes are based on Java, there would be certain set of metrics that should be noted for monitoring. Heap memory usage is one of these important ones essentially in JobTracker and NameNode. Within JMX, it's easy to find heap memory information.

However, to determine process health is not straightforward from the numbers fetched based on the total memory subtracted by heap memory usage, as garbage collection plays an integral part. Used memory would be always on the lower side or similar to the memory committed for heap committed. Committed memory is the size allocated to the heap that is reserved for the system for the Hadoop process that is dedicated to be used if objects need them. Ideally, it's good to keep committed memory based on the pattern derived that determines normal memory usage in general.

If a process tries to go beyond the committed memory and is not able to reach further due to maximum memory, then this process might throw `Out of Memory` error. Following is the sample JVM memory metric fetched in the JSON format:

```
{
"name" : "java.lang:type=Memory",
"modelerType" : "sun.management.MemoryImpl",
"Verbose" : false,
"HeapMemoryUsage" : {
"committed" : 1035879412,
"init" : 1482712232,
"max" : 7534433624,
"used" : 429256722
},
"NonHeapMemoryUsage" : {
"committed" : 45842281,
"init" : 22353657,
"max" : 146617890,
"used" : 46282012
},
"ObjectPendingFinalizationCount" : 0
}
```

We would come across situations where memory allocation has to be bumped up as time goes by. However, the thing that needs to be taken care of is to keep it almost committed or to the maximum memory allocated, and then garbage collection does its job for normal usage.

The HDFS checks

Let's go back a few chapters and look at the HDFS architecture and its design. We saw that the NameNode basically tracks all the DataNodes and handles failures accordingly. If, for some reason, the NameNode becomes unavailable, it's not possible to continue further. The clients are going to see what the NameNode is going to see. The NameNode is the point of access for the clients to see the filesystem. Now to monitor HDFS, this design comes handy and saves us from the difficult task of relating the events and state information of the DataNodes, as it will be available in the NameNode. This will leave no room for even the slightest error in the health checks.

Let's look at some of the metrics available. Following are sample metrics in JSON format showing node information such as alive nodes, dead nodes, and more:

```
{
"name" : "Hadoop:service=NameNode,name=NameNodeInfo",
"modelerType" : "org.apache.hadoop.hdfs.server.namenode.FSNamesystem",
"Threads" : 55,
"Total" : 193868941611008,
"ClusterId" : "CID-908b11f7-c752-4753-8fbc-3d209b3088ff",
"BlockPoolId" : "BP-875753299-172.29.121.132-1340735471649",
"Used" : 50017852084224,
"Version" : "2.0.0-cdh4.0.0,
r5d678f6bb1f2bc49e2287dd69ac41d7232fc9cdc",
"PercentUsed" : 25.799828,
"PercentRemaining" : 68.18899,
"Free" : 132197265809408,
"Safemode" : "",
"UpgradeFinalized" : false,
"NonDfsUsedSpace" : 11653823717376,
"BlockPoolUsedSpace" : 50017852084224,
"PercentBlockPoolUsed" : 25.799828,
"TotalBlocks" : 179218,
"TotalFiles" : 41219,
"NumberOfMissingBlocks" : 4725,
"LiveNodes" : "{
\"hadoop02.cloudera.com\":{
\"numBlocks\":40312,
\"usedSpace\":5589478342656,
\"lastContact\":2,
\"capacity\":21540992634880,
\"nonDfsUsedSpace\":1287876358144,
\"adminState\":\"In Service\"
},
"DeadNodes" : "{}",
"DecomNodes" : "{}",
"NameDirStatuses" : "{
\"failed\":{},
\"active\":{
\"/data/2/hadoop/dfs/nn\":\"IMAGE_AND_EDITS\",
\"/data/1/hadoop/dfs/nn\":\"IMAGE_AND_EDITS\"
}
}"
}
```

Some fields in JSON are string that contain JSON blobs, such as the live nodes.

Now from this data, perform the following health checks:

The free HDFS capacity amount as seen in JSON here (21540992634880 bytes, it's generally in bytes) should be over a satisfactory threshold value.

The total number of active meta data paths (which can be accessed via `NameDirStatuses["active"]`) is equal to those specified in `dfs.name.dir` (which can found in `hdfs-site.xml`) and the failed metadata paths (`NameDirStatuses["failed"]`) is equal to zero.

The next level of metrics is going to involve the filesystem. Here is a sample JSON format file showing some metrics:

```
{
"name" : "Hadoop:service=NameNode,name=FSNamesystem",
"modelerType" : "FSNamesystem",
"tag.Context" : "dfs",
"tag.HAState" : "active",
"tag.Hostname" : "hadoop01.cloudera.com",
"MissingBlocks" : 4725,
"ExpiredHeartbeats" : 2,
"TransactionsSinceLastCheckpoint" : 58476,
"TransactionsSinceLastLogRoll" : 7,
"LastWrittenTransactionId" : 58477,
"LastCheckpointTime" : 1340735472996,
"CapacityTotalGB" : 180555.0,
"CapacityUsedGB" : 46583.0,
"CapacityRemainingGB" : 123118.0,
"TotalLoad" : 9,
"BlocksTotal" : 179218,
"FilesTotal" : 41219,
"PendingReplicationBlocks" : 0,
"UnderReplicatedBlocks" : 4736,
"CorruptBlocks" : 0,
"ScheduledReplicationBlocks" : 0,
"PendingDeletionBlocks" : 0,
"ExcessBlocks" : 0,
"PostponedMisreplicatedBlocks" : 0,
"PendingDataNodeMessageCount" : 0,
"MillisSinceLastLoadedEdits" : 0,
"BlockCapacity" : 8388608,
"TotalFiles" : 41219
}
```

Information such as capacity total (one is in bytes and the other is in GB) are common in both. All of the information is computed using the same snapshot and hence should be consistent with each other.

Points to keep an eye on are as follows:

- The number of missing blocks and/or corrupt blocks should be ideally zero, which is the best scenario. Even if this is not the case, make sure that they don't exceed beyond a certain threshold value.
- Keep an eye on the block capacity. This is basically the maximum number of blocks NameNode can write before it refuses to write any new files. This is to prevent memory leakage issues due to over allocation or less heap size. Increasing the heap size will automatically increase this size.
- You perform checks every 3 days. You can get that value with the result of current epochs minus the last time a NameNode checkpoint was performed.

The MapReduce checks

There are two major components to monitor the MapReduce:

- Tracking the framework
- Tracking all the individual jobs

Tracking all individual frameworks can be too application-specific, and would be a poor choice as it won't contain the entire framework, so we will track the entire framework instead.

Let's see some of the available metrics on which to keep an eye on.

To review all the metrics, simply go to `http://jobtracker-host:50030/metrics`. To review the metrics in JSON format, append a query parameter to the query. Simply add the `?format=json` parameter.

Let's look at one of the sample metrics generated in JSON:

```
{
"name" : "hadoop:service=JobTracker,name=JobTrackerInfo",
"modelerType" : "org.apache.hadoop.mapred.JobTracker",
"Hostname" : "m0507",
"Version" : "2.0.0-mr1-cdh4.0.1, rUnknown",
"ConfigVersion" : "default",
"ThreadCount" : 45,
"SummaryJson" : "{
\"nodes\":8,
\"alive\":8,
\"blacklisted\":0,
\"slots\":{
\"map_slots\":128,
\"map_slots_used\":128,
\"reduce_slots\":48,
\"reduce_slots_used\":0
},
\"jobs\":3
}",
"AliveNodesInfoJson" : "[
{
\"hostname\":\"hadoop01.cloudera.com\",
\"last_seen\":1343782287934,
\"health\":\"OK\",
\"slots\":{
\"map_slots\":16,
\"map_slots_used\":16,
\"reduce_slots\":6,
\"reduce_slots_used\":0
},
\"failures\":0,
\"dir_failures\":0
},
// Remaining hosts omitted...
]",
"BlacklistedNodesInfoJson" : "[]",
"QueueInfoJson" : "{\"default\":{\"info\":\"N/A\"}}"
}
```

This is provided in the form of JSON blob. We will keep an eye on the following parameters:

- Number of nodes currently alive
- Total map slots available
- Total map slots used
- Total job submissions
- Blacklisted nodes info

A blacklisted TaskTracker is one that is alive but repeatedly fails.

A few words of advice:

- A frequent check on whether the live nodes number is within a limit that allows jobs to be completed within the service level predefined time. This number is directly proportional to the size of the cluster and the criticality and severity of the jobs.
- Keep an eye on the blacklisted TaskTrackers. They should be below some percentage of the total number of the TaskTrackers in the whole cluster. This is to ensure that the failover ratio is below some percentage. Blacklisted TaskTrackers basically should be zero. But as a matter of fact, we can trigger false positives if a series of rapidly failing jobs are submitted. This number basically is an indicative that something is wrong somewhere.

As we can see in JSON, it reports all of its workers (active and blacklisted) along with all of their respective statuses.

Monitor the difference or deviation of the TaskTracker's number of failures from the average of other TaskTrackers. If we over monitor the individual TaskTrackers, we will be spending all our resources and it's not unusual to have common failures on this scale.

Cluster monitoring

A cluster is composed of a HDFS cluster and MapReduce cluster:

- A HDFS cluster basically consists of filesystem, one or more NameNodes that keep track of everything through the metadata information (as we have seen earlier), while the actual data is on distributed DataNodes.
- A MapReduce cluster consists of one master, that is, the JobTracker daemons, and other slaves that are the TaskTrackers. The JobTracker manages the lifecycle of the MapReduce jobs. It splits jobs in to smaller tasks and schedules the tasks to be run by the TaskTrackers.

So we can divide monitoring a Hadoop cluster into two main jobs: HDFS monitoring and MapReduce monitoring.

Managing the HDFS cluster

You can monitor a cluster using an `fsck` filesystem checking utility. Fsck basically takes in a file path as an argument and it will go on to check the health of all the files under this path recursively, and if you want to check the entire filesystem just call it with `/`.

To monitor an HDFS cluster, check the status of the root filesystem with the following command:

```
$ bin/hadoop fsck/
FSCK started by hduser from /10.157.165.25 for path / at Thu Feb
28 17:14:11 EST 2014.
/user/hduser/.staging/job_201302281211_0003/job.jar: Under replicated
blk_-665248265064328578_1016. Target Replicas is 10 but found 5
replica(s).
.................................Status: HEALTHY
Total size: 143108109768 B
Total dirs: 9726
Total files: 41532
Total blocks (validated): 42519 (avg. block size 3373632 B)
Minimally replicated blocks: 42419 (100.0 %)
Over-replicated blocks: 0 (0.0 %)
Under-replicated blocks: 0 (0.0 %)
Mis-replicated blocks: 0 (0.0 %)
Default replication factor: 3
Average block replication: 3.0
Corrupt blocks: 0
Missing replicas: 0 (0.0 %)
Number of data-nodes: 16
Number of racks: 2
```

The following are the parameters to be added along with `fsck`:

- `-files`: The `-files` option tells `fsck` to print out file information. For each file, it checks the file's path size and status.
- `-blocks`: The `-blocks` option goes one level further to tell to print out information about all the blocks. It will give information about the blocks' name, length, and number of replicas. So running this command would mean that `-files` has to be also included.
- `-locations`: This would include all the information about the replicas of the blocks. Running this command would mean that `-files` and `-blocks` have to be also included in the parameters lists.
- `-racks`: This would include all the rack names as location information. This would mean that when running this command `-files`, `-blocks`, and `-locations` should be included in each of the parameters list.

Have a look at the following example:

```
bin/hadoop fsck /user/hadoop/test -files -blocks -locations -racks
```

Get the status report of all the DataNodes with the following command. Use the `-report` option on the `dfsadmin` command:

```
#hadoop dfsadmin -report
Configured Capacity: 400GB
Present Capacity: 371.82 GB
DFS Remaining:351.47 GB
DFS Used: 5.35 GB
DFS Used%: 3%
```

```
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0
-------------------------------------------------
Datanodes available: 5 (5 total, 0 dead)
Name: 10.145.223.184:50010
Decommission Status : Normal
Configured Capacity: 88.75 GB
DFS Used: 1.17 GB
Non DFS Used: 3.4 GB
DFS Remaining: 62.18 GB
DFS Used%: 2.75%
DFS Remaining%: 91.65%
Last contact: Thu Feb 28 20:30:11 EST 2013
```

Refresh all the DataNodes using the following command:

```
hadoop dfsadmin -refreshNodes
```

Save the metadata file if needed as follows:

```
hadoop dfsadmin -metasave meta.log
```

The log file would be saved in `$HADOOP_HOME/logs`.

Many monitoring frameworks are available. Some of them are Ganglia and Nagios. Both are open source frameworks.

Logging

To understand what's happening in the system, the log files can be very helpful. All Hadoop daemons make log files.

Let's start by enabling system log files.

When you first installing Hadoop, the system log files are basically stored in `$HADOOP_INSTALL/logs` directory, which is configurable by changing the `HADOOP_LOG_DIR` variable in `hadoop-env.sh`. It's better to have logs in one place, so it is highly recommended to change the storage path. Just change the following path:

```
export HADOOP_LOG_DIR=/my/path/goes/here
```

[Make sure that the user has permission to write to this file.]

Now each Hadoop daemon is basically going to produce two log files. One that ends with `.log` is the primary one, this logs almost all application messages; and second, which ends with `.out`, has almost no output apart from daemon initializing and shutting down messages.

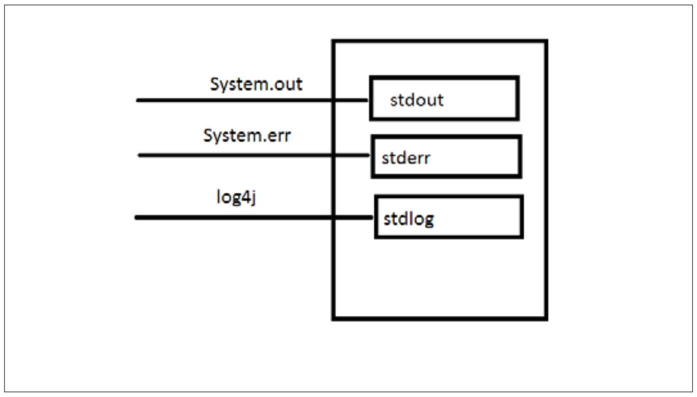
Log output written via log4j

This is the first file to look at in case of problems, as most of the application logs are written here. Now `log4j` is configured in such way that old log files are never deleted. So we have to manually arrange them to get them deleted, or soon we will encounter memory shortage issues. This file ends with `.log`. Have a look at the following code snippet:

```
Std output + std error
```

This file practically contains any output. Once daemon restarts, this file is rotated and only last 5 logs are retained. Old log files are suffixed with a number from 1 to 5, 5 being the oldest log file.

This can be understood with the following figure:



You can generally find the name of the user running the daemon and date in the name of the log file. The `.log` and the `.out` files are generally constructed using the following:

`Hadoop-<user-running-hadoop>-<daemon>-<hostname>.log`

For example, `hadoop-parth-datanode-ip-sturges.local.log.2015-01-01` is the name of the file after it has been rotated.

Now having a date file is what makes it achievable. If you want to change the user name, change `HADOOP_INTENT_STRING` in the `log4j.properties` to whatever identifier you want to use.

Setting the log levels

Enabling debug mode in Hadoop opens the door to conveniently solving a problem by watching the logs of what went wrong. It is very suitable to change log level temporarily for a particular component in the whole system. A webpage can be found for any Hadoop daemons to change the log level (debug, error, and so on) of any `log4j` log name, which is present at `/loglevel` in the `webUI` daemons. By custom, the log names basically correspond to `ClassName`, where the logging is done. But there may be some exceptions, so it's always better to consult the source code to look out for the log names.

Let's start by enabling logging for the JobTracker class. Just visit `http://jobtracker-host:50030/logLevel`.

Set the log named `org.apache.hadoop.mapred.JobTracker` to `DEBUG` level.

Use the following command:

```
% hadoop daemonlog -setlevel jobtracker-host:50030
org.apache.hadoop.mapred.JobTracker DEBUG
```

These changes are temporary. Everything will be reset once the daemon restarts. It will read only those properties that are `log4j.properties`.

To make a permanent change, just go to `log4j.properties` and change.

For example, `log4j.logger.org.apache.hadoop.mapred.JobTracker=DEBUG` will simply make the debug level mode permanent.

Getting stack traces

Hadoop provides developers a comfortable web page to play around. Hadoop daemons provide a web page that produces a thread dump of all the threads that are running in the current daemons JVM. It basically shows all the `stdout` and `stderr` on `WebUI`. The stack trace is printed on diagnostics that can be seen on demand. Also, if you started your tasks manually, they are going to be in `stdout` instead of the log files. There would be a stack trace with exception even if it didn't show up in the JobTracker page.

This would be generally under the following:

```
/usr/lib/hadoop-0.20/logs/userlogs//
```

Also, to access the web UI just hit `/stacks`. Following is an example to get a thread dump for JobTracker hit:

```
http://jobtracker-host:50030/stacks
Process Thread Dump:
43 active threads
Thread 3603101 (IPC Client (47) connection to ha-namenode02:9000:
State: TIMED_WAITING
Blocked count: 7
Waited count: 8
Stack:
  java.lang.Object.wait(Native Method)
  org.apache.hadoop.ipc.Client$Connection.waitForWork(Client.java:276)
  org.apache.hadoop.ipc.Client$Connection.run(Client.java:917)
```

Summary

In this chapter we went through the overview of monitoring. We also understood Hadoop metrics in detail that would help us to know which metrics are to be leveraged to get monitoring done right and accordingly what proactive actions are to be taken care of.

After going through node health and Hadoop process monitoring, we also covered HDFS and MapReduce checks that would help administrators to know the cause of the issue in case of failures. Obviously having all these without logging doesn't makes sense; this we understood pretty well in detail along with examples that gives us more confidence while identifying the cause of the issue.

Let us now head towards the last phase of the book about troubleshooting strategies and well-defined techniques.

8
Troubleshooting

So far in this book, we have learnt various topics associated with Hadoop backup and recovery, including Hadoop and clustering basics, understanding the needs of backup and recovery, backup strategies and ways to backup Hadoop, strategies to recover Hadoop and recovering Hadoop, and finally, monitoring the node health in Hadoop. *Chapter 7*, *Monitoring*, discussed the importance of monitoring and the different areas to be covered under monitoring, including host-level checks, Hadoop process monitoring, HDFS checks, and MapReduce checks and cluster monitoring.

While monitoring gives important information about the state of all the resources in Hadoop, it is equally important to understand what should be done in the case of a failure. As you know, Hadoop is a distributed system and there are multiple layers of services involved in the system. Having all these layers interact with one another makes the system more complex and more difficult to troubleshoot.

In this chapter, we will discuss Hadoop troubleshooting. We will discuss an approach to troubleshoot a Hadoop failure; various probable causes of the failure, such as human errors, issues with configuration, hardware issues, resource fatigue, and techniques to resolve these failures. Some of the very common techniques to resolve these failures are as follows:

- By configuration
- By architecture
- By process

We will discuss all the common failure points and their mitigation strategies in this chapter.

Understanding troubleshooting approaches

In order to solve any problem, we have to first understand the problem and the cause of the problem. Many problems are not getting solved because of an improper analysis of the problem. Think of an incident when you fell sick and went to see the doctor. A good doctor will first ask the history of your illness and will see all the symptoms; on the basis of the symptoms, he will figure out a few probable causes of the sickness and will ask you to undergo a few tests, and finally, on the basis of the results of the test, he will do his final diagnosis. Just read the last few lines again and think of what he has done and try to associate it with any other system that has an issue or a problem.

In order to solve any problem, it is very important that you do a proper diagnosis of the issue and identify the root cause of the issue. Once the root cause is found, it becomes easy to fix it and resolve the problem. Hadoop system failures are not an exception to this approach. You need to follow the approach mentioned as follows to find out the probable root cause of the issue in order to troubleshoot a Hadoop failure (or any other system failure):

1. Find out the history of the system. Understand if there any changes done to the system; it could be a configuration change, maintenance change, or a hardware change. It is also possible that a recent change could have caused the issue. Further, understand the history of the system to know if such an issue has occurred in the past. If so, find out the cause of the issue and the resolution.
2. Once you know the recent changes and history, try to list down all probable causes that you think can cause the failure or issue. Further, the history will help you to figure out the probability of each issue. You need to understand the environment that you are in to derive the probability of the issue.
3. After listing down the probable causes, run a few tests (remember how the doctor asks you to go undergo certain tests to determine the illness) to determine the root cause of the issue.
4. Once you find the root cause, fix it!

In any system, log files play an important part in troubleshooting. The system communicates with humans via log files. It is very important that you always go through the log files to understand the issue while going through the approach mentioned in this section.

At this point, you may be thinking what's new in this approach. This is something that we already know since childhood. However, remember that this basic approach will remain the same in the case of troubleshooting any issue. It is important to not just know it but also know how to apply it when needed. The purpose of this section was to remind you of the basics of troubleshooting.

Now, as you know the troubleshooting approach, it is important to know a few common points of the failure. These points will help you to list down the probable causes of the failure. So, let's get familiar with them.

Understanding common failure points

As discussed at the beginning of this chapter, Hadoop is a distributed system having multiple layered services; all these services communicate with one another. Further, there are multiple daemons in the host and multiple hosts in the cluster. The services can be dependent on each other in this distributed environment.

The more complex the system is, the larger the number of things that can go wrong. With a complex system, there can be plenty of things that can go wrong in Hadoop too. Being an administrator, you need to know the various factors that can cause a system failure. They can also help you to list down all the possible causes of the failures as discussed in the previous section. Here, we will discuss some of the very common failure points of the Hadoop system.

Human errors

One of the most common factors of any failure or disaster in the world is human error. We human beings tend to make mistakes, and these mistakes can cause failures. You, being an administrator, would also have had such an experience when a very small (or silly) mistake would have taken the entire system down. Sometimes, while doing routine tasks, we make small mistakes causing big issues. This factor is not specific to Hadoop. It is more of a general issue. No matter how careful we are, human errors will happen. As you cannot avoid this, it is very important to know the probability of human error while troubleshooting.

Configuration issues

In one of the presentations named *Hadoop Troubleshooting 101*, *Kate Ting*, Customer Operation Engineer at Cloudera, provided an interesting breakdown of the tickets that she and her team received over a period of time. She mentioned that out of all the tickets she received, 35 percent of the tickets were due to some kind of misconfiguration like resource allocation. According to her, misconfiguration was any diagnostic ticket that required a change to Hadoop or to the OS configuration files. Surprisingly, it took 40 percent of the overall time to fix the tickets having configuration issues.

The configuration issues happen only when the users of the system tweak the configuration parameters randomly, which cause the unintended behavior and failure of the system. Some of the very common misconfiguration issues can occur in the following scenarios:

- Memory mismanagement
- Thread mismanagement
- Disk mismanagement

Being an administrator, you may argue that it is very important to tweak the parameters in order to achieve the optimum performance. However, at the same time, it is very important to understand that tweaking the parameters without enough knowledge and experience can lead to system failures. If one changes multiple parameters at a time and causes a system failure, it becomes extremely difficult to find the parameter that caused the failure (which can be a combination of multiple parameters sometimes). Please ensure that you use the following tips to avoid misconfiguration issues:

- Develop in-depth knowledge of the configuration parameters. Before changing any parameter, ensure that you have good knowledge of what it does. If you don't know, first get information on the same before modifying it.
- Whenever you change any parameter, ensure that you are using the right unit of the parameter while making the change.
- Implement a proper tracking and auditing system to know the parameter change.
- Ensure that you have a proper configuration management system to make sure that all the files are up-to-date.
- Ensure that you run the required performance test after making a configuration change. This will help you to eliminate any performance issues due to the parameter change.

Along with all the configuration parameter changes, you also have to ensure that the operating system configuration is done properly. It is always recommended to automate cluster configuration to avoid any accidental misses due to manual intervention.

[One of the special types of misconfigurations is host identification and naming. The way that a worker host identifies itself to the NameNode and JobTracker is the same way clients will attempt to contact it. This leads to interesting types of failures. Here, a DataNode, due to a misconfigured `/etc/hosts` entry, reports the IP address of its loopback device to NameNode and successfully heartbeats, only to create a situation in which clients can never communicate with it. These types of problems commonly occur at the initial cluster setup time.]

Hardware failures

This is one of the issues that cannot be avoided at all. Since we have started using hardware, we have seen them failing at certain times. Depending on the type and quality of the hardware, the time varies but it fails for sure. It could be the memory, motherboard, processor, or hard drive. Out of all the hardware, the hard drive is one of the major failure points. It suffers from physical wear and tear during operations and fails at a certain stage. One of the major issues with the hardware failure is that they don't fail outright. The performance of the hardware reduces over a period of time with temporary failures. HDFS detects the corrupt data blocks and automatically creates new blocks without human intervention. This feature is not a solution to the hardware failure. To mitigate the hardware failure issue, you always need to have spare components ready for the critical servers. Further, you should observe a pattern in the corrupt HDFS blocks in order to mitigate such issues.

Resource allocation issues

While doing Hadoop administration, one of the very common questions we may get is, how many map slots and reduce slots should be allocated to a given machine? This may seem like a very simple question to answer, but the right value may depend on multiple factors such as the CPU power, disk capacity, network speed, application design, and the processes sharing the node. As you know, the CPU cycle, memory, disk space, and so on, are all fixed resources. All the processes in a Hadoop cluster contend for these resources. As an administrator, it is your job to ensure that you allocate the right resources by the configuration. This category is very similar to misconfiguration. If you make any mistakes in the configuration for resource allocation, it can go wrong and cause a system failure. You should always measure the failures in the system to identify the misbehaving processes. You need to ensure that resource allocation is properly done in order to meet the required throughput and SLAs.

Identifying the root cause

Have you ever been in a situation (or heard of one) when you called in to complain about the Internet not working and the person on the other side of the phone asks a question, "Is your modem plugged in?" You may get angry in this situation thinking, "I'm not that dumb," but believe or not, many of the calls are made because someone has not plugged in or turned the modem on. The technical support team has a list of questions that they normally go through in order to eliminate the common causes of the issue to identify the one to fix.

Although there is no such fixed, simple list of questions to follow in order to identify the root cause of a Hadoop failure, there are certainly a list of areas that you should go through in order to confirm or reject the probable cause of the failure. As discussed, in the *Understanding troubleshooting approaches* section earlier in this chapter, once you list down all the probable causes of a failure, you need to perform tests to determine the root cause of the failure. This could be a checklist that may be helpful for you to remember each part when you are troubleshooting:

- **Environment**: You need to carefully look at the environment that you're facing issues in. Do you find anything unusual or sense any obvious issue? You may look at the indicators that can give you a hint about the issue. See what has changed since you last saw the working system and now. The findings can help you to identify or nail down the failure.
- **Pattern**: Do you find any particular pattern in the failure? Do all the tasks that are failing belong to the same job? Is there any particular code/library that is involved in the failure? If you can derive the pattern, it would be easy for you to reach to the root cause of the failure.
- **Logs**: This is one of the very obvious ways to identify the root cause. For any system, log files play an important role in troubleshooting. You should look not only at the errors but also at the exceptions. You need to carefully review the exceptions that you get in the log files. They can reveal the root cause of the issue that you're facing. You need to have a practice of reviewing the logs of the component that fails or behaves inappropriately.
- **Resources**: As one of the common failure points is resources, you need to keep an eye on the resource utilization. As discussed earlier, all the resources are shared among different processes in Hadoop, and resource fatigue is very common. Whenever you're identifying the root cause of the issue, look into the resource utilization, which can sometimes reveal an interesting angle of the issue.

On top of all these areas, try looking for events that might have caused the issue. As discussed at the beginning of this chapter, the system history helps you to understand the probable cause of this issue. Sometimes, looking at the history and analyzing the events occurred before/during the failure can also help us to identify the root cause of the failure. The mentioned list can help you to take a systematic approach to identify the root cause. Some of the items can be easy and straightforward to follow, such as identifying resource utilization or looking into the environment, whereas the other items such as the log file review and pattern identification need in-depth knowledge and experience of the system. You need to know which one to use when, and what kind of details that it can provide you.

Knowing issue resolution techniques

So far, we have discussed an approach to troubleshooting the issue. Ideally, we first need to understand the problem, identify the root cause of the issue, and then take corrective measures to fix the issue. However, it is very common that rather than following this approach, one directly jumps into the problem and start taking different actions hoping one of them will fix the issue. Sometimes, the issue may be fixed temporarily, but the problem may resurface or the steps that we took may lead to a new problem. In this scenario, we may end up spending more time and effort taking a series of steps to fix the issue again.

There are multiple ways and approaches to solve a problem. The ideal scenario, as we discussed is to take an approach similar to what doctors take—list down the probable causes, perform certain tests to diagnose the problem, identify the root cause, and then cure it permanently. This is an ideal approach, but it may not be possible in all scenarios. This section will help you to understand the different approaches that we can take to fix the problem:

- **Permanent fix**: As discussed earlier in the section, this is an ideal scenario where we find the root cause of the issue and fix it permanently. However, this happens very rarely. This normally takes place when we have some kind of hardware failure or disk space issue. Let's say if we identify that one of the hard drives failed, we can replace it and fix it. This kind of fix would be easy to implement.

- **Mitigation by configuration**: This kind of fix is used as an intermediate solution to the issue until the root cause is identified and fixed permanently. Here, the root cause of the issue may be unknown, or it may take a long time to fix the issue. Sometimes it is also very risky to fix the issue immediately without carrying out a proper analysis. In these scenarios, we go with a temporary fix until we either identify the root cause or get enough time to fix the issue. Let's say someone has written code that is using excessive memory and we are getting an out of memory error while executing a certain piece of code. Ideally, we need to perform a proper code review and fix the issue in the code. However, this could take a long time, so we may temporarily increase the memory to fix the issue and then work on the code to take care of it permanently.
- **Mitigation by architecture**: Sometimes, it is better to make architectural changes rather than changing the code or fixing it through the configuration. This approach can fix the problem permanently and can improve the overall performance of the system. Let's say that there is a system where we are doing data processing and displaying the data to the end user. The data is processed and stored in the database all the time. Every time a user wants to access the data, it takes it from the database. Having concurrent users degrades the system performance. In this case, we cannot block the users from accessing the data. However, we need to make a decision to implement a caching layer in order to store the data for the access. Further, at a certain interval or event (such as data update), the cache should be updated. For this kind of solution, we usually need to involve the architect and the development team.
- **Mitigation by process**: This is one of the best ways to resolve the issue. Many times issues occur due to improper processes or the absence of a process. Remember the issue that we discussed in the *Mitigation by configuration* part about a code using excessive memory? Here, fixing the issue by configuration or changing the code could be fine; however, it could be avoided if we have proper code review processes in place. Similarly, if we have a proper change management process, it can help us to avoid issues that can occur due to a change in the configuration files or it would be easy to identify and fix the issues if the changes are well documented. Mitigation by processes would be one of the best options for fixing or avoiding such issues.

Here, we have gone through few approaches to fixing an issue. As discussed, we should list down the probable causes or the issues, perform tests to find the root cause, and then fix the issue. However, once the issue is fixed, it is very important to document the issue and its resolution. It is all about learning from mistakes. Once you carry out a proper analysis of the issue, factors that caused the issue, and the resolution that you provided, always make a habit of documenting it. This can be helpful to other members in the organization, and many times to you, as well. Sometimes, people avoid doing such an analysis just because they may think that they will be blamed for the issue. However, as an administrator of the organization, you are also responsible for ensuring that such an analysis or meeting does not end up in crucifying someone for the issue but gets the reason behind the issue and ensures that appropriate actions are taken to avoid such an issue in the future.

Summary

In this chapter, we discussed an approach that we should take for troubleshooting. We learnt about troubleshooting techniques, areas to look into when an issue occurs, and ways to fix the issues. These are very general troubleshooting techniques that you can apply not only to Hadoop, but also to other systems.

Index