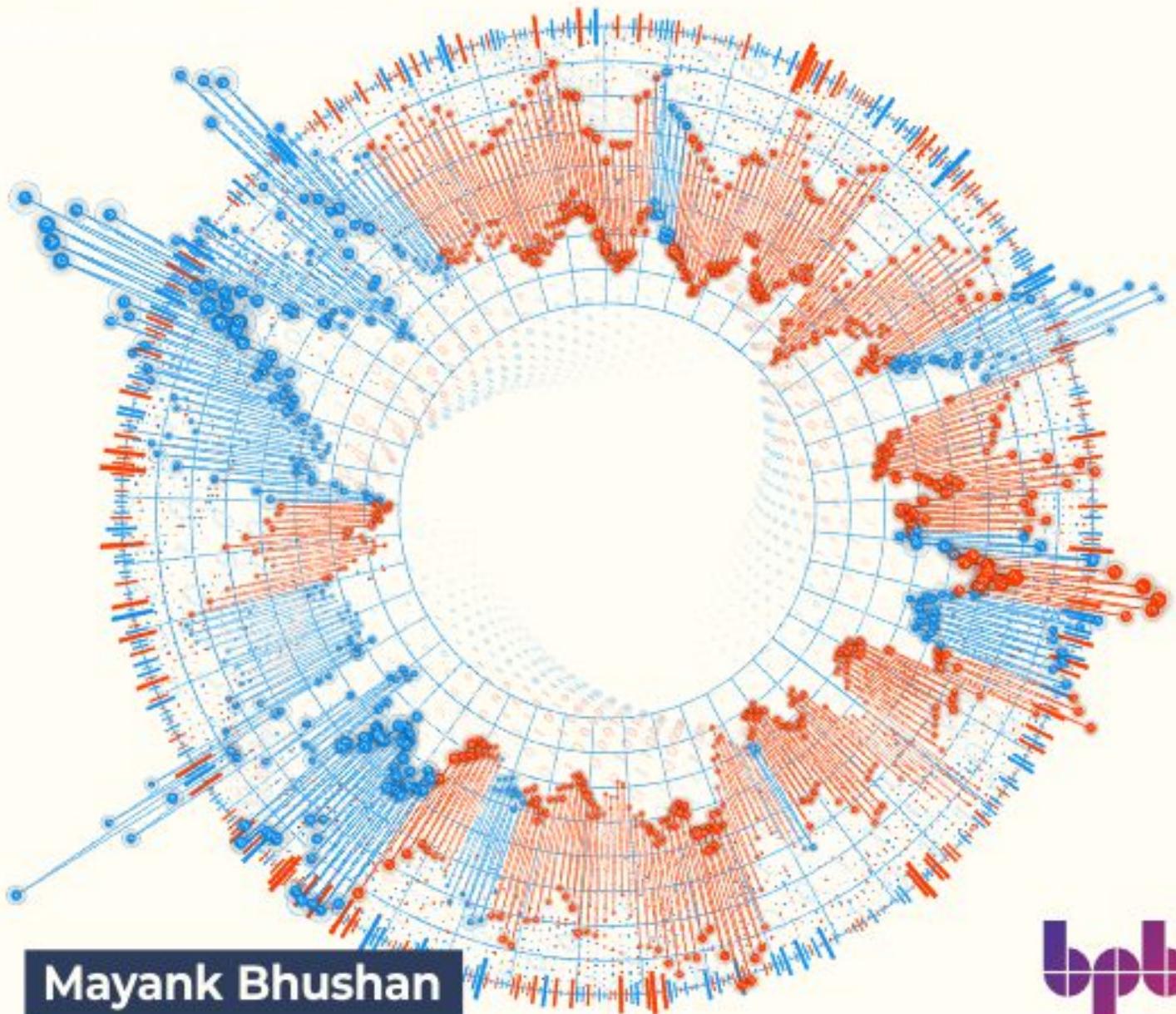


2<sup>nd</sup>  
Edition

# Big Data and Hadoop

Fundamentals, tools, and techniques for data-driven success



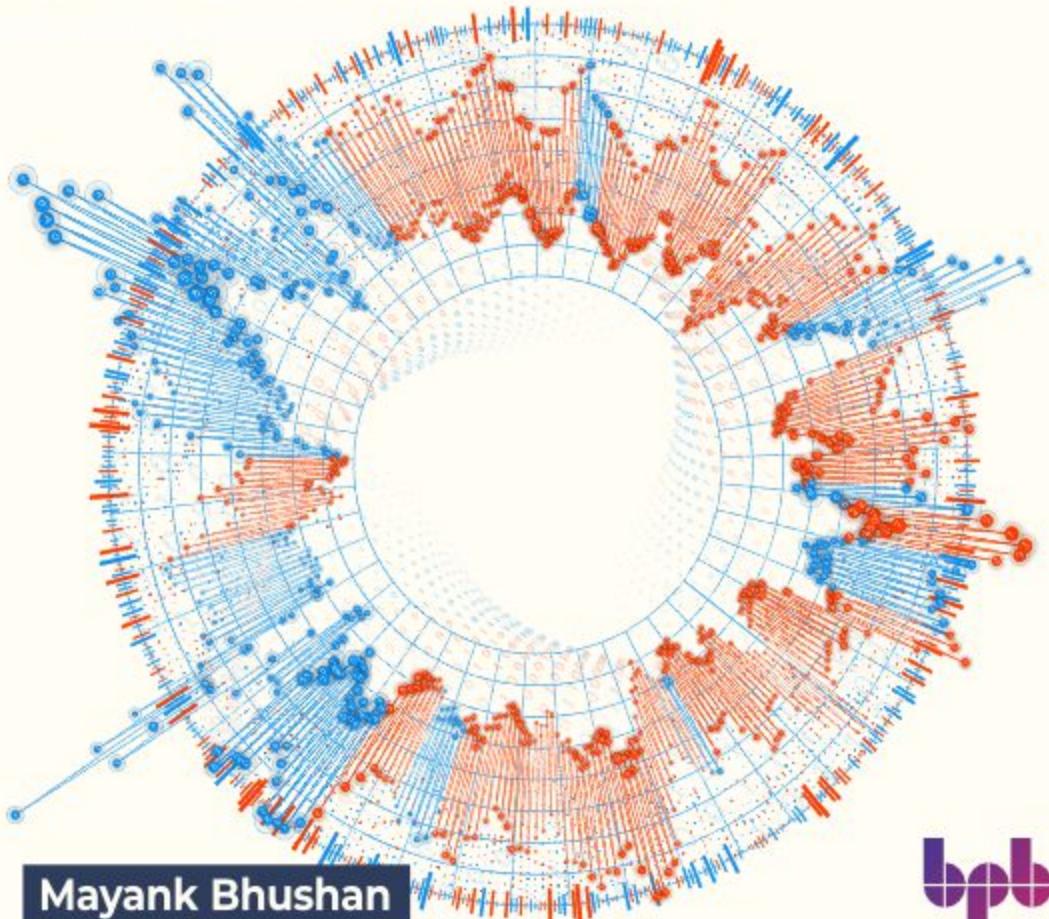
Mayank Bhushan



2<sup>nd</sup>  
Edition

# Big Data and Hadoop

Fundamentals, tools, and techniques for data-driven success



Mayank Bhushan



# **Big Data and Hadoop**

## **2<sup>nd</sup> Edition**

---

*Fundamentals, tools, and  
techniques for data-driven success*

---

**Mayank Bhushan**



[www.bpbonline.com](http://www.bpbonline.com)

Copyright © 2024 BPB Online

*All rights reserved.* No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor BPB Online or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

BPB Online has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, BPB Online cannot guarantee the accuracy of this information.

First published: 2018

Second published: 2024

Published by BPB Online  
WeWork  
119 Marylebone Road  
London NW1 5PU

**UK | UAE | INDIA | SINGAPORE**

ISBN 978-93-55516-664

[www.bpbonline.com](http://www.bpbonline.com)

**Dedicated to**

*My beloved Parents*  
**Smt. Neelam Sharma**  
**Sh. Gopal Krishna Sharma**  
**&**  
*My wife Apoorva*

## About the Author

**Mayank Bhushan** has a teaching experience of more than 15 years. He holds a B.Tech. degree in Computer Science and Engineering and an M.Tech. degree in the same field from Motilal Nehru National Institute of Technology Allahabad, Prayagraj. In addition to having good grades, he is certified to have global experience in Big Data Analytics and Salesforce-Cloud computing. Besides that, he has a certificate in Computer Networking from IIT Kharagpur, especially in the Linux platform. Along with this book, he has written various books tailored for vocational studies.

Throughout his career, the privilege of sharing knowledge through lectures at both private and government engineering colleges has been experienced. The focus during these lectures has been on the subject of Big Data and Hadoop. Commitment to education is deeply held by him and a self-designed course on Big Data and Cloud Computing has been developed. In this course, not only knowledge is imparted by him, but also valuable project ideas and real-time solutions to address any doubts are provided.

He has written many books in this area and is known for making important contributions to international study. With a lot of experience, he has written a number of important research papers that have been read around the world. Aside from his study, Mayank Bhushan has been an inspiration to many scholars, helping them with their theses and being a valuable mentor.

His knowledge and devotion have not only made academic literature better, but they have also had a huge impact on the academic careers of people who want to become researchers. His commitment to advancing knowledge and nurturing the next generation of scholars is evident in his prolific research output and mentorship roles.

## About the Reviewer

**Jai** is an experienced Data Engineer well-versed in large-scale data processing and the design of critical data models for cybersecurity and data privacy products. His expertise in machine learning and generative AI has played a significant role in advancing data governance in large corporations.

With an impressive 18-year career across diverse sectors such as FinTech, HealTech, AdTech, and Media, Jai's current research focuses on Children's privacy-enhancing technologies about social media and gaming platforms. His work is aimed at helping both small and large enterprises that gather personal data from minors to enhance their privacy practices and secure the personal information of millions of children and teenagers who frequently use these platforms globally. Currently, Jai is working with four startups to implement best practices in data cloud, data engineering, and data privacy.

In addition to his role as a data engineer, Jai is an author and speaker, primarily on topics related to data security, privacy, and children's rights and regulations. He has recently contributed as an editor to O'Reilly and as an author and peer reviewer for journals by IEEE and ACM.

Jai is a recognized figure in the Data Privacy field, having authored multiple articles and presented his work at various conferences. His ultimate vision is to nurture a community of privacy experts to solve real-world problems with data-driven solutions.

## Acknowledgement

I would like to express my heartfelt gratitude to the numerous individuals who supported me throughout the creation of this book. To all those who provided guidance, engaged in discussions, read and reviewed the content, shared their insightful comments, allowed me to quote their valuable remarks, and contributed to the editing, proofreading, and design process, I extend my sincere thanks.

Writing this book was a collaborative effort, and I am deeply thankful to the Hadoop community, whose continuous efforts have been a significant source of learning and inspiration for me.

Gratitude is a sentiment that we all share when others extend their helpful hands during challenging phases of life, assisting us in achieving our set goals. While it is impossible to individually thank everyone who played a role, I humbly make an effort to express my appreciation to some of them.

First and foremost, I offer my thanks to the Almighty, who invisibly guides us all and has kept us on the right path throughout this journey.

I owe a profound debt of gratitude to the following individuals:

1. Prof. (Dr.) Munesh Chandra Trivedi, National Institute of Technology, Agartala
2. Prof. (Dr.) Shailesh Tiwari, Director, KIET Ghaziabad (UP)
3. Prof. (Dr.) Mayank Pandey, MNNIT Prayagraj
4. Dr. Nitin Shukla, Asst. Prof. Thapar University
5. Dr. Shaswati Banerjea, Asst. Prof. MNNIT Prayagraj
6. Mr. Suraj Deb Barma, Principal, Govt. Polytechnic College, Agartala

7. Dr. Sumit Yadav, Director, Income Tax Dept.
8. Dr. Shailendra Kumar, Asst. Prof. IIT Bhilai
9. Dr. Saurabh Kumar Rajput, Asst. Prof. Madhav Institute Gwalior (MP)

Their unwavering support has been invaluable, and without their assistance, this book would not have been possible.

I extend my heartfelt thanks to all my Gurus, friends, and colleagues who provided unwavering support and kept me motivated throughout the writing process.

I am also grateful to the Publisher and the entire team at BPB Publications, technical reviewer of this book Mr. Jai and especially Mr. Manish Jain, for their efforts in presenting this text in a polished and presentable form.

In closing, I extend my appreciation to everyone who directly or indirectly contributed to the completion of this authentic work.

# Preface

Welcome to the world of Big Data! In today's data-driven landscape, the ability to harness and process vast amounts of information has become not just an asset but a necessity for businesses, researchers, and individuals alike. This book, titled *Big Data and Hadoop: Fundamentals, tools, and techniques for data-driven success* is your gateway to understanding and mastering the fascinating realm of Big Data.

**Chapter 1: Big Data Introduction and Demand** – In this opening chapter, we embark on a journey to explore the foundations of Big Data. We will delve into the very concept of Big Data, its significance in today's world, and the growing demand for solutions that can handle its challenges. We will also examine industry examples of how Big Data is being utilized and the myriad of possibilities it presents.

**Chapter 2: NoSQL Data Management** – This chapter takes us into the realm of NoSQL databases, offering an introduction to these non-relational data stores. We will compare SQL and NoSQL databases, explore the nuances of data consistency in NoSQL, and take a deep dive into the HBase database. Additionally, we will discuss the MapReduce paradigm and key concepts like partitioning and combining.

**Chapter 3: MapReduce Technique** – This chapter discusses a paradigm widely employed in the realm of distributed computing, that revolutionizes the processing of vast datasets with efficiency and scalability. Developed by Google, this technique serves as a cornerstone in the field of big data analytics. By harnessing the power of parallel processing and fault tolerance, MapReduce enables the seamless analysis of massive datasets across distributed clusters, making it a pivotal tool in addressing the challenges posed by the ever-expanding volume of data in diverse domains.

**Chapter 4: Basics of Hadoop** – To lay a solid foundation for your journey into Big Data technologies, this chapter introduces you to the basics of Hadoop. We will cover essential topics like data formats, analyzing data with Hadoop, scaling strategies, and the design of the Hadoop Distributed File System (HDFS). Concepts such as data flow, Hadoop I/O, compression, serialization, and Avro file-based data structures will be explored in detail.

**Chapter 5: Hadoop Installation** – Getting hands-on with Hadoop is crucial, this chapter guides you through the step-by-step process of installing Hadoop on various platforms. Whether you’re using Ubuntu or setting up a fully distributed Hadoop system, this chapter provides detailed instructions to help you get started.

**Chapter 6: MapReduce Applications** – This chapter is all about MapReduce, a fundamental programming model for processing Big Data. We will help you understand the principles behind MapReduce, walk you through the traditional way of using it, and explain the MapReduce workflow.

**Chapter 7: Hadoop Related Tools-I: HBase and Cassandra** – This chapter introduces you to two important tools in the Big Data ecosystem: HBase and Cassandra. You will discover how to install HBase, explore its conceptual architecture, and gain practical insights into its implementation. We will also delve into HBase’s key differences from traditional relational databases. The chapter then shifts focus to Cassandra, explaining its data model, providing examples, and discussing its integration with Hadoop.

**Chapter 8: Hadoop Related Tool-II: PigLatin and HiveQL** – It introduces two more essential tools: PigLatin and HiveQL. You will learn how to install PigLatin, understand its execution types, and explore the Pig data model. We will also guide you through the development and testing of PigLatin scripts. Next, we delve into Hive, exploring its data types, file formats, and comparing HiveQL with traditional database querying languages.

**Chapter 9: Practical and Research-based Topics** – This chapter is dedicated to practical and research-based topics in the world of Big Data.

You will explore real-world applications like data analysis with X, the use of Bloom Filters in MapReduce, leveraging Amazon Web Services, analyzing documents archived from The New York Times, mobile data mining, and Hadoop diagnostics.

**Chapter 10: Spark** – As we conclude our journey through Big Data and related technologies, this chapter introduces Apache Spark, a powerful framework for distributed data processing. We will explore its capabilities and understand how it fits into the Big Data landscape, setting the stage for your next adventure in data processing.

This book is designed to provide you with a comprehensive understanding of Big Data technologies, enabling you to tackle real-world challenges and leverage the opportunities presented by the ever-expanding world of data. Whether you are a student, a professional, or a curious explorer, we hope this book equips you with the knowledge and skills to thrive in the era of Big Data.

It is said “To err is human, to forgive divine”. In this light I wish that the shortcomings of the book will be forgiven. At the same I am open to any kind of constructive criticisms and suggestions for further improvement.

Happy reading and happy data processing!

## **Code Bundle and Coloured Images**

Please follow the link to download the ***Code Bundle*** and the ***Coloured Images*** of the book:

**<https://rebrand.ly/jb064ll>**

The code bundle for the book is also hosted on GitHub at **<https://github.com/bpbpublications/Big-Data-and-Hadoop-2nd-Edition>**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **<https://github.com/bpbpublications>**. Check them out!

## **Errata**

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

**[errata@bpbonline.com](mailto:errata@bpbonline.com)**

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePUB files available? You can upgrade to the eBook version at

[www.bpbonline.com](http://www.bpbonline.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

[business@bpbonline.com](mailto:business@bpbonline.com) for more details.

At [www.bpbonline.com](http://www.bpbonline.com), you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

### Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at [business@bpbonline.com](mailto:business@bpbonline.com) with a link to the material.

### If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit [www.bpbonline.com](http://www.bpbonline.com). We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

### Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit [www.bpbonline.com](http://www.bpbonline.com).

## Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



# Table of Contents

## 1. Big Data Introduction and Demand

Introduction

Structure

Objectives

Big data

Characteristics of big data

Why big data is required

Hadoop

*History of Hadoop*

*Name of Hadoop*

*Hadoop ecosystem*

Convergence of key trends

*Convergence of big data into business*

*Big data versus other techniques*

Unstructured data

*Mining unstructured data*

*Unstructured data and large data*

*Implementing unstructured data management*

Industry examples of big data

*Use of big data: Hadoop at Yahoo*

*RackSpace for log processing*

*Hadoop at Facebook*

Usages of big data

*Machine learning tools*

*Entertainment fields*  
*Web analytics*  
*Big data and marketing*  
*Big data and fraud*  
*Risk management in big data with credit card*  
*Big data and algorithm trading*  
*Big data in health care*

Conclusion

## 2. NoSQL Data Management

Introduction  
Structure  
Objectives  
Terminology used in NoSQL and RDBMS  
Database used in NoSQL  
    *Key-value database*  
    *Document database*  
        *Apache CouchDB*  
        *MongoDB*  
    *Column family database*  
        *Components*  
            *Table representation of Google Bigtable*  
            *BigTable derivatives*  
    *Graph database*  
        *Neo4j*  
        *GraphQL*  
SQL versus NoSQL  
    *Denormalization*  
    *Data distribution*  
    *Data durability*

Consistency issues in NoSQL

*ACID versus BASE*

*Relaxing consistency*

Hbase

*Installation of Hbase*

*History of Hbase*

*Hbase data structure*

*Physical storage*

*Components*

*Hbase shell commands*

*Different usages of the scan command*

*Terminologies*

*Version stamp*

*Region*

*Locking*

Conclusion

### 3. MapReduce Technique

Introduction

Structure

Objectives

MapReduce architecture

MapReduce datatype

*File input format*

*Java MapReduce*

Partitioner and combiner

*Example of MapReduce*

*Situation for partitioner and combiner*

Use of combiner

Composing MapReduce calculations

## Conclusion

### 4. Basics of Hadoop

Introduction

Structure

Objectives

Data distribution

Data format

Analyzing data with Hadoop

Scale-in versus scale-out

*Number of reducers used*

*Driver class with no reducer*

Hadoop streaming

*Streaming in Ruby*

*Streaming in Python*

*Streaming in Java*

Hadoop pipes

Design of HDFS

*Very large*

*Streaming data access*

*Commodity hardware*

*Low-latency data access*

*Lots of small files*

*Arbitrary file modifications*

*HDFS concept*

*Blocks*

*Namenodes and DataNodes*

*HDFS group*

*All-time availability*

*Hadoop files system*

Java interface

*HTTP*

*APIs in C language*

*Filesystem in Userspace*

*Reading data using the Java interface*

*Reading data using Java interface (FileSystem API)*

Data flow

*File read*

*File write*

*Coherency model*

*Cluster balance*

*Hadoop archive*

Hadoop I/O

*Data integrity*

*Local file system*

Compression

*Codecs*

*Compression and input splits*

*Map output*

Serialization

Avro file-based data structure

*Data type and schemas*

*Serialization and deserialization*

*Avro MapReduce*

Conclusion

## 5. Hadoop Installation

Introduction

Structure

Objectives

Using standalone (local) mode

*VmWare*

*On Ubuntu 16.04*

Fully distributed mode

*Installation and configuration of multi-node cluster*

Conclusion

## 6. MapReduce Applications

Introduction

Structure

Objectives

Understanding MapReduce

*Traditional way*

*MapReduce workflow*

*Map side*

*Reduce side*

*Sample program using MapReduce*

*Introduction of Web UI*

*Debugging MapReduce job*

*Job chaining and job control*

*Anatomy of MapReduce job*

*Anatomy of file write*

*Anatomy of file read*

MapReduce job run

*Classic MapReduce: MapReduce 1*

*Failure in MapReduce1*

*MapReduce2 YARN*

*Failure in MapReduce 2*

Conclusion

## 7. Hadoop Related Tools-I: HBase and Cassandra

Introduction

Structure

Objectives

Installation of Hbase

Conceptual architecture

*Regions and region server*

*Master Server*

*Locking*

Implementation

HBase versus RDBMS

HBase client

*Class HTable*

*Class Put*

*Class Get*

*Class Delete*

*Class Result*

HBase examples and commands

HBase using Java APIs

*Creating a table*

*List of the tables in HBase*

*Disable a table*

*Add column family*

*Deleting column family*

*Verifying the existence of the table*

*Deleting table*

*Disabling table*

*Stopping HBase*

Challenges

## Cassandra

*CAP theorem*

*Explanation in terms of intersection points*

*Characteristics of Cassandra*

*Installing Cassandra*

*Basic CLI commands*

## Cassandra data model

*Super column family*

*Clusters*

*Keyspaces*

*Column families*

*Super columns*

## Cassandra examples

*Creating a keyspace*

*Alter keyspace*

*Dropping a keyspace*

*Create table*

*Primary key*

*Alter table*

*Truncate table*

*Executing batch*

*Delete entire row*

*Describe*

## Cassandra client

*Thrift*

*Avro*

*Hector*

## Hadoop integration

## Use cases

*eBay*

*Hulu*  
Conclusion

## 8. Hadoop Related Tools-II: PigLatin and HiveQL

Introduction  
Structure  
Objectives  
Apache PigLatin  
Installation  
Execution type  
*Local mode*  
*MapReduce mode*  
The platform for running Pig programs  
*Script*  
*Grunt*  
*Embedded*  
*Grunt Shell*  
*Example*  
*Commands in grunt*  
Pig data model  
*Scalar*  
*Complex*  
PigLatin  
*Input and output*  
*Store*  
*Relational operations*  
*Examples*  
*User-defined functions*  
Developing and testing the PigLatin script

*Dump operator*  
*Describe operator*  
*Explanation operator*  
*Illustration operator*

## Hive

*Installing Hive*  
*Hive architecture*  
*Hive services*

## Data type and file format

## Comparison of HiveQL with traditional database

*Schema on read versus write*  
*Update, transactions and indexes*

## HiveQL

*Data definition language*  
*Data manipulation language*

## Conclusion

## 9. Practical and Research-based Topics

### Introduction

### Structure

### Objectives

### Data analysis with X

*Using flume*  
*Using MapReduce*

### Use of Bloom filter in MapReduce

*The function of the bloom filter*  
*Working of Bloom filter*  
*Application of Bloom filter*  
*Implementation of bloom filter in MapReduce*

Amazon Web Service

*Setting up AWS*

*Setting up Hadoop on EC2*

Examples of data analysis

*Document archived from NY Times*

Data mining in mobiles

Hadoop diagnosis

*System's health*

*Setting permission*

*Managing quotas*

*Enabling trash*

*Removing DataNode*

Conclusion

## 10. Spark

Introduction

Structure

Objectives

Spark programming model

Record linkage

Spark shell

*SCALA programming model*

*Features of Scala*

*Work on Scala*

Resilient Distributed Dataset

Spark methods for data processing

*Aggregate*

*Cartesian*

*Checkpoint*

*Repartition*  
*Cogroup*  
*Collect*  
*CollectAsMap*  
*CombineByKey*  
*Compute*  
*Count*  
*CountByKey*  
*CountByValue*  
*countApproxDistinct*  
*Dependencies*  
*Distinct*  
*first*  
*filter and filterWith*  
*filter Transformation*  
*fold*  
*foreach*  
*getStorageLevel*  
*groupBy*  
*Histogram*  
*id*  
*join*  
*leftOuterJoin*

Example of programs using Scala

*Shuffling*  
*Common Spark memory issues*

Conclusion

## Index

# CHAPTER 1

## Big Data Introduction and Demand

*...data is useless without the skill to analyze it*

– Jeanne Harris

*Taking a hunch you have about the world and pursuing it in a structural, mathematical way to understand something new about the world*

– Hilary Mason

### Introduction

In today's scenario, we all are surrounded by a bulk of data. We, as humans, are also an example of big data as we are surrounded by devices that generate data every minute.

*I spend most of my time assuming the world is not ready for the technology revolution that will be happening to them soon*

– Eric Schmidt  
Executive Chairman Google

As a matter of fact, if we compare the present situation to the past scenario, we can find that we are creating as much information in just two days as we did up to 2003, which means we are creating five exabytes of data every two days.

The real problem is user-generated data that they are producing continuously. At the time of data analysis, we have challenges in storing and analyzing those data.

## **Structure**

The following are the topics to be discussed in this chapter:

- Big data
- Characteristics of big data
- Why big data is required?
- Introduction of Hadoop
- Convergence of key trends
- Unstructured data
- Industry examples of big data
- Usages of big data

## **Objectives**

This chapter provides an introduction to big data and its need in the current scenario. Every device can generate data in each second, creating problems with its storage and, after that, its analysis for further usage. These kinds of issues will be addressed in this chapter. Here, we also collect introductory knowledge of Hadoop, which is used for processing and storing a large amount of data. It also helps in ending the confusion of different types of data generated by devices. Data has become a critical resource for businesses and organizations of all sizes in recent years. However, with the increasing volume, velocity, and variety of data, traditional data processing techniques and technologies have become insufficient to handle the data deluge. This is where the concept of big data comes into play. Big data refers to large and complex data sets that traditional data processing applications cannot handle effectively. Big data technologies enable the storage, processing, and analysis of these massive data sets to extract valuable insights and create new opportunities. Hadoop, one of the most popular big data technologies, is an open-source framework that facilitates the distributed processing of large data sets across clusters of commodity hardware. With Hadoop, organizations can analyze vast amounts of data in a cost-effective, scalable, and flexible manner, unlocking new insights and business opportunities.

*The real issue is user-generated content*

— Schmidt

## Big data

Big data refers to the vast and complex volumes of structured and unstructured data that exceed the processing capabilities of traditional data management systems. This data is characterized by its sheer size, velocity, variety, and, more recently, its value. Big data typically encompasses information from diverse sources, including social media, sensors, machine-generated data, and traditional databases. It presents unique challenges and opportunities as organizations seek to capture, store, analyze, and extract meaningful insights from this data to make data-driven decisions and gain a competitive edge.

Mostly, it helps Google to analyze the data and sell data analytics to companies who require it. We are producing data only through mobile as we already logged in when we bought the system:

- **Map:** It collects data on our traveling.
- **App:** It gathers information about our daily life activities and records activities in which we are most involved.
- **E-commerce sites:** It collect information about our requirement and shows whatever we are supposed to buy.
- **E-mails:** It is important to note that while Google scans e-mail content, it claims not to read e-mails for personal information or sensitive data. However, this practice has raised privacy concerns, and Google has faced legal challenges related to e-mail scanning in the past.

Indeed, over the past few decades, advances in technology, such as remote sensing, **Geographic Information Systems (GIS)**, and **Global Positioning Systems (GPS)**, have revolutionized the way we understand and analyze the distribution of human populations across the world. For that scenario, we need to map those population data to a meaningful survey that is performed

by big companies. As a result, spatially careful changes across scales of days, weeks, or months, or maybe year to year, area units are tough to assess and limit the applying of human population maps in things within which timely data is needed, such as disasters, conflicts, or epidemics. Information being collected daily by mobile network suppliers across the planet, the prospect of having the ability to map up-to-date and ever-changing human population distributions over comparatively short intervals exists, paving the approach for brand new applications and a close to period of time understanding of patterns and processes in human science.

Some of the facts related to exponential data production are as follows:

- Currently, over 2 billion people worldwide are connected to the internet, and over 5 billion individuals own mobile phones. By 2030, 150 billion devices are expected to be connected to the internet. At this point, predicted data production will be 44 times greater than that in 2009.
- To provide a rough idea, by 2020, global internet traffic was estimated to be measured in exabytes per month (1 exabyte = 1 billion gigabytes). Cisco's **Visual Networking Index (VNI)** forecasted that monthly global IP traffic would reach 260 exabytes per month by 2020. Facebook alone stores, accesses, and analyzes 30 + PB of user-generated data.
- In 2018, Google was processing 20,000 TB+ of data daily.
- Walmart processes over 1 million customer transactions, thus generating data in excess of 2.5 PB as an estimate.
- More than 5 billion people worldwide call, text, tweet, and browse on mobile devices.
- The number of e-mail accounts created worldwide is expected to increase from 3.3 billion in 2012 to over 4.3 billion by late 2016 at an average annual rate of 6% over the four years. In 2012, a total of 89 billion e-mails were sent and received daily, and this value is expected to increase at an average annual rate of 13% over the next four years to exceed 143 billion by the end of 2016; by this rate, it is expected to reach 500 billion + accounts by 2030.

- [Boston.com](#) reported that in 2021, approximately 1507 billion e-mails were sent daily. Currently, an e-mail is sent every  $3.5 \times 10^{-12}$  seconds. Thus, the volume of data increases per second as a result of rapid data generation. At this rate, an imaginary figure can be taken for us, which would be beyond our thinking.
- By 2030, enterprise data is expected to total 400 ZB, as per International Data Corporation.
- The New York Stock Exchange generates about one terabyte of data for new trade.

Based on this estimation, **business-to-consumer (B2C)** and internet-**business-to-business (B2B)** transactions will amount to 450 billion per day.

All are the facts that are sufficient to prove that the world is generating large amounts of data that are not only structured. That case leads to the innovation or thinking that can provide solutions for solving those issues.

Big data is the one which is used to deal with the current scenario. Big data is the concept for handling unstructured and structured data other than the traditional way. [Table 1.1](#) shows the flow of data from bottom to top. In today's scenario, any type of data is possible to store and process:

Number	Symbol	In Binary
Bit	b	1 bit
Nibble/Nibble	nibble	4 bits
Byte	B	8 bits
KiloByte	KB	1024 B
MegaByte	MB	1024 KB
GigaByte	GB	1024 MB
TeraByte	TB	1024 GB
PetaByte	PB	1024 TB

Number	Symbol	In Binary
HexaByte	HB	1024 PB
ZettaByte	ZB	1024 HB
YottaByte	YB	1024 ZB

**Table 1.1:** Introduction of data

## Characteristics of big data

Big data is data that gives the capacity to think beyond the traditional database system. As data that can be used in big data may be structured or unstructured data with a huge amount of capacity, it requires fast movement, fast storage, and fast processing other than conventional database techniques. These requirements of processing data demand tools that can perform functions fast and meaningful that are difficult by any traditional database tools. Properties of big data provide a next-generation way to handle situations and provide an easy and efficient way to handle data for an organization. As we all see around, there are a lot of devices that are continuously generating data with exponential increments, and all human beings are digging themselves into social networking. These types of unstructured and structured data create challenges in storing and processing data.

Every day, the world creates 2.5 quintillion bytes of data that are 90% of the data in the world today that was created in the last two years alone, and sources of those data from sensors, videos, post, Twitter, WhatsApp, Facebook, and many more digital sites of many users.

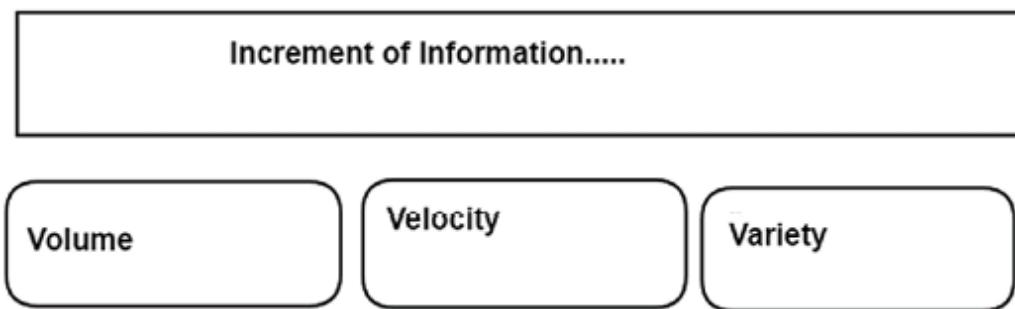
The following is the comparison between big data and traditional techniques of databases:

Traditional Database “Schema on Write”	Big data “Schema on Read”
There is a need to create a schema before	Data is firstly copied to HDFS; after

data is loaded into the database.	that, transformation is needed.
The load operator performs explicitly to transform the database.	Only required columns are extracted to perform operations.
It uses the scale-in property to the enhancement of data on the server side.	There is the use of scale-out property to enrich data at any time.

**Table 1.2:** Traditional database and big data schema

There are 3 V's that define its characteristics in a very clear manner:



**Figure 1.1:** 3 V's of big data

*Figure 1.1* shows three initial V's on which big data is dependent. Volume refers to any large amount of data that needs storage for analytics of data. As data is increasing exponentially, up to YB of data processing can be possible. Companies can think of it now with solutions. The volume of data is growing. Consultants predict that the amount of information within the world can grow to 25 YB in 2030, that is, with an exponential rate of increment.

An article could be a few weight unit bytes, a sound file could be a few megabytes, whereas a full-length pic could be a few gigabytes. Additional sources of information area unit adding on a continuous basis. For any company, at this time, all information is generated not only by the company's employees but also by its machines, such as CCTV cameras, punching machines, or sensible sensors. More sources of information with a bigger size data mix to extend the amount of information that needs to be analyzed. If we look around, there is no cost of GB of data in commodity systems. Soon, all will be replaced by TB's of data.

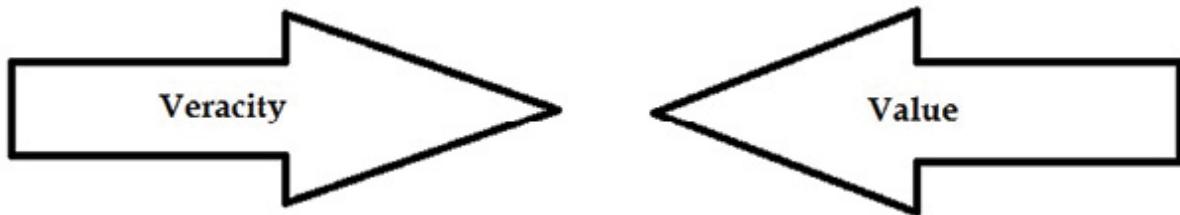
Velocity refers to the speed of data that exponentially increases. Data is increasingly accelerating the velocity at which it is created and at which it is integrated. We have moved from batch to a real-time business. At the start, there was a trend to analyze data in batch processing since the amount of data is large, which simply means that there is a need to submit data on the server and wait for its processing. It is obvious that the result will get delayed. With the latest source of data, there is a different type of data produced by machines that can be handled by big data easily. The data is now processed into the server in real time scenario, in a continuous fashion; delivery of data output also depends on the delay of sources omitting data.

It is not guaranteed that data comes to the machine in bulk. It might be slow sometimes. So, when there is a need to handle the pace variance of data flow techniques, there is an easy and accurate solution by big data.

Variety shows different types of input that are required for information extraction. Fact says that 80% of the world's data is unstructured, while we have options in traditional data handle techniques. Text (SMS), photo, audio, video, Web, GPS data, sensor data, relational data bases, documents, pdf, flash, and so on are the data that are flowing and require control to store and process it. Facebook, e-mails, and so on. Having no control over input that can be provided by any user. The variety of data sources continues to increase. It includes the following:

- Internet data (that is, click stream, social media, and social networking links).
- Primary research (that is, surveys, experiments, and observations).
- Secondary research (that is, competitive and marketplace data, industry reports, consumer data, and business data).
- Location data (that is, mobile device data, geospatial data, and GPS).
- Image data (that is, video, satellite image, and surveillance).
- Supply chain data (that is, EDI, vendor catalogs and pricing, and quality information).
- Device data (that is, sensors, PLCs, RF devices, LIMs, and telemetry).

Apart from the preceding data collection techniques, two additional characteristics have been introduced in big data, which are shown in [Figure 1.2](#):



**Figure 1.2: Additional V's**

There are two additional V's (as shown in [Figure 1.2](#)) that are useful to take the attention of the user in showing characteristics of big data. As all, we can find out the messiness of data around, such as Twitter hashtags and smiley with text. All these data are very typical to handle when there is a need for mining. Big data makes it easy to store and meaningful. Hash tag (#) in Twitter is used to categorize the topic so that at the time of extraction, meaningful or required data can be fetched out, and trustworthiness will remain with users. Nowadays, every company wants its survey and needs to do performance analysis; that is the reason hashtags are growing in popularity.

There is no need if there is no value of data. In that case, big data provides value-specific mining for enhancing the quality of data and time for its processing.

## Why big data is required

We are mostly dependent on devices nowadays, and these devices are creating data continuously, either structured or unstructured way. In that case, big data helps us lead in the market, research, and education. The following points can be considered which help us for a better life:

- **Considerate and target customers:** In today's scenario, big data is gaining popularity, and it relates itself to the latest technology with synchronization of the present one. This feature creates a better understanding of its knowledge among customers. Companies are

continuously storing a variety of data that are typical to handle with sensors, browser logs, social media, and so on, so it is preferable to store data first without much expectation of its format. It is easily used in the prediction of the behavior of machines, as well as humans:

- Using big data, Telecom companies now have better customer churn.
- Wal-Mart can predict what products will sell and where.
- Car insurance companies understand how well their customers actually drive and what offer can be provided to target next.
- Government election campaigns can be optimized using big data analytics, as we all are aware of central elections based on analytics.
- Large numbers of customers are targeted only by using online techniques.
- **Ease in business process:** As per earlier discussion, prediction on data can make business easy, moreover, to target customers. Big data is also increasingly used to optimize business processes. Any process of analytics in business needs historical data for an accurate model:
  - Retailers are now optimizing their stock based on predictions generated from social media, Web trends, and weather forecasts. They also predict about targeted area of companies for selling goods.
  - People can be easily tracked with their roaming behavior as all relate to GPS that are log-based. Many times, we can observe about route optimization with the help of analytics of data.
  - Not even the HR department is untouched by big data exponential growth. Moneyball is a style to optimize talent in any field.
- **Personal growth and optimization:** If all of us look around, we can find that we are the only ones who are targeted by companies to increase their sales. Nowadays, many gadgets are sold by companies that are tracking all habits of users, that useful for personal growth as well.:

- We can now take advantage of data-generated devices such as wearable devices, smart bracelets, and so on.
  - UP band from Jawbone is also an activity tracker to collect data and observe it for processing to consumption of calories and sleeping patterns. This company does not have sleep data of 60 years of individuals that can be taken for business purposes and personal growth also for individuals.
  - Processing a large amount of data bring analysis for individual user like love online sites, marriage sites, and recommendation engines; all these are based on analysis. More data that gives more accurate results.
- **Regarding health improvisation:** Big data allow prediction and analysis of the string of patterns that are useful in the cure of disease. DNA data analysis pattern is one of them. Companies with data on health that is flowing from wearable watches, band, and so on can be recognized by its pattern for solving the disease of many individuals. Many antibiotics follow the same pattern to diagnose and cure disease. DNA data analysis using big data techniques allows us to understand and better cure within min. Big data techniques are already being used to monitor premature babies with prescriptions and suggestions by recording and analyzing every heartbeat and breathing pattern of the baby; by analyzing patterns, there are also predictions can be made about infections. There is an algorithm developed that predicts a cure of infection based on a pattern in which the baby is there. Big data analytics allow for monitoring and predicting the development of epidemics and disease outbreaks.

Social media is also very useful for predicting upcoming diseases. All this can be done by comments that are posted on Twitter or Facebook. Sensitive viruses are also predicted before their entry into place. Zika virus is an example of predictive analysis in the medical field by social media.

- **Improving sports performance:** Many of sports are interestingly in the big data field for their accuracy prediction. Most selected sports have now embraced big data analytics:

- IBM SlamTracker tool use for tennis tournaments.
  - Video analytics track the performance of individual players in a football or baseball game.
  - Sensor technology in sports equipment such as basketball or golf clubs allows us to get feedback (via smartphones and cloud) on our game and how to improve it.
  - Many sports teams also track athletes outside of the sporting environment regarding their patterns and habits.
- **Improving science and research:** Science and research are also not untouched by big data analytics; these are also producing new opportunities and possibilities, for example, CERN, the Swiss nuclear physics lab with its Large Hadron Collider, the world's largest and most powerful particle accelerator. The CERN data center has 450,000 processors and 10,000 servers to analyze its petabytes of data. It uses the computing powers of thousands of computers distributed across 170 data centers worldwide to analyze the data. Such a powerful setup can fetch data process in the use of research and development.
  - **Enhancing and optimizing device performance:** Big data analytics help machines and devices become smarter and independent. For example, the self-operated Google's car. The Toyota Prius is fitted with cameras, GPS, and powerful computers and sensors to drive safely on the road without the intervention of humans. All these devices are well-trained with intelligence systems only when they have large amount of data. These are also capable of making real-time decisions for handling situations.
  - **Improvising security features:** Big data is applied seriously in improving security and enabling law enforcement. **National Security Agency (NSA)** use data to foil terrorist plot and spy on them. In cyber-attacks, there is the use of big data. With a large amount of data on behavior analysis, we can easily track security concerns. The police department can also use fraud detection to catch criminals, especially in the case of internet deals.

- **Improving and optimizing cities and countries:** Big data is used to improve many aspects of our cities and countries. Since the government is very serious about managing smart cities in the country and making any city smarter, there is a need to analyze bulk amount of data to take appropriate decisions such as traffic flow, weather data, and many sensor information. It will also be helpful in analyzing how to reduce manmade problems.
- **Financial trading:** There is the use of big data in trading purposes with a high frequency of trading. It needs to take wise decisions based on algorithms of intelligence. For real implementation of trade scenarios, there is raw data that will be used from social media mostly to help take decisions in buying, selling, or keeping things with us.

## Hadoop

Two problems before the world are as follows:

- Data storage
- Data analysis

It will be a waste if we are not able to collect an amount of data. So, there will be a need of storing data with the scale-out property. The traditional way used to collect data on the server side required special maintenance with its own limitations that said scale-in property while scale-out property deals with commodity hardware to store data.

Apache Hadoop is a framework that allows for the distributed processing of large data sets across clusters of commodity computers using a simple programming model, and it is open source.

## History of Hadoop

Following is the history of Hadoop depicted:

2002	Doug Cutting, a Graduate of Stanford		
------	--	--	--

	<p>University, and Mike Cafarella, an Associate Professor at the University of Michigan, started working on Apache Nutch.</p>	 <p><i>Doug Cutting</i></p>	
2003	<p>A successful 100-million-page demonstration system was developed. To meet the multi-machine processing needs of the crawl and index tasks, the Apache Nutch project has also implemented a MapReduce facility and a distributed file system. The two facilities have been spun out into their own sub-project, called Hadoop.</p>		
2004	<p>Doug Cutting adds DFS and to Nutch. Google also released a</p>		

	<p>paper on GFS by Sanjay Ghemawat.</p>	
2005	<p>New technology awaited under review</p>	
2006	<p>The initial code for Hadoop will be copied from Nutch. NDFS+ MapReduce moved out of Apache Nutch to create Hadoop. Hadoop 0.1.0. released. And importantly, Yahoo hires Cutting.</p>	 <p>Welcome Mr. Cutting &amp; Co.</p>
2007	<p>NY Times converts 4 TB of Image archives over 100 EC2s. The first release of Hadoop that includes HBase. Yahoo Labs creates</p>	 <p>Pig</p>

	Pig and donates it to the ASF.	
2008	<p>Twenty companies on the “Powered by Hadoop Page.” Yahoo moved its Web index onto Hadoop. First Hadoop Summit15. Hadoop World record is the fastest system to sort a terabyte of data. Running on a 910-node cluster, Hadoop sorted one terabyte in 209 seconds. Hadoop wins Terabyte Sort Benchmark. Cloudera, a Hadoop distributor, was founded. Google MapReduce implementation sorted one terabyte in 68 seconds. Facebook launched Hive.</p>	<p><a href="#">Site index</a> · <a href="#">List index</a></p> <p><b>Message view</b></p> <p><b>From</b> "Ajay Anand" &lt;aan...@yahoo-inc.com&gt;  <b>Subject</b> RE: Hadoop summit / workshop  <b>Date</b> Wed, 20 Feb 2008 20:10:09 GMT</p> <p>The registration page for the Hadoop summit is at <a href="http://developer.yahoo.com/hadoop/summit/">http://developer.yahoo.com/hadoop/summit/</a></p> <p>Space is limited, so please sign up early if you are attending.</p> <p>About the summit:      Yahoo! is hosting the first summit on Apache Hadoop in Sunnyvale. The summit is sponsored by the Computation Cluster Consortium (CCC) and brings together leaders from the Hadoop and MapReduce communities. The speakers will cover topics in the ecosystem, including new features being developed for Hadoop, case studies of applications deployed on Hadoop, and a discussion on future directions for the platform.</p> <p>Agenda:</p> <ul style="list-style-type: none"> <li>8:30-8:55 Breakfast</li> <li>8:55-9:00 Welcome to Yahoo! &amp; Logistics - Ajay Anand</li> <li>9:00-9:30 Hadoop Overview - Doug Cutting / Eric Phillips</li> <li>9:30-10:00 Pig - Chris Olston, Yahoo!</li> <li>10:00-10:30 JAQL - Kevin Beyer, IBM</li> <li>10:30-10:45 Break</li> <li>10:45-11:15 DryadLINQ - Michael Isard, Microsoft</li> <li>11:15-11:45 Monitoring Hadoop using X-Trace - Avinash Venkataraman, UC Berkeley</li> <li>11:45-12:15 Zookeeper - Ben Reed, Yahoo!</li> <li>12:15-1:15 Lunch</li> <li>1:15-1:45 Hbase - Michael Stack, Powerset</li> <li>1:45-2:15 Hbase App - Bryan Duxbury, Raptleaf</li> <li>2:15-2:45 Hive - Joydeep Sen Sarma, Facebook</li> <li>2:45-3:00 Break</li> <li>3:00-3:20 Building Ground Models of Southern California - Matt Harrah, University of Southern California</li> </ul> <p><i>Mail of First Summit</i></p>

2009	<p>Yahoo runs 17 clusters with 24,000 machines.</p> <p>Hadoop sorts petabytes.</p> <p>Yahoo! used Hadoop to sort one terabyte in 62 seconds.</p> <p>Second Hadoop Summit. HDFS is now a separate subproject.</p> <p>Doug Cutting joins Cloudera.</p>
------	--



*Cutting joins Cloudera*

**Table 1.3:** History of Hadoop

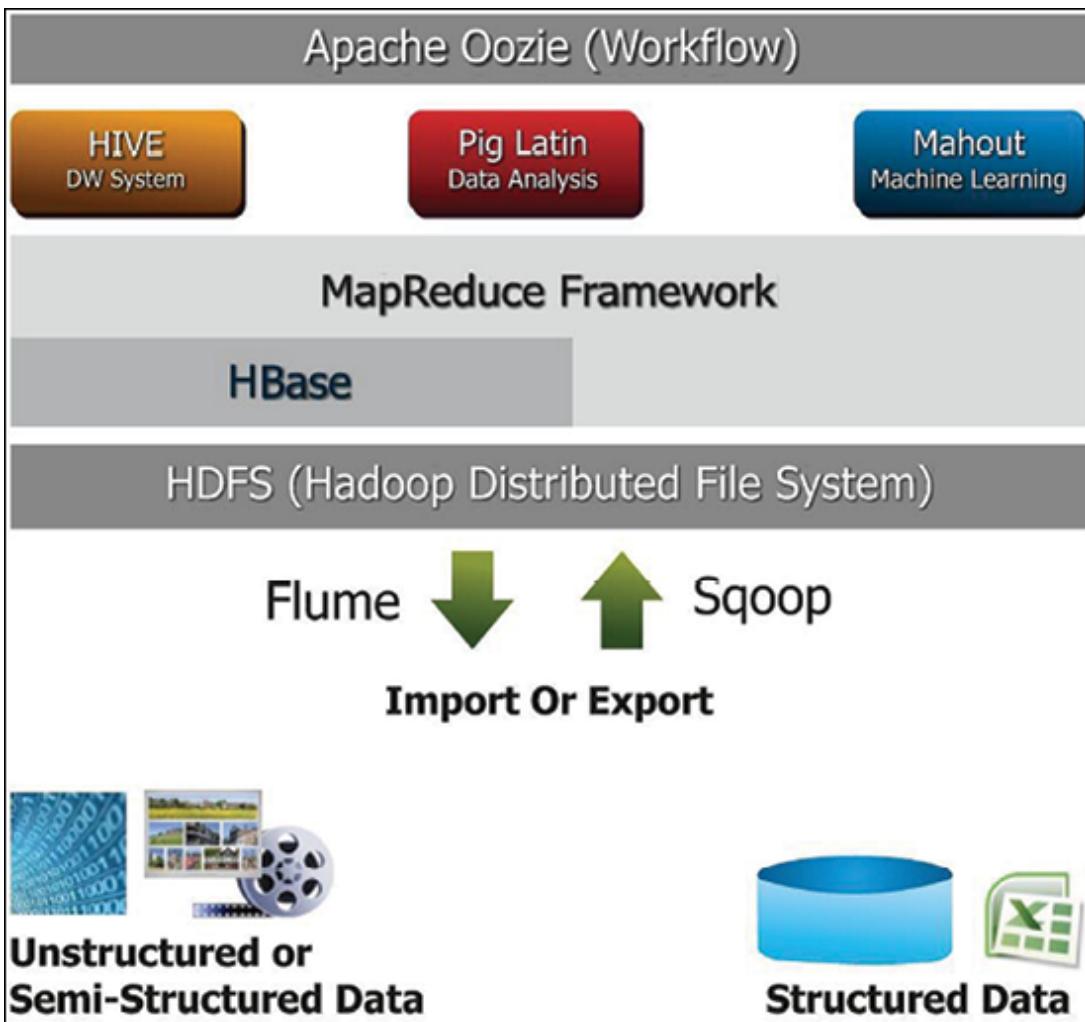
## Name of Hadoop

Hadoop was developed by *Doug Cutting* and his friend, and he is the one who suggested its name that was meaningless. According to him:

*The name my kid gave a stuffed yellow elephant. Short, relatively simple to spell and pronounce, meaningless, and not used elsewhere. Kids are good at generating such. Googol is a kid's term.*

## Hadoop ecosystem

Hadoop is the framework that allows structured and unstructured data for processing. Hadoop provides an ecosystem for processing data with MapReduce, PigLatin, Hive, HBase, Sqoop, Flume, and many more:



*Figure 1.3: Hadoop ecosystem*

*Figure 1.3* shows the Hadoop ecosystem in which Flume and Sqoop are the tools that are used for importing and exporting data from different resources. These resources are independent of the type of data. With streaming of data that flows to the distributed environment with a NoSQL database is processed by the MapReduce framework; furthermore, it can be processed by other tools such as Pig, Hive, and so on.

Following is a brief introduction to components used in the ecosystem:

- **HDFS:** HDFS is a complicated than different file systems, given the complexities and uncertainties of networks. This is required for storing data on the data node. It required a connection of the master node with the slave node. The second node sort could be a data-storing node that

acts as a slave node. This sort of node comes in multiples set up with Hadoop configuration. Other than these, there is the use of a secondary name node that reflects an image of the master node, which can be used to avoid a single point of failure. HDFS stores files in blocks; the default block size is 64 MB/128 MB in Hadoop 1.0 and Hadoop 2.0, respectively. All HDFS files are unit replicated in multiples to facilitate the multiprocessing of huge amounts of information.

- **HBase:** HBase could be a management system that is an ASCII text file, versioned and distributed, and supported by the BigTable of Google. This method is column- instead of row-based, other than traditional storage, which accelerates the performance of operations over similar values across giant knowledge sets. For instance, browse and write operations involve all rows, however, solely a little set of all columns. HBase is accessible through **Application Programming Interfaces (APIs)** such as Thrift, Java, and **representational state transfer (REST)**.
- **ZooKeeper:** ZooKeeper maintains the configuration of amounts of information. It conjointly provides distributed synchronization and cluster services. This instance allows distributed processes to manage and contribute to one another through a reputation area of information registers (z-nodes) that is shared by a classification system. Alone, ZooKeeper could be a distributed service that contains master and slave nodes and stores configuration info.
- **HCatalog:** HCatalog manages HDFS. It stores data and generates tables for large amounts of information. HCatalog depends on Hive metastore and integrates it with different services, as well as MapReduce and Pig, using a common knowledge model. With this knowledge model, HCatalog may also expand to Hbase. HCatalog simplifies user communication with HDFS and could be a supply of information sharing between tools and execution platforms.
- **Hive:** Hive structures warehouses in HDFS and different input sources, such as Amazon S3. Hive could be a subplatform within the Hadoop scheme and produce its own search language (HiveQL). This

language is compiled by MapReduce and allows **user-defined functions (UDFs)**. The Hive platform is based totally on three connected structures: tables, partitions, and buckets. Tables correspond to HDFS directories and may be distributed in numerous partitions.

- **Pig**: The Pig framework generates a high-level scripting language (Pig Latin) and operates a run-time platform that permits users to execute MapReduce on Hadoop. Pig is a lot more elastic than Hive with reference to potential data formatting, given its knowledge model. Pig has its own knowledge sort, map, that represents semi-structured knowledge, as well as JSON and XML.
- **Mahout**: This is the library for machine learning and data processing. It is divided into four main groups: collaborative filtering, classification, clustering, and mining of parallel frequent patterns.
- **Oozie**: Oozie coordinates, executes, and manages job flow. It is incorporated into different Apache Hadoop frameworks, such as Hive, Pig, Java MapReduce, Streaming MapReduce, and Sqoop. Oozie combines actions and arranges Hadoop tasks using a **directed acyclic graph (DAG)**. This model is usually used for numerous tasks.
- **Avro**: Avro serializes knowledge, conducts remote procedure calls, and passes knowledge from one program or language to a different one. During this framework, knowledge area unit self-describing and area unit perpetually keep supporting their own schema as a result of these qualities, area unit notably suited to scripting languages like Pig.
- **Flume**: Flume is especially accustomed to mixture and transfer giant amounts of information (that is, log data) in and out of Hadoop. It uses two channels, namely, sources and sinks. Sources embody Avro, files, and system logs, whereas sinks check with HDFS and HBase. Through its personal engine for question process, Flume transforms every new batch of massive knowledge before it is shuttled into the sink.

## Convergence of key trends

A study at *McKinsey Global Institute* predicted that by 2030 the annual GDP in manufacturing and retail industries would touch \$525+ billion with the use of big data analytics. This study also gained knowledge about the \$285 billion mark, totaling \$610 billion per annum in health care and government services.

More than 80% of all Fortune 500 companies will be shifted to Hadoop. Eight per cent by 2030 from all organizations, whereas the rest of the industries are in experimental stages and thinking of shifting to Hadoop or they are outsourcing their work.

In the present scenario, 90% of data is generated by social media, and trend analysis becomes part of that. “Big data” itself generates about 8 million results under the news section and approximately 54 million results on a simple Google search. Research says that about 8 Zettabytes of big data are created by e-mail and simple Google results. So, facts that there is a need and requirement to switch on big data before companies.

### **Convergence of big data into business**

In business, big data is used to customize the requirement of customer needs and its prediction analysis. Businesses demand the right customer at the right time to deal with. By this, there is a lot of time saved for fixing targets. It raises business competition among companies for accuracy and efficiency.

Following is the table that shows the technical specifications of companies in different areas for big data:

<b>Company</b>	<b>Business</b>	<b>Technical Specification</b>	<b>Usages</b>
Facebook	Social Site	Eight cores and 12 TB of storage	Used as a source for reporting and machine learning

<b>Company</b>	<b>Business</b>	<b>Technical Specification</b>	<b>Usages</b>
X	Social Site		Hadoop has been used since 2010 to store and process tweet log files using the LZO compression technique, as it is fast and also helps release CPU for other tasks.
LinkedIn	Professional Social Site	2X4 and 2X6 cores—6X2 TB SATA 4100 nodes	LinkedIn's data flows through Hadoop clusters. User activity, server metrics, images, and transaction logs stored in HDFS are used by data analysts for business analytics, like discovering people you may know.
Yahoo!	Online Portal	4500 nodes—1 TB storage, 16 GB RAM	Used for scaling tests.
AOL	Online Portal	ETL style processing and statistics generation	Target machines and dual processors
eBay	ECommerce	4K+ nodes cluster	With 300+ million users browsing more than 350 million products listed on their website, eBay has one of the largest Hadoop clusters in the industry that run prominently on MapReduce Jobs. Hadoop is used at eBay for Search Optimization and Research.
Alibaba	ECommerce	Processes 15-node cluster business data	Analyzes vertical search engine

Company	Business	Technical Specification	Usages
Adobe	Publishing and editing software	30 nodes running HDFS, 5 to 14 nodes HBase	Social services to structured data storage
Infosys	IT Consulting	Per client requirements	Client projects in finance, telecom, and retail.
A9.com	Product and Visual Search	1 to 100 nodes	Search indices

**Table 1.4:** Technical specification

**Table 1.4** shows the usage of big data in different companies with its requirements. Although Hadoop is open source, many companies are using it on their own by making it paid. For example, SAP uses a big data market with SAP HANA, an in-memory database platform for real-time analytics. SAP HANA shows the role of big data in current as well as traditional scenarios as well.

## Big data versus other techniques

Big data approach is not new, but it is a system that gives direction to think other than traditional conventional methods. It provides batch approaches to solve data analytical techniques. It is an innovative dimension for a person to process requests. Rackspace engineers bring this technique in the mining of e-mails that come into their servers. MapReduce facilitates a large amount of data with quick and accurate results, the most general question that we all can think over about its comparison with RDBMS. Following are the reasons why big data became so popular and in trend in the present scenario:

- **RDBMS versus big data:** First, there is a difference between the types of data both techniques can process. MapReduce can work easily on unstructured, semi-structured, or structured data, whereas RDBMS can work mainly on structured data. RDBMS mostly have

the structure of data that can easily be stored. It provides a specified format. On the Other hand, MapReduce uses data in a spreadsheet with a grid of cells that can hold any form of data to store. It uses a key-value pair, which is also not a basic property of data to analysis.

Relational data needs to follow the property of ACID, which keeps integrity and removes redundancy of data; moreover, it keeps normalization of data. For example, Web server logs for the record that is not normalized and can be analyzed only by MapReduce in the present scenario. It uses a key-value pair in its linear scalable programming model. The size of the cluster will impact on the speed of data processing, whereas this is not the case in SQL.

Some of the techniques can make MapReduce more reliable, such as Pig, Hive, and so on, whereas this is not done in SQL:

<b>Attribute</b>	<b>RDBMS</b>	<b>MapReduce</b>
Data Size	GigaBytes	PetaBytes
Access	Interactive	Batch
Updates	Read and write many times	Write once and read many times
Structure	Static	Dynamic
Integrity	High	Low
Scaling	Nonlinear	Linear

**Table 1.5: RDBMS versus MapReduce**

SQL database is most suitable for updating small and medium portions of data. The B-Tree pattern to handle data is less efficient than the MapReduce system.

- **Grid computing versus big data:** For many years, there has been the use of large data processing for APIs as a message-passing interface. This is used for the distribution of work across a cluster that accesses a

common distributed file system. This works fine with a limited amount of data and a network of machines. But when a large amount of data needed for processing MapReduce takes place, that is called *data locality*.

For large data centers, there is reliability on network connectivity that is easily provided by MapReduce, which uses namenode and datanode family structure. With network topology structure, all systems can give the property of data scalability. A pair of key-value pair will run on all nodes that store data. So, we can clearly observe the MapReduce technique in grid computing.

- **Cloud computing versus big data:** There is very obvious doubt about similarities between them. In cloud computing, there is no need to worry about setup and storage of data. Even there is no way to think about its materialization. It can be a source of big data, but it is different in computing terms.

Although big data demands computing with proper setup and processing daemons, it also takes care of the infrastructure that needs to be provided for the proper functionality of the system. Dropbox, Amazon Web Services, gdrive, and so on are the services that provide cloud-based systems without knowing infrastructure knowledge. It might be possible to set up cloud by keeping big data set up in mind.

Between big data and the cloud, there is a thin line of difference that shows mainly in its implementation, as shown in *Table 1.6*:

Big data	Cloud computing
Volume	SaaS
Velocity	PaaS
Variety	IaaS

	<b>Volume</b>	<b>Variety</b>	<b>Velocity</b>
--	---------------	----------------	-----------------

SaaS	Semantics	Visualization	Real Time
PaaS	Distributed Processing	Schema less	Integration
IaaS	Scalable Storage	Federated Storage	On demand Storage

**Table 1.6:** Difference between cloud and big data

As we have already discussed, big data is a requirement for large unstructured data and processing of it. For the extension of this task in February 2008, Yahoo used it with 10,000 core Hadoop cluster nodes. By this time, other companies such as last.fm, Facebook, and the New York Times also started working on it. In April 2008, it enhanced its usability to become the fastest system to sort terabytes of data running on a 910-node cluster by sorting it in 910 seconds, beating a record of 297 seconds. This also further decreased to 68 seconds, as Google reported.

## Unstructured data

All data need to be structured when we talk about the traditional database, but at the time when data is flooding from any sources and producing texts, videos, audios, images, geographical data, and so on, there is a requirement to think beyond traditional data that is unstructured data. Semi-structured data is the combination of data of structured and unstructured one. For example, in any support center, there are a minimum of three information needed that are name of customer, date of purchase, and complaint. In the complaint section, users can fill in any detail such as text, image, video, and so on.

The diversity of data sources continues to increase. Traditionally, key resources of data are from CRM and from ERPs for analytics purposes. Data sources are increasing rapidly with a wide variety of data such as:

- Internet data (for example, clicks, surveys, social media, and so on).
- Location data (for example, satellite data, mobile device data, and so on).
- Device data (for example, Sensors, RFIDs, and so on).

- Search data (for example, Google search data and so on).
- Image data (for example, Surveillance data, image, and so on).
- Supply chain data (for example, EDI, pricing data, quality information data)

So, unstructured data is information that is not actually pre-defined, cannot fit into a relational database, and does not require database schema. While semi-structured data does not contain pre-defined tags that separate it. Following are the reasons that we need to think over unstructured data in spite of so much take care of its variety:

- Data is increasing exponentially by double of its rate.
- Unstructured data is generated rapidly, that is, almost 95% of data with a variety of images, videos, or texts, unlike structured data.
- Most of the unstructured data is unutilized and is stored continuously without preparing its processing.

## **Mining unstructured data**

Many organizations believe that their unstructured data stores embrace info that would facilitate them to create higher business choices. Sadly, it is typically tough to investigate unstructured data. To assist with the matter, organizations have turned to a variety of software system solutions designed to look at unstructured knowledge and extract vital info. The first advantage of these tools is the ability to harvest unjust info, which will facilitate a business to achieve competitive surroundings.

Because the quantity of unstructured data is growing quickly, several enterprises conjointly address technological solutions to assist them higher managing and storing their unstructured knowledge. These will embrace hardware or software system solutions that alter them to create the foremost economical use of their out there space for storing.

## **Unstructured data and large data**

As mentioned earlier, unstructured data is the opposite of structured data. Structured data usually resides in a computer database, and as a result, it is generally known as **relational data**. This sort of information will be simply mapped into pre-designed fields. For instance, an information designer might create fields for phone numbers, nada codes, and MasterCard ranges that settle for a definite number of digits. Structured data has been or will be placed in fields like these. Against this, unstructured knowledge is not relative and does not work into these styles of pre-defined knowledge models.

In addition to structured and unstructured, there is conjointly a third category: semi-structured data. Semi-structured data is info residing in a computer database; however, that does have some structure properties that make it easier to investigate. Samples of semi-structured data would possibly embrace XML documents and NoSQL databases.

The term *big data* is closely related to unstructured data. Massive data refers to extraordinarily massive datasets that square measure tough to investigate with ancient tools. Massive knowledge will embrace each structured and unstructured data; however, IDC estimates that 90% of massive knowledge is unstructured knowledge. Several of the tools designed to investigate massive knowledge will handle unstructured knowledge.

## **Implementing unstructured data management**

Organizations use a type of totally different software system tools to assist them in organizing and managing unstructured data. These will embrace the following:

- **Big data tools:** Software systems like Hadoop and Iceberg will method stores of each unstructured and structured data that square measure extraordinarily massive, terribly complicated, and dynamically quickly.
- **Business intelligence software:** Conjointly called metallic element, this can be a broad class of analytics, data processing, dashboards, and reportage tools that facilitate corporations' be of their structured and unstructured data for the aim of creating higher business choices.

- **Data integration tools:** These tools mix data from disparate sources so that they will be viewed or analyzed from one application. They often embrace the aptitude to unify structured and unstructured data.
- Document management systems that are also known as enterprise content management systems, that used to track, store, and share unstructured data that is saved within document files, which can be used for processing purposes.
- Information management solutions, which sort of software system tracks structured and unstructured enterprise data throughout its lifecycle.
- Search and classification tools that tools used to retrieve information from unstructured data files like documents, websites, and photos.

## **Industry examples of big data**

This is best practice to know about the relation of big data in the current scenario. Evolving big data does not come by overnight effort, and many experts say that this is not a new concept, but this is a new way to think about this. But this gives so much more efficient approach to the analysis of data in a time-efficient manner. This generates the era of interrelated and interdependent links of data.

One related example of big data with data scientists such as *Drew Conway*, *Jared Lander*, and *Jake Porway* at the coffee shop. Three of them started to share their work over meetings that were based on big data analytics. They used their skills as a data scientist to solve problems for humanity.

*Jared Lander* helps a large healthcare organization solve big data problems related to patient data and helps them recover from disaster recovery. He analyzes the needs of villagers; they need bottled water or boats, rice or wheat, and shelter or toilets. After following their choices by collecting data by survey, follow-up operations were tracked and helped directly from disaster and gave them relief.

*Jake Porway* at *DataKind* with *Drew Conway* helps data-driven sectors by analyzing social sectors and educational resources through tools of big data.

## **Use of big data: Hadoop at Yahoo**

Building internet-scale search engines needs large amounts of knowledge and, thus, large numbers of machines to implement it. Yahoo! Search consists of four primary components:

- The Crawler downloads pages from internet servers.
- The WebMap, which builds a graph of the identifiable Web, builds a reverse index to the most effective pages that answer users' queries. The WebMap could be a graph that consists of roughly one trillion boundaries, every representing an online link and a hundred billion nodes, every representing distinct URLs. Analyzing such a graph requires capable computers running for several days. In early 2005, these computers, called dreadnoughts, scaled from 20 to 600 nodes.
- Dreadnought (a very old technology) is comparable to MapReduce in several ways; however, it provides additional flexibility and fewer structure.
- *Eric Baldeschwieler* leads a team that started planning and prototyping a replacement framework written in C++ once GFS and MapReduce to replace dreadnought. This progress was in observance of Hadoop, which was part of Nutch. In January 2006, Yahoo! employed Doug Cutting and a month later adopted Hadoop.

## **RackSpace for log processing**

Rackspace is currently hosting more than 1 million users and thousands of companies on several servers.

Being a large e-mail service provider, Rackspace needs to analyze more than 150 GB of data per day. It was so useful to analyze since it is required to know about customer feedback and performance. In that case, troubleshooting was difficult to conduct because of the traditional system. If an associate e-mail fails to be delivered or a client is unable to log in, it is very important that the customer support team is ready to search out enough data regarding the matter to start the debugging method to form it possible to search out that data quickly, it is not possible to leave the logs on the

machines that generated them or in their original format. Instead, they use Hadoop to try and do a substantial quantity of processes, with the result being indexes that client support queries.

Earlier log storage was maintained in SQL servers that became hard to process because data was growing exponentially. That case adds a reason for Rackspace to switch to Hadoop.

A distributed environment also allows to store duplicate records that are difficult to identify as unique. These need to have a unique identifier called message-id that is generated by a host at the originate point, but a bad client could easily send out duplicates. For that situation, there comes a concept of queue-id that certainly unique a lifetime of that message on the local machine. Following are the lines that show log processing for data of mail:

```
Nov 12 17:36:54 gate8.gate.sat.mlsrvr.com  
postfix/smtpd[2552]: connect from hostname
```

```
Nov 12 17:36:54 relay2.relay.sat.mlsrvr.com  
postfix/qmgr[9489]: 1DBD21B48AE:
```

```
from=<mapreduce@rackspace.com>, size=5950, nrcpt=1  
(queue active)
```

```
Nov 12 17:36:54 relay2.relay.sat.mlsrvr.com  
postfix/smtpd[28085]: disconnect from  
hostname
```

```
Nov 12 17:36:54 gate5.gate.sat.mlsrvr.com  
postfix/smtpd[22593]: too many errors  
after DATA from hostname
```

```
Nov 12 17:36:54 gate5.gate.sat.mlsrvr.com  
postfix/smtpd[22593]: disconnect from  
hostname
```

Nov 12 17:36:54 gate10.gate.sat.mlsrvr.com  
postfix/smtpd[10311]: connect from  
hostname

Nov 12 17:36:54 relay2.relay.sat.mlsrvr.com  
postfix/smtp[28107]: D42001B48B5:  
to=<mapreduce@rackspace.com>, relay=hostname[ip],  
delay=0.32, delays=0.28/0/0/0.04,  
dsn=2.0.0, status=sent (250 2.0.0 Ok: queued as  
1DBD21B48AE)

Nov 12 17:36:54 gate20.gate.sat.mlsrvr.com  
postfix/smtpd[27168]: disconnect from  
hostname

Nov 12 17:36:54 gate5.gate.sat.mlsrvr.com  
postfix/qmgr[1209]: 645965A0224: removed

Nov 12 17:36:54 gate2.gate.sat.mlsrvr.com  
postfix/smtp[15928]: 732196384ED: to=<m  
apreduce@rackspace.com>, relay=hostname[ip],  
conn\_use=2, delay=0.69, delays=0.04/  
0.44/0.04/0.17, dsn=2.0.0, status=sent (250 2.0.0  
Ok: queued as 02E1544C005)

Nov 12 17:36:54 gate2.gate.sat.mlsrvr.com  
postfix/qmgr[13764]: 732196384ED: removed

Nov 12 17:36:54 gate1.gate.sat.mlsrvr.com  
postfix/smtpd[26394]: NOQUEUE: reject: RCP  
T from hostname 554 5.7.1  
<mapreduce@rackspace.com>: Client host rejected:

The

```
sender's mail server is blocked; from=
<mapreduce@rackspace.com> to=<mapred
uce@rackspace.com> proto=ESMTP
he lo=mapreduce@rackspace.com
```

MapReduce is the best way to deal with it by making a pair of key-value pairs from message logs. First, need to map all lines with a unique queue-id and pass it to reduce phase to determine the log message value as a single line. After completing that, there is a need to group by message-id as key and list of log lines as value. All these are processed with MapReduce to make the system more convenient to read out all logs and process them easily.

## **Hadoop at Facebook**

It was a great challenge before Facebook, when it started growing. The rate of increment in logs and dimensions data exponentially increases, which requires a technique to perform it on a daily basis. For that purpose, Facebook used Hadoop at its center and started functioning with a Hive. For that scenario, it was required to store it in a distributed manner so that it could be scalable at any time with any rate of scalability. Hadoop provides reliable resources that maintain a system for processing.

Initially, Facebook used Oracle for processing of entire database. However, Oracle was not reliable as data was rapidly increased at its center, and Facebook became an indirect user for the processing of data. That can be harmful for them in the future. So, Facebook looks for open-source techniques for investigation purposes. Hadoop also became attractive because Yahoo was using it successfully, and they already deployed clusters for processing. NYT also showed an example of attraction. Google also has given experimental proof by using GFS in a reasonable manner.

All wanted to use compatible favorite artificial language for a process (using Hadoop streaming). This made it even easier for users by having the ability to precise common computations within the style of SQL, a language with

which most engineers and analysts measure. Nowadays, Facebook is running the second largest Hadoop cluster in the world, as it holds 10 TB of information daily in its Hadoop cluster, all of which are unstructured. They claim that the Hadoop instance has two 400 cores and concerning 9 TB of memory and runs at 100% utilization at several points throughout the day, and all of them are able to scale out this cluster rapidly in response to growth with modification of cluster in Hadoop.

There are at least four interrelated but distinct classes of uses for Hadoop on Facebook:

- Producing daily and hourly summaries over large amounts of data. These summaries are used for a number of different purposes within the company, like reports based on these summaries are used by engineering and non-engineering functional groups to drive product selections. These summaries embody reports on the growth of the users, page views, and average time spent on the location by the users and provide performance numbers concerning promotional material campaigns that square measure run on Facebook and backend process for website options like individuals.
- Running unexpected jobs over historical information. These analyses facilitate answering queries from our product teams and government teams.
- As a factual long depository store for our log data sets.
- To see up log events by specific attributes (where logs square measure indexed by such attributes), that is used to take care of the integrity of the location and defend users against spam bots.

## **Usages of big data**

In the present scenario, Hadoop is required for storing and processing purposes. This is not only needed from a technical point of view but also for expanding business. It has lots of usages in many different areas. The following are the fields in which Hadoop is now in use.

## **Machine learning tools**

Big data plays a pivotal role in enabling advanced machine learning use cases by providing the vast and diverse datasets necessary for training and improving machine learning models. In supervised learning, big data allows for the creation of more accurate and robust models by providing a larger pool of labeled examples. For instance, in natural language processing, big datasets like Common Crawl or Wikipedia text enable the development of sophisticated language models like GPT-3. These models can perform tasks such as text generation, translation, and sentiment analysis with remarkable accuracy because they have learned from the vast amount of text data available on the internet.

Furthermore, big data facilitates the development of unsupervised learning techniques such as clustering and dimensionality reduction. For example, when dealing with massive datasets in industries like finance or health care, dimensionality reduction techniques like **Principal Component Analysis (PCA)** can be used to extract meaningful patterns and reduce the computational burden. Similarly, clustering algorithms can help identify hidden structures within large datasets, aiding in customer segmentation, anomaly detection, and recommendation systems. In essence, big data empowers machine learning to tackle complex real-world problems by providing the necessary volume, variety, and velocity of data for training and inference, ultimately leading to more accurate and insightful predictions and decisions.

## **Entertainment fields**

Big data plays a significant role in the operations and success of streaming platforms such as Amazon Prime, Netflix, and YouTube. These platforms rely on data-driven decision-making to enhance user experiences, optimize content delivery, and improve their overall business performance. The following are some key ways in which big data is used in these streaming services:

- **Content recommendation and personalization:**
  - **Netflix:** Netflix uses big data to analyze user viewing history, preferences, and interactions. This data is then used to recommend

personalized content to each user. The **Recommended for You** section is a prime example.

- **Amazon Prime Video:** Amazon employs big data to suggest products and content to users based on their past shopping and viewing habits, creating a seamless shopping and streaming experience.
- **YouTube:** YouTube's recommendation system uses big data to suggest videos that are likely to keep users engaged and on the platform longer, increasing ad revenue.
- **Content creation and acquisition:**
  - **Netflix:** Netflix analyzes viewership patterns and preferences to inform decisions about which original content to produce or acquire. Hits like "Stranger Things" were greenlit based on data-driven insights.
  - **Amazon Prime Video:** Amazon uses big data to identify promising content for Prime Video and to guide its investments in original programming.
  - **YouTube:** YouTube collects data on trending topics and popular video genres to inform its content creators and provide them with insights into what content is likely to perform well.
- **Quality of Service (QoS):** All three platforms use big data to monitor and optimize the quality of video streaming. They collect data on factors like network performance, device capabilities, and user bandwidth to deliver the best possible streaming experience.
- **Content delivery optimization:** To ensure that users can stream content smoothly, these platforms use big data to optimize content delivery through **Content Delivery Networks (CDNs)**. This involves selecting the most efficient server locations and routing algorithms based on real-time data.
- **Audience analysis and Ad targeting:**
  - **Netflix:** While Netflix is primarily subscription-based and ad-free, it uses data for audience segmentation to understand its user base

better.

- **Amazon Prime Video:** Amazon uses data to target users with relevant ads in other parts of its ecosystem, such as the Amazon shopping website.
- **YouTube:** Google, which owns YouTube, uses big data to refine its ad targeting capabilities, ensuring that ads are shown to users who are more likely to engage with them.
- **Fraud detection and security:** All three platforms use big data to detect and prevent fraud, such as account sharing and illegal content uploads. They also use it to ensure the security of user data.
- **Viewer insights and analytics:** Big data is used to gain insights into viewer behavior, such as when and where they watch content. This information helps these platforms make data-driven decisions about content release schedules and marketing strategies.

Big data is an integral part of the operations of streaming platforms such as Amazon Prime, Netflix, and YouTube. It is used to enhance user experiences, optimize content delivery, make data-driven business decisions, and maximize revenue through targeted advertising and content recommendations.

## Web analytics

There is a scenario in which users want to read websites for reading newspapers from apps or other sources on the internet. This should be based on their own choices, as we all have also observed that on news sections, YouTube sections, or any e-commerce sites give preferences to our choices. They all provide solutions of desired results using Google Analytics or Omniture's click stream tool, by which top results can be seen. It is a kind of reading mind of users. People can read the sports section, World section, electronic section, or entertainment section. It happens because click stream collects piece of data that a page has to be rendered with code of analysis. Telemarketing is so popular nowadays because it has extracted databases from customer requirements and its feed database. In 1990 e-mail started to show variety in their database, so as affect people started to check e-mail as

an opportunity to enhance business through internet and the World Wide Web. As the market of internet extended itself and accelerated, it pulled the fashion of e-marketing that had an effect on launching campaigning of real-time marketing. Web analysis gives an opportunity for enhancement to record individual behavior.

## **Big data and marketing**

As all of us are living in a digital world so it is obvious to change according to users' need. Because marketing strategies are changing rapidly so there is race with time has been started.

In today's scenario, the customer is of many types; they used to spend their time online, offline, captivated, distracted, vocal, annoyed, or quiet at any moment. So they are unpredictable at any moment. So there is a strategy needs to be built for having relevant marketing as per demand, that is, called cross-marketing.

Once it involves information regarding the people whom an attempt has been made to sell. Marketers understand this so that appearance of massive information has been a serious blessing for them over the past few years. The tech-driven world of massive information encompasses a heap in common with the creativity-driven world of promoting hunger for a lot of info.

The quantified self-development is gaining momentum, and each side of a person's life will currently be measured, stored, calculated, and analyzed to derive conclusions that measure usefully from advertisers to politicians to sociologists.

All this information is a dream for marketers as targeting potential customers with such pinpoint accuracy has never been easier. The techniques marketers will currently use to approach audiences were just about not possible a decade ago.

Traditionally, old-school promoting efforts like e-mail subscriptions and newsletters have created means for contemporary promoting methods that

square measure supported people's Web browsing habits that measure very revealing.

For each native and international promotion, keep an in-depth eye on Google trend results. It is one of the foremost simple and approachable massive information sources offered online these days. The platform permits you to see, however, in style-bound search terms measure and the way they compare to average daily volumes. Choosing a replacement trend move into the information is straightforward, and it will facilitate information on everything from long campaign focuses to what hashtags to use day's Tweet.

One way of mistreatment massive information is mistreatment the Google Trends online tools. This is often among the simplest ways in which of mistreatment massive information. It provides the user with the trending topics that area unit associated with the search word that they need to enter. Marketers will filter the results supported by WHO they require to focus on either globally or regionally. It provides a broad variety of choices to line filter choices that embody location, regions, countries, and interests, among any others.

## **Big data and fraud**

Fraud is intentional deception created for private gain or to break individual privacy. The most happening dishonest activity is credit card fraud. The MasterCard fraud rate in USA and other countries is increasing. As per Javelin's analysis, "8th Annual Card Issuers' Safety Scorecard: Proliferation of Alerts result in faster Detection Time and Lower Fraud prices," MasterCard fraud incidence inflated by 87% in 2011, culminating fraud loss of \$6 billion. However, despite the significant increase in incidence, the total value of MasterCard fraud inflated solely 20%. The relatively little rise in total value may be attributed to an increasing sophistication of fraud detection mechanisms.

Even though fraud detection is rising, the speed of incidents is rising. This means banks want a lot of proactive approaches to fraud. While issuers' investments in detection and determination have resulted in a flow of customer-facing tools associated degreed falling average detection times

among MasterCard fraud victims, the rising incidence rate indicates that MasterCard issuers ought to rate preventing fraud. Social media and mobile phones are forming the new frontiers for fraud. Despite warnings that social networks, it is measuring a good resource for fraudsters who share a major quantity of personal data, often accustomed to evidence of a shopper's identity.

According to Javelin's *2012 Identity Fraud Report: Social Media and Mobile Forming the New Fraud Frontier*, 68% of individuals with public social media profiles shared their birthday data (with 45% sharing month, date, and year); 63 % shared their high school's name; 18 % shared their phone number; and 12 % shared their pet's name—all area unit prime samples of personal data an organization would use to verify your identity.

To stop the fraud, the MasterCard transactions area unit is monitored and checked in close to real-time. If the checks determine pattern inconsistencies and suspicious activity, the dealing is known for review and increase.

The Capgemini monetary Services team believes that because of the character of data streams and processes needed, massive knowledge technologies give AN optimal technology resolution supported by the subsequent 3 Vs:

- High volume. Years of client records and transactions (150 billion 1 records per year)
- High rate. Dynamic transactions and social media data.
- High selection. Social media and different unstructured knowledge like customer e-mails, and center conversations, further as transactional structured knowledge Capgemini's new fraud massive knowledge initiative focuses on drooping the suspicious MasterCard transactions to stop fraud in close to time period via multi-attribute watching.

Time period inputs involving dealing knowledge and customers records area unit monitored via validity checks and detection rules. Pattern recognition is performed against the info to get and weigh individual transactions across every one of the foundations and marking dimensions.

An additive score is then calculated for every dealing record and compared against thresholds to make your mind up if the dealing is probably suspicious or not.

The Capgemini team showed that they use an associate degree ASCII text file weapon named Elastic Search, which could be a distributed, open-source search server based on Apache Lucene. It will be accustomed to search all reasonable documents at close to a period of time. They use the tool to index new transactions, which are sourced over a period of time, that permits analytics to run in a distributed fashion using the info specific to the index. Victimization of this tool, giant historical data sets will be used in conjunction with a period of time knowledge to spot deviations from typical payment patterns. This huge knowledge part permits overall historical patterns to be compared and contrasted and permits the number of attributes and characteristics concerning client behavior to be terribly wide, with little impact on overall performance.

Once the group action knowledge has been processed, the percolator question then performs the functioning of characteristic new transactions that have raised profiles. Percolator could be a system for incrementally processing updates to giant knowledge sets. Percolator is the technology that Google employed in building the index that links keywords and URLs—used to answer searches on the Google page. Percolator questions will handle each structured and unstructured knowledge. This provides measurability to the event process framework and allows specific suspicious transactions to be enriched with extra unstructured information, phone location/geospatial records, client travel schedules, and so on. This ability to complement the dealings will scale back false positives and increase the expertise of the client, redirecting fraud efforts to actual instances of suspicious activity.

Another approach to finding fraud with huge information is **Social Network Analysis (SNA)**. SNA is the precise analysis of social networks. Social network analysis views social relationships and makes assumptions. SNA may reveal all people concerned in dishonest activity, from perpetrators to their associates, and perceive their relationships and behaviors to spot a bust-out fraud case.

According to a recent article in [bankersonline.com](#) announced by Experian, “bust out” could be a hybrid credit and fraud drawback, and therefore, the theme is usually defined by the subsequent behavior:

- The account in question is delinquent or charged-off.
- The balance is near or over the limit.
- One or more payments are coming back.
- The client cannot be settled.
- The on top of conditions exist with quite one account and/or money institution.

There are some huge information resolutions within the market, like SAS’s SNA solution, which helps establishments and goes on the far side of individual and account views to analyze all connected activities and relationships at a network dimension. The network dimension permits you to ascertain social networks and see hidden connections and relationships that probably may be a bunch of fraudsters.

## **Risk management in big data with credit card**

Many of the planet’s high analytics professionals add risk management. It would be sarcasm to mention that risk management is data-driven without advanced knowledge of analytics. The two commonest kinds of risk management square measure credit risk management and market risk management. Three kinds of risk are operational risk management, International Relations, and Security Networks, as common as credit and market risk.

The techniques for risk professionals usually embody avoiding risk, reducing the negative impact or chance of risk, or accepting some or all the potential consequences in exchange for a possible top-side gain.

Credit risk analytics concentrate on past credit behaviors to predict the chance that a receiver can neglect any kind of debt by failing to create payments, which they are indebted to try to do. As an example, *Is this person seemingly to neglect their \$300,000?* Market risk analytics

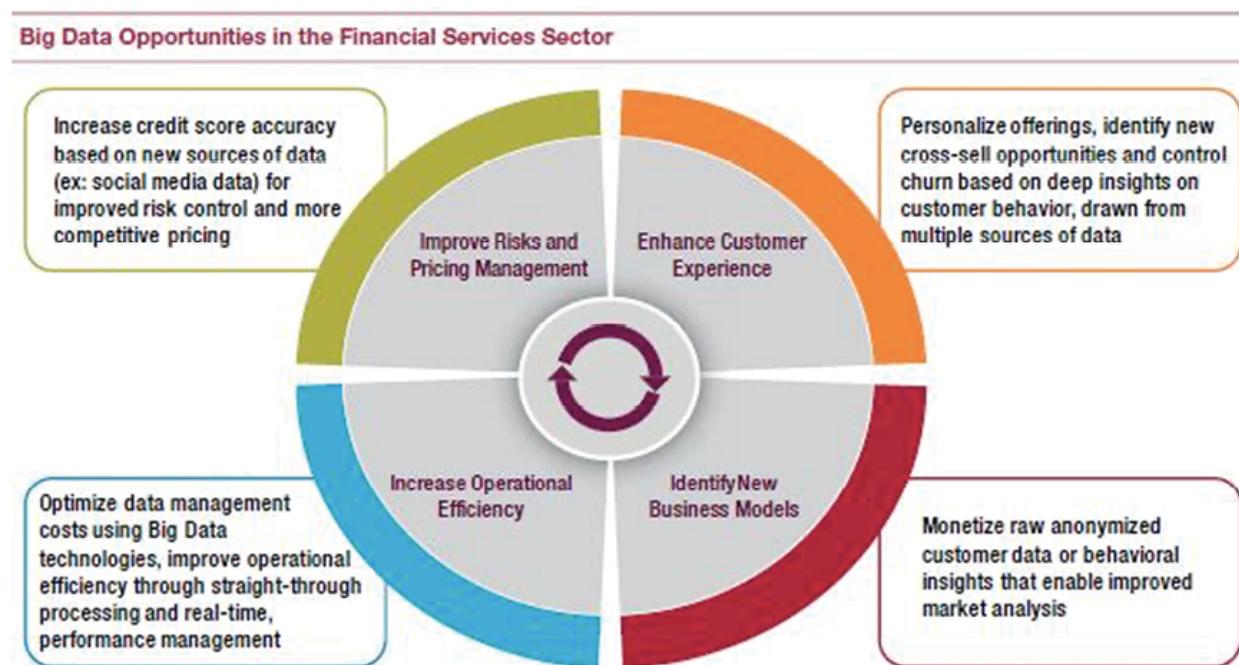
concentrate on understanding the chance that the worth of a portfolio can decrease owing to the modification available costs, interest rates, foreign exchange rates, and trade goods costs. As an example, *Should we tend to sell this holding if the worth drops another ten percent?*

Traditionally, credit risk management was non-moving within the philosophy of minimizing losses. However, over time, credit risk professionals and business leaders came to meet for acceptable levels of risk, which will boost profit on the far side, which would usually be achieved by merely focusing on avoiding write-offs. The shift to the additional profitable credit risk management approach has been assisted in expanding the availability of knowledge, tools, and advanced analytics. Credit risk professionals square measure stakeholders in key selections that address all aspects of a business, from finding new and profitable customers to maintaining and growing relationships with existing customers. Increasing the chance and reward opportunities requires that risk managers perceive their client portfolio, permitting them to leverage a uniform credit approach while acknowledging that they treat each client constantly. As businesses grow, what starts out as a manual and faultfinding method of making credit selections offers which appreciate to an additional structured and progressively automatic process during which data-driven selections become the core.

The huge quantity of each qualitative info obtainable to credit risk professionals will be overwhelming to digest and might slow down a method with potential sales in danger. With advanced analytical tools, these bumper and complicated information sources will be distilled into implemented straightforward solutions. Ancient marking ways target predicting the chance of delinquency or bankruptcy; however, further marking solutions also can facilitate firms establish the gain potential of consumers or, from a promoting perspective, the propensity to pay. The credit risk management lifecycle, wealthy information sources, and advanced analytics square measure instrumental throughout from a client acquisition perspective; credit risk managers decide whether or not to increase credit and how much. Lacking any previous expertise with the prospect, they rely heavily on third-party credit reports and scores and should assist promoting organizations in

using made-to-order look-alike models to assist in establishing prospective best customers, as shown in *Figure 1.4*.

The future of credit risk management can still be amended as it has a tendency to leverage new information sources emanating from an extremely digital and mobile world. As an example, social media and mobile phone usage information measure gaps in new opportunities to uncover client behavior insights that may be used for credit decisions.



*Figure 1.4: Credit card system in the financial sector*

## Big data and algorithm trading

*Partha Sen* is the business executive of Fuzzy Logix, a corporation that focuses on high-performance, cross-platform information and **graphics process unit (GPU)** analytics. Sen spent over 15 years as a quantitative analyst within the financial services business. Over the course of his career, he developed over 700 extremely parallelized algorithms. He, together with a team of terribly gifted quantitative professionals, currently leverages his formidable experience to assist customers across a variety of industries.

Sen has seen a big shift in the use of knowledge within the money services industry over the past decade. Financial establishments, he says, particularly investment banks, are at the forefront of applying analytics for risk management, proprietary commercialism, and portfolio management. As most of you recognize, several investment banks use recursive commercialism, an extremely subtle set of processes within which insights area units created actionable via machine-driven decisions. Recursive commercialism depends on subtle mathematics to work out get and sell orders for equities, commodities, interest rate and exchange rates, derivatives, and stuck financial gain instruments at glaring speed. Risk analysts facilitate banks to develop mercantilism rules and implement these rules to mistreat trendy technology. An algorithm involves a huge range of transactions with complicated mutually beneficial knowledge and every millisecond matters.

It is honest to mention that, of late, banks focus a lot of closely on market risk today than ever before. Market risk is largely the chance owing to a fluctuation in the price of assets within the marketplace. For a given portfolio, attempting to determine the likelihood that the worth of the portfolio can fall at intervals of a certain threshold at intervals of five days, at intervals of seven days, and at intervals of one month. With asset volatilities as high as they need been determined within the previous few years, a lot of stress is being placed on market risk.

Apart from investment banks, company, and retail banks additionally swear terribly heavily on quantitative techniques. Two area units that return to mind are marketing, where they solicit households for financial merchandise such as credit cards, and credit risk management, where banks attempt to perceive the likelihood that borrowers can neglect loans. The models used in these areas for future outcomes are units created with a large range of variables. For instance, a model of the default risk for credit cards may well be influenced by demographic factors, whether individuals have employment or not, what is the expansion within the economy, and interest rates. There may be many factors or variables for every credit card. A typical retail bank can judge somewhere north of 5,000 factors for one given model to determine or calculate the likelihood of every one of the borrowers defaulting. The quantity of calculations only for the chance issue will simply

climb into billions of calculations being performed to calculate risk for a portfolio.

To maintain competitive advantage, banks must be compelled to unceasingly measure their models, together with the performance of the assembly models, and conjointly unceasingly try to build new models to include new variables with new and evolving economic conditions in a very quicker means. Banks have conjointly captive from daily risk management to intraday risk management. Intraday risk management involves the evaluation of the whole portfolio and the calculative chance limits of each of the counterparties at intervals in the bank's portfolio. The matter gets terribly complex and computationally intensive. Let us take an example of risk analysis of equities. The potential changes at intervals daily embody the damage, the volatility of the underlying equity, and the innocent rate. If we tend to do some basic state of affairs analysis—say 100 risk-free rate eventualities that would manifest themselves throughout the course of the day, that means that shrewd a hundred eventualities for the terms of the equity throughout the course of the day, a hundred eventualities for volatility throughout the course of the day, and a hundred eventualities for safe rate throughout the course of the day. For the bank to try its basic situation analysis, it takes 1,000,000 calculations to determine the worth in danger for simply that one instrument. It should happen quickly enough so risk limits on the whole portfolio can be evaluated many times throughout the course of the day.

## **Big data in health care**

Big knowledge guarantees a colossal revolution in health care, with vital advancements in everything from the management of chronic sickness to the delivery of personalized drugs. Additionally, to save and raise lives, huge knowledge has the potential to rework the complete health care system by replacing guess and intuition with objective, data-driven science.

The US health-care system is progressively challenged by problems with price and access to quality care. Payers, producers, and suppliers are trying to realize improved treatment outcomes and effective benefits for patients within a disconnected health-care framework. Traditionally, these health-

care ecosystem stakeholders tend to figure at cross functions with alternative members of the health-care price chain. High levels of variability and ambiguity across these individual approaches increase prices, scale back overall effectiveness, and impede the performance of the health-care system as an entire.

Recent approaches to health care reform conceive to improve access to health care by increasing government subsidies and reducing the ranks of the uninsured. One outcome of the recently passed Responsible Care Act may be a revitalized specialization in price containment and, therefore, the creation of quantitative proofs of economic benefit by payers, producers, and suppliers. The additional fascinating unintended consequence is a chance for these health-care stakeholders to set aside historical variations and build a combined counterbalance to potential restrictive burdens established while not the input of the industry or the government is taking off to control.

The health-care system is facing severe economic, effectiveness, and quality challenges. These external factors are forcing a metamorphosis of the pharmaceutical business model. Health care challenges are forcing the pharmaceutical business model to undergo fast modification. Our business is moving from a standard model designed on restrictive approval and subsidence of claims to one of medical proof and proving economic effectiveness through improved analytics-derived insights. The success of this new business model is hooked into having access to information created across the complete health-care system.

## **Conclusion**

Big data has transformed the way organizations process, analyze, and utilize data to gain valuable insights for better decision-making. Hadoop, a powerful open-source framework, has emerged as a key technology in handling large, diverse, and complex datasets. With its distributed file system, MapReduce processing engine, and various ecosystem components, Hadoop provides a scalable, fault-tolerant, and cost-effective platform for big data processing and analysis. The versatility of Hadoop has made it popular across a wide range of industries, including finance, health care, retail, and more. As big data continues to grow in size and complexity,

Hadoop is expected to play a crucial role in enabling organizations to unlock the full potential of their data and gain a competitive edge in the market.

In the upcoming chapter, we will learn about NoSQL data management.

### **Join our book's Discord space**

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

[https://discord\(bpbonline\).com](https://discord(bpbonline).com)



## CHAPTER 2

# NoSQL Data Management

### Introduction

Nowadays, **Relational Database Management Systems (RDBMS)** is the predominant technology for storing structured data in Web and business applications. The publication of Codd's paper titled A Relational Model of Data for Large Shared Data Banks in 1970, these data repositories, which rely on the relational calculus and offer extensive ad hoc querying capabilities through SQL, have been widely embraced and are commonly regarded as the exclusive option for storing data that can be accessed consistently by multiple clients.

Although there are totally different approaches over the years, like object databases or XML stores, these technologies have not gained a similar adoption and market share as RDBMSs. Rather, these alternatives have either been absorbed by RDBMS that enables storing XML and use it for functions like text classification, or they need to become niche products, for example, OLAP or stream processing.

NoSQL was used for the first time in 1998 to suggest switching from traditional system RDBMS to NoSQL. This again raised its usability in 2009 with strong recommendations for that system to change from the existing one. Last.fm developer Jon Oskarsson organized a meeting to show its usage and popularity in San Francisco in which most technocrats agreed to adopt the changes.

In contrast to RDBMS systems, most NoSQL databases are designed to scale well in the horizontal direction and do not believe exclusive hardware is purposely used. Machines are often additional and removed (or crash) without inflicting equivalent operational efforts to perform sharding in RDBMS cluster-solutions; some NoSQL datastores even give automatic sharding. Javier Soltero, CTO of SpringSource, puts it this way: *Oracle would tell you that with the correct degree of hardware and also the right configuration of Oracle RAC (Real Application Clusters) and alternative associated magic code*, Last.fm states: *Web two.0*

*corporations will take a chance and that they would like quantifiability. Once you have these two things together, it makes [NoSQL] terribly compelling.* Blogger Nati Shalom agrees with that: *cost pressure conjointly forced several organizations to look at less expensive alternatives, and therewith came analysis that showed that distributed storage-supported goods hardware is even additional reliable than several high-end databases.*

NoSQL represents a completely different information model from earlier models, such as hierarchical (which represents relationships in a strict hierarchy) and network (which represents many to many relationships by introducing the notion of record sets).

## **Structure**

In this chapter, we will go through the following topics:

- Terminologies of NoSQL and RDBMS
- Different databases in NoSQL
- Difference between traditional database and NoSQL
- Consistency issues with NoSQL
- Introduction of HBase

## **Objectives**

The purpose of this NoSQL chapter is to provide an overview and understanding of NoSQL databases, their characteristics, and their applications. It aims to explore the motivations behind the development of NoSQL databases, the key features they offer, and the trade-offs they make compared to traditional relational databases.

The chapter also discusses the different types of NoSQL databases, such as document stores, key-value stores, columnar databases, and graph databases, highlighting their strengths and suitable use cases. Furthermore, it may delve into the scalability and performance advantages of NoSQL databases, and considerations for data modeling and querying in a NoSQL environment. Overall, this chapter aims to provide readers with a comprehensive understanding of NoSQL databases and their role in modern data management systems.

## **Terminology used in NoSQL and RDBMS**

**RDBMS:** Partitions, Table, Row, Column

**NoSQL:** Shard, Document root element (JSON/XML), Aggregated, Attribute/field

## Database used in NoSQL

The following are the four types of databases that are used by NoSQL:

### Key–value database

In this kind of database, all records are stored in key–value pairs. The key is unique, and value can be a whole line which relevant to the key. This helps in the quick retrieval of data using a key. Data can also refer using key for further use. Examples of key values are as follows:

- URL and its Web contents
- Account number and account holder name
- Page number and contents
- Roll number and student name

It is possible to refer data using key, but there is no implicit ordering. The key can further be changed with the condition of uniqueness with its record. Following are the examples of usages of key–value in the industry:

- **Amazon's dynamo** is an example of a database that uses a key–value database of NoSQL. They use commodity hardware, standard mode of operation, loosely coupled, and service-oriented architecture of hundreds of services. Because of using commodity hardware usages, it is scalable in nature. Objects stored with versioned data. To maintain consistency during updates, Dynamo uses a quorum-like technique and a protocol for decentralized replica synchronization. *Table 2.1* shows the problem that dynamo handles with key–value databases.

In Dynamo, all nodes have equal responsibilities; there are not any distinguished nodes having special roles. In addition, its design favors *decentralized peer-to-peer techniques over centralized control* because the latter has *resulted in outages* in the past at Amazon. Storage hosts added to the system will have heterogeneous hardware that dynamo must consider distributing work *proportionally to the capabilities of the individual servers*. As Dynamo is operated in Amazon's own administrative domain, the

environment and all nodes are considered non-hostile, and hence, no security-connected options such as authorization and authentication are implemented in Dynamo.

As Dynamo is meant to be *always writable* (that is, a data-store that is extremely available for *write conflict* resolution must happen throughout reads. If application developers do not need to implement such a business logic-specific reconciliation strategy, Dynamo also provides easy methods they can simply use, such as last write wins a timestamp-based reconciliation.

*System interface* of dynamo consists of two operations that are used to interact with users are as follows:

- **get(key)**, returning a list of objects and a context.
- **put(key, context, object)**, with no return value

With **get** operation more than one object can be read with key. It also returned system metadata as an object version stored. Put operation can have a context object as a parameter. Key and object values are not interpreted by Dynamo but handled as *an opaque array of bytes*. The key is hashed by the MD5 algorithm to determine the storage nodes accountable for this key-/value-pair:

Characteristics	Techniques	Advantages
1. Partitioning	Consistent hashing	Incremental scalability
2. High availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
3. Handling temporary failures	Sloppy Quorum and hinted hand off	Provides high availability and durability guarantee when some of the replicas are not available.
4. Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
5. Membership and failure	Gossip-based membership	Preserves symmetry and avoids having a centralized registry for storing

detection	protocol and failure detection.	membership and node liveness
-----------	---------------------------------	------------------------------

**Table 2.1:** Amazon's Dynamo

To provide incremental scalability, Dynamo uses consistent hashing to dynamically partition data across the storage hosts that are present in the system at a given time. To ensure scalability and availability of data in Dynamo, use replication factor with N nodes. Each data is replicated by N times where N can be configured *per-instance* of Dynamo. An *instance* typically refers to a single data record or observation within a dataset. An instance represents a single data point or a specific unit of information.

- **Project Voldemort:** Project Voldemort is a key-/value-store initially developed for and still used at LinkedIn. Key and value are automatically replicated over multiple servers, and data is automatically partitioned so each server contains only a subset of the total data:
  - get(key), returning a value object
  - put(key, value)
  - delete(key)

Key and value that are used in its data-store can be complex and consist of lists and maps. It has been claimed in the project that, as compared to relational databases, its database is simply designed, and the API of key-value data-store is not complex. [Figure 2.1](#) shows the architecture of its design pattern:

# Logical Architecture

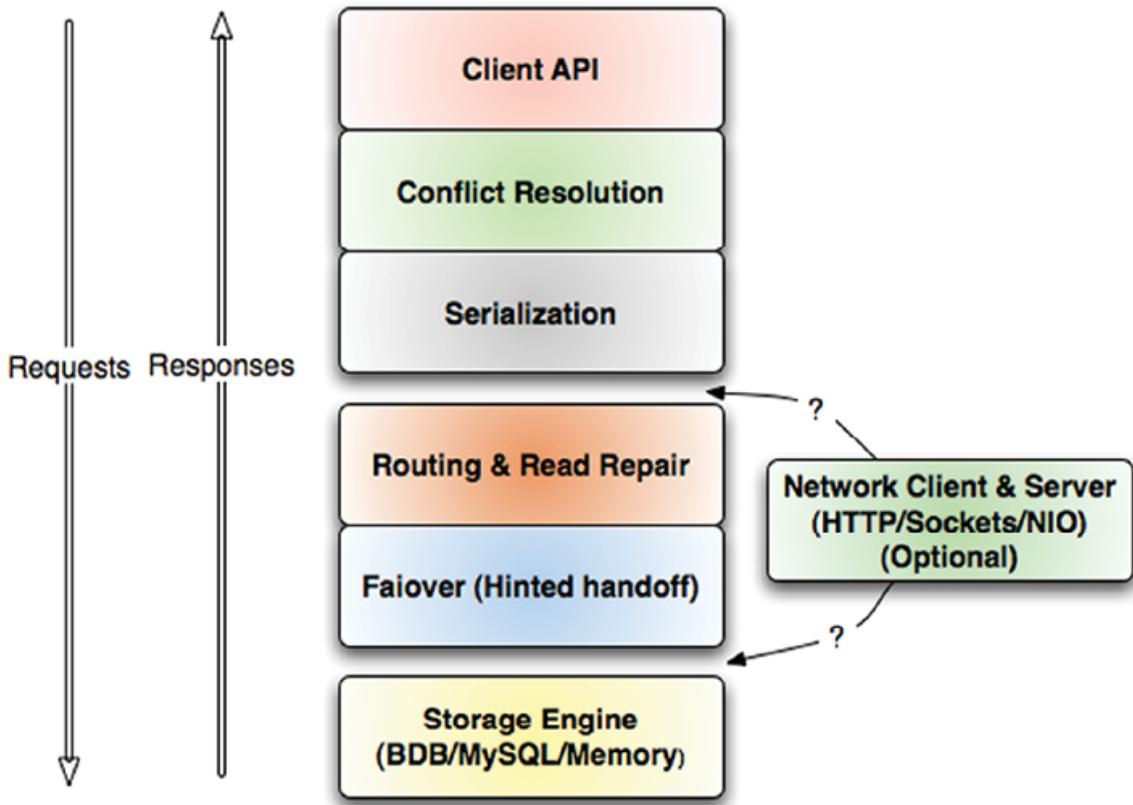


Figure 2.1: Design diagram of Voldemort

Every layer of this performs its own function for the operation of get, put, and delete. If, for example, the put operation is invoked on the routing layer, it is accountable for distributing this operation to all nodes in parallel and for possible errors.

Project Voldemort permits namespaces for key-/value-pairs known as **stores**, in which keys are distinctive. While each key's related to precisely one value, values are allowed to contain lists and maps also as scalar values. Operations in Project Voldemort are atomic to precisely one key-/value-pair. Once a get operation is executed, the value is streamed from the server via a pointer. Documentation of Project Voldemort considers this approach to not work all right together with values consisting of large lists that *should be kept on the server and streamed lazily via a cursor*; in this case, breaking the query into subqueries is seen as a lot of efficient.

Project Voldemort offers possibilities of data types used in:

Type	Storable Sub-types	Bytes used	Java-Type	JSON Example	Definition Example
Number	int8, int16, int32, int64, float32, float64, date	8, 16, 32, 64, 32, 64, 32	Byte, Short, Integer, Long Float, Double, Date	1	“int32”
String	string, bytes	2 + length of string or bytes	String, byte[]	“hello”	“string”
Boolean	Boolean	1	Boolean	true	“boolean”
Object	object	1 + size of contents	Map<String, Object>	{“key1”:1, “key2”:”2”, “key3”:false}	{“name”:”st “height”:”in
Array	array	Size * sizeof(type)	List<?>	[1, 2, 3]	["int32"]

**Table 2.2:** JSON Serialization format data types

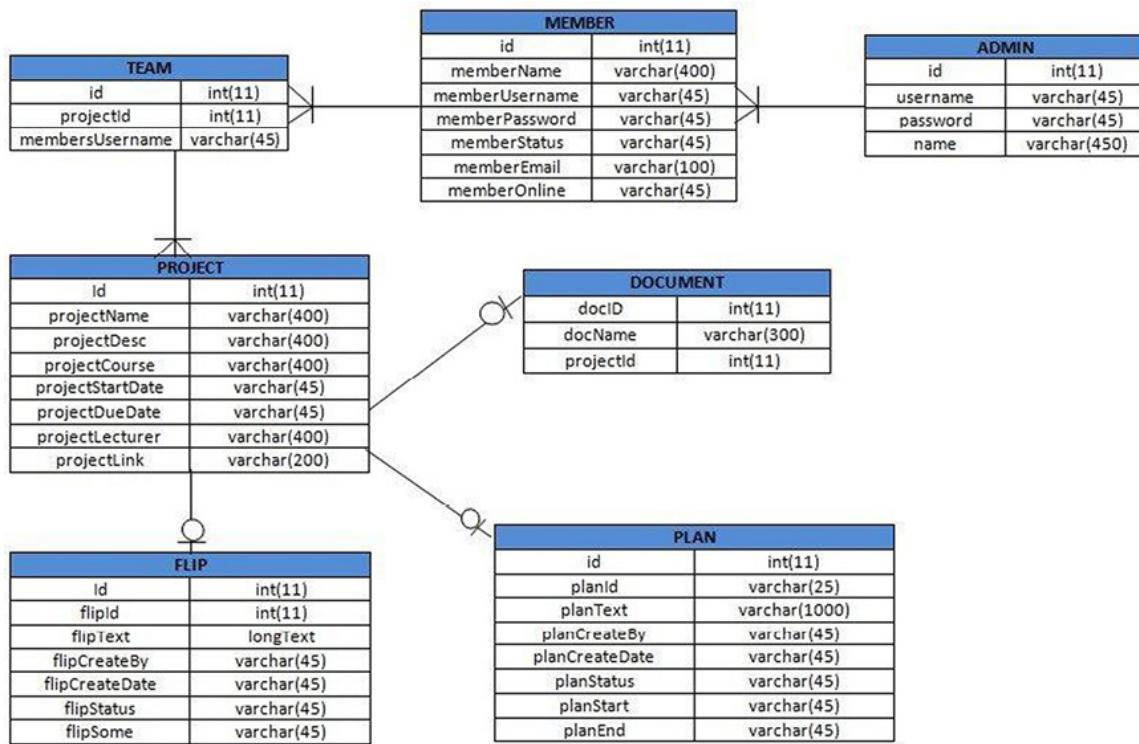
The data type definition for the **JSON (JavaScript Object Notation)** serialization format shown in [Table 2.2](#) allows project Voldemort to check values and store them with efficiency (as the JSON is stored in binary format), albeit the data types for values cannot be advantage for data queries and requests. To prevent invalidated data caused by the redefinition of value data types, project Voldemort is storing a version along with the data, allowing schema migrations.

- **Tokyo cabinet and Tokyo tyrant** are datastore that also build on key–value pair of databases. Tokyo cabinet is the core library of this data persistence and extracts data based on B++ tree structure or hash indexes. This data-store compresses pages by LZW algorithm that satisfies the output by giving a good compression ratio and partitioning data automatically with a similar

approach to SQL. With respect to the order of keys, it can provide a lookup with matches of results. The Toyko suite is developed actively, well documented, and widely regarded as high-performance: 1 million records can be stored in 0.7 seconds using the hash-table engine and in 1.6 seconds using the b-tree.

## Document database

Document (sometimes known as aggregate) databases are very much like key-value databases, except that the worth associated with a key contains structured or semi-structured data, which can be labeled as a document. In contrast to in a key-value database, there is a question against the structure of the document as well as components inside that structure and returns only parts of the document because of the results of the query. An example of document-oriented information could be a book database in which the key is the book title and the value is book metadata expressed as an XML document or JSON, as shown in [Figure 2.2](#):



*Figure 2.2: Document database*

Apache CouchDB and MongoDB are the two-leading symbolic for the class of document databases:

## Apache CouchDB

Many of technocrats also call it a *Cluster of unreliable commodity hardware* of document databases that is written or implemented in Erlang. CouchDB will be considered a descendant of Lotus Notes that CouchDB's main developer, Damien Katz, worked at IBM before he later initiated the CouchDB project on his own. A lot of ideas from Lotus Notes can be found in CouchDB, documents, views, distribution, and replication between servers and clients. The approach of CouchDB is to make such a document database from scratch with technologies of the Web space like representational State Transfer, JSON as a data interchange format, and also the ability to integrate with infrastructure elements like load balancers and caching proxies, and so on. CouchDB can be shortly characterized as a document database that is accessible via a restful HTTP-interface, containing schema-free documents in a flat address area. For these documents, JavaScript functions choose and aggregate documents and representations of them in a MapReduce manner to make views of the database that also get indexed. CouchDB is distributed and ready to replicate between server nodes similarly to clients and servers incrementally. Multiple concurrent versions of an equivalent document are allowed in CouchDB, and the database is ready to detect conflicts and manage their resolution, which is delegated to client applications. The foremost notable use of CouchDB in production is Ubuntu, the cloud storage and replication service for Ubuntu Linux. CouchDB is additionally a part of the BBC's new net application platform. Moreover, some blogs, wikis, social networks, Facebook apps, and smaller internet sites use CouchDB as their datastore.

CouchDB databases are addressed via a RESTful HTTP interface that allows one to read and update documents. As per its name, document database CouchDB is the database that is based on documents with key name and value. Document in the database cannot be nested as per its limitations.

```
" Title " : " CouchDB " ,  
" Last editor " : "192.59.223.99" ,  
" Last modified " : "25/10/1985" ,  
" Categories " : [ " Database " , " NoSQL " , " Document  
Database " ] ,  
" Body " : " CouchDB is a ..." ,  
" Reviewed " : yes
```

**Functioning:** Besides fields, documents may additionally have attachments, and CouchDB maintains some metadata like a unique symbol, revision number, and a sequence id for every document. The document id is a 128-bit value; the revision number could be a 32-bit value determined by a hash-function. CouchDB considers itself as semi-structured information. Whereas relational databases are designed for structured and interdependent data and key-/value-stores operate on uninterpreted, isolated key-/value-pairs document databases like CouchDB pursue a third path: data is contained in documents that do not correspond to a set schema (schema-free); however, have some inner structure known to applications further as the database itself. The benefits of this approach are that: first, there is no requirement for schema migrations, which cause lots of effort within the relative databases world; second, compared to key-/value-stores data can be evaluated a lot of sophisticatedly (for example, within the calculation of views). Within the internet application field, there are lots of document-oriented applications that CouchDB addresses as its data model fits this category of applications, and therefore, the possibility to iteratively extend or change documents may be done with lots less effort compared to a relational database. Each CouchDB database consists of precisely one flat/non-hierarchical namespace that contains all the documents that have a unique symbol calculated by CouchDB. A CouchDB server will host more than one among these databases. Documents were once stored as XML documents; however, nowadays they are serialized in a JSON-like format to disk. Document indexing is done in B-Trees that index the document's id and revision number (sequence id, column family).

**Views:** CouchDB's way to query, present, combine, and report the semi-structured document data are views. A typical example for views is to separate differing kinds of documents (such as journal posts, comments, and authors in a Web log system) that are not distinguished by the database itself as all of them are simply documents to that. Views are outlined by JavaScript functions that neither amend nor save or cache the underlying documents but solely gift them to the requesting user or client application. Thus, documents additionally as views (which are really special documents, referred to as design documents) may be replicated, and views do not interfere with replication. Views are calculated on demand. There is no limitation concerning the quantity of views in one database or the quantity of representations of documents by views. The JavaScript functions process a view referred to as map and reduce with similar responsibilities as in Google's MapReduce approach. The map gets a document as a parameter, can do any calculation, and should emit arbitrary data for it if it matches the view's criteria; if the given document does not match these criteria, the map operates and emits

nothing. Samples of emitted information for a document are the document itself, extracts from it, and references to or contents of alternative documents (for example, semantically related ones like the comments of a user in a forum, journal, or wiki). The data structure emitted by the map operation may be a triple consisting of the document ID, a key, and a value, which can be chosen by the map function. Documents get sorted by the key that does not have to be compelled to be unique; however, it will occur for over one document; the key as a sorting criterion may be used to, for example, define a view that sorts journal posts descending by date for a blog's home page. The value emitted by the map function is optional and should contain arbitrary data.

The document ID is set by CouchDB implicitly and represents the document that was given to the emitting map function as an argument. After the map function has been executed, its results get passed to an optional reduce function, which is optional but can do some aggregation on the view. As all documents of the database are processed by a view's functions, this can be time-consuming and resource-intensive for large databases. Therefore, a view is not created and indexed when write operations occur but on demand (at the first request directed to it) and updated incrementally when it is requested again. To provide incremental view updates, CouchDB holds indexes for views. As mentioned before, views are defined and stored in special documents as in a B-tree structure, similar to the structure holding the documents. The B-tree structure enables CouchDB for very fast lookups of rows by key and efficient streaming of rows in a certain key range. These design documents can contain functions for more than one view if they are named uniquely. View indexes are maintained based on these design documents, and no single views are contained in them. Hence, if a user requests a view, its index and the indexes of all views defined in the same design document get updated. Incremental view updates furthermore have the precondition that the map function is required to be referentially transparent, which means that for the same document, it has to emit the same key and value each time it is invoked. To update a view, the component responsible for it (called view-builder) compares the sequence id of the whole database and checks if it has changed since the last refresh of the view. If not, the view-builder determines the documents changed, deleted, or created since that time; it passes new and updated documents to the view's map, reduces functions, and removes deleted documents from the view. As changes to the database are written in an append-only fashion to disk, the incremental updates of views can occur efficiently as the number of disk head seeks is minimal. A further advantage of the append-only index persistence is that the system crashes during the update of indexes, the previous state remains

consistent, CouchDB omits the incompletely appended data when it starts up, and it can update an index when it is requested the next time. While the view-builder is changing, view data from the view's recent state will be scanned by clients. It is also possible to present the previous state of the view to one client and the new one to a different client as view indexes are also written in an append-only manner, and the compaction of view data does not omit a recent index state, whereas a client remains reading from it.

**Versioning:** Versioning of document represent different types of documents that are updated from time to time according to modification. Documents are updated optimistically, and update operations do not imply any locks. If an update is issued by some client, the contacted server creates a replacement document revision in a copy-on-modify manner, and a history of recent revisions is stored in CouchDB till the database gets compacted subsequent time.

A document, therefore, is known by a document id/key that sticks to that till it gets deleted, and a revision number is created by CouchDB once the document is formed and every time it is updated. If a document is updated, not only this revision number is stored but also a list of revision numbers preceding it to permit the database (when replicating with another node or processing read requests) as well as client applications to reason on the revision history within the presence of conflicting versions. CouchDB does not think about version conflicts as an exception but rather a standard case. They will not only occur by totally different clients operating on a similar CouchDB node but also as a result of clients operating on different replicas of a similar database. It is not prohibited by the database to have a vast number of concurrent versions. A CouchDB database will deterministically discover that versions of documents succeed each other and that are in conflict and have to be resolved by the client application. Conflict resolution may occur on any duplicate node of a database as the node that receives the resolved version transmits it to all replicas that have to accept this version as valid. It should occur that conflict resolution is issued on different nodes concurrently; the locally resolved versions on each node then are detected to be in conflict and get resolved similar to all different version conflicts. Version conflicts are detected at read time, and the conflicting versions come to the client, who is responsible for conflict resolution. A document with the most up-to-date versions in conflict is excluded from views.

**Distribution and replication:** CouchDB follows a peer pattern to set up server distribution without any individual roles (such as in master/slave-setups, standby-clusters, and so on). Totally different database nodes will, by design, operate fully

independently and process read and write requests. Two database nodes will replicate databases (documents, document attachments, and views) bilaterally if they reach one another via a network.

The replication method works incrementally and may detect conflicting versions in a simple manner, as every update of a document causes CouchDB to create a new revision of the updated document, and a listing of out-of-date revision numbers along with the key value is stored. By this revision number, as well as the list of out-of-date revision numbers, CouchDB will verify if they are conflicting or not; if there are version conflicts, each node has a notion of them and may increase the conflicting versions to clients for conflict resolution; if there are no version conflicts the node not having the most recent version of the document updates it. Distribution scenarios for CouchDB include clusters, offline-usage on a notebook, or at company locations distributed over the planet wherever live access to a company's or organization's local network is slow or unstable. Within the latter two scenarios, one will work on a disconnected CouchDB instance and is not restricted in its usage. If the network association to duplicate nodes is established again, the database nodes can synchronize their state again.

The replication method operates incrementally and document-wise. Incrementally means that only data changed since the last replication gets transmitted to a different node and not even whole documents are transferred but only modified fields and attachment blobs; document-wise means that every document successfully replicated does not need to be replicated again if a replication method crashes. Besides replicating whole databases, CouchDB also permits partial replicas. For these, a JavaScript filter function is often outlined that passes through the data for replication and rejects the remainder of the database. This partial replication mechanism is often used to share information manually by shaping totally different filters for every CouchDB node. If this cannot be used and no extension like Lounge CouchDB replicates all data to all nodes and will no sharding automatically and thus, by default, behaves just like MySQLs replication.

According to the CouchDB documentation, the replication mechanisms are designed to distribute and replicate databases with very little effort. On the other hand, they should even be ready to handle extended and more elaborate distribution scenarios, for example, divided databases or databases with full revision history. To achieve this CouchDB replication model can be changed for different distributed update models. As an example, the storage engine can be *enhanced to permit multi-document update transactions* to make it possible to *perform Subversion-like all or nothing atomic commits once replicating with an*

*upstream server, specified any single document conflict or validation failure can cause the whole update to fail* a lot of data on validation are often found as follows.

CouchDB never overwrites committed information or associated structures, so an information file is in a consistent state at each time. Hence, the database also does not want a shutdown to terminate correctly, but its method will simply be killed.

Document updates are executed in two stages to produce transactions while maintaining the consistency and durability of the database:

- The updated documents are serialized to disk synchronously. An exception to the current rule is BLOB data in a document that gets written concurrently.
- The updated info header is written in two identical and subtenant chunks to disk.

Now, if the system crashes when step one is executed, the incompletely written information is ignored once CouchDB restarts. If the system goes down in Step 2, there is the probability that one of the two identical database headers is already written; if this is not the case, the inconsistency between database headers and database contents is discovered as CouchDB checks the database headers for consistency once it starts up. Besides these checks of database headers, there are not any checks or purges required:

- Read requests are never blocked, never have to wait for a reader or writer, and are never interrupted by CouchDB. It is guaranteed that a reading client checks a uniform snapshot of the database from the beginning to the end of a read operation.
- As previously mentioned, CouchDB databases are indexed in B-Trees. On update operations on documents, new revision numbers (sequence-ids) for the updated documents are generated and indexed, and the updated index is written in an append-only way to disk.
- To store documents with efficiency, a document and its metadata are combined in a therefore referred to as a buffer first, then written to disk sequentially. Hence, documents may be read by clients and the database (for example, indexing purposes and view-calculation) efficiently in one go.

## MongoDB

MongoDB is also an open-source project which is also called schema-free database. MongoDB also uses JSON-like documents. (It does support schemas; it is just that they are optional.) It is written in C++ and owned by 10gen Inc., which later changed its name to MongoDB Inc. in 2013; MongoDB, Inc. is the organization behind the development and maintenance of MongoDB, the database. They provide commercial products and services related to MongoDB, including support, consulting, and enterprise versions of the database. Its main motive is to fill the gap between traditional technology and the latest one. It also provides solutions for issues of traditional RDBMS with scalable key/value database. [SourceForge.net](#), foursquare, the New York Times, the URL-shortener bit.ly, and the distributed social network DIASPORA are the main users of MongoDB.

Following are the features of MongoDB:

- **Database and collection:** A database contains one or more collections consisting of documents. MongoDB databases reside on a MongoDB server that may host over one of such databases that are independent and kept separately by the MongoDB server.

So, regulating access to the database, a group of security credentials is also defined for the databases.

Collections within databases are mentioned by the MongoDB manual as *named groupings of documents*. As MongoDB is schema-free, the documents within a set may be heterogeneous, although the MongoDB manual suggests making *one database collection for every of your high-level objects*. Once the primary document is inserted into a database, a collection is created automatically, and the inserted document is another to this collection. Such an implicitly created collection gets organized with default parameters by MongoDB if individual values for choices like auto-indexing, reallocated disc space, or size-limits are demanded; collections may also be produced explicitly by the **createCollection** command:

```
db . createCollection ( < name > , { < configuration  
parameters >} )
```

If need to create a collection with the name **mycoll** and 10,000,000 bytes of preallocated disk space and *no automatically generated* and indexed document-field `_id`:

```
db . createCollection (" mycoll " , { size :10000000 ,  
autoIndexId : false } ) ;
```

MongoDB uses the convention of dot-notation wot show any hierarchical namespace, for example, the collections **wiki.articles**, **wiki.categories**, and **wiki.authors** residing under the namespace wiki. The MongoDB manual notes that this is simply an organizational mechanism for the user, and the collection namespace is flat from the database's perspective.

The following is the sample database to represent MongoDB representation:

```
{  
  title : " MongoDB " ,  
  last_editor : "192.51.223.42" ,  
  last_modified : new Date ("25/10/1985") ,  
  body : " MongoDB is a ..." ,  
  categories : [" Database " , " NoSQL " , " Document  
Database "] ,  
  reviewed : false  
}
```

To insert such a document in a collection of MongoDB following command can be used:

```
db . < collection >. insert ( { title : " MongoDB " ,  
  last_editor : ... } ) ;
```

After insertion it also can be retrieved by:

```
db . < collection >. find ( { categories : [ "  
NoSQL " , " Document Databases " ] } ) ;  
db . < collection >. save ( { ... } ) ;
```

The document of MongoDB is the size of 16 MB.

Following are the data types that can be used in this kind of database:

- **Scalar types:** boolean, integer, double character sequence types: string (for character sequences encoded in UTF- 8), regular expression, and

code (JavaScript)

- Object (for BSON “Binary JSON” or “Binary JavaScript Object Notation” objects)
- Object id is a data type for 12-byte long binary values used by MongoDB. The object id datatype is composed of the following components:
  - timestamp in seconds since epoch (first 4 bytes)
  - id of the machine assigning the object id value (next 3 bytes)
  - id of the MongoDB process (next 2 bytes)
  - counter (last 3 bytes)
  - null
  - array
  - date and Timestamp
- **Operations:** MongoDB supports a wide range of operations for data management and manipulation. Some of the key operations in MongoDB include:
  - **CRUD operations:** MongoDB supports Create, Read, Update, and Delete operations for managing data. These operations allow you to insert new documents, query and retrieve data, update existing documents, and remove documents from a collection.
  - **Querying:** MongoDB provides a powerful query language that allows you to retrieve data based on various criteria, such as field values, comparison operators, logical operators, and regular expressions. Queries can be performed using the **find()** method with various modifiers and options.
  - **Indexing:** MongoDB supports the creation of indexes (using **getIndexes()**) on fields to improve query performance. Indexes can be created on single fields or multiple fields together. They help in speeding up data retrieval by allowing the database to quickly locate the relevant documents.
  - **Aggregation:** MongoDB offers an aggregation framework that allows you to perform advanced data analysis and transformation operations. It

provides a set of pipeline stages that can be used to group, filter, project, sort, and perform calculations on data.

- **Data manipulation:** MongoDB provides operators and methods to perform various data manipulation tasks, such as updating specific fields within a document, performing atomic updates, incrementing or decrementing field values, and performing array operations like adding elements, removing elements, and updating elements within an array.
- **Text search:** MongoDB includes a powerful text search feature that enables you to perform full-text search queries on text fields. It supports language-specific stemming, stop words, and text index optimization for efficient searching.
- **Transactions:** MongoDB supports multi-document transactions, allowing you to perform multiple operations as a single atomic unit of work. Transactions ensure data consistency and integrity in complex operations involving multiple documents.
- **GridFS:** MongoDB provides a built-in file system specification called GridFS for storing and retrieving large files. Actual files are stored within MongoDB. It allows you to split files into smaller chunks and store them as documents in MongoDB, enabling efficient storage and retrieval of large files. GridFS does not function like a traditional file system on your operating system (it does not organize files in directories, for example), it serves as a protocol and a set of conventions for storing and retrieving large files in a database.

Following are few examples of the operations available in MongoDB. It offers a comprehensive set of features and capabilities for data storage, retrieval, manipulation, and analysis:

- To establish foreign keys and to establish connections between documents, MongoDB provides references that connect different documents. It does not connect these automatically. It can be set manually by `_id` field.

```
{ $ref : < collectionname > , $id : < documentid >[  
    , $db : < dbname >] }
```

- The selection query of MongoDB refers as the query object. Find the parameter used to collect queries.

```
db . < collection >. find ( { title : " MongoDB " }  
;
```

Other than this lot of operators are also allowed.

```
< fieldname >: {$ < operator >: < value >}
```

```
< fieldname >: {$ < operator >: < value > , $ <  
operator >: value } // AND - junction
```

The preceding table use operator with value to allow for selection. For selecting modulo following table is required.

```
{ age : { $mod : [2 , 1]} } // to retrieve  
documents with an uneven age
```

To select and compare one element from an array.

```
{ categories : { $in : [" NoSQL " , " Document  
Databases "] } }
```

Size of array can be retrieved by.

```
{ categories : { $size : 2} }
```

When there is a need to select one among all, there is the case of either-or. For that, there is a need to define.

```
{ $or : [ { reviewed : { $exists : true } } , {  
categories : { $size : 2} } ] }
```

For neglecting case, there is a need to use not operation.

```
{ $not : { categories : { $in : {" NoSQL "}} } } //  
category does not contain " NoSQL "
```

```
{ $not : { title : /^ Mongo / i } } // title does  
not start with " Mongo "
```

- **Projection:** To provide a limit for selecting data from fields, there is the use of projection. The field specification is also relevant to fields of embedded objects using a dot notation (**field.subfield**) and to ranges within arrays.

```
db . < collection >. find ( { < selection criteria >} , { < field_1 >:1 , ...} ) ;
```

If only certain fields are excepted from the documents returned by the find operation, they are assigned to the value 0:

```
db . < collection >. find ( { < selection criteria >} , { < field_1 >:0 , < field_2 >:0 , ...} ) ;
```

- **Sort:** After selecting a field, there is a need to further process that can be its sorting.

```
db . < collection >. find ( ... ) . sort ( { < field >: <1| -1 >} ) . limit ( < number >) . skip ( < number >) ;
```

It is equivalent to **ORDERBY()** clause of SQL. By this, ascending and descending data can be sorted. To retrieve result fast and efficient, there is use of count() operation.

```
db . < collection >. find ( ... ) . count () ;
```

- **Cursor:** Return value of find can be used for further process using the cursor.

```
var cursor = db . < collection >. find ( {...} ) ;  
cursor . forEach ( function ( result ) { ... } ) ;
```

To request a single document—typically identified by its default primary key field `_id`—the **findOne** operation should be used instead of finding.

```
db . < collection >. findOne ( { _id : 921394} ) ;
```

- **Insert:** To insert some document into collection of MongoDB, there is use of insert operation.

```
db . < collection >. insert ( < document > ) ;
```

Using saved document can also be inserted into collection.

```
db . < collection >. save ( < document > ) ;
```

- **Update:** Save operation can be used to update the document. There is an explicit method also that use to update documents.

```
db . < collection >. update ( < criteria > , < new document > , < upsert > , < multi > ) ;
```

The selection criteria will be selected by the first argument, and if needed to update the field, it has to be provided in the same syntax as for the find operation. The second argument is used to select a matching document for the replacement of existing objects in the collection. In the case of the true value of the third argument second argument is inserted even if there is no document inserted in the collection in first argument (**upsert** is short for **update** or **insert**). In case of true of last argument, all document that match, the criteria are replaced, otherwise, only the first matching document is updated. The last two arguments are set to false by default for updation.

- **Atomic updates:** In MongoDB, update operation is non-blocking by default. Atomicity is used to update multiple documents using a single command. It needs desired results that keep up to date without affecting consistency. For that \$atomic flag set to be true, it needs to be added in update criteria. With the use of update modifier, all operations are atomic, but there is also the use of *update if current* and *compare and swap* command to use it for collection update.

```
wikiArticle = db . wiki . findOne ( { title : " MongoDB " } ) ; // select document  
oldRevision = wikiArticle . revision ; // save old revision number  
wikiArticle . revision ++; // increment revision number in document  
db . wiki . update ( { title : " MongoDB " ,  
revision = oldRevision } , wikiArticle ) ; // Update if Current
```

Since the last statement uses revision numbers that need to be the same for the updation of any article in MongoDB. In the case of changing the first tree lines, the selection criteria in the first argument of the fourth line will no longer match any document.

In case of a single document that needs to be updated with the command **findAndModify**, which returns the document after updation with insurance

of atomicity.

```
db . < collection >. findAndModify ( query : {...}
,
sort : {...} ,
remove : < true | false > ,
update : {...} ,
new : < true | false > ,
fields : {...} ,
upsert : < true | false >) ;
```

In the case of a sharded document, **findAndModify** works well and provides the desired result. The preceding commands will also sort documents with conditions written over there with a set of order. In case of removal, set to true return document shall be removed from the collection. The update command is used to update and modify the document by placing whole or partial with a modifier expression. If there is a need to return a new document, then there is a need to set a new field as true. In the case of restriction of fields, there is a need to define its certainty in fields argument by 1 or 0 for selecting and deselecting them. The last argument, **upsert** executes if a document shall be created if the result set of the query is empty.

- **Delete:** With some certain criteria, some documents can also be removed from the collection list.

```
db . < collection >. remove ( { < criteria > } ) ;
```

This is similar to the **find** statement in the query, so there is a similar manner to define criteria to eliminate unwanted documents without deleting the criteria argument. With the use of id (**\_\_id**) single document can be deleted from the collection. Concurrent deletion is also allowed in MongoDB with **remove** operation. If there is no requirement to remove concurrent document, then there is a need to maintain the atomicity of collection with the following syntax:

```
db . < collection >. remove ( { < criteria > ,
$atomic : true } ) ;
```

- **Server-side execution:** MongoDB also provides server-side execution with the help of the following keywords. It is similar to SQL database:

**eval operation - Execution arbitrary code on a single database node**

```
db . eval ( function ( < formal parameters >) { ... } , < actual parameters >) ;
```

To execute arbitrary blocks of code locally on a database server, the code needs to be enclosed by an anonymous JavaScript operate and passed to MongoDB's generic eval operation. If the operate passed to eval has formal parameters, these have to be certain to actual parameters by passing them as arguments to eval. Though eval could also be useful to method large amounts of information locally on a server, it is to be used carefully; a write lock is a command throughout execution. A drawback is that the eval operation is not supported in sharded setups; in order for the MapReduce-approach has to be used in these scenarios, a function containing arbitrary code may also be saved under a key (with the fieldname **\_id**) on the database server in a special collection named system.js and later be invoked via their key, for example,

```
system . js . save ( { _id : " myfunction " , value : function ( ... ) { ... } } ) ;  
db . < collection >. eval ( myfunction , ... ) ; // invoke the formerly saved function
```

- **Aggregation:** via the operations count, group, and distinct.

```
db . < collection >. count ( < criteria > ) ;
```

Count, distinct, group, bucket, and bucketAuto operations are provided by MongoDB, a programming language that is executed on server-side. The preceding function provides a list of count operation that returns the number of documents. If selection criteria are used, the document fields used in it should be indexed to the execution of the count.

To retrieve **distinct** values, there is use of **distinct** keyword with **runCommand** operation.

```
db . runCommand ( { distinct : < collection > , key  
: < field > [ , query : < criteria > ]} ) ;  
  
db . < collection >. distinct ( < document field >  
[ , { < criteria > } ] ) ;
```

In MongoDB, the **Group** operation is similar to the **GROUP BY** clause in SQL and allows you to perform aggregation operations on a collection:

```
db . < collection >. group ( {  
key : { < document field to group by > } ,  
reduce : function ( doc , aggrcounter ) { <  
aggregation logic > } ,  
initial : { < initialization of aggregation  
variable ( s ) > } ,  
keyf : { < for grouping by key - objects or nested  
fields > }  
cond : { < selection criteria > } ,  
finalize : function ( value ) { < return value  
calculation > }  
} ) ;
```

The bucket operation groups documents into discrete *buckets* based on the specified expression or field. You can define the boundaries for each bucket using an array of values.

```
db.sales.aggregate([  
{  
  $bucket: {  
    groupBy: "$price",  
    boundaries: [0, 100, 500, 1000],  
    default: "Other",
```

```
        output: {
            count: { $sum: 1 },
            totalAmount: { $sum: "$amount" }
        }
    }
]);

```

The **bucketAuto** operation is similar to the bucket, but it automatically determines the bucket boundaries based on the distribution of values in the input documents.

You specify the number of desired buckets, and MongoDB evenly distributes the documents into these buckets. This is useful when you want to create histogram-like groupings without explicitly specifying boundaries.

```
db.sales.aggregate([
{
    $bucketAuto: {
        groupBy: "$price",
        buckets: 5,
        output: {
            count: { $sum: 1 },
            totalAmount: { $sum: "$amount" }
        }
    }
}
]);

```

Following are the key components of aggregate function:

- **Key:** It is mandatory with the condition. It is used on the base of a grouping shall happen.
- **Reduce:** It is mandatory, and this function *aggregates* (reduces) the objects iterated. Operations of a reduce function include sum and count with two arguments: the current document being iterated over and the aggregation counter object.
- **Initial:** It is not mandatory, but it is used to initialize the aggregate function.
- **Keyf:** It is mandatory in case of key is not specified. A key object has to be specified when the function is returning the key object; in this case, the key will not be used, and it will be replaced by **keyf**.
- **Cond:** This specifies the condition for selecting the document with **cond** field. This is used to provide certain criteria for selecting fields. It can also be used with like and count operation. If no criteria are specified, then the document will be processed by group. It is not a mandatory field.
- **Finalize:** An optional function to be run on each item in the result set just before the item is returned. It is not mandatory.

The following is the code from the MongoDB manual for providing group information:

```
db . < collection >. group (  
  { key : { a : true , b : true } ,  
    cond : { active :1 } ,  
    reduce : function ( doc , acc ) { acc . csum += obj  
      . c ; } ,  
    initial : { csum : 0 }  
  }  
);
```

Group operation cannot be used in sharded objects. For that case, mapreduce comes into existence.

- **MapReduce** code executed on many database nodes that stored sharded data.

This approach is based on a research paper of Google that is, **Google File System (GFS)**. It concerns with two modules that is map and reduce. It is a programming module that is basically used to filter data from a large data set to a small data set, and after that, it provides basic operation on that since reduce is usually an aggregation operation. That is the reason it is helpful in sharded data. The output of the map phase will be the input of the reduce phase. It also needs a distributed system to perform a mapreduce model. This technique is used in the Hadoop framework for processing large amount of data. MongoDB uses the following code for generating mapreduce code:

```
db . < collection >. mapreduce ( map : <map -  
function > ,  
reduce : < reduce - function > ,  
query : < selection criteria > ,  
sort : < sorting specification > ,  
limit : < number of objects to process > ,  
out : < output - collection name > ,  
outType : <" normal "|" merge "|" reduce "> ,  
keeptemp : < true | false > ,  
finalize : < finalize function > ,  
scope : < object with variables to put in global  
namespace > ,  
bypassDocumentValidation: < true | false >,  
);
```

Key concepts of Map Reduce in java scripts are as follows:

- Map and reduce phases are mandatory to write that assigned to JavaScript function. To avoid temporary storage in collection, there is a need to

specify out, or keeptemp is set to true, then the outcome of the MapReduce data processing is saved in a permanent collection.

- The **outType** parameter specifies about how the collection is populated. If it is set to *normal*, the new results are written into the collection; if set to **merge**, old and new results value are merged, and the latest MapReduce job overwrites previous jobs with key that relate it to the old and new results, if set to **reduce** the reduce operation of the MapReduce job is executed for keys that have old and new values.
- The **finalize** function is for all results that execute mapreduce job. In comparison to reduce, finalize is invoked once for a given key and value, whereas reduce is invoked iteratively and called many times for a particular key. Finalize is always called after the reduce function.
- The scope argument is defined in a global namespace and assigned with a JavaScript object.
- If **bypassDocumentValidation** is set to true at the time of MapReduce job execution.

The functions map, reduce, and finalize are specified as follows:

Map function requires a key and a whole document list that needs to be processed. With an output of that map function, reduce function comes into the picture and starts functioning.

```
function map ( void ) --> void
```

```
function reduce ( key , value_array ) --> value
```

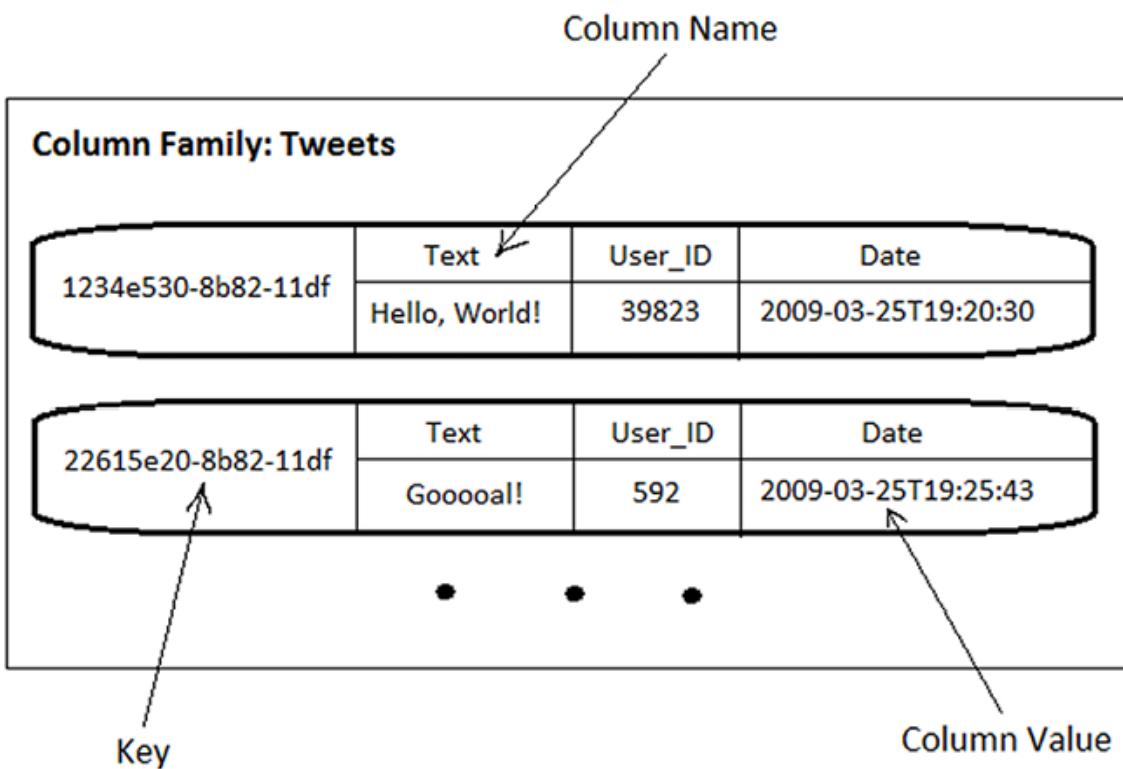
Optional finalize use after reduce phase as follows:

```
function finalize ( key , value ) --> final_value
```

So, mapreduce the usefulness of sharded objects in MongoDB that are distributed across the nodes. Each node consists of a different document list that needs to be processed. For that, mapreduce is the only efficient way to deal with it.

## Column family database

Column-family databases might be the most difficult to understand. One way to consider these databases is that they are very massive tables with zillions of rows and zillions of possible columns; however, every row truly features a relatively little range of columns compared with the full number potential. Mathematicians recognize this arrangement as a distributed matrix; programmers could acknowledge it as a hash table or dictionary mapping a key to a collection of key-value pairs, as shown in *Figure 2.3*. An example of such information is one in which the key is a URL and each column represents a revision of the document. One column-family contains metadata about that page; another column-family contains data regarding when the page was changed, what was modified, the extent of the modification, and so on.



**Figure 2.3:** Column family database

Google Bigtable with its derivatives and Cassandra will be discussed in this category. Google Bigtable: Bigtable provide a system to manage a large amount of data in a distributed manner to manage with structured data that also scales to large size easily. Petabytes of information can be stored using these techniques on commodity servers. As of 2006, over sixty projects are working with this technique, including Web indexing, Google Earth, Google Analytics, Orkut, and Google Docs. The Bigtable clusters used by these products span a large range of

configurations, from some to thousands of servers, and store up to many hundred terabytes of data. Google shows that Bigtable has achieved several goals: wide relevancy, scalability, high performance, and high availability.

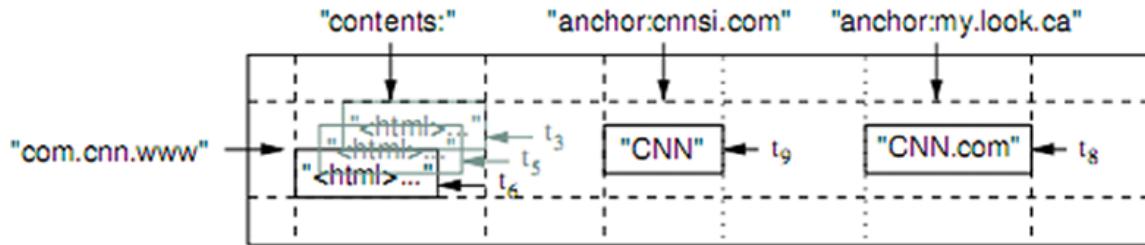
It provides flexibility with high performance and high availability that is easily provided with the Bigtable technique, which supports massive data with a large capacity to enhance it. For Google as an organization, the design and implementation of Bigtable has shown to be advantageous because it has *gotten a considerable quantity of flexibility from designing our own data model for Bigtable*.

Bigtable is represented as a database by *Google as it shared several implementation methods with databases, for example, parallel and main-memory databases. However, it distinguishes itself from relative databases because it does not support a full relational data model but a simpler one which will be dynamically controlled by clients. Bigtable, moreover, permits clients to reason about the locality properties of the data that are reflected in the underlying storage.* A further identifying proposition is that Bigtable permits data to be delivered out of memory or from disk, which might be given via configuration.

For any distributed data base key–value pair play the most important role to play as it was only weapon for sharded data too. But that should not be the only block to the provider; it is also the reason for big table to come into existence. It is richer than key–value pair, and it supports to semi-structured sparse data. Google’s Bigtable is known as *a sparse, distributed, persistent multidimensional sorted map* data structure for processing distributed data. In this particular database, values are stored as arrays of bytes, which are not interpreted or processed by the data store itself. The addressing mechanism used in this database involves a triplet consisting of the row key, column key, and timestamps.

Each value in the database is associated with a unique combination of row key, column key, and timestamp. This triple acts as an address or identifier for accessing and retrieving specific data within the database. By using this addressing scheme, you can retrieve the desired value by specifying the appropriate row key, column key, and timestamp combination.

It is worth noting that the database does not interpret or manipulate the byte arrays directly. The interpretation or processing of the data within the byte arrays is typically handled by the application or client accessing the database. The database itself treats the data as opaque bytes and focuses on efficiently storing and retrieving them based on the provided addressing scheme:



**Figure 2.4:** Google Bigtable structure

Since Google also works on Web crawler to check customer need and requirement all the time, [Figure 2.4](#) represent a simple example of that. In this example, Map contains a variable number of rows and columns that are used in reading the crawler. The value that will be stored in processing will be (domain-name, column-name, and timestamps).

Column content: The page details, whereas anchor :<**domain-name**> store link for referring those contents. Each value over the data consists of timestamps such as t3, t5, t8, and so on that represent page contents. For example, t3, t5, and t6 represent (as shown in the previous figure) the pages of **com.cnn.www**, where t9 represent CNN context while t8 link text from **Mylook**.

Keys, which are atomic, in a row in Bigtable, that of a string of 64KB size. Rows are in lexicographic order and partitioned into tablets, which are dynamic means they can be updated easily. [Figure 2.4](#) represents the row key as a domain name and hierarchically descending storage (for example, **com.cnn.blogs**, **com.cnn.www** in contrast to **blogs.cnn.com**, **www.cnn.com**).

There is often no strict limitation on column names, and the number of columns can be dynamic. Columns are typically organized into column families, which are groups of columns that share a key prefix and often have special purposes and properties associated with them. Columns are represented as column-family, which are grouped together by key prefixes with special purposes and properties.

The preceding example has two column-family that is content and anchor. Again, content has its family that consists of one column, whereas anchor has a family of two columns that refer to a domain name for the site. Timestamps are the 64-bit integers that are used in Bigtable to discriminate with different reversions of cell value or chosen by client applications that must be unique. Bigtable arranges timestamp values in decreasing order so that the most recently used data of the site can be read first.

Bigtable can perform the following operations in APIs:

---

<b>Operation</b>	<b>Description</b>
Read operations	Include the lookup and selection of rows by their key, the limitation of column families, as well as timestamps and iterators for columns.
Write operations for rows	It includes the creation, update, and deletion of values for a column of a particular row.
Write operations for tables and column families	It covers their creation and deletion.
Administrative operations	It allows to change of cluster, table, and column family metadata, such as access control rights.
Server-side code execution	It provides various forms of data transformation, filtering based on arbitrary expressions, and summarization via a variety of operators.
MapReduce operations	It uses the contents of Bigtable maps as their input source as well as output target.

**Table 2.3:** Various operations in table

## Components

Bigtable consists of three main components, namely, tablet server, client library, and master server. Each tablet contains a range of rows from the table, and these tablets are dynamic in nature, meaning they can be automatically split or merged to accommodate changes in data volume or distribution.

Tablet servers, which are used to manage tablets' requests to read and write for tablets, are handled by multiple tablet servers. These are also responsible for splitting of tablets according to need and condition. The sizes of tablet servers are not fixed, and they can be scaled as needed.

**Client library** provided database that is responsible to interact with bigtable instances. It provides the library for tablet that are responsible for performing read and write requests to client applications.

**Master servers** have multiple responsibilities, like managing the tablets and tablet servers. It allows the tablets to access their corresponding servers, add it, detect it,

and remove it, if not in use at a particular server. It also has the responsibility to distribute workload among all servers equally.

It also takes care of changes in a schema of bigtable in the creation of tables and column-family. It also allows itself for detecting instances of garbage-collector. The load on client-server libraries tends to be low because clients often do not require immediate responses or do not rely heavily on the server's processing capabilities.

**Tablet lifecycle**, in the context of Bigtable, the tablet assignment process involves the creation, deletion, and assignment of tablets to tablet servers by the master server. Each tablet in a Bigtable is assigned to, at most, one tablet server at a given time.

When a tablet is formed, typically due to a table split or the creation of a new table, the master server is responsible for determining which tablet server will be assigned to handle that specific tablet. The tablet server is responsible for serving read and write requests for the assigned tablet. If a tablet is no longer needed or requires reassignment, such as during table merging or resizing operations, the master server can delete the tablet, making it available for reassignment to another tablet server if necessary.

The tablet assignment process ensures that each tablet in Bigtable is assigned to a single tablet server, allowing for efficient and distributed data storage and processing within the system.

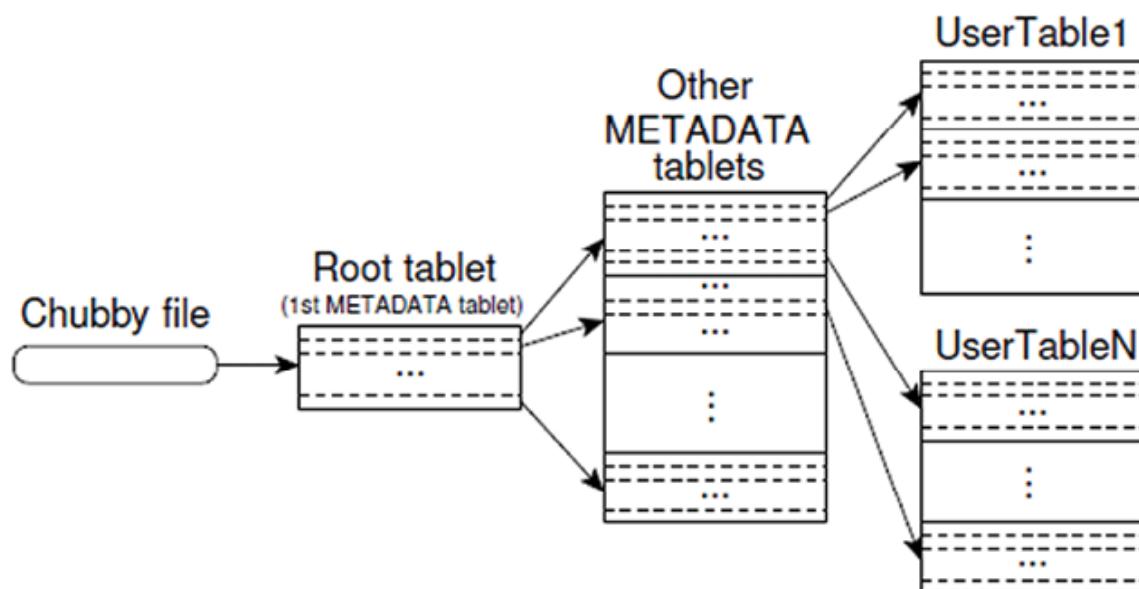
At the most, one is because tablets can also be unassigned till the master server finds a tablet server that has enough capability to serve that tablet. Tablets can also get united by the master server and split by a tablet server that has to inform the master server regarding the split.

**Tablet server lifecycle:** When a tablet server starts, it creates a uniquely named enter a predefined directory of a **chubby** namespace (Chubby is a distributed lock service developed by Google) as in [Figure 2.5](#) and acquires an exclusive lock for this. The master server of a Bigtable instance constantly monitors the tablet servers by asking them whether or not they have still secured their file in Chubby; if a tablet server does not respond, the master server checks the Chubby directory to see whether or not the particular tablet server still holds its lock. If this is often not the case, the master server deletes the file in Chubby and puts the tablets served by this tablet server into the set of unassigned tablets. The tablet server itself stops serving any tablets once it loses its chubby lock. If the tablet server continues to be

up and running but is not able to hold its lock because of, for example, network partitioning, it will attempt to acquire that lock again if its file in Chubby has not been deleted. If this file is no longer present, the tablet server stops itself. If a tablet server is shut down in a controlled fashion by administrators, it tries to use its chubby lock service so that the master server will reassign tablets sooner.

**Master server lifecycle:** Once a master server starts up, it also places a special file into a chubby namespace and acquires an exclusive lock for it (to stop *concurrent master instantiations*). If the master server is not ready to hold that lock so that its chubby session expires, it takes itself down because it cannot monitor the tablet servers properly without a reliable association to chubby. Hence, the provision of a Bigtable instance depends on the reliability of the connection between a master server and also the chubby service used by a Bigtable instance. In addition to registering itself via a file and its lock-in chubby, the master server processes the subsequent steps when it starts up:

1. Discover tablet servers that are alive by scanning the chubby directory with tablet server files and checking their locks.
2. Connect to every alive tablet server and invite the tablets it serves at the moment.
3. Scan the metadata table to construct a list of all tables and tablets via subtraction of the tablets already served by the running tablet servers, and it derives a group of unassigned tablets, which is represented in [Figure 2.5](#):

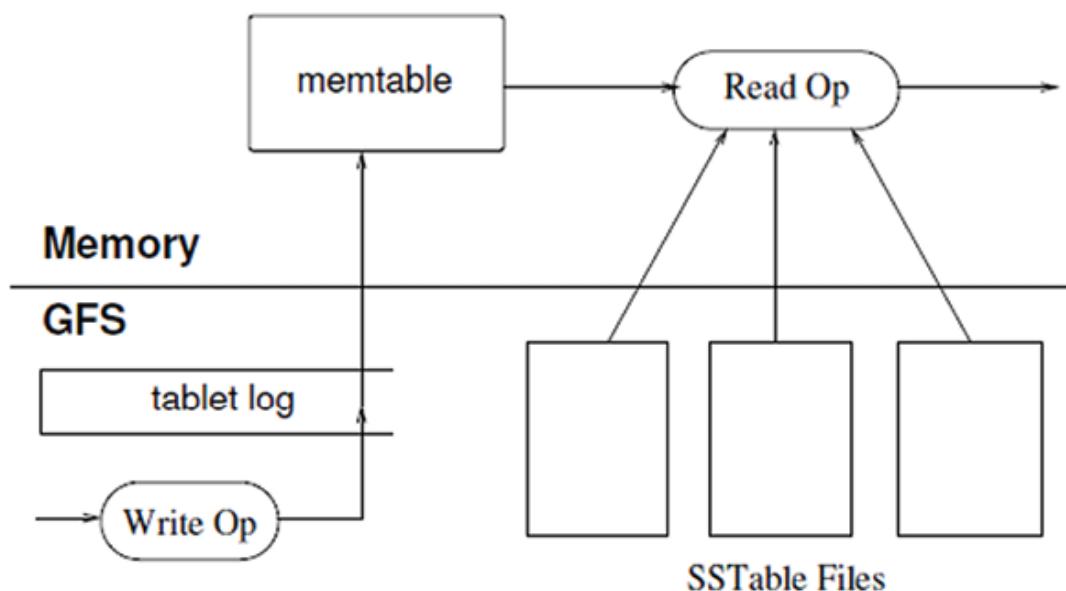


**Figure 2.5:** Chubby namespace file

### Table representation of Google Bigtable

Google Bigtable is a distributed, columnar, and NoSQL database system that is designed to handle large amounts of structured data. Although Bigtable does not use traditional tables like in relational databases, it can be conceptually represented as a table for easier understanding:

- All write operations are needed to commit with a commit log and persist in the **Google file system (GFS)**. These committed writes are continuously updated with a RAM buffer that is called **memtable**, as shown in [Figure 2.6](#).
- Memtable has a certain capacity; when it is filled, it gets freezes and transfers all updated records to **SSTable** with the formation of a new **memtable**, and in parallel, it is also written into GFS; this process is called minor compaction. Minor compaction typically refers to the process of compacting a few SSTables at a time, while major compaction involves compacting all SSTables in a column family. It helps in maintaining data structure and improving read performance. That means that older records are kept updated with **SSTable**, and newer data are located in memory.
- A metadata stores the information about **SSTable** with its location and data regarding this, along with commit logs that need to be updated with the tablet server. Every time a write operation needs to get authorization, when it forms a commit log and **memtable**.



**Figure 2.6:** Table structure

- Information on authorization is kept updated with chubby namespace.
- Read operation also checks the authorization of requests to issue them. **SSTable** and **memtable** are merged for read operation with lexicographic order.
- Occasionally, there is performing operation of freeze, transformation, and persistence with **SSTable**, and these acts perform asynchronously.
- Apart from minor compaction, there is major compaction as well when there is production of one **SSTable** out of the number of SSTables.

## BigTable derivatives

Hypertable, Hbase and Cassandra are the derivatives of Bigtable, which means these are the technologies of NoSQL that are connected to Bigtable.

- **Hypertable:** It is derived from Google's Bigtable for handling large amount of data. It was started by *Zvents Inc.*, and it became open-sourced under the GPL in 2007 and is sponsored by Baidu, the leading Chinese search engine since 2009. It is open source, and its development has been stopped.

Since hypertable is written mostly in C++, it depends on the distributed system that can support HDFS. So, it can also support to process large amount of data. Indifferent from HBase, it is also available for column-oriented architecture. It supports all features that are in Bigtable and column families availability in hypertable. In this, all tables are partitioned with a row key, that also done in Bigtable with a replicated structure. Hypertable does processing with its own language, that is, Hypertable Query Language, and exposes structure of C++.

- **Hbase:** Hbase was developed in Java and is a totally column-family-oriented database. It represents fault-tolerant sparse data. It took the concept of memory and a disk-oriented approach from bigtable that are compaction of immutable or append-only files. It is very much compatible with Hadoop to process large amount of data that is stored in HDFS. It can also synchronize with mapreduce jobs. Facebook has been using Hbase in its messaging system since 2010.
- **Cassandra:** Companies such as Twitter, Rackspace, and Digg initiated and used this database. It is based on the idea of bigtable and dynamo that has been discussed previously. It also supports a distributed approach for

scalability issue for database. It can also manage large amount of structured data but without supporting a full relational database.

Facebook used this technique before it moved to HBase for its messaging system, for storage, read, and forward. It provides facilities to store the message part and forward it when needed, and this problem is called the Inbox search problem. With Cassandra, Facebook got scalability availability for its messaging system. Cassandra uses the following parts in its database:

Rows	These are identified by a string key of arbitrary length. There are atomic operations required.
Column-family	Need to define in advance. These are not limited. Columns can be added dynamically.
Column	It is the name and values that are stored in the table that are identified by timestamps.
Supercolumns	It has a name and an arbitrary number of columns associated with it. Again, the number of columns per super-column may differ per row.

**Table 2.4:** *Cassandra's attributes*

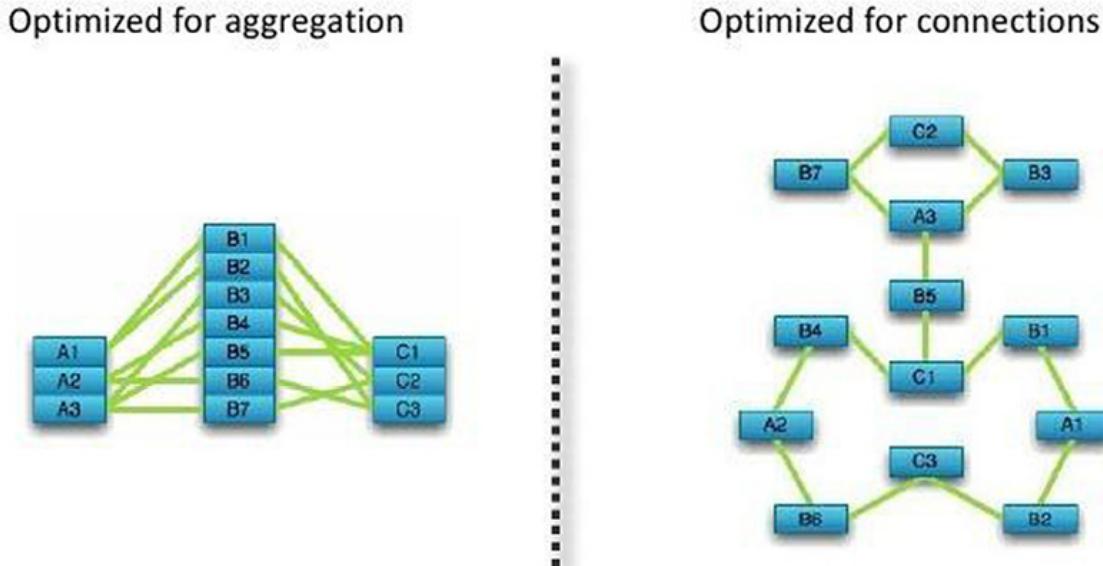
Cassandra stored values in triplet that is (row-key, column-key, timestamp) with column-key as **column-family:column** are as follows:

```
get(table, key, columnName)
insert(table, key, rowMutation)
delete(table, key, columnName)
```

Any number of column-family can be added; these are the APIs that are used in Cassandra as database.

## Graph database

Graph databases are totally different from the previous three types of databases in this relationship. Nodes within the graph represent a particular kind of entity (person or thing), and the edges between nodes are labeled as particular kinds of relationships. Edges point in one direction, and an edge itself could have attributes, as shown in *Figure 2.7*:



**Figure 2.7: Graph database**

A graph database is a specialized type of database management system designed for efficiently storing, managing, and querying data that is inherently interconnected and represented as a graph. In a graph database, data is organized into nodes, which represent entities or objects, and edges, which represent relationships or connections between nodes. This graph-based data model is particularly well-suited for scenarios where relationships between data points are complex and critical for the understanding of the data. Graph databases excel in applications such as social networks, recommendation systems, fraud detection, knowledge graphs, and network and infrastructure management.

One of the key advantages of graph databases is their ability to perform highly efficient and expressive queries for traversing relationships and patterns within the data. This makes it possible to answer complex questions and uncover hidden insights that would be challenging or slow to retrieve using traditional relational databases. Examples of popular graph databases include Neo4j, Amazon Neptune, and Microsoft Azure Cosmos DB. These databases provide powerful tools and query languages specifically designed for working with graph data, making them valuable assets for organizations dealing with interconnected and relationship-rich data structures.

## Neo4j

Neo4j is a graph database management system, whereas GraphQL is a query language and runtime for APIs. They serve different but complementary roles in the realm of data management and API development:

- **Database type:** Neo4j is a graph database, which means it is designed to store and manage data in the form of nodes (representing entities) and relationships (representing connections between entities). It is highly efficient for queries that involve complex relationships and traversal of connected data.
- **Use cases:** Neo4j is often used for applications that require modeling and querying highly connected or graph-like data, such as social networks, recommendation engines, fraud detection, and knowledge graphs.
- **Query language:** Neo4j uses the Cypher query language, specifically designed for querying graph data. Cypher allows you to express complex graph traversal and pattern-matching queries concisely.
- **Benefits:** Neo4j's strength lies in its ability to efficiently represent and query connected data, making it a powerful tool for applications that rely on relationships between entities.

## GraphQL

GraphQL is a query language for APIs. It allows clients to precisely request the data they need, and nothing more, by defining the structure of the response in the query itself. This helps in reducing over-fetching and under-fetching of data.

The following are the characteristics of GraphQL:

- **Server-side runtime:** GraphQL requires a server-side runtime to interpret queries and respond with the requested data. Popular server frameworks like Apollo Server and Express can be used to build GraphQL APIs.
- **Use cases:** GraphQL is used to create APIs for a wide range of applications, from Web and mobile apps to microservices and IoT devices. It excels in scenarios where clients have varying data requirements.
- **Flexibility:** GraphQL provides flexibility to clients by allowing them to shape their queries according to their specific needs. Clients can request data from multiple sources in a single query.
- **Benefits:** GraphQL simplifies the process of building and consuming APIs, as clients can request the exact data they need, reducing the need for multiple API endpoints and versioning.

In practice, Neo4j and GraphQL can be used together in scenarios where you want to expose graph data stored in Neo4j through a GraphQL API. This allows clients

to query and manipulate graph data using GraphQL's flexibility and expressiveness while leveraging Neo4j's efficiency in handling graph-related operations.

## SQL versus NoSQL

NoSQL databases and relational databases supply totally different capabilities and guarantees. For handling structured data that is extremely transactional, an RDBMS is perhaps the way to go. Otherwise, you can opt for splitting management, with an RDBMS managing the structured information and a NoSQL solution managing the unstructured data. The latter sort of system, known as polyglot persistence, will be a key solution till NoSQL databases support multiple data structures. Style data management design and systems to play as the strengths of each sort of solution.

The following table shows the differences between NoSQL and relational database:

NoSQL	Relational database
1. Column-oriented	1. Row oriented
2. Flexible schema: add column later	2. Fixed schema
3. Good with sparse table	3. Not optimized for sparse matrix
4. Join using MR—not optimized	4. Optimized for join operations
5. Tight integration with key–value system	5. Not integrated
6. Horizontal scalability	6. Hard to shard and scale
7. Good for semi-structured, unstructured, and structured data	7. Only for structured data

**Table 2.5: NoSQL vs Relational database**

NoSQL provided an alternate method for industry purposes in which there is no requirement to have a schema of data. If it does not have, then also NoSQL will do it easily. Following is the factual information that was useful for industry purposes:

- NoSQL documents do not require each document to have the same attributes.
- NoSQL is explicitly aware of how data will be used in a particular application.
- At the physical database level, NoSQL is aware that it runs in a cluster and takes that fact into account in its data management strategy.
- NoSQL applications can specify a replication factor to ensure a particular level of consistency.

## **Denormalization**

Normalizing, within the RDBMS world, is the act of performing a series of transformations on a database design till the design meets bound tests. Each of those tests describes a particular normal form. There are many forms of normal forms: 1NF, 2NF, 3NF, 4NF, and BCNF (Boyce-Codd traditional form). Depending on data, the first three forms could also be enough to retrieve from the database.

While the process of gathering data into a single logical table is called denormalizing, it improves read query performance. NoSQL databases that are designed to support queries from the beginning usually denormalize the information based on the anticipated queries to be created against that info. If designing a database by thinking about the read queries instead of thinking about the information, then it may decide that as a result of most of the applying queries concern objects and embed the information within the object.

## **Data distribution**

Data distribution can create issues of consistency. There is a need to follow the properties of ACID so that all users can trust the NoSQL database for use. Most NoSQL databases relax ACID constraints. A NoSQL database could guarantee that an update of an individual document is atomic; that is, it happens or it does not. No such guarantees are created, however, if the logical transaction requires updates to many tables within the same database, not even if the database is running on the computer that holds the information.

The RDBMS was not ACID-compliant when it had been originally created. It added ACID support over time. NoSQL databases are following this trend. MarkLogic and FoundationDB offer ACID consistency in data distribution.

Data can be distributed in the following two ways:

- **Sharding:** It is also referred to as *horizontal partitioning*, which involves partitioning the database on the value of some field. According to the value of the document, it is stored in various machines that machines are not very specific with respect to hardware, and these are commodity hardware. These distributions of data are purposely used for processing and storing purposes. Few NoSQL databases require an admin role to configure the data, but most of databases also give responsibility for the distribution of data automatically, which is called **rebalancing**.

The drawback of manually configuring rules is that they unknowingly have an associated degree imbalance of data between nodes. This might end in a slower response time from some nodes. Associate degree RDBMS supports partitioning, but sometimes, the partitions stay on one machine (in the form of explicit disk partitions).

The following are the various techniques to apply sharding:

- **Use a key value** to shard your data: In this method, users may use different locations to store data with help of key value pair. All data can be easily accessed by key of that data. It makes it easy to store data irrespective of its location storage.
- **Use load balancing** to shard your data: Database can take individual decisions for storing data in different locations. Large sharding can also be split into short sharding that reframe decisions by the database itself.
- **Hash the key:** In this development, the key can be arranged by hashing its value. All assignments can be hashed to store all document. Consistent hashing assigns documents with a particular key value to one of the servers in a hash ring (a collection of servers, each of which is responsible for a particular range of hash values). Again, the details are hidden from the application programmer.
- **Replication:** Replication mode can be performed in either master/slave way or peer-to-peer way. However, both methods do not give a guarantee for consistency. Even the graph database is still vertical enhancement and performs action on a single system. All methods also take consistency issues very seriously because sooner or later, the problem or replication can arrive. Eventually, consistency means the state of the database is not consistent until the change is committed to the replica node.

Let us delve into the intricacies of two fundamental replication architectures:

- **Master/Slave replication:** It is very easy to make replication in master/slave architecture. All data can be stored in slave node, and at the time of enhancement, slave nodes can be added. An issue occurs when there is a failure of the master node, in this case, the slave needs to elect a new master to continue its functioning for storing data in a horizontal manner. In the best solution master node contains all write property, and the slave contains all read property. With that said, master/slave balancing is not needed if the NoSQL server stuffs a replica to the new master and has ACID manageable properties, and then no inconsistencies are achievable.
- **Peer-to-peer replication:** Peer-to-peer replication implies that no master node exists. Any node will settle for reads or writes. Writes are propagated to each peer, and reads will occur from any of the servers. Replication is the workhorse of the many NoSQL databases. However, the main downside in replicating the information on multiple nodes is that these replicas will become inconsistent. Consistency becomes a problem once two users update a similar document at the same time on totally different peers. A read inconsistency will occur if an update to a document has not propagated from one peer to another: One person sees one value returned, and another person sees a distinct value.

The count may be completely different for various individuals. With low-value data, that may not be a problem. However, an inconsistent view of inventory may cause a problem. For example, if one person sees one count and the other is actually lower, you may be surprised once you run out of inventory.

The RDBMS resolved this downside an extended time ago by using journals and log shipping among clusters. Some NoSQL solutions are starting to provide this feature; however, you may need to consider handling the matter in application development.

## **Data durability**

Durability represents whether or not a changed document is held on permanently once an update is finished. During a NoSQL information, durability is usually enforced via replication. Eventually, all nodes within the cluster want to have identical copies of the data, as a result, a read operation could be served by any of the replicas and wishes to get the same data from all of them.

The number of nodes needed to make sure the same read is called the replication issue. Some nodes during a cluster are participating in replication. Of these nodes, you need some variety of nodes to reply to an update (a write request) and a few numbers to reply to a question (a read request). The replication factor is the variety of nodes that are required to participate to guarantee consistency. In master/slave replication, there is a need to own only one node; the master answers an update because relying on that, the changes will be sent to the slave nodes.

If the setup is of multiple *readers* (**R**) and *writers* (**W**) participating in replication, then there is a need to own over half the total number of *nodes* (**N**) to reply to the update to ensure that the data has been saved. An information designer will play with these numbers to achieve completely different effects, such as reading consistent, always consistent, or eventually consistent.

## Consistency issues in NoSQL

Consistency in NoSQL databases refers to the level of data consistency or coherence that can be expected when multiple clients or nodes access and modify the database concurrently. Unlike traditional relational databases, which often prioritize strong consistency with immediate and guaranteed data synchronization, NoSQL databases often offer various consistency models to cater to different application requirements. These models trade off some level of consistency for increased availability, fault tolerance, and performance.

One of the well-known consistency models in NoSQL is eventual consistency, which allows for temporary data inconsistencies between nodes. In an eventually consistent system, if no further updates are made to a particular piece of data, all replicas or nodes will eventually converge to the same value. This flexibility is valuable in distributed systems where network partitions or node failures can occur, as it ensures that the system remains operational even under adverse conditions. However, it requires careful design and management to handle potential data conflicts and reconciliation.

The choice of consistency model in NoSQL databases depends on the specific needs of an application. Some scenarios, such as real-time financial transactions, may require strong consistency to avoid data discrepancies at all costs, whereas others, like content delivery networks, can benefit from eventual consistency to maintain availability and low-latency access. NoSQL databases provide the flexibility to select the appropriate consistency model to strike the right balance between data correctness and system performance.

## **ACID versus BASE**

Traditional lovers of DBMS are strict to **Atomicity, Consistency, Isolation, and Durability (ACID)** properties because these ways follow all desired results by keeping the database up-to-date.

Relational database allows to use traditional way to restrict the use of ACID property. So it stops making data corrupt and more use it to reliable for use. Wherever NoSQL does not guarantee to provide functionalities of ACID that need to follow strict rules of consistency. According to MarkLogic and Neo4j, some of the NoSQL databases provide the check and set verification to achieve consistency of documents. MongoDB is one of them, which implements the Check and Set method for performing the read and write method. It locks the data during of updation and provides consistency for read and write operations.

**Basic Availability, Soft State, and Eventual Consistency (BASE)** property states to maintain the balance of consistency and availability of database itself. It can be achieved by making cluster as primary manager of database and other nodes acting as replica for read-only. A two-phase commit is used to achieve these functionalities. To ensure about to get the update by all the client it is necessary to write the primary node that holds the data, and it needs to be locked until read replicas are up to date. Changes are made locally, but take permission from all nodes to get updated data from time to time.

In BASE, after the commit of the transaction, the read-only replica is updated only. In this approach, transactions commit faster and read and also get quicker access to data. Clients can find out-of-date data as replicas in the specified amount of time. For example, when there is any update in data on Facebook, so it is not critical if data will reflect after a few min. So, it is more important to get a commit of data and then create a replica of that. This approach will commit data faster and quicker. So, it provides logic for out-of-date data for unspecified time. But in the case of transaction in bank there is a need of immediate action. This may be the downside of that. Alternatively, there is also an approach of shared-nothing in which a node serves in cluster, and only one took part in database.

Shared-nothing does not state to lose the data to any node or cluster; it means to store data in any other secondary area that can be used further for processing, but only one node will remain the master of cluster for performing read and write at any time. This approach requires a two-phase commit for replication. This approach is faster than any other method as it does not get any request to read of database.

Shared-everything and shared-storage are two architectural approaches used in the design of computer systems and databases, primarily in the context of parallel and distributed computing. These approaches describe how system resources are shared and managed among multiple processors or nodes.

Let us explore each of these concepts:

- Shared-everything architecture:
  - In a shared-everything architecture, all processing nodes (often referred to as servers or processors) have equal access to all system resources, including memory, storage, and I/O devices.
  - This architecture is often associated with **symmetric multiprocessing (SMP)** systems, where multiple processors share a common memory pool and can execute tasks concurrently. SMP systems are commonly used in multi-core processors on a single server.
  - The key advantage of shared-everything architectures is their simplicity and ease of programming because all processors can access the same resources without complex communication protocols.
  - However, scalability can be a limitation. As more processors are added, contention for shared resources like memory and bus bandwidth can become a bottleneck, leading to reduced performance.
- Shared-storage architecture:
  - In a shared-storage architecture, multiple processing nodes share access to a centralized storage subsystem, but they may have their own local memory.
  - Unlike shared-everything architectures, where all resources are commonly shared, shared-storage systems focus mainly on shared access to data storage, often using **network-attached storage (NAS)** or **storage area network (SAN)** solutions.
  - This architecture is common in clustered and distributed computing environments, where nodes need to access and share data stored in a central location.
  - Shared-storage architectures can provide better scalability than shared-everything systems, as nodes can have local memory and cache data. However, they still require careful design to manage concurrent access to shared data effectively.

- Examples of systems that use shared-storage architectures include many **high-performance computing (HPC)** clusters and some types of database clusters.

The choice between shared-everything and shared-storage architectures depends on the specific requirements of the system, including factors such as scalability, ease of programming, and the nature of data access patterns. Both architectures have their strengths and trade-offs, and their suitability varies depending on the use case and workload.

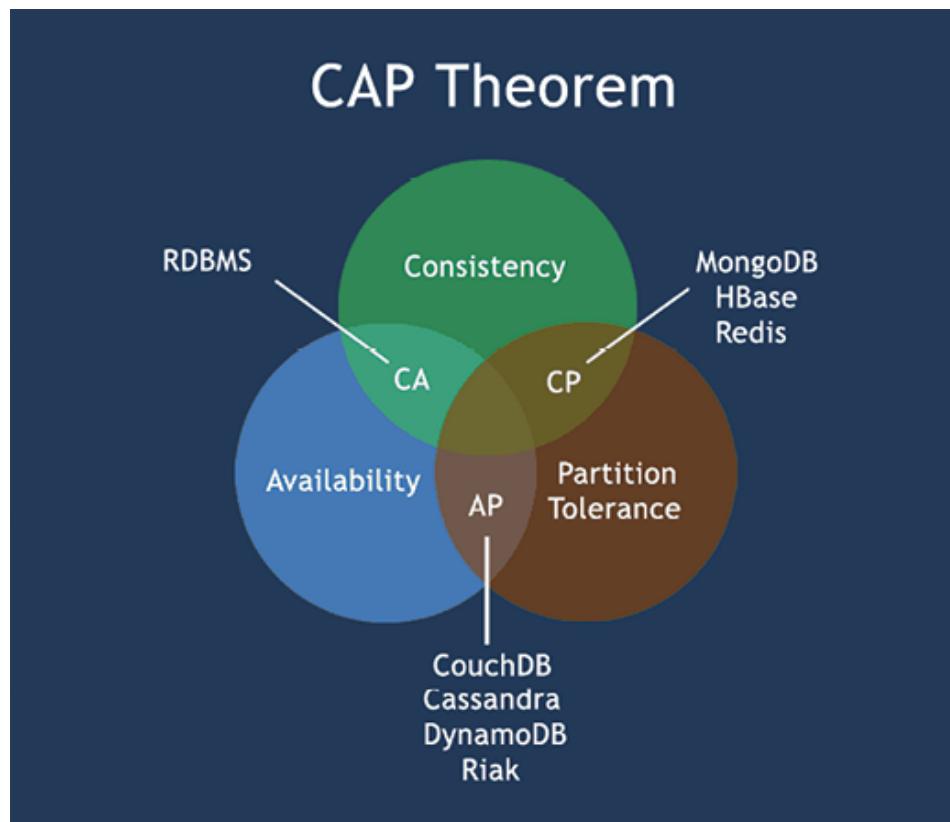
## **Relaxing consistency**

NoSQL databases relax with consistency and allows operations to perform in parallel. CAP theorem, defined by *Eric Brewer*, states about stages of consistency, availability, and partition tolerance:

<b>Consistency</b>	Every read receives the most recent write or an error
<b>Availability</b>	Every request receives a response
<b>Partition Tolerance</b>	The system continues to operate despite an arbitrary number of messages being dropped or delayed by the network between nodes

**Table 2.6: CAP theorem**

According to this theorem, Hbase, MongoDB, and Bigtable, like NoSQL databases, perform consistency and partition tolerance behavior, whereas Cassandra, couchDB, and dynamo database follows with availability and partition tolerance. MySQL and other traditional database follow availability and consistency as given in [Figure 2.8](#):



**Figure 2.8: CAP theorem**

### CAP theorem:

- **C:** Single copy consistency of data across all the physical nodes within the cluster.
- **A:** Availability system is usually available to method requests.
- **P:** Network partition.

The key finding of the CAP theorem is it is not possible to attain all three properties in an exceedingly distributed system. At the most, two properties will be achieved.

In the event of failure, it's not possible to keep up a consistent copy of information on all the replicas and still be offered for write operations. If there is a tendency to prohibit writes, get **consistency (C)**; however, availability is compromised because, currently, no node will process any write request. If there is a tendency to enable writes, then get **availability (A)**; however, consistency is compromised since updates can cause divergence in replicas.

So, the selection is between C and A, therefore giving two types of systems, CP and AP. However, a stronger thanks to impute and use the CAP theorem is not to select and choose from CP and AP or something. It is concerned properly understanding the implications of those properties and coming up with to attain them to some extent. This can be what is done in most NoSQL databases.

Consistency in CAP is not the same as Consistency in ACID, and they are completely different. The previous concerns single copy consistency across the replicas in an exceedingly distributed system, whereas the latter has additional meanings there than simply single copy consistency results committed and consistent as of sure purpose in time, therefore achieving C of ACID, and this has nothing to try and do with C of CAP. In an exceedingly distributed computer database, the 2PC protocol has got to guarantee C of each CAP and ACID.

In the following section whole story of hbase will be discussed, including step-by-step installation.

## Hbase

HBase is an open-source, distributed, and scalable NoSQL database that is part of the Apache Hadoop ecosystem. It is designed to handle massive amounts of structured or semi-structured data and provides real-time access with low-latency reads and writes. HBase follows a wide-column store data model and is suitable for applications that require high-performance, random access to large datasets. With its distributed architecture, HBase offers fault tolerance and high availability, making it an excellent choice for various use cases, such as real-time analytics, time-series data storage, and **Internet of Things (IoT)** applications.

## Installation of Hbase

Hbase is one of the most used NoSQL databases. It is based on key–value pair database. Before installing Hbase, there is a certain requirement of the system, and it should be properly configured with a bashrc file. Hbase runs on top of Hadoop, so it should be installed properly.

The following are the steps involved in its installation:

1. Downloading the tar file of HBASE and extract it from:

```
 wget http://www-us.apache.org/dist/hbase/
```

Downloading a stable version is suggested:

```
mayank@mayank-Compaq-510:~$ wget http://www-us.apache.org/dist/hbase/
--2017-07-28 09:14:45-- http://www-us.apache.org/dist/hbase/
Resolving www-us.apache.org (www-us.apache.org)... 140.211.11.105
Connecting to www-us.apache.org (www-us.apache.org)|140.211.11.105|:80... connected.
HTTP request sent, awaiting response... 200 OK
```

*Figure 2.9: Download Hbase*

2. Untar the file that has been downloaded, check version, it may differ from the given version in the following figure:

```
sudo tar -xvf hbase-1.2.4-bin.tar.gz
```

```
mayank@mayank-Compaq-510:~$ sudo tar -xvf hbase-1.2.4-bin.tar.gz
```

*Figure 2.10: unzip hbase file*

The following screenshot is after untar the file:

```
hbase-1.2.4/lib/hbase-resource-bundle-1.2.4.jar
hbase-1.2.4/lib/jetty-sslengine-6.1.26.jar
hbase-1.2.4/lib/jsp-2.1-6.1.14.jar
hbase-1.2.4/lib/jsp-api-2.1-6.1.14.jar
hbase-1.2.4/lib/servlet-api-2.5-6.1.14.jar
hbase-1.2.4/lib/jamon-runtime-2.4.1.jar
hbase-1.2.4/lib/disruptor-3.3.0.jar
hbase-1.2.4/lib/hadoop-client-2.5.1.jar
hbase-1.2.4/lib/hadoop-hdfs-2.5.1.jar
hbase-1.2.4/lib/hadoop-mapreduce-client-app-2.5.1.jar
hbase-1.2.4/lib/hadoop-mapreduce-client-common-2.5.1.jar
hbase-1.2.4/lib/hadoop-yarn-client-2.5.1.jar
hbase-1.2.4/lib/jersey-client-1.9.jar
hbase-1.2.4/lib/hadoop-yarn-server-common-2.5.1.jar
hbase-1.2.4/lib/leveldbjni-all-1.8.jar
hbase-1.2.4/lib/hadoop-mapreduce-client-shuffle-2.5.1.jar
hbase-1.2.4/lib/hadoop-mapreduce-client-jobclient-2.5.1.jar
hbase-1.2.4/lib/commons-daemon-1.0.13.jar
hbase-1.2.4/lib/libthrift-0.9.3.jar
hbase-1.2.4/lib/jruby-complete-1.6.8.jar
hbase-1.2.4/lib/spymemcached-2.11.6.jar
mayank@mayank-Compaq-510:~$
```

*Figure 2.11: installing HBase*

3. Move the untar file to **/usr/local/hbase** because, in most of the cases, Hadoop is installed in the same folder. So, there will be a folder name created with **hbase**.

```
sudo mv hbase-1.2.4-bin /usr/local/hbase
```

It will ask for the password, and after giving it, all files will move to folder **hbase** in **/usr/local/hbase**.

Verify it.

4. Go to the **hbase** folder.

```
cd /usr/local/hbase
```

```
hduser@mayank-Compaq-510:/home/mayank$ cd /usr/local/hbase/  
hduser@mayank-Compaq-510:/usr/local/hbase$ █
```

*Figure 2.12: reach folder*

There will be a folder name **conf** inside **hbase** in which there will be a file **hbase-site.xml**.

5. Reach there and open **hbase-site.xml** using **gedit**.

```
cd conf
```

```
sudo gedit hbase-site.xml
```

There will be lines of :

```
<configuration>  
</configuration>
```

In between lines, there is a need to add more number of lines, as shown in the following figure:

```
<configuration>

    <property>
        <name>hbase.rootdir</name>
        <value>/usr/local/hbase/hbasestorage</value>
    </property>

    <property>
        <name>hbase.cluster.distributed</name>
        <value>true</value>
    </property>

    <property>
        <name>hbase.zookeeper.quorum</name>
        <value>localhost</value>
    </property>

    <property>
        <name>hbase.zookeeper.property.dataDir</name>
        <value>/usr/local/hbase/hbasestorage/zookeeper</value>
    </property>

    <property>
        <name>hbase.zookeeper.property.clientPort</name>
        <value>2181</value>
    </property>

</configuration>
```

*Figure 2.13: Configuration property*

**Note:** By default, hbase.rootdir is set to /tmp/hbase-\${user.name} and similarly so for the default ZooKeeper data location, which means you will lose all your data whenever your server reboots unless you change it (Most operating systems clear /tmp on restart).

6. To start the shell of Hbase need to give commands:

**Hbase**

```

hduser@mayank-Compaq-510:/usr/local/hbase/conf$ hbase
Usage: hbase [<options>] <command> [<args>]
Options:
  --config DIR    Configuration direction to use. Default: ./conf
  --hosts HOSTS  Override the list in 'regionservers' file
  --auth-as-server Authenticate to ZooKeeper using servers configuration

Commands:
Some commands take arguments. Pass no args or -h for usage.
  shell          Run the HBase shell
  hbck          Run the hbase 'fsck' tool
  snapshot       Create a new snapshot of a table
  snapshotinfo   Tool for dumping snapshot information
  wal           Write-ahead-log analyzer
  hfile          Store file analyzer
  zkcli          Run the ZooKeeper shell
  upgrade        Upgrade hbase
  master         Run an HBase HMaster node
  regionserver   Run an HBase HRegionServer node
  zookeeper      Run a Zookeeper server
  rest           Run an HBase REST server
  thrift          Run the HBase Thrift server
  thrift2         Run the HBase Thrift2 server
  clean          Run the HBase clean up script
  classpath      Dump hbase CLASSPATH
  mapredcp       Dump CLASSPATH entries required by mapreduce
  pe             Run PerformanceEvaluation
  ltt            Run LoadTestTool
  version        Print the version
  CLASSNAME      Run the class named CLASSNAME

```

*Figure 2.14: All commands of Hbase*

## 7. To start the shell of **hbase** need to give commands:

```

hduser@mayank-Compaq-510:/usr/local/hbase/bin$ start-hbase.sh
localhost: starting zookeeper, logging to /usr/local/hbase/bin/../logs/hbase-hduser-zookeeper-mayank-Compaq-510.out
starting master, logging to /usr/local/hbase/logs/hbase-hduser-master-mayank-Compaq-510.out
OpenJDK 64-Bit Server VM warning: Ignoring option PermSize=128m; support was removed in 8.0
OpenJDK 64-Bit Server VM warning: Ignoring option MaxPermSize=128m; support was removed in 8.0
starting regionserver, logging to /usr/local/hbase/logs/hbase-hduser-1-regionserver-mayank-Compaq-510.out
hduser@mayank-Compaq-510:/usr/local/hbase/bin$ jps
5393 NameNode
5715 SecondaryNameNode
6932 Jps
5512 DataNode
5128 Main
5866 ResourceManager
6716 HMaster
6652 HQuorumPeer
6844 HRegionServer
5903 NodeManager
hduser@mayank-Compaq-510:/usr/local/hbase/bin$ hbase shell
2017-07-28 09:52:50,027 WARN [main] util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes
where applicable
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/local/hbase/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/local/hadoop/share/hadoop/common/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 1.2.4, r67592f3d0062743907f8c5ae00dbbe1ae4f69e5af, Tue Oct 25 18:10:20 CDT 2016
hbase(main):001:0> 

```

*Figure 2.15: Start of Hbase deamon*

## 8. Exit used to leave **hbase** shell.

```

hbase(main):001:0> exit
hduser@mayank-Compaq-510:/usr/local/hbase/bin$ █

```

*Figure 2.16: Exit from Hbase command prompt*

These are complete steps involved in the installation of **hbase**. All operations can be performed in **hbase** that is related to the database.

## History of Hbase

HBase is an open-source, distributed, column-oriented database that is built on top of the Apache Hadoop ecosystem. It was initially inspired by Google's Bigtable paper and designed to provide a scalable and fault-tolerant storage solution for big data.

The history of HBase can be summarized as follows:

2006	BigTable Paper Release
2007	Hbase prototype created in Hadoop contribution and Hadoop 0.15 release with usable Hbase.
2008	Hbase became subproject of Hadoop.
2010	Hbase became top level project of apache.
2011	Hbase 0.92 version released.

*Table 2.7: History of Hbase*

## Hbase data structure

Since Hbase is based on column-family-oriented architecture, all data is stored in individual column-family. In that case new column-family can be introduced later, refer to the following table:

Row key	Personal data		Family data		
	Stu id	Name	City	Father's Name	Mother's Name
101	Alok	Hyderabad	R.D. Kumar	M. Kumar	
102	Nitin	Bangalore	Anand Sharma	Hema Sharma	

103	Suraj	Delhi	Keshav Mishra	Devi Mishra
104	Dinesh	Chennai	Madhav Tiwari	A. Tiwari

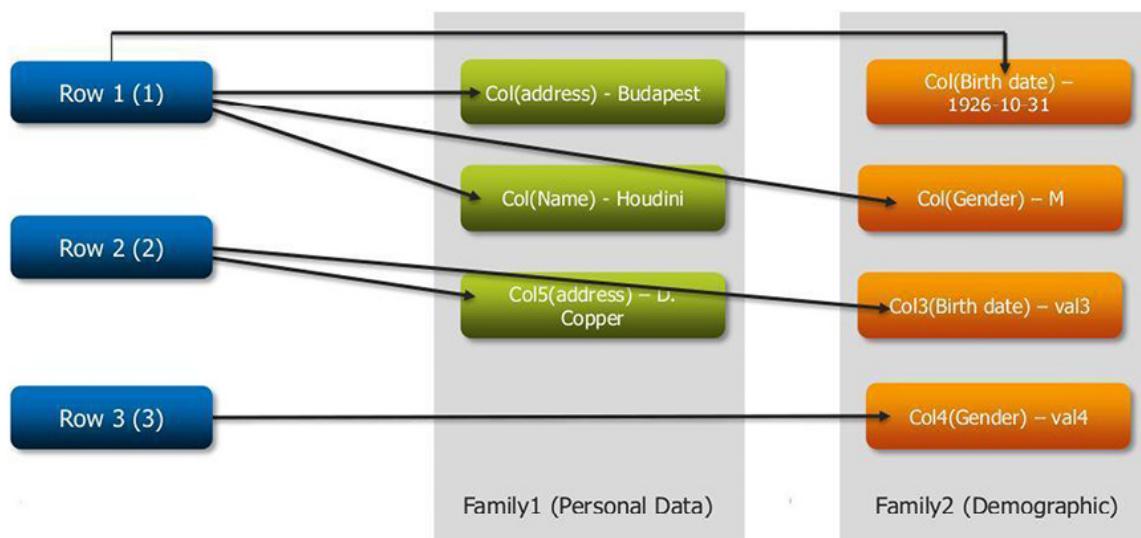
**Table 2.8:** Structure of Hbase database

In that case, personal data and family data are the column-family. Under column-family personal data, there are columns name: Name, city, and family data; there are columns name Father's name and Mother's name.

## Physical storage

The physical storage of HBase with column families involves organizing data into separate HFiles based on column families and regions based on row key ranges. This distributed storage architecture, along with compaction, caching, and MemStore mechanisms, allows HBase to efficiently handle large-scale data with high performance and low-latency access.

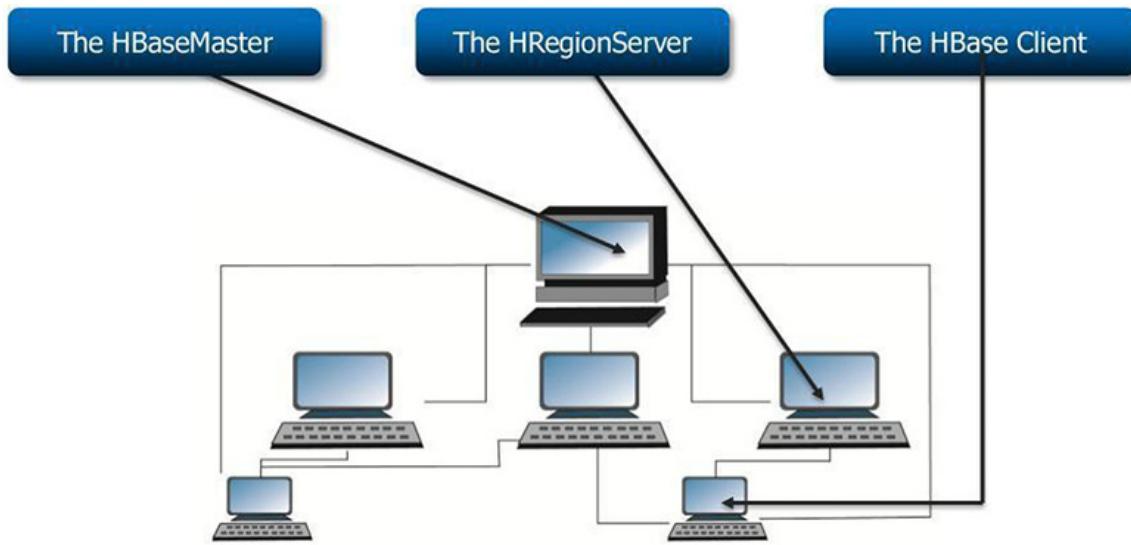
When you create an HBase table, you define one or more column families, and each column family is stored as a separate HFile. HFiles are the actual physical files where the data is written on a disk. These Hfiles, which use HDFS for storage, are then further organized into regions. Regions are a logical division of the data in a table, and they represent a range of row keys:



**Figure 2.17:** Storage of column-families

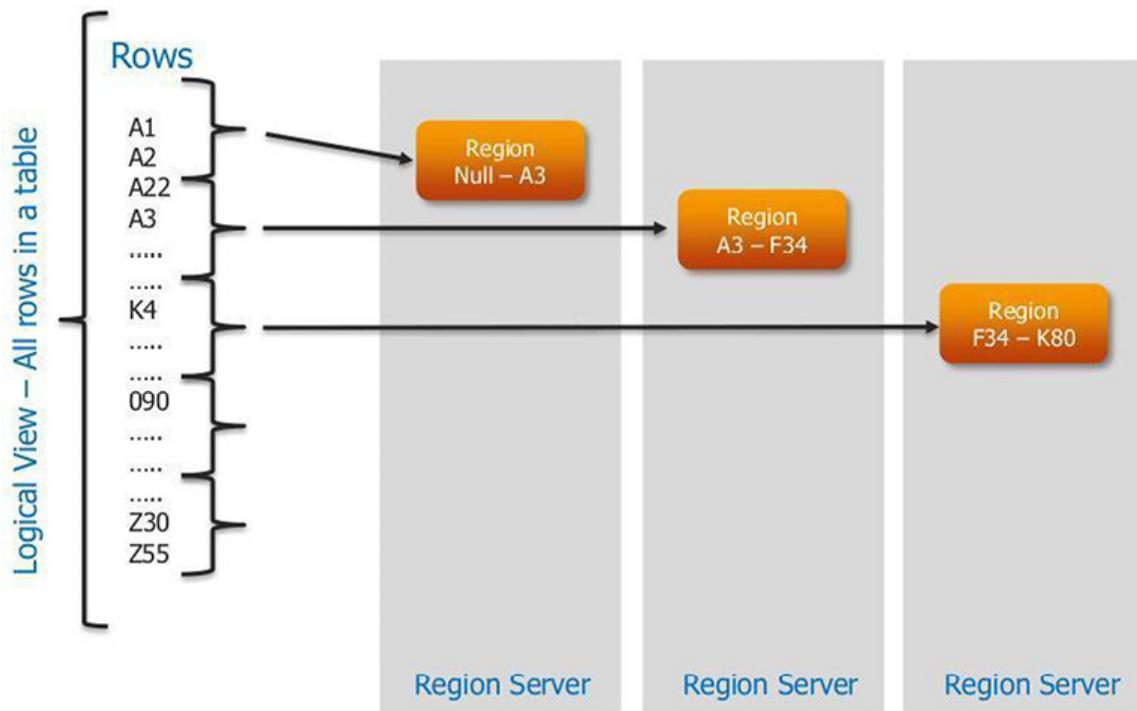
All data is stored in column-families that consist of different columns. Each column is identified uniquely by row-id; these ids play a role as the primary key

for particular records, as in [\*Figure 2.18\*](#):



*Figure 2.18: Physical architecture*

All these data are stored in different region servers, which are discussed in the following section. These region servers, as shown in [\*Figure 2.19\*](#), can partitioned according to storage of data in different systems. There are three layers of architecture used in the scenario. The primary layer is the Hbase master that governs Hregion servers that are in the second layer. The third and final layer consists of Hbase clients. There are a number of clients that are connected to Hregion servers. All these scenarios make the cluster of computers.



**Figure 2.19:** Logical view of region server

There are a number of rows in the system, and these rows are defined by the region server. All region servers have a certain number of row data. All region servers have their limitations to store data. At its threshold level, it is partitioned into separate regions:



**Figure 2.20:** Hbase physical component

[Figure 2.20](#) shows the physical component that connects the region server through Hfile and **write-ahead log (WAL)**.

## Components

There are four kinds of files used in the Hbase system; all these files have three replications of data that are stored in physical structure as in [Figure 2.20](#) called region server:

- **Region server:** There is physical storage in the system in which tables are divided by row key. A region specified in the system for the starting key to the end key. All these keys that are in the system are called region servers. These are responsible for read and write data. One thousand regions can be in one region server.
- **Hmaster:** Hmaster is responsible for creating, deleting, and updating of table. It also coordinates for recovery and load balancing by coordinating with the region server.
- **Memstore:** When there is any data written in the region server, it first appends to the commit log table and then added to the memory *memstore*. When memstore fills itself, then it goes to the filesystem HFile, that is, HDFS. The content of *memstore* keeps alive because the log is hosted on HDFS that remains available by a region server. There is also a master that is consulted first when there is any crash reported in the system of data. This information is reported by the zookeeper. This master took information from *memstore* and write it to disk. Priority goes to memstore first if required versions are available. If not, then the master consults with flush files, that is, Hfile from order that is newest to oldest till the required version find out.
- **Hfile:** After flushing data from the region of memstore it goes to Hfile. Hfile stores multiple files per column with key–value instances, as shown in [Figure 2.21](#). It is recommended to have a minimum block size between 8 KB to 1 MB to improve performance. There is one memstore per column family; after it fill out, it is flushed and goes to hfile with a sequence number. Data goes to HDFS when any Hfile has been filled. HDFS store is with its way in a distributed manner. The multi-level index is like a b+tree:
  - Key value pairs are stored in ascending order.
  - Data stored in 64KB “blocks”
  - Each block has its own index

- The last key of each block is put in the intermediate index
- The root index points to the intermediate index

Following is the Hfile structure as shown in *Figure 2.21*:

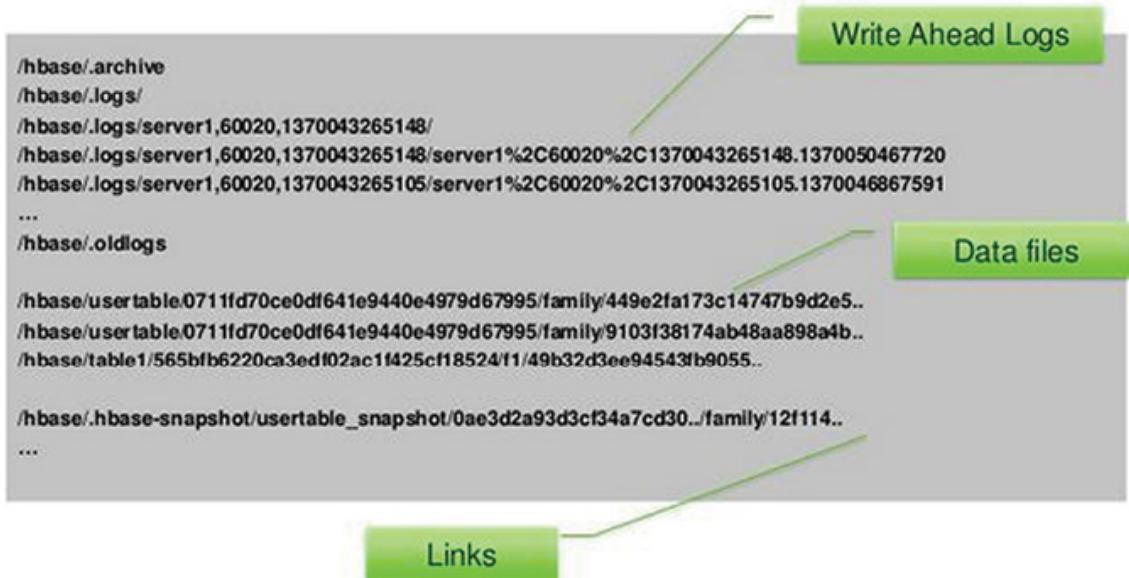
“Scanned block” section	Data Block		
	...		
	Leaf index block / Bloom block		
	...		
	Data Block		
	...		
	Leaf index block / Bloom block		
“Non-scanned block” section	...		
	Meta block	...	Meta block
Intermediate Level Data Index Blocks (optional)			
“Load-on-open” section	Root Data Index		Fields for midkey
	Meta Index		
	File Info		
	Bloom filter metadata (interpreted by StoreFile)		
Trailer	Trailer fields	Version	

*Figure 2.21: Hfile structure*

- **Write Ahead Log:** There is a file in a distributed environment that stores data that is no longer required to be stored permanently, but it can be used in case of failure of data. Every region server uses a **Write Ahead Log (WAL)** to add updates like puts and deletes. WAL maintains the status of machine behavior like redo and undo type of information. So, it may be considered as the lifeline of the system at the time when the system is in trouble because it can record checkpoints about server status at the time of its crash. So, it can be stated that in case of failure of WAL whole system may fail.

HLog is the file that implements WAL in structure. Only one instance of HLog class per **HRegionServer** with **append()** method that calls **doWrite()** method internally. HLog keeps changes of track with sequence

numbers that can start from zero or the last persisted number in the file system. This field is stored as a meta field in each **HLog** file:

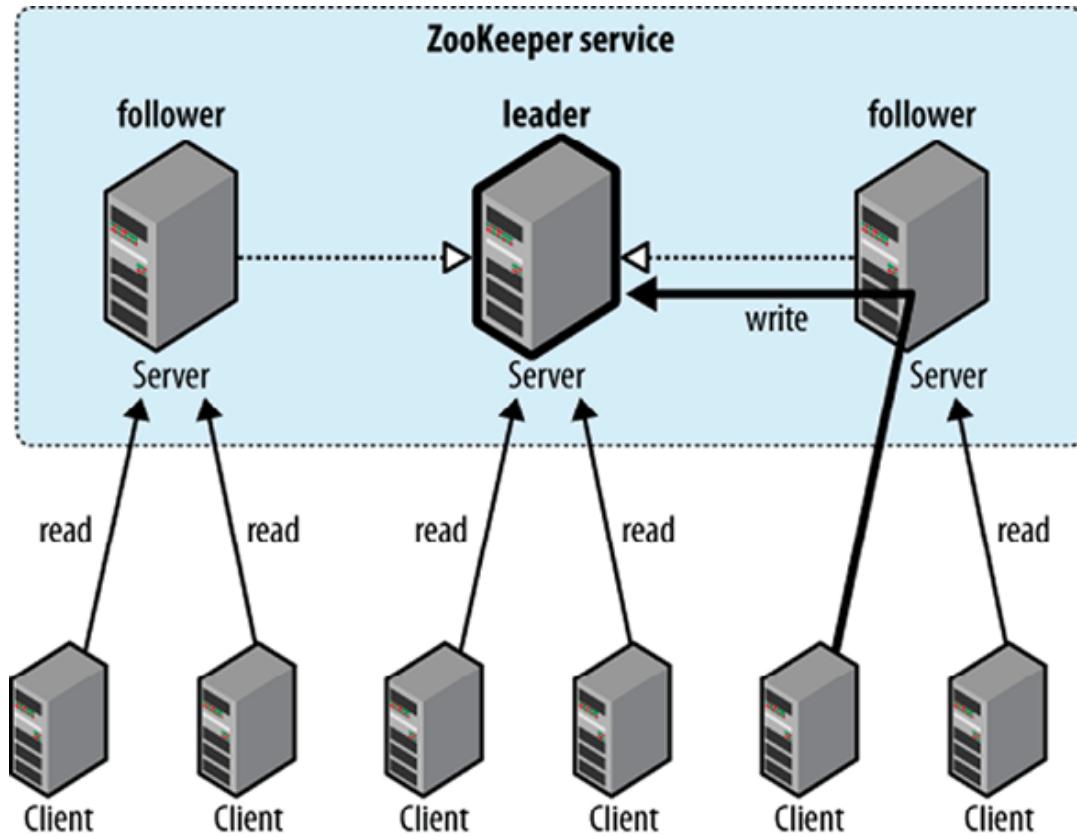


**Figure 2.22:** Sample log file

- **Zookeeper:** Zookeeper is the most important component in the cluster system that is used to manage the cluster state. It contains information like the location of the root catalog and the address of the master node of the running cluster. Since all data is stored in a distributed manner, it is very typical task to know about fault if it happens in cluster. It may lead to partial failure of system. These failures may be the reason for network issue, node issue, data issue, and so on ([Figure. 2.22](#)). Zookeeper is not responsible for not happening of partial failure, but they can provide a solution to handle such type of situations with the following characteristics:
  - Zookeeper is simple with its file system that elaborates on sample operations involved.
  - Zookeeper is expressive in coordination with data structure and protocols used to build class.
  - It avoids machines to get failure with single-point value. So, it is highly available all the time.
  - It is an open source that shared repository with some coordination pattern.
  - ZooKeeper can help mitigate issues like deadlocks and race conditions in distributed systems, although it does not completely eliminate them.

ZooKeeper is a distributed coordination service that provides primitives for managing distributed processes and ensuring their synchronization.

- It can perform lots of operations per second, as Yahoo expressed 10,000 programs per second with this. ([Figure 2.23](#)):



*Figure 2.23: Zookeeper*

## Hbase shell commands

The HBase shell command is a command-line interface tool provided by HBase that allows users to interact with an HBase cluster and perform various administrative and data manipulation tasks. The HBase shell provides a convenient way to execute HBase commands and interact with the underlying HBase database:

Task	Command and output
To create table	hbase(main):001:0> create 'tablename', 'columnfamily1', 'columnfamily2'

Task	Command and output
	<pre>hbase(main):001:0&gt; create 'student', 'personaldatal', 'familydata' 0 row(s) in 1.1300 seconds =&gt; Hbase::Table - student</pre>
To check listing of the created table	<pre>hbase(main):001:0&gt; list</pre> <p>TABLE</p> <p>student</p> <p>2 row(s) in 0.0340 seconds</p>
To enter value in table	<p>This is used to enter value in table:</p> <pre>hbase(main):001:0&gt;put 'tablename', 'rowname', 'columnvalue', 'value'</pre> <pre>hbase(main):001:0&gt;put 'student', 'r1', 'c1', value, 10 hbase(main):001:0&gt;put 'student', 'r1', 'c1', value, 15 hbase(main):001:0&gt;put 'student', 'r1', 'c1', value, 20</pre>
To check status of Hbase	<pre>hbase(main):001:0&gt;status</pre> <pre>hbase(main):001:0&gt;status 'student'</pre> <p>It will show detail of status of table student.</p>
To check version	<pre>hbase(main):001:0&gt;version</pre> <pre>hbase(main):001:0&gt;version (This will show current version of hbase)</pre>
To get help in table	<pre>hbase(main):001:0&gt;table_help</pre>
TTL(Time To Live) - Attribute	<p>In HBase, Column families can be set to time values in seconds using TTL. HBase will automatically delete rows once the expiration time is reached.</p>

Task	Command and output
To check description of table	hbase(main):001:0>describe 'tablename' hbase(main):001:0>describe 'student' (This will provide detail about schema of student table)
To disable table	hbase(main):001:0>disable 'tablename' hbase(main):001:0>disable 'student'
To enable table	hbase(main):001:0>enable 'tablename' hbase(main):001:0>enable 'student'
To displays all the filters present in HBase	hbase(main):001:0> show_filters This command displays all the filters present in HBase like ColumnPrefix Filter, TimestampsFilter, PageFilter, FamilyFilter, and so on.
To drop a table	hbase(main):001:0>drop 'tablename' hbase(main):001:0>drop "student" (Before drop any table it needs to disable first)
To verify table enable or not	hbase(main):001:0> is_enabled 'tablename' hbase(main):001:0> is_enabled 'tablename' (It use to check whether table is enabled or not)
To alter table	hbase(main):001:0>alter 'tablename', NAME=>'column familyname', VERSIONS=>5

Task	Command and output
	<pre>hbase(main):001:0&gt; alter 'student', NAME=&gt;'mayank', VERSIONS=&gt;5</pre> <p>Define two new columns to our existing table student.</p> <pre>hbase(main):001:0&gt; alter 'student', 'cf', {NAME =&gt;'Nitin', IN_MEMORY =&gt; true}, {NAME =&gt;'Suraj', VERSIONS =&gt; 5}</pre> <p>To change or add the “Alok” column family in table “student” from the current value to keep a maximum of five cell VERSIONS:</p> <pre>hbase(main):001:0&gt;alter 'student', NAME=&gt;'alok', VERSIONS=&gt;5</pre>
To count of a number of rows in a table	<pre>hbase(main):001:0&gt; count 'tablename', CACHE =&gt;1000</pre> <pre>hbase(main):001:0&gt; count 'student', CACHE =&gt;1000 (to extract every 1000 rows, by default it is 10)</pre>
To count row in intervals	<pre>hbase(main):001:0&gt;count 'tablename', INTERVAL =&gt;10, CACHE=&gt; 1000</pre> <pre>hbase(main):001:0&gt;count 'tablename', INTERVAL =&gt;10, CACHE=&gt; 1000 (Count 1000 rows with interval of 10)</pre>
To get a row or cell contents present in the table	<pre>hbase(main):001:0&gt; get 'tablename', 'rowname', { Additional parameters}</pre> <p>&lt;Additional Parameters&gt; include TIMERANGE, TIMESTAMP, VERSIONS and FILTERS.</p> <pre>hbase(main):001:0&gt;get 'student', 'r1', {COLUMN =&gt;'c1'} (For table 'student' row r1 and column c1 values will display using this command)</pre>
To delete a table from database	<pre>hbase(main):001:0&gt; delete 'tablename', 'rowname', 'columnname'</pre> <pre>hbase(main):001:0&gt;delete 'student', 'r1', 'c1' (This will delete r1 row and c1 column in student table)</pre>

Task	Command and output
<b>To delete all table from database</b>	hbase(main):001:0> deleteall 'tablename', 'rowname'
	hbase(main):001:0> deleteall 'student', 'r1' (This will delete all cells in r1 row)
<b>To truncate table</b>	hbase(main):001:0>truncate 'tablename'
	hbase(main):001:0>truncate 'student' (After truncating of an hbase table, the schema will present but not the records)
<b>Scan entire table and show content</b>	hbase(main):001:0>scan 'tablename'
	hbase(main):001:0>scan 'student' (This will scan and give all schema information of student table)

**Table 2.9: Table commands**

## Different usages of the scan command

The scan command in the HBase shell is used to retrieve data from an HBase table. It allows you to perform scans across rows, columns, or the entire table based on the specified criteria. Here are different usages of the scan command in HBase:

Command	Description
hbase> scan '.META.', {COLUMNS =>'info:regioninfo'}	Display all metadata information related to columns that are present in the tables in HBase
hbase> scan 'guru99', {COLUMNS => ['cf1', 'cf2'], LIMIT => 10, STARTROW =>'xyz'}	It displays contents of table for student with their column families cf1 and cf2 limiting the values to 10
hbase> scan 'student', {COLUMNS =>'c1', TIMERANGE => [1303668804, 1303668904]}	It displays contents of student with its column name cf1 with the values present in between time range attribute value
hbase> scan 'student', {RAW => true, VERSIONS =>10}	In this command, RAW=> true provides an advanced feature like to display all the cell values present in the table student

---

**Table 2.10:** Various Hbase commands

## Terminologies

Here are the terms that are used in the Hbase database:

### Version stamp

The applications that store data use labeled table for that, which consists of rows and columns. The intersection of rows and columns in table is called **versioned**. If there are no user changes, then **timestamps** are the default version for any table that are auto-assigned by Hbase. It can be stated that any value that is in row can be treated as serialized data structure. All these keys in table are sorted by the primary key that is byte-ordered. Row columns are into a family that is column-family. These families are uniquely distributed and can be added further easily without giving null to blank data. In the preceding situation, personal data and family data were the column-family that is uniquely represented in database.

The traditional database does not provide much flexibility to add data in columns after creating schema. Even after creating a schema, if there is no data entered then there is a mandatory requirement to show null as entry in database column. But in the NoSQL database if in future any new column-family can be easily added and can be used with unique individual columns in these families.

These new column-families can also be added on demand basis as per the requirement of system.

All column-families are stored physically in storage and conceptually in filesystem. So, it provides more accurate result because of systematic column-oriented architecture. In Hbase only cells are versioned, which means rows are sorted and keep record as it is updating itself, for example, the password of an individual on any page, many of companies keep a record of the last three to five passwords that remind the user about the old password.

### Region

Tables that are in Hbase are automatically distributed horizontally into regions. These regions consist of rows in table. At the start, a single region started to store column-families of table, and at the threshold level, these regions can be further distributed to grow their size. The new structure of the family was created with

equal size. These regions make clusters, and as all tables are stored physically, these clusters are stored physically anywhere.

## **Locking**

Locking mechanisms in NoSQL databases differ significantly from those in traditional relational databases due to the distributed and often schema-less nature of NoSQL systems. NoSQL databases typically aim to provide high availability and scalability, which can make traditional locking approaches impractical. Instead, they often use optimistic concurrency control and distributed consensus algorithms.

In many NoSQL databases, locks are not used to control concurrent access to data. Instead, they rely on techniques like versioning or timestamps to manage concurrent modifications. When a client retrieves a piece of data, it may also retrieve a version or timestamp associated with that data. When the client wants to update the data, it sends the new data and the version or timestamp it originally retrieved. The database checks whether the current version matches the version provided by the client. If they match, the update is applied; if not, it indicates that someone else has modified the data in the meantime, and the client may need to resolve the conflict.

In distributed NoSQL databases, consensus algorithms like Paxos or Raft are often used to ensure that nodes in the cluster agree on the order of operations. These algorithms help manage distributed locking and coordination, allowing for consistency and high availability even in large, geographically distributed systems. However, the specifics of how locking is implemented can vary widely among different NoSQL databases, depending on their architecture and design goals.

## **Conclusion**

The NoSQL database revolution has transformed the way we store and manage data in the modern digital era. NoSQL databases offer unparalleled scalability, flexibility, and speed, making them a critical component for organizations dealing with large volumes of data or dynamic data models. These databases are not a one-size-fits-all solution, but they provide a valuable alternative to traditional relational databases, enabling businesses to adapt to changing data requirements, embrace distributed computing, and support the demands of modern Web and mobile applications. As the NoSQL landscape continues to evolve, it is essential for organizations to carefully assess their specific data needs and choose the right

NoSQL database technology to harness the full potential of their data-driven initiatives.

In this chapter, we have explored the various types of NoSQL databases, including document stores, key-value stores, column-family stores, and graph databases, each suited to different use cases. We have also delved into the advantages and challenges of NoSQL databases, highlighting their strengths in horizontal scalability and their flexibility in handling unstructured or semi-structured data. However, it's crucial to acknowledge that NoSQL databases are not a silver bullet, and they may not be the best fit for every data scenario. Organizations should carefully evaluate their data requirements and consider factors such as data consistency, query capabilities, and the complexity of their applications when deciding whether to adopt NoSQL databases as part of their data management strategy.

In the upcoming chapter, we will study about various operations on database using mapreduce.

### **Join our book's Discord space**

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



# CHAPTER 3

## MapReduce Technique

### Introduction

MapReduce is a programming model that is used for processing purposes of large amounts of data. It supports Java, C++, Ruby, and Python. The most preferable language used in this is Java and Python. This book will focus on Java for MapReduce programming. MapReduce consists of two phases of programming for processing, for example, map and reduce. Each phase of programming required input as well as output. The output of the mapper class goes as input in the reducer class.

The only possible approach to coping with large-data issues these days is to divide and conquer, an elementary concept in computer science. The fundamental plan is to partition a large downside into smaller sub-problems. To the extent that the sub-problems are independent, they will be tackled in parallel by totally different workers threads in a processor core, cores in a multi-core processor, multiple processors in a machine, or several machines in a cluster. Intermediate results from every individual employee are then combined to yield the final output.

MapReduce works on key–value pair data. These pair function to read and perform write operations on data. A simple example of key–value can be for a document is page offset as key and whole line as value. Other examples may be page number as key and data as value, account number as key account detail as value.

Key–value pairs form the essential data structure in MapReduce. Keys and values may be primitives such as integers, floating point values, strings, and

raw bytes, or they will be arbitrarily advanced structures (lists, tuples, associative arrays, and so on). Programmers usually want to define their own custom data types, although the variety of libraries, like Protocol Buffers, is the task. Part of the design of MapReduce algorithms involves imposing the key–value structure on arbitrary datasets. For a group of websites, keys may be URLs, and values might be the actual markup language content. For a graph, keys may represent node IDs, and values might contain the closeness lists of those nodes. In some algorithms, input keys are not significantly meaningful and are simply unnoticed throughout process, whereas in other cases, input keys are used to unambiguously determine information (such as a record ID).

## Structure

In this chapter, we will learn the following topics:

- MapReduce architecture
- MapReduce datatype
- MapReduce program structure
- Partitioner and combiner
- Program using partitioner and combiner
- Composition of MapReduce

## Objectives

The primary objective of MapReduce in a program is to provide a scalable and efficient framework for processing and analyzing large volumes of data in a parallel and distributed manner. Breaking down complex data tasks into two key phases—mapping and reducing. MapReduce allows for the seamless execution of computations across a cluster of machines. The mapping phase involves transforming and filtering the input data into key–value pairs, whereas the reducing phase focuses on aggregating and summarizing these pairs to produce meaningful results. The overarching goal is to leverage the processing power of multiple nodes, thus significantly

speeding up data processing and enabling the analysis of data sets that would be otherwise impractical to handle using traditional methods.

Another crucial objective of MapReduce is fault tolerance. In a distributed computing environment, hardware failures and system glitches are common occurrences. MapReduce is designed to handle such challenges gracefully by ensuring that tasks are automatically reassigned to available nodes when failures occur. This robustness ensures that data processing jobs can continue without manual intervention, making MapReduce a reliable or dependable framework for managing and analyzing vast datasets. Overall, the main objectives of MapReduce in a program are to achieve scalability, parallelism, and fault tolerance, ultimately making it a versatile tool for big data processing and analytics. In this chapter, our focus will be exclusively on delving into MapReduce techniques. Subsequently, in later sections of this book, we will delve into the practical implementation of MapReduce for processing and analyzing data.

## **MapReduce architecture**

There are two main components that are used in MapReduce: map and reduce. With key–value pair, all forms of data can be defined:

For example, **Firstname/Mayank**

**Transaction\_amount/20.07**

**Search\_technique/database**

All the data can be referred with MapReduce architecture. All intermediate data of map part get combined and goes to the reduce part. The following is an example of data that is of patent. There are patent and sub-patent (*Figure 3.1*) in data separated by a space; there is a need to find number of sub-patent associated with each ID.

2	2.12
2	2.93
2	2.75
2	2.20
2	2.29
3	3.201
3	3.175
3	3.196
3	3.42
3	3.14
3	3.108
3	3.4
4	4.201
4	4.228
4	4.17
4	4.89
5	5.234
5	5.101
5	5.177
5	5.42
5	5.86
6	6.86
6	6.16
6	6.171
6	6.6

**Figure 3.1:** Sample data with two field

These are billions numbers of records that need to be processed with their sub-patent id. The desired output should be name of patent and its corresponding sub-patent ID.

These lines are calculated as key line numbers and the whole line data as value:

(0, 1 1.16

22, 1 1.11

83, 1 1.14

104, 1 1.27

.....

206, 5 5.45

217, 5 5.78

.....

310, 8 8.98

319 8 8.76

..... . )

The first part of the line is called as offset, considered as a key that is used in map phase. The value part will be a whole line of data, and this data is considered as integer. So, the reducer function will get the input as follows:

(1 [1.12, 1.45, 1.67, 1.89..... . ] )

(2 [2.13, 2.45, 2.56, 2.78, 2.98.....] )

..... .

..... .

This can be easily written as the sum of all values that is considered as value part in that. Whenever processing is done, the key and value field changes accordingly.

## MapReduce datatype

Following are the data types used in MapReduce:

**ByteWritable, IntWritable, LongWritable, FloatWritable,  
DoubleWritable, Text, BooleanWritable, and NullWritable**

The preceding datatypes are the alternative of traditional data types of Java, that is, Byte, int, long, float, double, string, and Boolean, respectively.

## File input format

The following are the types that can be used for processing purposes:

- XML (most common to use)
- JSON (mostly downloaded data from the Web in that format)
- SequentialFile (key–value pair format)
- Avro (record-oriented, not key–value oriented)
- Thrift (do not use directly)
- CSV (other format)
- Parquet (well-suited for big data processing frameworks and analytics)

## Java MapReduce

The following program consists of four parts: import files, map phase, reduce phase, and driver detail:

```
// Externally import file

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
```

```
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.TextInputFormat;
import org.apache.hadoop.mapred.TextOutputFormat;

// Map phase

public static class Map extends MapReduceBase
implements
Mapper<LongWritable, Text, Text, Text> {
    Text k= new Text();           Text v= new Text();
    @Override
    public void map(LongWritable key, Text value,
    OutputCollector<Text, Text> output, Reporter
    reporter)
    throws IOException {
        String line = value.toString();
        StringTokenizer tokenizer = new
        StringTokenizer(line,"");
        while (tokenizer.hasMoreTokens()) {
            String jiten= tokenizer.nextToken();
            k.set(jiten);
            String jiten1= tokenizer.nextToken();
```

```
v.set(jiten1);

output.collect(k,v);

}

        }

// Reduce phase

public static class Reduce extends MapReduceBase
implements

Reducer<Text, Text, Text, IntWritable> {

@Override

public void reduce(Text key, Iterator<Text> values,
OutputCollector<Text, IntWritable> output, Reporter
reporter) throws IOException {

int sum = 0;

while (values.hasNext()) {

values.next();    sum ++;                    }

output.collect(key, new IntWritable(sum));

}      }

// Driver Class

public static void main(String[] args) throws
Exception {

JobConf conf = new JobConf(patent.class);
```

```
conf.setJobName("patent");
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(IntWritable.class);
conf.setMapOutputKeyClass(Text.class);
conf.setMapOutputValueClass(Text.class);
conf.setMapperClass(Map.class);
conf.setReducerClass(Reduce.class);
conf.setInputFormat(TextInputFormat.class);
conf.setOutputFormat(TextOutputFormat.class);
FileInputFormat.setInputPaths(conf, new Path(args[0]));
FileOutputFormat.setOutputPath(conf, newPath(args[1]));
JobClient.runJob(conf);
```

The first phase of any program is the declaration of files required for running programs. Hadoop-supportive files are not in-built in the Java framework, so there is a need to import these files.

Second, there is a map phase declared, and this map phase extends the class of **MapReduceBase** and implements the Mapper class. Inside, it needs to define key and value pair for input and output, here **LongWritable** and Text are datatypes for input and output. Map function is being overridden in which key datatype is defined as **LongWritable** and Text as value for that input which is overridden. After that, logic can be implemented.

The third phase consists of reduce concept. This phase of reduce also extends **MapReduceBase** and implement reducer class. In that logic, sum will be written that summarized all sub-patent records in a single id field.

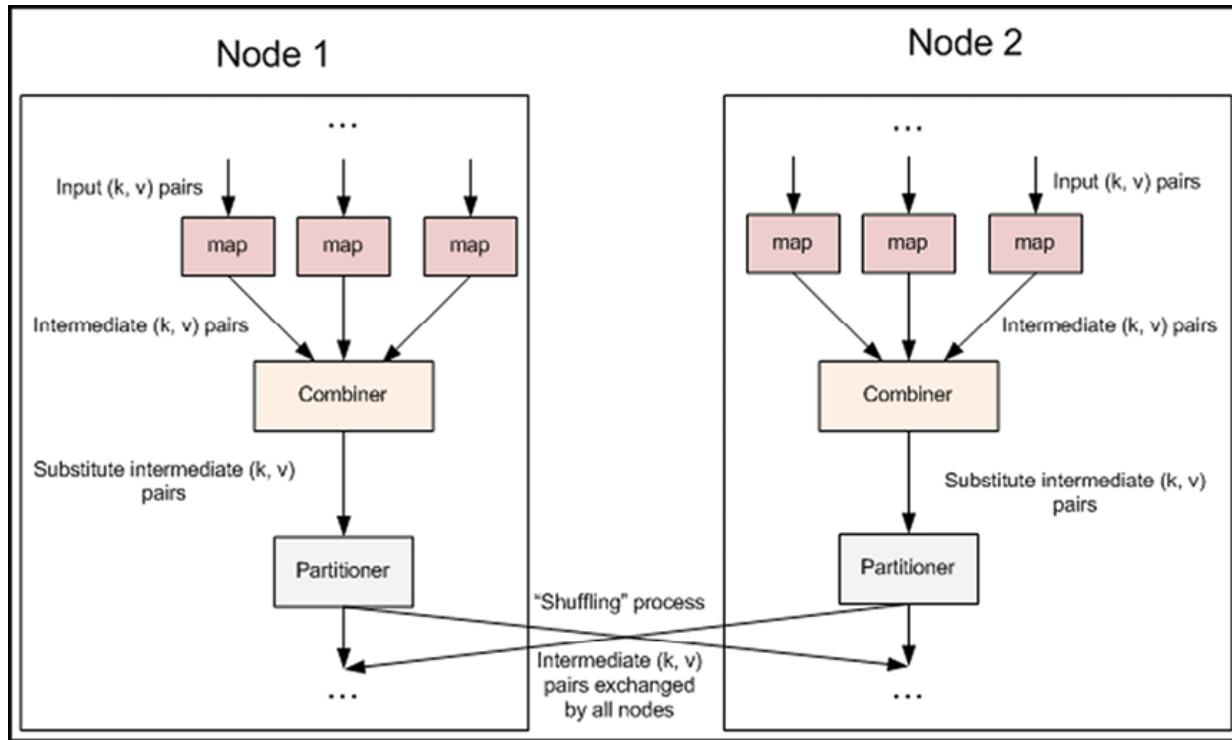
Sometimes reduce phase becomes optional and does not require for the execution of the program.

At last, there is the driver class. This is a common class usually because of the common factors that are involved in a declaration, such as output key and output value class, that is, in which format needs the output that needs to be collected. Map output with key and value pair can also be defined with text. Importantly, input format and output format also define that instruct program about the format in which data is accepted.

## Partitioner and combiner

Partitioner and combiner are special-purpose programming utilities that are used for special purposes. As *Figure 3.2* shows, if there are two nodes and these nodes store data in a distributed manner, then the output will be combined results from both of them. Now, if our requirement is to collect the output in different nodes as per our requirement, then these methods are used. As in the previous section of the map, reduce functionality of all data divided with key–value pair. These pair of data values generate separate output from each of the map functions. So, there is a collective list of outputs that can be combined; here, the combination comes into the picture, and it does the function of a combination of output that was the output of map functions. So, a combiner can be stated as a mini reducer. The output of this *mini reducer* goes to the input of the partitioner. Now, it is the partitioner's responsibility to store it into separate nodes; for that, it is required to define the number of reducers. Number of reducer are the number of output value that the partitioner will be produced; refer to the following figure:

map----->combiner----->partitioner-----  
>shuffle and sort----->reduce----->output

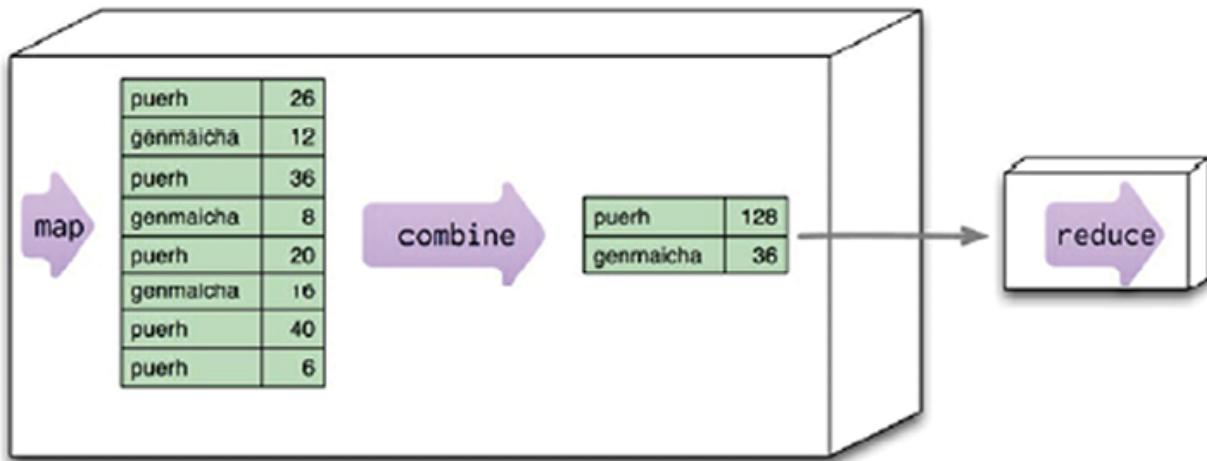


**Figure 3.2: Partitioner and combiner**

Following are the step-wise functions that the partitioner and combiner perform:

- Output data collective from the map is responsible for partitioner functionality. This is specified by the user.
- The output of partitioner's data is responsible for giving input for reducers.
- Partitioner divided output of map with specified criteria with intermediate key–value pair.
- Partitioning involves the hash value of the key and then takes the mood of the value with a number of reducers used in that.

The primary goal of the combiner is optimization by saving as much bandwidth as possible. It can be done by minimizing the number of key/value pairs that will be shuffled across the network and provided as input to the reducer, as shown in [Figures 3.2 and 3.3](#):



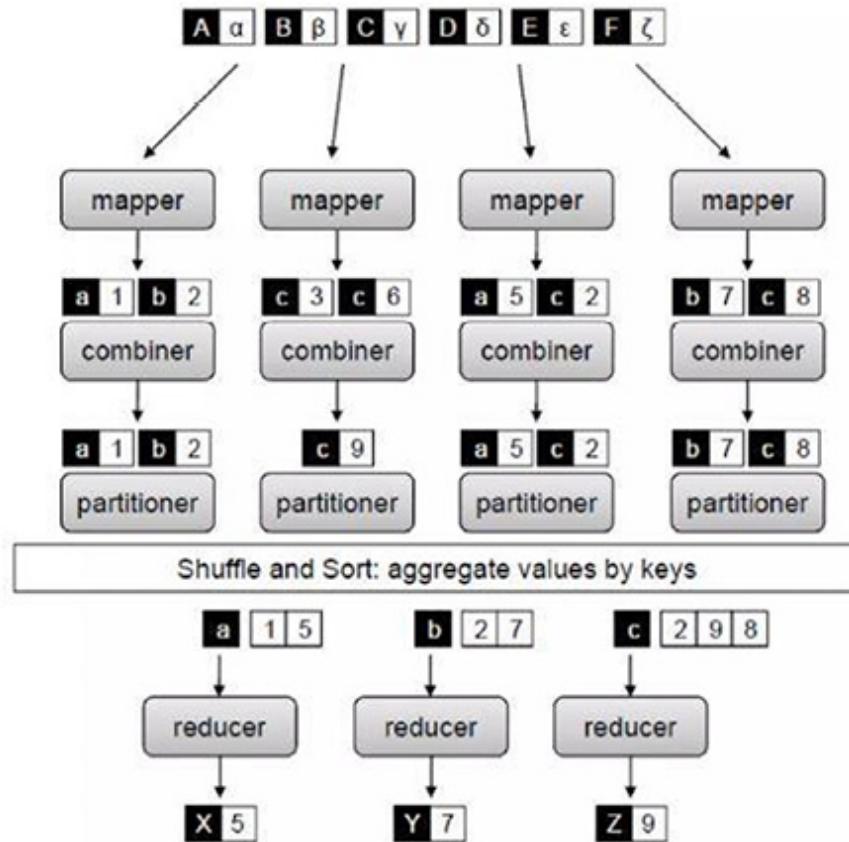
**Combining reduces data before sending it across the network.**

*Figure 3.3: Use of combiner*

- Combiner is an optimization in MapReduce that uses local aggregation before the shuffle and sort method.
- If a combiner is used, then key–value pair is not immediately written into output.
- A number of reducers used in the program is declared in the driver class.
- A number of partitioners will be equal to a number of combiners.

*Figure 3.4* shows the description of the role of the practitioner and combiner:

## MapReduce with Partitioner and Combiner



**Figure 3.4:** Full description of partitioner and combiner

### Example of MapReduce

Following is the sample program in which partitioner and combiner are used:

```
import java.io.IOException;
import java.util.Iterator;
import java.util.StringTokenizer;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
```

```
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Partitioner;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.TextInputFormat;
import org.apache.hadoop.mapred.TextOutputFormat;

public class WithPartitioner {

    public static class Map extends MapReduceBase
        implements

            Mapper<LongWritable, Text, Text,
IntWritable> {
```

```
    @Override
    public void map(LongWritable key, Text
value,
                    OutputCollector<Text, IntWritable>
output, Reporter reporter)
                    throws IOException {

        String line = value.toString();
        StringTokenizer tokenizer = new
StringTokenizer(line);

        while (tokenizer.hasMoreTokens()) {
            value.set(tokenizer.nextToken());
            output.collect(value, new
IntWritable(1));
        }
    }

    // Output types of Mapper should be same as
arguments of Partitioner
}
```

```
public static class MyPartitioner implements  
Partitioner<Text, IntWritable> {  
  
    @Override  
  
    public int getPartition(Text key,  
IntWritable value, int numPartitions) {  
  
        String myKey =  
key.toString().toLowerCase();  
  
        if (myKey.equals("hadoop")) {  
  
            return 0;  
        }  
        if (myKey.equals("data")) {  
  
            return 1;  
        } else {  
  
            return 2;  
        }  
  
    }  
  
    @Override  
    public void configure(JobConf arg0) {
```

```
    }
```

```
}
```

```
public static class Reduce extends  
MapReduceBase implements  
    Reducer<Text, IntWritable, Text,  
IntWritable> {  
  
    @Override  
    public void reduce(Text key,  
Iterator<IntWritable> values,  
        OutputCollector<Text, IntWritable>  
output, Reporter reporter)  
        throws IOException {  
  
        int sum = 0;  
        while (values.hasNext()) {  
            sum += values.next().get();  
  
        }  
}
```

```
        output.collect(key, new
IntWritable(sum));

    }

}

public static void main(String[] args) throws
Exception {

    JobConf conf = new
JobConf(WithPartitioner.class);

    conf.setJobName("wordcount");

    // Forcing program to run 3 reducers
    conf.setNumReduceTasks(3);

    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);

    conf.setPartitionerClass(MyPartitioner.class);

    conf.setOutputKeyClass(Text.class);
```

```

        conf.setOutputValueClass(IntWritable.class);

        conf.setInputFormat(TextInputFormat.class);

conf.setOutputFormat(TextOutputFormat.class);

        FileInputFormat.setInputPaths(conf, new
Path(args[0]));

        FileOutputFormat.setOutputPath(conf, new
Path(args[1]));

JobClient.runJob(conf);

}

}

```

## **Situation for partitioner and combiner**

Let us suppose there are elections to be held in the country, and there is a need to store output data in different nodes; in that situation, there is a requirement to process data with a combiner and partitioner. The machine will be defined with 29 partitioners that produce output results in 29 nodes, and these nodes run 29 reducers that will be already defined with the driver class of the program. So, this method provides us flexibility so that the program can behave according to demand.

In the preceding program, simple word count refer to store each word to an individual node as output. So, as per programmer need, it can identify record to store in to node as per requirement.

## Uses of combiner

Following are the points that describe the uses of a combiner:

- Combiner helps to reduce amount of data transfer because it is called a mini reducer. This can save bandwidth for transferring data.
- It increases the efficiency of system with a MapReduce job.
- Combiner provides optimized result, so it is not guaranteed to run number of combiners.
- Number of combiners depends on the intermediate record of data from map input, which is the property of `min.num.spills.for.combine`.
- Number of combiners defined in driver class of MapReduce job.

## Composing MapReduce calculations

As discussed, MapReduce consists of two parts in computations: map and reduce. Commands used in Hadoop will be discussed later in this book. Let us take a scenario of running a program with MapReduce calculations. There is an “abc” file on which operations need to be performed, and WC1.jar is the program name by which all instructions with bundling of jar files will be executed on the “abc” file:

```
hduser@mayank-Compaq-510:/home/mayank$ hadoop jar WC1.jar /abc/out
17/08/01 13:53:34 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
17/08/01 13:53:35 INFO Configuration.deprecation: session.id is deprecated. Instead, use dfs.metrics.session-id
17/08/01 13:53:35 INFO jvm.JvmMetrics: Initializing JVM Metrics with processName=JobTracker, sessionId=
17/08/01 13:53:35 INFO jvm.JvmMetrics: Cannot initialize JVM Metrics with processName=JobTracker, sessionId= - already initialized
17/08/01 13:53:35 WARN mapreduce.JobSubmitter: Hadoop command-line option parsing not performed. Implement the Tool interface and execute your application with ToolRunner to remedy this.
17/08/01 13:53:35 INFO mapred.FileInputFormat: Total input paths to process : 1
17/08/01 13:53:35 INFO mapreduce.JobSubmitter: number of splits:1
17/08/01 13:53:36 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_local1815802876_0001
17/08/01 13:53:36 INFO mapreduce.Job: The url to track the job: http://localhost:8080/
17/08/01 13:53:36 INFO mapreduce.Job: Running job: job_local1815802876_0001
17/08/01 13:53:36 INFO mapred.LocalJobRunner: OutputCommitter set in config null
17/08/01 13:53:36 INFO mapred.LocalJobRunner: OutputCommitter is org.apache.hadoop.mapred.FileOutputCommitter
17/08/01 13:53:36 INFO mapred.LocalJobRunner: Waiting for map tasks
17/08/01 13:53:36 INFO mapred.LocalJobRunner: Starting task: attempt_local1015002076_0001_m_000000_0
17/08/01 13:53:36 INFO Mapred.Task: Using ResourceCalculatorProcessTree : []
17/08/01 13:53:37 INFO mapred.MapTask: Processing split: hdfs://localhost:54310/abc:0+56
17/08/01 13:53:37 INFO mapred.MapTask: numReduceTasks: 1
17/08/01 13:53:37 INFO mapred.MapTask: (EQUATOR) 0 kvl 26214396(104857584)
17/08/01 13:53:37 INFO mapred.MapTask: mapreduce.task.io.sort.mb: 100
17/08/01 13:53:37 INFO mapred.MapTask: soft limit at 83886080
17/08/01 13:53:37 INFO mapred.MapTask: bufstart = 0; bufvoid = 104857600
17/08/01 13:53:37 INFO Mapred.MapTask: kvstart = 26214396; length = 6553600
17/08/01 13:53:37 INFO mapred.MapTask: Map output collector class = org.apache.hadoop.mapred.MapTask$MapOutputBuffer
```

*Figure 3.5: Scenario of executing commands*

Composing MapReduce is an essential skill in the world of big data processing and analysis. The reason for this is that many real-world data

tasks are complex and cannot be efficiently solved with a single map and reduce operation. Composing MapReduce allows us to break down these intricate tasks into smaller, more manageable subproblems. By structuring MapReduce calculations effectively, we can harness the power of parallelism and distribute these subproblems across a cluster of machines, significantly speeding up data processing. This approach not only improves performance but also enhances fault tolerance, as the MapReduce framework can recover from node failures and ensure that the computation continues to progress.

Furthermore, composing MapReduce enables data engineers and scientists to design more sophisticated data transformations and analyses. It facilitates the chaining of multiple MapReduce stages, where the output of one stage becomes the input for the next. This flexibility is crucial for performing iterative algorithms, such as machine learning algorithms like k-means clustering or PageRank, and for handling various data formats and structures. By mastering the art of composing MapReduce, practitioners can unlock the full potential of this powerful framework, making it a fundamental skill for anyone working with large-scale data processing tasks in today's data-driven world.

It is required to store the output in a separate file, as shown in *Figure 3.5*. All of these files will perform an operation in the DFS system, which will be covered later in the book in detail. Components that are used in performing operations will be like: File system counter and MapReduce framework, that is, number of bytes read, number of bytes written, read operations, write operations, map input records, and so on. Map output records, input split bytes, split record, file input format counter, and so on, as shown in *Figure 3.6*:

```

17/08/01 13:53:38 INFO mapred.LocalJobRunner: reduce > reduce
17/08/01 13:53:38 INFO mapred.Task: Task 'attempt_local1815802876_0001_r_000000_0' done.
17/08/01 13:53:38 INFO mapred.LocalJobRunner: Finishing task: attempt_local1815802876_0001_r_000000_0
17/08/01 13:53:38 INFO mapred.LocalJobRunner: reduce task executor complete.
17/08/01 13:53:38 INFO mapreduce.Job: map 100% reduce 100%
17/08/01 13:53:38 INFO mapreduce.Job: Job job_local1815802876_0001 completed successfully
17/08/01 13:53:38 INFO mapreduce.Job: Counters: 38
  File System Counters
    FILE: Number of bytes read=7060
    FILE: Number of bytes written=508940
    FILE: Number of read operations=0
    FILE: Number of large read operations=0
    FILE: Number of write operations=0
    HDFS: Number of bytes read=112
    HDFS: Number of bytes written=64
    HDFS: Number of read operations=13
    HDFS: Number of large read operations=0
    HDFS: Number of write operations=4
  Map-Reduce Framework
    Map input records=4
    Map output records=4
    Map output bytes=72
    Map output materialized bytes=86
    Input split bytes=78
    Combine input records=0
    Combine output records=0
    Reduce input groups=4
    Reduce shuffle bytes=86
    Reduce input records=4
    Reduce output records=4
    Spilled Records=8
    Shuffled Maps =1
    Failed Shuffles=0
    Merged Map outputs=1
    GC time elapsed (ms)=91
    CPU time spent (ms)=0
    Physical memory (bytes) snapshot=0
    Virtual memory (bytes) snapshot=0
    Total committed heap usage (bytes)=345505792
  Shuffle Errors
    BAD_ID=0
    CONNECTION=0
    IO_ERROR=0
    WRONG_LENGTH=0
    WRONG_MAP=0
    WRONG_REDUCE=0
  File Input Format Counters
    Bytes Read=56
  File Output Format Counters
    Bytes Written=64

```

**Figure 3.6:** Composing of MapReduce

## Conclusion

The MapReduce framework has revolutionized the way we process and analyze vast amounts of data. It offers a scalable and fault-tolerant approach to distributed computing, making it an indispensable tool in the world of big data. By dividing tasks into two key phases—mapping and reducing—MapReduce simplifies complex data processing workflows, allowing for efficient parallel execution on clusters of machines. Although it has been widely adopted in the industry and academia, it is worth noting that newer

technologies and frameworks have emerged, such as Apache Spark, which offers improvements in terms of performance and ease of use. Nonetheless, the core concepts and principles of MapReduce remain influential in the field of distributed computing and continue to serve as a foundation for understanding and designing data processing systems at scale.

In the upcoming chapter, we will learn about the basics of Hadoop and implementation of all kinds of technicalities with Hadoop.

### **Join our book's Discord space**

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



# CHAPTER 4

## Basics of Hadoop

### Introduction

In today's scenario, the human body is surrounded with data. They are consuming the data as well as producing it in seconds. A paper published with name A survey of results on mobile phone datasets analysis in Springer stated that mobile data is producing a larger amount of data when compared to any other sources. Facebook, Instagram, X, WhatsApp, and so on are real examples of daily life. This exponential growth of data was the main reason to invent some new techniques for providing and retaining business for Google, Yahoo, Amazon, Microsoft, and so on, who are providing large business from customers that are part of that technology.

Google's GFS file system was one that started this technology to retain the file in chunks format. This provides an idea for retaining files in a distributed environment using commodity hardware and makes them workable.

Performing computation on massive volumes of information has been done before, typically during a distributed setting. What makes Hadoop distinctive is its simplified programming model that permits the user to quickly write and check distributed systems and its economical, automatic distribution (which will be discussed in the Hadoop file system in detail) of knowledge and work across machines and successively using the underlying similarity of the CPU cores.

### Structure

In this chapter, we will go through the following topics:

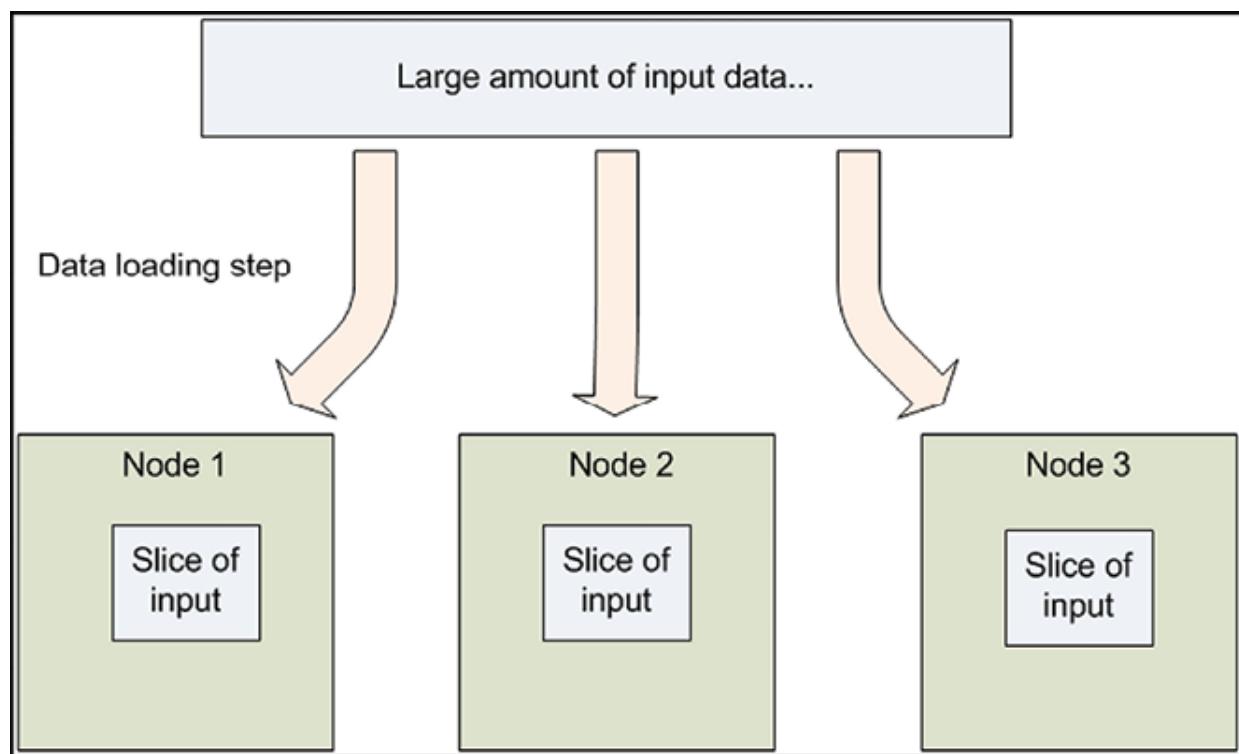
- Data distribution
- Data format
- Data analysis with Hadoop
- Scale-in and scale-out property
- Hadoop streaming
- Data streaming with different programming languages
- Hadoop file system
- Java Interface
- Data flow in Hadoop
- Hadoop I/O
- Compression of data with Hadoop
- Serialization
- Avro file-based data structure

## **Objectives**

The objective of this chapter is to address the challenges of processing and analyzing massive data efficiently. It achieves scalability, empowering businesses to handle large datasets. Fault tolerance ensures uninterrupted processing by redirecting tasks during node failures. Distributed computing enables parallel processing across nodes, optimizing resource utilization. Data locality minimizes network transfers, improving performance and keeping costs low because moving computing is cheaper than moving data. Hadoop's flexibility supports various processing models, enabling batch, interactive, and real-time processing. By focusing on scalability, fault tolerance, distributed computing, data locality, and flexibility, Hadoop revolutionizes data-intensive workloads. It empowers organizations to store, process, and analyze vast amounts of data, offering a solid foundation for big data processing in today's data-driven world.

## **Data distribution**

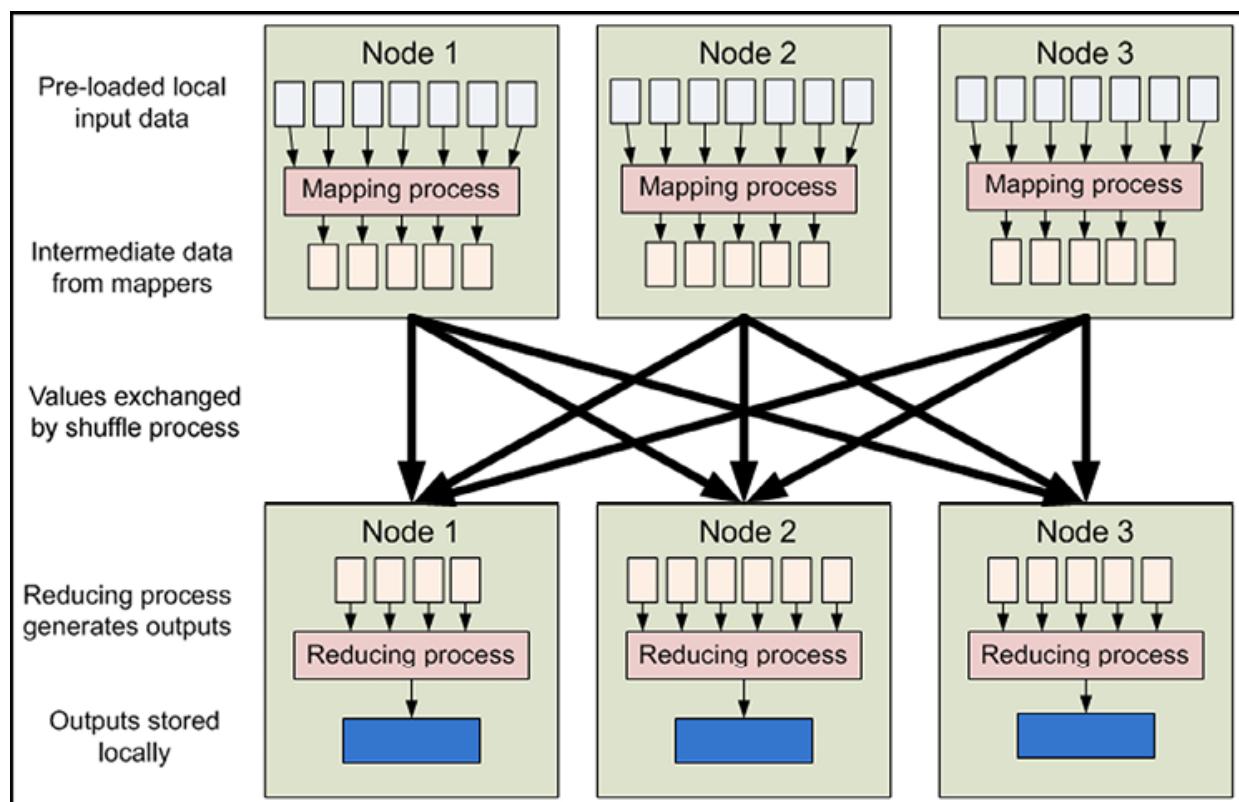
During a Hadoop cluster, the information is distributed to all or any of the nodes of the cluster because it is being loaded in. The **Hadoop Distributed File System (HDFS)** can split massive data files into chunks, as shown in *Figure 4.1*, which is managed by completely different nodes in the cluster. Additionally, every chunk is replicated across many machines, which is called the replication factor, so one machine failure does not result in any data being unavailable. A lively observance system then re-replicates the information in response to system failures, which might lead to partial storage. Although the file chunks are replicated and distributed across many machines, they form one namespace; therefore, their contents are universally accessible.



*Figure 4.1: Distributing data*

Hadoop fixes the processing with communication through the processor, as every record is processed from clusters individually by a task that is given externally by the programmer. However, this only seems like a significant limitation initially, and it later makes the total framework more reliable.

Hadoop will not simply run any program and distribute it across a cluster. Programs should be written to evolve into a selected programming model named **MapReduce**. In MapReduce, records are processed in isolation, as shown in *Figure 4.2*, by tasks referred to as Mappers. The output from the Mappers is then brought along into a second set of tasks referred to as Reducers, wherever results from totally different mappers are integrated along.



*Figure 4.2: Processing data among clusters*

## Data format

The file that is used for processing reads data line by line. MapReduce (processing tool) uses files for processing objectives and extracts valuable information from those files. Following is the example of data that has been taken from the National Climate Data Center:

0057

332130 # USAF weather station identifier  
99999 # WBAN weather station identifier  
19500101 # observation date  
0300 # observation time  
4  
+51317 # latitude (degrees x 1000)  
+028783 # longitude (degrees x 1000)  
FM-12  
+0171 # elevation (meters)  
99999  
V020  
320 # wind direction (degrees)  
1 # quality code  
0072  
00450 # sky ceiling height (meters)  
1 # quality code  
010000 # visibility distance (meters)  
1 # quality code  
-0128 # air temperature (degrees Celsius x 10)  
1 # quality code

```

-0139          # dew point temperature (degrees
Celsius x 10)

1            # quality code

10268         # atmospheric pressure (hectopascals
x 10)

1            # quality code

```

This particular input format uses data and its associated information. This can be perceived as a two-column format used for data storage, which may also include undesired data stored. Following are the input and output formats, as shown in [Table 4.1](#), that are supported in the MapReduce engine for processing of data:

	<b>TextInputFormat</b>	<b>Reads lines of text file</b>
<b>Input format</b>	<b>KeyValueInputFormat</b>	Parses lines into key value
	<b>SequenceFileInputFormat</b>	Hadoop's specific high performance binary format
<b>Output format</b>	<b>TextOutputFormat</b>	Default: Writes in key value format
	<b>SequenceFileOutputFormat</b>	Writes binary files
	<b>NullOutputFormat</b>	Disregards its input

**Table 4.1:** Format used in Hadoop

## Analyzing data with Hadoop

MapReduce is the programming paradigm for analyzing data with Hadoop. As all the data stored in clusters is distributed in the environment, mappers and reducers do their jobs of processing. For analysis purposes, we can take the raw data with a text input format, in which all the data is read line by line. The offset of the line will be treated as the key, and the rest of the line

will be the value of the data. *Figure 4.3* shows raw data that is related to the weather data. All data is not useful, and there is also data that are not in use for processing:

63891	20130101	5.102	-86.61	32.85	12.8	9.6	11.2	11.6	19.4	-9999.00	U	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-99.000
-99.000	-99.000	-99.000	-99.000	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	0.4	-9999.00	U	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-99.000
63891	20130102	5.102	-86.61	32.85	10.1	3.8	7.8	6.2	0.0	-9999.00	U	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-99.000
-99.000	-99.000	-99.000	-99.000	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	0.0	-9999.00	U	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-99.000
63891	20130103	5.102	-86.61	32.85	7.0	-2.2	2.4	2.9	0.0	-9999.00	U	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-99.000
-99.000	-99.000	-99.000	-99.000	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	0.0	-9999.00	U	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-99.000
63891	20130104	5.102	-86.61	32.85	11.8	-3.7	4.1	3.0	0.0	-9999.00	U	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-99.000
-99.000	-99.000	-99.000	-99.000	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	0.2	-9999.00	U	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-99.000
63891	20130105	5.102	-86.61	32.85	8.6	-1.1	3.8	3.7	0.2	-9999.00	U	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-99.000
-99.000	-99.000	-99.000	-99.000	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	0.0	-9999.00	U	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-99.000
63891	20130106	5.102	-86.61	32.85	10.5	1.9	6.2	6.1	0.0	-9999.00	U	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-99.000
-99.000	-99.000	-99.000	-99.000	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	0.0	-9999.00	U	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-99.000
63891	20130107	5.102	-86.61	32.85	13.4	-0.1	6.7	5.2	0.0	-9999.00	U	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-99.000
-99.000	-99.000	-99.000	-99.000	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	0.0	-9999.00	U	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-99.000
63891	20130108	5.102	-86.61	32.85	16.2	1.4	8.8	10.2	0.0	-9999.00	U	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-99.000
-99.000	-99.000	-99.000	-99.000	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	0.4	-9999.00	U	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-99.000
63891	20130109	5.102	-86.61	32.05	17.0	12.4	15.1	15.0	-99.000	-99.000	-99.000	-99.000	-99.000	-99.000	-99.000	-99.000	-99.000	-99.000	
-99.000	-99.000	-99.000	-99.000	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-99.000	-99.000	-99.000	-99.000	-99.000	-99.000	-99.000	-99.000	-99.000	-99.000	

*Figure 4.3: Raw data*

So, the meaningful data, which is essential for processing and analysis, typically undergoes treatment using analysis tools. Upon manual analysis, it is determined that columns 2, 3, 6, 7, 8, and 9 can be selected for processing purposes. However, further analysis reveals that only columns 2, 6, and 8 are ultimately considered, which need to be trimmed. Let us evaluate the performance of the program using MapReduce functionality, specifically on the designated rows and columns:

```
public class WeatherData {

    public static class MaxTemperatureMapper
        extends MapReduceBase implements
            Mapper<LongWritable, Text, Text, Text>
    {
        @Override
        public void map(LongWritable arg0, Text
Value,
```

```
        OutputCollector<Text, Text> output,  
Reporter arg3)
```

```
throws IOException {
```

```
    String line = Value.toString();
```

```
    // Example of Input
```

//	Date	Max	Min
----	------	-----	-----

	// 25380 20130101	2.514	-135.69
--	-------------------	-------	---------

58.43	8.3	1.1	4.7	4.9	5.6
-------	-----	-----	-----	-----	-----

0.01 C	1.0	-0.1	0.4	97.3	36.0
--------	-----	------	-----	------	------

69.4	-99.000	-99.000	-99.000	-99.000	-99.000
------	---------	---------	---------	---------	---------

-9999.0	-9999.0	-9999.0	-9999.0	-9999.0	-9999.0
---------	---------	---------	---------	---------	---------

```
    String date = line.substring(6, 14);
```

```
    float temp_Max =  
Float.parseFloat(line.substring(39, 45).trim());
```

```
    float temp_Min =  
Float.parseFloat(line.substring(47, 53).trim());
```

```
    if (temp_Max > 40.0) {
```

```
        // Hot day
```

```
        output.collect(new Text("Hot Day "
+ date),
                new
Text(String.valueOf(temp_Max)));
}

if (temp_Min < 10) {
    // Cold day
    output.collect(new Text("Cold Day "
+ date),
                new
Text(String.valueOf(temp_Min)));
}

}

public static class MaxTemperatureReducer
extends MapReduceBase implements
    Reducer<Text, Text, Text, Text> {

    @Override
    public void reduce(Text Key, Iterator<Text>
Values,
```

```
        OutputCollector<Text, Text> output,
Reporter arg3)

        throws IOException {

    // Find Max temp yourself ?

    String temperature =
values.next().toString();

    output.collect(Key, new
Text(temperature));

}

}
```

```
public static void main(String[] args) throws
Exception {

    JobConf conf = new
JobConf(WeatherData.class);

    conf.setJobName("temp");

    // Note:- As Mapper's output types are not
default so we have to define

    // the

    // following properties.
```

```
        conf.setMapOutputKeyClass(Text.class);
        conf.setMapOutputValueClass(Text.class);

conf.setMapperClass(MaxTemperatureMapper.class);

conf.setReducerClass(MaxTemperatureReducer.class);

        conf.setInputFormat(TextInputFormat.class);

conf.setOutputFormat(TextOutputFormat.class);

        FileInputFormat.setInputPaths(conf, new
Path(args[0]));

        FileOutputFormat.setOutputPath(conf, new
Path(args[1]));

        JobClient.runJob(conf);

    }

}
```

**JobConf** object is specific to take control over any program that runs the MapReduce functionality. It is defined in the driver class, which covers the details about the mapper class, reducer class, and its format. Following is the screenshot of the program processing with MapReduce:

```

17/08/13 22:50:38 INFO mapred.Task: Task attempt_local404383680_0001_r_000000_0 is allowed to commit now
17/08/13 22:50:38 INFO output.FileOutputCommitter: Saved output of task 'attempt_local404383680_0001_r_000000_0' to hdfs://localhost:54310/weath
er1/_temporary/0/task_local404383680_0001_r_000000
17/08/13 22:50:38 INFO mapred.LocalJobRunner: reduce > reduce
17/08/13 22:50:38 INFO mapred.Task: Task 'attempt_local404383680_0001_r_000000_0' done.
17/08/13 22:50:38 INFO mapred.LocalJobRunner: Finishing task: attempt_local404383680_0001_r_000000_0
17/08/13 22:50:38 INFO mapred.LocalJobRunner: reduce task executor complete.
17/08/13 22:50:39 INFO Mapreduce.Job: map 100% reduce 100%
17/08/13 22:50:39 INFO Mapreduce.Job: Job job_local404383680_0001 completed successfully
17/08/13 22:50:39 INFO mapreduce.Job: Counters: 38
  File System Counters
    FILE: Number of bytes read=127826
    FILE: Number of bytes written=687403
    FILE: Number of read operations=0
    FILE: Number of large read operations=0
    FILE: Number of write operations=0
    HDFS: Number of bytes read=72380
    HDFS: Number of bytes written=3886
    HDFS: Number of read operations=13
    HDFS: Number of large read operations=0
    HDFS: Number of write operations=4
  Map-Reduce Framework
    Map input records=167
    Map output records=4648
    Map output bytes=51157
    Map output materialized bytes=60459
    Input split bytes=86
    Combine input records=0
    Combine output records=0
    Reduce input groups=468
    Reduce shuffle bytes=60459
    Reduce input records=4648
    Reduce output records=468
    Spilled Records=9296
    Shuffled Maps =1
    Failed Shuffles=0
    Merged Map outputs=1
    GC time elapsed (ms)=106
    CPU time spent (ns)=0
    Physical memory (bytes) snapshot=0
    Virtual memory (bytes) snapshot=0
    Total committed heap usage (bytes)=340651520

```

*Figure 4.4: Program processing*

Following is the screenshot of sample data:

```

0067011990999991950051507004...9999999N9+00001+99999999999...
0043011990999991950051512004...9999999N9+00221+99999999999...
0043011990999991950051518004...9999999N9-00111+99999999999...
0043012650999991949032412004...0500001N9+01111+99999999999...
0043012650999991949032418004...0500001N9+00781+99999999999...

```

*Figure 4.5: Sample data*

Following is the screenshot of the after-processing of data:

```

(0, 0067011990999991950051507004...9999999N9+00001+99999999999...)
(106, 0043011990999991950051512004...9999999N9+00221+99999999999...)
(212, 0043011990999991950051518004...9999999N9-00111+99999999999...)
(318, 0043012650999991949032412004...0500001N9+01111+99999999999...)
(424, 0043012650999991949032418004...0500001N9+00781+99999999999...)

```

*Figure 4.6: After-processing of data*

## Scale-in versus scale-out

Since there is a structure famous for processing and maintaining data with client-server architecture, it consists of a node that governs all other nodes that are called clients. All clients need to report to a single system, so all the data and programs need to be stored on the server side. Whenever there is a need to process something, it needs to get permission from the server because it is the one that stores data. In the future, there will be a need to upgrade server capabilities to handle larger data. All data will need to be moved from the client side to the server side when there is a requirement to process it. Hence, scaling on the server side is called the scale-in (scale-up) property. This property states that capacity can be enhanced in terms of data handling, processing, and so on, which means enhancing HDD storage and RAM capacity.

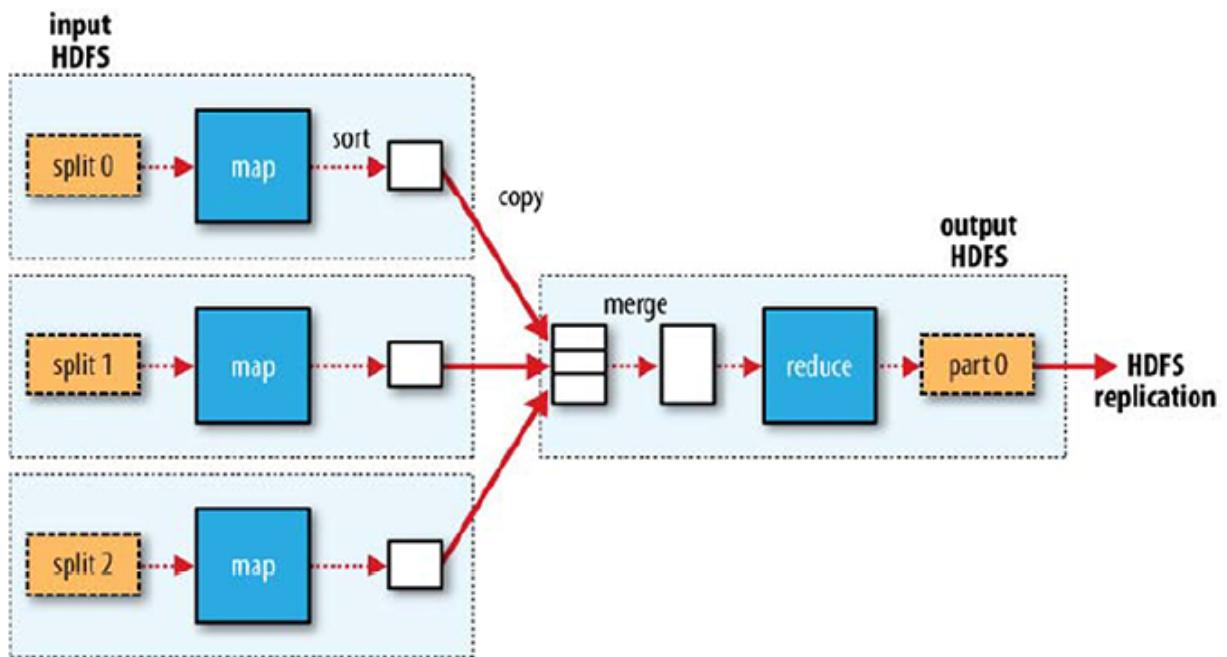
On the other hand, when data needs to be stored on the client side in the form of a distributed system, it is called master-slave architecture. This is useful when a large amount of data needs to be processed. In Hadoop terminology, task trackers are said to be clients and job trackers are said to be server-side. Their functionality differs from traditional functions. While task trackers will be responsible for storing data in chunks form, that is, in a distributed manner, job trackers will be responsible for putting program of MapReduce to different chunks in different systems. Job trackers will control all the data that is stored in task trackers, and this system is called HDFS. In HDFS, when there is a requirement to increase the capacity of storage, nodes can be added to it. This phenomenon is said to be scale-out property. For example, if there is 200 TB of data expected to be used in the present time but after five years of business, there is a requirement of 100 TB more data, in this scenario, number of nodes worth 100 TB of data can be added easily without any interference.

When data is divided into multiple splits and distributed across different machines, the individual processing time for each split is typically shorter compared to processing the entire input as a whole. This is because when the splits are processed in a parallel and distributed manner, the workload is effectively distributed among the machines. Consequently, for optimal load balancing, it is advantageous for the splits to be small and optimal size. This is because faster machines can process a greater number of smaller splits within the same time frame, allowing for a more balanced utilization of

resources. Even when the machines are identical, the presence of unsuccessful processes or concurrent jobs running on the machines makes load balancing crucial. Moreover, as the splits become finer-grained, the quality of load balancing improves, leading to more efficient processing of data.

## Number of reducers used

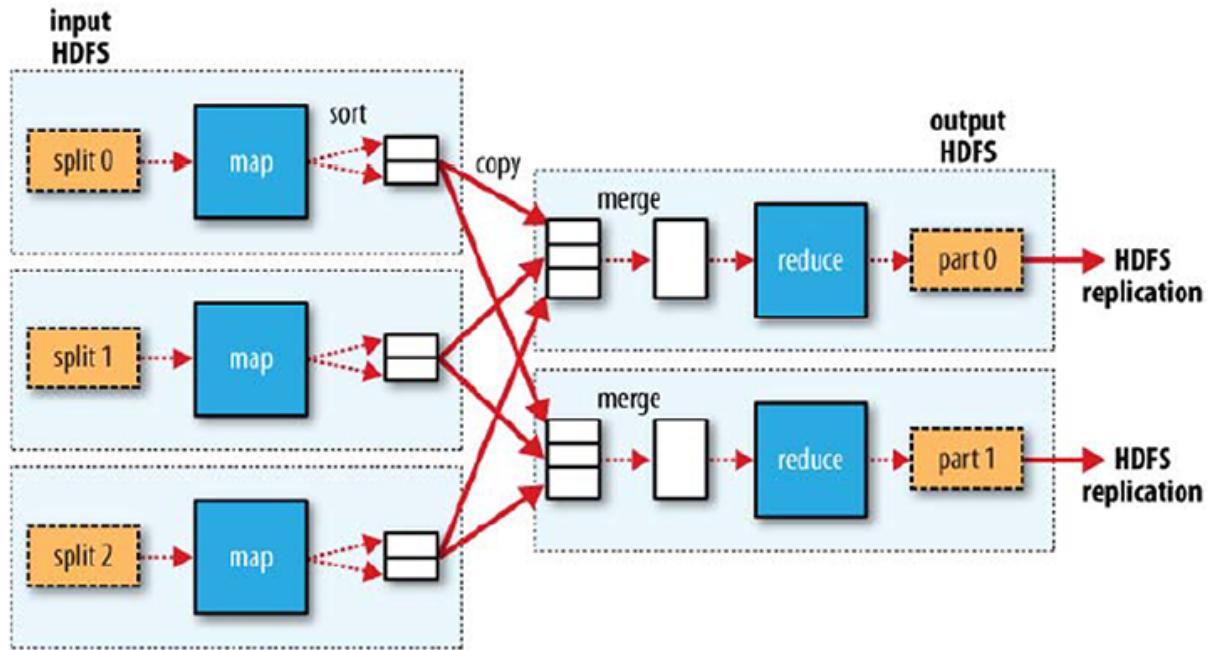
Hadoop will do its best to run the map task on a node wherever the input data resides in HDFS. This is often known as the information locality improvement. [Figure 4.7](#) shows the processing with a single reducer:



*MapReduce data flow with a single reduce task*

[Figure 4.7: MapReduce job splits with a single reducer](#)

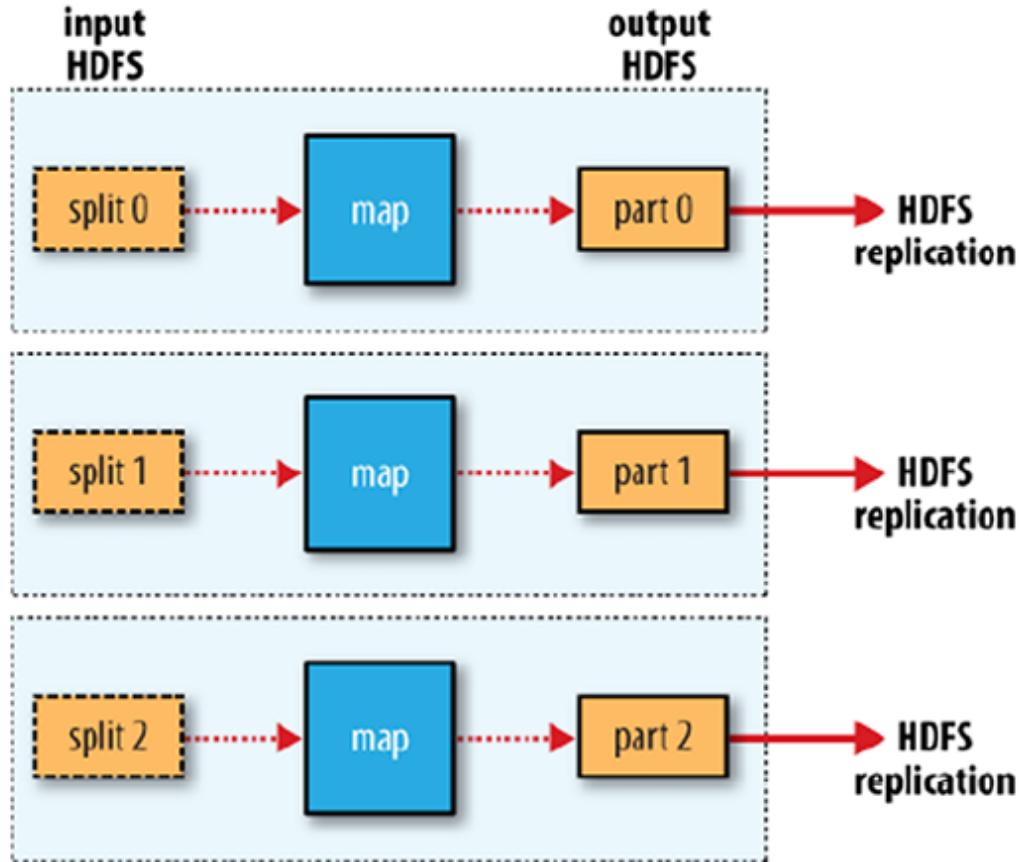
The total number of reduce tasks does not depend on the size of input used for processing purposes. It can be specified independently. Users can also increase the number of reducers depending upon requirement. In case of multiple inputs, as shown in [Figure 4.8](#), all map tasks are mapped with the reducer task and provide inputs to its reducer:



*MapReduce data flow with multiple reduce tasks*

**Figure 4.8:** MapReduce job splits with multiple reducer

There can be case if program using zero reducers. In that case there is no use of shuffle operations as shown in [Figure 4.9](#):



*MapReduce data flow with no reduce tasks*

**Figure 4.9:** MapReduce job splits with zero reducer

### Driver class with no reducer

The following program shows the driver class in which no reducer is used; it states a scenario of distributed cache. There is a need to create a name for the job that sets my **MyExample** class. **Job** is the object that is used to call methods. Number of reducers used in the task will be 0:

```
public static void main(String[] args)
    throws IOException, ClassNotFoundException,
    InterruptedException {
    Job job = new Job();
```

```
job.setJarByClass(MyExample.class);
job.setJobName("DCTest");
job.setNumReduceTasks(0);

try{
    DistributedCache.addCacheFile(new
URI("/abc.dat"), job.getConfiguration());
} catch(Exception e){
    System.out.println(e);
}

job.setMapperClass(MyMapper.class);

job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(Text.class);
FileInputFormat.addInputPath(job, new
Path(args[0]));

FileOutputFormat.setOutputPath(job, new
Path(args[1]));

job.waitForCompletion(true);

}
```

## Hadoop streaming

Hadoop provides different APIs that are useful for enabling users to write programs in any language other than Java. Unix can be defined as a standard stream used as an interface between Hadoop and a program of users. So, any language can read standard inputs and provide standard outputs of MapReduce program. Streaming of text data requires a line-oriented view of data. The output of map data is passed to the reducer as input. This creates key-value pair of data with a tab-delimited line.

Suppose here is the example of finding maximum temp in year streaming; it can be written in Ruby and discussed as follows.

### Streaming in Ruby

To stream Hadoop data in Ruby, you can use the Hadoop Streaming API and write code that interacts with it:

```
STDIN.each_line do |line|
  val = line
  year, temp, q = val[15,4], val[87,5], val[92,1]
  puts "#{year}\t#{temp}" if (temp != "+9999" && q =~ /[01459]/)
end
```

Since the script just operates on standard input and output, it is trivial to test the script without using Hadoop, simply using Unix pipes:

```
% cat input/ncdc/sample.txt |
ch02/src/main/ruby/max_temperature_map.rb
1950 +0000
1950 +0022
1950 -0011
```

```
1949 +0111
```

```
1949 +0078
```

The reduce function is shown as follows:

```
last_key, max_val = nil, 0
STDIN.each_line do |line|
key, val = line.split("\t")
if last_key && last_key != key
puts "#{last_key}\t#{max_val}"
last_key, max_val = key, val.to_i
else
last_key, max_val = key, [max_val, val.to_i].max
end
end
puts "#{last_key}\t#{max_val}" if last_key
```

We can now simulate the whole MapReduce pipeline with a Unix pipeline, we get the following result:

```
% cat input/ncdc/sample.txt |
ch02/src/main/ruby/max_temperature_map.rb | \
sort | ch02/src/main/ruby/max_temperature_reduce.rb
1949 111
1950 22
```

## Streaming in Python

To stream Hadoop data in Python, you can use the Hadoop Streaming API and write code that interacts with it as follows:

```
import re

import sys


for line in sys.stdin:
    val = line.strip()
    (year, temp, q) = (val[14:18], val[25:30],
val[31:32])
    if (temp != "9999" and re.match("[01459]", q)):
        print "%s\t%s" % (year, temp)

import sys
(last_key, max_val) = (None, -sys.maxint)
for line in sys.stdin:
    (key, val) = line.strip().split("\t")
    if last_key and last_key != key:
        print "%s\t%s" % (last_key, max_val)
    (last_key, max_val) = (key, int(val))
    else:
        (last_key, max_val) = (key, max(max_val,
int(val)))
```

```
if last_key:  
    print "%s\t%s" % (last_key, max_val)
```

## Streaming in Java

To stream Hadoop data in Java, you can use the Hadoop Streaming API and write code using the Java programming language:

```
public static class Map extends  
Mapper<LongWritable, Text, Text, IntWritable> {  
  
    Text k= new Text();  
  
    public void map(LongWritable key, Text  
value, Context context)  
        throws IOException, InterruptedException {  
            String line = value.toString();  
            StringTokenizer tokenizer = new  
StringTokenizer(line, " ");  
            while (tokenizer.hasMoreTokens())  
{  
  
                String year=  
tokenizer.nextToken();  
                k.set(year);
```

```
        String temp=
tokenizer.nextToken().trim();

        int v =
Integer.parseInt(temp);

        context.write(k,new
IntWritable(v));

    }

}

public static class Reduce extends
Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key,
Iterable<IntWritable> values, Context context)
throws IOException, InterruptedException {

    int maxtemp=0;

    for(IntWritable it : values) {

        int temperature= it.get();

        if(maxtemp<temperature)

        {

            maxtemp =temperature;

        }

    }

}
```

```
        }

        context.write(key, new
IntWritable(maxtemp));

    }

}
```

## Hadoop pipes

Unlike streaming that is used only in standard inputs and outputs for communication with map and reduce codes, pipes is the name of interface of C++. Pipes is a socket that is used as a medium of communication with methods of map and reduce.

The following example shows the source code for the map and reduce functions in C++:

```
class MaxTemperatureMapper : public
HadoopPipes::Mapper

{

public:
MaxTemperatureMapper(HadoopPipes::TaskContext&
context) { }

void map(HadoopPipes::MapContext& context)

{

    // statement to convert the input data into
string

    // statement to obtain year and temp using the
substring method
```

```
        // statement to place the year and temp into a
set
    }

};

class MapTemperatureReducer : public
HadoopPipes::Reducer
{
public:
    MapTemperatureReducer(HadoopPipes::TaskContext&
context) {}

    void reduce(HadoopPipes::ReduceContext& context)
    {
        // statement to find the maximum temperature of
        // an each year

        // statement to put the max. temp and its year in
        // a set
    }
};

int main(int argc, char *argv[])
{
    return HadoopPipes::runTask( HadoopPipes::
TemplateFactory < MaxTemperatureMapper,
```

```
MapTemperatureReducer>() );  
}
```

The map and reduce functions are defined by extending the mapper and reducer classes defined in the **HadoopPipes** namespace and providing implementations of the **map()** and **reduce()** methods in each case.

These methods take a context object (of type **MapContext** or **ReduceContext**), which provides the means for reading input and writing output, as well as accessing job configuration information via the **JobConf** class. The **main()** method is the application entry point. It calls **HadoopPipes::runTask**, which connects to the Java parent process and marshals data to and from the Mapper or Reducer. The **runTask()** method passes a factory object so that it can create instances of the Mapper or Reducer. The Java parent controls which one it creates over the socket connection. There is an overloaded templates factory method for setting a combiner, partitioner, record reader, or record writer.

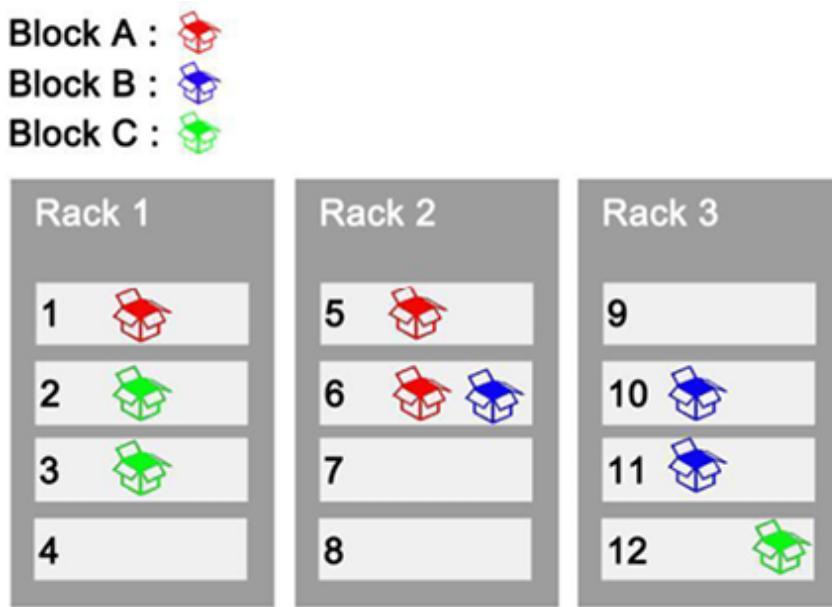
Pipes do not run in standalone (local) mode because they rely on Hadoop's distributed cache mechanism, which works only when HDFS is running.

## Design of HDFS

The data used for storage and processing purposes is stored in a distributed file system. It is divided into chunks to facilitate storage in different locations, which is why it is referred to as distributed storage. Specifically in the context of Hadoop, this distributed storage is known as **Hadoop Distributed File System (HDFS)**. By leveraging commodity hardware and offering fault tolerance, scalability, high throughput, data locality, a simple coherency model, and replication/recovery mechanisms, HDFS is well-suited for streaming, storing, and processing large-scale data in Hadoop environments.

## Very large

Any size of file can be easily fitted into the file system of Hadoop. That means megabytes, petabytes, and gigabytes of data can be used with the Hadoop cluster. This data storage is not costly because it uses commodity hardware for storage purposes. Every data uses chunks formed in different locations, as shown in *Figure 4.10*:



*Figure 4.10: Fault-tolerance*

## Streaming data access

HDFS is based on a write-once and read-many-times system, and this adds efficiency in processing data. Various sources are used in storing the data in storage over time for processing purposes. Every processing requires a large amount of data for getting the output.

## Commodity hardware

Hadoop does not require any expensive hardware for storing data. It uses general hardware that is commonly used for storage. It is designed to run clusters, so there is a chance of high node failure of the cluster. So HDFS uses fault tolerance by having a default of three copies (this default value can be adjusted by using the replication factor parameter) of data in the storage part in different positions.

## **Low-latency data access**

HDFS is useful for delivering high throughput data while it is not fit for low-latency data in a tens of milliseconds range.

## **Lots of small files**

The Namenode in Hadoop is responsible for storing metadata about the data stored in the HDFS rather than storing the data directly. The amount of storage available is determined by the amount of memory used by the Namenode. Generally, each file, directory, and block in HDFS takes up around 150 bytes of memory. For instance, if you had one million files, with each file taking up one block, you would need at least 300 MB of memory. It is feasible to store millions of files in HDFS, but the current hardware may not be capable of that capacity.

## **Arbitrary file modifications**

Files in HDFS are written as single writers, so writers are made at the end of the file system. There is no support for multiple file modifications required.

## **HDFS concept**

HDFS is related to the file system, so the following terminologies related to HDFS are useful for the understanding of the file system.

### **Blocks**

Blocks reflect the minimum data that can be read or written. The file system reads by reading data per block that are the multiples of block size. Blocks are defaulted in the memory until you customize it to a certain amount. These blocks are used for writing and reading the data. HDFS has 64MB of default data size. It can be understood that if there are a large number of files, then they will be divided into blocks of 64MB each. This is called a chunk of files. HDFS blocks can be compared with a disk block, but unlike a disk block, it is large in size.

If the seek time is 10 ms and the transfer rate is 100 MB/s, to ensure that the seek time is 1% of the transfer rate, the block size should be adjusted to 100 MB.

In the HDFS, the default block size is 64 MB. However, if you need to use 100 MB of data and want to maintain the seek time at 1% of the transfer rate, you should consider adjusting the block size to match the desired size of 100 MB. This adjustment can help optimize data access and improve performance within the HDFS block system.

There will then be two blocks of 64 MB and 36 MB in size, so it reflects no wastage of space. It is also possible to have a large file that is equal to the disk size, but that does not mean HDFS requires a large size of blocks and disk; at that stage, the mode cluster comes into concept. Large clusters have lots of space for storing files. Each file is stored in chunks form, and it can be distributed among several nodes. Hence, there is no issue with having any size of the file and any size of the block. Dealing with blocks rather than files is the simplest way, and this simplicity comes because of distributed behavior. The system also maintains metadata regarding the storage over clusters. These blocks are fit with replications and availability. Replication is a matter of fault tolerance. To avoid failure, any block can be stored in different locations. In Hadoop, **fsck** command is used for checking the behavior of the blocks over the network.

## Namenodes and DataNodes

In HDFS, the cluster that is the master–slave environment, there are two kinds of node functions. Namenode (act as master) and DataNode (act as slave). This scenario differs from client-server architecture. Here, all the data is stored in DataNode, while Namenode will store all metadata information as namespace images and edit logs. Namenode knows about all information about individual nodes, that is, location, replication factor, rack information block id, and so on. It also knows about the file on which the operation is to be performed.

Clients access the filesystem from a user perspective by establishing communication between Namenode and DataNode. The client uses a

POSIX-like filesystem interface, which means there is no need for the user to know about the architecture. DataNode is used for storage perspective, and all the data stored in it is in chunks form. When there is a need to solve mining, then programs need to travel to DataNode; not traditionally, data will travel to the master side. The DataNodes constantly signal to the server about its status.

The Namenode is a critical component of the Hadoop system, as it retains information about the storage of all data stored in the DataNode. Another important component is the secondary Namenode, which serves as a backup for the Namenode. Although it cannot replace the Namenode, the secondary Namenode continuously updates itself to ensure that data remains safe in the event of a failure of the Namenode. It runs on a separate machine and prevents the log files of the Namenode from becoming too large by keeping them separate and recording them in a merged namespace image file. This file can be used to restore the system in the event of a Namenode failure. However, in the event of a total failure of the Namenode, the usual course of action is to copy the Namenode's metadata files from the NFS to the Secondary Namenode and run it as the new primary.

## **HDFS group**

Namenode keeps track of every file that is used for analysis purposes. Systems have a cluster of systems, and each cluster retains a bunch of files. These files do not get stored in a single location. It stores it in a chunk form, and these chunks are stored in a different location; finally, these locations and other file attributes like replication factor, permissions, ownership, timestamp, and so on. are stored as metadata in Namenode.

This system has a namespace volume for storing Namenode information and a block list of storing block-pool. These namespaces are independent of each other and do not affect each other's tasks. To get the access of the HDFS cluster, clients use client-side tables to map file paths to Namenodes. This is managed by using the View File System,

**viewfs://myhadoopclusterX/foo/bar**

## **All-time availability**

Although replicating Namenode data on multiple filesystems and using a secondary Namenode to make checkpoints protects against data loss, it does not ensure the high availability of the filesystem. In the event of a Namenode failure, all clients, including MapReduce jobs, would be unable to perform any read, write, or list file operations as the Namenode is the sole repository of metadata and file-to-block mapping. This means that the entire Hadoop system would be unavailable until a replacement Namenode is brought online.

To overcome a failed Namenode scenario, an administrator can start a new primary Namenode with one of the filesystem metadata replicas and configure DataNodes and clients to use this new Namenode. However, the new Namenode cannot serve requests until it has loaded its namespace image into memory, replayed its edit log, and received enough block reports from the DataNodes to depart safe mode. This long recovery time may also be a disadvantage for routine maintenance. In fact, since sudden failure of the Namenode is rare, planned maintenance periods are more necessary in practice.

To improve Namenode availability, the following steps should be taken:

1. The Namenodes should use highly available shared storage to share the edit log. This may require an NFS filer, but future releases may provide more options, such as a BookKeeper-based system built on ZooKeeper.
2. DataNodes should send block reports to each Namenode and secondary namenode since the block mappings are stored in a Namenode's memory and not on disk.
3. Clients should be configured to handle Namenode failover using a mechanism that is clear to users.

## **Hadoop files system**

The Hadoop ecosystem primarily revolves around the Hadoop Distributed File System (HDFS), which is the primary storage system. However, there are other storage systems and file system-like interfaces that are compatible with Hadoop.

Following is *Table 4.2* for the file system:

<b>File system</b>	<b>Type</b>	<b>Java implementation</b>	<b>Description</b>
Local	file	<code>fs.LocalFileSystem</code>	A filesystem for connected disk side checks. RawLocalFile for a local file: no checksums.
HDFS	hdfs	<code>hdfs.DistributedFileSystem</code>	Hadoop's filesystem. designed to be efficient in with MapReduce.
HFTP	hftp	<code>hdfs.HftpFileSystem</code>	A filesystem read-only access over HTTP. (name, HFTP connection will be lost)
HSFTP	hsftp	<code>hdfs.HsftpFileSystem</code>	A filesystem read-only access over HTTPS.
WebHDFS	webhdfs	<code>hdfs.web.WebHdfsFileSystem</code>	A filesystem for secure read-write HDFS over WebHDFS. WebHDFS is its replacement for HSFTP.

<b>File system</b>	<b>Type</b>	<b>Java implementation</b>	<b>Description</b>
HAR	har	<code>fs.HarFileSystem</code>	A filesystem another files archiving file Archives are used for archiv HDFS to r Namenode's usage
KFS (Cloud-Store)	kfs	<code>fs.kfs.KosmosFileSystem</code>	CloudStore Kosmos filesy distributed file HDFS or Goo written in C++.
FTP	ftp	<code>fs.ftp.FTPFileSystem</code>	A filesystem b FTP server.
Distributed RAID	hdfs	<code>hdfs.DistributedRaidFileSystem</code>	A <i>RAID</i> versio designed for storage. For HDFS, a (smal file is creat allows the replication to from three to reduces disk us to 30%, while probability of the same. Distrib requires that RaidNode dae cluster.

**Table 4.2:** Hadoop file system

## Java interface

Hadoop is written in Java, so Java is the medium for processing any data in the cluster. The file system in Hadoop uses Java as an interface. There are also other interfaces by which Hadoop is compatible.

## HTTP

HTTP is the way to access data directly or by using proxy servers. In the first case, directory listings are served by the Namenode's embedded net server (which runs on port **50070**) formatted in XML or JSON, whereas file data is streamed from DataNodes by their net servers (running on port **50075**). The original direct hypertext transfer protocol interface (HFTP and HSFTP) was read-only, whereas the new WebHDFS implementation supports all filesystem operations, as well as Kerberos authentication. WebHDFS should be enabled by setting `dfs.webhdfs.enabled` to true for you to be able to use WebHDFS URIs. The second method of accessing HDFS over hypertext transfer protocol depends on one or more standalone proxy servers. All traffic to the cluster passes through the proxy.

This permits stricter firewall and bandwidth-limiting policies to be put in place. It is common to use a proxy for transfers between Hadoop clusters situated in different data centers. The original HDFS proxy (in **src/contrib/hdfsproxy**) was read-only and will be accessed by clients exploiting the HSFTP FileSystem implementation (hsftp URIs). From release 0.23, there is a new proxy called HttpFS that has browse and write capabilities and exposes an equivalent hypertext transfer protocol interface as WebHDFS; therefore, clients will access either using WebHDFS URIs. The hypertext transfer protocol REST API that WebHDFS exposes is formally outlined in a specification, so it is possible that over time, clients in languages apart from Java will be written that use it directly.

## APIs in C language

The **libhdfs** is the API in C that is provided by Hadoop, and it provides functionality to facilitate the Java file system. It functions with **Java native interface (JNI)** to call the filesystem. The C API is very similar to the Java one, but it typically lags the Java one, so newer features may not be

supported. You can find the generated documentation for the C API in the **libhdfs/docs/api** directory of the Hadoop distribution.

## Filesystem in Userspace

**Filesystem in Userspace (FUSE)** enables the integration of a Unix file system implementation within another system, making it easier for Hadoop to contribute to the file system. To access the file system from programming languages using POSIX libraries, Fuse-DFS is implemented in C. It uses the **libhdfs** interface to connect with HDFS. The storage component of the Fuse-DFS system can be examined in the **src/contrib/fuse-dfs** directory. This directory provides insight into the storage aspect of the Fuse-DFS system within the Hadoop ecosystem.

FUSE is indeed a mechanism that allows a Unix-like operating system to implement its own file system. It enables users to create custom file systems without modifying the kernel. FUSE provides an interface between the kernel and user space, allowing user-level programs to handle file system operations.

In the context of Hadoop, FUSE can be used to facilitate the integration of HDFS with the Unix file system. By implementing FUSE for HDFS, you can interact with HDFS using familiar Unix commands such as ls and cat.

When using FUSE with HDFS, the POSIX libraries provide access to the file system from programming languages, enabling developers to interact with HDFS programmatically. The **libhdfs** library is commonly used as the interface between the FUSE-DFS implementation and HDFS.

In Hadoop, the **src/contrib/fuse-dfs** directory traditionally contains the source code and related files for the FUSE-DFS implementation in C. FUSE-DFS is a component that enables you to mount **Hadoop Distributed File System (HDFS)** as a local file system using FUSE. This allows you to interact with HDFS using standard Unix-like commands.

By leveraging FUSE and the FUSE-DFS implementation, you can enhance the interoperability between HDFS and Unix-like systems, making it easier

to work with HDFS using familiar Unix file system commands and programming interfaces.

## Reading data using the Java interface

Following is the sample representation to read data using Hadoop URL in Java programming language:

```
InputStream in = null;  
try {  
    in = new URL("hdfs://host/path").openStream();  
    // process in  
} finally {  
    IOUtils.closeStream(in);  
}
```

This block executes in a static block when the class is loaded in memory. It executes once per JVM. This can be done by calling the **setURLStreamHandlerFactory** method on the URL with an instance of **FsUrlStreamHandlerFactory**.

## Reading data using Java interface (FileSystem API)

There are several static methods that are used to read file systems using the Java interface:

```
public static FileSystem get(Configuration conf)  
throws IOException  
  
public static FileSystem get(URI uri, Configuration  
conf) throws IOException  
  
public static FileSystem get(URI uri, Configuration  
conf, String user) throws IOException
```

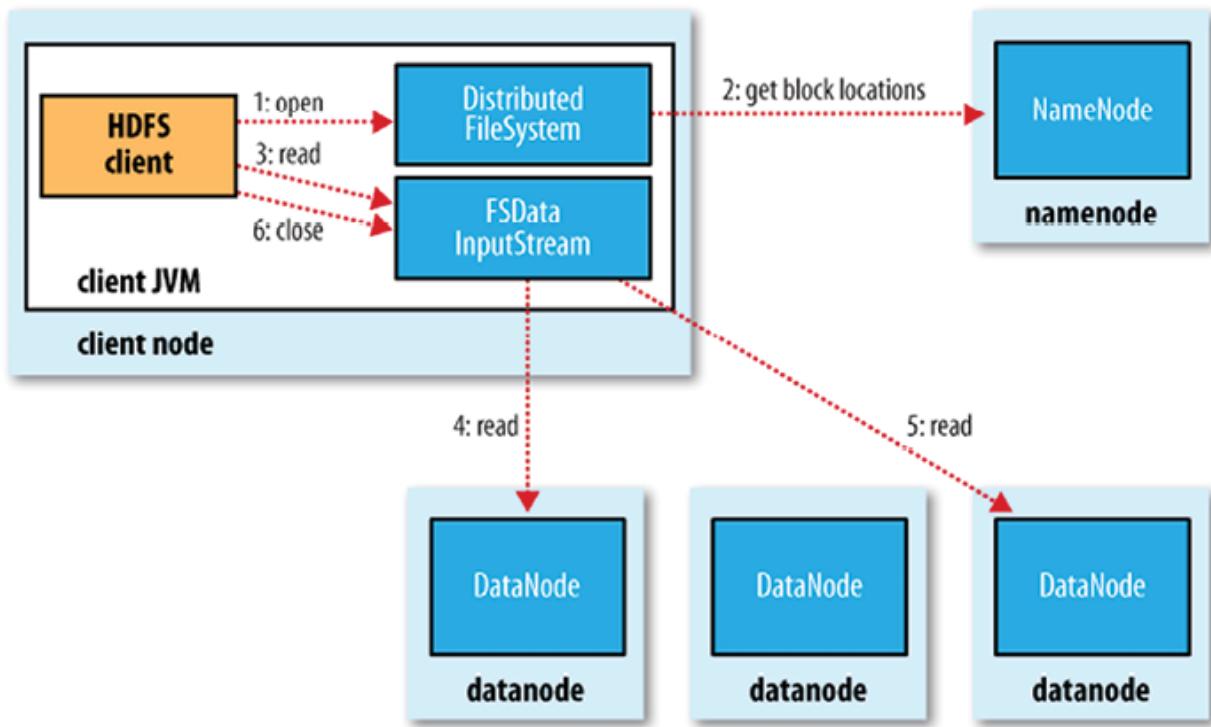
Configuration files can be set using **conf/core-site.xml**.

## Data flow

Data can flow in two ways, either from file to system or system to file, which is called File read and File write.

## File read

*Figure 4.11* shows the anatomy of the file read from HDFS to the client:



*Figure 4.11: Anatomy of file read*

Functions involved in file read are explained as follows:

- **Open()** is used to open the file to read for reading purposes. The file is in DataNode and is stored in a distributed environment for fault tolerance purposes, as discussed in the previous section.
- The client can retrieve the location of a file from the Namenode, which stores metadata information. The client can also act as a node and read data from its own local filesystem. The

**DistributedFileSystem** returns an **FSDataInputStream** to the client, which wraps a **DFSInputStream** managing the DataNode and Namenode I/O.

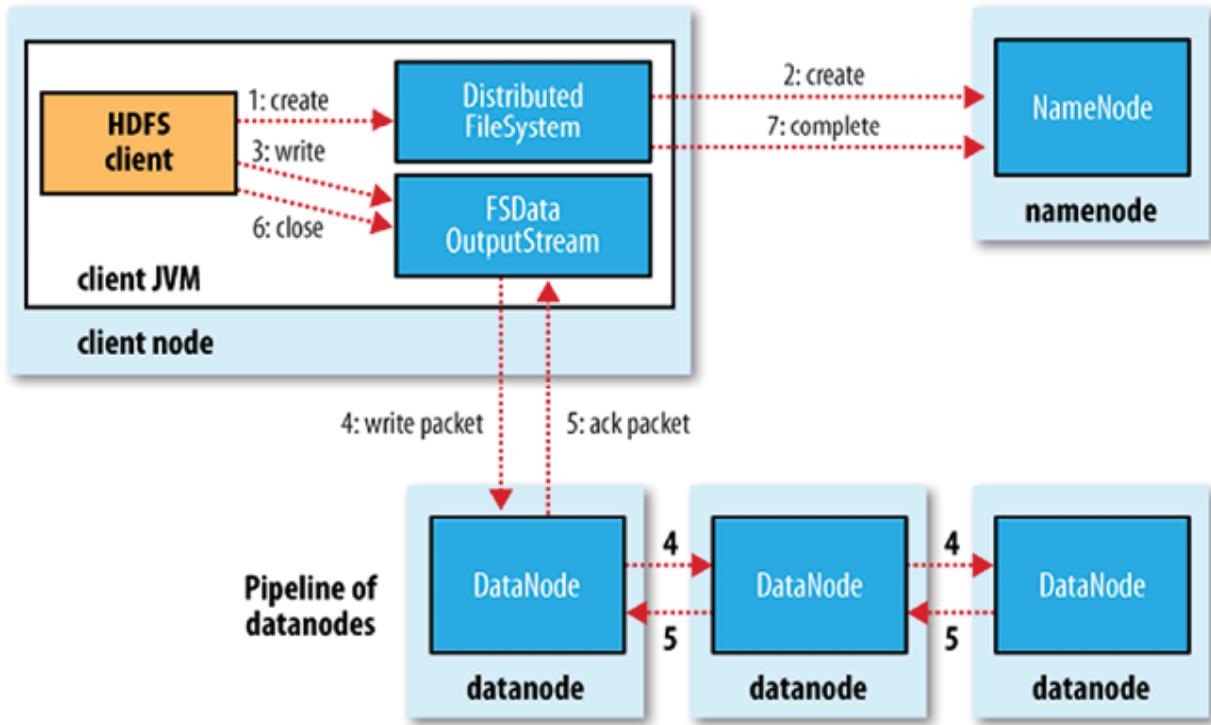
- The **FSDataInputStream** calls the **read()** function and retrieves the DataNode addresses for the first few blocks in the file. It then connects to the first DataNode for the first block.
- The data is streamed from the DataNode back to the client, which repeatedly calls **read()** on the stream.
- When the end of the block is reached, the **DFSInputStream** closes the connection to the DataNode and finds the best DataNode for the next block. This process is transparent to the client, which sees a continuous stream.
- Blocks are read in order, with the **DFSInputStream** opening new connections to DataNodes as the client reads through the stream. The Namenode is also called to retrieve the DataNode locations for the next batch of blocks as needed. When the client finishes reading, it calls **close()** on the **FSDataInputStream**.

If the **DFSInputStream** encounters an error with a DataNode, it will try the next nearest DataNode for that block. It will also remember the failed DataNodes to avoid unnecessary retries for later blocks. The **DFSInputStream** also verifies checksums for the data transferred to and from the DataNode. The Namenode then attempts to re-replicate the duplicate block as if it was under-replicated. The corrupted block does not get deleted until that re-replication succeeds.

This design allows HDFS to scale to a large number of concurrent clients since data traffic is spread across all DataNodes in the cluster. The Namenode only needs to service block location requests and does not serve data, which would quickly become a bottleneck as the number of clients increases.

## File write

When a file is written to various positions, as illustrated in *Figure 4.12*, it simultaneously conveys essential information to the Namenode. This information encompasses operations, block IDs, block names, rack IDs, and more.



4

*Figure 4.12: File write*

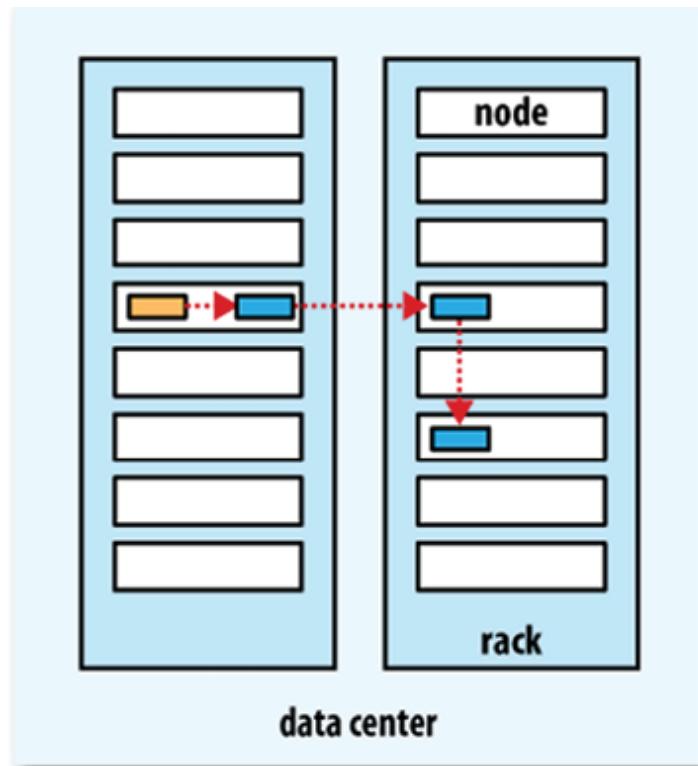
Functions involved in file write are explained as follows:

- To create a file, the client invokes the **create()** function on **DistributedFileSystem**.
- The process of creating a new file in Hadoop's **DistributedFileSystem** begins with a **Remote Procedure Call (RPC)** made to the Namenode. This RPC request aims to establish a new file within the filesystem's namespace without any initial data blocks assigned to it. To ensure the integrity and permissions of the filesystem, the existing Namenode performs several checks on both the filesystem itself and the client's rights to create files. If any of

these checks fail, an **IOException** is raised, indicating a problem with the request. Assuming all checks pass, the **DistributedFileSystem** grants the client permission to initiate the file creation process, returning an **FSDataOutputStream** for data writing. This output stream leverages a **DFSOutputStream**, which is responsible for managing the connection with both the DataNodes for data storage and the Namenode for metadata handling. In essence, this procedure ensures that Hadoop's **DistributedFileSystem** maintains control and accountability over file creation, enforcing access rights while orchestrating the distributed storage of data across the cluster's DataNodes.

- The **DFSOutputStream** splits the data into chunks or packets and creates a queue to store them in different locations, known as the data queue. The **DataStreamer** is responsible for placing the data queue and allocating a new block by requesting it from the Namenode, picking a list of suitable DataNodes to store the replicas.
- The data queue contains a pipeline that consists of three nodes, which is the default replication factor. The **DataStreamer** streams the chunks or packets to the initial DataNode connected to other nodes in the pipeline. All nodes keep sending data to each other in a pipeline manner.
- It involves the **DFSOutputStream** holding an internal queue of packets waiting to be acknowledged by DataNodes, known as the ack queue. The process also entails the **DFSOutputStream** maintaining an internal queue referred to as the ack queue. A packet is removed from the ack queue only if all DataNodes within the pipeline have acknowledged it. If any DataNode fails, the subsequent actions include closing the pipeline and adding any packets in the ack queue to the front of the data queue so that downstream DataNodes do not miss any packets. The failing DataNode is removed from the pipeline, and the remaining block data is written to the two good DataNodes in the pipeline. The Namenode notes that the block is under-replicated and arranges for an additional replica to be created on another node. Subsequent blocks are then treated as normal.

- When the client has finished writing data, it calls `close()` on the stream. It is possible, but unlikely, for multiple DataNodes to fail while a block is being written. As long as `dfs.namenode.replication.min` replicas as in *Figure 4.13* (default one) are written, the write can succeed, and the block will be asynchronously replicated across the cluster until its target replication factor is reached (`dfs.replication`, which defaults to three).
- All the remaining packets or chunks in the pipelines wait for an acknowledgment signal before creating contact with **Namenode**. **Namenode** has all the information about block locations, and it waits for minimal replicas before doing the task successfully.



*Figure 4.13: Replication*

## Coherency model

This model focuses on describing the data visibility of files in the filesystem to read and write. Some of the operations may create different behavior than expected due to POSIX requirements.

After creating a file, it is visible in the filesystem namespace, as expected:

```
Path p = new Path("p");
fs.create(p);

assertThat(fs.exists(p), is(true));

However, any content written to the file is not guaranteed to be visible, even
if the stream is flushed. So, the file appears to have a length of zero:

Path p = new Path("p");
OutputStream out = fs.create(p);
out.write("content".getBytes("UTF-8"));
out.flush();

assertThat(fs.getFileStatus(p).getLen(), is(0L));
```

Once more than a block's worth of data has been written, the first block will be visible to new readers. This is true of subsequent blocks, too: it is always the current block being written that is not visible to other readers.

## Cluster balance

Hadoop forms the cluster, which consists of DataNodes. These DataNodes are connected together and store the data. These data perform well when data is stored into distributed evenly; otherwise, their performance decreases. For example, there are 1,000 nodes, and out of that, only 100 nodes are in use to store data then it leads to disbalance. It is best to start by running distcp with the default of 20 maps per node. **distcp** is a tool used for large inter/intra cluster copying and uses MapReduce underneath to effect its distribution.

## Hadoop archive

Hadoop does not perform well when it stores data in a small number of files. It decreases the efficiency of the system. On the other side, large amount of data use large memory from the Namenode. Hadoop introduced the **Hadoop Archives (HAR)** files facility that bundles files into HDFS blocks more efficiently so that memory used in Namenode can be reduced and transparency can be maintained. It can be used as input to MapReduce programming models.

Archive tool used to run MapReduce job to process the data.

```
% hadoop fs -lsr /my/files  
-rw-r-r-- 1 tom supergroup 1 2010-05-10 14:11  
/my/files/a  
drwxr-xr-x- tom supergroup 0 2010-05-10 14:11  
/my/files/dir  
-rw-r-r-- 1 tom supergroup 1 2010-05-10 14:11  
/my/files/dir/b
```

Now we can run the **archive** command:

```
%  
hadoop archive -archiveName files.har  
/my/files /my
```

Then, we can analyze the results with **fsck**:

```
hdfs fsck / -files > ./fsckAfterHARCreated.txt
```

Archives files are identified by their **.har** extension and are comprised of two index files and one or more part files, which are merged versions of original files. The index files enable the lookup of a particular part file, along with its offset and length, where an archived file is stored. The application is shielded from these technicalities and interacts with the HAR files via the HAR URI scheme. The underlying filesystem, which is HDFS

in this case, has a HAR filesystem layer built on top of it to facilitate these interactions.

## Hadoop I/O

When dealing with multiple data set Hadoop takes the consideration of data integrity, compression, and multiterabyte blocks of data.

### Data integrity

Hadoop users expect their data to be stored and processed without loss or corruption. However, the high volume of data flowing through the system means that there is a greater probability of errors occurring during I/O operations on a disk or network. To detect corrupted data, verification is computed when the data enters the system and again when it is transmitted across an unreliable channel. If the newly generated verification does not match the original, the data is considered corrupt, but this does not provide a way to fix it.

To prevent corruption, a separate checksum is created for each **io.bytes.per.checksum** bytes of data, with a default value of 512 bytes. DataNodes are responsible for verifying the data they receive before storing it, and clients verify checksums when reading data from DataNodes. If an error is detected, the client receives a **ChecksumException** that it must handle by retrying the operation. DataNodes keep a persistent log of checksum verifications to detect unhealthy disks.

HDFS stores replicas of blocks and can replace corrupted blocks by copying one of the good replicas. When a client detects an error when reading a block, it reports the block and DataNode to the Namenode, which marks the replica as corrupt and schedules a new replica to be replicated on another DataNode. The corrupt replica is then deleted.

It is possible to disable verification of checksums if needed by setting the **dfs.checksum** Type property to NULL on the **hdfs-site**, but this should only be done in special cases, such as examining a corrupt file to determine if it can be salvaged.

## Local file system

The Hadoop **LocalFileSystem** performs client-side checksumming. This suggests that when you write a file referred to as a computer file name, the filesystem client transparently creates a hidden file, **.filename.crc**, within the same directory containing the checksums for every chunk of the file. Like HDFS, the chunk size is controlled by the **io.bytes.per.checksum** property, which defaults to 512 bytes. The chunk size is kept as metadata within the **.crc** file so that the file may be read back properly even if the setting for the chunk size has been modified.

Checksums are verified once the file is read, and if an error is detected, **LocalFileSystem** throws a **ChecksumException**. Checksums are fairly low cost to calculate (in Java, they are enforced in native code), typically adding a number of percent overhead to the time to read or write a file. For most applications, this can be an acceptable price to get data integrity. It is, however, possible to disable checksums, generally when the underlying filesystem supports checksums natively. This can be accomplished by using **RawLocalFileSystem** in situ of local **FileSystem**. This globally in an application, it suffices to remap the implementation for file URIs by setting the property **fs.file.impl** to the value **org.apache.hadoop.fs.RawLocalFileSystem**, or else, you can directly create a **Raw LocalFileSystem** instance, which can be helpful if you wish to disable confirmation verification for less than some reads; for example:

```
Configuration conf = ...  
FileSystem fs = new RawLocalFileSystem();  
fs.initialize(null, conf);
```

## Compression

Lossless and lossy compression are two distinct approaches used in Hadoop and other big data processing environments to reduce data size and storage requirements. Each method has its advantages and is chosen based on specific use cases and data characteristics. Lossless compression aims to

reduce data size without any loss of information. In Hadoop, this technique is valuable when data integrity is of utmost importance. It is commonly used for structured data, text files, and code files where every piece of information needs to be preserved. Hadoop offers several lossless compression codecs such as Snappy, Gzip, and LZO. While lossless compression can provide excellent compression ratios for certain types of data, it might not achieve the same level of compression as lossy methods. However, the key advantage is that you can always fully recover the original data without any loss. On the other hand, lossy compression is used when some degree of data loss is acceptable, typically for multimedia data like images, audio, and video. In Hadoop, popular lossy compression codecs include JPEG for images and MP3 for audio. Lossy compression reduces data size significantly but does so by discarding some of the less critical or perceptually insignificant details. While this can achieve high compression ratios, it is crucial to note that repeated compression and decompression can lead to a loss of quality over time. Therefore, lossy compression is chosen when the focus is on minimizing storage space and when some degradation in data quality can be tolerated. It is important to select the compression method that aligns with the specific requirements and constraints of your big data processing tasks in Hadoop.

Compression of files brings two major advantages: that are space and speed. There will be less space needed to store data with increased speed of data transfer. It is beneficiary for transferring large amounts of data across the network. All data is distributed and stored, so compression brings significant change with respect to speed and storage.

Following are the different compression techniques as in *Table 4.3* that are used as extensions for files:

<b>Compression format</b>	<b>Tool</b>	<b>Algorithm</b>	<b>Extension</b>	<b>Splittable</b>
DEFLATE	NA	DEFLATE	.deflate	No
gzip	gzip	DEFLATE	.gz	No

bzip2	bzip2	Bzip2	.bz2	Yes
LZO	lzop	LZO	.lzo	No
Snappy	NA	Snappy	.snappy	No

**Table 4.3:** Different compression techniques

All techniques have its own properties with respect to space/time trade off. Following are the points that can be taken into consideration:

- Gzip is the most optimized compression technique.
- Bzip2 is efficient than gzip but slow with respect to speed.
- Bzip2 having faster decompression speed rather than compression speed.
- LZO and Snappy are optimized for speed but it is faster than gzip but less effectively.
- Snappy is also faster in comparison to LZO for decompression.

## Codecs

Codecs, short for *coder-decoder*, are software or hardware components used to compress and decompress digital data. They are widely employed in various applications to reduce the size of data for storage or transmission and then restore it to its original form when needed. This is the technique for compression and decompression. A codec is represented by the **CompressionCodec** interface. **GzipCodec** encapsulates compression and decompression for gzip. Following are the comparison techniques (refer to [Table 4.4](#)) useful for Hadoop:

Compression format	Hadoop compression codec
DEFLATE	org.apache.hadoop.io.compress.DefaultCodec

gzip	org.apache.hadoop.io.compress.GzipCodec
bzip2	org.apache.hadoop.io.compress.BZip2Codec
LZO	com.hadoop.compression.lzo.LzopCodec
Snappy	org.apache.hadoop.io.compress.SnappyCodec

**Table 4.4:** Hadoop compression codecs

Following are the points that need to be in consideration when we talk about compression:

- **CompressionCodec** works with two methods that allow for compress or decompress data.
- **CreateOutputStream** method use to write uncompressed data in compressed form to underlying stream with **CompressionOutputStream**.
- To read decompress data from input stream, **createInputStream** use that allow to read uncompressed data.

```
public class StreamCompressor {

    public static void main(String[] args) throws
Exception {

    String codecClassname = args[0];

    Class<?> codecClass =
    Class.forName(codecClassname);

    Configuration conf = new Configuration();

    CompressionCodec codec = (CompressionCodec)
    ReflectionUtils.newInstance(codecClass, conf);

    CompressionOutputStream out =
    codec.createOutputStream(System.out);
```

```
    IOUtils.copyBytes(System.in, out, 4096, false);  
    out.finish();  
}  
}
```

The preceding code shows the program to compress data read from standard input and write it to standard output.

## Compression and input splits

In a distributed environment like Hadoop, accessing and using data can be challenging due to its distributed nature. Compression plays a crucial role in addressing this challenge.

Let us consider an uncompressed file stored in HDFS with a size of 1 GB. Since the default block size in HDFS is 64 MB, this file would be divided into 16 blocks. Each block can hold up to 64 MB of data.

When a MapReduce job, which is a programming structure used for processing data in Hadoop, uses this file as input, it will generate 16 input splits. Each input split represents a portion of the file and is processed independently as input to a separate map task. This division allows for parallel processing of the data, with each map task working on a specific input split.

By dividing the data into smaller input splits and processing them independently, Hadoop can efficiently distribute the workload across multiple nodes in the cluster, enabling faster and more scalable data processing.

In Hadoop's distributed environment, data is typically divided into blocks and stored across multiple nodes in the HDFS. Compression is often employed to reduce the storage space required and improve data processing efficiency.

Let us consider an uncompressed file in HDFS with a size of 1 GB. In HDFS, the default block size is typically 64 MB. Therefore, the 1 GB file

would be divided into approximately 16 blocks, with each block holding 64 MB of data. Each block is replicated across multiple nodes in the Hadoop cluster to provide fault tolerance.

When a MapReduce job is executed on this file, it will be processed in parallel by multiple map tasks. The number of input splits generated by the MapReduce framework depends on the number of blocks in the input file. In this case, since we have 16 blocks, there will be 16 input splits. Each input split represents a portion of the file that is processed independently by a separate map task.

The MapReduce framework ensures that each input split is processed by only one map task. The map tasks run in parallel across the cluster, with each task processing its assigned input split. This parallel processing enables efficient utilization of the distributed resources in the Hadoop cluster and enables faster data processing.

By dividing the input file into smaller input splits and processing them in parallel, Hadoop's MapReduce framework can effectively handle large-scale data processing tasks in a distributed environment. Compression can further optimize storage and processing efficiency by reducing the size of the data that needs to be transmitted over the network and processed by the map tasks.

Imagine currently the file could be a gzip-compressed file, whose compressed size is 1 GB. As before, HDFS could store the file in 16 blocks. However, making a split for every block will not work since it is impossible to begin reading at discretionary purpose within the gzip stream, and therefore not possible for a map task to read its split severally from the others. The gzip format uses DEFLATE to store the compressed data, and DEFLATE stores data as a series of compressed blocks. The matter is that the beginning of every block is not distinguished in any manner that may enable a reader positioned at an arbitrary purpose in the stream to advance to the start of the succeeding block, thereby synchronizing itself with the stream. For this reason, gzip does not support splitting. In this case, MapReduce can do the correct thing and not try to split the gzipped file since it is aware that the input is gzip-compressed and that gzip does not support splitting. This can work, however, at the expense of locality: a single map can method the 16

HDFS blocks, most of which cannot be native to the map. Also, with fewer maps, the task is a smaller amount granular, so it could take longer to run.

To compress the output of a MapReduce job, in the job configuration, set the **mapred.output.compress** property to true and the **mapred.output.compression.codec** property to the **classname** of the compression codec you want to use:

```
public class MaxTemperatureWithCompression {  
    public static void main(String[] args) throws Exception {  
        if (args.length != 2) {  
            System.err.println("Usage:  
MaxTemperatureWithCompression <input path> " + "  
<output path>");  
            System.exit(-1);  
        }  
        Job job = new Job();  
        job.setJarByClass(MaxTemperature.class);  
        FileInputFormat.addInputPath(job, new Path(args[0]));  
        FileOutputFormat.setOutputPath(job, new Path(args[1]));  
        job.setOutputKeyClass(Text.class);  
        job.setOutputValueClass(IntWritable.class);  
        FileOutputFormat.setCompressOutput(job,  
true);
```

```
FileOutputFormat.setOutputCompressorClass(job,  
GzipCodec.class);  
  
job.setMapperClass(MaxTemperatureMapper.class);  
  
job.setCombinerClass(MaxTemperatureReducer.class);  
  
job.setReducerClass(MaxTemperatureReducer.class);  
System.exit(job.waitForCompletion(true) ? 0  
: 1);  
}  
}
```

## Map output

Map output, which is the input of the reducer, can be compressed data. Map data is written into a disk and circulated for input purposes on the network. Fast compressors like LZO and Snappy can be used for this purpose. All this is because the volume of data is subject to transfer during operation:

```
Configuration conf = new Configuration();  
conf.setBoolean("mapred.compress.map.output",  
true);  
  
conf.setClass("mapred.map.output.compression.codec"  
, GzipCodec.class,  
CompressionCodec.class);  
Job job = new Job(conf);
```

## Serialization

Serialization is the process of converting structured objects into a byte stream, which can be transmitted over a network or written to persistent storage. The reverse process is known as deserialization, where the byte stream is converted back into a series of structured objects.

Serialization is used for two main purposes: interprocess communication and persistent storage. For interprocess communication, **Remote Procedure Call (RPC)** protocols are commonly used to establish communication between two nodes. Serialization is used to convert the message into a binary stream that is sent to the remote node, and deserialization is used to reverse the process.

There are several criteria that are desirable in a serialization protocol, including split and reconstruct data easily:

- **Compact:** First, it should be compact to make the best use of network bandwidth and use less storage, which is a scarce resource in a data center.
- **Fast:** Second, it should be fast, as interprocess communication is a critical component of a distributed system.
- **Extensible:** Third, it should be extensible, allowing the protocol to evolve in a controlled manner as new requirements arise. For example, new arguments can be added to a method call, and old clients can still communicate with new servers without any issues.
- **Interoperable:** Finally, the format should be interoperable, allowing clients written in different languages to communicate with the server without any compatibility issues.

*Table 4.5* shows the primitives use serialization:

Java primitive	Writable implementation	Serialized size (bytes)
boolean	BooleanWritable	1

Java primitive	Writable implementation	Serialized size (bytes)
byte	ByteWritable	1
short	ShortWritable	2
int	IntWritable	4
float	FloatWritable	4
long	LongWritable	8
double	DoubleWritable	8

**Table 4.5:** Writable wrapper class for Java primitives

## Avro file-based data structure

Avro is a data serialization system that provides a compact, fast, and schema-based binary format for data exchange. Avro files are based on a well-defined Avro schema, which defines the structure and types of the data stored within the file. Here are some lines explaining Avro file-based data structures:

- **Avro schema:** Avro files rely on an Avro schema to define the structure of the data. The schema is written in JSON format and specifies the fields, data types, and hierarchical relationships of the data elements.
- **Dynamic typing:** Avro supports dynamic typing, allowing schema evolution over time. This means that Avro files can handle changes to the schema, such as adding or removing fields, without requiring data migration or schema versioning.
- **Compact and efficient:** Avro files are designed to be compact and efficient, using binary encoding and efficient data compression

techniques. This results in smaller file sizes and faster data serialization and deserialization processes.

- **Schema resolution:** Avro files support schema resolution, which allows compatibility between different versions of schemas. This enables reading data written with a previous schema version using a newer schema, as long as the changes are backward-compatible.
- **Metadata and data interchangeability:** Avro files can include metadata, such as field names and data types, within the file itself. This self-describing nature makes Avro files highly portable and enables data interchangeability across different programming languages and systems.
- **Integration with Hadoop ecosystem:** Avro is well-integrated with the Hadoop ecosystem, allowing seamless integration with tools like Apache Hive, Apache Pig, and Apache Spark. Avro files can be used as input or output formats for various Hadoop data processing tasks.

Apache Avro is a data serialization system that is language-neutral. This project was created by Doug Cutting, the creator of Hadoop, to handle the key downside of Hadoop Writables: lack of language portability. Having a data format that will be processed by many languages (currently C, C++, C#, Java, Python, and Ruby) makes it easier to share data sets with a wider audience than one tied to a single language. It is additionally a lot of future-proof, allowing data to potentially outlive the language accustomed read and write it.

But why does a new data serialization system, Avro, contain a set of options that, taken alone, differentiate it from alternative systems like Apache Thrift or Google's Protocol Buffers. Like these systems, Avro data is represented using a language-independent schema. However, not like another system, code generation is optional in Avro, which means data can be read and write that conforms to a given schema even if the code has not seen schema before.

The following are the points that can be considered for Avro file-based data structure:

- Avro schemes are written in JSON and encoded in binary format.
- Avro IDL is a C-like language that is more familiar to developers.
- It uses a JSON-based data encoder, which is human-readable and useful for debugging and prototyping purposes of Avro.
- Avro indeed offers robust schema resolution capabilities, allowing for flexible data reading and writing processes. The schema resolution in Avro allows the schema used for reading the data to have constraints that need not be identical to the schema used for writing the data.
- Avro uses a schema evolution approach that allows for schema compatibility between different versions of data. This means that the schema used to read data can be more permissive or have additional constraints than the schema used to write the data.
- New and old clients are able to read old data, while new clients of Avro write new data in new criteria.
- An Avro data file has a metadata section that is useful in self-declaration.
- Avro data file supports compression, and it is easy to split-based data, which is useful for MapReduce purposes, and it keeps data input format.

## **Data type and schemas**

Avro uses data types that are useful to build data applications.

Interoperability implementation is the base for the data type of Avro. *Table 4.6* shows primitive data type, and *Table 4.7* shows complex data type:

Type	Description	Schema
Null	Absence of value	“null”
Boolean	Binary value	“nicode”
Int	32-bit signed integer	“int”
Long	64-bit signed integer	“long”

Type	Description	Schema
Float	Single precision (32 bit)	“float”
Double	Double precision (64 bit)	“double”
Bytes	Sequence of 8-bit unsigned bytes	“bytes”
String	Sequence of nicode characters	“string”

**Table 4.6:** Primitive data type

Following are the table that shows complex data types:

Type	Description	Example
Array	An ordered collection of objects. All objects in a particular array must have the same schema.	{ "type": "array", "items": "long" }
map	An unordered collection of key–value pairs. Keys must be strings, values may be any type, although within a particular map all values must have the same schema.	{ "type": "map", "values": "string" }

Type	Description	Example
Record	A collection of named fields of any type.	<pre>{   "type": "record",   "name": "WeatherRecord",   "doc": "A weather reading.",   "fields": [     {"name": "year", "type": "int"},     {"name": "temperature", "type": "int"},     {"name": "stationId", "type": "string"}] }</pre>

Type	Description	Example
enum	A set of named values.	{           "type": "enum",           "name": "Cutlery",           "doc": "An eating utensil.",           "symbols": ["KNIFE", "FORK", "SPOON"]         }
fixed	A fixed number of 8-bit unsigned bytes.	{           "type": "fixed",           "name": "Md5Hash",           "size": 16         }
union	A union of schemas. A union is represented by a JSON array, where each element in the array is a schema.	"null",         "string",         {"type": "map",         "values": "string"}

**Table 4.7:** Complex data structure

The following points of avro represent its importance for data types and data structures:

- Each API of Avro has a representation of Avro type for its language; that is, Avro's double is represented in different languages such as in C, C++, and Java by double and float in Python and Ruby.
- There may be more than one representation for the language, that is, dynamic mapping, which is similar to generic mapping in Java.
- Java and C++ can generate code for representing data for Avro schema.
- Specific mapping is the mapping when code is known before implementing it that is used in Java programming.
- Reflection mapping maps avro data type onto preexisting Java types, and this is slower than the previous two mapping techniques.

Reflect mapping is not recommended for new applications. *Table 4.8* shows the difference between all mapping types used in Java with avro types:

<b>Avro type</b>	<b>Generic Java mapping</b>	
null	Null type	
boolean	boolean	
int	int	
long	long	
float	float	
double	double	
bytes	Java.nio.ByteBuffer	
string	Org.apache.avro.util.utf8 or java.lang.String	
array	Org.apache.avro.generic.GenericArray	
map	Java.util.Map	

<b>Avro type</b>	<b>Generic Java mapping</b>	
record	org.apache.avro.generic.GenericRecord	org.apache
enum	java.lang.String	Generated Ja
fixed	org.apache.avro.genereic.GenericFixed	Generated org.apache
union	java.lang.Object	

**Table 4.8:** Avro Java type mapping

## Serialization and deserialization

Avro provides APIs for the same, which is useful to integrate with pre-existing systems. A Java program to read and write Avro data to and from streams:

```
{
    "type": "record",
    "name": "StringPair",
    "doc": "A pair of strings.",
    "fields": [
        {"name": "left", "type": "string"},
        {"name": "right", "type": "string"}
    ]
}
```

If this schema is saved in a file on the classpath called **StringPair.avsc** (.avsc is the conventional extension for an Avro schema), then we can load it using the following two lines of code:

```
Schema.Parser parser = new Schema.Parser();

    Schema schema =
parser.parse(getClass().getResourceAsStream("String
Pair.avsc"));
```

An instance of an Avro record using the generic API is as follows:

```
GenericRecord datum = new
GenericData.Record(schema);

    datum.put("left", "L");
    datum.put("right", "R");
```

Serialize the record to an output stream:

```
ByteArrayOutputStream out = new
ByteArrayOutputStream();

    DatumWriter<GenericRecord> writer = new
GenericDatumWriter<Generi      cRecord>(schema);

    Encoder encoder =
EncoderFactory.get().binaryEncoder(out, null);

    writer.write(datum, encoder);
    encoder.flush();
    out.close();
```

## Avro MapReduce

There are a number of classes used to run MapReduce programs on Avro data. **AvroMapper** and **AvroReducer** in **org.apache.avro.mapred** package are the specializations of Hadoop. It eliminates key-value for input and output because Avro data files are a sequence of values:

```
import java.io.IOException;
```

```
import org.apache.avro.Schema;
import org.apache.avro.generic.GenericData;
import org.apache.avro.generic.GenericRecord;
import org.apache.avro.mapred.AvroCollector;
import org.apache.avro.mapred.AvroJob;
import org.apache.avro.mapred.AvroMapper;
import org.apache.avro.mapred.AvroReducer;
import org.apache.avro.mapred.AvroUtf8InputFormat;
import org.apache.avro.mapred.Pair;
import org.apache.avro.util.Utf8;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
public class AvroGenericMaxTemperature extends
Configured implements Tool {
```

```
    private static final Schema SCHEMA = new
Schema.Parser().parse(
    "{" +
        " \"type\": \"record\", " +
        " \"name\": \"WeatherRecord\", " +
        " \"doc\": \"A weather reading.\", " +
        " \"fields\": [ " +
            " " +
            "{\"name\": \"year\", \"type\": \"int\"}, " +
            " " +
            "{\"name\": \"temperature\", \"type\":
\"int\"}, " +
            " " +
            "{\"name\": \"stationId\", \"type\":
\"string\"} " +
            " ]" +
        " }"
);
public static class MaxTemperatureMapper
    extends AvroMapper<Utf8, Pair<Integer,
GenericRecord>> {
    private NcdcRecordParser parser = new
NcdcRecordParser();
```

```
    private GenericRecord record = new
GenericData.Record(SCHEMA);

    @Override

    public void map(Utf8 line,
                    AvroCollector<Pair<Integer, GenericRecord>>
collector,
                    Reporter reporter) throws IOException {

        parser.parse(line.toString());
        if (parser.isValidTemperature()) {
            record.put("year", parser.getYearInt());
            record.put("temperature",
parser.getAirTemperature());
            record.put("stationId",
parser.getStationId());
            collector.collect(
                new Pair<Integer, GenericRecord>
(parser.getYearInt(), record));
        }
    }

    public static class MaxTemperatureReducer
        extends AvroReducer<Integer, GenericRecord,
GenericRecord> {

    }

    @Override
```

```
    public void reduce(Integer key,
Iterable<GenericRecord> values,
                    AvroCollector<GenericRecord> collector,
                    Reporter reporter)
                    throws IOException {
    GenericRecord max = null;
    for (GenericRecord value : values) {
        if (max == null ||
            (Integer) value.get("temperature") >
            (Integer) max.get("temperature")) {
            max = newWeatherRecord(value);
        }
    }
    collector.collect(max);
}
private GenericRecord
newWeatherRecord(GenericRecord value) {
    GenericRecord record = new
    GenericData.Record(SCHEMA);
    record.put("year", value.get("year"));
    record.put("temperature",
    value.get("temperature"));
    record.put("stationId",
    value.get("stationId"));
}
```

```
        return record;
    }

    @Override
    public int run(String[] args) throws Exception
{
    if (args.length != 2) {
        System.err.printf("Usage: %s [generic
options] <input> <output>\n",
getClass().getSimpleName());
        ToolRunner.printGenericCommandUsage(System.err);
        return -1;
    }

    JobConf conf = new JobConf(getConf(),
getClass());
    conf.setJobName("Max temperature");
    FileInputFormat.addInputPath(conf, new
Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new
Path(args[1]));
    AvroJob.setInputSchema(conf,
Schema.create(Schema.Type.STRING));
    AvroJob.setMapOutputSchema(conf,
```

```

        Pair.getPairSchema(Schema.create(Schema.Type.INT),
SCHEMA));

        AvroJob.setOutputSchema(conf, SCHEMA);

conf.setInputFormat(AvroUtf8InputFormat.class);

        AvroJob.setMapperClass(conf,
MaxTemperatureMapper.class);

        AvroJob.setReducerClass(conf,
MaxTemperatureReducer.class);

        JobClient.runJob(conf);

        return 0;

    }

}

public static void main(String[] args) throws
Exception {

    int exitCode = ToolRunner.run(new
AvroGenericMaxTemperature(), args);

    System.exit(exitCode);

}

```

The following are the important points regarding Avro MapReduce:

- Use of **org.apache.avro.mapred.Pair** to wrap the map output key and value in **MaxTemeratureMapper**.
- The key is the year, and the value is the weather record that is represented in **GenericRecord**.

- Avro **MapReduce** content key–value pair for input to the reducer.
- The use of **AvroJob** creates a difference from regular **MapReduce**.
- The map output schema is a pair schema whose key schema is an Avro **int** and whose value schema is the weather record schema. The final output schema is the weather record schema, and the output format is the default, **AvroOutputFormat**, which writes to Avro data files.

## Conclusion

The basics of Hadoop provide a solid foundation for understanding and harnessing the power of this distributed data processing framework. We have explored key concepts and objectives that underpin Hadoop’s design and functionality.

Scalability is a core objective, enabling Hadoop to handle massive data sets by distributing the workload across clusters of commodity hardware. Fault tolerance ensures uninterrupted data processing by replicating data and automatically redirecting tasks in the event of hardware failures.

Hadoop’s distributed computing model allows for efficient utilization of resources through parallel processing of data. Data locality further optimizes performance by processing data on the same node where it resides, minimizing network transfer.

Flexibility is a key feature, enabling Hadoop to handle various data types and support different processing models. Batch processing with MapReduce, interactive querying with Hive, and real-time processing with Apache Storm are among the capabilities offered by Hadoop.

By embracing these basics, organizations can effectively store, process, and analyze large amounts of data. Hadoop’s scalability, fault tolerance, distributed computing, data locality, and flexibility make it a powerful framework for tackling the challenges of big data.

As technology continues to evolve, Hadoop remains at the forefront of the big data landscape, enabling businesses to leverage the insights hidden within their vast data stores. Understanding the basics of Hadoop sets the

stage for further exploration and application of advanced concepts and techniques in the world of distributed data processing.

In the upcoming chapter, we will learn different ways to install Hadoop technology.

### **Join our book's Discord space**

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



# CHAPTER 5

## Hadoop Installation

### Introduction

Hadoop requires Linux as the operating system. It can act as a platform either by VmWare or direct installation. Hadoop is a powerful open-source framework designed to process and store vast amounts of data across a distributed cluster of computers. It is widely used for big data processing and analytics. To set up Hadoop on your system, you need to follow specific steps to ensure a successful installation.

In this chapter, we will explore its key components, **Hadoop Distributed File System (HDFS)** and Hadoop MapReduce, and learn how they form the backbone of big data processing. From choosing the right Hadoop distribution to configuring essential files and services, we will cover all the essential steps to get you up and running with Hadoop.

### Structure

In this chapter, we will go through the following topics:

- Using standalone (local) mode
- Implementing pseudo-distributed mode
- Configuring fully distributed mode

### Objectives

The objective of installing Hadoop is to set up a distributed and scalable framework that enables efficient storage, processing, and analysis of large volumes of data across clusters of commodity hardware. By implementing

Hadoop, organizations can harness the power of parallel processing to handle big data workloads, achieve high availability through data replication, and utilize its ecosystem of tools for various data-centric tasks such as batch processing, real-time data streaming, and complex analytics, ultimately facilitating informed decision-making and insights extraction from diverse and extensive datasets.

## Using standalone (local) mode

In Hadoop, the Standalone mode is the simplest and most basic mode of operation. It is primarily used for development, testing, and debugging purposes. Unlike the distributed mode, where Hadoop runs on a cluster of machines, in standalone mode, Hadoop runs as a single Java process on a single machine without utilizing the distributed capabilities of Hadoop.

Here is an overview of the Standalone mode of Hadoop:

- **Single-node configuration:** In standalone mode, Hadoop operates on a single machine, which acts as both the master and worker node. There is no need to set up a multi-node cluster.
- **No HDFS:** In standalone mode, Hadoop does not use the **Hadoop Distributed File System (HDFS)**. Instead, it reads data directly from the local file system of the machine where it is running. This means that the input data is limited to the local storage of that machine.
- **No distributed processing:** Unlike the fully distributed mode of Hadoop, where tasks are distributed and executed across multiple nodes, standalone mode runs all tasks within the same Java process. This can be useful for quick testing and debugging purposes on small datasets.
- **Use of Local JobRunner:** Standalone mode uses the **LocalJobRunner**, which simulates the MapReduce process within a single **Java Virtual Machine (JVM)**. The **LocalJobRunner** splits the input data into smaller chunks, processes them within the same JVM, and aggregates the results locally.
- **Simplified configuration:** In standalone mode, you do not need to set up complex configurations for Hadoop components like the **NameNode**, **DataNode**, **ResourceManager**, and **NodeManager**, as they are not used in this mode.

Standalone mode is particularly helpful for developers who want to experiment with Hadoop without the overhead of setting up a full-fledged distributed cluster. It allows you to run MapReduce jobs on your local machine and get familiar with the basic Hadoop programming paradigms. However, it should be noted that the standalone mode is not suitable for handling large-scale production workloads as it lacks the scalability and fault-tolerance features provided by the fully distributed mode.

To run Hadoop in standalone mode, you can simply execute the desired MapReduce job using the **hadoop** command-line tool, and Hadoop will run the job within the same JVM on your local machine.

Example of running a Hadoop MapReduce job in standalone mode:

```
$ hadoop jar your-mapreduce-job.jar input-path output-path
```

Overall, the Standalone mode in Hadoop serves as a starting point for learning and developing MapReduce jobs before transitioning to a full Hadoop cluster setup for real-world big data processing.

### **Implementing pseudo-distributed mode**

Pseudo-distributed mode is a type of Hadoop installation that simulates a fully distributed Hadoop cluster on a single machine. In this mode, each Hadoop component runs as a separate process, just like it would in a real distributed environment, but all processes run on the same machine. Pseudo-distributed mode is often used for testing and development purposes, allowing users to experiment with Hadoop's distributed features on a single-node setup before deploying it on a multi-node cluster.

Here is an overview of the pseudo-distributed mode for Hadoop installation:

- **Single machine configuration:** In pseudo-distributed mode, all the required Hadoop daemons, such as **NameNode**, **DataNode**, **ResourceManager**, **NodeManager**, and **JobHistoryServer**, are set up to run on a single machine.
- **Hadoop Distributed File System (HDFS) usage:** In this mode, Hadoop uses the HDFS for storing data. Although it runs on a single machine, it mimics the behavior of a real distributed file system, allowing you to experiment with data storage and replication.

- **Separate Java processes:** Each Hadoop component runs as a separate Java process. For example, the **NameNode**, **DataNode**, and other services each have their own Java process running on the same machine.
- **Configuration files:** To set up Hadoop in pseudo-distributed mode, you need to configure the necessary XML files (for example, **core-site.xml**, **hdfs-site.xml**, **yarn-site.xml**) to define the properties for the Hadoop services.
- **Network communication:** Although all components run on the same machine, they communicate with each other as if they were on separate machines. This simulates the communication patterns that occur in a real distributed Hadoop cluster.
- **Testing distributed jobs:** Pseudo-distributed mode allows you to run MapReduce jobs as if you were running them on a real distributed cluster. This makes it easier to test and debug your Hadoop applications before deploying them on a production cluster.
- **Learning environment:** Pseudo-distributed mode is also a valuable learning tool for those new to Hadoop. It provides an opportunity to explore Hadoop's architecture and practice working with distributed data processing.

Hadoop is installed and configured on a single system in this mode, with all file configurations including **hdfs-site.xml**, **mapred-site.xml**, and **core-site.xml**. It provides a real environment to test the data used in Hadoop. Java Home path is also set to check file activity over it. A single node is treated as the master as well as the data node. All daemons run in the pseudo-distributed mode in separate Java process, while there is no daemon run in standalone mode.

To install Hadoop in pseudo-distributed mode, you typically follow these steps:

1. Download the Hadoop distribution and extract it on your local machine.
2. Set up the necessary environment variables like **JAVA\_HOME** and **HADOOP\_HOME**.
3. Configure the Hadoop XML files (**core-site.xml**, **hdfs-site.xml**, **yarn-site.xml**) to specify settings such as file system paths, replication factor, and resource allocation.

4. Format the HDFS file system to prepare it for use.
5. Start the Hadoop daemons (**NameNode**, **DataNode**, **ResourceManager**, **NodeManager**, **JobHistoryServer**) one-by-one.

Once you have successfully set up Hadoop in pseudo-distributed mode, you can run MapReduce jobs and test the functionality of HDFS as if you were working with a real Hadoop cluster. However, keep in mind that pseudo-distributed mode is not suitable for large-scale production workloads, as it still runs on a single machine with limited resources and lacks the true fault-tolerance and scalability of a real distributed Hadoop cluster.

The system can be configured either with VmWare or directly installed Hadoop on Ubuntu. Following are the two methods for configuring Hadoop on system:

## VmWare

VMware provides virtualization and cloud computing software and services. It is known for its flagship product, VMware vSphere, which enables the creation and management of virtualized computing environments.

### Installing Hadoop using Ubuntu

Following are the steps to follow for installing Hadoop using Ubuntu:

1. Download Ubuntu 16.04 (Linux flavor or a different version) by using the following link:

<https://www.vmware.com/in/products/workstation-pro/workstation-pro-evaluation.html>

Download VmWare workstation and configure it by  
[https://my.vmware.com/web/vmware/info?slug=desktop\\_end\\_user\\_computing/vmware\\_workstation/11\\_0](https://my.vmware.com/web/vmware/info?slug=desktop_end_user_computing/vmware_workstation/11_0).

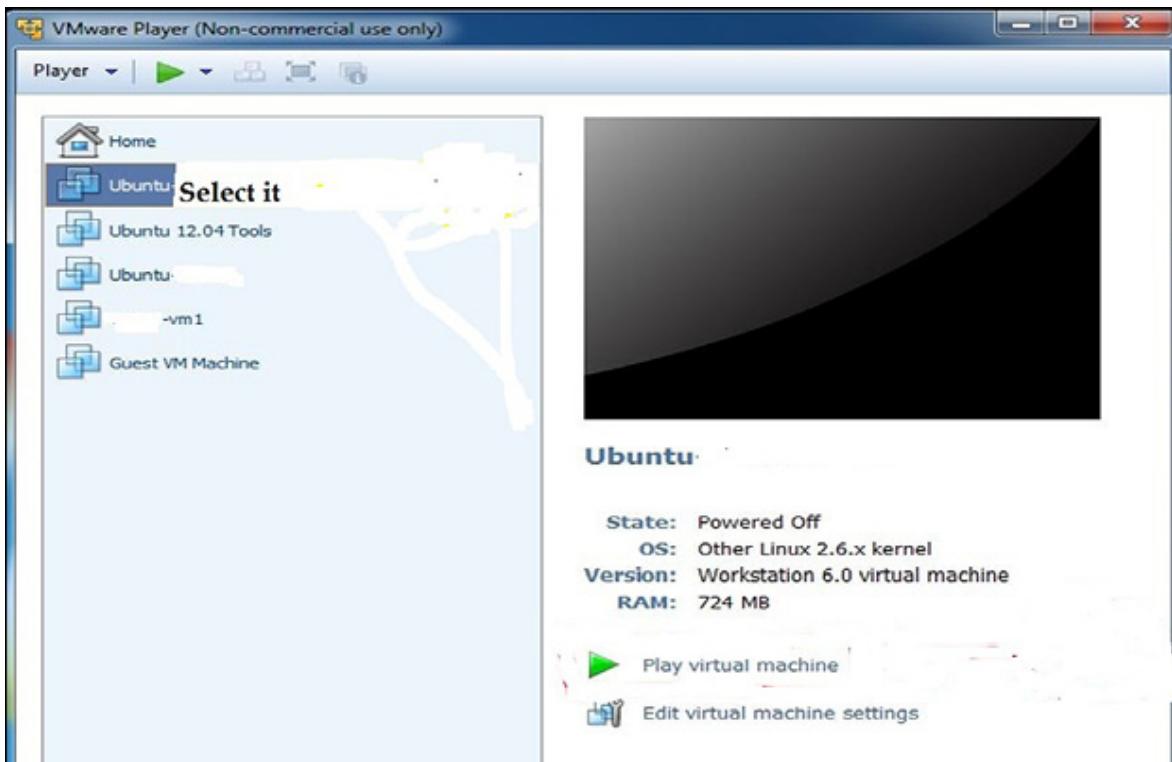
2. Open VMware Player and click **Open Virtual Machine** and select a path where an image of Ubuntu has been extracted. After that select the **.vmx** file and click **Ok**.

*Figure 5.1* shows the path of the installation file of VmWare while installing it on our machine:



**Figure 5.1:** Installation path of VmWare

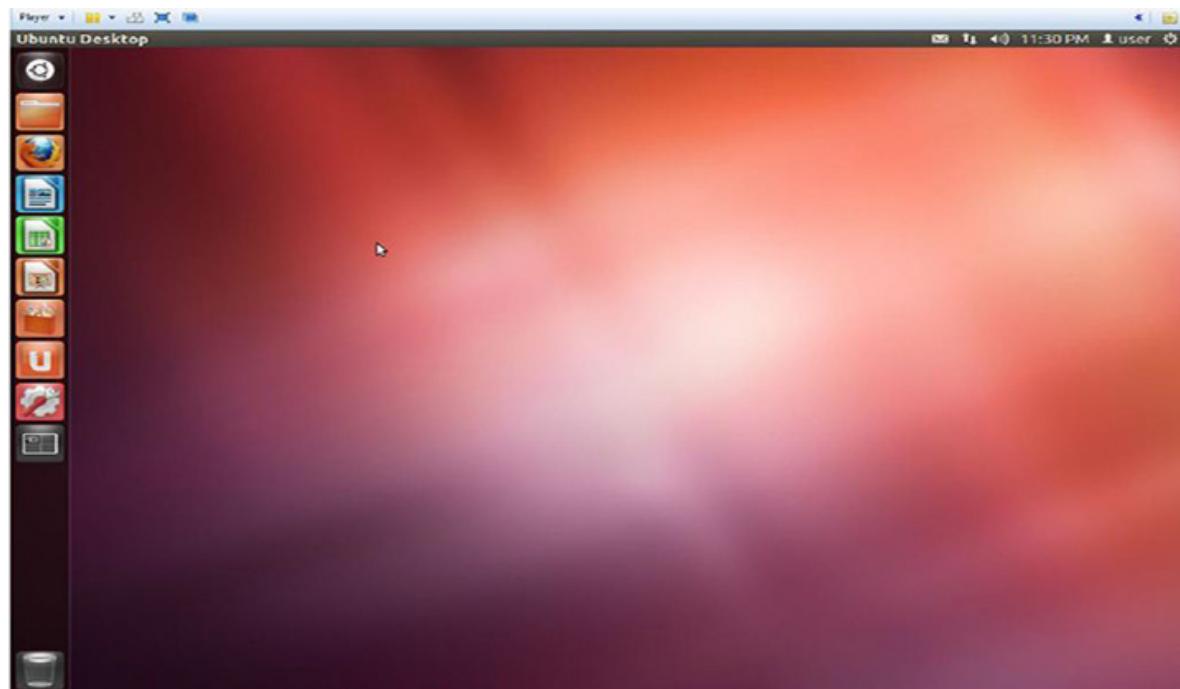
**Figure 5.2,** shows the VMware Player screen:



**Figure 5.2:** VmWare Player Screen

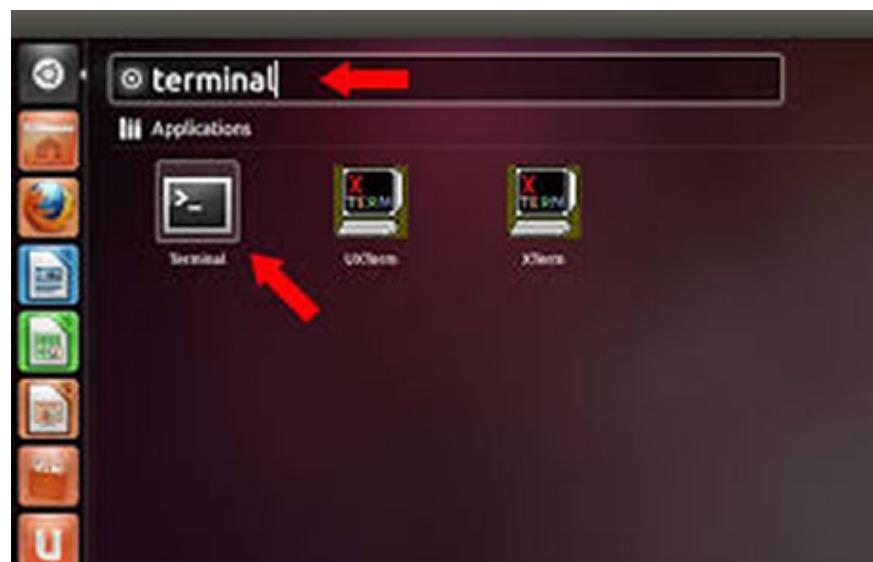
3. Double click on Ubuntu present in VMware Player which is indicated as 1 in **Figure 5.2.** After that click on **Play Virutal Machine**, which runs the

virtual machine on the system and a screen, as shown in *Figure 5.3*, will appear:



**Figure 5.3:** VmWare Terminal

Open the terminal.



**Figure 5.4:** Terminal for commands

4. *Figure 5.1* will show the reflection of Ubuntu operating system using VmWare. After installing it click on top left button and type terminal. Update the repository using **sudo apt-get update** as shown in *Figure 5.5*:

```
user@ubuntu:~$ sudo apt-get update
```

*Figure 5.5:* Update the repository using the command

5. When the repository has been updated, then use **sudo apt-get install openjdk-8-jdk**. The command is used to install OpenJDK 8 (Java Development Kit) on a system. OpenJDK is an open-source implementation of the Java Platform, Standard Edition (Java SE) as shown in *Figure 5.6*:

```
user@ubuntu:~$ sudo apt-get install openssh-server  
[sudo] password for user:
```

*Figure 5.6:* Command for installing OpenJDK 8

6. To check if Java has been installed on your system or not, use the command **java -version**.

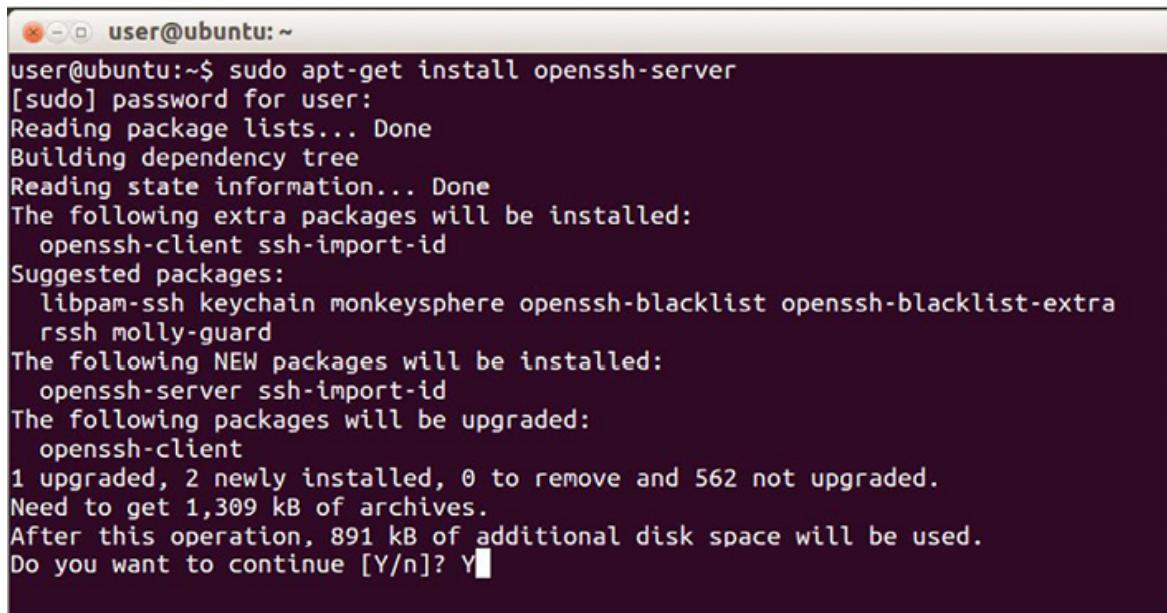
The command **sudo apt-get install openssh-server** is used to install the OpenSSH server on a system running a Linux distribution, such as Ubuntu. OpenSSH is a widely-used open-source implementation of the **SSH (Secure Shell)** protocol, which allows secure remote access and file transfer between computers over a network.

*Figure 5.7* reflects the successful installation of OpenSSH server and asks for a password. This password would be the same which was created at the time of admin mode setup of Ubuntu:

```
user@ubuntu:~$ sudo apt-get install openjdk-8-jdk
```

*Figure 5.7:* Post successful installation

The system needs to be connected with the Internet to provide all packages which are needed for the installation, as shown in *Figure 5.8*:



```
user@ubuntu:~$ sudo apt-get install openssh-server
[sudo] password for user:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  openssh-client ssh-import-id
Suggested packages:
  libpam-ssh keychain monkeysphere openssh-blacklist openssh-blacklist-extra
  rssh molly-guard
The following NEW packages will be installed:
  openssh-server ssh-import-id
The following packages will be upgraded:
  openssh-client
1 upgraded, 2 newly installed, 0 to remove and 562 not upgraded.
Need to get 1,309 kB of archives.
After this operation, 891 kB of additional disk space will be used.
Do you want to continue [Y/n]? Y
```

*Figure 5.8: Packets installation*

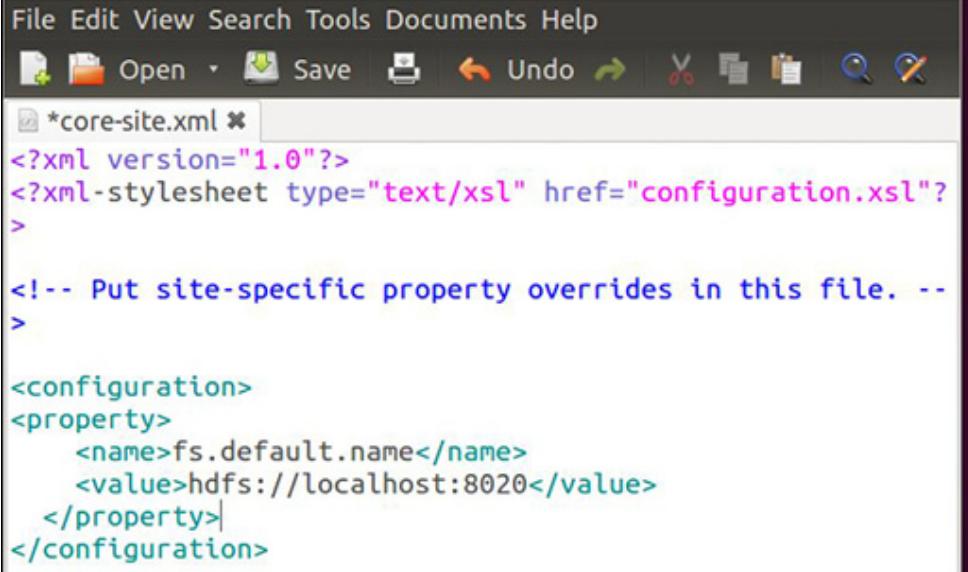
7. Download Hadoop from:

<https://dlcdn.apache.org/hadoop/common/hadoop-3.2.4/hadoop-3.2.4-src.tar.gz>

The commands to be used are:

```
wget
http://www.apache.org/dyn/closer.cgi/hadoop/common/hadoop
-3.2.4/hadoop-3.2.4-src.tar and tar -xvf hadoop-
3.2.4.tar.gz
```

8. Edit **core-site.xml**. It is one of the configuration files used in Apache Hadoop, a popular open-source distributed computing framework. It is an XML file that contains core configuration settings for the Hadoop ecosystem, providing critical information about the HDFS and other core components. Use the command **sudo gedit hadoop-3.2.4/conf/core-site.xml**:



The screenshot shows a text editor window with the title bar "File Edit View Search Tools Documents Help". The menu bar includes "Open", "Save", "Undo", and "Redo". The toolbar includes icons for Open, Save, Undo, Redo, Cut, Copy, Paste, Find, and Replace. The main editor area contains the XML code for `core-site.xml`:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>
<property>
    <name>fs.default.name</name>
    <value>hdfs://localhost:8020</value>
  </property>
</configuration>
```

*Figure 5.9: Edit core-site.xml*

The `core-site.xml` file is located in the `conf` directory of the Hadoop installation and is read by various Hadoop components, including the Hadoop Common library. It defines common configuration parameters that are applicable across different Hadoop modules.

9. Edit `hdfs-site.xml` using the command:

```
sudo gedit hadoop-3.2.4/conf/hdfs-site.xml
```



The screenshot shows a text editor window with the title bar "hdfs-site.xml \*". The main editor area contains the XML code for `hdfs-site.xml`:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>
<property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
<property>
    <name>dfs.permissions</name>
    <value>false</value>
  </property>
</configuration>
```

*Figure 5.10: Edit hdfs-site.xml*

**hdfs-site.xml** is another important configuration file used in Apache Hadoop. It specifically contains configuration settings related to the HDFS, which is a key component of Hadoop responsible for storing data across a distributed cluster. This file is located in the **conf** directory of the Hadoop installation.

10. Edit **mapred-site.xml**:

Command: **sudo gedit hadoop-3.2.4/conf/mapred-site.xml**



```
*mapred-site.xml * 
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

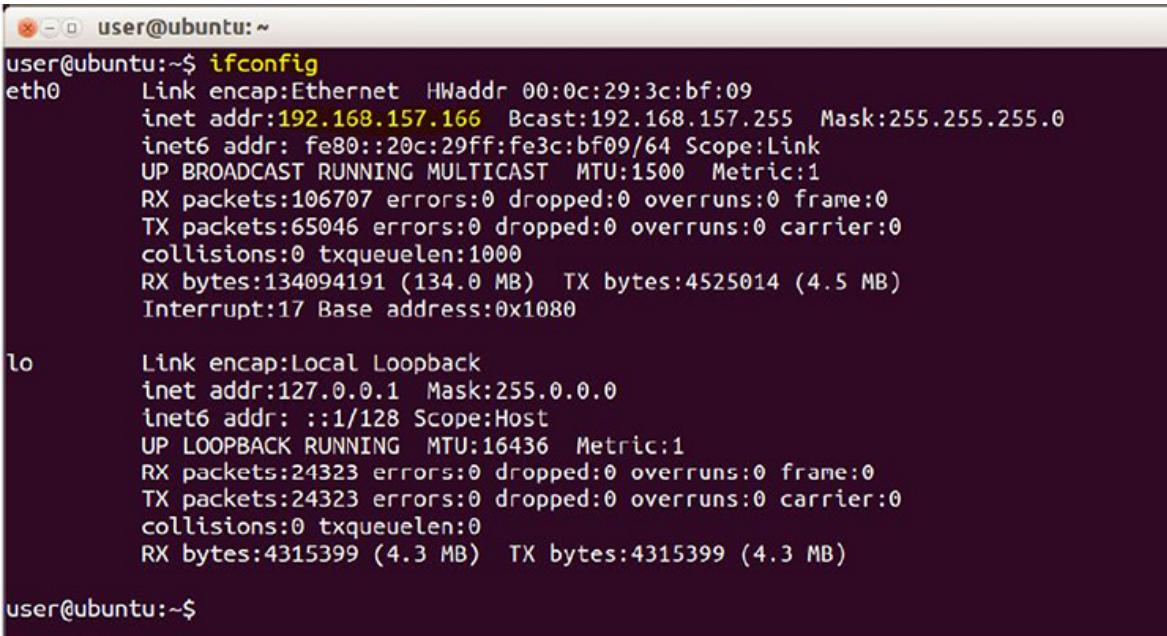
<!-- Put site-specific property overrides in this file. -->

<configuration>
<property>
    <name>mapred.job.tracker</name>
    <value>localhost:8021</value>
</property>
</configuration>
```

*Figure 5.11: Edit mapred-site.xml*

**mapred-site.xml** is another important configuration file used in Apache Hadoop. It contains configuration settings related to Hadoop's MapReduce framework, which is a core component responsible for distributed data processing and job execution in Hadoop.

11. Get your IP address using **ifconfig** and **sudo gedit /etc/host** ([Figure 5.12](#)):



```
user@ubuntu:~$ ifconfig
eth0      Link encap:Ethernet HWaddr 00:0c:29:3c:bf:09
          inet addr:192.168.157.166 Bcast:192.168.157.255 Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe3c:bf09/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:106707 errors:0 dropped:0 overruns:0 frame:0
          TX packets:65046 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:134094191 (134.0 MB) TX bytes:4525014 (4.5 MB)
          Interrupt:17 Base address:0x1080

lo        Link encap:Local Loopback
          inet addr:127.0.0.1 Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING MTU:16436 Metric:1
          RX packets:24323 errors:0 dropped:0 overruns:0 frame:0
          TX packets:24323 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:4315399 (4.3 MB) TX bytes:4315399 (4.3 MB)

user@ubuntu:~$
```

*Figure 5.12: ifconfig*

The command **ifconfig** is used to display network interface configuration information on Unix-like operating systems, including Linux. However, as of September 2021, the **ifconfig** command is considered deprecated in many Linux distributions in favor of the **ip** command, which provides more advanced networking functionalities.

### Usage:

Open a terminal or command prompt and simply type **ifconfig** followed by the Enter key.

### Output:

The **ifconfig** command displays information about all active network interfaces on your system, including Ethernet interfaces, wireless interfaces, and loopback interfaces. Each network interface is represented by a section with details such as:

- Interface name (**eth0**, **wlan0**)
- MAC address (hardware address)
- IP address assigned to the interface
- Network mask (subnet mask)

- Broadcast address (for IPv4)
- Status (eUP, DOWN) indicating whether the interface is enabled or disabled
- **Maximum Transmission Unit (MTU)**, which determines the maximum size of data packets that can be sent over the interface

We need to add all the IP addresses which are required for the processing as shown in *Figure 5.13*:



*Figure 5.13: Add all working machines using ip addresses*

12. To create a ssh key use the command, **ssh-keygen -t rsa -P ""**

Move the key to the authorized key using the command **cat \$HOME/.ssh/id\_rsa.pub >> \$HOME/.ssh/authorized\_keys**.

13. Reboot the system after completion of all installation commands and prepare for configuring Java Home.

Add **JAVA\_HOME** in hadoop-env.sh file. The command used is **sudo gedit hadoop-3.2.4/conf/hadoop-env.sh**.

Type: **export JAVA\_HOME= /usr/lib/jvm/java-8-openjdk-i386**

Uncomment the **java\_home** command line which is by default in the configuration file. After uncommenting this line system will get information where java is installed.

Java path needs to be shown for all MapReduce functions of Hadoop. *Figure 5.14* shows the path of Java programming:

```
*hadoop-env.sh *  
# Set Hadoop-specific environment variables here.  
  
# The only required environment variable is  
JAVA_HOME. All others are  
# optional. When running a distributed  
configuration it is best to  
# set JAVA_HOME in this file, so that it is  
correctly defined on  
# remote nodes.  
  
# The java implementation to use. Required.  
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-i386  
  
# Extra Java CLASSPATH elements. Optional.  
# export HADOOP_CLASSPATH=
```

Figure 5.14: Java path

14. Format the name node by **bin/hadoop namenode-format**. To start the **namenode**, **datanode**, use **bin/start-dfs.sh**.

To start the task tracker and job tracker, use **bin/start-mapred.sh**. To check if Hadoop started correctly, use **jps** (Java Virtual Machine Process Status).

In Hadoop, the **jps** command is a utility provided by Java to list all Java processes running on a machine. It is particularly useful for monitoring Hadoop daemons and verifying whether they are running as expected.

Here is how to use the **jps** command in Hadoop:

- **Running jps command:** Open a terminal or command prompt on the machine where Hadoop is installed and running. Simply type **jps** and press *Enter*. The command will list all Java processes along with their respective Process IDs (PIDs) and process names.
- **Hadoop daemons:** When Hadoop is running, it typically involves several daemons, such as the **NameNode**, **DataNode**, **ResourceManager**, **NodeManager**, Secondary **NameNode** (if configured), and others. Each daemon is represented by a corresponding Java process when viewed using **jps**.

- **Process names:** The process names shown by **jps** are helpful in identifying which Hadoop daemon each process corresponds to. For example, the process names usually include **NameNode**, **DataNode**, **ResourceManager**, **NodeManager**, **SecondaryNameNode**, and more.
- **Checking Hadoop status:** By using **jps**, you can quickly verify if all the necessary Hadoop daemons are running. If any daemon is missing, it indicates a potential issue with the Hadoop startup or configuration.

Here is an example output of the **jps** command for a running Hadoop cluster.

*Figure 5.15* shows the output that the **NameNode**, **DataNode**, **ResourceManager**, **NodeManager**, and **SecondaryNameNode** are running:

```
user@ubuntu:~$ jps
10117 TaskTracker
9802 SecondaryNameNode
9370 NameNode
9903 JobTracker
9587 DataNode
10449 Jps
user@ubuntu:~$
```

*Figure 5.15: jps*

Please note that the **jps** command may not be available if your Java installation does not include the JDK or if the **JAVA\_HOME** environment variable is not set correctly. If **jps** is not available, you can still check the status of Hadoop daemons through other means, such as using Hadoop-specific scripts or tools provided by your Hadoop distribution.

## Installing Hadoop using Oracle VirtualBox

Following are the set-ups to install Hadoop using Oracle VirtualBox:

1. Download Oracle VirtualBox from <http://www.oracle.com/technetwork/server-storage/virtualbox/downloads/index.html>, as shown in *Figure 5.16*:

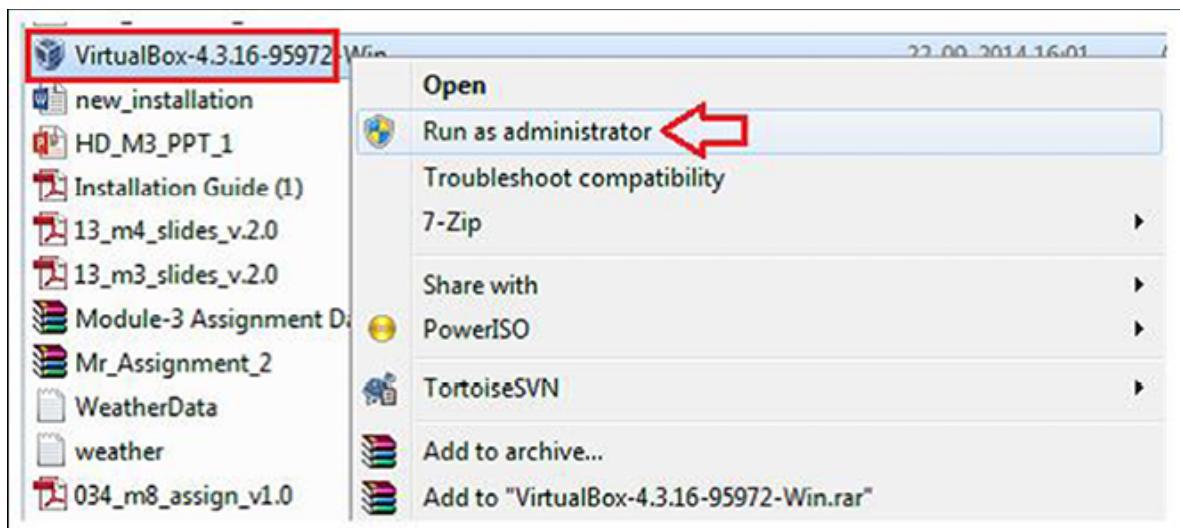
Oracle VM VirtualBox	
Freely available for Windows, Mac OS X, Linux and Solaris x-86 platforms:	
Platform	File
Windows (32-bit/64-bit)	VirtualBox-4.3.16-95972-Win.exe
Mac OS X	VirtualBox-4.3.16-95972-OSX.dmg
Solaris 10 5/08 and later (64-bit)	VirtualBox-4.3.16-95972-SunOS.tar.gz
Linux 32-bit Platforms	For Linux 32 bit
Ubuntu 13.04 (Raring Ringtail), 13.10 (Saucy Salamander), 14.04 (Trusty Tahr)	virtualbox-4.3_4.3.16-95972~Ubuntu~raring_i386.deb
Ubuntu 12.10 (Quantal Quetzal)	virtualbox-4.3_4.3.16-95972~Ubuntu~quantal_i386.deb
Ubuntu 12.04 (Precise Pangolin)	virtualbox-4.3_4.3.16-95972~Ubuntu~precise_i386.deb
Ubuntu 10.04 LTS (Lucid Lynx)	virtualbox-4.3_4.3.16-95972~Ubuntu~lucid_i386.deb

**Figure 5.16:** Installation using Oracle virtual box

Another method is to use oracle virtual box to install all deamons of Hadoop.

## 2. Run the setup.

**Figure 5.17** will reflect the folder where VirtualBox will be present. Install it as an administrator of system:



*Figure 5.17: Installation*

- a. Click **Next**.

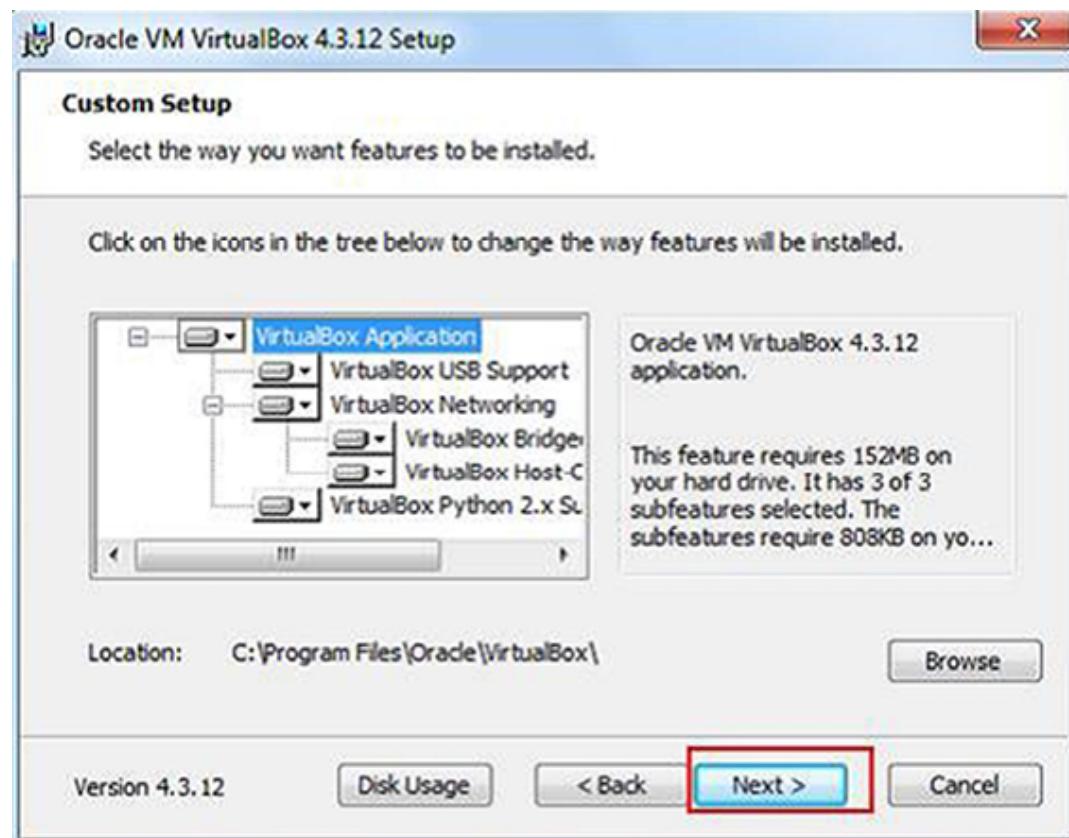
*Figure 5.18* shows the setup file of virtual box:



*Figure 5.18: Setup*

b. Install it to the desired location.

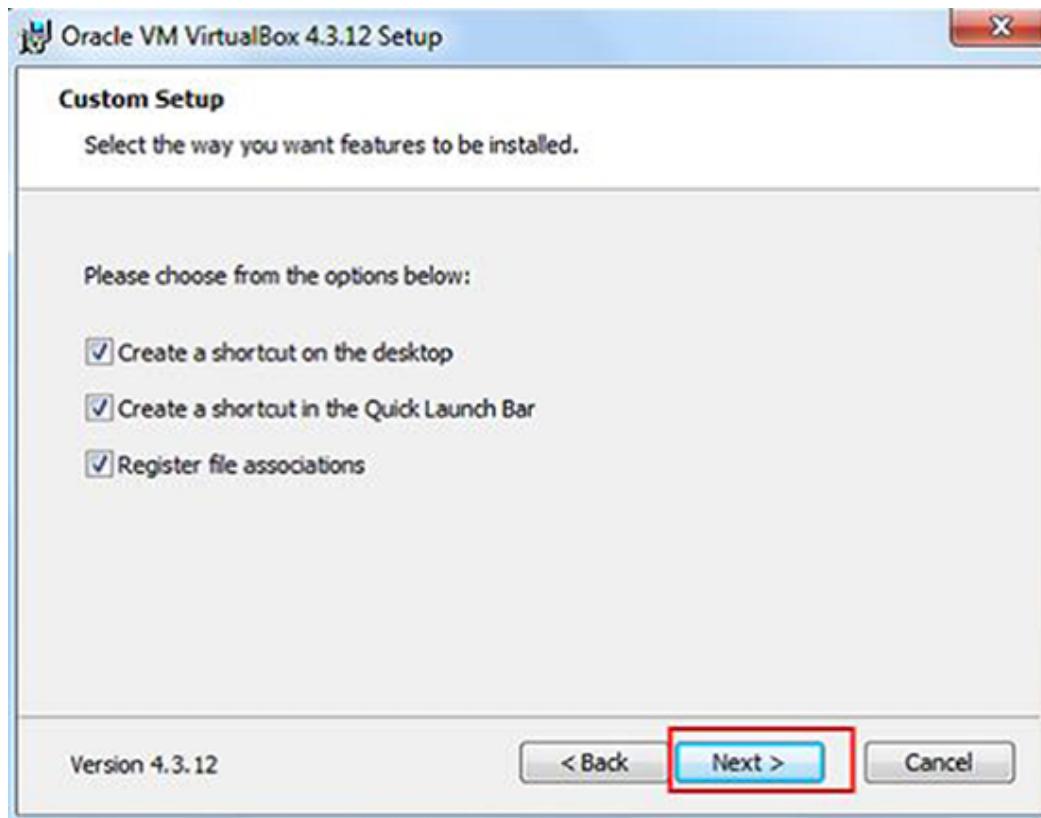
Keep clicking on **Next** to install virtual box properly as shown in *Figure 5.19*:



*Figure 5.19: Setup steps*

c. Click **Next** and continue.

*Figure 5.20* shows the custom setup which means we can change the settings manually also. Here it not required so we keep as it is:



*Figure 5.20: Setup Virtual box*

d. Click **Yes** in VM Virtual Box.

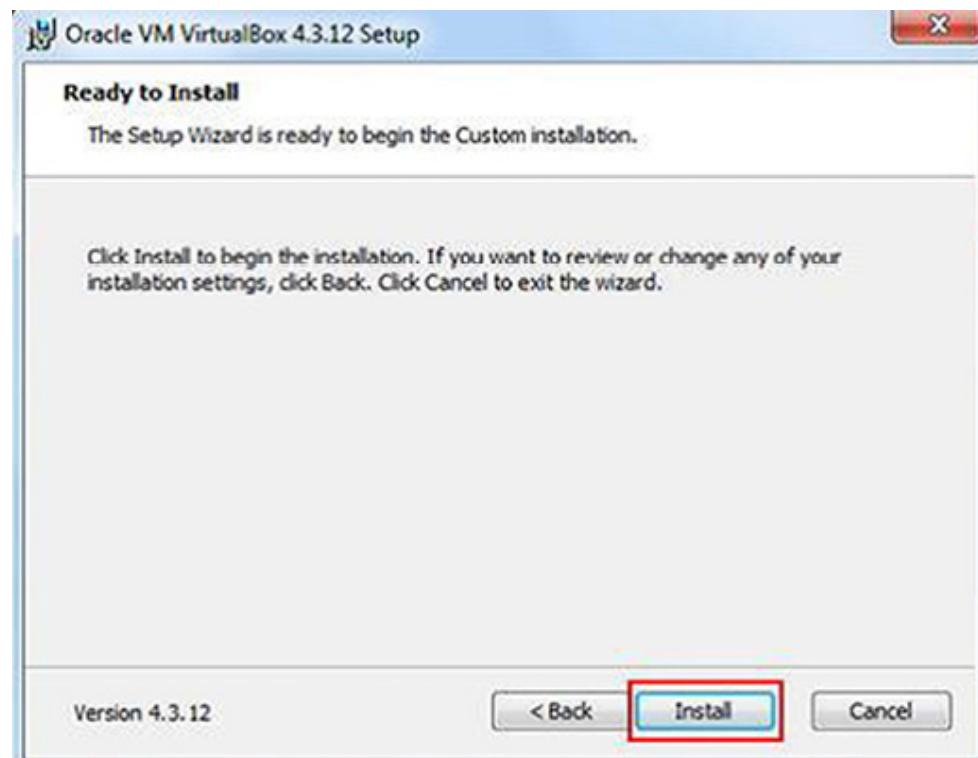
*Figure 5.21* asks for the network setup, so we need to be connected with internet and click on **Yes** button:



*Figure 5.21: Setup*

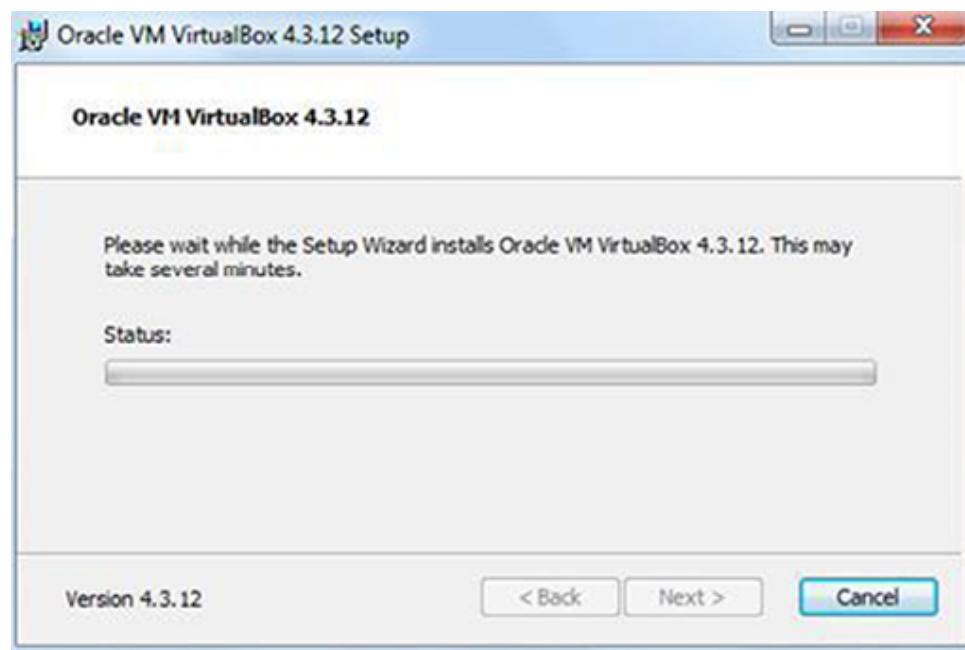
- e. Click **Install** to begin installation.

*Figure 5.22* shows the last step to configure the virtual box setup. By clicking on install, all packages will be installed automatically:



*Figure 5.22: Setup install*

*Figure 5.23* will show the process of setup:



*Figure 5.23: Process of setup*

To proceed with the installation, you will need to click the **Install** button on the security pop-up. This action is usually required to confirm your consent for the software to be installed on your computer.

When you click the **Install** button, the installation process will continue, and the software will be installed on your system. This is a standard security measure implemented by operating systems to ensure that you are aware of and authorize the installation of new software. *Figure 5.24* shows the virtual box of oracle which has the pre-install configuration files by Hadoop:

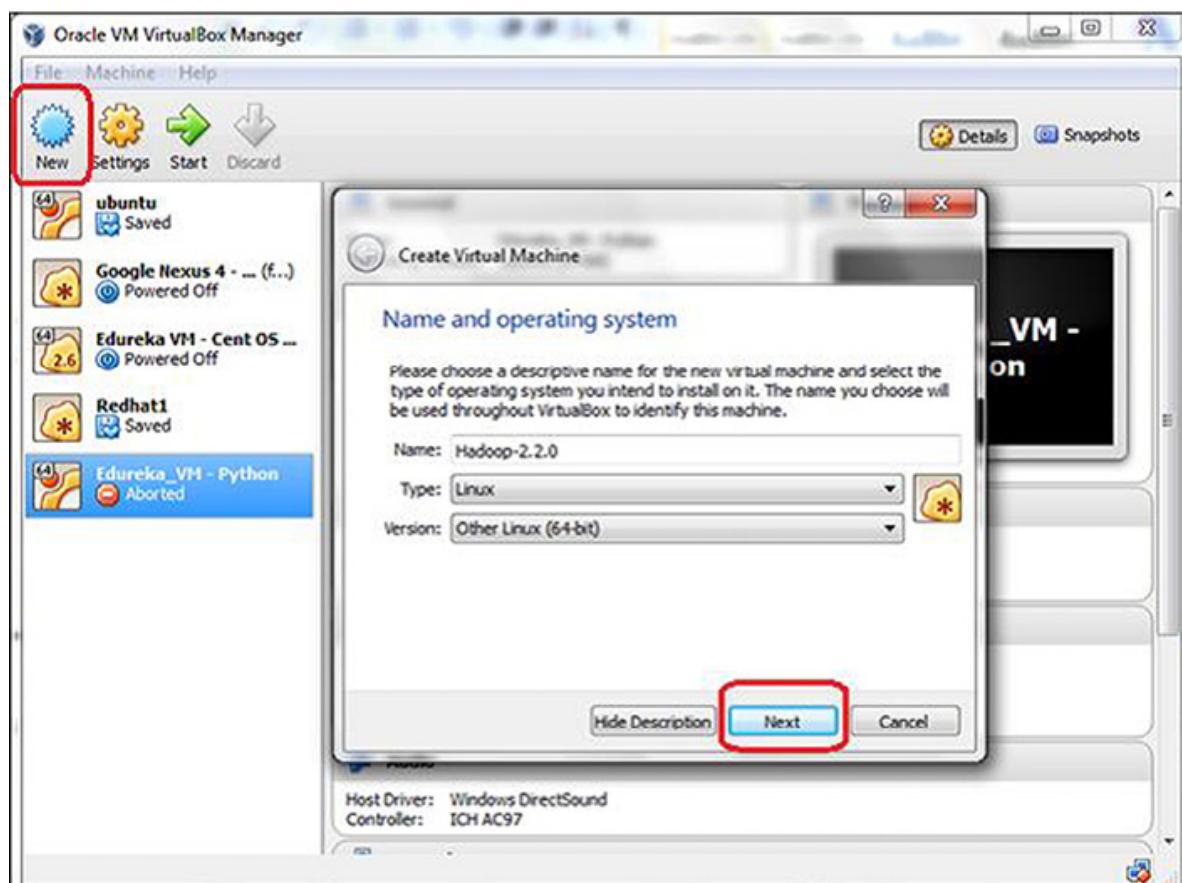


*Figure 5.24: VirtualBox*

3. Download Hadoop file that is compatible with Oracle VM virtual box using the following link:

<http://sourceforge.net/projects/centos-6-vmware/files/centos-6-x86-64-vmware-image/centos-6.2-x64-virtual-machine-org.zip/download>

4. Configure Hadoop on Virtual Box and select the **Ram size** that you want to allocate in virtual box:

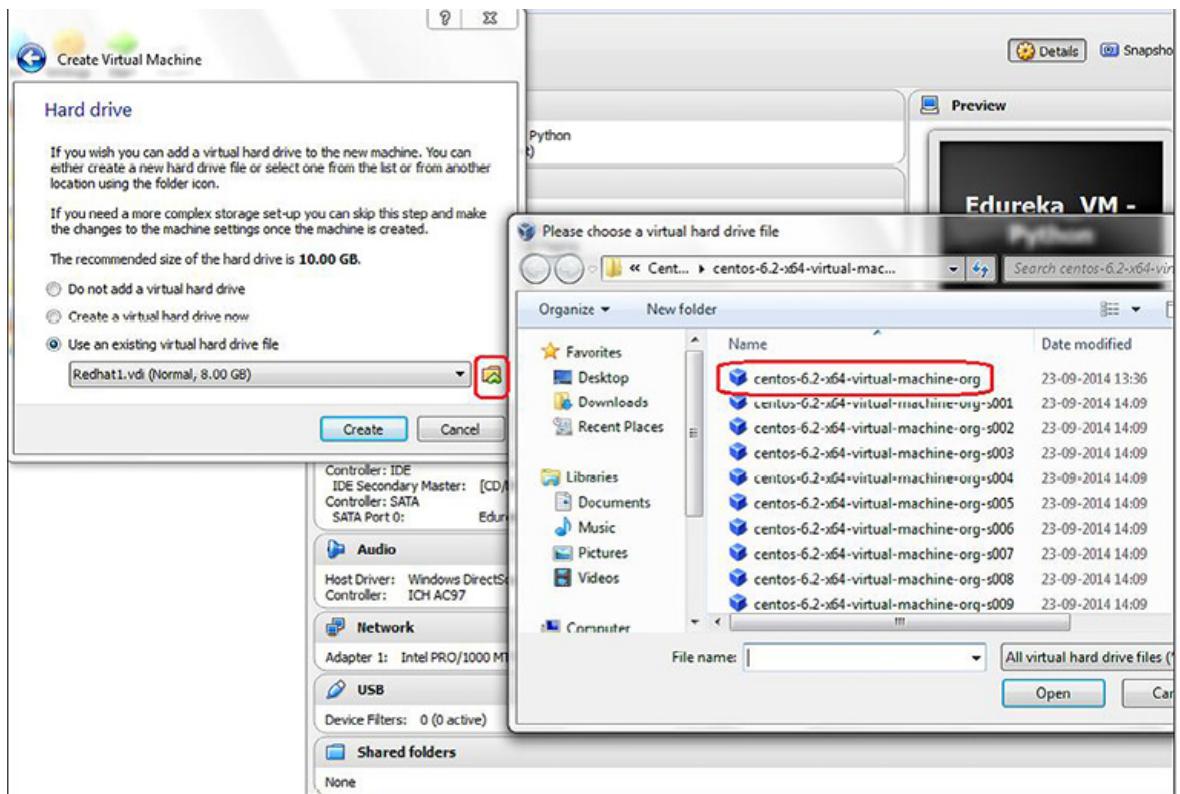


**Figure 5.25:** After VirtualBox imports Linux version

After installing VirtualBox, we need to install the Linux version.

5. Import CentOS file. CentOS is the Linux version which file needs to be imported from virtual box. We can assign RAM accordingly to it.

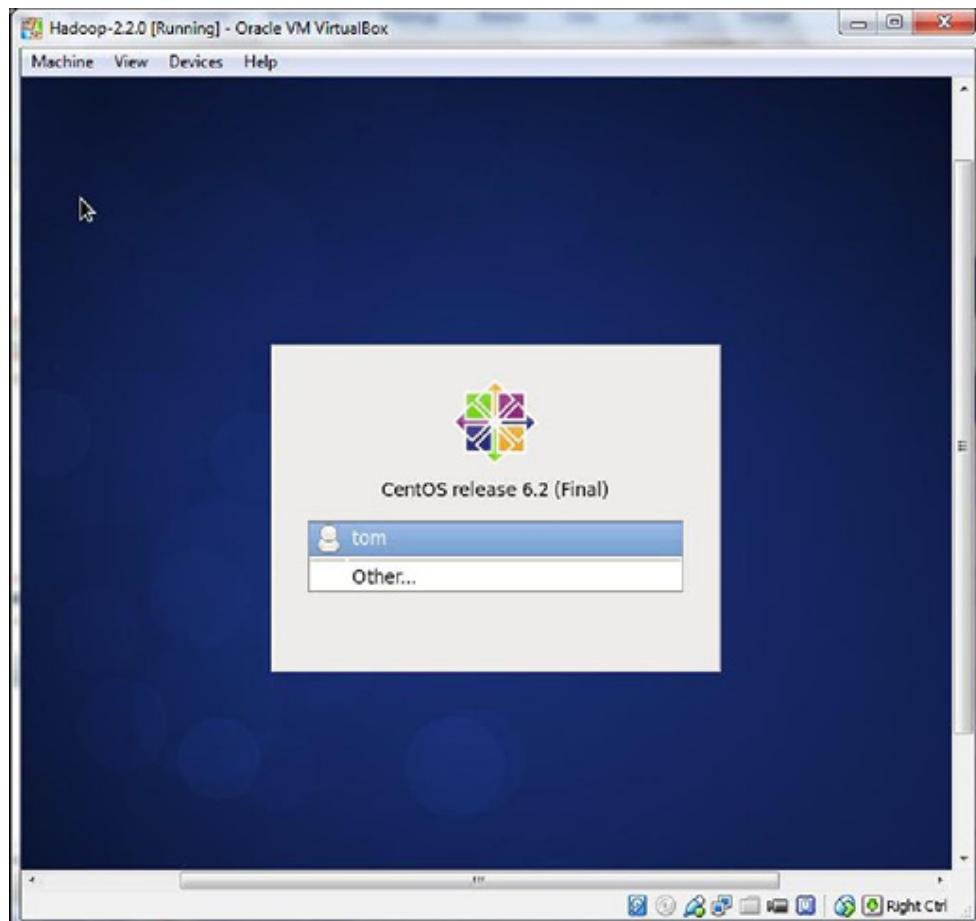
**Figure 5.26** will show the folder which is needed for importing the CentOS file from local machine:



**Figure 5.26:** Choose CentOS file

6. Start the installation by using **tom** as the username and **tomtom** as the password.

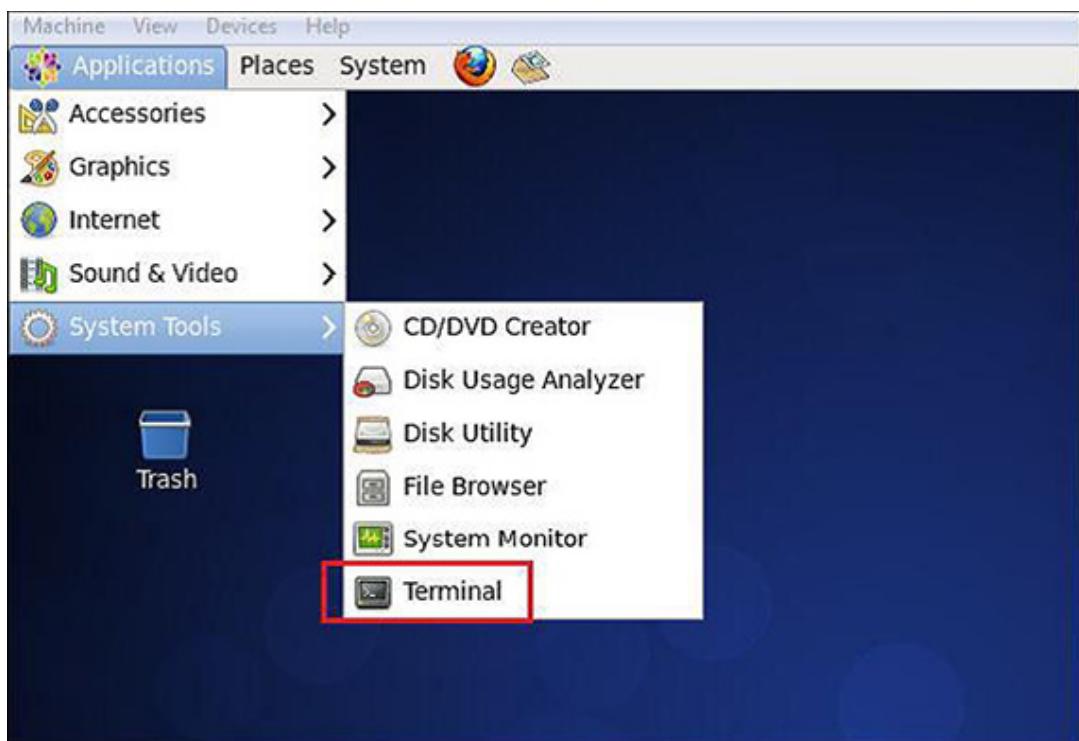
*Figure 5.27* shows the installation screen of CentOS:



*Figure 5.27: Installation screen*

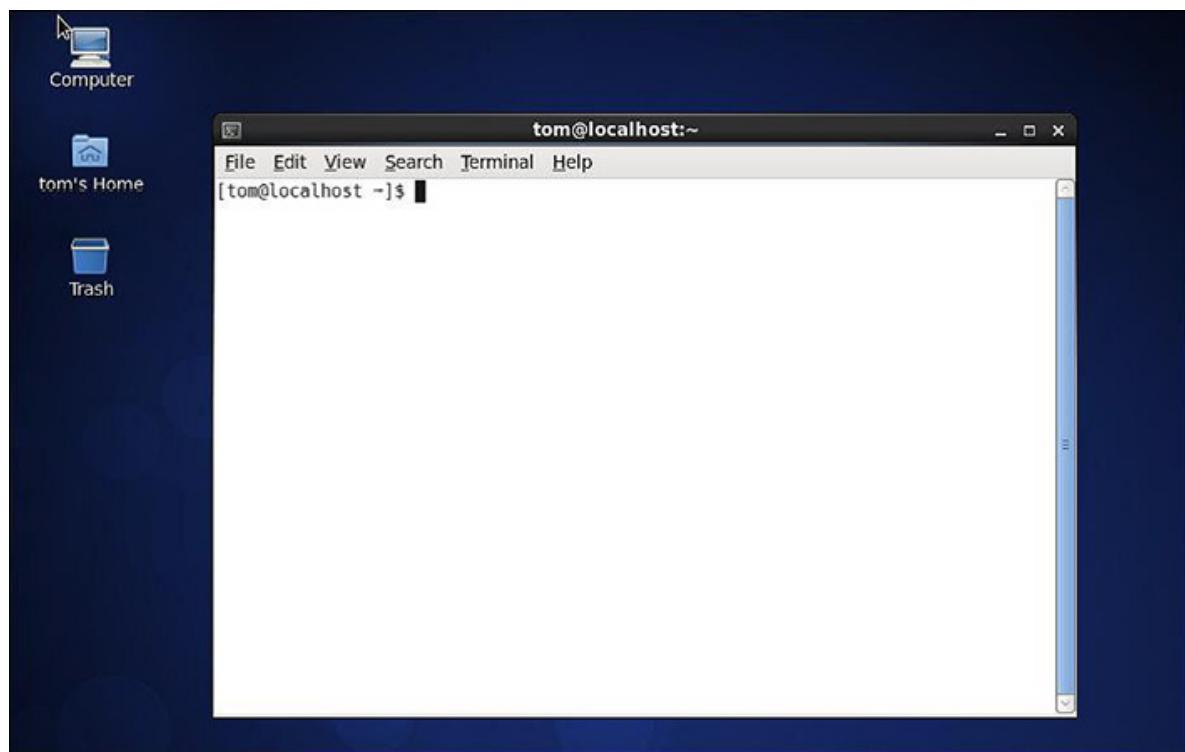
7. Open the terminal.

*Figure 5.28* shows the terminal for providing command for file configuration:



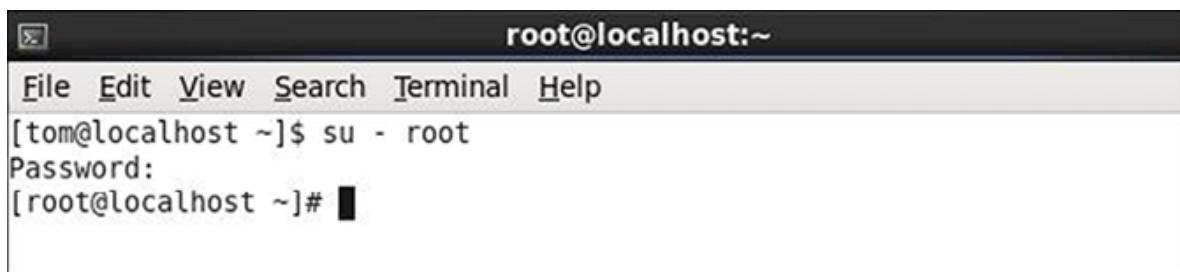
**Figure 5.28:** Terminal

[Figure 5.29](#) shows the terminal screen, here tom is the username and it is running on localhost:



**Figure 5.29:** Terminal Screen

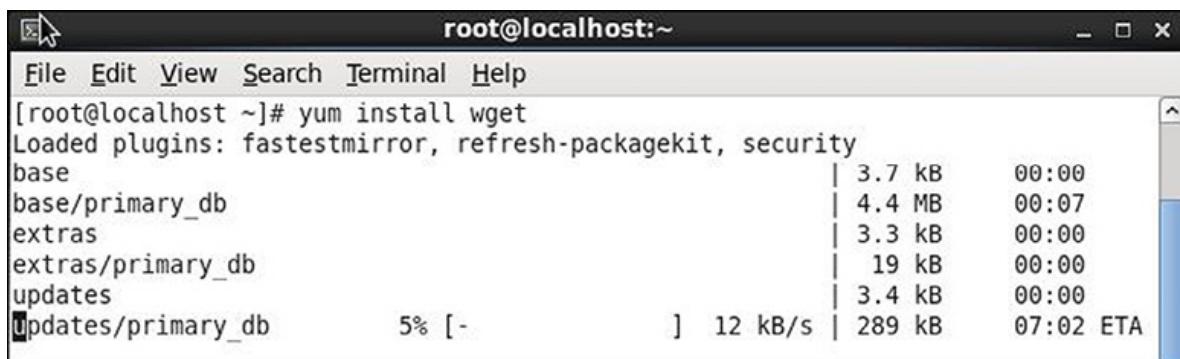
8. Log in **root** user by **#su - root** as shown in [Figure 5.30](#):



A terminal window titled "root@localhost:~". The menu bar includes File, Edit, View, Search, Terminal, and Help. The command line shows [tom@localhost ~]\$ su - root, followed by a password prompt, and finally [root@localhost ~]#.

**Figure 5.30:** Switching from user to admin

Root is the highest directory in Linux system so before installation we need to switch profile from user to root:



A terminal window titled "root@localhost:~". The menu bar includes File, Edit, View, Search, Terminal, and Help. The command line shows [root@localhost ~]# yum install wget. The output lists several packages being downloaded, including base, base/primary\_db, extras, extras/primary\_db, updates, and updates/primary\_db. The progress shows 5% download at 12 kB/s with an ETA of 07:02.

**Figure 5.31:** Commands

The command **yum install wget** ([Figure 5.31](#)) is used to install the **wget** package on Linux distributions that use the **Yellowdog Updater Modified (YUM)** package manager. YUM is commonly used in **Red Hat Enterprise Linux (RHEL)**, CentOS, and Fedora, among others.

Wget is a command-line utility used to download files from the internet. It supports downloading files using various protocols, such as HTTP, HTTPS, FTP, and others.

9. Configure JDK and set the path using: # wget --no-cookies --no-check-certificate --header "Cookie: gpw\_e24=http%3A%2F%2Fwww.oracle.com%2F; oraclelicense=accept-securebackup-cookie"

**<http://download.oracle.com/otn-pub/java/jdk/7u65-b17/jdk-7u65-linux-x64.tar.gz>**

To install certificate for security, we need to install through **wget** command as shown in *Figure 5.32*:



The screenshot shows a terminal window titled "root@localhost:~". The user has run the command "wget --no-cookies --no-check-certificate --header "Cookie: g\_pw\_e24=http%3A%2F%2Fwww.oracle.com%2F; oraclelicense=accept-securebackup-cookie" "http://download.oracle.com/otn-pub/java/jdk/7u65-b17/jdk-7u65-linux-x64.tar.gz". The output shows the download process, including a redirect from Oracle's main site to their delivery site, and the final successful download of the Java tar file.

```
[root@localhost ~]# wget --no-cookies --no-check-certificate --header "Cookie: g_pw_e24=http%3A%2F%2Fwww.oracle.com%2F; oraclelicense=accept-securebackup-cookie" "http://download.oracle.com/otn-pub/java/jdk/7u65-b17/jdk-7u65-linux-x64.tar.gz"
--2014-09-23 11:26:35-- http://download.oracle.com/otn-pub/java/jdk/7u65-b17/jdk-7u65-linux-x64.tar.gz
Resolving download.oracle.com... 125.56.222.17, 125.56.222.11
Connecting to download.oracle.com[125.56.222.17]:80... connected.
HTTP request sent, awaiting response... 302 Moved Temporarily
Location: https://edelivery.oracle.com/otn-pub/java/jdk/7u65-b17/jdk-7u65-linux-x64.tar.gz [following]
--2014-09-23 11:26:36-- https://edelivery.oracle.com/otn-pub/java/jdk/7u65-b17/jdk-7u65-linux-x64.tar.gz
```

*Figure 5.32: To install security certificate*

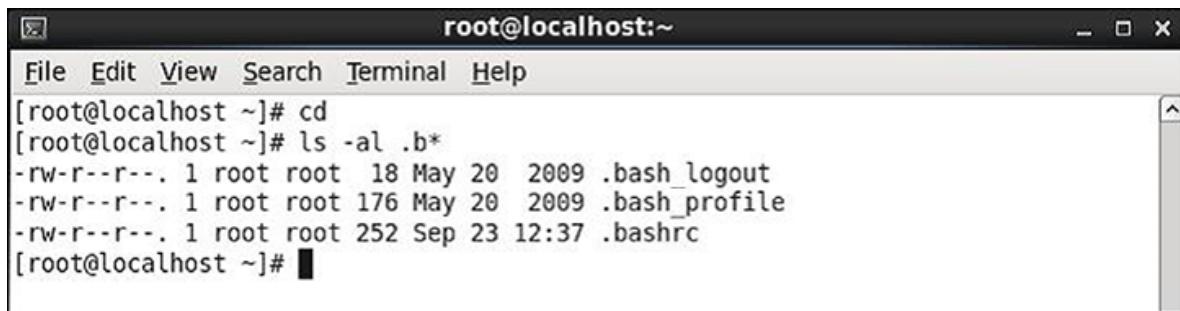
#### 10. Set the path of **.bashrc**.

The **.bashrc** file is a script file used in Unix-like operating systems, including Linux and macOS, to configure the behavior of the Bash shell for individual users. Bourne Again Shell (Bash) is a popular command-line shell and scripting language used in many Unix-based systems.

When a user logs into the system or starts a new interactive Bash session, the **.bashrc** file is automatically executed, allowing users to customize their shell environment. It is typically located in the user's home directory (**~/**) and starts with a dot (**.**), making it a hidden file. You can view and edit it using a text editor, such as nano or vim.

The primary purpose of the **.bashrc** file is to set environment variables, define aliases, customize the prompt appearance, and add functions or scripts that should be available in the shell session.

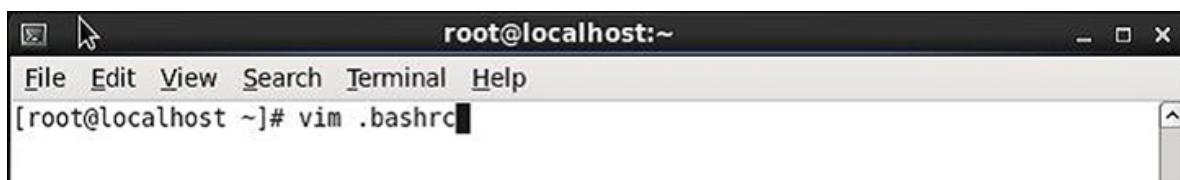
*Figure 5.33* shows the settings to change the directory.



```
root@localhost:~  
File Edit View Search Terminal Help  
[root@localhost ~]# cd  
[root@localhost ~]# ls -al .b*  
-rw-r--r--. 1 root root 18 May 20 2009 .bash_logout  
-rw-r--r--. 1 root root 176 May 20 2009 .bash_profile  
-rw-r--r--. 1 root root 252 Sep 23 12:37 .bashrc  
[root@localhost ~]#
```

**Figure 5.33:** Change directory

[Figure 5.34](#) shows the configuration file:



```
root@localhost:~  
File Edit View Search Terminal Help  
[root@localhost ~]# vim .bashrc
```

**Figure 5.34:** Configuration file

Opening the **.bashrc** file using the vim text editor allows you to edit and customize your Bash shell environment.

```
Export JAVA_HOME= /usr/lib/jvm/jdk1.7.0_65
```

```
Export PATH=$PATH:$JAVA_HOME/bin
```

To set the **JAVA\_HOME** environment variable and update the **PATH** variable to include the Java **bin** directory. When setting environment variables in the shell, there should be no space between the variable name, the equal sign (=), and the variable value.

[Figure 5.35](#) shows the configuration of Java path installed in the machine:



The screenshot shows a terminal window titled "root@localhost:~". The window contains the contents of the .bashrc file. The code includes aliases for rm, cp, and mv, sourcing /etc/bashrc if it exists, and setting JAVA\_HOME and PATH environment variables.

```
# .bashrc
#
# User specific aliases and functions

alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'

# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi

export JAVA_HOME=/usr/lib/jvm/jdk1.7.0_65
export PATH=$PATH:$JAVA_HOME/bin
~
```

**Figure 5.35:** Configuration of Java path

11. Update the configuration files /usr/local/hadoop/etc/hadoop/hdfs-site.xml:

- `~/.bashrc` :

To configure Hadoop environment variables, open the `.bashrc` file located in your `home` directory and add the following lines:

```
#HADOOP VARIABLES START

export JAVA_HOME=/usr/lib/jvm/java-7-openjdk-
amd64

export HADOOP_INSTALL=/usr/local/hadoop

export PATH=$PATH:$HADOOP_INSTALL/bin

export PATH=$PATH:$HADOOP_INSTALL/sbin

export HADOOP_MAPRED_HOME=$HADOOP_INSTALL

export HADOOP_COMMON_HOME=$HADOOP_INSTALL

export HADOOP_HDFS_HOME=$HADOOP_INSTALL

export YARN_HOME=$HADOOP_INSTALL

export

HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_INSTALL/lib
```

```
/native  
export HADOOP_OPTS="-  
Djava.library.path=$HADOOP_INSTALL/lib"  
  
#HADOOP VARIABLES END
```

Before modifying **.bashrc** file, ensure you have correctly set the path where Java has been installed to the **JAVA\_HOME** environment variable and then save the path:

- **/usr/local/hadoop/etc/hadoop/hadoop-env.sh :**  
Export JAVA\_HOME=/usr/lib/jvm/java-8-openjdk-amd64
- **/usr/local/hadoop/etc/hadoop/core-site.xml :**

The configuration properties used by Hadoop during startup are stored in the **/usr/local/hadoop/etc/hadoop/core-site.xml** file. This file can be edited to modify the default settings of Hadoop. To modify the settings, open the file and enter the following code between the **<configuration></configuration>** tags:

```
<configuration>  
  <property>  
    <name>hadoop.tmp.dir</name>  
    <value>/app/hadoop/tmp</value>  
    <description>A base for other temporary  
directories.</description>  
  </property>  
  <property>  
    <name>fs.default.name</name>  
    <value>hdfs://localhost:54310</value>  
    <description> A URI whose
```

```
        scheme and authority determine the
FileSystem implementation. The
        uri's scheme determines the config
property (fs.SCHEME.impl) naming
        the FileSystem implementation class. The
uri's authority is used to
        determine the host, port, etc. for a
filesystem.</description>
</property>
</configuration>
```

This code specifies two properties, the first sets the base for other **temporary** directories to **/app/hadoop/tmp**, while the second sets the default file system name to **hdfs://localhost:54310**.

- **/usr/local/hadoop/etc/hadoop/mapred-site.xml** :

To configure MapReduce framework in Hadoop, follow these steps:

By default, the **/usr/local/hadoop/etc/hadoop/** folder contains the **mapred-site.xml.template** file which needs to be renamed/copied with the name **mapred-site.xml**.

Open the file **/usr/local/hadoop/etc/hadoop/mapred-site.xml** and enter the following content in between the **<configuration>** **</configuration>** tag:

```
<configuration>
<property>
    <name>mapred.job.tracker</name>
    <value>localhost:54311</value>
    <description>The host and port that the
MapReduce job tracker runs
```

at. If "local", then jobs are run in-process as a single map and reduce task.

```
</description>  
</property>  
</configuration>
```

This specifies the hostname and port number where the MapReduce job tracker runs. If it is set to **local**, then the jobs will be executed in-process as a single map and reduce task.

- **/usr/local/adoop/etc/adoop/hdfs-site.xml:**

After configuring the **hdfs-site.xml** file, we need to format the new Hadoop file system. To do so, open the terminal and execute the command **adoop namenode -format**.

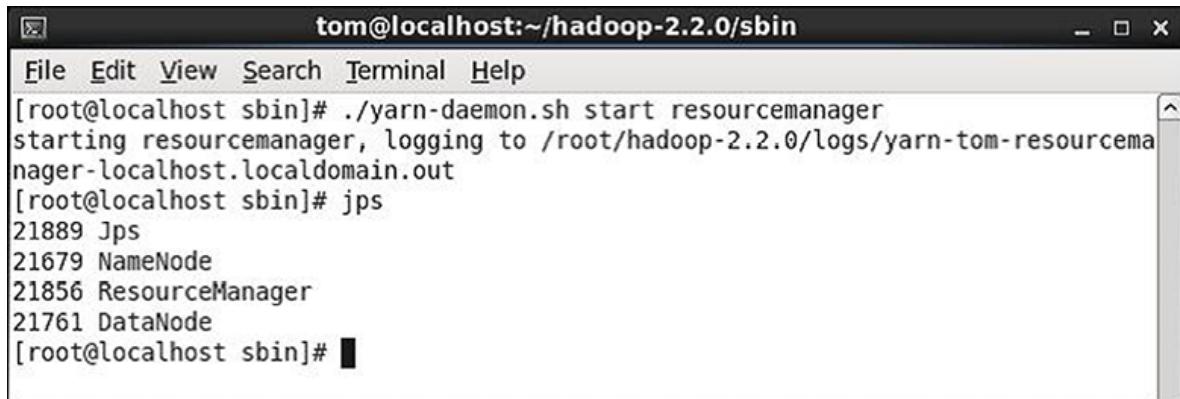
This command will initialize the namenode and format the Hadoop filesystem. It may take a few minutes to complete, depending on the size of the filesystem. Once the command has finished, we can start the Hadoop daemons by executing the command: **start-all.sh**.

This command will start all the Hadoop daemons, including the namenode, the datanodes, and the jobtracker. To verify that the Hadoop installation is working properly, we can open the web interface by entering the following URL in a web browser, **http://localhost:50070**.

This will display the Hadoop NameNode web interface, which shows the status of the Hadoop installation. We can use this interface to verify that the namenode and the datanodes are running and to monitor the progress of Hadoop jobs.

12. Start the yarn daemon (in Hadoop **Yet Another Resource Negotiator (YARN)**, a daemon refers to a long-running process that runs continuously in the background to provide a specific service. YARN is a component of Apache Hadoop responsible for resource management and job scheduling in a Hadoop cluster.)

*Figure 5.36* shows the yarn daemon which reflect the starting point:



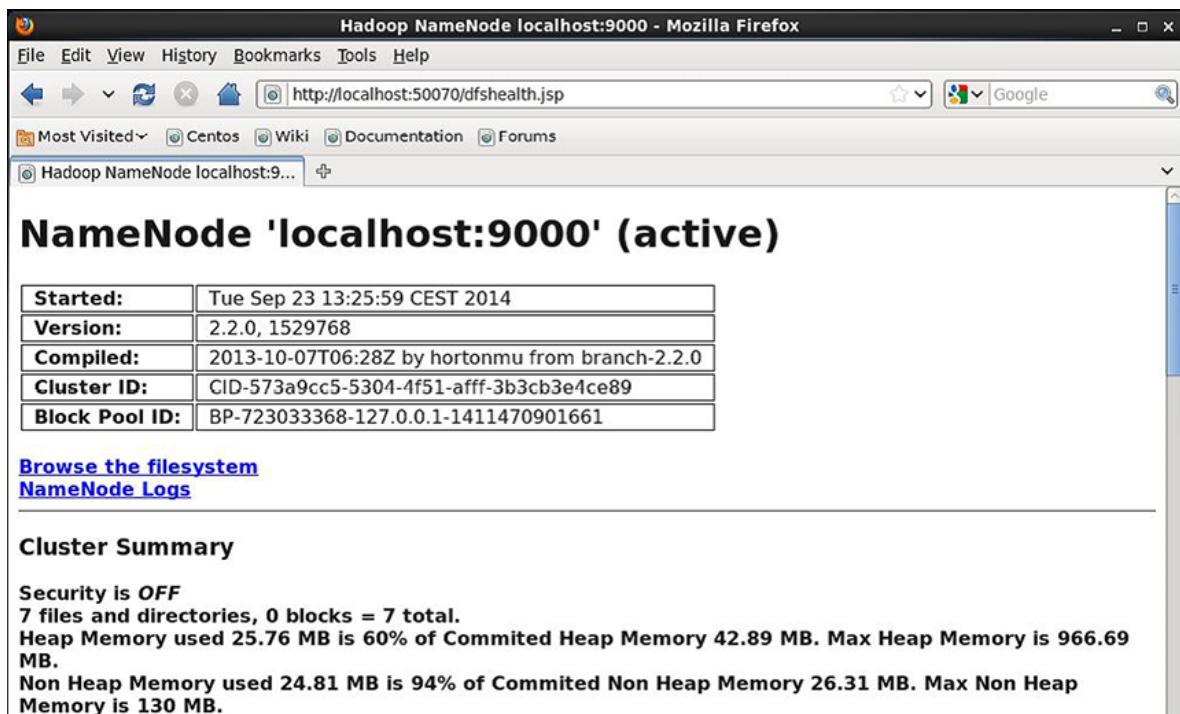
A terminal window titled "tom@localhost:~/hadoop-2.2.0/sbin". The window contains the following text:

```
[root@localhost sbin]# ./yarn-daemon.sh start resourcemanager
starting resourcemanager, logging to /root/hadoop-2.2.0/logs/yarn-tom-resourcemanager-localhost.localdomain.out
[root@localhost sbin]# jps
21889 Jps
21679 NameNode
21856 ResourceManager
21761 DataNode
[root@localhost sbin]#
```

*Figure 5.36: Yarn Daemon*

13. Check the **NameNode** status by typing following on URL **<http://localhost:50070/dfshealth.jsp>**.

*Figure 5.37* shows the status of name node using URL on screen in browser. It is reflected on a local machine:



A screenshot of a Mozilla Firefox browser window. The title bar says "Hadoop NameNode localhost:9000 - Mozilla Firefox". The address bar shows "http://localhost:50070/dfshealth.jsp". The page content is as follows:

## NameNode 'localhost:9000' (active)

Started:	Tue Sep 23 13:25:59 CEST 2014
Version:	2.2.0, 1529768
Compiled:	2013-10-07T06:28Z by hortonmu from branch-2.2.0
Cluster ID:	CID-573a9cc5-5304-4f51-afff-3b3cb3e4ce89
Block Pool ID:	BP-723033368-127.0.0.1-1411470901661

[Browse the filesystem](#)  
[NameNode Logs](#)

---

### Cluster Summary

Security is OFF  
7 files and directories, 0 blocks = 7 total.  
Heap Memory used 25.76 MB is 60% of Committed Heap Memory 42.89 MB. Max Heap Memory is 966.69 MB.  
Non Heap Memory used 24.81 MB is 94% of Committed Non Heap Memory 26.31 MB. Max Non Heap Memory is 130 MB.

*Figure 5.37: Status of Name node*

14. Check the job history.

*Figure 5.38* shows the running job status in local machine:

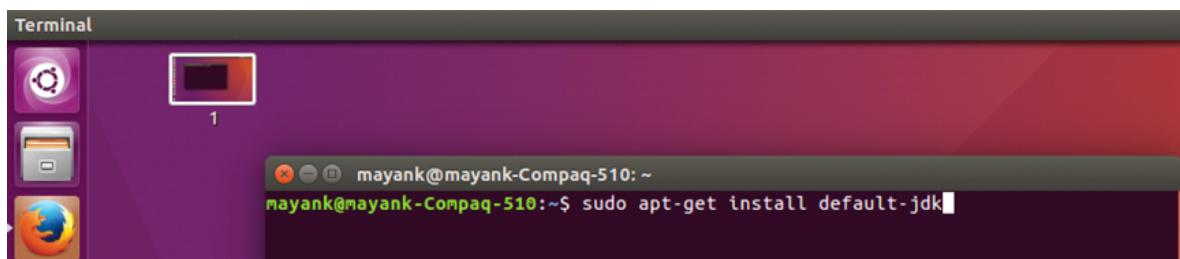
The screenshot shows a Mozilla Firefox window with the title "JobHistory - Mozilla Firefox". The address bar displays "http://localhost:19888/jobhistory". The main content area features the Hadoop logo and the title "JobHistory". On the left, there is a sidebar with a tree view under "Application" showing "About" and "Jobs" (which is currently selected). Below the sidebar is a "Tools" link. The main content area has a heading "Retired Jobs" and a table with the following columns: Start Time, Finish Time, Job ID, Name, User, Queue, State, Maps Total, Maps Completed, Reduces Total, and Reduces Completed. A search bar is at the top of the table area. Below the table, a message says "No data available in table". At the bottom of the table area, there are buttons for "First", "Previous", "Next", and "Last". The status bar at the bottom of the browser window shows "Showing 0 to 0 of 0 entries".

*Figure 5.38: Status of running job*

## On Ubuntu 16.04

Here are all commands that are required for installing Hadoop:

1. Update the system with Java. *Figure 5.39* shows the screenshot of JDK installation in machine:

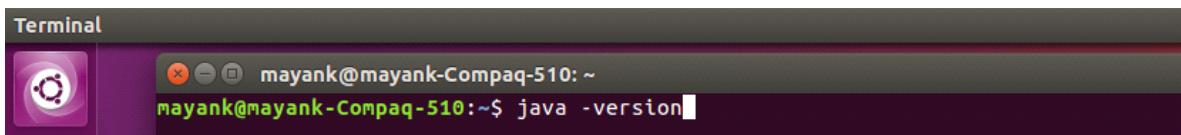


*Figure 5.39: jdk installation*

2. In the context of installing a Java version or any software system-wide, the reason for using **sudo** is to ensure that only authorized users with administrative privileges can make system-wide changes. Regular users typically have restricted permissions for **system** directories and configurations to prevent unintended modifications that could potentially disrupt the system's stability or security.

By using sudo, you are prompted to enter your password to verify your identity and authorization. If you have the necessary permissions, the command is then executed with elevated privileges. This mechanism ensures that sensitive actions are carried out only by authorized users and helps maintain system integrity.

*Figure 5.40* shows to check which version is running in system:

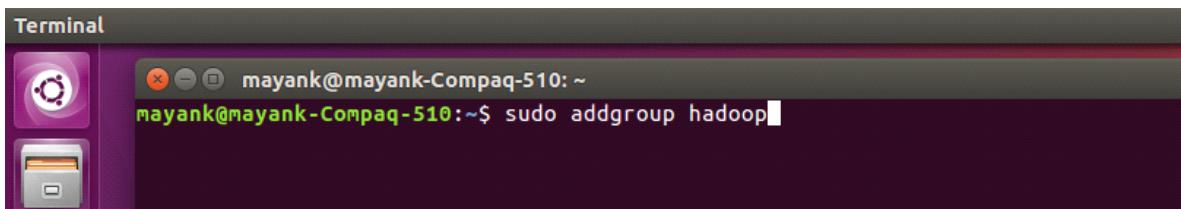


```
Terminal
mayank@mayank-Compaq-510: ~
mayank@mayank-Compaq-510:~$ java -version
```

*Figure 5.40: Check Java version*

3. Check the current Java version.

*Figure 5.41* shows the command by using users of machine add into group so that single command can be provided:

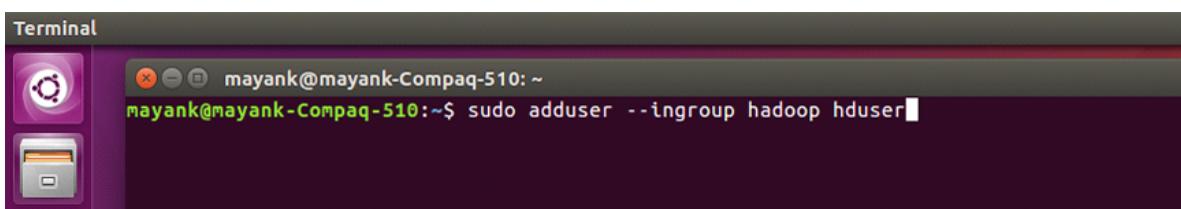


```
Terminal
mayank@mayank-Compaq-510: ~
mayank@mayank-Compaq-510:~$ sudo addgroup hadoop
```

*Figure 5.41: Make group of users*

4. Adding group **hadoop**.

This command is often used in Hadoop installations to create a dedicated user for running Hadoop services and managing Hadoop-related files as shown in *Figure 5.42*:



```
Terminal
mayank@mayank-Compaq-510: ~
mayank@mayank-Compaq-510:~$ sudo adduser --ingroup hadoop hduser
```

*Figure 5.42: Create dedicated user*

Let us break down the command:

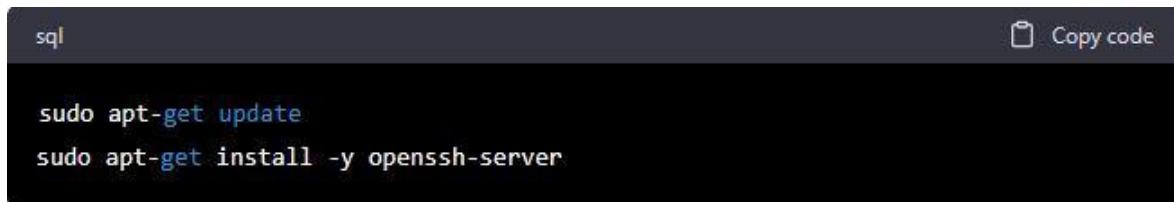
- **sudo**: It is used to execute the following command with administrative privileges. You will likely need to enter the password for the user account with **sudo** privileges to proceed with the user creation.
- **adduser**: This is a command used to add new users to the system.
- **--ingroup hadoop**: This option specifies the primary group for the new user. In this case, the user **hduser** will be added to the **hadoop** group.
- **hduser**: This is the username of the new user that you want to create. In this example, the user will be named **hduser**.

After running the command, the system will create a new user named **hduser** and add them to the **hadoop** group. The **home** directory for the new user will typically be created in the **/home/hduser** directory, and the user will be able to log in to the system using the password assigned during user creation.

Creating a separate user for running Hadoop services is a good security practice as it limits the access and privileges of Hadoop processes to only the necessary directories and resources. It also helps in organizing user accounts and access control in the Hadoop cluster.

Assuming you are using Ubuntu, you can install SSH and its server (sshd) using the following command (*Figure 5.43*):

```
sudo apt-get update
sudo apt-get install -y openssh-server
```



A screenshot of a terminal window. The title bar says "sql". The main area contains the following text:

```
sudo apt-get update
sudo apt-get install -y openssh-server
```

In the top right corner of the terminal window, there is a "Copy code" button with a clipboard icon.

*Figure 5.43: Update of SSH*

Let us break down each command:

**sudo apt-get update**:

This command updates the package list on your system. It contacts the package repositories and checks for the latest versions of packages available. It is a good idea to run this command before installing any new software to ensure you have the latest package information.

```
sudo apt-get install -y openssh-server:
```

This command installs the OpenSSH server package on your system. The **-y** option is used to automatically answer yes to any prompts that may come up during the installation process, which allows for a non-interactive installation.

After running these commands, your system will have the OpenSSH server installed and enabled, allowing you to accept incoming SSH connections. The OpenSSH server enables secure remote access to your system, which is useful for various purposes, including remote administration and file transfers.

This will update the package repository and install the OpenSSH server package. Once the installation is complete, the sshd daemon should start automatically. You can check its status using the following command: **systemctl status ssh** as shown in *Figure 5.44*:

```
systemctl status ssh
```

*Figure 5.44: Status of OpenSSH*

Let us breakdown the command:

- **systemctl**: It is a command-line utility used to control and manage **systemd** services on Linux systems.
- **status**: This is an option used with **systemctl** to display the status of a specific service.
- **ssh**: This is the name of the SSH service unit that **systemd** will check the status of.

If it is running, you should see an output similar to *Figure 5.45*:

```
● ssh.service - OpenBSD Secure Shell server
  Loaded: loaded (/lib/systemd/system/ssh.service; enabled; vendor preset: enabled)
  Active: active (running) since Tue 2023-03-28 10:00:00 UTC; 3s ago
    Main PID: 1234 (sshd)
       Tasks: 1 (limit: 1138)
      Memory: 1.1M
         CGroupl: /system.slice/ssh.service
                   └─1234 /usr/sbin/sshd -D
```

*Figure 5.45: Status of ssh services*

ssh.service - OpenBSD Secure Shell server

  Loaded: loaded  
  (/lib/systemd/system/ssh.service; enabled; vendor  
  preset: enabled)

  Active: active (running) since Tue 2023-03-28  
  10:00:00 UTC; 3s ago

  Main PID: 1234 (sshd)

  Tasks: 1 (limit: 1138)

  Memory: 1.1M

  CGroupl: /system.slice/ssh.service

    └─1234 /usr/sbin/sshd -D

Note that the SSH service is enabled by default on most Linux distributions, including Ubuntu, but it is always a good idea to check its status after installation to make sure it is running.

Adding user 'hduser'...

Adding new user 'hduser' with group 'hadoop'

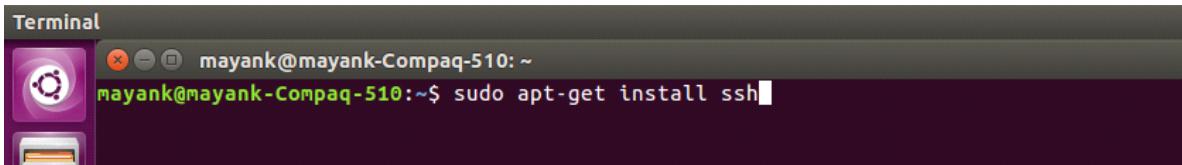
Creating home directory '/home/hduser'...

Copying files from '/etc/skel'...

Enter new UNIX password:

Retype new UNIX password:

*Figure 5.46* shows the installation of ssh if we are installing fresh Hadoop on system:

A screenshot of a Linux terminal window titled "Terminal". The window has a dark background and a light-colored title bar. In the title bar, it says "Terminal" and "mayank@mayank-Compaq-510: ~". Below the title bar, there is a purple icon of the Ubuntu logo. The main area of the terminal shows a command line interface. The command "mayank@mayank-Compaq-510:~\$ sudo apt-get install ssh" is visible, with the cursor at the end of the command.

*Figure 5.46: Install ssh*

##### 5. Create and setup SSH certificate:

Creating and setting up SSH certificates is essential when installing Hadoop for several crucial reasons. Firstly, Hadoop is a distributed computing framework that relies on secure and efficient communication among its various components, such as the **NameNode** and **DataNode**. SSH certificates enhance the security of these communications by employing strong encryption and authentication mechanisms. Moreover, using SSH certificates simplifies the authentication process, eliminating the need for password-based logins, which can be vulnerable to unauthorized access and brute-force attacks. This not only strengthens the security of your Hadoop cluster but also streamlines the management of multiple nodes. In summary, setting up SSH certificates for Hadoop installation is a fundamental step in ensuring the reliability, security, and efficiency of your distributed data processing environment.

*Figure 5.47* shows the generation of key in SSH using RSA algorithm:

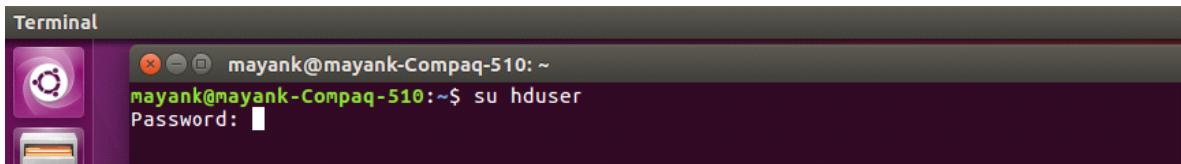
```
ssh-keygen -t rsa -P "" -f ~/.ssh/id_rsa
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
chmod 0600 ~/.ssh/authorized_keys
```

*Figure 5.47: Key generation*

The first command generates a new RSA key pair without a passphrase and saves it to the default location `~/.ssh/id_rsa`. The second command appends the public key to the `authorized_keys` file, which is used by SSH to authenticate the user. The third command sets the correct

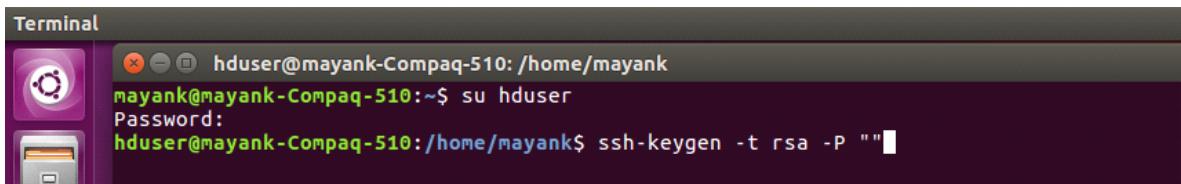
permissions on the **authorized\_keys** file to ensure that only the owner can read and write it.

*Figure 5.48* shows the switching user to **hduser** on which we are configuring the Hadoop system:



*Figure 5.48: Switch user*

*Figure 5.49* shows the key generation for ssh using RSA algorithm:



*Figure 5.49: Key generation on ssh*

These commands generated an SSH key pair. The private key is saved in **/home/hduser/.ssh/id\_rsa** and the public key is saved in **/home/hduser/.ssh/id\_rsa.pub**. The key fingerprint is **50:6b:f3:fc:0f:32:bf:30:79:c2:41:71:26:cc:7d:e3** and the key's **randomart** image is shown. These keys will be used for SSH public key authentication, which eliminates the need for a password when connecting to a remote machine using SSH.

The key's **randomart** image is:

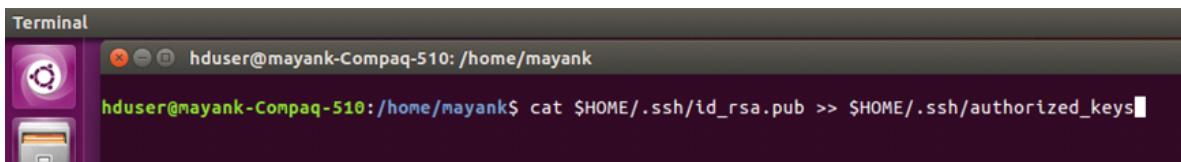
```
+-- [ RSA 2048] ----+  
|          .00.0      |  
|     . .0=. 0       |  
|     . + .  o .    |  
|     o =     E     |  
|     S +           |
```

```

| . +
| 0 +
|   0 o
|   0 ..

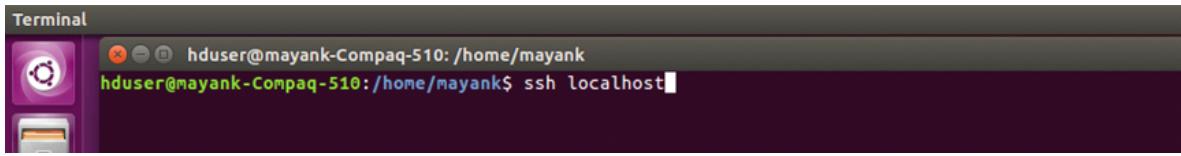
```

As shown in *Figure 5.50*, Unix-like operating systems, the cat command is used to concatenate and display the contents of one or more files. It is a simple yet powerful command-line utility that is commonly used to read and display the contents of text files:



*Figure 5.50: Concatenate and display the contents*

The second command adds the newly created key to the list of authorized keys so that Hadoop can use **ssh** without prompting for a password. In *Figure 5.51*, establishing secure connection at localhost machine is shown. It seems like you have successfully established an SSH connection to your local machine (**localhost**) and logged into the Ubuntu operating system:



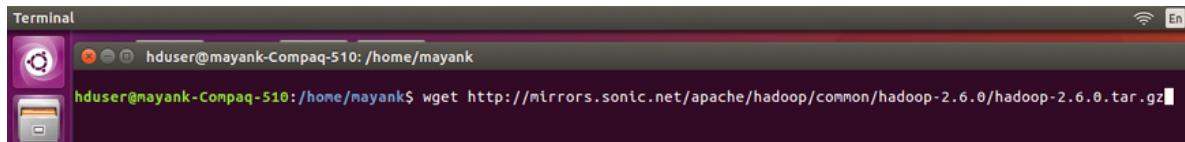
*Figure 5.51: Establish secure connection*

The message about the authenticity of the host and its ECDSA key fingerprint is normal and expected the first time you connect to a new machine over SSH. The warning message asks for confirmation that you trust the host you are connecting to, and you have responded with yes to continue connecting. The last line of the message is simply a welcome message displaying the version and type of operating system you have logged into.

6. Download Hadoop using the following command:

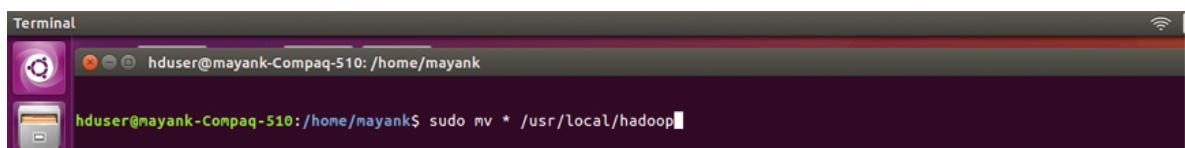
```
tar xvzf hadoop-2.6.0.tar.gz
```

*Figure 5.52* shows all the after-installation prerequisites we need to copy for Hadoop from internet:



*Figure 5.52: Get copy of Hadoop*

Move the Hadoop installation to the **/usr/local/hadoop** directory using the following command as shown in *Figure 5.53*:

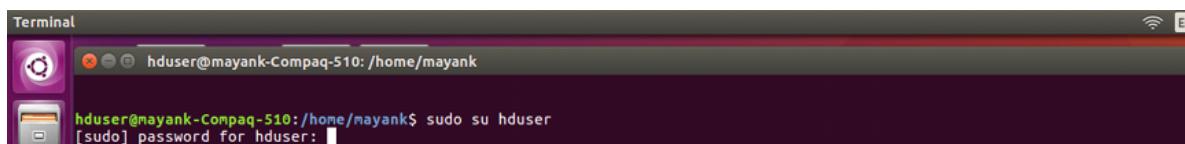


*Figure 5.53: Move the Hadoop installation*

7. Cut file and paste it to **/usr/local/hadoop**.

If **hadoop** folder is not present, then create it. If **hduser** is not in **sudoer list**, then run command in username list (**mayank@mayank-Compaq-510**)

Adding user **hduser** to group **sudo** as shown below in *Figure 5.54*:



*Figure 5.54: Switch user*

```
sudo mv * /usr/local/hadoop
```

```
sudo chown -R hduser:hadoop /usr/local/hadoop
```

8. Setup configuration files

These are some of the important configuration files in a Hadoop installation are as follows:

**~/.bashrc:**

This is a user-specific file that contains various shell settings, including environment variables that are used by Hadoop.

Before editing the `.bashrc` file in per `home` directory, we need to find the path where Java has been installed to set the `JAVA_HOME` environment variable using the following command:

```
$JAVA_HOME
```

The preceding commands export the necessary environment variables for Hadoop to work correctly. The `JAVA_HOME` variable points to the location where Java is installed. The `HADOOP_INSTALL` variable points to the directory where Hadoop is installed.

The `PATH` variable is updated to include the `bin` and `sbin` directories of the Hadoop installation. The `HADOOP_MAPRED_HOME`, `HADOOP_COMMON_HOME`, `HADOOP_HDFS_HOME`, and `YARN_HOME` variables are set to the same location as the `HADOOP_INSTALL` variable.

The `HADOOP_COMMON_LIB_NATIVE_DIR` variable points to the location of the native libraries used by Hadoop. Finally, `HADOOP_OPTS` sets the Java library path to the location of the native libraries.

By sourcing this file, these environment variables will be available to all terminal sessions. The comment `#HADOOP VARIABLES START` suggests that the following lines of code are related to setting up Hadoop variables. In Hadoop installations, you often configure environment variables in a configuration file like `.bashrc` or `.bash_profile` to define essential paths and options required by Hadoop.

Here is an example of how Hadoop variables might be defined in a shell configuration file:

```
```bash
# HADOOP VARIABLES START
export HADOOP_HOME="/path/to/hadoop"
export HADOOP_CONF_DIR="$HADOOP_HOME/etc/hadoop"
export HADOOP_LOG_DIR="/path/to/hadoop/logs"
```

```
export PATH="$HADOOP_HOME/bin:$PATH"  
# HADOOP VARIABLES END  
` ` `
```

Let us go through each variable:

- **HADOOP\_HOME**: This variable specifies the **root** directory where Hadoop is installed.
- **HADOOP\_CONF\_DIR**: This variable points to the directory containing Hadoop configuration files. It is typically located within the Hadoop installation and is used by Hadoop to locate its configuration files.
- **HADOOP\_LOG\_DIR**: This variable specifies the directory where Hadoop log files will be stored. It is a good practice to keep logs separate from the main installation directory.
- **PATH**: This variable includes the Hadoop binary directory in the system's **PATH** environment variable. This ensures that Hadoop commands (hdfs, yarn, mapred) can be executed directly from the command line without specifying the full path.

These environment variables help Hadoop and related tools locate the necessary files and directories and allow you to use Hadoop commands conveniently from any location in the terminal.

After setting these variables in the configuration file, you need to apply the changes by either logging out and logging back in or running the following command to reload the configuration in the current terminal session:

```
` ` ` bash  
source ~/.bashrc  
` ` `
```

Please note that the actual paths and variable names may vary based on your specific Hadoop installation and preferences. Always make sure to adjust the paths according to your Hadoop setup and verify the correctness of the paths and variable names for your particular environment.

*Figure 5.55*, shows setting up the Hadoop variable from start to end:

```

echo '#HADOOP VARIABLES START
export JAVA_HOME=/usr/lib/jvm/java-7-openjdk-amd64
export HADOOP_INSTALL=/usr/local/hadoop
export PATH=$PATH:$HADOOP_INSTALL/bin
export PATH=$PATH:$HADOOP_INSTALL/sbin
export HADOOP_MAPRED_HOME=$HADOOP_INSTALL
export HADOOP_COMMON_HOME=$HADOOP_INSTALL
export HADOOP_HDFS_HOME=$HADOOP_INSTALL
export YARN_HOME=$HADOOP_INSTALL
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_INSTALL/lib/native
export HADOOP_OPTS="-Djava.library.path=$HADOOP_INSTALL/lib"
#HADOOP VARIABLES END' >> ~/.bashrc

```

*Figure 5.55: Setting up hadoop variable*

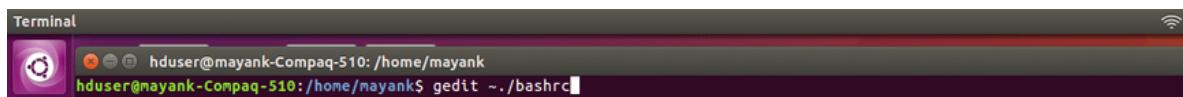
**#HADOOP VARIABLES END**, it is to indicate the end of a section or block where Hadoop-related variables or configuration settings have been defined.

*Figure 5.56* shows the editing file of **.bashrc** file. The correct command to open the **.bashrc** file using the **gedit** text editor is as follows:

```

``` bash
gedit ~/.bashrc
```

```



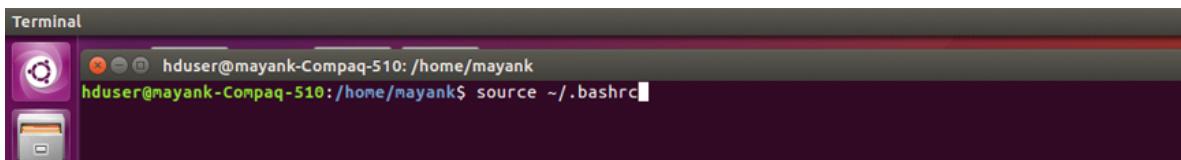
*Figure 5.56: Editing bashrc file*

Here is what the command does:

- **gedit**: This is the command to open the **gedit** text editor, which is the default text editor for the **GNOME** desktop environment on Linux.
- **~/.bashrc**: This specifies the path to the **.bashrc** file in your **home** directory. The tilde (~) represents your **home** directory, so **~/.bashrc** points to the **.bashrc** file in your **home** directory.

Running this command will open the `.bashrc` file in the `gedit` text editor, allowing you to view and edit its contents.

The `source` command as shown in *Figure 5.57* is used to read and execute the contents of a script or configuration file in the current shell session. By running `source ~/.bashrc`, you apply any changes made to the `.bashrc` file immediately in your current terminal session, without the need to log out and log back in.



*Figure 5.57: Execute content of file*

After running the command, the environment variables, aliases, and other configurations defined in the `.bashrc` file will take effect in the current shell. This allows you to use any new settings or variables without starting a new terminal session.

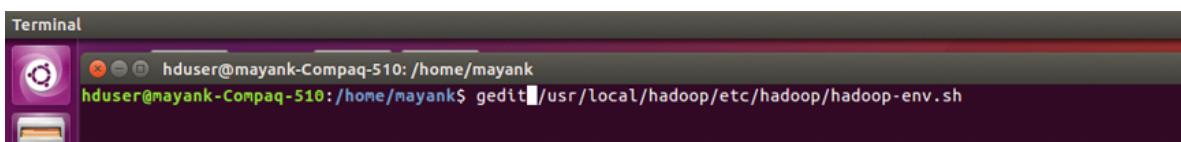
Remember that the `~/.bashrc` file is typically executed automatically when you start a new terminal, so you usually don't need to manually run `source ~/.bashrc` unless you want to apply the changes in your current session without logging out.

Save the path.

```
/usr/local/hadoop/etc/hadoop/hadoop-env.sh
```

This file contains environment variables as shown in *Figure 5.58* that are used by Hadoop, such as the Java `home` directory and Hadoop's `log` directory.

```
export JAVA_HOME= /usr/lib/jvm/java-8-openjdk-amd64
```

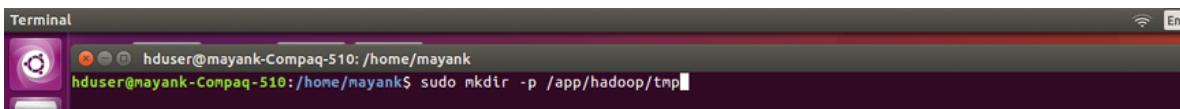


*Figure 5.58: Edit Hadoop enviornment*

### **/usr/local/hadoop/etc/hadoop/core-site.xml:**

This file contains configuration settings for the Hadoop core, such as the default file system and the network address of the **NameNode**.

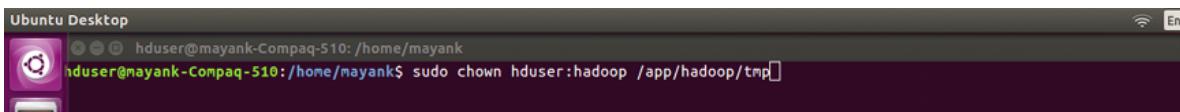
The **/usr/local/hadoop/etc/hadoop/core-site.xml** file contains configuration properties that Hadoop uses when starting up. The file can be used to override the default setting that Hadoop started with. *Figure 5.59* shows the command to create a directory which will be used for storing the intermediate data:



```
Terminal
hduser@mayank-Compaq-510:~/home/mayank
hduser@mayank-Compaq-510:~/home/mayank$ sudo mkdir -p /app/hadoop/tmp
```

*Figure 5.59: Creating directory*

The **chown** command in Linux is used to change the ownership of files, as shown in *Figure 5.60* and directories, and using **sudo** before the command ensures that the operation is performed with administrative privileges:



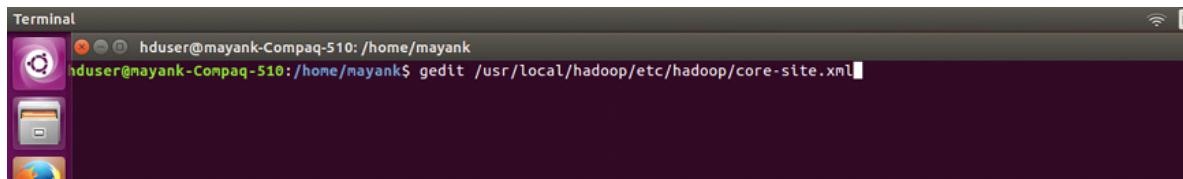
```
Ubuntu Desktop
hduser@mayank-Compaq-510:~/home/mayank
hduser@mayank-Compaq-510:~/home/mayank$ sudo chown hduser:hadoop /app/hadoop/tmp
```

*Figure 5.60: Change ownership*

Here is a breakdown of the command:

- **sudo:** The **sudo** command allows you to execute the following command with superuser (**root**) privileges. It may prompt you to enter your password to verify your identity.
- **chown:** This is the command used to change the ownership of files and directories.
- **hduser :hadoop:** This specifies the new user and group that you want to set as the owner of the directory. In this case, **hduser** is the new user, and Hadoop is the new group.
- **/app/hadoop/temp:** This is the path to the directory whose ownership you want to change.

After running this command, the directory `/app/hadoop/temp` will be owned by the user **hduser** and the group **hadoop**. This can be useful, for example, when configuring Hadoop to ensure that certain directories are accessible and modifiable by the appropriate user and group. *Figure 5.61* shows the editing of **core-site** file as follows:



*Figure 5.61: Editing core-site file*

Open the file and enter the following in between the `<configuration>` `</configuration>` tag:

```
<configuration>
  <property>
    <name>hadoop.tmp.dir</name>
    <value>/app/hadoop/tmp</value>
    <description>A base for other temporary
directories.</description>
  </property>
  <property>
    <name>fs.default.name</name>
    <value>hdfs://localhost:54310</value>
    <description>
```

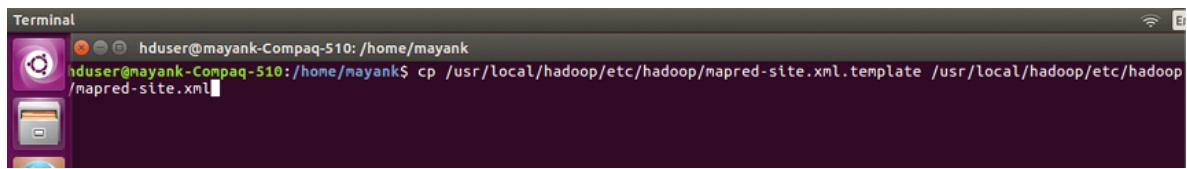
The name of the default file system. A URI whose scheme and authority determine the **FileSystem** implementation. The URI's scheme determines the **config** property (`fs.SCHEME.impl`) naming the **FileSystem** implementation class. The URI's authority is used to determine the **host**, **port**, and so on. for a filesystem:

```
  </description>
```

```
</property>
</configuration>
/usr/local/hadoop/etc/hadoop/mapred-site.xml
```

This file contains configuration settings for the MapReduce framework, such as the number of map and reduce tasks to run in parallel.

By default, the **/usr/local/hadoop/etc/hadoop** folder contains as shown in *Figure 5.62*:

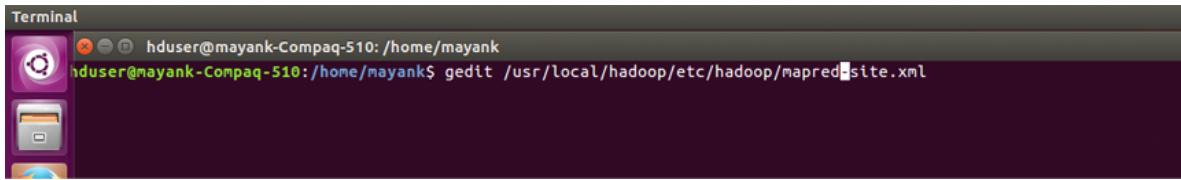


*Figure 5.62: Copy content of file*

/usr/local/hadoop/etc/hadoop/mapred-site.xml.template file which has to be renamed/copied with the name mapred-site.xml. The mapred-site.xml file is used to specify which framework is being used for MapReduce.

We need to enter the following content which shown after executing command *Figure 5.63* in between the **<configuration>** **</configuration>** tag:

```
<configuration>
<property>
    <name>mapred.job.tracker</name>
    <value>localhost:54311</value>
    <description>
        </description>
    </property>
</configuration>
```



**Figure 5.63:** Editing file

The host and port that the MapReduce job tracker run at. If *local*, then jobs are run in process as a single map and reduce task. The *host and port that the MapReduce job tracker run at* refers to the network address (hostname and port number) where the job tracker service is running. This setting specifies how other Hadoop components or clients can connect to and communicate with the job tracker.

`/usr/local/hadoop/etc/hadoop/hdfs-site.xml`

This file contains configuration settings for the HDFS, such as the block size and the replication factor. There is an error in the provided text which is by default in files.

`<value> </value>`

The closing `</value>` tag in the last line is unnecessary and should be removed. Here is the corrected text and the required data path that must be entered in it:

```
<value>
file:/usr/local/hadoop_store/hdfs/datanode
</value>
```

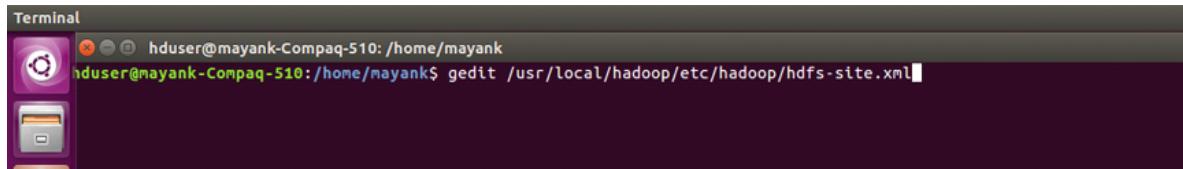
The `/usr/local/hadoop/etc/hadoop/hdfs-site.xml` file needs to be configured for each host in the cluster that is being used. It is used to specify the directories which will be used as the **namenode** and the **datanode** on that host. Before editing this file, we need to create two directories which will contain the **namenode** and the **datanode** for this Hadoop installation. This can be done using the following commands:

```
sudo mkdir -p /usr/local/hadoop_store/hdfs/namenode
sudo mkdir -p /usr/local/hadoop_store/hdfs/datanode
sudo chown -R hduser:hadoop /usr/local/hadoop_store
```

Open the file and enter the following content in between the **<configuration></configuration>** tag:

```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
    <description>
    </description>
  </property>
  <property>
    <name>dfs.namenode.name.dir</name>
    <value>file:/usr/local/hadoop_store/hdfs/namenode</value>
    <description>
      Path on the local filesystem where the NameNode stores the metadata for
      the HDFS file system.
    </description>
  </property>
  <property>
    <name>dfs.datanode.data.dir</name>
    <value>file:/usr/local/hadoop_store/hdfs/datanode</value>
    <description>
      Path on the local filesystem where the DataNode stores the data for the
      HDFS file system (Figure 5.64):
    </description>
  </property>
</configuration>
```

```
</description>  
    </property>  
</configuration>
```



**Figure 5.64:** Edit hdfs-site file

```
<configuration>  
    <property>  
        <name>dfs.replication</name>  
        <value>1</value>  
        <description>  
            (Default block replication. The actual number of replications can be  
            specified when the file is created. The default is used if replication is not  
            specified in create time.)  
        </description>  
    </property>  
    <property>  
        <name>dfs.namenode.name.dir</name>  
  
<value>file:/usr/local/hadoop_store/hdfs/namenode</value>  
    </property>  
    <property>  
        <name>dfs.datanode.data.dir</name>  
  
<value>file:/usr/local/hadoop_store/hdfs/datanode</value>
```

```

/value>

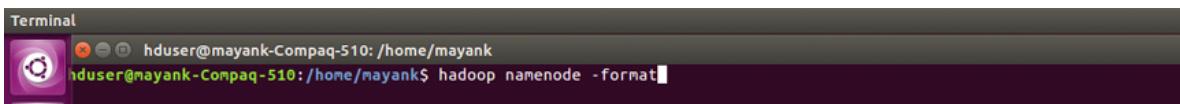
</property>

</configuration>

```

## 9. Format the New Hadoop Filesystem.

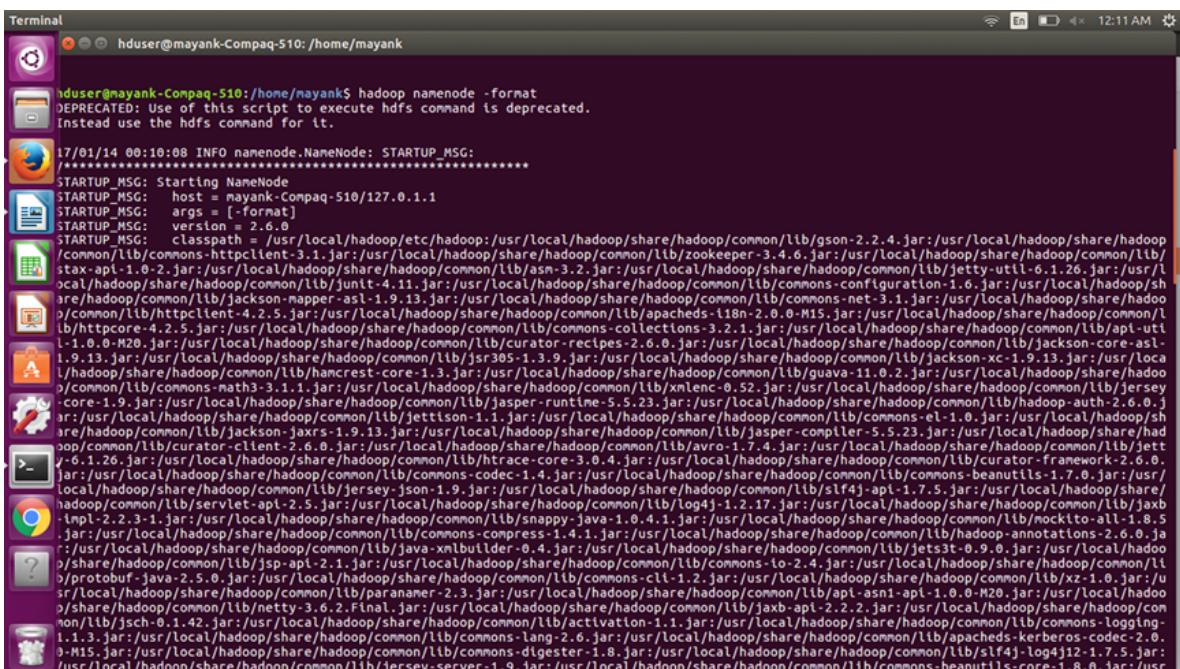
Command shown in *Figure 5.65* initializes the HDFS by formatting the NameNode's metadata directories. It creates the directory structure and files needed for the HDFS to function properly:



*Figure 5.65: Formatting name node*

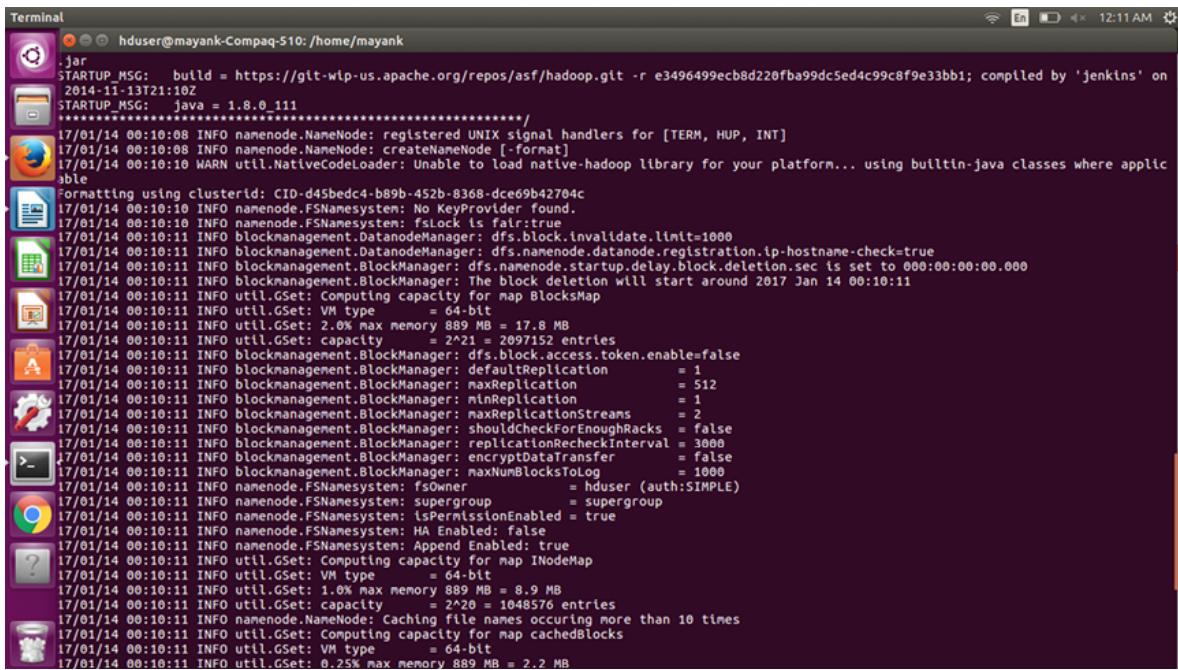
After the formatting is complete, you will see a message indicating that the formatting was successful, along with some additional information. Start the Hadoop daemons, including the **NameNode** and **DataNodes**, to bring the Hadoop cluster back online.

*Figure 5.66* shows the execution file while running the command for **namenode format**:



**Figure 5.66:** Execution of namenode format

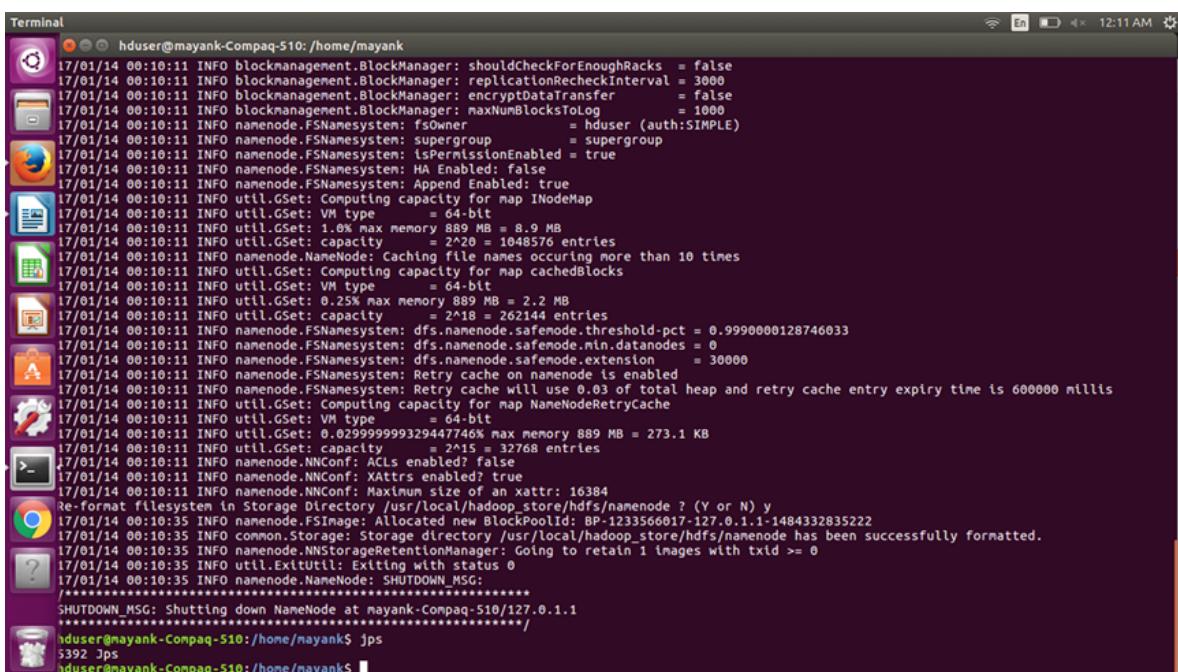
**Figure 5.67** shows the execution of **namenode** format file running:



```
Terminal
hduser@mayank-Compaq-510: /home/mayank
.jar
STARTUP_MSG: build = https://git-wip-us.apache.org/repos/asf/hadoop.git -r e3496499ecb8d220fba99dc5ed4c99c8f9e33bb1; compiled by 'jenkins' on
2014-11-13T21:10Z
STARTUP_MSG: java = 1.8.0_111
*****
17/01/14 00:10:08 INFO namenode.NameNode: registered UNIX signal handlers for [TERM, HUP, INT]
17/01/14 00:10:08 INFO namenode.NameNode: createNameNode [-format]
17/01/14 00:10:10 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Formatting using clusterid: CID-d45bedc4-b89b-452b-8368-dce69b42704c
17/01/14 00:10:10 INFO namenode.FSNamesystem: No KeyProvider found.
17/01/14 00:10:11 INFO blockmanagement.DatanodeManager: dfs.block.Invalidate.limit=1000
17/01/14 00:10:11 INFO blockmanagement.DatanodeManager: dfs.namenode.datanode.registration.ip-hostname-check=true
17/01/14 00:10:11 INFO blockmanagement.BlockManager: dfs.namenode.startup.delay.block.deletion.sec is set to 000:00:00:00.000
17/01/14 00:10:11 INFO blockmanagement.BlockManager: The block deletion will start around 2017 Jan 14 00:10:11
17/01/14 00:10:11 INFO util.GSet: Computing capacity for map BlocksMap
17/01/14 00:10:11 INFO util.GSet: VM type      = 64-bit
17/01/14 00:10:11 INFO util.GSet: 2.0% max memory 889 MB = 17.8 MB
17/01/14 00:10:11 INFO util.GSet: capacity      = 2^21 = 2097152 entries
17/01/14 00:10:11 INFO blockmanagement.BlockManager: dfs.block.access.token.enable=false
17/01/14 00:10:11 INFO blockmanagement.BlockManager: defaultReplication          = 1
17/01/14 00:10:11 INFO blockmanagement.BlockManager: maxReplication             = 512
17/01/14 00:10:11 INFO blockmanagement.BlockManager: minReplication            = 1
17/01/14 00:10:11 INFO blockmanagement.BlockManager: maxReplicationStreams     = 2
17/01/14 00:10:11 INFO blockmanagement.BlockManager: shouldCheckForEnoughRacks = false
17/01/14 00:10:11 INFO blockmanagement.BlockManager: replicationRecheckInterval = 3000
17/01/14 00:10:11 INFO blockmanagement.BlockManager: encryptDataTransfer        = false
17/01/14 00:10:11 INFO blockmanagement.BlockManager: maxNumBlocksToLog         = 1000
17/01/14 00:10:11 INFO namenode.FSNamesystem: fsOwner                  = hduser (auth:SIMPLE)
17/01/14 00:10:11 INFO namenode.FSNamesystem: supergroup               = supergroup
17/01/14 00:10:11 INFO namenode.FSNamesystem: isPermissionEnabled        = true
17/01/14 00:10:11 INFO namenode.FSNamesystem: HA Enabled                = false
17/01/14 00:10:11 INFO namenode.FSNamesystem: Append Enabled           = true
17/01/14 00:10:11 INFO util.GSet: Computing capacity for map INodeMap
17/01/14 00:10:11 INFO util.GSet: VM type      = 64-bit
17/01/14 00:10:11 INFO util.GSet: 1.0% max memory 889 MB = 8.9 MB
17/01/14 00:10:11 INFO util.GSet: capacity      = 2^20 = 1048576 entries
17/01/14 00:10:11 INFO namenode.NameNode: Caching file names occurring more than 10 times
17/01/14 00:10:11 INFO util.GSet: Computing capacity for map cachedBlocks
17/01/14 00:10:11 INFO util.GSet: VM type      = 64-bit
17/01/14 00:10:11 INFO util.GSet: 0.25% max memory 889 MB = 2.2 MB
17/01/14 00:10:11 INFO util.GSet: capacity      = 2^18 = 262144 entries
17/01/14 00:10:11 INFO namenode.FSNamesystem: dfs.namenode.safemode.threshold-pct = 0.9990000128746033
17/01/14 00:10:11 INFO namenode.FSNamesystem: dfs.namenode.safemode.mln.datanodes = 0
17/01/14 00:10:11 INFO namenode.FSNamesystem: dfs.namenode.safemode.extension   = 30000
17/01/14 00:10:11 INFO namenode.FSNamesystem: Retry cache on namenode is enabled
17/01/14 00:10:11 INFO namenode.FSNamesystem: Retry cache will use 0.03 of total heap and retry cache entry expiry time is 600000 millis
17/01/14 00:10:11 INFO util.GSet: Computing capacity for map NameNodeRetryCache
17/01/14 00:10:11 INFO util.GSet: VM type      = 64-bit
17/01/14 00:10:11 INFO util.GSet: 0.029999999329447746% max memory 889 MB = 273.1 KB
17/01/14 00:10:11 INFO util.GSet: capacity      = 2^15 = 32768 entries
17/01/14 00:10:11 INFO namenode.NNConf: ACLs enabled? false
17/01/14 00:10:11 INFO namenode.NNConf: XAttrs enabled? true
17/01/14 00:10:11 INFO namenode.NNConf: Maximum size of an xattr: 16384
Re-format filesystem in Storage Directory /usr/local/hadoop_store/hdfs/namenode ? (Y or N) y
17/01/14 00:10:35 INFO namenode.FSImage: Allocated new BlockPoolId: BP-1233566017-127.0.1.1-148433283522
17/01/14 00:10:35 INFO common.Storage: Storage directory /usr/local/hadoop_store/hdfs/namenode has been successfully formatted.
17/01/14 00:10:35 INFO namenode.NNStorageRetentionManager: Going to retain 1 images with txId >= 0
17/01/14 00:10:35 INFO util.ExitUtil: Exiting with status 0
17/01/14 00:10:35 INFO namenode.NameNode: SHUTDOWN_MSG:
*****SHUTDOWN MSG: Shutting down NameNode at mayank-Compaq-510/127.0.1.1
*****
hduser@mayank-Compaq-510: /home/mayank$ jps
5392 Jps
hduser@mayank-Compaq-510: /home/mayank$
```

**Figure 5.67:** Continuation of namenode format-1

Refer to **Figure 5.68**, to see the execution of **namenode** format file:



```
Terminal
hduser@mayank-Compaq-510: /home/mayank
17/01/14 00:10:11 INFO blockmanagement.BlockManager: shouldCheckForEnoughRacks = false
17/01/14 00:10:11 INFO blockmanagement.BlockManager: replicationRecheckInterval = 3000
17/01/14 00:10:11 INFO blockmanagement.BlockManager: encryptDataTransfer        = false
17/01/14 00:10:11 INFO blockmanagement.BlockManager: maxNumBlocksToLog         = 1000
17/01/14 00:10:11 INFO namenode.FSNamesystem: fsOwner                  = hduser (auth:SIMPLE)
17/01/14 00:10:11 INFO namenode.FSNamesystem: supergroup               = supergroup
17/01/14 00:10:11 INFO namenode.FSNamesystem: isPermissionEnabled        = true
17/01/14 00:10:11 INFO namenode.FSNamesystem: HA Enabled                = false
17/01/14 00:10:11 INFO namenode.FSNamesystem: Append Enabled           = true
17/01/14 00:10:11 INFO util.GSet: Computing capacity for map INodeMap
17/01/14 00:10:11 INFO util.GSet: VM type      = 64-bit
17/01/14 00:10:11 INFO util.GSet: 1.0% max memory 889 MB = 8.9 MB
17/01/14 00:10:11 INFO util.GSet: capacity      = 2^20 = 1048576 entries
17/01/14 00:10:11 INFO namenode.NameNode: Caching file names occurring more than 10 times
17/01/14 00:10:11 INFO util.GSet: Computing capacity for map cachedBlocks
17/01/14 00:10:11 INFO util.GSet: VM type      = 64-bit
17/01/14 00:10:11 INFO util.GSet: 0.25% max memory 889 MB = 2.2 MB
17/01/14 00:10:11 INFO util.GSet: capacity      = 2^18 = 262144 entries
17/01/14 00:10:11 INFO namenode.FSNamesystem: dfs.namenode.safemode.threshold-pct = 0.9990000128746033
17/01/14 00:10:11 INFO namenode.FSNamesystem: dfs.namenode.safemode.mln.datanodes = 0
17/01/14 00:10:11 INFO namenode.FSNamesystem: dfs.namenode.safemode.extension   = 30000
17/01/14 00:10:11 INFO namenode.FSNamesystem: Retry cache on namenode is enabled
17/01/14 00:10:11 INFO namenode.FSNamesystem: Retry cache will use 0.03 of total heap and retry cache entry expiry time is 600000 millis
17/01/14 00:10:11 INFO util.GSet: Computing capacity for map NameNodeRetryCache
17/01/14 00:10:11 INFO util.GSet: VM type      = 64-bit
17/01/14 00:10:11 INFO util.GSet: 0.029999999329447746% max memory 889 MB = 273.1 KB
17/01/14 00:10:11 INFO util.GSet: capacity      = 2^15 = 32768 entries
17/01/14 00:10:11 INFO namenode.NNConf: ACLs enabled? false
17/01/14 00:10:11 INFO namenode.NNConf: XAttrs enabled? true
17/01/14 00:10:11 INFO namenode.NNConf: Maximum size of an xattr: 16384
Re-format filesystem in Storage Directory /usr/local/hadoop_store/hdfs/namenode ? (Y or N) y
17/01/14 00:10:35 INFO namenode.FSImage: Allocated new BlockPoolId: BP-1233566017-127.0.1.1-148433283522
17/01/14 00:10:35 INFO common.Storage: Storage directory /usr/local/hadoop_store/hdfs/namenode has been successfully formatted.
17/01/14 00:10:35 INFO namenode.NNStorageRetentionManager: Going to retain 1 images with txId >= 0
17/01/14 00:10:35 INFO util.ExitUtil: Exiting with status 0
17/01/14 00:10:35 INFO namenode.NameNode: SHUTDOWN_MSG:
*****SHUTDOWN MSG: Shutting down NameNode at mayank-Compaq-510/127.0.1.1
*****
hduser@mayank-Compaq-510: /home/mayank$ jps
5392 Jps
hduser@mayank-Compaq-510: /home/mayank$
```

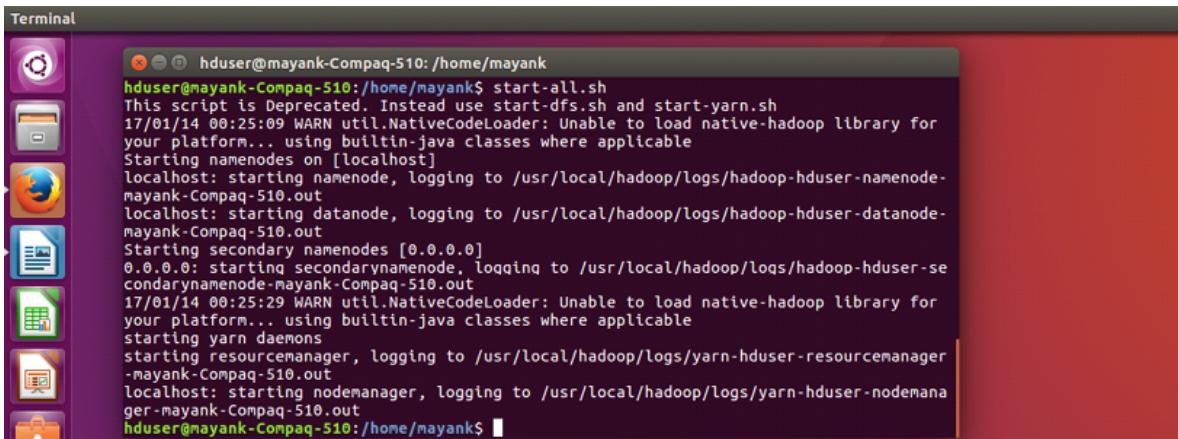
**Figure 5.68:** Continuation of namenode format-2

Note that **hadoop namenode-format** command should be executed once before we start using Hadoop. If this command is executed again after Hadoop has been used, it will destroy all the data on the Hadoop file system.

## 10. Starting Hadoop.

Now it is time to start the newly installed single node cluster.

We can use **start-all.sh** or (**start-dfs.sh** and **start-yarn.sh**). The command shown in [Figure 5.69](#), **start-all.sh** is used to start all Hadoop daemons on a Hadoop cluster. It is a convenient script that starts the essential daemons, such as the **NameNode**, **DataNodes**, **ResourceManager**, **NodeManagers**, and other services, required to run a Hadoop cluster.



The screenshot shows a terminal window titled "Terminal". The command entered is "hduser@mayank-Compaq-510:~/home/mayank\$ start-all.sh". The output indicates that the script is deprecated and suggests using "start-dfs.sh" and "start-yarn.sh" instead. It then lists the startup of various daemons: NameNode, DataNode, and ResourceManager. The output ends with "hduser@mayank-Compaq-510:~/home/mayank\$".

**Figure 5.69:** Restarting of all daemons

Here is how to use the start-all.sh command:

1. Open a terminal or command prompt on the machine where your Hadoop cluster is installed.
2. Navigate to the Hadoop installation directory. This is the directory where you extracted or installed Hadoop.
3. Use the following command to start all Hadoop daemons:

`` `

`./start-all.sh`

The `./` at the beginning of the command is used to indicate that the script should be executed from the current directory.

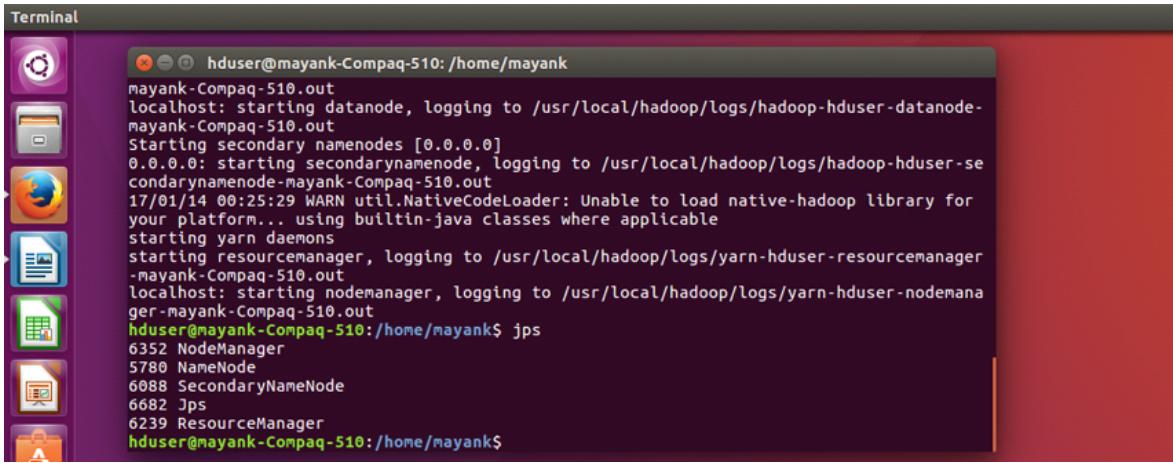
4. The script will start all the necessary daemons, and you will see output indicating the status of each daemon as they start.

Please note that the `start-all.sh` script assumes that your Hadoop cluster is set up correctly, and the necessary configuration files (`core-site.xml`, `hdfs-site.xml`, `yarn-site.xml`) are properly configured.

Also, keep in mind that running this script requires administrative privileges, so you might need to use the `sudo` command if you do not have the necessary permissions to start the Hadoop daemons.

Starting all Hadoop daemons using `start-all.sh` is a convenient way to quickly start a single-node or small Hadoop cluster. It starts all the daemons listed in the Hadoop configuration file (`hadoop-env.sh`) **NameNode**, **DataNode**, **SecondaryNameNode**, **ResourceManager**, **NodeManager**, and **JobHistoryServer**. For more extensive Hadoop clusters, you might have a more complex setup with separate scripts or tools to manage the daemons individually or in specific groups. Additionally, on production clusters, you may prefer to start and manage the daemons using tools like Apache Ambari or Cloudera Manager, which provide more advanced management and monitoring capabilities.

*Figure 5.70* shows the current daemons status on Hadoop. This output shows that the machine is running fine with all its components:

A screenshot of an Ubuntu desktop environment. On the left, there's a dock with icons for Dash, Home, Applications, and others. A terminal window is open in the center, titled 'Terminal'. The terminal shows the command 'jps' being run and its output. The output includes logs for starting DataNodes, Secondary NameNodes, and ResourceManagers, followed by the process IDs for NodeManager (6352), NameNode (5780), SecondaryNameNode (6088), Jps (6682), and ResourceManager (6239).

```
hduser@mayank-Compaq-510: /home/mayank
mayank-Compaq-510.out
localhost: starting datanode, logging to /usr/local/hadoop/logs/hadoop-hduser-datanode-
mayank-Compaq-510.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /usr/local/hadoop/logs/hadoop-hduser-se-
condarynamenode-mayank-Compaq-510.out
17/01/14 00:25:29 WARN util.NativeCodeLoader: Unable to load native-hadoop library for
your platform... using builtin-java classes where applicable
starting yarn daemons
starting resourcemanager, logging to /usr/local/hadoop/logs/yarn-hduser-resourcemanager-
mayank-Compaq-510.out
localhost: starting nodemanager, logging to /usr/local/hadoop/logs/yarn-hduser-nodeman-
ager-mayank-Compaq-510.out
hduser@mayank-Compaq-510: /home/mayank$ jps
6352 NodeManager
5780 NameNode
6088 SecondaryNameNode
6682 Jps
6239 ResourceManager
hduser@mayank-Compaq-510: /home/mayank$
```

**Figure 5.70:** jps

## Fully distributed mode

In contrast to Local Mode, in hadoop, fully distributed mode refers to a deployment configuration where Hadoop runs on a cluster of multiple machines, each serving a specific role in the distributed computing environment. Fully distributed mode is the ideal setup for running Hadoop in production, as it allows for horizontal scalability and efficient distributed processing of large-scale data.

In fully distributed mode, the Hadoop cluster consists of the following key components:

- **NameNode:** One machine in the cluster acts as the primary **NameNode**, responsible for storing the metadata of the HDFS. It keeps track of the locations of data blocks and manages file system namespace operations.
- **Secondary NameNode (Optional):** Although called the **Secondary NameNode**, it does not act as a backup for the primary **NameNode**. Instead, it periodically performs a checkpoint of the metadata to assist in the recovery process and reduce the time taken during **NameNode** restarts.
- **DataNodes:** Multiple machines in the cluster serve as **DataNodes**. They store the actual data blocks of files in the HDFS. **DataNodes** are responsible for serving read and write requests from clients and replication of data blocks for fault tolerance.
- **ResourceManager:** One machine serves as the **ResourceManager**, responsible for managing resources (CPU, memory) in the cluster. It

receives resource requests from clients (for example, MapReduce jobs) and schedules containers (task execution units) on available **NodeManagers**.

- **NodeManagers:** Multiple machines in the cluster act as **NodeManagers**. They manage resources on individual nodes, monitor container status, and execute tasks assigned by the **ResourceManager**.
- **JobHistoryServer (Optional):** It stores job-related information and allows users to view the history of completed MapReduce jobs.

In fully distributed mode, data is spread across **DataNodes** for fault tolerance and parallel processing. HDFS ensures that data blocks are replicated across multiple **DataNodes** to ensure data availability even in the event of node failures.

The processing of data is distributed across the cluster using the MapReduce programming model (or newer processing frameworks like Apache Spark) to efficiently process large datasets in parallel.

Fully distributed mode requires careful configuration of Hadoop's core components (**core-site.xml**, **hdfs-site.xml**, **yarn-site.xml**) and coordination between the different machines in the cluster. It provides the advantages of scalability, fault tolerance, and efficient data processing, making it suitable for handling big data workloads in production environments.

## Installation and configuration of multi-node cluster

Following are the steps for installation and configuration of multi-node cluster:

1. We will name **Masternode** as **HadoopMaster** and the 3 different slave nodes as **HadoopSlave2**, **HadoopSlave4**, **HadoopSlave5** respectively in **/etc/** directory in file name hosts.

After deciding a hostname of all nodes, assign their names by updating hostnames. In master node provide IP address of the **masternode** and all slave nodes. Also make an entry of IP address of the master node and data node in **/etc/hosts** file at each **datanode**.

Add following hostname and their IP in hosts:

172.29.0.193	hadoopmaster
172.29.0.179	hadoopslave2
172.29.0.159	hadoopslave5
172.29.0.171	hadoopslave4

2. We will provide IP address of master node in hostname file of master node and data node IP address in hostname file of the respective **datanodes**. It means the hostname file will contain the IP address of the machine itself. The command for the same is **sudo gedit /etc/hostname**.
3. The following command needs to be executed to create a group called Hadoop and a user called **hduser** in all machines:

```
sudo addgroup hadoop
```

```
sudo adduser --ingroup hadoop hduser
```

If you want to give **sudo** privileges to the user **hduser**, you can use the following command:

```
sudo adduser hduser
```

4. By adding the user **hduser** to the **sudo** group, you are granting that user the ability to run commands with administrative privileges. This is often done to allow certain users to perform tasks that require elevated permissions, while still maintaining a level of security by not giving them full root access all the time. Here, the new user is added in admin mode.
  5. This command installs the **rsync** package on the Ubuntu or Debian-based system, which is used for efficient file synchronization and transfer between systems. The **rsync** package can be used to share the Hadoop source code with other machines in the cluster.
- ```
sudo apt-get install rsync
```
6. To reflect the above changes, we need to reboot all the machines using **sudo reboot**.
  7. Disabling IPv6- For getting your IPv6 disabled in your Linux machine, you need to update **/etc/sysctl.conf** by adding the following line of codes at the end of the file.

Display IPv6:

```
Net.ipv6.conf.all.disable_ipv6=1  
Net.ipv6.conf.default.disable_ipv6=1  
Net.ipv6.conf.lo.disable_ipv6=1
```

8. Firstly we need to configure the `~/.bashrc` file on every master node and slave node and provide Java path and `hadoop` path to the system.

```
vi ~/.bashrc
```

```
#HADOOP VARIABLES START
```

```
export JAVA_HOME=/usr/lib/jvm/java-7-openjdk-amd64  
export HADOOP_INSTALL=/usr/local/hadoop  
export PATH=$PATH:$HADOOP_INSTALL/bin  
export PATH=$PATH:$HADOOP_INSTALL/sbin  
export HADOOP_MAPRED_HOME=$HADOOP_INSTALL  
export HADOOP_COMMON_HOME=$HADOOP_INSTALL  
export HADOOP_HDFS_HOME=$HADOOP_INSTALL  
export YARN_HOME=$HADOOP_INSTALL  
export  
HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_INSTALL/lib/n  
ative  
export HADOOP_OPTS="-  
Djava.library.path=$HADOOP_INSTALL/lib"  
  
#HADOOP VARIABLES END
```

The `JAVA_HOME` environment variable is set to the path where Java is installed. The `HADOOP_INSTALL` variable is set to the path where Hadoop is installed. The `PATH` environment variable is updated to include the Hadoop `bin` and `sbin` directories. The remaining environment variables are set to

point to the various components of Hadoop, such as the MapReduce and HDFS directories. Finally, **HADOOP\_OPTS** is set to specify the location of the Hadoop native libraries.

```
hduser@ram:~$ source ~/.bashrc
```

## 9. Hadoop configuration steps:

This statement explains that some changes need to be made to the Hadoop configuration files before setting up a Master-Slave architecture. These changes will be common for both Master and Slave nodes and will be applied to a single node Hadoop setup before distributing the Hadoop files to other machines/nodes.

Changes in master and slave node clusters are:

### a. Update **core-site.xml**.

To update the **core-site.xml** file, change the hostname from localhost to **HadoopMaster** by either pasting or updating these lines into the **<configuration>**:

```
<property>
  <name>fs.default.name</name>
  <value>hdfs://HadoopMaster:9000</value>
</property>
```

### b. Update **hdfs-site.xml**.

To update the **hdfs-site.xml** file, update the replication factor from 1 to 3 by pasting or updating these lines into the **<configuration>**.

```
<property>
  <name>dfs.replication</name>
  <value>3</value>
</property>
```

### c. Update **yarn-site.xml**

To update the **yarn-site.xml** file, paste or update these lines into the **<configuration>**.

```
<property>
    <name>yarn.resourcemanager.resource-
tracker.address</name>
    <value>HadoopMaster:8025</value>
</property>
<property>

<name>yarn.resourcemanager.scheduler.address</na
me>
    <value>HadoopMaster:8035</value>
</property>
<property>
    <name>yarn.resourcemanager.address</name>
    <value>HadoopMaster:8050</value>
</property>
```

d. Update **mapred-site.xml**.

To update the **mapred-site.xml** file, update and add the following properties by pasting or updating these lines into the **<configuration>**

```
<property>
    <name>mapreduce.job.tracker</name>
    <value>HadoopMaster:5431</value>
</property>
<property>
    <name>mapred.framework.name</name>
    <value>yarn</value>
```

```
</property>
```

10. Now update the **Masters** file on `/usr/local/hadoop/etc/hadoop` on the data node in Hadoop cluster.

Add IP address of Hadoop master: **172.29.0.193**

11. Now update **Masters** and **Slaves** file on Hadoop master node.

#### **Update masters:**

Update the file of master nodes of Hadoop cluster.

Add IP address of Hadoop master: **172.29.0.193**

#### **Update slaves:**

Update the file of slave nodes of Hadoop cluster.

Add IP address of Hadoop master and Hadoop slaves:

**172.29.0.193**

**172.29.0.171**

**172.29.0.179**

**172.29.0.159**

12. The following commands use **rsync** to distribute the configured Hadoop source to the rest of the nodes in the network.

In **HadoopSlave2** machine:

```
sudo rsync -avxP /usr/local/hadoop/  
hduser@HadoopSlave2:/usr/local/hadoop/
```

In **HadoopSlave4** machine:

```
sudo rsync -avxP /usr/local/hadoop/  
hduser@HadoopSlave4:/usr/local/hadoop/
```

In **HadoopSlave5** machine:

```
sudo rsync -avxP /usr/local/hadoop/  
hduser@HadoopSlave5:/usr/local/hadoop/
```

These commands will copy the files located within the `/usr/local/hadoop/` directory to the slave nodes specified (`HadoopSlave2`, `HadoopSlave4`, and `HadoopSlave5`) in the `/usr/local/hadoop/` directory of each respective node. The `-a` flag preserves the permissions and timestamps of the files being transferred, while `-v` enables verbose mode to show the progress of the transfer, `-x` ensures that the transfer doesn't cross file system boundaries, and `-P` displays progress information during the transfer.

13. These steps are intended to configure the Hadoop **NameNode** on the master node:

These are some configurations to be applied over Hadoop **MasterNodes** (since we have only one master node, it will be applied to only one master node.)

Remove the existing Hadoop **data** directory:

```
sudo rm -rf /usr/local/hadoop_tmp/
```

Create the **data** directory and **NameNode** directory:

```
sudo mkdir -p /usr/local/hadoop_tmp/hdfs/namenode
```

Change the owner of the **data** directory to the **hduser** user:

```
sudo chown hduser:hadoop -R /usr/local/hadoop_tmp/
```

Format the **NameNode**:

```
hdfs namenode -format
```

Start the Hadoop daemons:

```
start-dfs.sh
```

```
start-yarn.sh
```

14. Applying Slave node specific Hadoop configuration: (only for slave node).

Since we have three slave nodes, we will be applying the following changes over `HadoopSlave4`, `HadoopSlave2`, and `HadoopSlave5` nodes.

15. Remove the existing **Hadoop\_data** folder (which was created while single node Hadoop setup):

```
sudo rm -rf /usr/local/hadoop_tmp/hdfs/
```

16. Creates same (**/usr/local/hadoop\_tmp/**) directory/folder inside this folder again create **DataNode** (**/usr/local/hadoop\_tmp/hdfs/datanode**) directory/folder.

```
sudo mkdir -p /usr/local/hadoop_tmp/
```

```
sudo mkdir -p /usr/local/hadoop_tmp/hdfs/datanode
```

17. Make **hduser** as owner of that directory

```
sudo chown hduser:hadoop -R /usr/local/hadoop_tmp/
```

18. The **ssh-copy-id** command is used to copy the public SSH key of the user from the Master node to the **authorized\_keys** file on the Slave nodes. This allows the user to SSH into the Slave nodes without having to enter a password each time.

The command **ssh-copy-id -i \$HOME/.ssh/id\_rsa.pub** **hduser@HadoopSlave2** will copy the public SSH key to the **authorized\_keys** file of **hduser** on **HadoopSlave2**. Similarly, the command **ssh-copy-id -i \$HOME/.ssh/id\_rsa.pub** **hduser@HadoopSlave4** and **ssh-copy-id -i \$HOME/.ssh/id\_rsa.pub** **hduser@HadoopSlave5** will copy the public SSH key to the **authorized\_keys** file of **hduser** on **HadoopSlave4** and **HadoopSlave5** respectively.

19. Corrected command. Run this command from **Masternode**:

```
hduser@hadoopmaster:/usr/local/hadoop$ hdfs namenode -format
```

20. It is recommended to start HDFS daemons in a specific order: **Namenode** should be started first, followed by the **Datanodes**.

Here are the steps to start the HDFS daemons on the **MasterNode**:

- a. Start the **Namenode** daemon:

```
hduser@hadoopmaster:/usr/local/hadoop$ hdfs --daemon start namenode
```

- b. Start the **Datanode** daemon(s) on all **SlaveNodes**:

```
hduser@hadoopmaster:/usr/local/hadoop$ hdfs --daemon start datanode
```

c. Verify that the daemons are running using the following command:

```
hduser@hadoopmaster:/usr/local/hadoop$ jps
```

d. You should see output similar to the following:

```
3874 Jps
```

```
3698 DataNode
```

```
3542 NameNode
```

Optionally, you can start the Secondary NameNode daemon:

```
hduser@hadoopmaster:/usr/local/hadoop$ hdfs --daemon start secondarynamenode
```

**Note:** The secondary NameNode daemon should be started after the Namenode daemon is started.

To start all HDFS daemons in one go, you can use the following command:

```
hduser@hadoopmaster:/usr/local/hadoop$ start-dfs.sh
```

This will start the following daemons:

NameNode, DataNode, SecondaryNameNode, Namenode  
SecondaryIndexer, JournalNode

These daemons will be started in the background. You can check their status using the following command:

```
$ hadoop dfsadmin -status
```

21. Starting from Hadoop 2.x, the MapReduce daemons are part of YARN, so starting YARN is sufficient to start both HDFS and YARN.

To start MapReduce (YARN) daemons, run the following command on the **MasterNode**:

```
hduser@HadoopMaster:/usr/local/hadoop$ start-yarn.sh
```

22. The **jps** command lists all the Java processes running on the current node. To verify Hadoop daemons specifically, you can run **jps** command on the Hadoop Master node as follows:
23. This will list all the Java processes running on the Hadoop Master node, including the Hadoop daemons such as **NameNode**, **SecondaryNameNode**, **DataNode**, **ResourceManager**, and **NodeManager**:

**Name Node**

**Secondary Name Node**

**jps**

**Resource Manager**

Verify Hadoop daemons on all slave nodes:

**hduser@HadoopSlave2: jps**

**hduser@HadoopSlave4: jps**

**hduser@HadoopSlave5: jps**

**jps**

**Node Manager**

**Data Node**

24. Let us consider the overview of an example where the file **data\_odd\_even\_prime.txt** is copied on a Hadoop cluster and data is generated.

*Figure 5.71* shows the status of all working nodes in the multimode cluster:

Datanode Information

In operation

Node	Http Address	Last contact	Capacity	Blocks	Block pool used	Version
✓ hadoopmaster:50010 (172.29.0.193:50010)	hadoopmaster:50075	0s	303.49 GB	9	1.04 GB (0.34%)	2.8.0
✓ hadoopslave2:50010 (172.29.0.179:50010)	hadoopslave2:50075	0s	98.16 GB	3	291.23 MB (0.29%)	2.7.3
✓ hadoopslave4:50010 (172.29.0.171:50010)	hadoopslave4:50075	1s	100.8 GB	8	1.01 GB (1%)	2.8.0
✓ hadoopslave5:50010 (172.29.0.159:50010)	hadoopslave5:50075	1s	41.34 GB	7	807.25 MB (1.91%)	2.8.0

**Figure 5.71:** Information of all nodes

If we want to check information regarding our data block size and replicas created on the data node, it can be shown in [Figure 5.72](#), it shows that block size and replicas are 128 MB in size, by which all the data can be stored. It is variable in size and multiple of 64 MB:

Browsing HDFS - Mozilla Firefox

Browsing HDFS

Browser Directory

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	hduser	supergroup	1.03 GB	Jun 21 16:41	3	128 MB	data_odd_even_prime.txt

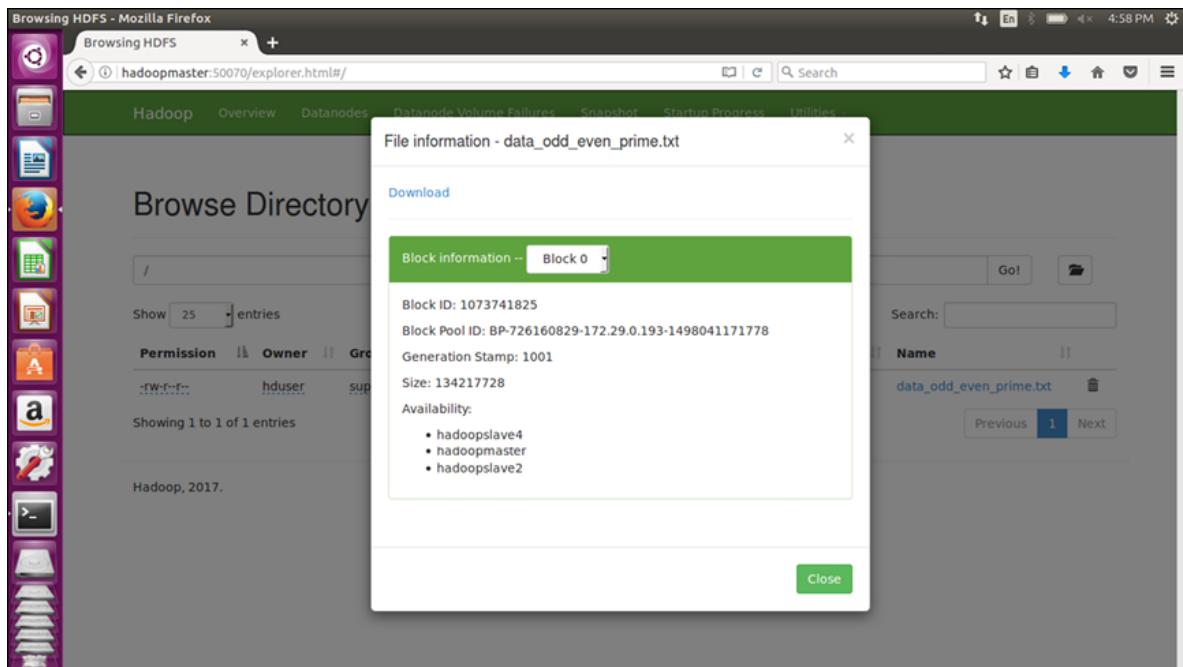
Showing 1 to 1 of 1 entries

Hadoop, 2017.

**Figure 5.72:** Block size

25. Let us now see how many data nodes are involved in generating data and creating a replica of the file.

*Figure 5.73* shows the block information:



*Figure 5.73: Block information*

The installation of a multi-node cluster offers a multitude of advantages in the realm of distributed computing and data processing. By distributing computing tasks across multiple nodes, it leverages parallel processing to significantly enhance performance and handle larger workloads. This results in improved processing speed, scalability, and fault tolerance, making it ideal for Big Data applications. Multi-node clusters also provide redundancy, ensuring data reliability and high availability, as well as allowing for effective resource sharing and load balancing. Furthermore, they offer the ability to accommodate diverse workloads and efficiently harness the collective computational power of multiple machines, thereby optimizing the utilization of resources and delivering robust, responsive, and resilient computing environments.

## Conclusion

Setting up a Hadoop cluster is crucial for big data processing and analysis. Throughout this chapter, we explored the fundamental aspects of configuring a

Hadoop cluster, from deployment modes to core configuration files and security considerations. We emphasized the significance of hardware, network, data replication, and fault tolerance. Testing and troubleshooting were also crucial steps in the setup process. As readers take their first strides in the Hadoop world, continuous learning and exploration of advanced topics are encouraged. With the completion of the Hadoop setup, readers are empowered to process vast volumes of data and drive innovation through data exploration and analysis.

Next, we will delve into processing with Hadoop's MapReduce functions.

## **Join our book's Discord space**

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



# CHAPTER 6

## MapReduce Applications

### Introduction

MapReduce structure is useful to process large amounts of data without manual intervention. MapReduce programs can be written in various languages. The book uses Java as the chosen programming language for data processing. MapReduce is based on two fundamental portions, that is, the Map part and the Reduce part. The flow of a sample program has been described before. Now, here we will be understanding MapReduce from scratch.

### Structure

This chapter includes the following points:

- Understanding of MapReduce
- Traditional way
- Word count program
- Map side data flow
- Reduce side data flow
- MR unit flow
- Debugging of MapReduce program
- Data flow of the program
- Anatomy of MapReduce structure
- MapReduce 2 Yarn

## Objectives

This chapter discusses the fundamental components and workflow of MapReduce applications, which are essential for processing and analyzing data in Hadoop. This chapter provides a comprehensive understanding of how MapReduce applications are designed and executed, enabling readers to develop efficient and scalable data processing solutions. MapReduce applications are commonly used for tasks such as data transformation, aggregation, and analysis, and they are particularly valuable in scenarios where data is too large to be processed on a single machine. The objective is to harness the power of distributed computing resources to handle vast amounts of data efficiently, reducing the processing time and resource requirements. This makes MapReduce a fundamental tool in the field of big data processing, enabling businesses and organizations to extract valuable insights and knowledge from their massive datasets. The ultimate goal is to extract valuable insights, patterns, or summaries from vast datasets, improving decision-making and facilitating data-driven insights.

## Understanding MapReduce

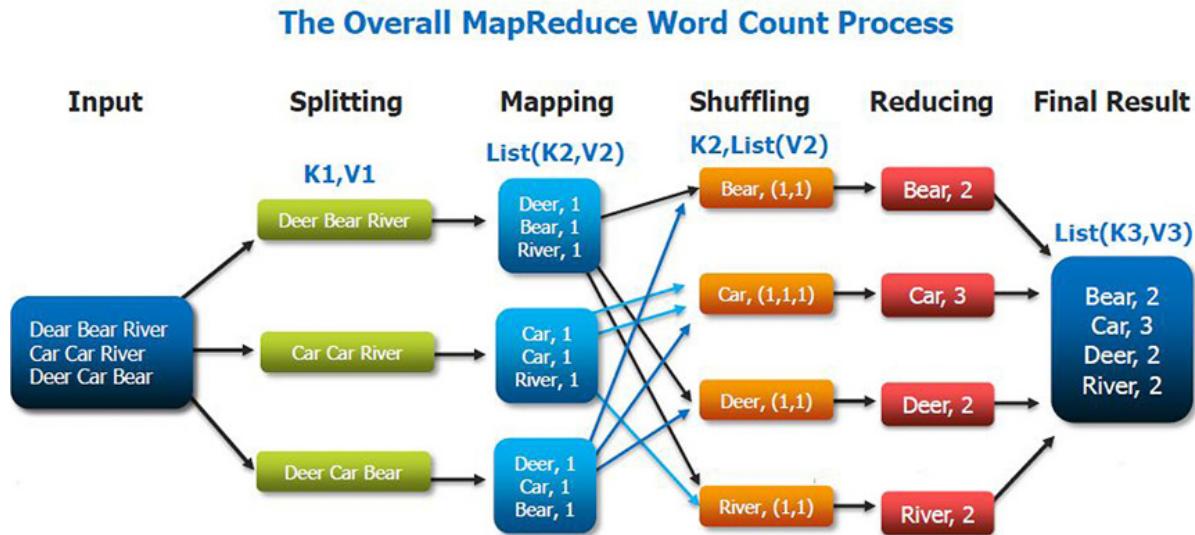
MapReduce is a programming model and framework designed for processing and analyzing large-scale data in a distributed and parallel manner. The process in MapReduce can be summarized in two main phases: the Map phase and the Reduce phase.

In the Map phase, the input data is divided into smaller chunks, and a set of map tasks are executed in parallel across a cluster of machines. Each map task takes a chunk of data, processes it, and generates a set of intermediate key–value pairs. These key–value pairs are often used to extract and filter relevant information from the input data.

In the Reduce phase, the intermediate key-value pairs generated by the Map phase are sorted, shuffled, and grouped based on their keys. Then, a set of reduce tasks are executed in parallel to process these groups of key–value pairs. These tasks can perform operations like aggregation, summarization, or further processing of the data. The final output of the Reduce phase is typically the desired result of the MapReduce job, which can be stored or used for subsequent analysis. MapReduce is a powerful and scalable approach for

handling big data processing tasks, as it leverages distributed computing resources to efficiently process and extract valuable insights from large datasets. Here is an example of MapReduce in a simple manner.

The wordcount process is shown in the following figure:



*Figure 6.1: Wordcount process*

Let us start with the word count program process. When dealing with large data, two main challenges arise: reading the data and processing it.

Traditionally, reading the entire file once and manually dividing it can be inconvenient. However, Hadoop provides a convenient way to automatically read files regardless of their size. The file is read line by line using offsets and line values. *Figure 6.1* illustrates the program that reads the file with its offset value. These offset values and their corresponding lines are called key–value pairs of data, with the offset becoming the key and the whole line becoming the value. The file processing involves six basic steps from the beginning to the result. To process a file, follow the given steps:

1. Input the file for processing. The file contains a group of lines, each containing a key–value pair of data. Using this method, the entire file can be read.
2. In the next step, the file will be in splitting mode. This mode will divide the file into key–value pairs of data. The key will be the offset, and the

data will be the value part of the program. Each line will be read individually, eliminating the need to split the data manually.

3. This step involves processing the value of each line along with the associated counting number. Each individual item separated by a space is counted, and that number is written with each key. This process is known as **mapping**, and the programmer needs to write the logic for it.
4. The shuffling process is performed, where each key is associated with a group of numbers that were generated during the mapping step. Now, the scenario becomes a key with a **string**, and the value will be a list of numbers. This data will be sent as input to the reducer.
5. In the reducer phase, whole numbers are counted, and each key is associated with final counting, which is the sum of all numbers that leads to the result.
6. Output of the reducer phase will lead to results. This result contains individual wordcount.

The wordcount scenario is independent of the size of the file used for processing. It can handle any size of data in a similar manner. This concept forms the foundation of many data processing techniques.

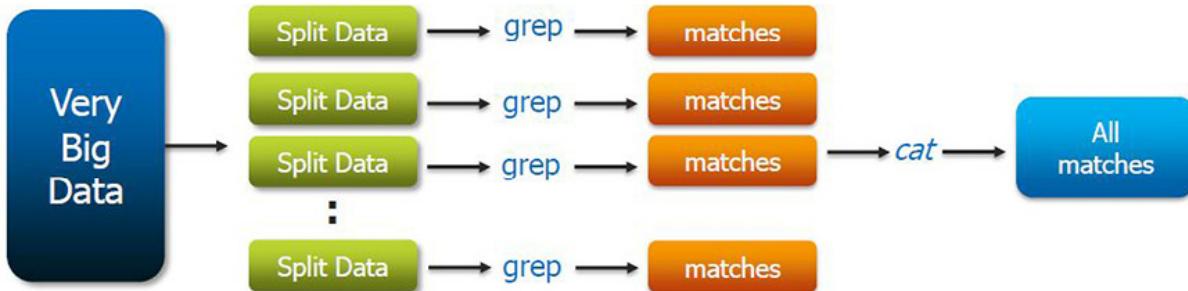
## **Traditional way**

Traditionally, before the advent of MapReduce and distributed data processing frameworks, data processing was primarily done using sequential or single-machine approaches. In this traditional way, data would be stored on a single server, or a limited number of machines, and processing tasks would be executed sequentially on that machine. This approach severely limits the scalability and efficiency of data processing, especially when dealing with large volumes of data.

In the traditional method, as data grew, organizations often faced challenges such as long processing times, resource constraints, and the risk of hardware failures disrupting data processing tasks. Furthermore, it was challenging to parallelize data processing effectively, making it unsuitable for big data applications. As a result, the traditional approach struggled to meet the demands of modern data-intensive tasks, leading to the development and

adoption of distributed frameworks like MapReduce, which revolutionized the way organizations process and analyze large datasets by enabling parallel, scalable, and fault-tolerant data processing.

*Figure 6.2* depicts the traditional method of file processing using techniques other than MapReduce. Take a look at the following figure:



*Figure 6.2: Traditional way*

In the traditional approach, when dealing with a large amount of data, it must be manually split into separate portions. However, this manual splitting may lead to data loss. To match the required records, the dataset is split using the **grep** command in Linux. After matching all the records, each individual record needs to be collected in the same place for further use, which is accomplished using Linux commands like **cat**. However, this process introduces the possibility of data loss during the collection phase.

## MapReduce workflow

The shuffling operation is automated and depends on the Hadoop configuration settings. The following code snippet shows a wordcount program with mapper and reducer functions:

```
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
```

```
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.mapreduce.Job;
import
org.apache.hadoop.mapreduce.lib.input.TextInputFormat
;
import
org.apache.hadoop.mapreduce.lib.output.TextOutputForm
at;
import
org.apache.hadoop.mapreduce.lib.input.FileInputFormat
;
import
org.apache.hadoop.mapreduce.lib.output.FileOutputForm
at;
import org.apache.hadoop.fs.Path;

public class WordCount {
    public static class Map extends
Mapper<LongWritable,Text,Text,IntWritable>{
        public void map(LongWritable key, Text value,
Context context)
            throws IOException,InterruptedException {
            String line = value.toString();
            StringTokenizer tokenizer = new
StringTokenizer(line);
```

```
        while (tokenizer.hasMoreTokens()) {
            value.set(tokenizer.nextToken());
            context.write(value, new IntWritable(1));
        }
    }

    public static class Reduce extends
Reducer<Text, IntWritable, Text, IntWritable>{
        public void reduce(Text key,
Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
            int sum=0;
            for(IntWritable x: values)
            {
                sum+=x.get();
            }
            context.write(key, new IntWritable(sum));
        }
    }

    public static void main(String[] args) throws
Exception {
    Configuration conf= new Configuration();
    Job = new Job(conf, "wordcount");
```

```

        job.setJarByClass(WordCount.class);

        job.setMapperClass(Map.class);
//conf.setMapperClass(Map.class);

        job.setReducerClass(Reduce.class);
//conf.setReducerClass(Reduce.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

job.setInputFormatClass(TextInputFormat.class);

        job.setOutputFormatClass(TextOutputFormat.class);

        Path outputPath = new Path(args[1]);
//Configuring the input/output path from the
filesystem into the job

        FileInputFormat.addInputPath(job, new
Path(args[0]));

        FileOutputFormat.setOutputPath(job, new
Path(args[1]));

//deleting the output path automatically from hdfs so
that we don't have delete it explicitly

outputPath.getFileSystem(conf).delete(outputPath);

        //exiting the job only if the flag value becomes
false

        System.exit(job.waitForCompletion(true) ? 0 : 1);

    }

}

```

The output is shown in the following figures:

```

hduser@mayank-Compaq-510:/home/mayank$ hadoop jar WC1.jar /wordcount /wordcountout
17/09/08 16:25:06 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
17/09/08 16:25:07 INFO Configuration.deprecation: session.id is deprecated. Instead, use dfs.metrics.session-id
17/09/08 16:25:07 INFO jvm.JvmMetrics: Initializing JVM Metrics with processName=JobTracker, sessionId=
17/09/08 16:25:07 INFO jvm.JvmMetrics: Cannot initialize JVM Metrics with processName=JobTracker, sessionId= - already initialized
17/09/08 16:25:07 WARN mapreduce.JobSubmitter: Hadoop command-line option parsing not performed. Implement the Tool interface and execute your application with ToolRunner to remedy this.
17/09/08 16:25:07 INFO mapred.FileInputFormat: Total input paths to process : 1
17/09/08 16:25:08 INFO mapreduce.JobSubmission: number of splits:1
17/09/08 16:25:08 INFO mapreduce.JobSubmission: Submitting tokens for job: job_local889866976_0001
17/09/08 16:25:08 INFO mapreduce.Job: The url to track the job: http://localhost:8080/
17/09/08 16:25:08 INFO mapreduce.Job: Running job: job_local889866976_0001
17/09/08 16:25:08 INFO mapred.LocalJobRunner: OutputCommitter set in config null
17/09/08 16:25:08 INFO mapred.LocalJobRunner: OutputCommitter is org.apache.hadoop.mapred.FileOutputCommitter
17/09/08 16:25:08 INFO mapred.LocalJobRunner: Waiting for map tasks
17/09/08 16:25:08 INFO mapred.LocalJobRunner: Starting task: attempt_local889866976_0001_m_000000_0
17/09/08 16:25:08 INFO mapred.Task: Using ResourceCalculatorProcessTree: []
17/09/08 16:25:09 INFO mapred.MapTask: Processing split: hdfs://localhost:54310/wordcount:0+48836
17/09/08 16:25:09 INFO mapred.MapTask: numReduceTasks: 1
17/09/08 16:25:09 INFO mapred.MapTask: (EQUATOR) 0 kvi 26214396(104857584)
17/09/08 16:25:09 INFO mapred.MapTask: mapreduce.task.io.sort.mb: 100
17/09/08 16:25:09 INFO mapred.MapTask: soft limit at 83886080
17/09/08 16:25:09 INFO mapred.MapTask: bufstart = 0; bufvoid = 104857600
17/09/08 16:25:09 INFO mapred.MapTask: kvstart = 26214396; length = 6553600
17/09/08 16:25:09 INFO mapred.MapTask: Map output collector class = org.apache.hadoop.mapred.MapTask$MapOutputBuffer
17/09/08 16:25:09 INFO mapred.LocalJobRunner: 
17/09/08 16:25:09 INFO mapred.MapTask: Starting flush of map output
17/09/08 16:25:09 INFO mapred.MapTask: Spilling map output
17/09/08 16:25:09 INFO mapred.MapTask: bufstart = 0; bufend = 77987; bufvoid = 104857600
17/09/08 16:25:09 INFO mapred.MapTask: kvstart = 26214396(104857584); kvend = 26184992(104739968); length = 29405/6553600
17/09/08 16:25:09 INFO mapred.MapTask: Finished spill 0
17/09/08 16:25:09 INFO mapreduce.Job: Job job_local889866976_0001 running in uber mode : false
17/09/08 16:25:09 INFO mapreduce.Job: map 0% reduce 0%
17/09/08 16:25:09 INFO mapred.Task: Task:attempt_local889866976_0001_m_000000_0 is done. And is in the process of committing
17/09/08 16:25:09 INFO mapred.LocalJobRunner: hdfs://localhost:54310/wordcount:0+48836
17/09/08 16:25:09 INFO mapred.Task: Task 'attempt_local889866976_0001_m_000000_0' done.
17/09/08 16:25:09 INFO mapred.LocalJobRunner: Finishing task: attempt_local889866976_0001_m_000000_0
17/09/08 16:25:09 INFO mapred.LocalJobRunner: Map task executor complete.
17/09/08 16:25:09 INFO mapred.LocalJobRunner: Waiting for reduce tasks

```

*Figure 6.3: Output of wordcount program-I*

Take a look at the following figure that displays the output:

```

17/09/08 16:25:11 INFO mapreduce.Job: map 100% reduce 100%
17/09/08 16:25:11 INFO mapreduce.Job: Job job_local889866976_0001 completed successfully
17/09/08 16:25:11 INFO mapreduce.Job: Counters: 38
    File System Counters
        FILE: Number of bytes read=192298
        FILE: Number of bytes written=784121
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
        HDFS: Number of bytes read=97672
        HDFS: Number of bytes written=3799
        HDFS: Number of read operations=13
        HDFS: Number of large read operations=0
        HDFS: Number of write operations=4
    Map-Reduce Framework
        Map input records=103
        Map output records=7352
        Map output bytes=77987
        Map output materialized bytes=92697
        Input split bytes=84
        Combine input records=0
        Combine output records=0
        Reduce input groups=351
        Reduce shuffle bytes=92697
        Reduce input records=7352
        Reduce output records=351
        Spilled Records=14704
        Shuffled Maps =1
        Failed Shuffles=0
        Merged Map outputs=1
        GC time elapsed (ms)=90
        CPU time spent (ms)=0
        Physical memory (bytes) snapshot=0
        Virtual memory (bytes) snapshot=0
        Total committed heap usage (bytes)=346554368
    Shuffle Errors
        BAD_ID=0
        CONNECTION=0
        IO_ERROR=0
        WRONG_LENGTH=0
        WRONG_MAP=0
        WRONG_REDUCE=0

```

**Figure 6.4:** Output of wordcount program-II

## Map side

Programmers develop the logic for the mapper class in a chosen programming language, which then directly interacts with the data. The mapper takes input data from the file and begins processing it, producing intermediate results that are passed on for further processing. The following are the key aspects of how Hadoop internally handles this process:

- Each map task has a circular memory buffer that it uses to write to output. This buffer is 100 MB by default and can change by the user from **io.sort.spill.percent**.
- By default, 80% of the output generated by the mapper will be written to a buffer. There is the threshold of buffer; when it reaches the threshold, the background thread starts to spill the contents to disk.

- The map output continues to write data to the buffer, and if the buffer fills up, the map blocks until the spill is complete.
- Spills are filled in a round-robin fashion that is specified in the **Hadoop** directory **mapred.local.dir** property.
- Data is divided into partitions before writing it to disk, and this division will be sent to reducers.
- In each partition thread, execute a **sort** operation based on the key. If a combiner function is used, it will run on the output of the **sort** operation.
- When a spill file is full, a new spill file is created, allowing the map task's output to be written into multiple spill files.
- There is a requirement to analyze or process data in a specific order, such as finding the top N elements, identifying trends over time, or performing range-based queries. Sorting ensures that data is organized and presented in a structured manner, allowing MapReduce tasks, particularly the Reduce phase, to work efficiently and **accurately**. **Sorting** done in an intermediate stage can be configured by **io.sort.factor**. By default, it is 10.
- If there are three spill files, then the combiner is run before the output is created and written into a disk.
- The output of the map can be compressed before sending it to reducer, which makes significant changes in transferring data. It can be done manually from **mapred.map.output.compression.codec**. It is not compressed by default.
- Output of file's partitioner for the reducer that is controlled by **TaskTracker.http.threads** property, it is for individual task trackers.
- The maximum number of threads used depends on the processor of the machine.

## **Reduce side**

The reduce part takes the data from the map as input and processes it for the final output. Since all map tasks may take different timings to finish their tasks, the reducer waits for all map tasks to complete before proceeding. The reducer

processes the data from multiple map tasks, performs the necessary operations, and produces the final output. Here are the key aspects of the reduce side:

- The *Copy phase* of the reduce task refers to the period during which the reduce task waits for the map task outputs to complete.
- The reducer task consists of a small number of threads of the copier responsible for fetching map outputs as input. By default, this number is set to five until modified using the configuration property **mapred.reduce.parallel.copies**.
- The reducer task's JVM memory is used to copy the map outputs if the size is small enough to fit into the buffer; otherwise, it is copied to disk. The threshold value for the buffer can be adjusted using the property **mapred.inmem.merge.threshold**, and if exceeded, the data is spilt to disk.
- When the copies are written to disk, threads merge them into larger segments to efficiently sort the file. This process optimizes the time taken for file merging.
- All map outputs move to the **sort** phase, where they are merged with other map outputs while maintaining the sort ordering based on the keys.
- If there are 50 map outputs and the merge factor is set to 10, there will be 5 rounds of merging. Each round will merge 10 files into one, resulting in 5 intermediate files.
- The final merge can consist of a mixture of in-memory and on-disk segments, depending on the available memory and data size.

The following table contains the name of the property, its type and default values associated with it:

Property name	Type	Default value
<code>io.sort.mb</code>	<code>int</code>	100
<code>io.sort.record.percent</code>	<code>float</code>	0.05
<code>io.sort.spill.percent</code>		

	float	0.80
io.sort.factor	int	10
min.num.spills.for.combine	int	3
TaskTracker.http.thread	int	40
mapred.reduce.parallel.copies	int	5
mapred.reduce.copy.backoff	int	300
io.sort.factor	int	10

**Table 6.1:** Map and Reduce tuning properties

## Sample program using MapReduce

Every data processing program must undergo two essential steps, namely, Map and Reduce. The context is used to generate the output, which represents the keyword.

To find the maximum temperature from a large amount of mixed data, use the following code:

```
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
```

```

import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import
org.apache.hadoop.mapreduce.lib.input.FileInputFormat
;
import
org.apache.hadoop.mapreduce.lib.input.TextInputFormat
;
import
org.apache.hadoop.mapreduce.lib.output.FileOutputForm
at;
import
org.apache.hadoop.mapreduce.lib.output.TextOutputForm
at;

public class Max_temp {
    public static class Map extends
Mapper<LongWritable, Text, Text, IntWritable> {
        Text k= new Text();

```

### **Testing mapper class**

The following is the provided code for a mapper, which focuses on data collection and initiates the processing:

```

@Override
public void map(LongWritable key, Text value, Context
context)
throws IOException, InterruptedException {

```

```
//Converting the record (single line) to string and
storing it in a String variable line
String line = value.toString();

//StringTokenizer is breaking the record (line)
according to the delimiter whitespace

StringTokenizer tokenizer = new
StringTokenizer(line, " ");

//Iterating through all the tokens and forming the
key value pair

while (tokenizer.hasMoreTokens()) {

//The first token is going in year variable of type
string

String year= tokenizer.nextToken();

k.set(year);

//Takes next token and removes all the
whitespaces around it and then stores it in the
string variable called temp

String temp= tokenizer.nextToken().trim();

//Converts string temp into integer v

int v = Integer.parseInt(temp);

//Sending to output collector which inturn passes
the same to reducer

context.write(k,new IntWritable(v));

}
```

```
}
```

```
}
```

## Testing reducer class

The following reducer class. It gathers data from the mapper as key–value pairs and processes it to produce the output. Execute the following code:

```
//Reducer

@Override

public void reduce(Text key, Iterable<IntWritable>
values, Context context)

    throws IOException, InterruptedException {

    //Defining a local variable maxtemp of type int
    int maxtemp=0;

    for(IntWritable it : values) {

        //Defining a local variable temperature of type
        int which is taking all the temperature
        int temperature= it.get();

        if(maxtemp<temperature)
        {
            maxtemp =temperature;
        }
    }

    //Finally the output is collected as the year and
    maximum temperature corresponding to that year
}
```

```
    context.write(key, new IntWritable(maxtemp));  
}  
}
```

### Testing driver class of program

Following is the driver class that summarizes about **mapper** and **reducer** classes in the program to the compiler. This information is collected by **NameNode**. Execute the following code:

```
//Driver  
  
public static void main(String[] args) throws  
Exception {  
  
    //reads the default configuration of cluster from  
    //the configuration xml files  
  
    Configuration conf = new Configuration();  
  
    //Initializing the job with the default  
    //configuration of the cluster  
  
    Job job = new Job(conf, "Max_temp");  
  
    //Assigning the driver class name  
  
    job.setjarByClass(Max_temp.class);  
  
    //Defining the mapper class name  
  
    job.setMapperClass(Map.class);  
  
    //Defining the reducer class name  
  
    job.setReducerClass(Reduce.class);  
  
    //Defining the output key class for the final  
    //output i.e. from reducer  
  
    job.setOutputKeyClass(Text.class);
```

```
//Defining the output value class for the final  
output i.e. from reducer  
  
job.setOutputValueClass(IntWritable.class);  
  
//Defining input Format class which is  
responsible to parse the dataset into a key value  
pair  
  
job.setInputFormatClass(TextInputFormat.class);  
  
//Defining output Format class which is responsible  
to parse the final key-value output from MR framework  
to a text file into the hard disk  
  
job.setOutputFormatClass(TextOutputFormat.class);  
  
//setting the second argument as a path in a path  
variable  
  
Path outputPath = new Path(args[1]);  
  
//Configuring the input/output path from the  
filesystem into the job  
  
FileInputFormat.addInputPath(job, new  
Path(args[0]));  
  
FileOutputFormat.setOutputPath(job, new  
Path(args[1]));  
  
//deleting the output path automatically from  
hdfs so that we don't have delete it explicitly  
  
outputPath.getFileSystem(conf).delete(outputPath);  
  
//exiting the job only if the flag value becomes  
false  
  
System.exit(job.waitForCompletion(true) ? 0 : 1);
```

}

}

## Test output of program

This program provides output as shown in the following figures:

1900	46
1901	48
1902	49
1903	35
1904	46
1905	35
1906	32
1907	49
1908	44
1909	38
1910	47
1911	48
1912	44
1913	43
1914	49
1915	49
1916	18
1917	35
1918	49
1919	42
1920	47

*Figure 6.5: Final output of data*

The output is shown in the following figure:

```

hduser@mayank-Compaq-510:/home/mayank$ hadoop jar maxtemp.jar /Tempinput /Tempoutput
17/09/08 18:47:11 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
17/09/08 18:47:12 INFO Configuration.deprecation: session.id is deprecated. Instead, use dfs.metrics.session-id
17/09/08 18:47:12 INFO jvm.JvmMetrics: Initializing JVM Metrics with processName=JobTracker, sessionId=
17/09/08 18:47:12 WARN mapreduce.JobSubmitter: Hadoop command-line option parsing not performed. Implement the Tool interface and execute your application with ToolRunner to remedy this.
17/09/08 18:47:13 INFO input.FileInputFormat: Total input paths to process : 1
17/09/08 18:47:13 INFO MapReduce.JobSubmitter: NUMBER OF SPLITS:1
17/09/08 18:47:13 INFO mapred.JobSubmitter: Submitting tokens for job: job_local586826758_0001
17/09/08 18:47:13 INFO mapreduce.Job: The url to track the job: http://localhost:8080/
17/09/08 18:47:13 INFO mapreduce.Job: Running job: job_local586826758_0001
17/09/08 18:47:13 INFO mapred.LocalJobRunner: OutputCommitter set in config null
17/09/08 18:47:13 INFO mapred.LocalJobRunner: OutputCommitter is org.apache.hadoop.mapreduce.lib.output.FileOutputCommitter
17/09/08 18:47:13 INFO mapred.LocalJobRunner: Waiting for map tasks
17/09/08 18:47:13 INFO mapred.LocalJobRunner: Starting task: attempt_local586826758_0001_m_000000_0
17/09/08 18:47:14 INFO mapred.Task: Using ResourceCalculatorProcessTree : []
17/09/08 18:47:14 INFO mapred.MapTask: Processing split: hdfs://localhost:54310/Tempinput:0+9360
17/09/08 18:47:14 INFO mapred.MapTask: (EQUATOR) 0 kvi 26214396(104857584)
17/09/08 18:47:14 INFO mapred.MapTask: mapreduce.task.io.sort.mb: 100
17/09/08 18:47:14 INFO mapred.MapTask: soft limit at 83886080
17/09/08 18:47:14 INFO mapred.MapTask: bufstart = 0; bufvoid = 104857600
17/09/08 18:47:14 INFO mapred.MapTask: kvstart = 26214396; length = 6553600
17/09/08 18:47:14 INFO mapred.MapTask: Map output collector class = org.apache.hadoop.mapred.MapTask$MapOutputBuffer
17/09/08 18:47:14 INFO mapred.LocalJobRunner:
17/09/08 18:47:14 INFO mapred.MapTask: Starting flush of map output
17/09/08 18:47:14 INFO mapred.MapTask: Spilling map output
17/09/08 18:47:14 INFO mapred.MapTask: bufstart = 0; bufend = 10791; bufvoid = 104857600
17/09/08 18:47:14 INFO mapred.MapTask: kvstart = 26214396(104857584); kvend = 26209604(104838416); length = 4793/6553600
17/09/08 18:47:14 INFO mapred.MapTask: Finished spill 0
17/09/08 18:47:14 INFO mapred.Task: Taskattempt_local586826758_0001_m_000000_0 is done. And is in the process of committing
17/09/08 18:47:14 INFO mapreduce.Job: Job job_local586826758_0001 running in uber mode : false
17/09/08 18:47:14 INFO mapreduce.Job: map 0% reduce 0%

```

*Figure 6.6: Output of temperature program-I*

The output is given in the following figure:

```

17/09/08 18:47:15 INFO mapreduce.Job: map 100% reduce 100%
17/09/08 18:47:15 INFO mapreduce.Job: Job job_local586826758_0001 completed successfully
17/09/08 18:47:15 INFO mapreduce.Job: Counters: 38
    File System Counters
        FILE: Number of bytes read=35552
        FILE: Number of bytes written=548121
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
        HDFS: Number of bytes read=18720
        HDFS: Number of bytes written=912
        HDFS: Number of read operations=13
        HDFS: Number of large read operations=0
        HDFS: Number of write operations=6
    Map-Reduce Framework
        Map input records=1199
        Map output records=1199
        Map output bytes=10791
        Map output materialized bytes=13195
        Input split bytes=97
        Combine input records=0
        Combine output records=0
        Reduce input groups=114
        Reduce shuffle bytes=13195
        Reduce input records=1199
        Reduce output records=114
        Spilled Records=2398
        Shuffled Maps =1
        Failed Shuffles=0
        Merged Map outputs=1
        GC time elapsed (ms)=166
        CPU time spent (ms)=0
        Physical memory (bytes) snapshot=0
        Virtual memory (bytes) snapshot=0
        Total committed heap usage (bytes)=571473920
    Shuffle Errors
        File Input Error=0
        File Output Error=0
        Record Reader Error=0
        Record Writer Error=0
        Partition Mismatch Error=0

```

*Figure 6.7: Output of temperature program-II*

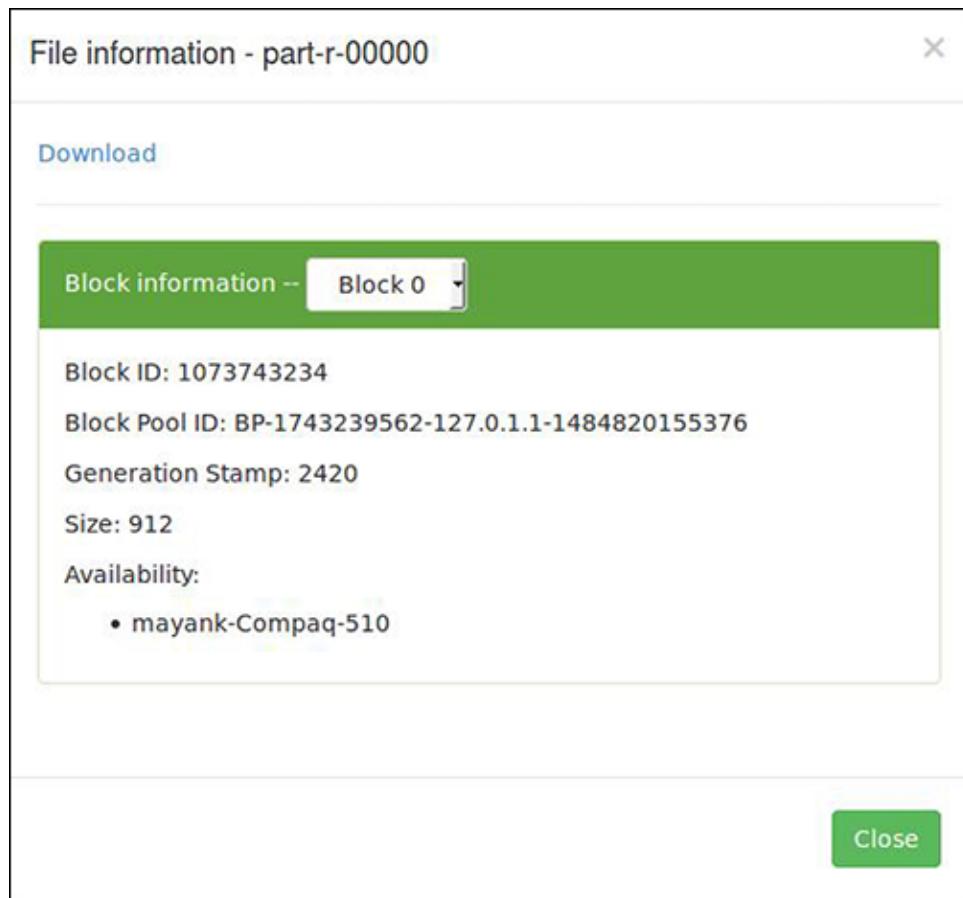
## Test data and local data check

Once the job is completed, it is stored in a distributed environment with the name **part-r-00000**, as depicted in *Figure 6.8*. The storage continues with subsequent numbers for each job. To verify the results, one can access the Hadoop User Interface by typing **localhost:50070** into the Web browser in a single-system cluster setup. The output window is shown as follows:

Permission	Owner	Group	Size	Replication	Block Size	Name
-rw-r--r--	hduser	supergroup	0 B	1	128 MB	_SUCCESS
-rw-r--r--	hduser	supergroup	912 B	1	128 MB	part-r-00000

*Figure 6.8: UI of Hadoop for NameNode*

The following figure shows the download block results:



**Figure 6.9:** Download block result

The result can be viewed by downloading it into the system, as shown in [Figure 6.9](#). This can be downloaded using block information. As discussed in [Chapter 2, NoSQL Data Management](#), with partitioner and combiner, the part can be customized with the user program.

## Introduction of Web UI

A Web User Interface (WebUI) of MapReduce jobs refers to a graphical interface, as shown in [Figures 6.8](#) and [6.9](#), that provides users with a visual representation of the progress, status, and details of MapReduce jobs running on a Hadoop cluster. This interface typically displays essential information such as job status (for example, running, completed, and failed), the progress of map and reduce tasks, resource utilization, and job-specific logs and counters. The WebUI enhances the manageability and monitoring of MapReduce jobs, enabling users to troubleshoot issues, optimize performance, and gain insights into the execution of data processing tasks in real-time, making it a valuable

tool for Hadoop administrators and developers working with large-scale data processing workflows.

## Debugging MapReduce job

Debugging programs traditionally involve using print statements, which can also be applied in Hadoop. However, this method becomes complicated when dealing with programs running on numerous nodes, where it is difficult to locate and examine scattered output statements. To address this issue, in cases where an uncommon scenario is being searched for, it is recommended to use a correct statement to log to standard error, along with a message to update the task's status message. The Web UI can make this process easier. Additionally, creating a custom counter to count the number of records with implausible temperatures in the entire data set can provide valuable information on how to handle the situation. It is always recommended to ask whether a counter can be used to obtain the necessary information when attempting to debug a job. If the number of log information produced during debugging is too large, writing the information to the map's output for analysis and aggregation by the reduce can be a better approach. However, this approach usually requires structural changes to the program, so it is advisable to try other techniques first.

Alternatively, a program can be written to analyze the logs produced by the job.

It is also possible to add a debugger to the mapper using the following code:

```
public class MaxTemperatureMapper  
extends Mapper<LongWritable, Text, Text, IntWritable>  
{  
    enum Temperature {  
        OVER_100  
    }  
  
    private NcdcRecordParser parser = new  
    NcdcRecordParser();  
  
    @Override
```

```

    public void map(LongWritable key, Text value,
Context context)

        throws IOException, InterruptedException {
    parser.parse(value);
    if (parser.isValidTemperature()) {
        int airTemperature =
parser.getAirTemperature();
        if (airTemperature > 1000) {
            System.err.println("Temperature over 100
degrees for input: " + value);
            context.setStatus("Detected possibly corrupt
record: see logs.");
        }
        context.getCounter(Temperature.OVER_100).increment(1)
        ;
    }
    context.write(new Text(parser.getYear()), new
IntWritable(airTemperature));
}
}
}

```

In the case when the temperature exceeds 100°C, it is recommended to print a line to standard error containing the suspicious line and update the map's status message using the **setStatus()** method on **Context**, leading to its appearance in the log. It is also suggested to increment a counter, which in Java can be represented by a field of an **enum** type. To accomplish this, a single field **OVER\_100** can be defined in the program to keep track of the number of

records with a temperature above 100°C. After implementing this modification, the code needs to be recompiled, and the JAR file needs to be recreated before rerunning the job. While the job is running, it is advisable to check the tasks page.

## Job chaining and job control

Chaining multiple MapReduce jobs with Hadoop is a powerful technique to perform complex data processing tasks by breaking them down into a sequence of smaller, interconnected MapReduce jobs. This approach allows for a more modular and organized way of handling large-scale data analysis. Typically, the output data from the first MapReduce job becomes the input for the second job, and so on, creating a data processing pipeline. Each job in the chain can perform a specific transformation, aggregation, or analysis of the data, contributing to the overall goal of the data processing workflow.

One of the key advantages of job chaining in Hadoop is flexibility and scalability. Different MapReduce jobs in the chain can be designed to perform specialized tasks, enabling parallel processing and optimized resource utilization. Moreover, job chaining simplifies the debugging and maintenance of the data processing pipeline since each job is focused on a specific operation. This approach is particularly valuable in scenarios where the data analysis process involves multiple stages or complex dependencies between tasks, allowing organizations to efficiently process and extract insights from vast datasets while benefiting from Hadoop's distributed computing capabilities.

When there is over one job during a MapReduce workflow, the question arises: how to manage the roles so they are executed in order? There are many approaches, and it mainly depends on whether you have a linear chain of jobs or a lot of complex **Directed Acyclic Graph (DAG)** of jobs. The most effective strategy for a linear chain is to execute each job sequentially, ensuring that each job is successfully completed before proceeding to the next one:

```
JobClient.runJob(conf1);  
JobClient.runJob(conf2);
```

If a job fails, the **runJob( )** methodology can throw an **IOException**; thus, later jobs within the pipeline do not get executed. Depending on the application, you may wish to catch the exception and shut down any intermediate data made by any previous jobs.

For anything more complex than a linear chain, libraries are available to help orchestrate workflows.

The simplest option is the **JobControl** category within the **org.apache.hadoop.mapreduce.jobcontrol** package. An instance of **JobControl** represents a graph of jobs to be executed. To use it, you add the job configurations and specify the dependencies between jobs. **JobControl** runs in a separate thread and executes the jobs in the correct dependency order. You can also poll for progress and wait until all the jobs have finished.

## Anatomy of MapReduce job

In Hadoop 0.20, the execution details are determined by the **mapred.job.tracker** property. The Local job runner is used to configure and run a job on a single system. It runs the job using a single **Java Virtual Machine (JVM)**, which is optimized for testing and running with small datasets.

If the **mapred.job.tracker** property is configured with a port number separated by a colon, and it is treated as the **JobTracker** address. This allows the job to be executed on the specified Hadoop **JobTracker**. Following is the scenario to execute the anatomy of the job as in [\*Figure 6.10\*](#), which is about to execute with MapReduce plan, whereas **JobTracker** working has been discussed in [\*Chapter 2: NoSQL Data Management\*](#).

## Anatomy of file write

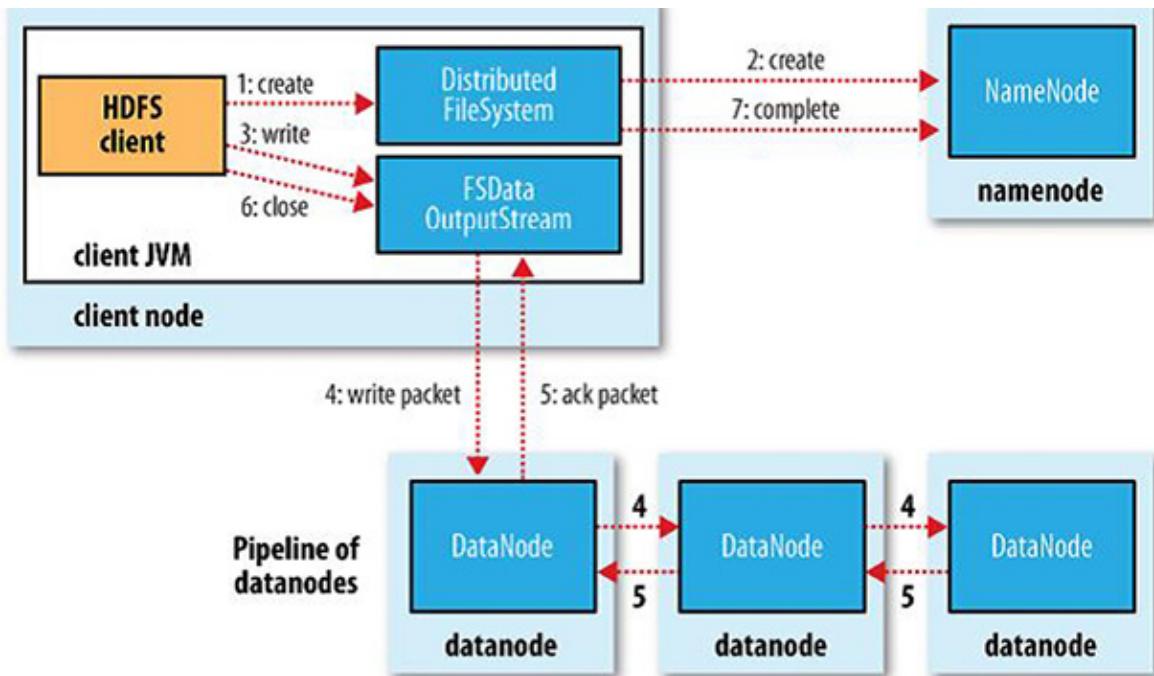
Following are the steps involved in writing a file in a distributed environment:

1. On **DistributedFileSystem** client creates the file by calling **create()**.
2. Through an RPC call, the **NameNode** initiates the creation of a new file in its file system, initially without associating any data blocks. Subsequently,

the **NameNode** performs several operations to verify the absence of duplicate records for the file. The client is granted permission to proceed with file creation only if no existing file shares the same name; otherwise, an exception is raised. The **DistributedFileSystem**, in conjunction with the **FSOutputStream**, facilitates this process by coordinating with various **DataNodes** and the **NameNode**. Together, they manage the creation of the file along with its metadata and data storage.

3. The **FSDataOutputStream** is responsible for creating a pipeline to the **DataNodes** and writing data to them. This is achieved through the creation of a data queue within the pipeline. The queue is then consumed by a Data Scanner, which handles block allocation and storage of replicas for the new data blocks.
4. All **datanodes** in the system use a pipeline structure to store data. Data is continuously associated with previous data and flows through the system, being stored in different **datanodes**. This pattern is consistently followed for data storage, creating a pipeline-like mechanism.
5. **FSDataStream** receives acknowledgments for all the data it processes through the **ackqueue**. This queue includes acknowledgment signals to indicate the successful receipt of data. However, if any **datanode** in the pipeline fails, the pipeline is closed, and the acknowledgement signals are halted to prevent further propagation.

The following figure illustrates the anatomy of a file write:



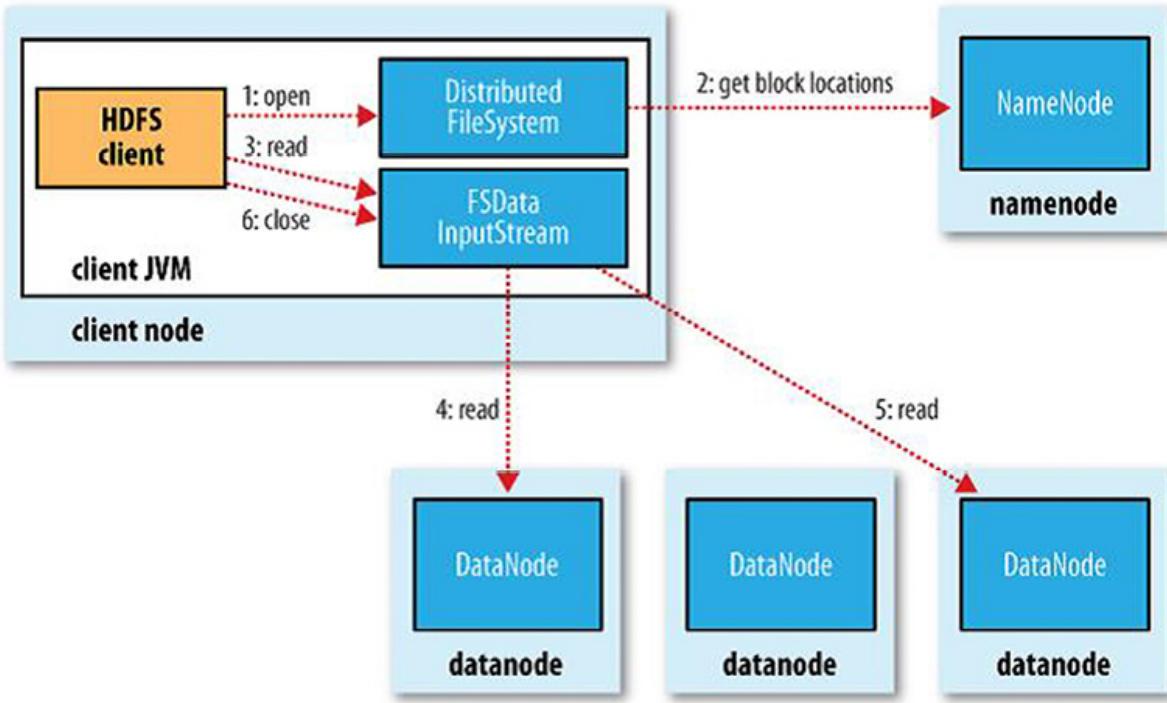
4

*Figure 6.10: Anatomy of file write*

- Once all the operations are completed, a signal is sent to the **NameNode** to indicate the completion of these operations. At this point, the **NameNode** possesses comprehensive information about each individual **datanode**, including the data it holds and its location within the distributed system. The **NameNode** continues to manage this information and updates it whenever new data or information needs to be saved within the distributed system.

## Anatomy of file read

*Figure 6.11* illustrates the process of file read in a distributed environment. In this scenario, a file is submitted and stored in the distributed storage. When there is a need to read the file, it is retrieved from the distributed storage for reading purposes. Unlike the writing process, which follows the pipeline method for data storage, reading can be directly performed from the client node without requiring a pipeline. The anatomy of a file read is shown in the following figure:



**Figure 6.11: Anatomy of file read**

Follow the following steps to read a file in distributed mode:

1. Initially, the client looks into the distributed mode to search for the required file that needs to be read. This is accomplished by calling the **open()** method.
2. After making the call, the client retrieves block details by fetching information from the **NameNode** using a **Remote Procedure Call (RPC)**. This information provides the program with specific details about the blocks.
3. The **DistributedFileSystem** responds by providing an **FSDataInputStream**, which the client uses to read the data. **FSInputStream** manages the I/O between the **datanodes** and the **NameNode**.
4. Once all the necessary information is collected, the client begins reading the data using the **read()** method. The **DFSInputStream** establishes a connection to the first closest **datanode** for the initial read, and then the data is streamed from the **datanode** for the subsequent blocks.

5. All the **datanodes** directly connect to the client, eliminating any unnecessary delays during the reading process and ensuring continuous streaming.
6. After all the connections are established and the reading is completed, the client calls the **close()** method to discontinue the connections from the client to the data.

During the reading process, if the **DFSInputStream** encounters an error while communicating with a **datanode**, it will attempt to connect with the nearest **datanode** for that specific block. To avoid unnecessary retries, it also keeps track of **datanodes** that have failed previously.

The **DFSInputStream** is responsible for verifying checksums for the data received from the **datanode**. If a corrupted block is detected, the information is reported back to the **NameNode** before attempting to read a replica of the block from another **datanode**.

To optimize data retrieval, the client directly contacts **datanodes** as guided by the **NameNode** to access the best **datanode** for each block. This design enables HDFS to handle a large number of concurrent clients effectively, as the data traffic is distributed across all the **datanodes** in the cluster. Meanwhile, the **NameNode** focuses on servicing block location requests, efficiently storing them in memory, and does not handle data retrieval. This separation of responsibilities prevents the **NameNode** from becoming a bottleneck as the number of clients increases.

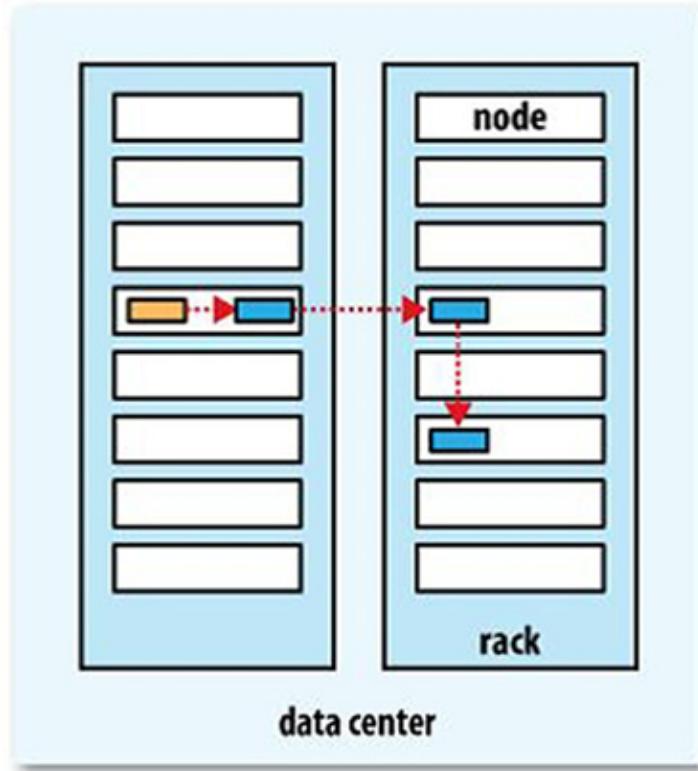
## Replica management

Datanode replication in Hadoop's **Hadoop Distributed File System (HDFS)** is a fundamental mechanism designed to enhance data reliability, fault tolerance, and data availability in distributed storage. Data is replicated to mitigate the risks associated with hardware failures, data corruption, or node unavailability. Each **datanode** stores multiple copies (typically three) of the same data block across different nodes in the cluster. If a **datanode** becomes unavailable due to hardware failure or other issues, the Hadoop system can seamlessly retrieve the required data from one of the replicated copies on another healthy **datanode**. This replication strategy ensures that data remains accessible even in the presence of hardware failures, thereby enhancing the overall robustness and

durability of the HDFS storage system, which is crucial for big data processing and analytics tasks.

When replicating a **datanode**, the decision of which **datanode** to use is determined by considering both network bandwidth and storage capacity. The placement strategy involves a trade-off between reliability and write/read bandwidth. For instance, placing all replicas on a single node incurs the lowest write bandwidth penalty but offers no redundancy in case of node failure. On the other hand, placing replicas in different data centers maximizes redundancy but reduces bandwidth. Even within the same data center, various placement methods can be used.

Hadoop modified its placement strategy in release 0.17.0 to ensure a more even distribution of blocks across the cluster. From 0.21.0, block placement policies can be customized. Newer versions of Hadoop, including 3.3.6, have introduced various performance enhancements, optimizations, and additional features to improve the efficiency, scalability, and reliability of data replication and management within the HDFS. Hadoop's HDFS continuously monitors the health of **datanodes**. If a **datanode** fails or becomes unavailable, HDFS automatically replicates the missing or under-replicated data blocks to maintain the desired replication factor, ensuring data availability and reliability. By default, the first replica is placed on the same node as the client, the second replica is placed on a different rack from the first, and the third replica is placed on the same rack as the second but on a different node. Additional replicas are placed on random nodes on the cluster while avoiding placing too many replicas on the same rack. Once the replica locations are chosen, a pipeline is created, taking network topology into account. For a replication factor of three, the pipeline might look like [\*Figure 6.12\*](#). This strategy balances reliability (blocks are stored on two racks), write bandwidth (writes only need to traverse one network switch), read performance (there is a choice of two racks to read from), and block distribution across the cluster (clients only write one block on the local rack). The following figure illustrates a sample replica storage:



*Figure 6.12: Sample replica storage*

## MapReduce job run

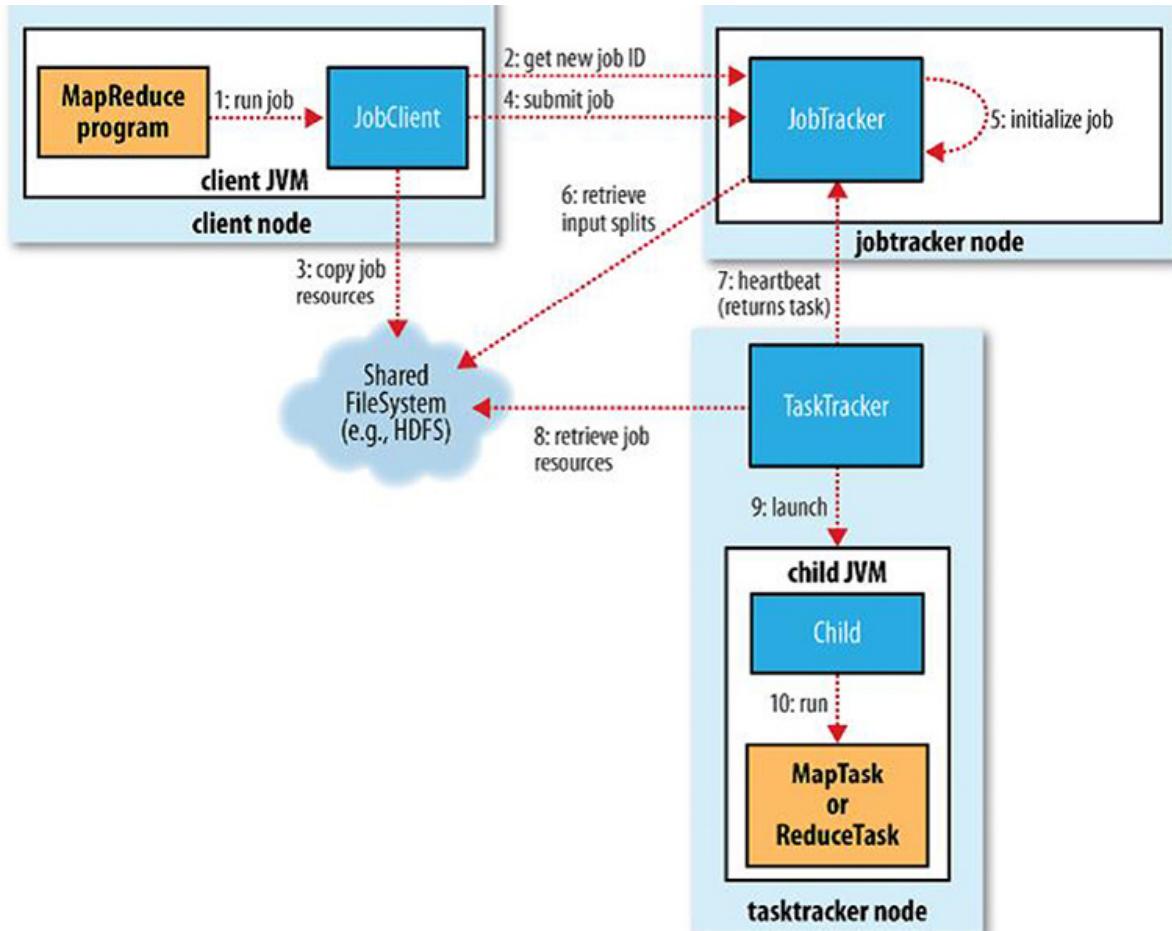
The job execution begins when the `submit()` method is called on the job object. If the job has not started execution, it will run `waitForCompletion()`. Additionally, the `Mapreduce.framework.name` specifies the framework used for the MapReduce job.

There are two versions of MapReduce: Classic MapReduce and YARN. In classic MapReduce, the `NameNode` stored all information of blocks and managed the resource manager responsibilities, resulting in a double load on a single system. To address this issue, the resource management work was delegated to another system called YARN.

## Classic MapReduce: MapReduce 1

In this mode of MapReduce, four entities are involved in the implementation, as shown in *Figure 6.13*: client, `JobTracker`, `TaskTracker`, and distributed file system. The client is responsible for submitting the MapReduce job for

execution. The **JobTracker** coordinates the job throughout the program. The **TaskTracker** is responsible for implementing the job into individual **datanodes**, taking information from the **NameNode**. The distributed file system is used to share the job with all entities involved. Take a look at the following figure:



*Figure 6.13: MapReduce1*

To understand the functioning of **MapReduce1** let us go through the following points:

- **Job submission:** The MapReduce program is submitted to the job client using the client JVM, which, in turn, sends it to HDFS. Upon successful completion of a job, the job counter is displayed; otherwise, an error is shown. The job submission process is managed by the **JobSubmitter** and follows these steps:
  - By calling **getNewJobId()** on **JobTracker**, it asks for a new job ID.

- It is mandatory to specify the output directory; otherwise, it throws an error and does not proceed further.
- Input splits must be computed; in case it is not specified, the program shows an error.
- To run a job, resources are needed. These resources are included in JAR files, that is, the configuration file. Each job is recognized with a job ID, which is included in the JAR file. This job ID is used to take the program into a distributed environment.
- The computation mentioned provides the job tracker with information about the job ID and its execution process.
- **Job initialization:** Upon submission, the jobs are placed in an internal queue known as the job queue through the **submitJob()** method. The **JobScheduler** then selects the next job to be executed from this queue. The subsequent steps for execution are as follows:
  - An object is created to encapsulate the task and store information about its status and progress.
  - The **JobTracker** collects input splits from the data, and the number of map tasks needed is based on the number of splits. Splits are logical divisions of the data.
  - The number of reduce tasks is determined by the program, with the default being one. It can be changed using the **mapred.reduce.tasks** property in the **setNumReduceTask()** method.
  - In addition to map and reduce tasks, two more tasks are created in parallel: the job setup task and the job cleanup task. **TaskTrackers** run these tasks to set up the job before any map task is executed.
- **Task assignment:** It is **TaskTracker**'s responsibility to send a heartbeat at regular intervals to **JobTracker** so that it can be considered alive. The subsequent steps for execution are as follows:
  - The heartbeat not only indicates that the task tracker is alive but also shows that it is ready to receive more data for execution and is available for the next instruction.

- The job tracker must select a job from the queue before selecting any task to execute. The execution of a job depends on the priority of the job.
- Task trackers have a fixed number of slots for executing tasks with map and reduce tasks. The total number of map and reduce tasks depends on the number of cores and the system's capacity.
- First, the map slots are filled before the reduce slots, if they are free; otherwise, the reduce slots are given priority to fill them.

As there is no data locality concept, the job tracker selects the next task to execute for reduce tasks. However, for map tasks, it depends on the task tracker's network location, and the nearest task is selected for execution.

Tasks can be data local, rack local, or neither. When data is on the same node, it is called data local, whereas data on the same rack is referred to as rack local. Some data may not be local to either the data node or the rack.

- **Task execution:** For executing the task on data, consider the following steps:
  - **TaskTracker**'s filesystem shares the JAR file, and the job JAR is copied with all the required information.
  - The JAR file is then un-JARed to extract all the necessary information needed for execution.
  - An instance of **TaskRunner** is created to run the task, and it starts the Java virtual machine to execute the individual task.
  - Communication between the child process and its parent is done through the umbilical interface, which informs the parent about the task's progress every second.
  - After completion of any task, the JVM performs cleanup tasks as determined by the **OutputCommitter** for the job.
  - A flag is set to indicate the status change, which is reported in the progress report. This flag is checked every three seconds with a thread.

- Meanwhile, a heartbeat is sent every five seconds by the **TaskTracker** to keep the system aware of its aliveness and readiness for the next instruction.
- **Job completion:** After completion of the job, it is said to be successful, and the status of the job is updated to completion and return **waitForCompletion()** method:
  - **JobTracker** sends an HTTP job notification if it is configured for receiving notifications by the client.
  - **Job.end.notification.url** property is for having all records of completion.

## Failure in MapReduce1

Let us start by examining what happens when a child's task fails. Typically, this occurs when the user code within the map or reduce task generates a runtime exception, causing the child JVM to report the error to its parent **TaskTracker** before exiting. The error is then recorded in the user logs, and the slot is assigned to another task when the **TaskTracker** marks it as a failure. In the case of **Streaming** tasks, if the **Streaming** method ends with a nonzero exit code, it is also marked as failed, a behavior that is controlled by the **stream.non.zero.exit.is.failure** property.

Another way in which child tasks can fail is when the child JVM abruptly exits due to a JVM bug that is triggered by the MapReduce user code's particular set of circumstances. In such cases, the **TaskTracker** marks the task as failed and exits. However, hanging tasks are handled differently. The **TaskTracker** identifies that it has not received a progress update for a while and marks the task as failed, which causes the child JVM process to be automatically killed after a timeout period. By default, the timeout period for tasks is ten minutes, but it can be customized on a per-job or cluster basis by setting the **mapred.task.timeout** property to a value in milliseconds. If the timeout is set to zero, then long-running tasks are never marked as failed, and a hanging task may never release its slot, leading to a cluster slowdown over time.

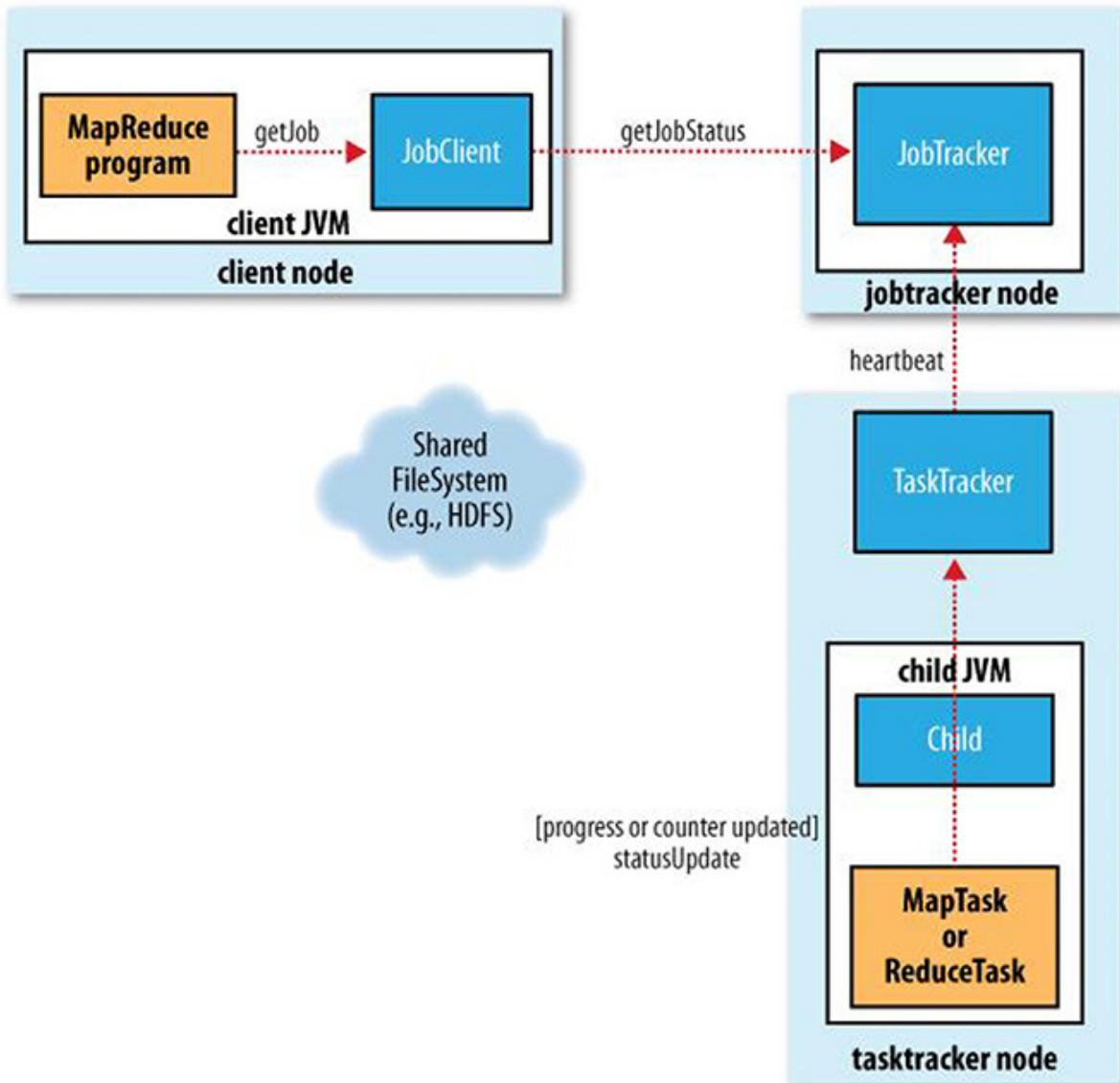
It is advisable to avoid relying solely on task retries to handle failures and instead ensure that tasks are reporting progress periodically. When a task attempt fails, and the **TaskTracker** notifies the **JobTracker** via a heartbeat

call, the **JobTracker** will reschedule the execution of the task. However, the **JobTracker** can try to avoid rescheduling the task on a **TaskTracker** where it has previously failed. Additionally, if a task fails four times or more, it will not be retried any further, as specified by the **mapred.map.max.attempts** property for map tasks and **mapred.reduce.max.attempts** for reduce tasks. By default, if any task fails four times or more, the entire job will fail. However, in some cases, it may be desirable to continue using the results of the job despite some failures. In such cases, the maximum percentage of allowed task failures without triggering job failure can be set for the job using the **mapred.max.map.failures.percent** and **mapred.max.reduce.failures.percent** properties, which control map tasks and reduce tasks independently.

It is worth noting that a task attempt may also be killed, which is different from failing. A task attempt may be killed because it is a speculative duplicate or because the **TaskTracker** it was running on failed, and the **JobTracker** marked all the task attempts running on it as killed. Killed task attempts do not count against the maximum number of attempts to run the task, as set by the **mapred.map.max.attempts** and **mapred.reduce.max.attempts** properties, as it was not the task's fault that the attempt was killed. Additionally, users can also kill or fail task attempts using the Web UI or command-line interface (using Hadoop job to see the options), and jobs can also be killed using the same mechanisms.

## MapReduce2 YARN

When dealing with many nodes in a system, resource management becomes a serious concern. The system must handle both processing tasks and resource management, which can lead to resource overload. To address this issue, MapReduce2, also known as **Yet Another Resource Negotiator (YARN)**, comes into play. YARN's primary purpose is to efficiently manage resources in a distributed environment. It separates the processing tasks from the resource management tasks, allowing for better resource utilization and more efficient processing. This approach is depicted in [Figure 6.14](#):



**Figure 6.14: MapReduce2 YARN**

Consider the following points regarding YARN:

- YARN splits the responsibilities of the **JobTracker** into two parts: job scheduling and task progress monitoring.
- It separates the roles of resource management and application master, managing the lifecycle of the running cluster and listing its resources.
- The application master synchronizes with the resource manager to handle large amounts of data efficiently and ensures that applications do not consume more resources than necessary.

- Each MapReduce job has its own application master that runs throughout the duration of the application.
- Map and reduce jobs are coordinated and run in a set of programs with task management.
- The distributed shell runs scripts on a set of nodes with one type of YARN application.
- Different versions of MapReduce can work with the same YARN application, allowing for better utilization and manageability.
- YARN involves entities such as the client, YARN resource manager, YARN node manager, MapReduce application master, and distributed file system.

YARN follows the given processes:

- 1. Job submission:** Similar to MapReduce1, YARN follows a similar process to submit jobs. It implements the **clientProtocol** for the specified MapReduce.framework.name. The Job ID is obtained from the resource manager to run the job. YARN generates splits for jobs using **yarn.app.mapreduce.am.compute-splits-in-cluster** and copies them to job resources through a JAR file that includes split information and configuration. Finally, the job is submitted to the resource manager using **submitApplication()**.
- 2. Job initialization:** Upon receiving a decision through **submitApplication()**, the resource manager hands off the request to the scheduler. The scheduler then allocates a container, and the resource manager launches the application master's method under the node manager's management. The application master for MapReduce jobs is a Java application with the main class **MRAppMaster**. Its main task is to initialize various **bookkeeping** objects to track the job's progress and receive progress and completion reports from the tasks.

Next, the application master retrieves the input splits computed by the client from the shared filesystem. For each split, it creates a map task object, and the number of reduce task objects is determined by the **mapreduce.job.reduces** property. The application master then decides how to run the tasks that constitute the MapReduce job.

In the case of small tasks, the application master may choose to run them within the same JVM as itself. This decision considers the overhead of allocating new containers and running tasks in parallel, weighing it against the benefits of running them consecutively on one node. This approach differs from **MapReduce1**, where small jobs are never run on one **TaskTracker**. Such a job is referred to as an uberized job or an uber task.

3. **Task assignment:** If the task does not qualify as an uber task, the application master requests containers for all the map and reduce tasks within the job from the resource manager. Each request, piggybacked on heartbeat calls, includes information about each map task's information locality, specifically the hosts and corresponding racks where the input split resides. The scheduler uses this data to make scheduling decisions, attempting to place tasks on data-local nodes ideally. If this is impossible, the scheduler prefers rack-local placement over non-local placement. The requests also specify the memory requirements for tasks. By default, each map and reduce task is allocated 1024 MB of memory, but this can be configured by setting `mapreduce.map.memory.mb` and `mapreduce.reduce.memory.mb` properties.

The memory allocation approach is different from **MapReduce1**, where **TaskTrackers** have fixed slots with a maximum memory allowance, leading to underutilization and job failures due to memory constraints. In YARN, resources are more fine-grained, avoiding these issues. Applications can request a memory capacity between the minimum and maximum allocation, with the default minimum allocation being 1,024 MB and the default maximum allocation being 10,240 MB. Tasks can request any memory allocation between 1 and 10 GB in multiples of 1 GB, and the hardware will round to the closest multiple if needed. This can be achieved by setting the appropriate `mapreduce.map.memory.mb` and `mapreduce.reduce.memory.mb` properties.

4. **Task execution:** The task is executed by a Java application whose main category is **YarnChild**. Before running the task, it localizes the resources required by the task, including the work configuration, JAR file, and any files from the distributed cache. After that, it runs the map or reduce task.

The **YarnChild** runs in a dedicated JVM for the same reason that **TaskTrackers** spawn new JVMs for tasks in **MapReduce1**: to isolate user code from long-running system daemons. However, unlike **MapReduce1**, YARN does not support JVM reuse, so every task runs in a new JVM. **Streaming** and **Pipes** programs work in the same way as **MapReduce1**. The **YarnChild** launches the **Streaming** or **Pipes** method and communicates with it using standard input/output or a socket, respectively.

## Failure in MapReduce 2

Failure of the running task is comparable to the classic case. Runtime exceptions and sudden exits of the JVM are propagated back to the application master, and therefore, the task try is marked as failing. Likewise, hanging tasks are noticed by the application master by the absence of a ping over the umbilical channel (the timeout is about by **mapreduce.task.timeout**), and once more, the task try is marked as failing. The configuration properties for crucial, once a task is failing, are the same because the classic case: a task is marked as failing when four make an attempt (set by **mapreduce.map.maxattempts** for map tasks and **mapreduce.reduce.maxattempts** for reducer tasks). Employment will be failing if more than **mapreduce.map.failures.maxpercent** per cent of the map tasks within the job fails, or quite **mapreduce.reduce.failures.maxpercent** per cent of the reduce tasks fail. Similar to MapReduce tasks, many make an attempt to succeed (in the face of hardware or network failures); applications in YARN are tried multiple times in the event of failure. By default, applications are marked as failing if they fail once; however, this will be increased by setting the property **yarn.resourcemanager.am.max-attempts**. An application master sends periodic heartbeats to the resource manager, and in the event of application master failure, the resource manager can find the failure and start a new instance of the master running in a very new container, and these settings are managed in `yarn-site.xml` (managed by a node manager). Within the case of the MapReduce application master, it will recover the state of the tasks that had already been passed the (failed) application in order that they do not have to be compelled to be rerun. By default, recovery is not enabled; therefore, failing

application masters will not rerun all their tasks; however, you can flip it on by setting `yarn.app.mapreduce.am.job.recovery.enable` to true.

The client polls the appliance master for progress reports; therefore, if its application master fails, the client has to find a new instance. Throughout job initialization, the consumer asks the resource manager for the application master's address and then caches it so it does not overload the resource manager with a request on every occasion it has to poll the application master. If the application master fails, however, the client can experience a timeout once it problems a status update; for that purpose, the client can return to the resource manager to ask for the new application master's address:

- **Job scheduling:** In earlier versions, Hadoop used a simple first-in first-out scheduler to order jobs submitted by users, with each job occupying the entire cluster. However, this approach presented problems in terms of fairly allocating resources between users, particularly as it prioritized jobs based on submission time rather than their size or complexity. To address these issues, Hadoop later introduced a priority system for jobs, but even high-priority jobs could still be blocked by low-priority jobs that were already running.

To provide users with a more equitable share of the cluster, Hadoop offers different scheduling options, including the original FIFO scheduler, the fair scheduler, and the capacity scheduler. (The Capacity Scheduler is a resource allocation and job scheduling mechanism in the Hadoop ecosystem, primarily used in Hadoop clusters. It is designed to efficiently allocate cluster resources among multiple users or applications based on configured capacities, queues, and user-defined policies). The fair scheduler is designed to allocate cluster resources fairly over time so that each user has a reasonable chance of getting their jobs completed in a timely manner. Jobs are placed into pools, with each user typically having their own pool, and free task slots are allocated to jobs in such a way as to ensure that no user is starved of resources. The scheduler also supports preemption, enabling it to kill tasks in over-capacity pools and allocate their slots to under-capacity pools.

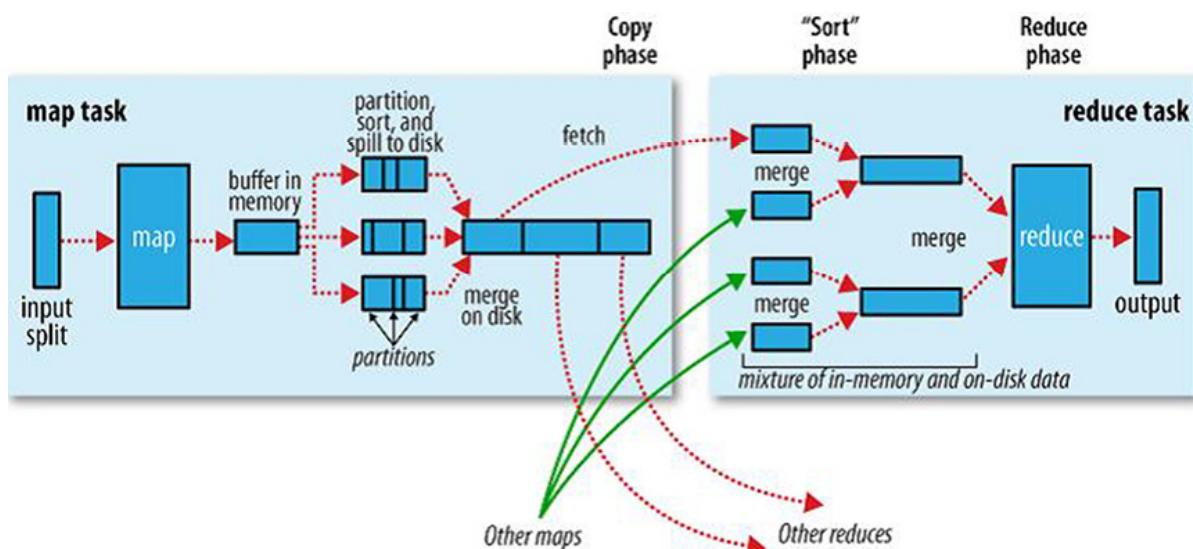
To use the fair scheduler, users need to enable the `contrib` module by copying its JAR file from Hadoop's `contrib/fairscheduler` directory to the lib directory and then set the

`mapred.JobTracker.taskScheduler` property to `org.apache.hadoop.mapred.FairScheduler`. The fair scheduler also allows users to define custom pools with specific minimum capacities and weightings for each pool.

- Shuffle and sort: In YARN-based MapReduce (**MapReduce2**), the Shuffle and Sort phases are decoupled from the MapReduce framework. YARN allows for more flexibility in how these phases are managed. The Shuffle and Sort operations are treated as separate components and can be customized or replaced with alternative implementations if needed. This decoupling provides greater control and optimization opportunities for organizations, allowing them to choose different Shuffle and Sort implementations that may be more suitable for their specific use cases. The fundamental concepts of Shuffle and Sort persist in both traditional MapReduce and YARN-based MapReduce (**MapReduce2**), but the latter offers more flexibility and modularity in how these phases are managed, making it a more versatile option for organizations with diverse data processing requirements.

## Map side

The function starts with the map part, and when it starts producing data, it writes its output to buffers. The capacity of this buffer is defined that is 100 MB by default. It can be further edited with `io.sort.spill.percent` property. Take a look at the following figure:



**Figure 6.15:** Shuffle and sort in MapReduce

The data spills generated during the **Map** phase in Hadoop are written to specific directories in a round-robin manner within a job-specific subdirectory as defined by the **mapred.local.dir** property. Prior to writing the data to disk, the thread responsible for the spill operation partitions the data based on the reducers to which they will be sent. Then, within each partition, an in-memory sort is performed on the data by key, and if a combiner function is present, it is executed on the sorted output.

### Reduce side

The map output file is initially located on the local disk of the machine that processed the map task. However, it is needed by the machine responsible for the reduce task for that partition. Since the map tasks may finish at different times, the reduce task starts copying their outputs as soon as each map task is completed. This copying process is known as the copy phase of the reduce task. To enable parallel fetching of map outputs, the reduce task has a small number of copier threads. By default, there are five threads, but this number can be adjusted using the **mapred.reduce.parallel.copies** property. If the map outputs are small enough, they are loaded into the reduce task's JVM memory. If not, they are copied to disk. When the in-memory buffer reaches a specified size or a specified number of map outputs (**mapred.inmem.merge.threshold**), it is merged and spilt to disk. If there is a combiner function, it can be applied during the merge to reduce the amount of data written to disk.

As the copied map outputs accumulate on disk, a background thread merges them into larger, sorted files. This process saves time during later merges. However, any compressed map outputs must be decompressed in memory before they can be merged. Once all the map outputs are copied, the reduce task moves to the **sort** phase, which merges the map outputs while preserving their sort order. The merge is performed in rounds, with each round merging a specified number of files. For example, if there are 50 map outputs and a merge factor of 10, there would be five rounds. Each round merges 10 files into one, resulting in five intermediate files.

Instead of having a final round that merges these five files into one sorted file, the merge saves a step by directly feeding the reduce function with what is

known as the last phase: the reduce phase. This final merge is created from a combination of in-memory and on-disk segments.

The general principle is to provide the shuffle with the maximum amount of memory possible. However, there is a trade-off in this map, and reduce functions get enough memory to control. This can be why it is best to write down your map and reduce functions to use as little memory as possible. Certainly, they must not use an infinite amount of memory. The amount of memory given to the JVMs within which the map and reduce tasks run is set by the **mapred.child.java.opts** property. On the reduce side, the best performance is obtained when the intermediate data can reside entirely in memory. By default, this does not happen because, for the general case, all the memory is reserved for the reduce function. However, if reduce perform has lightweight memory requirements, then setting **mapred.inmem.merge.threshold** to **0** and **mapred.job.reduce.input.buffer.percent** to **1.0**.

**Task execution:** Hadoop provides information to a map or reduce task about the environment in which it is running that has been described with sample programs. In this section, there is a discussion about more technicality of task execution.

The task execution method is defined in **configure()** method for the mapper and reducer. All objects can be passed to all methods in the mapper and reducer through APIs. Take a look at the following table:

Property name	Type
<code>mapred.job.id</code> (To see JOB ID and its format)	<code>String</code>
<code>mapred.tip.id</code> (To see TASK ID)	<code>String</code>
<code>mapred.task.id</code> (To see task attempt ID)	<code>String</code>
<code>mapred.task.partition</code> (No. of task in job)	<code>int</code>
<code>mapred.task.is.map</code> (Check about map task)	<code>boolean</code>

**Table 6.2:** Task execution

**Task JVM:** During Hadoop configuration, it is essential to install Java as well. Java is utilized to run its own virtual machine, ensuring tasks are isolated from one another. Initializing a new task takes a brief moment, allowing a large number of tasks to gain performance advantages for subsequent tasks. JVM runs all tasks sequentially, not concurrently, which means they run in parallel but with separate JVMs. The reuse of JVMs can be controlled using `mapred.job.reuse.jvm.num.tasks`, an integer property with a default value of 1. Additionally, the `NumTasksToExecutePerJvm()` function on `JobConf` can be used for configuration.

**Skipping bad records:** Handling big data poses several challenges, for example, dealing with messy data, and another challenge is the large amount of data itself. The following methods can be used to deal with bad records:

- Detect bad records and ignore them.
- Abort the job and throw an exception.
- Clean the data with the mentioned logic in the mapper and reducer.
- If the issue is with a third-party library, then use skipping mode for automatic skip of bad records.

Skipping mode in Hadoop involves reporting records back to the **TaskTracker** after processing. In the event of a task failure, the task is retried while skipping the problematic records. This approach is useful for handling occasional issues but is not a comprehensive solution for data analysis. For more effective data analysis, Hadoop allows bad records detected during processing to be saved as sequence files. These files are stored in the job's output directory under the `_logs/skip` subdirectory. Analysts can later inspect these saved files for analytical purposes once the job is completed, enabling them to identify patterns or problems that caused the failures.

**MapReduce input types:** Map and Reduce are the features of data analytics. Nothing can be processed without these components. MapReduce is based on the concept of collection in Java that is represented in the form of the following:

Map: (K1, V1) | list (K2, V2)

Reduce: (K2, list(V2)) | list (K3, V3)

Key and value pair is the base of data analysis. Map output is the same as the reduce input. The general format of map and reduce is as follows:

```
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
    public class Context extends MapContext<KEYIN,
VALUEIN, KEYOUT, VALUEOUT> {
        // ...
    }
    protected void map(KEYIN key, VALUEIN value,
                      Context context) throws IOException,
InterruptedException {
        // ...
    }
}
public class Reducer<KEYIN, VALUEIN, KEYOUT,
VALUEOUT> {
    public class Context extends ReducerContext<KEYIN,
VALUEIN, KEYOUT, VALUEOUT> {
        // ...
    }
    protected void reduce(KEYIN key, Iterable<VALUEIN>
values, Context context
                      Context context) throws IOException,
InterruptedException {
        // ...
    }
}
```

```
 }  
 }
```

In a MapReduce program, the placeholders **KEYIN** and **KEYOUT** can be replaced with specific data types that are used as input and output for the data. These data types can be chosen based on the programmer's needs and the requirements of the task at hand. The overall structure of the program can be defined as follows:

```
Map: (K1, V1)  list (K2, V2)  
Combine: (K2, list (V2))  list (K2, V2)  
Reduce: (K2, list (V2))  list (K3, V3)
```

The Combiner's output, denoted by **K3** and **V3**, remains the same as **K2** and **V2**, respectively. Therefore, the output of the **Combiner** serves as the input for the reducer. Additionally, the partitioner can also be used in the system with key-value pairs. It allows the data to be grouped and represented based on the key used.

**Input type:** The input types are determined using an input format, such as **TextInputFormat**, which generates **LongWritable** as the key and the entire line as the value (text). Alternatively, **IntWritable**, **FloatWritable**, and **DoubleWritable** can also be used as input with the whole line as output, which is represented as **Text**.

For analyzing text files, **KeyValueTextInputFormat** is used, whereas **SequenceFileInputFormat** is suitable for handling **Sequence** files.

The **Mapper** is a generic type that allows any key-value pair to be used in the program according to the programmer's needs. The sorting between the **Mapper** and **Reducer** is handled automatically.

Before reaching the **Reducer**, the partitioner maintains sorting based on the key-value pairs. By default, the **HashPartitioner** hashes records to determine the mapping of the partitioner and its records. The number of partitioners is equal to the number of reduce tasks for the job.

The default reducer is **Reducer**, again a generic type, which simply writes all its input to its output:

```
public class Reducer<KEYIN, VALUEIN, KEYOUT,  
VALUEOUT> {  
  
    protected void reduce(KEYIN key, Iterable<VALUEIN>  
values, Context context  
  
        Context context) throws IOException,  
InterruptedException {  
  
    for (VALUEIN value: values) {  
  
        context.write((KEYOUT) key, (VALUEOUT) value);  
  
    }  
  
}  
}
```

Some of the common input formats include the following:

- **TextInputFormat**: The default input format reads each line of the record as an input. The key represents the byte offset, which is the beginning of the line, whereas the value contains the entire line, including any terminations and special symbols. For instance, if the file contains the following data:

```
Hi, this is Big Data book  
Written by Mayank Bhushan  
It is practical approach of subject
```

This will be read as:

```
(0, Hi, This is Big Data book)  
(33, Written by Mayank Bhushan)  
(65, It is practical approach of subject)
```

Here, we can observe key and value pairs. Keys are the offset, and all lines are split into value parts. Offsets are the unique identification for each line.

- **KeyValueTextInputFormat**: This uses a tab separator by default. Separator property can be specified by `mapreduce.input.keyvaluelinerecordreader.key.value.separator` property.

(Line 1, Hi, This is Big Data book)

(Line 2, Written by Mayank Bhushan)

(Line 3, It is practical approach of subject)

- **NLineInputFormat**: When there is a need to receive a fixed number of lines as input, the **NLineInputFormat** is used. In this format, the keys represent the byte offset in the file, and the values are the lines of text.

**N** is set to one by default, and each mapper receives one line of input exactly.

(0, Hi, This is Big Data book)

(33, Written by Mayank Bhushan)

(65, It is practical approach of subject)

This takes as input depends on **N**. If **N** is set to 2 then lines used in mapper are 2 per mapper. But default, it is 1; this implies that in the preceding case, 3 mapper will be used to process it.

- **XML**: When dealing with large XML documents that span multiple splits, it is often better to use individual parsers for each split. If the XML document is not too large, a single mapper can be used. To process XML documents with start and end tags of records, the **StreamXmlRecordReader** class is used. The input format can be set to **StreamInputFormat**, and the `stream.recordreader.class.property` is set to `org.apache.hadoop.streaming.StreamXmlRecordReader`. The

data is contained within one large XML wrapper document, which contains a series of elements such as pages and associated metadata.

- **Binary input:** Binary input for MapReduce refers to a data format in which the input data to a MapReduce job is stored and provided in binary form, typically as raw binary files or serialized data objects. Unlike plain text formats like CSV or JSON, binary input is more compact and efficient in terms of storage and transmission, making it advantageous for scenarios where data size and processing speed are critical factors. Binary input is especially useful when dealing with structured or semi-structured data where the overhead of parsing text-based formats can be eliminated. To work with binary input in a MapReduce job, developers often need to implement custom **InputFormats** that can understand and deserialize the binary data into a format suitable for processing within the MapReduce framework. This approach is commonly used in situations where performance optimization and data size reduction are essential considerations, such as in high-throughput data pipelines and big data analytics systems.
- **SequenceFileInputFormat:** Other than text input as format **hadoop** also takes Sequence files as input. It is a variant of **SequenceFileInputFormat** that retrieves sequence file keys and values to text objects. Conversion is performed by calling **toString()** on the keys and values. This format makes sequence files suitable input for streaming.
- **Multiple input:** The input to the map and reduce functions may consist of multiple input files, which are read by the **InputFormat** and processed by the mapper. As data formats evolve, mappers may need to handle legacy formats. Data sources may provide the same type of data, but it could be in different formats. These different representations of data may need to be parsed differently.

For instance, if data is obtained from the UK Weather Department and needs to be combined with data from the **National Climatic Data Center (NCDC)**, the input might look like the following:

```
MultipleInputs.addInputPath(job, ncdcInputPath,
```

```

    TextInputFormat.class,
MaxTemeratureMapper.class);

MultipleInputs.addInputPath(job,
metOfficeInputPath,
    TextInputFormat.class,
MetOfficeMaxTemeratureMapper.class);

```

The data from both weather departments is in text format and is read using the **TextInputFormat**. However, the line format of the two data sources is different, so two different mappers are used. The **MaxOfficeMaxTemperatureMapper** is responsible for processing the NCDC input data and extracting the year and temperature fields from each line. On the other hand, the **MetOfficeMaxTemperatureMapper** processes the met office input data and extracts the year and temperature fields from its lines. This approach allows each mapper to handle its specific data format and extract the necessary information for further processing in the MapReduce job. **MultipleInput** class has an overloaded version of **addInputPath()** that does not take a mapper:

```

public static void addInputPath(Job job, path
path,
    Class<?extends InputForamt>inputForamtClass)

```

This is useful only when one mapper is used and multiple input formats are used.

- **Output type:** During the map function, the input is considered as the key, and the line is treated as the value. The output format needs to be specified in the driver class of the program. The output class can be Text, XML, and so on.
- **TextOutputFormat:** This is the default output format, which writes records as lines of text. The key and value can be of any type, as **TextOutputFormat** converts them to strings by calling **toString()**. Each pair is separated by a tab character, and this separator can be modified using the **mapreduce.output.textoutformat.separator** property.

- **Binary output:** Binary output can be achieved using **SequenceFileOutputFormat** and **SequenceFileAsBinaryOutputFormat**. **SequenceFileOutputFormat** writes sequence files for its output and is the recommended choice of output when it forms input to a MapReduce job because it supports compression, which can be controlled using static methods. On the other hand, **SequenceFileAsBinaryOutputFormat** writes keys and values in raw binary format into a **SequenceFile** container.
- **MapFileOutputFormat:** This format writes map files as output, and it requires the keys in map files to be in order for proper execution. It ensures that the emitted keys are in sorted order.
- **Multiple output:** After MapReduce operations, a set of files is produced as output, named part-r-00000, part-r-00001, and so on. Partition data can produce multiple types of data, which are considered as part of the output. However, this approach to generating data is not recommended as it can lead to unevenly sized data.
- **Lazy output:** A **FileOutputFormat** subclass creates output files like (**part-r-nnnn**) for a file, regardless of whether it is empty or not. In some applications, empty files are not created, and in such cases, **LazyOutputFormat** is used. This acts as a wrapper output format that ensures that the output file is created only when the first record is emitted.

## Conclusion

The MapReduce application chapter has provided valuable insights into the power and versatility of the MapReduce programming model in the context of big data processing and analysis. Throughout this chapter, we have explored various key aspects of MapReduce applications and their significance in solving real-world data challenges.

We began by understanding the fundamental concepts of mapping and reducing data, delving into the architecture of MapReduce applications and how they efficiently process vast datasets in parallel. By showcasing practical examples

and use cases across different industries, we demonstrated the wide-ranging applicability of MapReduce in extracting valuable insights from data.

The chapter also emphasized the importance of performance optimization and fault tolerance in MapReduce applications. We discussed best practices for optimizing data processing, efficient data merging, and handling errors gracefully, ensuring robustness and reliability in the face of failures.

Furthermore, we introduced readers to essential ecosystem tools within the Hadoop framework, such as Apache Pig, Apache Hive, Apache HBase, and Apache Spark, which enhance the capabilities of MapReduce and facilitate more seamless data processing and analysis.

As we explored the process of data merging and sorting during the reduce phase, we highlighted how the merge factor influences the number of intermediate files created, efficiently managing memory and disk usage.

In conclusion, this chapter has equipped readers with a comprehensive understanding of MapReduce and its role in the world of big data. Armed with this knowledge, they are now well-equipped to harness the potential of MapReduce to address complex data challenges, unlock new opportunities, and make informed data-driven decisions.

We encourage readers to continue exploring the vast possibilities within the realm of MapReduce, continuously learning and adopting advanced techniques to unlock the full potential of their data analysis endeavors. As they venture further into the world of big data, may their MapReduce applications pave the way for transformative insights and innovations, shaping the future of data analytics and decision-making in diverse domains.

In the upcoming chapter, we will learn about Hadoop-related tools.

## **Join our book's Discord space**

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



## CHAPTER 7

# Hadoop Related Tools-I: HBase and Cassandra

### Introduction

HBase is a distributed, column-oriented database built on top of the **Hadoop Distributed File System (HDFS)** that is designed to handle very large datasets with real-time read/write random-access requirements. Traditional database solutions are frequently ill-suited for large-scale and distributed environments. Implementing solutions like replication and partitioning add-ons can pose challenges in terms of installation and maintenance, often resulting in trade-offs with core RDBMS features. HBase approaches the scaling problem by designing from the ground up to scale linearly by adding nodes. It does not support SQL or RDBMS features such as joins, complicated queries, triggers, views, and foreign-key constraints, but it is able to host very large, sparsely populated tables on clusters made from commodity hardware. HBase uses the Bigtable data model and architecture to provide a scalable, distributed, and high-performance NoSQL database solution.

HBase is a powerful and scalable distributed NoSQL database that is closely related to the Hadoop ecosystem. It is designed to handle large-scale, semi-structured or structured data with low-latency access patterns. HBase is a column-family-based NoSQL database that provides flexible schema design, making it suitable for storing and managing diverse data types. NoSQL databases allow for dynamic schema changes without the need for downtime or extensive migrations. This is especially useful when dealing with evolving data structures. The feature of NoSQL makes it strong and recommended among all databases, such as scalability, flexible data models, handling big

data, agile development, real-time applications, geospatial and graph data, high availability and fault tolerance, and innovation and modernization.

HBase is an integral part of the Hadoop ecosystem, providing a highly scalable, distributed, and real-time data storage solution. Its flexible data model and strong consistency make it suitable for a wide range of use cases, particularly those requiring low-latency access to large-scale data.

## Structure

In this chapter, we will be covering the following topics:

- Installation of HBase
- Conceptual architecture
- Implementation
- HBase versus RDBMS
- HBase client
- HBase examples and commands
- HBase using Java APIs
- Praxis
- Cassandra
- Cassandra data model
- Cassandra examples
- Cassandra client
- Hadoop integration
- Use cases

## Objectives

This chapter explores key objectives of NoSQL databases: achieving scalability to handle growing data volumes, embracing flexible data modeling for diverse structures, and optimizing storage and retrieval of

unstructured data. Integration aims to support real-time applications, while leveraging NoSQL for data-intensive analytics, particularly in managing intricate relationships within data. These objectives highlight NoSQL's transformative role in modern data management.

## Installation of Hbase

Following is the step-wise procedure for installing and configuring **hbase** on Hadoop:

1. Download the tar file of **hbase** and extract it using the following command:

```
wget  
http://mirror.tcpdiag.net/apache/hbase/hbase- .  
90.6/hbase-0.90.6.tar.gz  
tar -xzf hbase-0.90.6.tar.gz
```

2. Create a folder:

```
mkdir hbaseStorage
```

3. Reach to **hbase** folder using **cd** command:

```
cd hbase-0.90.6/
```

4. Edit the **hbase-site.xml** using:

```
gedit conf/hbase-site.xml
```

Copy and paste the following configuration lines:

```
<?xml version="1.0"?>  
<?xml-stylesheet type="text/xsl"  
href="configuration.xsl"?>  
<configuration>  
  <property>
```

```

<name>hbase.rootdir</name>

<value>file:///home/ubuntu/hbaseStorage/hbase</value>
</property>
<property>

<name>hbase.zookeeper.property.dataDir</name>
<value>/hbaseStorage/zookeeper</value>
</property>
</configuration>
```

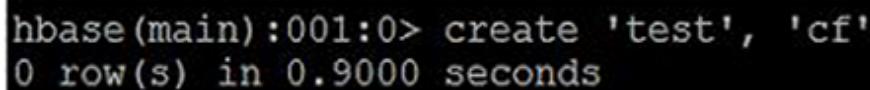
5. Start HBase:

**./bin/start-hbase.sh**

6. Hbase has been installed; to check it, type the following command:

**bin/hbase shell**

It will reflect the **hbase** shell as shown in *Figure 7.1*:



```
hbase(main):001:0> create 'test', 'cf'
0 row(s) in 0.9000 seconds
```

```
hbase(main):002:0> █
```

*Figure 7.1: HBase shell*

## Conceptual architecture

Hbase is involved with versioned data. This data keeps a versioned record of the database in the form of a table with rows and columns. Table row keys are also byte arrays, so theoretically, anything can serve as a row key, from

strings to binary representations of long or even serialized data structures. Table rows are sorted by row key, the table's primary key. The sort is byte-ordered. All table accesses are via the table primary key. All row families are grouped and recognized as column families, and all these can be extended with further rows. New column family members can be added and specified on-demand only. Any row can be represented by column **family:row** name. Column family can remain the same, but row name can be changed accordingly.

In physical view, column families are stored in the file system together. So, when it is fetched from physical storage, it can be fetched as column storage. Hbase table in which the database is stored can be partitioned into regions with a subset of table's row in each region. HDFS and MapReduce are built with different clients, masters and slaves, **namenode** and **datanodes**, and **jobtrackers** and **tasktrackers**.

HBase, a distributed and scalable NoSQL database, is built on a comprehensive architecture that incorporates HDFS and Apache ZooKeeper. HDFS serves as the underlying storage layer, where HBase stores its data files and maintains fault tolerance. ZooKeeper, on the other hand, plays a critical coordination role by managing distributed configuration and synchronization tasks for HBase's distributed environment. HBase uses ZooKeeper to maintain metadata, leader election for **regionservers**, and cluster coordination, ensuring data consistency and availability across the HBase cluster while leveraging HDFS for the storage and retrieval of data, making it a robust and reliable solution for managing large-scale, distributed datasets.

## **Regions and region server**

Tables in HBase are partitioned into different regions that consist of the number of subsets of table's row. Each table needs to belong to any of the regions, and it grows as per the usages of users. The size of regions grows exponentially, but initially, it comprises with single region, but when this size reaches its threshold value, it splits into further regions at row boundary with new same size of regions. Till there is only a split happens, all loading of database about tables hosted in a single server. At the time of the growth of

regions in numbers, it started to be distributed over the HBase cluster. The cluster is formed with a number of regions that can carry a number of tables.

Each **RegionServer** is responsible for managing and serving a subset of data regions (partitions) within HBase tables. These regions are distributed across the cluster, enabling horizontal scalability. **RegionServers** handle read and write requests from clients, efficiently managing data storage by utilizing mechanisms like data compression and block caching. They also maintain an in-memory **MemStore** for rapid write operations. Continuous communication with the Master Server ensures proper cluster coordination, and in the event of a **RegionServer** failure, HBase's fault tolerance mechanisms, such as ZooKeeper, facilitate region reassignment to ensure data availability and uninterrupted service.

## Master Server

The Master Server in HBase serves as the central coordinator and administrator of the HBase cluster. Its primary role is to manage and oversee various administrative tasks, including table creation, schema modifications, and load balancing. The Master Server is responsible for assigning regions (data partitions) to individual **RegionServers**, ensuring even data distribution and optimal cluster performance. While it plays a crucial role in managing the cluster's metadata and administrative functions, it is not involved in the day-to-day serving of data. Importantly, even if the Master Server becomes unavailable, the ongoing data operations and availability of data are not impacted, as **RegionServers** continue to serve data independently.

## Locking

In HBase, locking refers to the process of controlling concurrent access to data by multiple clients or processes. It ensures that only one client can modify a particular piece of data (such as a cell within a table) at a time to maintain data consistency and prevent conflicts. HBase uses locking mechanisms to handle situations where multiple clients are reading or writing to the same data simultaneously, which is essential in distributed and multi-user environments. There are two main types of locks in HBase:

- **Row-level locks:** HBase supports row-level locks, where a client can acquire a lock on an entire row of data. This means that when a client holds a lock on a specific row, no other client can modify that row's data until the lock is released. Row-level locks are used for operations that involve multiple cells within the same row to ensure atomicity.
- **Multi-Version Concurrency Control (MVCC):** HBase also employs MVCC, a mechanism that allows multiple versions of a cell's data to coexist. This means that while one client may be writing a new version of a cell, another client can read the previous version without blocking each other. This approach minimizes contention for read operations.

Locking in HBase is managed internally by the system to provide consistency and isolation for concurrent operations. Clients interact with HBase through its APIs, and the system handles lock acquisition and release to ensure data integrity. Properly managing locks is crucial in HBase to prevent issues like data corruption, lost updates, and race conditions when multiple clients access the same data simultaneously.

## Implementation

In the same manner, as HDFS builds with clusters of **namenode** and **datanode**, HBase also builds with clusters. HBase needs to be modelled with HBase master nodes that consist of a number of **regionservers**. The following are the points that need to be in consideration at the time of implementation:

- The HBase master is responsible for assigning regions to region servers for **regionserver** failure.
- **Regionserver** carries zero or more regions and requests of read/write.

HBase is a powerful and scalable distributed NoSQL database that is closely related to the Hadoop ecosystem. It is designed to handle large-scale, semi-structured or structured data with low-latency access patterns. Some key points to summarize the chapter on HBase in the Hadoop-related tools are as follows:

- **NoSQL database:** HBase is a column-family-based NoSQL database that provides flexible schema design, making it suitable for storing and managing diverse data types.
- **Distributed and scalable:** HBase is distributed in nature, which means it can handle massive data sets by storing data across multiple machines in a Hadoop cluster. This distribution also provides high availability and fault tolerance.
- **Integration with Hadoop:** HBase is tightly integrated with the Hadoop ecosystem, particularly with the HDFS. This allows for seamless data storage and data processing in the Hadoop environment.
- **HBase data model:** HBase follows a wide-column store data model, where data is organized into column families, and each row can have multiple column qualifiers. This design allows for flexible and efficient storage of data.
- **Real-time access:** HBase is optimized for low-latency random reads and writes, making it suitable for real-time applications where quick data access is crucial.
- **Use cases:** HBase is commonly used for various use cases, including real-time data analytics, time-series data storage, event logging, IoT data storage, and more.
- **Strong consistency:** HBase provides strong consistency guarantees through its distributed architecture, ensuring data integrity and reliability.
- **Horizontal scaling:** HBase allows horizontal scaling by adding more nodes to the cluster, making it capable of handling massive amounts of data.
- **Schema design:** Proper schema design is crucial for optimizing HBase performance. Carefully designing row keys, column families, and column qualifiers based on access patterns and use cases is essential.

- **HBase Shell and API:** HBase offers a command-line shell and APIs (Java, REST, and others) for interacting with the database, making it developer-friendly.

The following points reflect the features of HBase:

- HBase host location of root table and address of the master cluster.
- **Regionserver** slave nodes that are listed with **conf/regionservers** track with a list of **datanodes** and **tasktrackers** that can be obtained from the conf/slaves file.
- **conf/hbase-site.xml** describes cluster side configuration.
- **conf/hbase-env.sh** provides environmental set-up for the Hadoop project in order to build a database for it.
- HBase has the property to keep a catalog table named **-ROOT-** and **.META** for maintaining a list of currently used data, state, and location of all regions in the cluster. **-ROOT-** holds a list of **.META** table regions, whereas **.META** table holds the loss of user space regions.
- **ROOT-** cannot be listed in any table directly, so it needs to connect with ZooKeeper to learn about storage.
- All logs are needed to get recorded, and it keeps appended to **memstore** that flows to the filesystem after its storage.
- The commit log is always available through **regionserver**. At the time of its crashing, it commits log by region.
- Accessing the **memstore** is consulted first, and if it is found sufficient, the query completes; otherwise, files run out of flush files.

## **HBase versus RDBMS**

RDBMS is a traditional and known database for all professional and not frequent users, whereas HBase is useful with NoSQL database and requires proper knowledge and a place for its usage. HBase is not a traditional database, it is a column-oriented database which is useful in the distributed

storage of data. It provides reading and writing on a random basis. Following are the differences between the two databases:

- HBase handles large amount of data that are stored distributed manner in a column-oriented format, while RDBMS is systematic storage of a database that cannot support a random manner for accessing database.
- RDBMS strictly follows Codd's 12 rules with fixed schemas and a row-oriented manner of database and also follows ACID properties, whereas HBase follows **BASE** properties and implements complex queries.
- Secondary indexes, complex inner and outer joins, count, sum, sort, group and data of page and table can be easily accessible by RDBMS.
- From small to medium storage applications, there is use of RDBMS that provide solutions with MySQL and PostgreSQL that can increase size with concurrency and performance. Codd's rules always need to be kept in mind while using to extend the size of a database in the use of data processing.
- RDBMS focus on and emphasis on consistency, referential integrity, abstraction from the physical layer and complex queries through SQL language.

At the point of scaling of the database, there are problems faced with RDBMS. They are as follows:

- A proper and well-defined schema should be there when MySQL instances are moved from the local system to a shared manner with distributed storage.
- Too many services of read can be sought from the database, for that **memcached** are added, which expires when no strict ACID is needed.
- Scaling of MySQL is vertical and costly, like having 16 cores, 128 GB of RAM and a hard disk of 15,000 rpm.
- Denormalization of data increases complexities with too many joins.
- Server-side computations are not very compatible because of their slow processing with large data.

- Reading data is fine with RDBMS, but writing large data simultaneously with servers is slow and can cause scaling issues.

In comparison to RDBMS, HBase has the following characteristics:

- In this NoSQL database (HBase), rows are stored sequentially with columns within each row. The performance of table insertion will depend on table size.
- At the time of expanding the size of the database with more table insertion, automatic partitioning will be performed, and tables will split into regions with distribution across all available nodes.
- Regions have properties to share their load and rebalance with their region servers.
- Formation of clusters uses commodity hardware, so it is genuine in a cost manner, whereas RDBMS has more cost involved.
- There is no need to be concerned about a single point of failure because it is distributed in a way that can handle node downtime without any issues.
- With the integration of MapReduce, it allows parallel and distributed jobs against data with locality awareness. This form of batch processing is used for execution.

There is no concern of seriousness about expanding the database with HBase, as it is in the case of RDBMS.

**Relational Database Management Systems (RDBMS)** are sometimes a better choice than HBase in scenarios where data is structured, relationships between data entities are well-defined, and the application's requirements align with the features and strengths of RDBMS. RDBMS excel in use cases that demand complex queries, ACID-compliant transactions, and well-established schemas. These systems are ideal for applications involving financial transactions, traditional business applications, reporting, and scenarios where data consistency and integrity are paramount.

## **HBase client**

HBase is described in Java as its native API. All methods of HBase are defined in **org.apache.hadoop.hbase** package. Following are the methods and constructors that are involved in HBase configuration.

## Class HTable

**HTable** is an internal class of HBase that is used to communicate with a single table of HBase.

This class belongs to **org.apache.hadoop.hbase.client** class.

### Constructors:

- **HTable()**
- **HTable (TableName tableName, ClusterConnection connection, ExecutorService pool)**

Objects can be created to access an HBase table.

### Methods:

- **Void close():** It releases all resources of the **HTable**.
- **Void delete():** It deletes the specified row.
- **Boolean exists (Get get):** It can test the existence of columns in the table.
- **Result get (Get get):** It retrieves some cells from given rows.
- **TableName getName():** It returns table name instances.
- **HTableDescriptor getTableDescriptor():** It returns a description of the table.
- **byte[] getTableName():** It gives a byte representation of the table name.
- **void put (Put put):** Data can be inserted into a table using this method.

## Class Put

It belongs to the `org.apache.hadoop.hbase.client` package and is used to perform the insert operation for a single row.

### Constructor:

- **Put(byte[] row)**: This constructor creates a put operation for a given row.
- **Put(byte[] rowArray, int rowOffset, int rowLength)**: This is used for making a copy of the passed-in row key to keep local.
- **Put(byte[] rowArray, int rowOffset, int rowLength, long ts)**: A copy of the passed-in row key can be constructed with this.
- **Put(byte[] row, long ts)**: Put operation can be performed for a specified row in the given timestamp.

### Methods:

- **Put add(byte[] family, byte[] qualifier, byte[] value)**: It is used to add a given column and value.
- **Put add(byte[] family, byte[] qualifier, long ts, byte[] value)**: It adds a specified value of column with the timestamp.
- **Put add(byte[] family, ByteBuffer qualifier, long ts, ByteBuffer value)**: It adds a specified column and value with a specified timestamp.
- **Put add(byte[] family, ByteBuffer qualifier, long ts, ByteBuffer value)**: It also adds a specified column and value with a given timestamp.

## Class Get

This is used to perform read operations on a single row. This class belongs to the `org.apache.hadoop.hbase.client` package.

### Constructor:

- **Get(byte[] row)**: Get operation can be created for a specified row.
- **Get(Get get)**

#### Methods:

- **Get addColumn(byte[] family, byte[] qualifier)**: It is used to retrieve a column from a specific column-family.
- **Get addFamily(byte[] family)**: It is used to retrieve all columns from the specified column family.

## Class Delete

This is used to perform the delete operation on a single row. This class belongs to the **org.apache.hadoop.hbase.client** package.

#### Constructors:

- **Delete(byte[] row)**: It creates a delete operation for the given row.
- **Delete(byte[] rowArray, int rowOffset, int rowLength)**: It also creates a delete operation for a specified row and timestamp.
- **Delete(byte[] rowArray, int rowOffset, int rowLength, long ts)**: It creates a delete operation for a specified row with the timestamp.
- **Delete(byte[] row, long timestamp)**: It creates a delete operation for a given row in a specified timestamp.

#### Methods:

- **Delete addColumn(byte[] family, byte[] qualifier)**: It deletes the given column of the latest version in the database.
- **Delete addColumns(byte[] family, byte[] qualifier, long timestamp)**: It deletes the given columns from all versions with less timestamp than required.
- **Delete addFamily(byte[] family)**: It deletes all versions of all columns of the specified family.

- **Delete addFamily(byte[] family, long timestamp):** It deletes all columns of the given family with less timestamp.

## Class Result

This class is used to get a single-row result of a Get or a Scan query. This class is not thread-safe. It is designed to represent the result of a single read operation on a specific row from an HBase table.

### Constructors:

- **Result():** This constructor is used to get the latest version of a given column.

### Methods:

- **byte[] getValue(byte[] family, byte[] qualifier):** This is used to get the latest version of a specified column.
- **byte[] getRow():** This is used to retrieve the row key that communicates to the row from which the result was created.

## HBase examples and commands

Following are the commands that are frequently used in HBase:

### Inserting data using HBase shell:

```
put '<table  
name>', 'row1', '<colfamily:colname>', '<value>'
```

### Example:

```
hbase(main):004:0> put 'stu', '1', 'personal  
data:name', 'nitin'
```

```
0 row(s) in 0.3600 seconds
```

```
hbase(main):005:0> put 'stu', '1', 'personal  
data:city', 'allahabad'
```

```
0 row(s) in 0.0410 seconds  
hbase(main):006:0> put 'emp','1','professional  
data:designation','scholar'  
0 row(s) in 0.0240 seconds
```

Using this, many rows can be inserted using the **put** command for completion of the whole table. Using the **scan** command gets reflected as follows:

```
hbase(main):022:0> scan 'stu'
```

ROW	COLUMN+CELL
1	column=personal data:city, timestamp=1417524216501, value=allahabad
1	column=personal data:name, timestamp=1417524185058, value=nitin
1	column=professional data:designation, timestamp=1417524232601, value=scholar
1	column=professional data:salary, timestamp=1417524244109, value=50000
2	column=personal data:city, timestamp=1417524574905, value=banglore
2	column=personal data:name, timestamp=1417524556125, value=alok
2	column=professional data:designation, timestamp=1417524592204, value=soft.engg
2	column=professional data:salary, timestamp=1417524604221, value=30000

```
3 column=personal data:city,  
timestamp=1417524681780, value=delhi  
  
3 column=personal data:name,  
timestamp=1417524672067, value=sumit  
  
3 column=professional data:designation,  
timestamp=1417524693187, value=prof.  
  
3 column=professional data:salary,  
timestamp=1417524702514, value=25000
```

### **Updating data using HBase shell:**

```
put 'table name', 'row ', 'Column family:column  
name', 'new value'
```

### **Example:**

```
hbase(main):003:0> scan 'emp'
```

ROW	COLUMN + CELL
row1	column = personal:name, timestamp = 1418051555, value = nitin
row1	column = personal:city, timestamp = 1418275907, value = allahabad
row1	column = professional:designation, timestamp = 14180555, value = scholar

If there is need to change city name Allahabad to delhi then.....

```
hbase(main):002:0> put  
'stu','row1','personal:city','Delhi'  
0 row(s) in 0.0400 seconds
```

Now if scan whole table

```
hbase(main):003:0> scan 'stu'
```

ROW	COLUMN + CELL
-----	---------------

row1	column = personal:name, timestamp = 1418035791555, value = nitin
------	--

row1	column = personal:city, timestamp = 1418274645907, value = Delhi
------	--

row1	column = professional:designation, timestamp = 141857555, value = manager
------	---

row1	column = professional:salary, timestamp = 1418039555, value = 50000
------	---

1 row(s) in 0.0100 seconds

### **Reading data using HBase shell:**

```
get '<table name>', 'row1'
```

### **Example:**

```
hbase(main):012:0> get 'stu', '1'
```

COLUMN	CELL
--------	------

personal : city	timestamp = 1417521848375, value = delhi
-----------------	--

personal : name	timestamp = 1417521785385, value = nitin
-----------------	--

```
professional: designation timestamp =
1417521885277, value = scholar
```

3 row(s) in 0.0270 seconds

**Reading a given column:**

```
hbase> get 'table name', 'rowid', {COLUMN => 'column
family:column name'}
```

**Example:**

```
hbase(main):015:0> get 'stu', 'row1', {COLUMN =>
'personal:name'}
```

COLUMN	CELL
personal:name	timestamp = 1418035791555, value = suraj

1 row(s) in 0.0080 seconds

**Delete specific cell in a table:**

```
delete '<table name>', '<row>', '<column name >',
'<time stamp>'
```

**Example:**

```
hbase(main):006:0> delete 'stu', '1', 'personal
data:city',
```

1417521848375

0 row(s) in 0.0060 seconds

**Delete all cells in a table:**

```
deleteall '<table name>', '<row>',
```

**Example:**

```
hbase(main):007:0> deleteall 'stu','1'  
0 row(s) in 0.0240 seconds  
hbase(main):022:0> scan 'stu'
```

ROW

COLUMN + CELL

```
2 column = personal data:city, timestamp =  
1417524574905, value = banglore  
2 column = personal data:name, timestamp =  
1417524556125, value = alok  
2 column = professional data:designation, timestamp  
= 1417524204, value = soft.engg  
3 column = personal data:city, timestamp =  
1417524681780, value = delhi  
3 column = personal data:name, timestamp =  
1417524672067, value = sumit  
3 column = professional data:designation, timestamp  
= 1417523187, value = prof.
```

### **Scanning using HBase shell:**

```
scan '<table name>'
```

### **Example:**

```
hbase(main):010:0> scan 'stu'
```

ROW

COLUMN + CELL

```
1 column = personal data:city, timestamp =
1417521848375, value = Allahabad
1 column = personal data:name, timestamp =
1417521785385, value = nitin
1 column = professional data:designation, timestamp
= 1417585277, value = scholar
1 row(s) in 0.0370 seconds
```

### **Count:**

```
count '<table name>'
```

### **Example:**

```
hbase(main):023:0> count 'stu'
2 row(s) in 0.090 seconds
⇒ 2
```

### **Truncate:**

```
hbase> truncate 'table name'
```

### **Example:**

```
hbase(main):011:0> truncate 'stu'
Truncating 'one' table (it may take a while):
  - Disabling table...
  - Truncating table...
0 row(s) in 1.5950 seconds
```

After truncating the table when the table is scanned, it will be as follows:

```
hbase(main):017:0> scan 'stu'
```

ROW COLUMN + CELL

0 row(s) in 0.3110 seconds

**Disable table:**

hbase>disable 'tablename'

**Example:**

hbase(main):012:0>disable 'stu'

**Verification:**

hbase>is\_disable 'tablename'

**Example:**

hbase(main):012:0>is\_disable 'stu'

hbase(main):013:0>scan 'stu'

**Alter table:**

hbase> alter 'tablename', name => 'column-family',  
versions => 5

**Example:**

base(main):007:0> alter 'stu', name => 'personal  
data', versions => 5

Updating all regions with the new schema...

0/1 regions updated.

1/1 regions updated.

Done.

0 row(s) in 2.1060 seconds

**Scope operator for alter table:**

```
hbase>alter 't1', READONLY(option)
```

**Example:**

```
hbase(main):006:0> alter 'stu', READONLY  
Updating all regions with the new schema...
```

0/1 regions updated.

1/1 regions updated.

Done.

0 row(s) in 2.2140 seconds

**Deleting column family:**

```
hbase> alter ' table name ', 'delete' => ' column  
family '
```

**Example:**

```
hbase(main):007:0> alter  
'stu', 'delete'=>'professional'
```

Updating all regions with the new schema...

0/1 regions updated.

1/1 regions updated.

Done.

0 row(s) in 2.2380 seconds

```
hbase(main):003:0> scan 'employee'
```

ROW	COLUMN + CELL
-----	---------------

row1	column = personal:city, timestamp = 14181936767, value = allahabd
------	---

```
row1 column = personal:name, timestamp =  
1418193806767, value = nitin
```

```
1 row(s) in 0.0830 seconds
```

### **Existence of table:**

```
hbase> exists 'stu'
```

### **Example:**

```
hbase(main):024:0> exists 'emp'
```

```
Table emp does exist
```

```
0 row(s) in 0.0750 seconds
```

```
=====
```

```
hbase(main):015:0> exists 'student'
```

```
Table student does not exist
```

```
0 row(s) in 0.0480 seconds
```

### **Dropping a table:**

```
hbase> drop 'stu'
```

### **Example:**

```
hbase(main):018:0> disable 'emp'
```

```
0 row(s) in 1.4580 seconds
```

```
hbase(main):019:0> drop 'emp'
```

```
0 row(s) in 0.3060 seconds
```

### **Drop all table:**

```
hbase> drop_all 't.*'
```

**Example:**

```
hbase(main):002:0> disable_all 'raj.*'
```

```
raja
```

```
rajani
```

```
rajendra
```

```
rajesh
```

```
raju
```

```
Disable the above 5 tables (y/n)?
```

```
y
```

```
5 tables successfully disabled
```

```
hbase(main):018:0> drop_all 'raj.*'
```

```
raja
```

```
rajani
```

```
rajendra
```

```
rajesh
```

```
raju
```

```
Drop the above 5 tables (y/n)?
```

```
y
```

```
5 tables successfully dropped
```

## HBase using Java APIs

Java supports HBase, and it easily executes all its commands using Java programs.

## Creating a table

The table can be created in HBase using the `createTable()` method that is in the `HBaseAdmin` class, and this class belongs to `org.apache.hadoop.hbase.client` package. Follow the following steps:

1. **Instantiate `HBaseAdmin`:** This is to configure the object as a parameter with the configuration class as follows:

```
Configuration conf =  
HBaseConfiguration.create();  
HBaseAdmin admin = new HBaseAdmin(conf);
```

2. **Create `TableDescriptor`:** The descriptor class belong to `org.apache.hadoop.hbase` class.

```
//creating table descriptor  
HTableDescriptor table = new  
HTableDescriptor(toBytes("Table name"));  
  
//creating column family descriptor  
HColumnDescriptor family = new  
HColumnDescriptor(toBytes("column family"));  
  
//adding column family to HTable  
table.addFamily(family);
```

3. **Execute through Admin:** The table can be executed through admin by the method of the HBase admin class by `admin.createTable(table);`

This is a Java program that demonstrates how to create an HBase table using the `HBaseAdmin` class. Here is a brief explanation of the code:

First, we import the required HBase and Hadoop classes. In the `main()` method, we instantiate a configuration object and an `HBaseAdmin` object. We then create an `HTableDescriptor` object, which represents the table's schema. We add two column families to the table descriptor, labelled as personal and professional. Finally, we create the table using the `admin.createTable()` method and print a message to confirm that the table was created.

Note that this code uses the deprecated `HBaseAdmin` class. The code is as follows:

```
import java.io.IOException;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HColumnDescriptor;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.Admin;
import org.apache.hadoop.hbase.client.Connection;
import
org.apache.hadoop.hbase.client.ConnectionFactory;
public class CreateTable {

    public static void main(String[] args) throws
IOException {

    // Instantiating configuration class
    org.apache.hadoop.conf.Configuration config =
HBaseConfiguration.create();
```

```
// Instantiating Connection class  
  
Connection connection =  
ConnectionFactory.createConnection(config);  
  
  
// Instantiating Admin class  
  
Admin admin = connection.getAdmin();  
  
  
// Instantiating table descriptor class  
  
HTableDescriptor tableDescriptor = new  
HTableDescriptor(TableName.valueOf("emp"));  
  
  
// Adding column families to table descriptor  
  
tableDescriptor.addFamily(new  
HColumnDescriptor("personal"));  
  
tableDescriptor.addFamily(new  
HColumnDescriptor("professional"));  
  
  
// Execute the table through admin  
  
admin.createTable(tableDescriptor);  
System.out.println("Table created");  
  
  
// Closing connection and admin objects
```

```
    admin.close();

    connection.close();

}

}
```

It can be compiled in the traditional way, and the table will be created.

## List of the tables in HBase

All registered tables can be shown using the command list from **hbase** shell. Follow the following steps:

1. **List table:** `listTables()` method in class **HBaseAdmin** can get all list of table name in **HBase**.

```
// Creating a configuration object
Configuration conf =
HBaseConfiguration.create();

// Creating HBaseAdmin object
try (HBaseAdmin admin = new HBaseAdmin(conf)) {

    // Getting all the list of tables using
    HBaseAdmin object

    HTableDescriptor[] tableDescriptor =
admin.listTables();

    // Printing the list of tables
    for (HTableDescriptor descriptor :
tableDescriptor) {
```

```

        System.out.println(descriptor.getNameAsString())
    ;
}

} catch (IOException e) {
    e.printStackTrace();
}

```

Note that in the original code, the **HBaseAdmin** object is not being closed properly after use. It was enclosed within a **try-with-resources** block to ensure its closure once the list of tables was acquired. Furthermore, a **try-catch** block was introduced to manage any potential **IOException** that might occur during the creation of the **HBaseAdmin** object.

2. **Table descriptor:** **HTableDescriptor[]** is the array using the length variable of the **HTableDescriptor** class:

```

import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import
org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.client.Admin;
import
org.apache.hadoop.hbase.client.Connection;
import
org.apache.hadoop.hbase.client.ConnectionFactory
;
import org.apache.hadoop.hbase.TableName;

```

```
public class ListTables {  
  
    public static void main(String[] args) throws  
IOException {  
  
        // Instantiating a configuration class  
        Configuration conf =  
HBaseConfiguration.create();  
  
        // Instantiating Connection class  
        Connection connection =  
ConnectionFactory.createConnection(conf);  
  
        // Instantiating Admin class  
        Admin admin = connection.getAdmin();  
  
        // Getting all the list of tables using  
Admin object  
        HTableDescriptor[] tableDescriptor =  
admin.listTables();  
  
        // printing all the table names.  
        for (int i = 0; i < tableDescriptor.length;  
i++ ){  
  
System.out.println(tableDescriptor[i].getTableName)
```

```

        me().getNameAsString());
    }

    // closing the connection
    admin.close();
    connection.close();
}

}

```

## Disable a table

**TableDisabled()** method is used to verify method aliveness, and to disable a table, **disableTable()** method is used. These methods belong to the **HBaseAdmin** class:

### 1. Instantiate HBaseAdmin:

```

// Creating configuration object
Configuration conf =
HBaseConfiguration.create();

// Creating HBaseAdmin object
HBaseAdmin admin = new HBaseAdmin(conf);

```

### 2. Create TableDescriptor:

```
Boolean b = admin.isTableDisabled("emp");
```

### 3. Program to disable table:

```
import java.io.IOException;
```

```
import org.apache.hadoop.conf.Configuration;

import
org.apache.hadoop.hbase.HBaseConfiguration;
import
org.apache.hadoop.hbase.MasterNotRunningException;
import
org.apache.hadoop.hbase.client.HBaseAdmin;

public class DisableTable{

    public static void main(String args[]) throws
MasterNotRunningException, IOException{

        // Instantiating configuration class
        Configuration conf =
HBaseConfiguration.create();

        // Instantiating HBaseAdmin class
        HBaseAdmin admin = new HBaseAdmin(conf);

        // Verifying weather the table is disabled
        Boolean bool = admin.isTableDisabled("emp");
        System.out.println(bool);
    }
}
```

```

        // Disabling the table using HBaseAdmin
        object

        if(!bool){
            admin.disableTable("emp");
            System.out.println("Table disabled");
        }
    }
}

```

Check and compile it in the traditional way.

## Add column family

**addColumn()** of the **HBaseAdmin** class is used for adding columns. The steps are as follows:

### 1. Instantiate **HBaseAdmin**:

```

// Instantiating configuration object
Configuration conf =
HBaseConfiguration.create();

// Instantiating HBaseAdmin class
HBaseAdmin admin = new HBaseAdmin(conf);

```

### 2. Create **TableDescriptor**:

```

// Instantiating columnDescriptor object

HColumnDescriptor columnDescriptor = new
HColumnDescriptor("contactDetails");

```

### 3. Program to add column family:

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HColumnDescriptor;
import org.apache.hadoop.hbase.MasterNotRunningException;
import org.apache.hadoop.hbase.client.HBaseAdmin;

public class AddColoumn{

    public static void main(String args[]) throws
MasterNotRunningException, IOException{

        // Instantiating configuration class.
        Configuration conf =
HBaseConfiguration.create();

        // Instantiating HBaseAdmin class.
        HBaseAdmin admin = new HBaseAdmin(conf);
```

```

    // Instantiating columnDescriptor class
    HColumnDescriptor columnDescriptor = new
HColumnDescriptor("contactDetails");

    // Adding column family
    admin.addColumn("employee",
columnDescriptor);
    System.out.println("column added");
}

}

```

Compile it and run it for creating column family under the **hbase** database.

## **Deleting column family**

**deleteColumn()** of the **HBaseAdmin** class is used to delete a column family. The steps for it are as follows:

### **1. Instantiate HBaseAdmin:**

```

// Instantiating configuration object
Configuration conf =
HBaseConfiguration.create();

```

```

// Instantiating HBaseAdmin class
HBaseAdmin admin = new HBaseAdmin(conf);

```

### **2. Create TableDescriptor:**

```
// Deleting column family
```

```
admin.deleteColumn("employee",
"contactDetails");
```

### 3. Program to delete column family:

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.MasterNotRunningException;
import org.apache.hadoop.hbase.client.HBaseAdmin;

public class DeleteColoumn{

    public static void main(String args[]) throws
MasterNotRunningException, IOException{

        // Instantiating configuration class.
        Configuration conf =
HBaseConfiguration.create();

        // Instantiating HBaseAdmin class.
        HBaseAdmin admin = new HBaseAdmin(conf);
```

```

    // Deleting a column family

    admin.deleteColumn("employee", "contactDetails");
    System.out.println("column deleted");
}

}

```

Compile and check about the deletion of the column family.

## Verifying the existence of the table

The **tableExists()** method of the **HBaseAdmin** class is used to check the verification of the existence of a table in the database. The steps for it are as follows:

### 1. Instantiate **HBaseAdmin**:

Instantiate the **HBaseAdmin** class

```

// Instantiating configuration object
Configuration conf =
HBaseConfiguration.create();

// Instantiating HBaseAdmin class
HBaseAdmin admin = new HBaseAdmin(conf);

```

### 2. Program for verification:

```
import java.io.IOException;
```

```
import  
org.apache.hadoop.hbase.HBaseConfiguration;  
import org.apache.hadoop.conf.Configuration;  
import  
org.apache.hadoop.hbase.client.HBaseAdmin;  
  
public class TableExists{  
  
    public static void main(String args[])throws  
IOException{  
  
        // Instantiating configuration class  
        Configuration conf =  
HBaseConfiguration.create();  
  
        // Instantiating HBaseAdmin class  
        HBaseAdmin admin = new HBaseAdmin(conf);  
  
        // Verifying the existence of the table  
        boolean bool = admin.tableExists("emp");  
        System.out.println( bool);  
    }  
}
```

Compile it, and this will show the existence of a table in the database.

## **Deleting table**

The **deleteTable()** method in the **HBaseAdmin** class represents the deletion of the table from a database. The steps for it are as follows:

**1. Instantiate HBaseAdmin:**

```
// creating a configuration object  
Configuration conf =  
HBaseConfiguration.create();  
  
// Creating HBaseAdmin object  
HBaseAdmin admin = new HBaseAdmin(conf);
```

**2. Disable the table:**

```
admin.disableTable("stu");
```

**3. Program for deletion:**

```
import java.io.IOException;  
  
import  
org.apache.hadoop.hbase.HBaseConfiguration;  
import org.apache.hadoop.conf.Configuration;  
import  
org.apache.hadoop.hbase.client.HBaseAdmin;  
  
public class DeleteTable {  
  
    public static void main(String[] args) throws  
IOException {
```

```

    // Instantiating configuration class
    Configuration conf =
HBaseConfiguration.create();

    // Instantiating HBaseAdmin class
    HBaseAdmin admin = new HBaseAdmin(conf);

    // disabling table named emp
    admin.disableTable("stu");

    // Deleting emp
    admin.deleteTable("stu");
    System.out.println("Table deleted");
}

}

```

Compilation and execution will lead to the deletion of the table from the database.

## Disabling table

Disabling a table in HBase serves several important purposes. First, it allows you to temporarily halt any read or write operations on the table, which can be useful when performing maintenance or making significant schema changes. Additionally, disabling a table ensures data consistency by preventing concurrent modifications, which can be crucial for certain administrative tasks. Moreover, it safeguards against accidental data corruption during schema alterations or other operations that may impact the table's structure. Overall, disabling a table in **HBase** is a necessary step for

maintaining data integrity, executing administrative tasks, and ensuring the stability of the **HBase** cluster:

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.Admin;
import org.apache.hadoop.hbase.client.Connection;
import
org.apache.hadoop.hbase.client.ConnectionFactory;

public class DisableTableExample {
    public static void main(String[] args) throws
Exception {
        // Create an HBase configuration
        Configuration config =
HBaseConfiguration.create();

        // Create an HBase connection
        try (Connection connection =
ConnectionFactory.createConnection(config)) {
            // Get an instance of the HBase Admin
            try (Admin admin = connection.getAdmin()) {
                // Specify the table name
```

```

        TableName tableName =
TableName.valueOf("your_table_name");

        // Disable the table
admin.disableTable(tableName);

        System.out.println("Table " + tableName + " "
is disabled.");
    }

}
}
}

```

## Stopping HBase

**HBaseAdmin** class's **shutdown()** method is used for stopping **HBase**. The steps for the same are as follows:

### 1. Instantiate **HBaseAdmin**:

```

// Instantiating configuration object
Configuration conf =
HBaseConfiguration.create();

// Instantiating HBaseAdmin object
HBaseAdmin admin = new HBaseAdmin(conf);

```

### 2. Program for shutdown:

```
import java.io.IOException;
```

```
import  
org.apache.hadoop.hbase.HBaseConfiguration;  
import org.apache.hadoop.conf.Configuration;  
import  
org.apache.hadoop.hbase.client.HBaseAdmin;  
  
public class ShutDownHbase{  
  
    public static void main(String args[])throws  
IOException {  
  
        // Instantiating configuration class  
        Configuration conf =  
HBaseConfiguration.create();  
  
        // Instantiating HBaseAdmin class  
        HBaseAdmin admin = new HBaseAdmin(conf);  
  
        // Shutting down HBase  
        System.out.println("Shutting down hbase");  
        admin.shutdown();  
    }  
}
```

The program will shut down HBase.

## Challenges

The following are the common issues involved in HBase:

- **Versions:** Versions of HBase and Hadoop should be compatible with each other in order to run smoothly. Incompatible versions will show an error or exception about version mismatch. For example, HBase 0.20.2 would run on Hadoop 0.20.2, but HBase 0.19.5 would not run on Hadoop 0.20.0.

From HBase 0.90, version relationship has no longer affected Hadoop working because Hadoop releases cycles of version not aligned with HBase development. So, HBase 0.90.x will work with Hadoop 0.20.x and 0.21.x.

- **Security and access control:** Configuring security features like authentication and authorization can be complex, and misconfigurations can lead to security vulnerabilities. To mitigate this risk, it is important to carefully configure security features and to keep them up-to-date with the latest security patches.
- **Data compaction:** HBase stores data in **HFiles**, and over time, these files may become fragmented and inefficient to read. To improve performance, it is necessary to run regular compaction jobs to defragment and optimize **HFiles**.
- **Cluster management and configuration:** Setting up and configuring an HBase cluster can be challenging, especially for users new to the Hadoop ecosystem. To avoid performance issues, data loss, or instability, it is important to carefully configure an HBase cluster and to keep it up-to-date with the latest patches.
- **HDFS:** Use of HDFS in MapReduce and HBase, both are in different scenarios. HDFS files are opened with map task and then closed after use of it, whereas in HBase, data files are opened on starting and kept opened to avoid paying file open cost on each access.
  - Because there is a need to keep files open in HBase on a loaded cluster, it does not take long before running into the system, and Hadoop imposed limits.

- If there are three nodes and all of them are running an instance of a **datanode** and a **regionserver**, it is transferred into a table that is presently at 100 regions and 10 column families, which enables every column family to have, on average, two flush files. There is no need to have  $100 \times 10 \times 2$ , or 2,000, files open at any one time.
- In addition to these total miscellaneous alternative descriptors consumed by outstanding scanners and Java libraries, every open file consumes a minimum of one descriptor over on the remote **datanode**.
- The default limit on the amount of file descriptors per method is 1,024.
- In case of exceeding the limit, there is a complaint regarding too many open files in logs.
- The fix requires increasing the file descriptor **ulimit** count that can be verified from the HBase method that is running with sufficient file descriptors by watching the first few lines of a **regionserver** object's log. It emits vitals like the JVM being used and atmosphere settings such as the file descriptor **ulimit**.

In case of running out of **datanode** threads following points need to be considered:

The Hadoop **datanode** has a bound of 256 threads. It will run at any one time within the **datanode**; as of this writing, every open affiliation to a file block consumes a thread:

- The datanode log, see a complaint like **xceiverCount** 258 exceeds the limit of coinciding **xcievers** 256 but, again, likely see **HBase** act unpredictably before encountering this log entry.
- Increase the **dfs.datanode.max.xcievers** count in HDFS and restart the cluster.
- **Schema design:** In schema design for HBase, consideration of data modeling and structure is essential to optimize performance and meet specific use case requirements. Schema design in HBase involves

planning and organizing how your data will be stored in the tables. HBase is a NoSQL database that uses a wide-column store model, which allows for flexible schema design.

- HBase tables are similar to those in an RDBMS, except that the cells are versioned, rows are sorted, and columns may be added on the fly by the client as long as the column family they belong to preexists.
- These factors should be considered when coming up with schemas for HBase. However, the most necessary concern in coming up with schemas is the consideration of how the information is going to be accessed.
- All-access is via the primary key, so the key design should lend itself to however the information goes to be queried.
- The opposite property to keep in mind once designing schemas is that a defining attribute of column-oriented stores, like HBase, is that it will host wide and sparsely populated tables at no incurred cost.
- The compound row key features a station prefix that serves to cluster temperatures by the station.
- The reversed timestamp suffix creates it; therefore, temperatures might be scanned in order from most recent to oldest.
- A smart compound key can cluster data in ways amenable to how it will be accessed.
- To design compound keys, row keys need to be sorted properly. Otherwise, it runs into the problem wherever 10 sorts before 2 when only byte order is considered.
- If your keys are integers, use a binary representation rather than persist the string version of a number, as it consumes less space.

## Cassandra

Cassandra is also a distributed, decentralized, scalable, highly available, and open-source NoSQL database. Its design is based on Amazon's Dynamo, and the data model is based on Google's Bigtable. A row of data could contain any number of columns, each with a name and a value. A row could have a thousand different columns, whereas another row could have a thousand other columns. Rows do not have to have the same columns in a schema-less database, as there is no schema that each row must adhere to.

**Note:** There is an interesting fact behind the name Cassandra. In Greek mythology, Cassandra was the daughter of King Priam and Queen Hecuba of Troy. Cassandra was so beautiful that the god Apollo gave her the ability to see the future. But when she refused his amorous advances, he cursed her such that she would still be able to accurately predict everything that would happen—but no one would believe her. Cassandra foresaw the destruction of her city of Troy but was powerless to stop it. The Cassandra distributed database is named for her. We can speculate that it is also named as a kind of a joke on the Oracle at Delphi, another seer for whom a database is named.

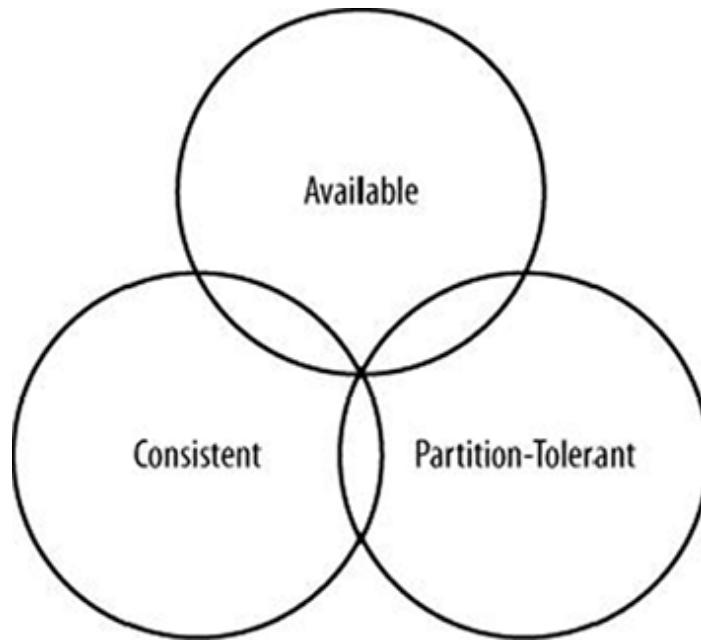
The following are the points that need to be considered for the Cassandra database:

- This NoSQL database created at Facebook can run on multiple machines as justification for its distributed property.
- Its design and code work with optimized performance, with multiple data centers racked across all centers geographically.
- Unlike others (MySQL, Bigtable), Cassandra is decentralized, meaning it is identical.
- Cassandra performs peer-to-peer protocol and maintains all nodes to keep synchronous with all other nodes.
- There is no single point of failure as it is decentralized means all clusters function exactly at the same time. This is also called server symmetry.
- Unlike all other distributed modes, multiple copies are stored in multiple machines that are replica that needs to get updated from time to time once there are changes required because this is not decentralized work of clusters.

- As Cassandra functions with a master/slave relationship, the master acts as an authoritative source of the database while the slave functions to synchronize their copies of data.
- A decentralized system is better than a master/slave system because, in that case, all nodes are equal, and there is no configuration required to support it.
- In master/slave, there is also a single point of failure, whereas this kind of problem is not in a decentralized approach.
- Because of a decentralized and distributed approach, Cassandra has failure single point of failure rate is zero with high availability.

## CAP theorem

Cassandra works on the CAP theorem that was given by *Eric Brewer*. Sometimes, it is also called Brewer's theorem. The name was given in the year 2000 at the ACM Symposium on the Principles of Distributed Computing. This theorem states three requirements: consistency, availability and partition tolerance. The CAP theorem was formally proved by *Seth Gilbert* and *Nancy Lynch* of MIT in 2002. This theorem leads to partition tolerance with availability and consistency options. At the time of tolerance, there is the need to make a choice between availability and consistency, as shown in [Figure 7.2](#):



**Figure 7.2: CAP Theorem**

The following are the points that must be taken into consideration while stating about CAP theorem:

- **Consistency (C)** represents all database clients that read the same value for the same query with concurrent updates.
- **Availability (A)** states all databases in which the client will always be able to read and write data.
- **Partition tolerance (P)** can split databases into multiple machines that can function in the form of network segmentation.

Following are the points that are considered by Stu Hood and are represented in [\*Figure 7.3\*](#):

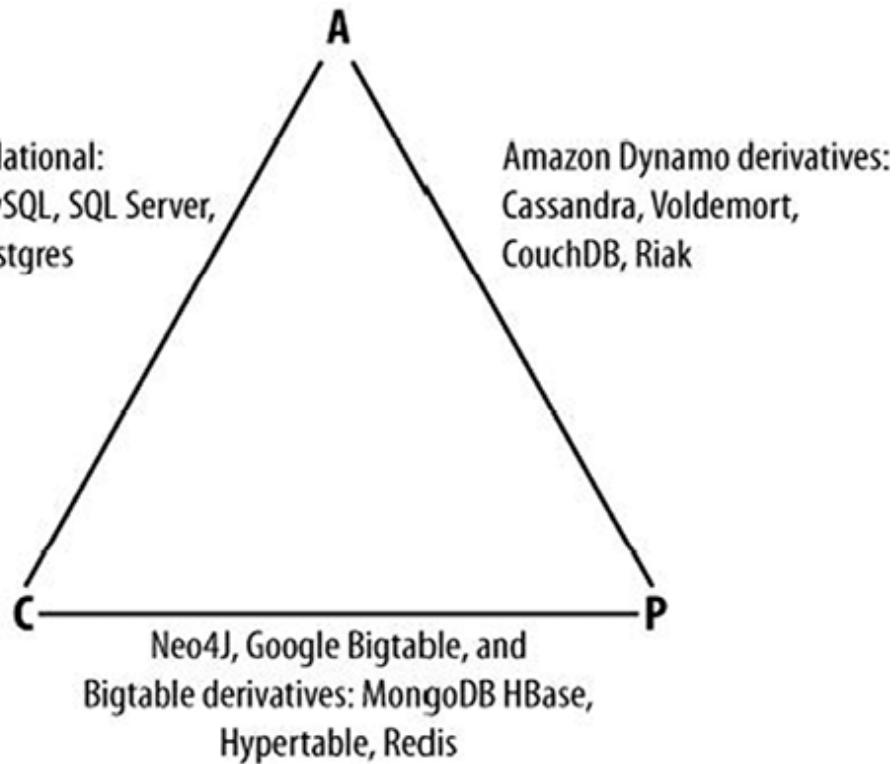
- A distributed MySQL database count is consistent only with Google's synchronous replication patches; otherwise, it is only available and partition-tolerant.
- CAP is independent of the orientation of the data storage mechanism with the support of a graph and document-oriented database.
- At the point of failure of any node with network error, consistency and availability can fail in a relational database because of a single master configuration.

- Graph databases like Neo4J are less focused on availability and more on consistency and partition tolerance.
- Databases like Amazon's Dynamo, which also includes Cassandra, CouchDB, Project, and so on, are focused on availability and partition tolerance.

## **Explanation in terms of intersection points**

Following is the explanation in terms of intersection points:

- **CA:** The area of CA is focused on supporting consistency and availability, that is, using two-phase commit for distributed transactions that refer at the time of network partition system will be blocked, and if the application requires level of scaling, it will be easy to manage.
- **CP:** For providing consistency and partitioning tolerance, sharding can be tried to scale data. Data will be consistent, but there is a risk of data unavailability or node failure.
- **AP:** To support availability and partition tolerance, systems can return inaccurate data. Network partitioning also does not affect availability. DNS can be picked for providing as its example.



*Figure 7.3: Database appearance*

## Characteristics of Cassandra

Cassandra is a distributed and NoSQL database which has the following characteristics:

- **Column-oriented:** Cassandra is a column-oriented NoSQL database and it represents data structure in a sparse multidimensional hash table, making data accessible. Every row has a unique key for accessing data uniquely.

Cassandra stores data in a multidimensional hash table, which means there is no need to mention anything about data structure. It stores data in a row-oriented manner with a unique key and performs row-oriented storage. If it is about columnar, then it is not wrong.

- **High performance:** Cassandra is useful due to its advantage of multiprocessor and multi-core machines. It can easily scale with multiple machine support with thousands of terabytes. Even with heavy traffic of data load, Cassandra performs well with availability,

scalability and tolerance. Adding more servers can increase its performance.

## Installing Cassandra

Cassandra plays an important role in database format. Following are the steps to install Cassandra:

1. **Download and extract:** From <http://cassandra.apache.org/download/> Cassandra's latest version can be downloaded with the name as **apache-cassandra-x.x.x-bin.tar.gz**, where **x.x.x** represents the version number. The download is around 10MB. After downloading, there is a need to extract it. In extraction, there will be bin, conf, interface, javadoc, and lib directories. These directories contain all the required external libraries that are needed at the time of running Cassandra. These directories also include Thrift and Avro RPC libraries for interaction purposes.
2. **Configure Cassandra:** From **bin** folder, there is **cassandra.yaml**: file that needs to be edited using:

```
$ gedit cassandra.yaml
```

The preceding command will open the said file. After that, there is a need for its configuration. By default, these values will be set to the specified directories:

```
data_file_directories "/var/lib/cassandra/data"  
commitlog_directory "/var/lib/cassandra/commitlog"  
saved_caches_directory  
"/var/lib/cassandra/saved_caches"
```

- i. **Create directories:** For writing data for Cassandra, you need to create directories in **/var/lib/Cassandra** and **/var./log/Cassandra**:

```
[root@mayank]# mkdir /var/lib/cassandra
```

```
[root@mayank]# mkdir /var/log/cassandra
```

- ii. Give permission to folders: You need to define permission to folders for Cassandra:

```
[root@mayank]# chmod 777 /var/lib/cassandra
```

```
[root@mayank]# chmod 777 /var/log/cassandra
```

3. **Start Cassandra:** After reaching the **home** folder of Cassandra, there is a need to start it using Cassandra as a database:

```
$ cd $CASSANDRA_HOME
```

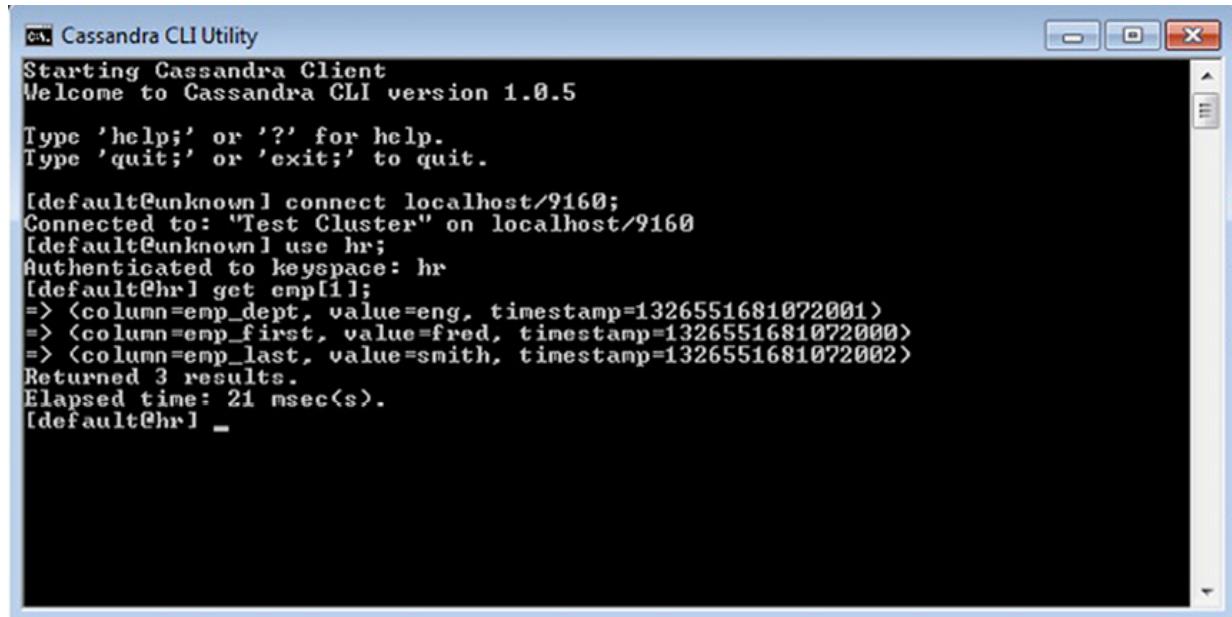
```
$./bin/cassandra -f
```

**-f** is optional as its function is to keep the process running in the foreground instead of running in the background process. From a programming point of view, there is a need to set up the following JAR files, and they need to be added to the Java environment:

- **slf4j-api-1.7.5.jar**
- **cassandra-driver-core-2.0.2.jar**
- **guava-16.0.1.jar**
- **metrics-core-3.0.2.jar**
- **netty-3.9.0.Final.jar**

All can be put into a folder, and that folder can be put directly in Hadoop as follows:

```
[hadoop@mayank]$ gedit ~/.bashrc  
export CLASSPATH =  
$CLASSPATH:/home/hadoop/Cassandra_jars/*
```



```
Cassandra CLI Utility
Starting Cassandra Client
Welcome to Cassandra CLI version 1.0.5

Type 'help;' or '??' for help.
Type 'quit;' or 'exit;' to quit.

[default@unknown] connect localhost/9160;
Connected to: "Test Cluster" on localhost/9160
[default@unknown] use hr;
Authenticated to keyspace: hr
[default@hr] get emp[1];
=> <column=emp_dept, value=eng, timestamp=1326551681072001>
=> <column=emp_first, value=fred, timestamp=1326551681072000>
=> <column=emp_last, value=smith, timestamp=1326551681072002>
Returned 3 results.
Elapsed time: 21 msec(s).
[default@hr] _
```

**Figure 7.4:** Cassandra shell for server

To run CLI on Linux, use the following command:

**>bin/cassandra-cli**

## Basic CLI commands

These are the following basic commands that can be used to check about the proper installation of Cassandra:

- **Help:** By using **[default@Keyspace1]**, help can elaborate on all lists of commands that are useful in showing a list of metadata and its configuration:

**[default@Keyspace1] help**

List of all CLI commands:

Display this message.

**help** Display this help.

**help <command>** Display detailed, command-specific help.

connect <hostname>/<port> Connect to thrift service.

use <keyspace> [<username> 'password'] Switch to a keyspace.

describe keyspace <keyspacename> Describe keyspace.

exit Exit CLI.

quit Exit CLI.

show cluster name Display cluster name.

show keyspaces Show list of keyspaces.

show api version Show server API version.

create keyspace <keyspace> [with <att1>=<value1> [and <att2>=<value2> ...]]

Add a new keyspace with the specified attribute and value(s).

create column family <cf> [with <att1>=<value1> [and <att2>=<value2> ...]]

Create a new column family with the specified attribute and value(s).

drop keyspace <keyspace> Delete a keyspace.

drop column family <cf> Delete a column family.

rename keyspace <keyspace> <keyspace\_new\_name>

Rename a keyspace.

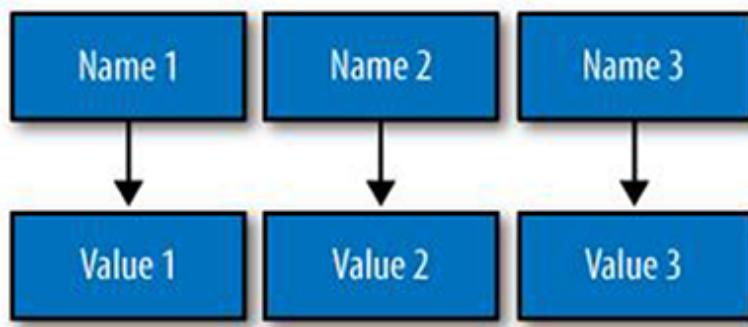
rename column family <cf> <new\_name> Rename a column family.

- **Creating keyspace and column family:** Cassandra is a sort of relational database that can define one or more column families. It shows the following output:

```
>bin/Cassandra-cli -host localhost -port9160
Starting Cassandra Client
Connected to "Test Cluster" on localhost/9160
Welcome to Cassandra CLI
Type 'help' or '?' for help. Type 'quit' or
'exit' to quit
[default@unknown]
```

## Cassandra data model

Traditionally, there are a number of useful data structures, but they are not semantically rich. For example, an array is the simplest form of storing data in which value can be easily stored and can be deleted. But in the case of empty data, there is no need to fill the created empty space, and that is useless. In that case, it is not for performing enhancement of data structure. To keep this in mind, take a look at *Figure 7.5*:



*Figure 7.5: Data model*

The preceding data model is suitable for one instance of any entity. But if there is a need to store multiple entities such as a person's details, tweets,

hotel addresses, and so on, then there will be a need for a collection of structures. Repetitions are also not desirable in case of multiple entries. In that case, the key to reference a group of columns is treated together as a group. Those set of columns form column families that are associated with similar data, meaning that there may be a hotel column family, address book column family, personal detail column family, and so on.

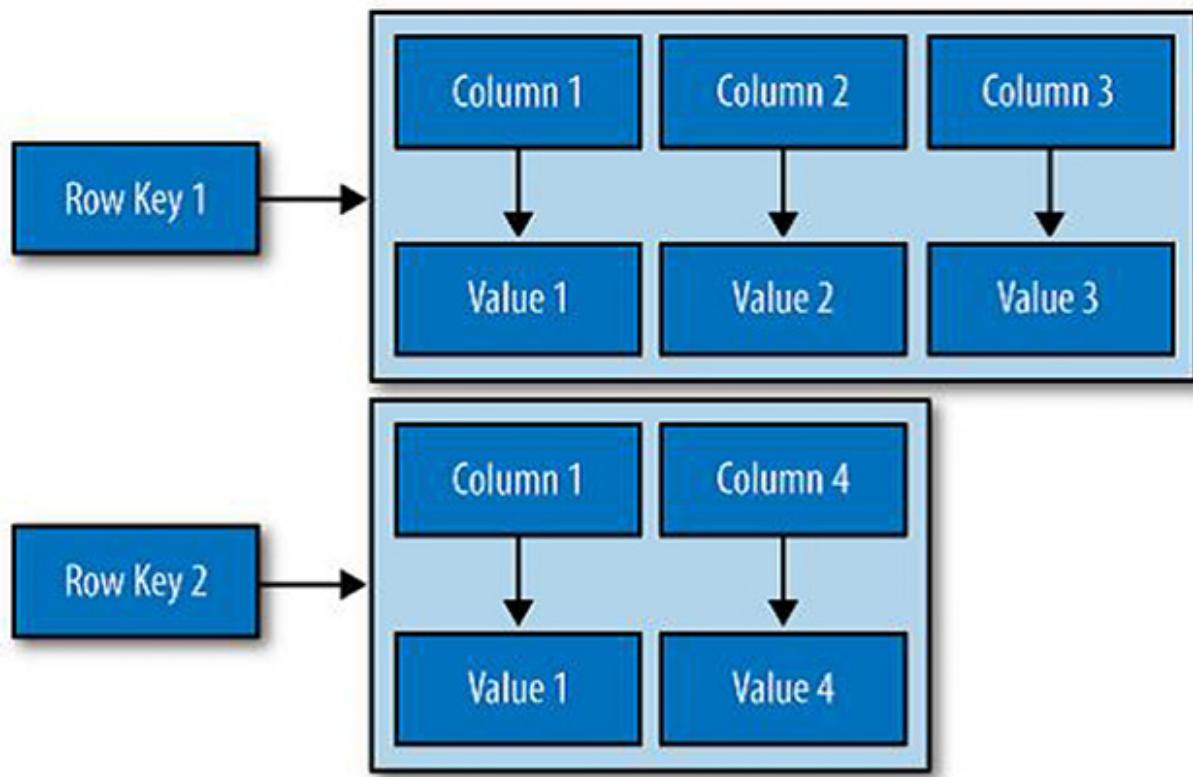
These column families are divisions as logical that form similar data. So, Cassandra forms data structures that have columns with name/value pairs and column families that contain rows with similar column sets. There is also no restriction to store column names as strings like relational database. Row and column both can be string. It can also be a long integer, UUIDs, byte array, and so on. Data can be stored in the key itself and not only in the value. Cassandra is useful for that kind of database that has multiple entries or blank entries. For example, some people want to enter phone number or not, or there might be a few people who do not want to enter other personal details that is sparse:

Player	ColumnFamily 1
xyz:	RowKey
email: <a href="mailto:xyz@gmail.com">xyz@gmail.com</a>	ColumnName:Value
game: cricket	ColumnName:Value
abc:	RowKey
email: <a href="mailto:abc@yahoo.com">abc@yahoo.com</a>	ColumnName:Value
student:	ColumnFamily 2
abc:	RowKey
pfunk: 1968-2010	ColumnName:Value

**Table 7.1:** Table schema of Cassandra

The preceding scenario is about two column families, that is, player and student. All records can be inserted using its row keys, and columns can be

inserted using column values with individual entries of column families. The multidimensional array data structure is shown in [Figure 7.6](#):

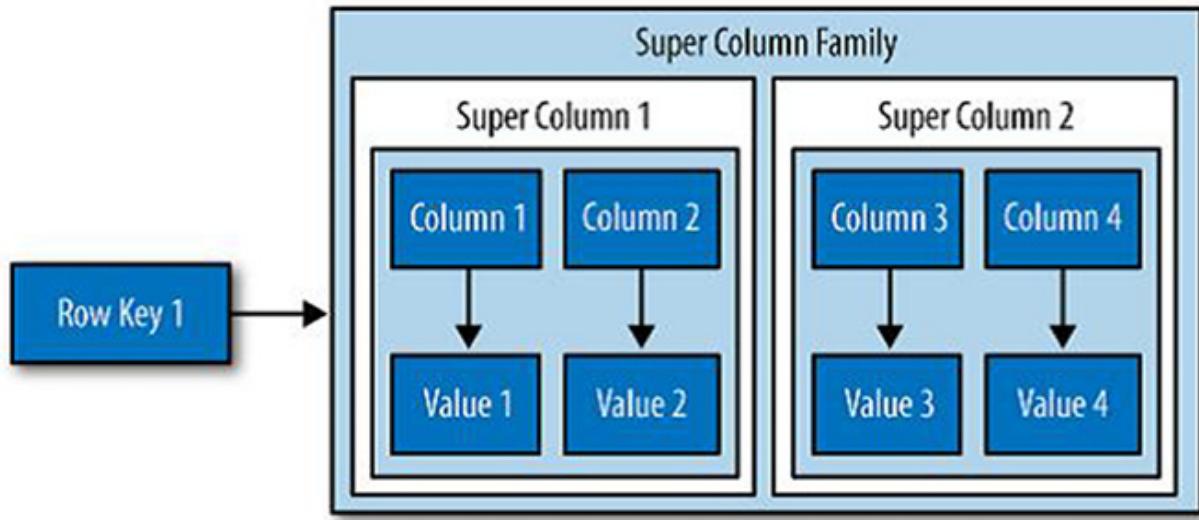


**Figure 7.6: Column family**

## **Super column family**

A group of related columns form another column family is called a super column family. To optimize performance for columns likely to be queried together.

It can be structured as a map of maps, as shown in [Figure 7.7](#):



*Figure 7.7: Super column family*

Multiple numbers of columns can be formed with super columns that have their unique name. Each row has its row key that identifies as a super column.

## Clusters

Cassandra is not useful if it takes care of just a single node. That is the requirement only for distributed environment clusters. These clusters can also be called rings because Cassandra assigns data to nodes in cluster-like arrangements in a ring.

Different ranges of data can be held with a node. If any time a node is down, then the replica can respond to the queries. Peer-to-peer allows data to replicate across all nodes to make it transparent to the user with a replication factor.

## Keyspaces

The outermost container for data in Cassandra is called keyspace. Keyspace has the name and set of attributes that define behavior. The most frequent way is to create a single keyspace with the application needed. Multiple key spaces on the same cluster can be considered depending on security constraints and partitioner. The following attributes can be set in a keyspace:

- **Replication factor:** It refers to the number of nodes that act as copies of each row of data. For example, three replication factors in the ring will have three copies in each row that are all transparent to the client.
- **Replica placement strategy:** This strategy is referred as its placement in a ring. These placements are based on determining which node gets the copies of keys.
- **Column families:** Column families are the collection of columns that are analogous to traditional databases. Each row has its ordered columns. This is useful for structuring the data of a table. Each keyspace has at least one and many other column families.

## **Column families**

It is a container with a set of rows with a collection of columns. At the time of physically creating a database, there is a requirement of mentioning keyspace (database), name of tables and name of column families of each table. In the Cassandra database, all column families are defined, and these are schema-free, whereas the columns are not. This means any columns can be added anytime as per requirement. Column families have two attribute names and comparators, which indicate that the column is sorted when it returns to the query.

Column families are stored as files, and these files cannot be treated as the table of a database like a traditional database. Relational databases create tables in which a combination of rows and columns exists. The user creates an entry that is stored as a row in each table while Cassandra can hold columns that are defined as super column family. Writing data to a column family in Cassandra means writing one or more columns in the database. Each row gets its uniqueness with its row key. These are the following column family options:

- **key\_cached**
- **row\_cached**
- **comment**
- **read\_repair\_chance**

- `preload_row_cache`

## Super columns

It is a special kind of column in Cassandra, and it stores byte array value that is map to a super column. A super column is specific and does not store value from other super columns. Each column family is already stored on disk as files, so it is important to keep the column together that is required to fetch at once as a group.

Here is an example of a super-column family definition called **PointOfInterest**. In the hotelier domain, a point of interest is a location near a hotel that travelers might like to visit, such as a park, museum, zoo, or tourist attraction:

`PointOfInterest (SCF)`

`SCkey: Cambria Suites Hayden`

`{`

`key: Phoenix Zoo`

`{`

`phone: 480-555-9999,`

`desc: They have animals here.`

`},`

`key: Spring Training`

`{`

`phone: 623-333-3333,`

`desc: Fun for baseball fans.`

`},`

```
}, //end of Cambria row

SCkey: (UTF8) Waldorf=Astoria
{
    key: Central Park
    desc: Walk around. It's pretty.

},
key: Empire State Building
{
    phone: 212-777-7777,
    desc: Great view from the 102nd floor.

}
}
}
```

## Cassandra examples

The following is the list of commands that are most frequently used, and these are basic for using Cassandra as a database.

### Creating a keyspace

Keyspace is a namespace that defines the replication of data on nodes in a distributed environment. A cluster can contain one **KEYSPACE** per node. The command is as follows:

```
CREATE KEYSPACE <identifier> WITH <properties>
```

```
CREATE KEYSPACE "KeySpace Name"
```

```
WITH replication = {'class': 'Strategy name',
'replication_factor': 'No.Of replicas'};
```

```
CREATE KEYSPACE "KeySpace Name"
```

```
WITH replication = {'class': 'Strategy name',
'replication_factor': 'No.Of replicas'}
```

```
AND durable_writes = 'Boolean value';
```

## Alter keyspace

It is used for changing properties of database like replication factor and **durable\_writes** of **KEYSPACE**. Look at the following example:

```
ALTER KEYSPACE <identifier> WITH <properties>
```

```
ALTER KEYSPACE "KeySpace Name"
```

```
WITH replication = {'class': 'Strategy name',
'replication_factor' : 'No.Of replicas'};
```

## Dropping a keyspace

Keyspace can be dropped using **DROP KEYSPACE** command, shown as follows:

```
DROP KEYSPACE <identifier>
```

```
DROP KEYSPACE "KeySpace name"
```

Apart from creating, deleting and altering the namespace, there are some other basic commands used in Cassandra.

## Create table

Tables can be created using this command with column and data type:

```
CREATE (TABLE | COLUMNFAMILY) <tablename>
('<column-definition>' , '<column-definition>')
(WITH <option> AND <option>)
column name1      data type,
column name2      data type,
example:
age      int,
name    text
```

## Primary key

This key is used to represent data uniquely in a table:

```
CREATE TABLE tablename(
    column1 name datatype PRIMARYKEY,
    column2 name data type,
    column3 name data type.
)
CREATE TABLE emp(
    emp_id int PRIMARY KEY,
    emp_name text,
    emp_city text,
    emp_sal varint,
    emp_phone varint
);
```

## Alter table

This is used to edit table attributes:

ALTER TABLE table name ADD new column datatype;	ALTER table name DROP column name;
ALTER TABLE stu ... ADD stu_email text;	ALTER TABLE emp DROP stu_email;

**Table 7.2: Alter table example**

## Truncate table

To delete a row permanently from the table, truncate can be used:

```
TRUNCATE <tablename>
```

```
TRUNCATE stu;
```

## Executing batch

Batch is useful for executing multiple statements simultaneously, such **insert**, **update**, and **delete**:

```
BEGIN BATCH
```

```
<insert-stmt>/ <update-stmt>/ <delete-stmt>
```

```
APPLY BATCH
```

```
BEGIN BATCH
```

```
... INSERT INTO emp (emp_id, emp_city, emp_name,  
emp_phone, emp_sal) values(  
1, 'Allahabad', 'nitin', 9810572618, 40000);
```

```
... UPDATE emp SET emp_sal = 50000 WHERE emp_id =1;
```

```
... DELETE emp_city FROM emp WHERE emp_id = 3;  
... APPLY BATCH;
```

## Delete entire row

To delete a particular row from the database, the following command is used:

```
DELETE FROM <identifier> WHERE <condition>;
```

```
DELETE FROM emp WHERE emp_id=2;
```

## Describe

It is used to describe the current cluster and its variants:

```
describe cluster; /describe keyspace;/describe  
table;
```

```
CREATE TABLE mayank.stu (  
    stu_id int PRIMARY KEY,  
    stu_city text,  
    stu_name text,  
    stu_phone varint,  
    stu_sub varint  
) WITH bloom_filter_fp_chance = 0.02  
AND caching = {'keys':'ALL',  
'rows_per_partition':'NONE'}  
AND comment = ''  
AND compaction = {'min_threshold': '4', 'class':
```

```
'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy',
    'max_threshold': '32'}

AND compression = {'sstable_compression':
    'org.apache.cassandra.io.compress.LZ4Compressor'}

AND dclocal_read_repair_chance = 0.1

AND default_time_to_live = 0

AND gc_grace_seconds = 864000

AND max_index_interval = 2048

AND memtable_flush_period_in_ms = 0

AND min_index_interval = 128

AND read_repair_chance = 0.0

AND speculative_retry = '99.0PERCENTILE';
```

```
CREATE INDEX stu_stu_sal_idx ON mayank.stu
(stu_sub);
```

## Cassandra client

Cassandra clients are software libraries or drivers that enable applications to interact with Apache Cassandra, a distributed NoSQL database. These client libraries provide the necessary APIs and functionality to perform various operations on Cassandra, including read and write operations. When it comes to write operations, Cassandra clients offer interfaces to insert, update, or delete data in Cassandra tables. They handle the complexity of data distribution, replication, and consistency levels, allowing developers to write

data to the database with ease. Clients often support features like batch writes, prepared statements, and data serialization/deserialization, making it straightforward for applications to store and manage data efficiently in Cassandra's distributed and fault-tolerant architecture. Popular Cassandra clients include DataStax Java Driver, Cassandra CQL Python Driver, and others tailored for different programming languages, offering developers flexibility in choosing the right client for their application stack.

Thrift serves as the base for client applications up to Cassandra 0.6 or earlier. But from 0.7, Avro also came into the picture due to the limitations of the thrift server interface. The Thrift server did not get any further updates since 2009 and is running with version 0.2.

The following sections will discuss clients that are considered as basic clients of Cassandra.

## Thrift

Thrift was developed at Facebook, and after that, it was used as an Apache project with Incubator status in 2008. Thrift uses C++, C#, Erlang, Java, Perl, PHP, Python, Smalltalk, and so on. All object-oriented languages are compatible for providing support for efficient RPC calls in a wide variety. It defines its service interface and data type in the service definition. With that service, it generates the RPC client code libraries for each language. The design of thrift focuses on the following features:

- Thrift is language independent, or it can be called language neutral because C++ structure can be exchanged with a Python dictionary.
- The same application code can be used in disk files, in-memory data, or streaming sockets. So, it provides a common transport method.
- Thrift is protocol-independent as it encodes and decodes the data type for use on any protocol.
- It also supports different versions for its update to client API.

File that use its services use **.thrift** extension. The files are stored in Cassandra's inner folder name **interface** in which the folder named

**cassandra.thrift** contains all this. The data structure of Cassandra will look like the following:

```
//data structures

struct Column {
    1: required binary name,
    2: required binary value,
    3: required i64 timestamp,
}

struct SuperColumn {
    1: required binary name,
    2: required list<Column> columns,
}

//exceptions

exception NotFoundException {
}

//etc...

//service API structures

enum ConsistencyLevel {
    ZERO = 0,
    ONE = 1,
    QUORUM = 2,
    DCQUORUM = 3,
```

```
DCQUORUMSYNC = 4,
ALL = 5,
ANY = 6,
}

struct SliceRange {
    1: required binary start,
    2: required binary finish,
    3: required bool reversed=0,
    4: required i32 count=100,
}

struct SlicePredicate {
    1: optional list<binary> column_names,
    2: optional SliceRange slice_range,
}

struct KeyRange {
    1: optional string start_key,
    2: optional string end_key,
    3: optional string start_token,
    4: optional string end_token,
    5: required i32 count=100
}

//service operations
```

```

service Cassandra {

    # auth methods

    void login(1: required string keyspace,
2:required AuthenticationRequest
            auth_request)

    throws (1:AuthenticationException authnx,
2:AuthorizationException

            authzx),

    i32 get_count(1:required string keyspace,
2:required string key,
3:required ColumnParent column_parent,
4:required ConsistencyLevel
consistency_level=ONE)

    throws (1:InvalidRequestException ire,
2:UnavailableException ue,
3:TimedOutException te),
//etc...

//meta-APIs

/** list the defined keyspaces in this cluster */
set<string> describe_keyspaces(),
//etc...

```

In Cassandra distribution, there is a folder **interface** in which there is a folder named with **.thrift** extension also has a folder with the **thrift** name that contains bindings of each language.

## **Avro**

This can be treated as a replacement for Thrift in Cassandra. Avro was created by Doug Cutting, inventor of Apache Hadoop and Apache MapReduce. Many facilities are similar to Thrift, like its robust data structure, efficient and binary format for RPC calls and easy integration with all languages such as Python, Ruby, C++, PHP, and so on.

Avro is also RPC like Cassandra 0.7. It generates code that remote clients can use for interaction with the database and support that grows with Hadoop projects.

## **Hector**

This is .NET-based and used for Apache Cassandra with a distributed environment. There are some features for clients:

- It provides an object-oriented interface to Cassandra.
- It is having a failover behavior on the client side.
- It provides configurable and extensible load balancing.
- It discovers the host automatically for the cluster.
- It provides type-safe approach to deal with Apache Cassandra's data model.

## **Hadoop integration**

In today's scenario, Cassandra plays an important role in getting business advantage with its integration with Hadoop. It is a leading platform for its framework in distributed environments. Cassandra provides linear scalability and high availability without any effect on performance. With the help of Cassandra and the Hadoop integration, the following benefits can be achieved:

- With the use of 800 pre-built connectors, Cassandra can connect quickly to any database.

- NoSQL connectivity provides access to Cassandra for speed development without prior knowledge of NoSQL.
- As it supports distributed environment so it is easily scalable. Once a Hadoop connector is configured, its code is automatically generated as a new data cluster.
- It ensures accuracy, integrity and completeness of data for better visibility.

Cassandra supports its configuration with Hadoop by two methods that is:

- MapReduce
- Apache Pig

## **Use cases**

Out of the Cassandra community, the following are the use cases that can help to understand about use of Cassandra in real scenarios:

### **eBay**

Nowadays, people are involved with e-commerce, and as a result, there are a lot of companies coming into the market. Every company wants to lead in this field. eBay is also a big giant in this scenario. To remain in competition, companies need to adopt the current scenario and requirements of users. For that case, they need to involve themselves with large data centers, quick solution, large storage, and play with unstructured/structured data at runtime. Every individual has their own priority, so for handling such scenario, eBay started a recommendation system that works on a personalize basis with quick analysis.

To fulfill this massive requirement of handling data, eBay requires Cassandra to handle its database that is stored in the data center across the world, and all analysis should be in real-time. DataStax enterprise provides its solution with eBay. Currently, eBay is handling over six billion writes per day with DataStax Enterprise clusters and over five billion reads per day.

eBay also processes real-time data with the priority, and it also detects fraud cases SOA requests/responses and maintains analytical reports.

## Hulu

It is an online video service provider that was established in 2007 in the USA. It is mainly focused on video series with current and past video records. It has more than six million paid customers. Hulu relies completely on Cassandra to stream videos without interruption.

Problems that Hulu faced included usage of multiple devices, real-time access and no interruption. The volume of data and its real-time solution are always supposed to have this application. Scaling of writes from millions of users and parallel read operations were the major challenges for Hulu.

To meet its requirement, Hulu experimented with Hbase, and then it found Hadoop to set up all the architecture of master/slave, which runs HDFS. It can also be affected by a single point of failure. Hulu can also be affected by cascading failure of systems with an all-region server.

With Cassandra, Hulu found 100% uptime and could meet a large workload through masterless architecture. This advantage also adds to its elimination of a single point of failure. Cassandra also performs ahead with its performance with availability and replication in comparison with other NoSQL databases.

## Conclusion

The study of NoSQL databases unveils a dynamic landscape of data management solutions that have reshaped the way we handle and process information in today's digital era. Through this exploration, we have gained valuable insights into the diverse objectives and benefits that NoSQL databases offer.

Our journey through NoSQL databases has highlighted their pivotal role in addressing the limitations of traditional relational databases. Their ability to seamlessly scale, adapt to evolving data structures, and cater to a wide range of data types has positioned them as indispensable tools for modern

applications. The flexibility they bring to data modeling, coupled with their capacity to handle large volumes of unstructured data, has redefined the boundaries of what is achievable in terms of data storage and retrieval.

Moreover, our examination of NoSQL databases has underscored their significance in enabling real-time applications and data-intensive analytics. Their proficiency in managing intricate relationships within data, whether in the form of graphs or geospatial information, has unlocked new dimensions for businesses to glean meaningful insights and make informed decisions.

It becomes evident that NoSQL databases are not just alternatives to traditional databases; they represent a paradigm shift that aligns with the demands of today's fast-paced, data-driven world. By embracing NoSQL databases, organizations empower themselves to innovate, scale, and adapt to the ever-evolving landscape of technology and data management. Through this understanding, we equip ourselves with the tools to navigate the challenges and harness the opportunities that arise in the realm of modern data architecture.

In essence, the study of NoSQL databases opens doors to a new era of data management possibilities, ushering in a future where flexibility, scalability, and performance converge to shape the data-driven applications of tomorrow.

In the upcoming chapters, we will learn about PigLatin and HiveQL languages, offering valuable insights into their utility in data processing.

## **Join our book's Discord space**

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

[https://discord.\(bpbonline.com](https://discord(bpbonline.com)



# CHAPTER 8

## Hadoop Related Tools-II: PigLatin and HiveQL

### Introduction

Hadoop offers a comprehensive solution for both storage and real-time data processing. In the realm of analytics, various tools are at your disposal. From a programming perspective, MapReduce can be applied in various scenarios, but it can be quite time-consuming to develop programs for tackling unstructured or structured data. MapReduce encompasses distinct map cycles that involve tasks like creating mappers, reducers, data streaming, data storage, compilation, packaging, and execution.

When it comes to structured data analytics, PigLatin and Hive are pivotal players. This chapter will delve into an in-depth exploration of the performance capabilities of both of these analytical tools.

### Structure

In this chapter, we will be covering the following topics:

- PigLatin
- Installation
- Execution types
- The platform for running Pig programs
- Grunt
- Pig data model
- PigLatin
- Developing and testing the PigLatin script

- Hive
- Data type and file format
- Comparisons of HiveQL with traditional database
- HiveQL

## Objectives

The primary objective of this chapter on PigLatin and HiveQL languages is to provide a comprehensive understanding of their roles and functionalities in the context of data processing and analysis. Through these chapters, readers will gain insights into how PigLatin and HiveQL enable efficient and scalable data manipulation, querying, and transformation within the realm of big data and distributed computing.

By exploring PigLatin, readers will grasp its purpose as a high-level scripting language designed to simplify complex data transformations on Hadoop clusters. The chapter aims to equip readers with the skills to write PigLatin scripts that effectively process large datasets, enabling them to harness the power of parallel processing and data flow optimization.

In the case of HiveQL, the objective is to familiarize readers with its role as a declarative query language for querying and managing structured data stored in Hadoop. The chapter aims to enable readers to craft HiveQL queries to extract valuable insights from data, bridging the gap between traditional SQL and the distributed nature of big data systems.

Overall, the objective of these chapters is to empower readers with practical knowledge and hands-on experience in using PigLatin and HiveQL languages as essential tools for data engineers and analysts working in modern data ecosystems. By mastering these languages, readers will be better equipped to efficiently process, analyze, and derive meaningful insights from large-scale datasets.

## Apache PigLatin

**Apache PigLatin** is an analytical platform that runs on Apache Hadoop. Pig has the capability to execute its assigned tasks on Hadoop by using the underlying MapReduce framework. This process takes place in the background and delivers

outcomes to users through queries. PigLatin was developed by *Yahoo researchers* and the *Apache Foundation*. It was released on 11 September 2008. Currently, version 0.17.0 is on the market, which was released on 19 June 2017. PigLatin, which is the language of the Pig, has the following key properties:

- It is easy to understand and program. All data transformation encoded with data flow sequence is easy to modify and maintain.
- It offers the possibility of granting users automated access to enhance efficiency and execution.
- Users can create their own functions for special purposes.
- **User Defined Functions (UDF)** are an important part of PigLatin that provide functionality for providing user-driven commands for handling data.

PigLatin is a scripting language tailored for handling extensive volumes of data, sidestepping the extensive development process associated with MapReduce. Its design places scalability at the forefront for developers. PigLatin encompasses functions related to loading, storing, filtering, grouping, and joining data. This language incorporates numerous integrated operators to streamline these operations, enabling users to perform analytical tasks without the need to write programs in MapReduce.

It is also designed for batch processing of data. Pig will not perform ideally when small amounts of data are involved for processing with a large dataset.

## Installation

The steps that are used in the installation of Pig over Hadoop with Ubuntu 16.04 are as follows:

(Refer: <http://mayankonweb.blogspot.in/2017/01/pig-installation-on-ubuntu-1604.html>)

1. Download Pig and store it in the **download** folder.

```
 wget http://www.apache.org/dist/pig/latest/pig-x.y.z.tar.gz
```

The preceding code is a set of commands to install and configure Apache Pig. The following is a breakdown of what each command does:

- **sudo mkdir /usr/lib/pig**: Creates a directory called pig in the **/usr/lib** directory with superuser permissions.
- **sudo cp Downloads/pig-0.16.0.tar.gz /usr/lib/pig/**: Copies the Pig installation file **pig-0.16.0.tar.gz** from the **Downloads** directory to the newly created **/usr/lib/pig** directory.
- **cd /usr/lib/pig/**: Changes the current directory to **/usr/lib/pig**.
- **tar -xvf pig-0.16.0.tar.gz**: Extracts the Pig installation files from the **pig-0.16.0.tar.gz** file.
- **gedit ~/.bashrc**: Opens the **.bashrc** file in the user's home directory with the **Gedit** text editor.

The last command is likely used to add Pig's binary path to the system's **PATH** variable by appending the following line to the end of the **.bashrc** file:

```
export PATH=$PATH:/usr/lib/pig/bin
```

This allows the Pig executable to be run from any directory in the command line.

2. Add the following in the **gedit** text editor:

```
# Pig Home directory
```

```
export PIG_HOME="/usr/lib/pig/pig-0.16.0"  
export PIG_CONF_DIR="$PIG_HOME/conf"  
export PIG_CLASSPATH="$PIG_CONF_DIR"
```

```
export PATH="$PIG_HOME/bin:$PATH"  
hduser@mayank-Compaq-510:/home/mayank$ pig -h
```

3. The following commands are used to check the availability of Pig, as shown in *Figure 8.1*:

```
hduser@mayank-Compaq-510:/home/mayank$ pig
17/01/14 21:23:55 INFO pig.ExecTypeProvider:
Trying ExecType : LOCAL
17/01/14 21:23:55 INFO pig.ExecTypeProvider:
Trying ExecType : MAPREDUCE
17/01/14 21:23:55 INFO pig.ExecTypeProvider:
Picked MAPREDUCE as the ExecType
17/01/14 21:23:55 WARN pig.Main: Cannot write to
log file: /home/mayank/pig_1484409235225.log
2017-01-14 21:23:55,228 [main] INFO
org.apache.pig.Main - Apache Pig version 0.16.0
(r1746530) compiled Jun 01 2016, 23:10:49
2017-01-14 21:23:55,270 [main] INFO
org.apache.pig.impl.util.Utils - Default bootup
file /home/hduser/.pigbootup not found
2017-01-14 21:23:56,350 [main] WARN
org.apache.hadoop.util.NativeCodeLoader - Unable
to load native-hadoop library for your platform...
using builtin-java classes where applicable
2017-01-14 21:23:56,356 [main] INFO
org.apache.hadoop.conf.Configuration.deprecation -
mapred.job.tracker is deprecated. Instead, use
mapreduce.jobtracker.address
2017-01-14 21:23:56,356 [main] INFO
org.apache.hadoop.conf.Configuration.deprecation -
fs.default.name is deprecated. Instead, use
fs.defaultFS
2017-01-14 21:23:56,356 [main] INFO
org.apache.pig.backend.hadoop.executionengine.HExe
```

```

cutionEngine - Connecting to hadoop file system
at: hdfs://localhost:54310

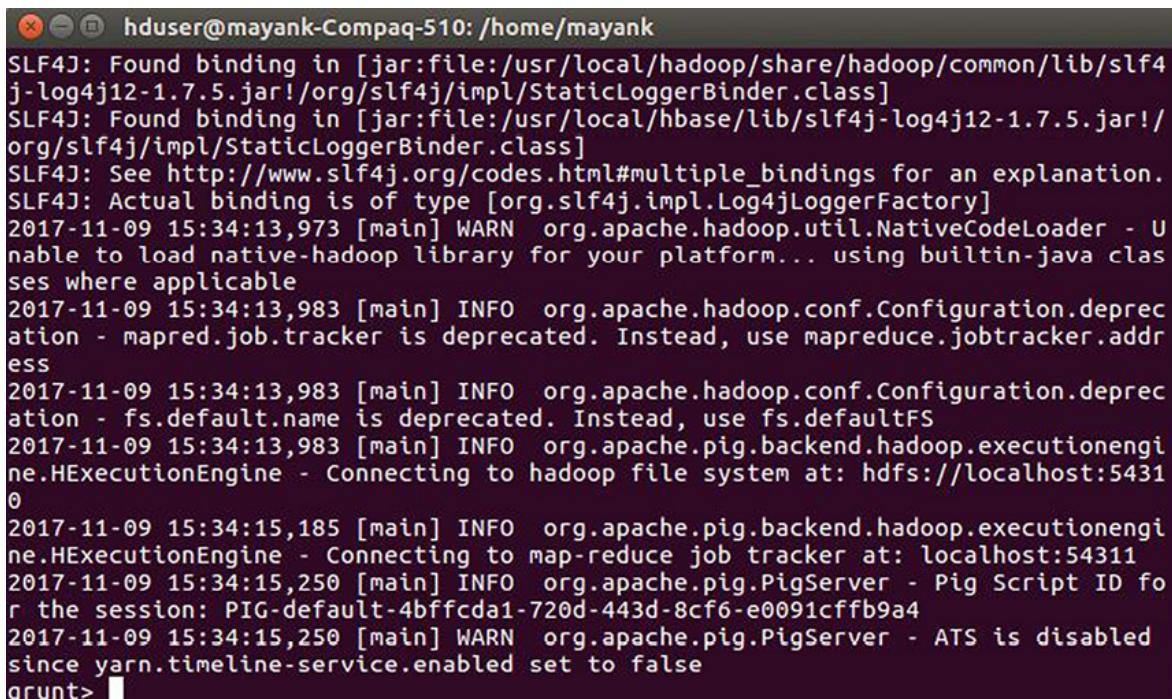
2017-01-14 21:23:57,215 [main] INFO
org.apache.pig.backend.hadoop.executionengine.HExe
cutionEngine - Connecting to map-reduce job
tracker at: localhost:54311

2017-01-14 21:23:57,249 [main] INFO
org.apache.pig.PigServer - Pig Script ID for the
session: PIG-default-e439e9ab-2054-46e3-9bff-
daa870b7450a

2017-01-14 21:23:57,249 [main] WARN
org.apache.pig.PigServer - ATS is disabled since
yarn.timeline-service.enabled set to false

grunt>

```



The screenshot shows a terminal window with the following log output:

```

hduser@mayank-Compaq-510: /home/mayank
SLF4J: Found binding in [jar:file:/usr/local/hadoop/share/hadoop/common/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/local/hbase/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
2017-11-09 15:34:13,973 [main] WARN org.apache.hadoop.util.NativeCodeLoader - Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
2017-11-09 15:34:13,983 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - mapred.job.tracker is deprecated. Instead, use mapreduce.jobtracker.address
2017-11-09 15:34:13,983 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
2017-11-09 15:34:13,983 [main] INFO org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to hadoop file system at: hdfs://localhost:54310
2017-11-09 15:34:15,185 [main] INFO org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to map-reduce job tracker at: localhost:54311
2017-11-09 15:34:15,250 [main] INFO org.apache.pig.PigServer - Pig Script ID for the session: PIG-default-4bffcda1-720d-443d-8cf6-e0091cffb9a4
2017-11-09 15:34:15,250 [main] WARN org.apache.pig.PigServer - ATS is disabled since yarn.timeline-service.enabled set to false
grunt> 

```

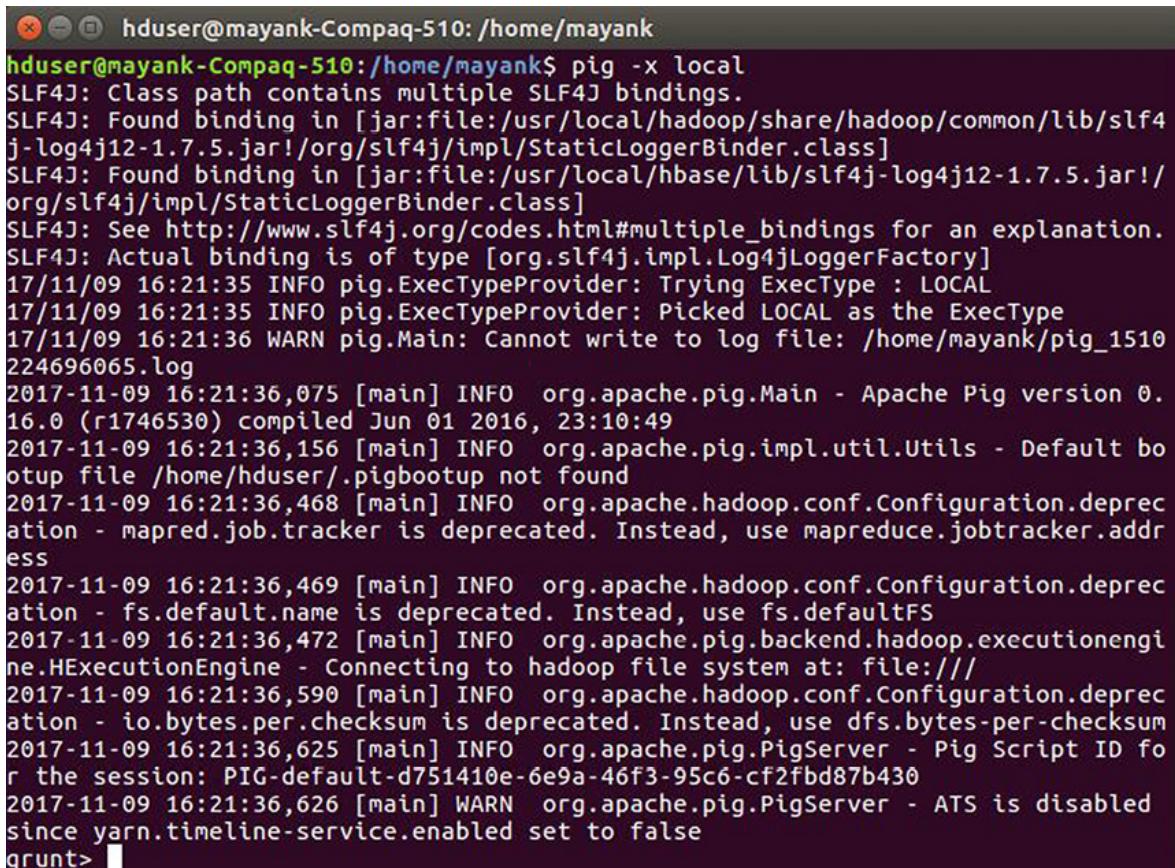
*Figure 8.1: Installation of Pig*

## Execution type

PigLatin functions with two types of mode: local mode and MapReduce mode.

## Local mode

In this mode, Pig runs with a single **Java Virtual Machine (JVM)** and accesses the local file system. This is useful and efficient with small datasets. To use it with the local system, refer to [Figure 8.2](#):



```
hduser@mayank-Compaq-510: /home/mayank
hduser@mayank-Compaq-510: /home/mayank$ pig -x local
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/local/hadoop/share/hadoop/common/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/local/hbase/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
17/11/09 16:21:35 INFO pig.ExecTypeProvider: Trying ExecType : LOCAL
17/11/09 16:21:35 INFO pig.ExecTypeProvider: Picked LOCAL as the ExecType
17/11/09 16:21:36 WARN pig.Main: Cannot write to log file: /home/mayank/pig_1510
224696065.log
2017-11-09 16:21:36,075 [main] INFO org.apache.pig.Main - Apache Pig version 0.
16.0 (r1746530) compiled Jun 01 2016, 23:10:49
2017-11-09 16:21:36,156 [main] INFO org.apache.pig.impl.util.Utils - Default bo
otup file /home/hduser/.pigbootup not found
2017-11-09 16:21:36,468 [main] INFO org.apache.hadoop.conf.Configuration.deprec
ation - mapred.job.tracker is deprecated. Instead, use mapreduce.jobtracker.add
ress
2017-11-09 16:21:36,469 [main] INFO org.apache.hadoop.conf.Configuration.deprec
ation - fs.default.name is deprecated. Instead, use fs.defaultFS
2017-11-09 16:21:36,472 [main] INFO org.apache.pig.backend.hadoop.executionengi
ne.HExecutionEngine - Connecting to hadoop file system at: file:///
2017-11-09 16:21:36,590 [main] INFO org.apache.hadoop.conf.Configuration.deprec
ation - io.bytes.per.checksum is deprecated. Instead, use dfs.bytes-per-checksum
2017-11-09 16:21:36,625 [main] INFO org.apache.pig.PigServer - Pig Script ID fo
r the session: PIG-default-d751410e-6e9a-46f3-95c6-cf2fdb87b430
2017-11-09 16:21:36,626 [main] WARN org.apache.pig.PigServer - ATS is disabled
since yarn.timeline-service.enabled set to false
grunt> █
```

*Figure 8.2: Pig as local mode*

## MapReduce mode

This mode turns queries into MapReduce jobs to run on Hadoop clusters. These clusters can be in pseudo or fully distributed mode. The pig version should be compatible with Hadoop to use MapReduce. This is the default case, so there is no need to specify MapReduce every time when need to start with the grunt cell, as shown in [Figure 8.3](#):

```
hduser@mayank-Compaq-510: /home/mayank
odemanager-mayank-Compaq-510.out
hduser@mayank-Compaq-510:/home/mayank$ pig -x mapreduce
17/11/10 15:15:36 INFO pig.ExecTypeProvider: Trying ExecType : LOCAL
17/11/10 15:15:36 INFO pig.ExecTypeProvider: Trying ExecType : MAPREDUCE
17/11/10 15:15:36 INFO pig.ExecTypeProvider: Picked MAPREDUCE as the ExecType
17/11/10 15:15:36 WARN pig.Main: Cannot write to log file: /home/mayank/pig_1510
307136315.log
2017-11-10 15:15:36,325 [main] INFO org.apache.pig.Main - Apache Pig version 0.
16.0 (r1746530) compiled Jun 01 2016, 23:10:49
2017-11-10 15:15:36,433 [main] INFO org.apache.pig.impl.util.Utils - Default bo
otup file /home/hduser/.pigbootup not found
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/local/hadoop/share/hadoop/common/lib/slf4
j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/local/hbase/lib/slf4j-log4j12-1.7.5.jar!/o
rg/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
2017-11-10 15:15:38,906 [main] WARN org.apache.hadoop.util.NativeCodeLoader - U
nable to load native-hadoop library for your platform... using builtin-java clas
ses where applicable
2017-11-10 15:15:38,927 [main] INFO org.apache.hadoop.conf.Configuration.deprec
ation - mapred.job.tracker is deprecated. Instead, use mapreduce.jobtracker.add
ress
2017-11-10 15:15:38,927 [main] INFO org.apache.hadoop.conf.Configuration.deprec
ation - fs.default.name is deprecated. Instead, use fs.defaultFS
2017-11-10 15:15:38,927 [main] INFO org.apache.pig.backend.hadoop.executionengi
ne.HExecutionEngine - Connecting to hadoop file system at: hdfs://localhost:5431
0
2017-11-10 15:15:40,579 [main] INFO org.apache.pig.backend.hadoop.executionengi
ne.HExecutionEngine - Connecting to map-reduce job tracker at: localhost:54311
2017-11-10 15:15:40,714 [main] INFO org.apache.pig.PigServer - Pig Script ID fo
r the session: PIG-default-6e8eb901-dc74-45ef-a7ad-90062f248c90
2017-11-10 15:15:40,715 [main] WARN org.apache.pig.PigServer - ATS is disabled
since yarn.timeline-service.enabled set to false
grunt> ■
```

*Figure 8.3: Pig in MapReduce mode*

## The platform for running Pig programs

There are three ways in which Pig programs can be executed. These will be discussed in detail in the upcoming sections.

### Script

A file with a .pig extension can be executed as a local command. This file serves as a concise script, encompassing all the logical commands needed to implement a user's conceptual ideas.

### Grunt

This is a Linux shell designed for executing Pig commands interactively. It operates through the Grunt shell, as illustrated in [Figure 8.1](#). This shell is particularly useful for efficiently processing a substantial number of files in a single command line. Additionally, within the Grunt shell, you have the option to employ the **run** and **exec** commands to execute tasks.

## Embedded

This results in the capability to programmatically access the Grunt shell through **PigRunner**, a component specified within **PigServer**.

## Grunt Shell

This shell script for Pig, known as PigPig, offers an editing feature that is used through command line arguments for executing Pig commands. PigPig operates in a mode where all commands are input in a single line, requiring users to enter a new line when they reach the end of a command, often referred to as a *token per line* approach.

- *Ctrl + E* takes the cursor to the end of the line.
- *Ctrl + P* or *Ctrl + N* is used for previous and next commands.
- Pressing the *Tab* button expands the keyword from the partially typed word.
- **Help** command provides all the information related to commands at once.
- **Quit** command in shell terminates the session.
- **PigPen** is a plug-in for eclipse that provides an environment for developing Pig programs.

## Example

As seen in wordcount in the previous chapter with MapReduce, the following is an example to solve the same problem through Pig:

1. Transfer the data to the distributed environment, as shown in [Figure 8.4](#):

```

drwxr-xr-x  - hduser supergroup      0 2017-01-24 06:54 /weatherdataoutnew
-rw-r--r--  1 hduser supergroup  48836 2017-08-01 14:02 /wordcount
drwxr-xr-x  - hduser supergroup      0 2017-09-09 14:13 /wordcountoutput
drwxr-xr-x  - hduser supergroup      0 2017-08-01 14:05 /wordcountoutputwit
hpartitioner
hduser@mayank-Compaq-510:/home/mayank$ █

```

*Figure 8.4: Data in a distributed environment*

2. To split the data into the number of lines, use the following command, as shown in *Figure 8.5*:

```
myinput = load '/wordcount' as (line);
```

```

grunt> myinput = load '/wordcount' as (line);
2017-11-18 13:42:17,953 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - mapred.job.tracker is deprecated. Instead, use mapreduce.jobtracker.address
2017-11-18 13:42:17,958 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> █

```

*Figure 8.5: Load file as line*

3. To generate a token from each line with the collection of records by separation of each one, use the following command, as shown in *Figure 8.6*:

```
words = foreach myinput generate
flatten(TOKENIZE(line)) as word;
```

```

grunt> myinput = load '/wordcount' as (line);
2017-11-18 13:42:17,953 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - mapred.job.tracker is deprecated. Instead, use mapreduce.jobtracker.address
2017-11-18 13:42:17,958 [main] INFO org.apache.hadoop.conf.Configuration.deprecation - fs.default.name is deprecated. Instead, use fs.defaultFS
grunt> words = foreach myinput generate flatten(TOKENIZE(line)) as word;
2017-11-18 13:43:20,285 [main] INFO org.apache.pig.impl.util.SpillableMemoryManager - Selected heap (PS Old Gen) of size 699400192 to monitor. collectionUsageThreshold = 489580128, usageThreshold = 489580128
2017-11-18 13:43:20,327 [main] WARN org.apache.pig.newplan.BaseOperatorPlan - Encountered Warning USING_OVERLOADED_FUNCTION 1 time(s).
2017-11-18 13:43:20,327 [main] WARN org.apache.pig.newplan.BaseOperatorPlan - Encountered Warning IMPLICIT_CAST_TO_CHARARRAY 1 time(s).
grunt> █

```

*Figure 8.6: Separation of the file as a word*

4. To collect words as a group, use the following command. Refer to *Figure 8.7*:

```
grpds = group words by word;
```

```

grunt> grpfd = group words by word;
2017-11-18 13:43:57,220 [main] WARN org.apache.pig.newplan.BaseOperatorPlan - E
ncountered Warning USING_OVERLOADED_FUNCTION 1 time(s).
2017-11-18 13:43:57,220 [main] WARN org.apache.pig.newplan.BaseOperatorPlan - E
ncountered Warning IMPLICIT_CAST_TO_CHARARRAY 1 time(s).
grunt> █

```

*Figure 8.7: Grouping of word*

- From each group, it counts the number of words that were stored in **cndt**. Refer to *Figure 8.8*.

```
cndt = foreach grpfd generate group, COUNT(words);
```

```

grunt> cndt = foreach grpfd generate group, COUNT(words);
2017-11-18 13:44:34,258 [main] WARN org.apache.pig.newplan.BaseOperatorPlan - E
ncountered Warning USING_OVERLOADED_FUNCTION 1 time(s).
2017-11-18 13:44:34,258 [main] WARN org.apache.pig.newplan.BaseOperatorPlan - E
ncountered Warning IMPLICIT_CAST_TO_CHARARRAY 1 time(s).
grunt> █

```

*Figure 8.8: Group and counting of word*

- Dump or store the words, that have been counted using the following command as shown in *Figures 8.9, Figure 8.10, and Figure 8.11*:

```
dump cndt;
```

```

grunt> dump cndt;
2017-11-18 13:45:10,303 [main] WARN org.apache.pig.newplan.BaseOperatorPlan - E
ncountered Warning USING_OVERLOADED_FUNCTION 1 time(s).
2017-11-18 13:45:10,303 [main] WARN org.apache.pig.newplan.BaseOperatorPlan - E
ncountered Warning IMPLICIT_CAST_TO_CHARARRAY 1 time(s).
2017-11-18 13:45:10,338 [main] INFO org.apache.pig.tools.pigstats.ScriptState -
Pig features used in the script: GROUP_BY

```

*Figure 8.9: Dump words*

Refer to the following figure to check the execution:

```

HadoopVersion PigVersion UserId StartedAt FinishedAt Features
2.6.0 0.16.0 hduser 2017-11-18 13:45:11 2017-11-18 13:45:21 GROUP_BY

Success!
Job Stats (time in seconds):
JobId Maps Reduces MaxMapTime MinMapTime AvgMapTime MedianMapTime MaxReduceTime MinReduceTime AvgReduceTime MedianRe
ducetime Alias Feature Outputs
job_local134969588_0001 1 1 n/a n/a n/a n/a n/a n/a n/a cndt,grpfd,myinput,words GROUP_BY,COMBINE
R hdfs://localhost:54310/tmp/temp290555563/tmp474912458,
Input(s):
Successfully read 103 records (10854086 bytes) from: "/wordcount"
Output(s):
Successfully stored 330 records (10761591 bytes) in: "hdfs://localhost:54310/tmp/temp290555563/tmp474912458"

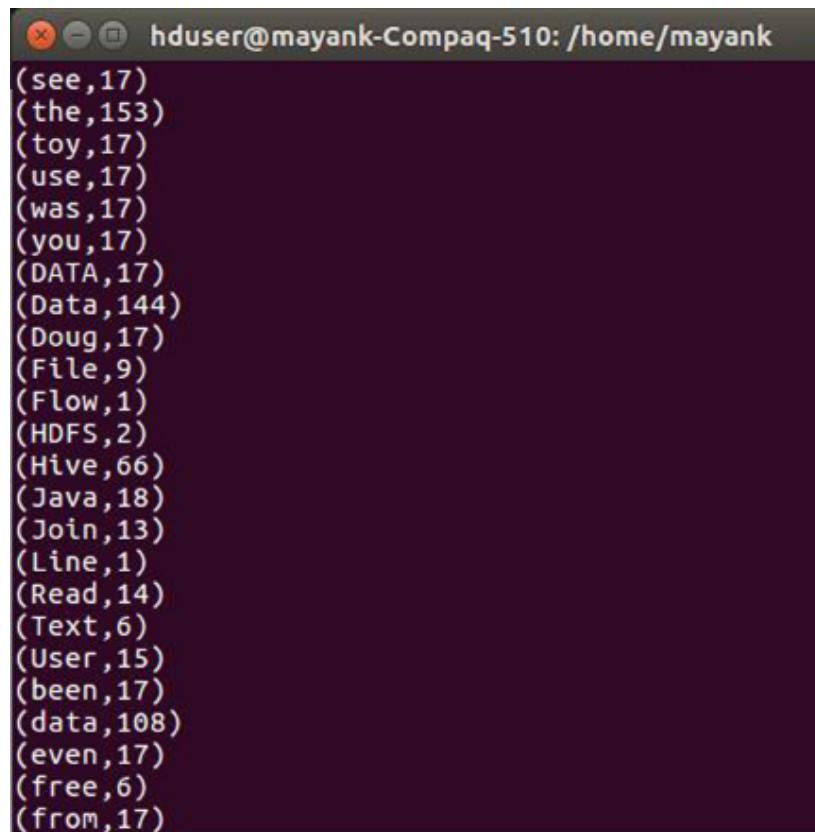
Counters:
Total records written : 330
Total bytes written : 10761591
Spillable Memory Manager spill count : 0
Total bags proactively spilled: 0
Total records proactively spilled: 0

Job DAG:
job_local134969588_0001

```

**Figure 8.10:** Checkup of execution

Refer to the following figure, to see the total word count:



The screenshot shows a terminal window with the title bar "hduser@mayank-Compaq-510: /home/mayank". The window contains a list of words and their counts, resulting from a word count operation. The output is as follows:

```
(see,17)
(the,153)
(toy,17)
(use,17)
(was,17)
(you,17)
(DATA,17)
(Data,144)
(Doug,17)
(File,9)
(Flow,1)
(HDFS,2)
(Hive,66)
(Java,18)
(Join,13)
(Line,1)
(Read,14)
(Text,6)
(User,15)
(been,17)
(data,108)
(even,17)
(free,6)
(from,17)
```

**Figure 8.11:** Total word count

## Commands in grunt

*Table 8.1* shows the commands that are used in the grunt shell to use data in an efficient manner:

Functionality	Operator	Description
Loading and storing	LOAD	Use to load data from the file system
	STORE	Save data in the local path
	DUMP	Display data on the screen
Filtering	FILTER	Remove not required row

	<b>DISTINCT</b>	Remove duplicate rows
	<b>FOREACH...GENERATE</b>	Add or remove fields from the relation
	<b>MAPREDUCE</b>	Run MapReduce using relation as input
	<b>STREAM</b>	Transform relation
	<b>SAMPLE</b>	Select a random sample of relation
Grouping and Joining	<b>JOIN</b>	Join two relations
	<b>COGROUP</b>	Group data in two or more relation
		Group data in a single relation
	<b>CROSS</b>	Create cross-product of two or more relation
Sorting	<b>ORDER</b>	Apply sorted order
	<b>LIMIT</b>	Limit the size of the relation
Combining and splitting	<b>UNION</b>	Combine two or more relation
Describe	<b>DESCRIBE</b>	Print schema of relation
	<b>EXPLAIN</b>	Print logical and physical plan
	<b>ILLUSTRATE</b>	Show sample execution of the logical plan
Register	<b>REGISTER</b>	Register the JAR file of the Pig program
	<b>DEFINE</b>	Create an alias for macro and use it in UDF
	<b>IMPORT</b>	Import macros defined in file

**Table 8.1:** Commands used in grunt shell

## Pig data model

The data model of **PigLatin** provides an understanding of the basics of any language. Here, the Pig data model discusses its data type and its relationship with others:

Category	Types
Scalar	<code>int: java.lang.Integer</code> (Store 4-byte signed integer) <code>long: java.lang.Long</code> (Store 8-byte signed integer) <code>float: java.lang.Float</code> (Store 4-byte signed integer) <code>double: java.lang.Double</code> (Store 8-byte signed integer) <code>chararray: java.lang.String</code> <code>bytearray: java.io.ByteArrayOutputStream</code>
Complex	<code>map: java.util.Map&lt;Object, Object&gt;</code> (chararray to data element mapping) <code>tuple: org.apache.pig.data.Tuple</code> (fixed length ordered collection of element) <code>bag: org.apache.pig.data.DataBag</code> (unordered collection of tuples) <code>null: any type of data can be null</code>

**Table 8.2:** Different data modes in Pig

## Scalar

It is the same as using any other programming language. Most of its types fall in the **java.lang** or **java.io** package. Following are the different data types of scalars:

- **int:** Constant integer can be expressed as an integer number, for example, 16. Its interface can be represented by **java.lang.Integer**.

- **long**: Its interface can be represented in **java.lang.Long**. For example, `700000000L`.
- **float**: This kind of number, for example, `2.56435f` can be expressed in **java.lang.Float**.
- **double**: This is stored as a double precision floating point number that is expressed in **java.lang.Double**. For example, `7.452e-32`.
- **chararray**: It is represented in the interface by **java.lang.String**. It can be expressed with single quotes and also using backslash codes. For example, `\u0001`, `'mayank'`.
- A **bytearray** It represented in Java as **java.io.ByteArrayOutputStream**, which wraps a Java byte array, for example, like this: `3D 22 4E`.

## Complex

These data types include maps, tuples, bags, and so on. It includes data of any type with unique structure and properties. Following are the different data types under the Complex data model:

- **map**: In Pig, a **Map** is used for mapping data elements, where each data element can belong to any Pig data type. Essentially, a data type can serve as a key, with its corresponding element acting as a value. Pig, by default, does not impose any restrictions on the value's data type, so it can be quite diverse. Initially, it is treated as a **bytearray**, allowing the entire data to be read at once. Subsequently, it can be parsed using delimiters. Importantly, all value-type data elements do not have to be of the same type; they can vary. This flexibility is what makes the mapping technique valuable for reading comprehensive information in a single pass from a data source.
- **tuple**: Tuples are divided into fields with one data element, which is an ordered collection of pig data. These data can be of any type. Tuples are fixed in length, and it is analogous to a row in SQL. Fields in tuples can be referred to by their position as they are ordered.
- **bag**: It is different from tuples as these are an unordered collection of tuples. It cannot be referred to by its position. Schema can describe all

tuples within the bag. It can be indicated with curly braces like `{('mayank', 77), ('nitin', 34), ('alok', 12)}`, these are constructed with three tuples with two fields in each. A point to be noted here is that Pig does not provide a set type or list that can store items of any kind, but it is possible by wrapping data type in a tuple of one field. When grouping items, bags can be too large to store, and they do not fit in memory.

It is not easy to calculate the actual amount needed by Pig data type for memory as Pig uses Java data type. It is, however, easy to predict for a single item but not for a whole document. This means that the multiplication factor between disk and memory required to store uncompressed data depends on the specific data being stored. In general, it takes approximately four times as much memory as disk space to hold uncompressed data.

- **null**: In Pig, any type of data can be null, and this concept of null is akin to SQL's notion of null, but it differs from null in programming languages such as C, Java, Python, and others. In Pig, when data is null, it is treated as unknown or missing, whereas other values are considered valid addresses or objects.

## PigLatin

In this section, there will be deep learning of PigLatin. PigLatin is a data flow language, so in each processing step, it creates a new dataset or relation that is assigned with the keyword. These relations need to be taken once in a single session; after that, it will end, and new keywords need to be assigned. In any case, if the customer records are considered, then according to Pig conditions query can be written as follows.

This Pig Latin script loads data from the `/input/custs` file using it as a delimiter and assigns the column names to the relation fields.

```
A = cust = LOAD '/input/custs' USING PigStorage(',')  
AS (  
    custid:chararray,  
    firstname:chararray,
```

```

lastname:chararray,
age:long,
profession:chararray
);

```

However, the preceding query is not recommended because when there is a reference as **A**, which is not easy to track and understand further. This issue can create problems during backtracking. A programmer can assign a reference **A** differently in various scenarios, which can result in confusion. Additionally, it is important to note that relation and field names in Pig must commence with an alphabetic character and consist of ASCII characters throughout.

PigLatin cannot decide its case sensitivity because the reference can be considered as **A** and **a**, and both are different references while there is no difference in **load** or **LOAD**.

There are two types of comments in Pig that can be drawn as `( - - )` and `(* */)`.

```

amt = limit cust 100;
/* amount received with limitation */
dump amt;
/*show all amount in terminal*/

```

## **Input and output**

*Table 8.3* shows the commands that are used for providing input and output for the system:

Command	Description	Example
LOAD	Load is the initial step to load all the data to pig directories so	<pre> cust      =   load      '/input/custs HbaseStorage('',''); </pre> <p>(Here, <code>input/custs</code> is the directory where raw data is needs to access in pig using <code>HbaseStorage</code> that is sepa</p>

that it can be further processed.

All raw data need to be in HDFS.

When specifying a directory as the input in a load statement in Pig, it will consider all files present in that directory (and any subdirectories) as input for the load statement. This means that if there are multiple data files present in the directory, Pig will read all of them as input. Additionally, if there are any subdirectories in the directory specified, Pig will also include the files present in those subdirectories as input.

---

```
cust = load '/input/custs' using PigStc
```

(In this, data will be processed with comma as delimiter and that is the default case.)

---

```
cust = load '/input/custs' using PigStc  
(custid:chararray, firstnam  
lastname:chararray, age:long, profession:
```

(That is the data analysis part; after checking the format of loaded, all columns are defined with their names with type

**Table 8.3:** Commands in PigLatin

Pig offers two built-in functions, **PigStorage** and **TextLoader**, which work with HDFS files and support globs. Globs allow you to read multiple files that are not necessarily located within the same directory or selectively read some files while excluding others within a directory. However, it is important to note that the validity of globs can vary depending on the Hadoop version in use. If you issue Pig Latin commands from a Unix shell command line, you may need to escape many of the glob characters to prevent the shell from automatically expanding them:

Glob	Description
*	Matches zero or more characters.
?	Matches any single character.
[abc]	Matches a single character from a character set (a, b, c).
[a-z]	Matches a single character from a range of set (a...z).
[^abc]	Matches a single character that is not in the set.
[^a-z]	Matches a single character that is not in range (a....z).
\c	Remove any special meaning of character C.
{ab, cd}	Matches string from string set {ab, cd}.

**Table 8.4:** Glob characters

## Store

*Table 8.5* showcases the commands that are used for showing the data either on the terminal or in any file:

Command	Description	Example
STORE	After processing the data, it can be stored separately or shown on the terminal.	dump result; (It used to show all data once at the

	<p>By default, case output can be written in a single directory, but it can be further processed with parallel processing of data so that it can be created with a number of parts of data.</p> <p>The format of the output of dump in Pig Latin is used to match the format of constants in Pig Latin up to version 0.8. This means that longs were followed by an L and float by an F, and maps were surrounded by [ ] (brackets), tuples by ( ) (parentheses), and bags by { } (braces). Starting from version 0.8, the L for longs and F for floats have been removed, although the markers for the complex types are still there. Nulls are indicated by missing values, and fields are separated by commas. Since each record in the output is a tuple, it is surrounded by ( ).</p>	<p>terminal; after showing data, it cannot proceed in another session.)</p> <pre><b>STORE result</b> <b>INTO</b> <b>'/outputpath';</b></pre> <p>(It is used to store results as the desired path in HDFS so that it can be used further.)</p>
--	--	---

***Table 8.5: Store command***

## Relational operations

***Table 8.6*** shows the commands that are used in relational operations:

Command	Description	Example
---------	-------------	---------

Command	Description	Example
<b>FILTER</b>	<p>This is used to select records that users want to retain in the data pipeline.</p> <p>To use these comparators with two tuples, both tuples must have the same schema. If the tuples have different schemas, you can use the <b>MATCHES</b> operator to compare fields by position. The <b>MATCHES</b> operator takes a regular expression and returns true if any field in the tuple matches the expression.</p> <p>For example, the following Pig Latin code uses the <b>MATCHES</b> operator to compare two tuples with different schemas:</p> <pre>A = LOAD 'data1' AS (a1: int, a2: chararray); B = LOAD 'data2' AS (b1: int, b2: chararray); C = FILTER A BY \$0 MATCHES B.\$1;</pre> <p>In this code, we compare the first field of tuple A with the second field of tuple B. The <b>MATCHES</b> operator allows us to compare these fields even though they have different names and types. If the regular expression matches, the <b>FILTER</b> operation returns the entire tuple A.</p>	<pre>filtered_result = FILTER records by projectname=='en'; (It filters the specified records from many sets of records.)</pre>
<b>FOREACH</b>	<p>It processes the records as a pipeline with a set of expressions and generates new records to send to the next operator. It is a projection operator that scans all databases.</p> <p>Tuple projection in Pig is done using the dot (.) operator. This operator can be used to reference a field in a tuple either by name or by position. If you reference a</p>	<pre>top100result = foreach top100 generate \$0, \$1, \$3, \$4, \$5, \$6; (It provides the result as top100 from a data set that name top100join with columns 0,1, 3, 4, 5, and 6.)</pre>

Command	Description	Example
	nonexistent positional field in the tuple, it will return null. Here is an example of tuple projection by name.	<pre>spendresult = foreach txnbycust generate group, SUM(txn.amount);</pre> <p>(It generates results with filtration)</p>
GROUP	This statement groups all the data as a whole and then extracts the necessary information through a filtering process, typically based on a key. This grouping is different from SQL, which uses aggregate functions. PigLatin has no direct connection between the group and aggregate functions, but it collects all into the same bag.	<pre>grouped_result = GROUP filtered_result by pagename;</pre> <p>(It leads to making groups of records with page names)</p>
ORDER BY	The Order statement sorts data based on specified criteria, enabling users to extract data according to their specific requirements. The total order of data not only involves sorting data within each partition but also guarantees the order of all records, ensuring a consistent sequence across the entire dataset. Each partition of data in HDFS is a part file that can be accommodated with a cat command. It is similar to a group statement and depends on the key for ordering purposes. Multiple keys can be used to order data without using parentheses.	<pre>orderresult = order spendbycust by \$1 desc;</pre> <p>(It provides ordered results based on column orientation that is decremented)</p>

Command	Description	Example
	<p><b>DISTINCT</b></p> <p>It is similar to an SQL statement and is required when there is a need to extract no duplicate record.</p> <p>Actually, the <b>DISTINCT</b> keyword in Pig does not always trigger a reduce phase.</p> <p>If the input relation is already sorted and the <b>DISTINCT</b> operation is done on the sorted key, then a reduced phase is not needed. Pig can directly remove duplicates in the map phase using a combiner. This is called map-side distinct and is more efficient than a full reduce phase.</p> <p>However, if the input is not sorted or the <b>DISTINCT</b> operation is done on a non-key field, then Pig needs to shuffle the data and perform a reduced phase to remove duplicates. This is called reduce-side distinct and is less efficient than map-side distinct.</p> <p>So, whether or not a reduced phase is triggered by <b>DISTINCT</b> in Pig depends on the input data and the way the operation is used.</p>	<pre>Result = distinct cust;</pre> <p>(It extracts distinct records from the name of all customers.)</p>

Command	Description	Example
JOIN	<p>Join is a concept similar to SQL statements; it is used to join multiple tables on the basis of keys. It takes the record from one table and combines it with another using some keys, and if these are equal, then all records are joined.</p> <p>A full outer join includes all records from both sides, even those that do not have a match on the other side. In Pig Latin, left, right, and full outer joins are specified using the JOIN keyword, followed by the join type (LEFT OUTER, RIGHT OUTER, or FULL OUTER), followed by the name of the relation to join with, followed by the ON keyword and the join condition.</p>	<pre>top100join = join top100cust by \$0, cust by \$0;  (It describes joining data based on a few columns that are specified in the query.)</pre>
LIMIT	It is used to create limitations on extracted records.	<pre>result = limit topresult 20;  (It extracts only 20 records)</pre>

Command	Description	Example
PARALLEL	<p>The parallel clause in Pig is used to control the number of reducers for a particular operation, not just limited to reduce side parallelism. It can be used with any relational operator, including group, join, order, and so on.</p> <p>Additionally, the number of reducers specified in the parallel clause applies to all subsequent operators in the script until another parallel clause is specified. Therefore, it carries through the script until overridden.</p> <p>It is important to note that setting a high number of reducers may not always result in better performance and can potentially cause resource contention and slow down the job. The optimal number of reducers is typically based on the size of the data being processed and the available resources.</p>	<pre>result = group custrecord by name parallel 10;</pre> <p>(It provides parallel records that are grouped already.)</p>

**Table 8.6:** Relational operations

## Examples

Let us suppose there is data with four columns that are as follows:  
**projectname**, **pagename**, **pagecount**, and **pagesize**.

en google.com 50 100

en yahoo.com 60 100

us google.com 70 100

en google.com 68 100

The desired result is to calculate the page count for each page site:

google.com 118

yahoo.com 60

```
records = LOAD '/webcount' using PigStorage(' ') as  
(projectname:chararray, pagename:chararray,  
pagecount:int, pagesize:int);
```

Records can indicate raw data from HDFS that is stored as pig storage with space delimiter and defined with **projectname**, **pagename** as **chararray** and **pagecount**, **pagesize** as **integer**. It is optional, and the name of the column can be taken as a different name too

```
filtered_records = FILTER records by  
projectname=='en';
```

Now, records can be used further for any other filtration. Here, **filtered\_records** is storing data that filter with project name whatever we were given name:

```
grouped_records = GROUP filtered_records by pagename;
```

Grouped records can combine all the records using any parameter of column, and here it is **pagename**:

```
results = FOREACH grouped_records generate  
group, SUM(filtered_records.pagecount);
```

The **foreach** keyword combines records to generate a group and sum of all **pagecount** for a particular record and indicate it with results. **FOREACH** iterates over each tuple in the **data** relation, and for each tuple, it generates a new tuple with the **name** field:

```
sorted_result = ORDER results by $1 desc;
```

All said records can be arranged in ascending or descending order using any of the keywords based on the column:

```
STORE sorted_result INTO '/YOUROUTPUT';
```

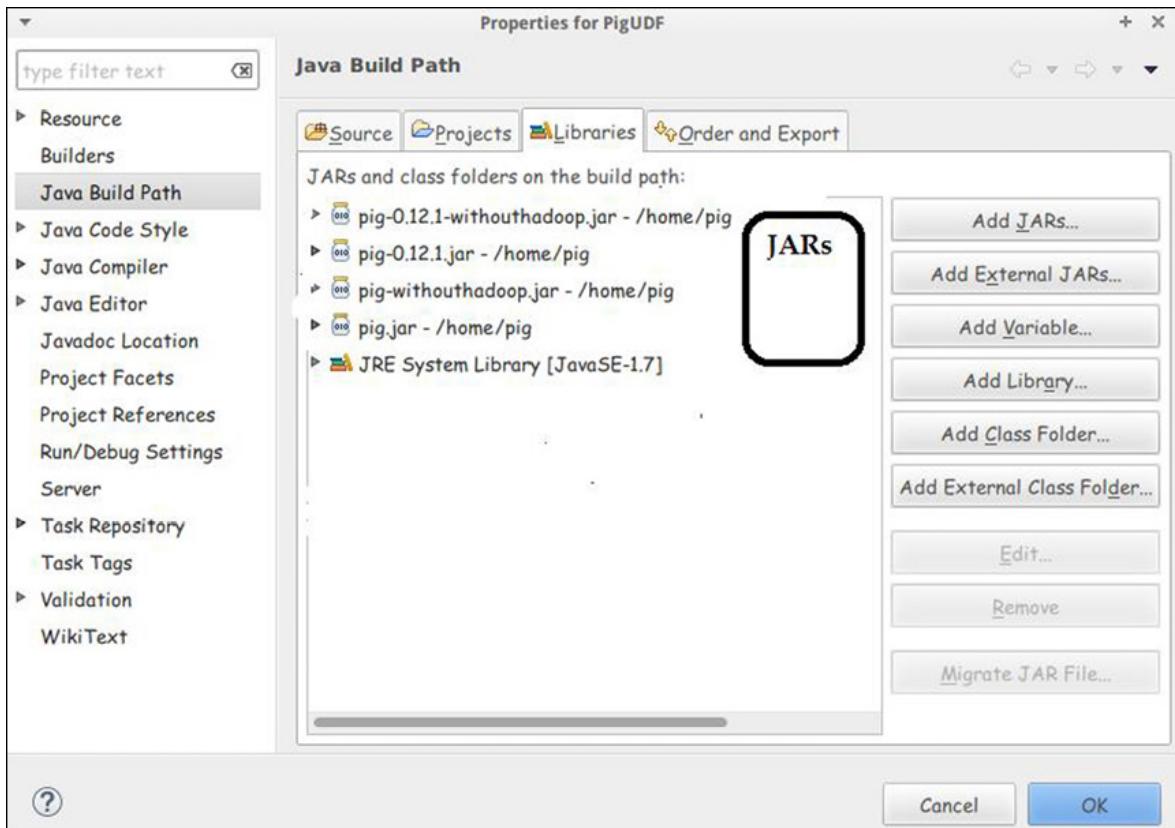
Finally all records can be dumped on the terminal and can be stored at any location in HDFS.

## User-defined functions

**User Defined Functions (UDF)** is the ability to provide all functions at once. PigLatin uses Java codes for performing functionality.

Following is the procedure to run sample UDF that converts data to upper case. It requires a Hadoop environment, configured pig in Hadoop, and JAR under the lib folder in Eclipse. Eclipse is an **Integrated Development Environment (IDE)** used primarily for software development. It provides a robust platform for writing, compiling, debugging, and testing code. Eclipse is known for its extensibility, allowing developers to customize it for various programming languages and development purposes.

Open Eclipse and add JARs, as shown in *Figure 8.12*:



*Figure 8.12: Add JARs files*

As UDFs support Java, there is a need to write **upper.java** file for converting raw data to an upper. The following **upper.java** file will contain code for the required changes:

```
import java.io.IOException;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;
import org.apache.pig.impl.util.WrappedIOException;

public class Upper extends EvalFunc<String> {

    // Override the exec method of the EvalFunc class
    @Override
    public String exec(Tuple input) throws IOException {
        // Check if input is null or empty
        if (input == null || input.size() == 0) {
            return null;
        }

        try {
            // Get the first field of the tuple and convert
            // to uppercase
            String str = (String) input.get(0);
            str = str.toUpperCase();

            // Return the uppercase string
            return str;
        }
    }
}
```

```

    }

    catch (Exception e) {
        // Wrap the exception in a WrappedIOException
        // and throw it

        throw WrappedIOException.wrap("Caught exception
processing input row ", e);
    }

}

}

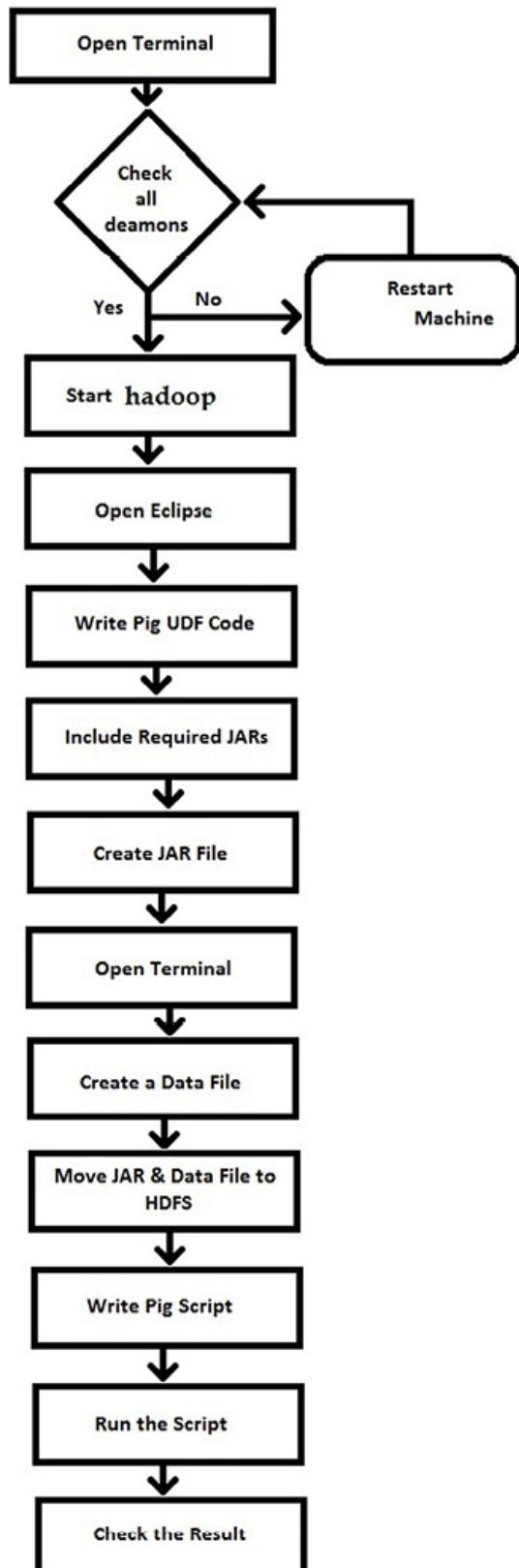
```

The main changes made are as follows:

- Addition of comments to explain the purpose of each section of code.
- Removal of the `@SuppressWarnings("deprecation")` annotation, as it is not necessary for this code.
- Addition of the `@Override` annotation to the exec method to indicate that it is overriding a method from the parent class `EvalFunc`.

After its implementation `upper.java` file needs to be exported as a JAR file that also includes external sources. Using this JAR file, follow all instructions as per MapReduce.

The output will show all data in its upper case as it is required as a task, and its code will be written in a Java file. *Figure 8.13* shows the flow of implementing UDFs. Another example will provide the execution of `eval` function using UDFs:



*Figure 8.13: Flow for implementing UDFs*

While the **EVAL** function itself is not a standalone function in Pig, it represents the portion of the script where data transformations, calculations, or evaluations are applied to columns in the input relation:

```
import java.io.IOException;
import org.apache.pig.EvalFunc;
import org.apache.pig.PigWarning;
import org.apache.pig.data.Tuple;

public class Pow extends EvalFunc<Long> {

    public Long exec(Tuple input) throws IOException {
        try {

            int base = (Integer)input.get(0);
            int exponent = (Integer)input.get(1);
            long result = 1;

            /* Probably not the most efficient method...*/
            for (int i = 0; i < exponent; i++) {
                long preresult = result;
                result *= base;
                if (preresult > result) {

                    // We overflowed. Give a warning, but do
                    not throw an exception.
                }
            }
        }
    }
}
```

```

        warn("Overflow!",
PigWarning.TOO_LARGE_FOR_INT);

        // Returning null will indicate to Pig that
we failed but

        // we want to continue execution.

        return null;

    }

}

return result;

} catch (Exception e) {

        // Throwing an exception will cause the task to
fail.

        throw new IOException("Something bad
happened!", e);

    }

}

```

After implementing the preceding programs, it can run as running MapReduce programs. Output can be checked in HDFS by the same method described previously. Distributed cache can also be implemented using Pig UDFs, as follows:

```

public class Regex extends EvalFunc<Integer> {

    static HashMap<String, String> map = new
HashMap<String, String>();

    public List<String> getCacheFiles() {

```

```
    Path lookup_file = new Path(  
"hdfs://localhost.localdomain:8020/user/mayank/next")  
;  
  
    List<String> list = new ArrayList<String>(1);  
    list.add(lookup_file + "#id_lookup");  
    return list;  
}  
  
public void VectorizeData() throws IOException {  
    FileReader fr = new FileReader("./id_lookup");  
    BufferedReader brd = new BufferedReader(fr);  
    String line;  
    while ((line = brd.readLine()) != null) {  
        String str[] = line.split("#");  
        map.put(str[0], str[1]);  
    }  
    fr.close();  
}  
  
@Override  
public Integer exec(Tuple input) throws IOException  
{  
    // TODO Auto-generated method stub
```

```
    return map.size();  
}  
}
```

## Eval function

**EvalFunc** class is the base for **Eval** functions that extends in UDF in **org.apache.pig.EvalFunc**. It has a return type of the UDF, which is **String**. The core method is exec in this class. The package can be of any name that consists of functions of programs. It is used to take one record and return one result, which is invoked for every record that needs to execute through the pipeline. It takes tuples that pass all its fields to UDF which is defined by the user. After that, it returns the type, which is parameterized **EvalFunc**.

## Aggregate functions

Aggregation functions are mathematical operations that perform a calculation on a set of values and return a single summarized result. They are often used in database queries to summarize data, such as calculating the sum, average, minimum, or maximum value of a set of records.

COUNT, MIN, MAX, AVERAGE, and more are examples of aggregate functions.

Following is the example of an aggregate function: **COUNT**

---

---

```
import java.io.IOException;  
import java.util.Iterator;  
import java.util.Map;  
  
import org.apache.pig.Algebraic;  
import org.apache.pig.EvalFunc;  
import org.apache.pig.data.DataBag;
```

```
import org.apache.pig.data.Tuple;
import org.apache.pig.impl.util.TupleFactory;

public class Count extends EvalFunc<Long> implements
Algebraic {

    public Long exec(Tuple input) throws IOException {
        return count(input);
    }

    public String getInitial() {
        return Initial.class.getName();
    }

    public String getIntermed() {
        return Intermed.class.getName();
    }

    public String getFinal() {
        return Final.class.getName();
    }

    static public class Initial extends EvalFunc<Tuple>
{
    public Tuple exec(Tuple input) throws IOException
{
```

```
        return
TupleFactory.getInstance().newTuple(count(input));

    }

}

static public class Intermed extends
EvalFunc<Tuple> {

    public Tuple exec(Tuple input) throws IOException
{

    return
TupleFactory.getInstance().newTuple(sum(input));

}

}

static public class Final extends EvalFunc<Long> {
    public Tuple exec(Tuple input) throws IOException
{

    return sum(input);

}

}

static protected Long count(Tuple input) throws
IOException {

Object values = input.get(0);
if (values instanceof DataBag) {

    return ((DataBag)values).size();
}
```

```

    } else if (values instanceof Map) {
        return new Long(((Map<?, ?>)values).size());
    } else {
        return 1L;
    }
}

static protected Long sum(Tuple input) throws
IOException {
    DataBag values = (DataBag)input.get(0);
    long sum = 0;
    for (Iterator<Tuple> it = values.iterator();
it.hasNext();) {
        Tuple t = it.next();
        sum += (Long)t.get(0);
    }
    return sum;
}
}

```

COUNT implements an **Algebraic** interface which looks like the following:

```

public interface Algebraic {
    public String getInitial();
    public String getIntermed();
    public String getFinal();
}

```

```
}
```

## Filter function

In Pig Latin, the **FILTER** operation is used to selectively filter rows or records from a relation based on a specified condition. It allows you to process only the data that meets the specified criteria.

In Pig Latin, the **IsEmpty** function is used to check whether a bag or a field is empty (contains no data) and returns a **Boolean** value (true if empty, false otherwise). You can use it as a part of your Pig Latin statements to conditionally process data based on whether a bag or field is empty.

The following example implements the **IsEmpty** function:

```
import java.io.IOException;
import java.util.Map;
import org.apache.pig.FilterFunc;
import
org.apache.pig.backend.executionengine.ExecException;
import org.apache.pig.data.DataBag;
import org.apache.pig.data.DataType;
import org.apache.pig.data.Tuple;

public class IsEmpty extends FilterFunc {

    public Boolean exec(Tuple input) throws IOException
    {
        try {
            Object values = input.get(0);
            if (values instanceof DataBag) {
```

```

        return ((DataBag) values).size() == 0;

    } else if (values instanceof Map) {
        return ((Map<?, ?>) values).size() == 0;
    } else {
        int errCode = 2102;
        String msg = "Cannot test a " +
        DataType.findTypeName(values) + " for emptiness.";
        throw new ExecException(msg, errCode,
PigException.BUG);
    }
} catch (ExecException ee) {
    throw ee;
}
}
}

```

## Developing and testing the PigLatin script

There are some diagnostic commands to verify the statements of PigLatin, which are as follows:

- Dump operator
- Describe operator
- Explanation operator
- Illustration operator

### Dump operator

Dump used to show all data as an output on the screen, as shown in *Figure 8.14*:

```
grunt> dump loading1;
2013-11-15 22:55:36,601 [main] INFO  org.apache.pig.tools.pigstats.ScriptState - Pig features used :
2013-11-15 22:55:36,601 [main] INFO  org.apache.pig.backend.hadoop.executionengine.HExecutionEngine
set to true. New logical plan will be used.
```

*Figure 8.14: Dump operator*

## Describe operator

To review the schema, there is the use of the **DESCRIBE** operator for a particular relation. The **DESCRIBE** operator is best used for debugging a script of PigLatin. The output can be seen in [Figure 8.15](#):

```
grunt> describe loading1;
loading1: {user: chararray,url: chararray,id: int}
grunt> █
```

*Figure 8.15: Describe operator*

## Explanation operator

The **EXPLAIN** operator prints the logical and physical explanation as seen in [Figure 8.16](#):

```

#-----#
# Physical Plan:
#-----
loading1: Store(fakefile:org.apache.pig.builtin.PigStorage) - scope-235
|---loading1: New For Each(false,false,false)[bag] - scope-234
|   |   Cast[chararray] - scope-226
|   |   |---Project[bytarray][0] - scope-225
|   |   Cast[chararray] - scope-229
|   |   |---Project[bytarray][1] - scope-228
|   |   Cast[int] - scope-232
|   |   |---Project[bytarray][2] - scope-231
|---loading1: Load(/first:PigStorage(',')) - scope-224

grunt> explain loading1;
2013-11-16 00:37:59,268 [main] INFO org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - pig.usenewlogicalplan is set to true. New logical plan will be used.
#-----#
# Logical Plan:
#-----
fake: Store 1-432 Schema: {user: chararray,url: chararray,id: int} Type: Unknown
|---loading1: Load 1-402 Schema: {user: chararray,url: chararray,id: int} Type: bag

#-----#
# Physical Plan:
#-----
loading1: Store(fakefile:org.apache.pig.builtin.PigStorage) - scope-235
|---loading1: New For Each(false,false,false)[bag] - scope-234
|   |   Cast[chararray] - scope-226
|   |   |---Project[bytarray][0] - scope-225
|   |   Cast[chararray] - scope-229
|   |   |---Project[bytarray][1] - scope-228
|   |   Cast[int] - scope-232
|   |   |---Project[bytarray][2] - scope-231
|---loading1: Load(/first:PigStorage(',')) - scope-224

```

*Figure 8.16: Explanation operator*

## Illustration operator

It is used for checking the sequence of data execution, that is, the review of data. Mostly, it is used with debugging operations and is widely used and helpful in testing purposes, as shown in *Figure 8.17*:

```

grunt> illustrate loading1;
2013-11-16 00:37:16,037 [main] INFO  org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to hadoop file system at: hdfs://localhost:8020
2013-11-16 00:37:16,037 [main] INFO  org.apache.pig.backend.hadoop.executionengine.HExecutionEngine - Connecting to map-reduce job tracker at: localhost:8021
2013-11-16 00:37:16,093 [main] INFO  org.apache.hadoop.mapreduce.lib.input.FileInputFormat - Total input paths to process : 1
2013-11-16 00:37:16,093 [main] INFO  org.apache.pig.backend.hadoop.executionengine.util.MapRedUtil - Total input paths to process : 1

| loading1 | user: bytearray | url: bytearray | id: bytearray |
|          | amr           | myblog       | 10          |
|-----|
| loading1 | user: chararray | url: chararray | id: int    |
|          | amr           | myblog       | 10          |

grunt> ■

```

*Figure 8.17: Illustrate operator*

## Hive

A hive is an analytic tool that is used to give power to the programmer for analysis with ease. It is like SQL that can perform the required output with a single line of query. It gives flexibility to organizations as it provides all the features that are used in SQL. It can also be called HiveQL, which was introduced by Facebook for the retrieval of faster results on a huge dataset. It also supports scalable data. The following are the features that can describe the existence and usefulness of Hive:

- Hive was created for analysis with SQL-type structures that support Java programming to run queries on a large amount of data.
- It requires data that is stored in HDFS and supports the scalability of data.
- No need for special training to work on Hive.
- Support **Open Data Base Connectivity (ODBC)** for integration with products.
- Useful for handling large amounts of databases with all features of Hadoop.
- Organize data into tables, which is a reflection of structural data that is stored in HDFS. The metadata of such tables, stored in the database, is called **metastore**.
- Hive is not fit for complex machine learning techniques for analysis purposes.

Hive is used to run queries like SQL that actually run MapReduce programs in the background and make life easier for programmers. Hive is convenient for running metastore on a local machine that is already configured at the time of installation of Hive. For installing Hive, there is a need of Java 6 in the system.

## Installing Hive

Installation of the hive can be installed in any relational database. Following are the procedures to install Derby and SQL. Using Derby database.

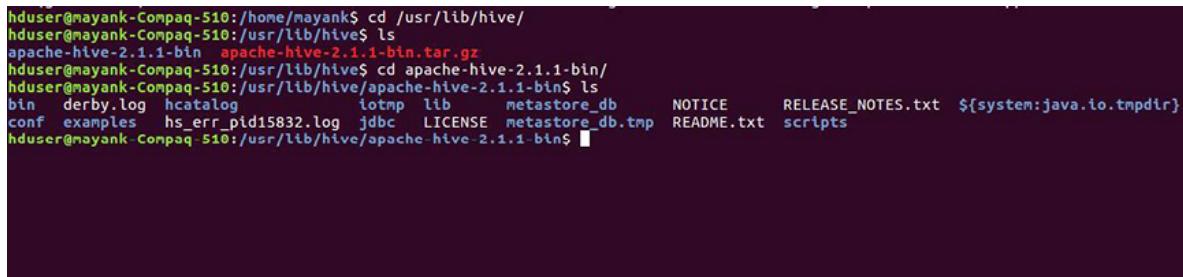
Following is the step-by-step process:

1. Download Hive tar using the following command:

```
 wget http://archive.apache.org/dist/hive/hive-  
2.1.0/apache-hive-2.1.0-bin.tar.gz
```

2. Extract the tar file using the following command:

```
 tar -xzf apache-hive-2.1.0-bin.tar.gz
```



```
hduser@mayank-Compaq-510:/home/nayank$ cd /usr/lib/hive/  
hduser@mayank-Compaq-510:/usr/lib/hive$ ls  
apache-hive-2.1.1-bin apache-hive-2.1.1-bin.tar.gz  
hduser@mayank-Compaq-510:/usr/lib/hive$ cd apache-hive-2.1.1-bin/  
hduser@mayank-Compaq-510:/usr/lib/hive/apache-hive-2.1.1-bin$ ls  
bin derby.log hcatalog iotmp lib metastore_db NOTICE RELEASE_NOTES.txt ${system:java.io.tmpdir}  
conf examples hs_err_pid15832.log jdbc LICENSE metastore_db.tmp README.txt scripts  
hduser@mayank-Compaq-510:/usr/lib/hive/apache-hive-2.1.1-bin$
```

**Figure 8.18:** Hive installation directory

3. Edit the **.bashrc** file to update the environment variables for the user:

```
 sudo gedit .bashrc
```

4. Add the **CLASSPATH** as shown at the end of the file:



```
./.bashrc
export PIG_CONF_DIR=$PIG_HOME/conf
export PIG_CLASSPATH="$PIG_CONF_DIR"
export PATH="$PIG_HOME/bin:$PATH"
#Hbase directory
export HBASE_HOME=/usr/local/hbase
export PATH=$PATH:$HBASE_HOME/bin
# Set HIVE_HOME
export HIVE_HOME=/usr/lib/hive/apache-hive-2.1.1-bin"
export PATH=$PATH:$HIVE_HOME/bin
export DERBY_HOME=/usr/local/derby
export PATH=$PATH:$DERBY_HOME/bin
export CLASSPATH=$CLASSPATH:$DERBY_HOME/lib/derby.jar:$DERBY_HOME/lib/derbytools.jar
```

**Figure 8.19:** Hive installation classpath

Also, make sure that the Hadoop path is set.

5. Run the following command to apply the changes in the same terminal.

```
source .bashrc
```

6. Check the Hive version.

7. Create Hive directories within HDFS. The directory warehouse is the location to store the table or data related to Hive. The commands to be used are as follows:

```
hdfs dfs -mkdir -p /user/hive/warehouse
hdfs dfs -mkdir /tmp
```

8. Set read/write permissions for the table. In this command, we are giving permission to write to the group:

```
hdfs dfs -chmod g+w /user/hive/warehouse
hdfs dfs -chmod g+w /tmp
```

9. Set the Hadoop path in **hive-env.sh**. Use the following command:

```
cd apache-hive-2.1.0-bin/
gedit conf/hive-env.sh
```

```

#!/bin/sh
# Set HADOOP_OPTS environment variable
# If HADOOP_OPTS is not set, export it
if [ -z "$HADOOP_OPTS" ]; then
    export HADOOP_OPTS="$HADOOP_OPTS -XX:NewRatio=12 -Xms10m -XX:MaxHeapFreeRatio=40 -XX:MinHeapFreeRatio=15 -XX:-UseGCOverheadLimit"
fi
# The heap size of the jvm started by hive shell script can be controlled via:
#export HADOOP_HEAPSIZE=1024
# Larger heap size may be required when running queries over large number of files or partitions.
# By default hive shell scripts use a heap size of 256 (MB). Larger heap size would also be
# appropriate for hive server (hwi etc).

# Set HADOOP_HOME to point to a specific hadoop install directory
#HADOOP_HOME=/usr/local/hadoop

# Hive Configuration Directory can be controlled by:
#export HIVE_CONF_DIR=/usr/lib/hive/apache-hive-2.1.0-bin/conf

```

**Figure 8.20:** Hive installation configuration directory

#### 10. Edit `hive-site.xml`:

```
gedit conf/hive-site.xml
```

Copy and replace the following from existing data and make sure of the path. We have mentioned the path which you need to replace by your path:

```

<name>javax.jdo.option.Multithreaded</name>
<value>true</value>
<description>Set this to true if multiple threads access metastore through JDO concurrently.</description>
</property>
<property>
<name>javax.jdo.option.ConnectionURL</name>
<value>jdbc:derby::databaseName=/usr/lib/hive/apache-hive-2.1.1-bin/metastore_db;create=true</value>
<description>
JDBC connect string for a JDBC metastore.
To use SSL to encrypt/authenticate the connection, provide database-specific SSL flag in the
connection URL.
For example, jdbc:postgresql://myhost/db?ssl=true for postgres database.
</description>
</property>
<property>
<name>hive.metastore.dbaccess.ssl.properties</name>
<value/>
<description>
Comma-separated SSL properties for metastore to access database when JDO connection URL
enables SSL access. e.g. javax.net.ssl.trustStore=/tmp/
truststore,javax.net.ssl.trustStorePassword=pwd.
</description>
</property>
<property>
<name>hive.hmsHandler.retry.attempts</name>
<value>10</value>
<description>The number of times to retry a HMSHandler call if there were a connection error.</description>
</property>

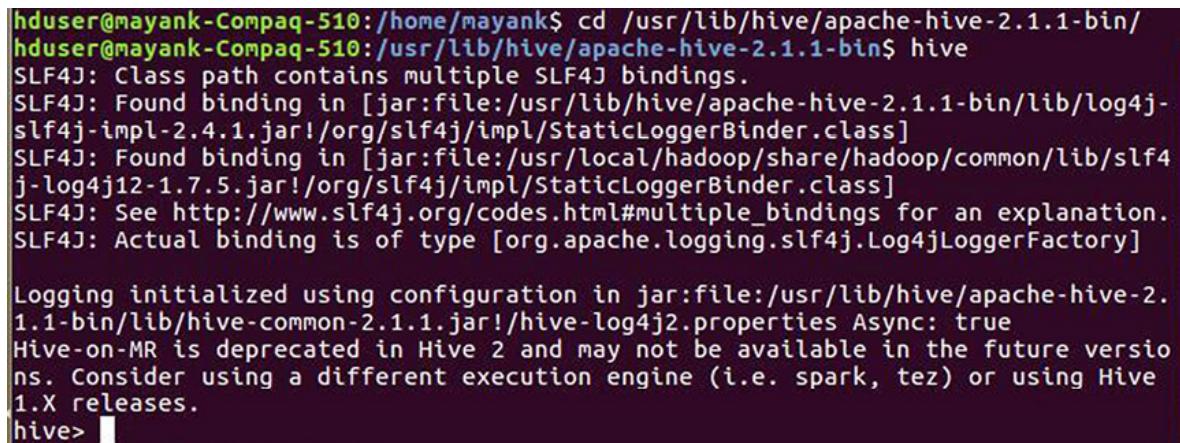
```

**Figure 8.21:** Hive configuration file edit

#### 11. By default, Hive uses the Derby database. Initialize the Derby database using the command:

```
bin/schematool -initSchema -dbType derby
```

12. Launch Hive using the command `hive`:



```
hduser@mayank-Compaq-510:/home/mayank$ cd /usr/lib/hive/apache-hive-2.1.1-bin/
hduser@mayank-Compaq-510:/usr/lib/hive/apache-hive-2.1.1-bin$ hive
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/lib/hive/apache-hive-2.1.1-bin/lib/log4j-slf4j-impl-2.4.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/local/hadoop/share/hadoop/common/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.apache.logging.slf4j.Log4jLoggerFactory]

Logging initialized using configuration in jar:file:/usr/lib/hive/apache-hive-2.1.1-bin/lib/hive-common-2.1.1.jar!/hive-log4j2.properties Async: true
Hive-on-MR is deprecated in Hive 2 and may not be available in the future versions. Consider using a different execution engine (i.e. spark, tez) or using Hive 1.X releases.
hive> ■
```

**Figure 8.22: Hive shell**

13. Run a few queries in the Hive shell.

```
show databases;

create table employee (id string, name string,
dept string) row format delimited fields
terminated by '\t' stored as textfile;

show tables.
```

14. Use `exit` to exit from Hive.

## Using MySQL

Previously, we discussed the installation of hive using a derby database. Now following is the procedure to install it using MySQL:

1. Install MySQL using the command: `$ sudo apt-get install mysql-server`. It will be prompted to set a password for root.
2. Install the MySQL Java Connector using the command: `$ sudo apt-get install libmysql-java`
3. Create a soft link for the connector in the Hive lib directory or copy the connector jar to `lib` folder using the command: `ln -s /usr/share/java/mysql-connector-java.jar $HIVE_HOME/lib/mysqlconnector- java.jar`

**HIVE\_HOME** points to the installed **hive** folder.

4. Create the Initial database schema using the **hive-schema-0.14.0.mysql.sql** file located in the following directory :**\$HIVE\_HOME/scripts/metastore/upgrade/mysql directory**.

```
$ mysql -u root -p
```

Enter password: \*\*\*\*

Password will be asked.

```
mysql> CREATE DATABASE metastore;  
mysql> USE metastore;  
mysql> SOURCE  
$HIVE_HOME/scripts/metastore/upgrade/mysql/hive-  
schema-  
0.14.0.mysql.sql;
```

5. There is a need for the MySQL user account for Hive to access the metastore. It is important to prevent this user account from creating or altering tables in the metastore database schema. Following are the steps to create a user account in the hive.

```
mysql> CREATE USER 'hiveuser'@'%' IDENTIFIED BY  
'hivepassword';  
mysql> GRANT all on *.* to 'hiveuser'@localhost  
identified by 'hivepassword';  
mysql> flush privileges;
```

**hiveuser** is the **ConnectionUserName** in **hive-site.xml**.

6. Create **hive-site.xml** in the **\$HIVE\_HOME/conf** folder with the following configuration:

```
<configuration>  
  <property>  
    <name>javax.jdo.option.ConnectionURL</name>
```

```
    <value>jdbc:mysql://localhost/metastore?  
createDatabaseIfNotExist=true</value>  
    <description>metadata is stored in a MySQL  
server</description>  
  </property>  
  <property>  
  
<name>javax.jdo.option.ConnectionDriverName</name>  
  <value>com.mysql.jdbc.Driver</value>  
  <description>MySQL JDBC driver  
class</description>  
  </property>  
  <property>  
  
<name>javax.jdo.option.ConnectionUserName</name>  
  <value>hiveuser</value>  
  <description>user name for connecting to mysql  
server</description>  
  </property>  
  <property>  
  
<name>javax.jdo.option.ConnectionPassword</name>  
  <value>hivepassword</value>  
  <description>password for connecting to mysql  
server</description>  
  </property>  
  <property>
```

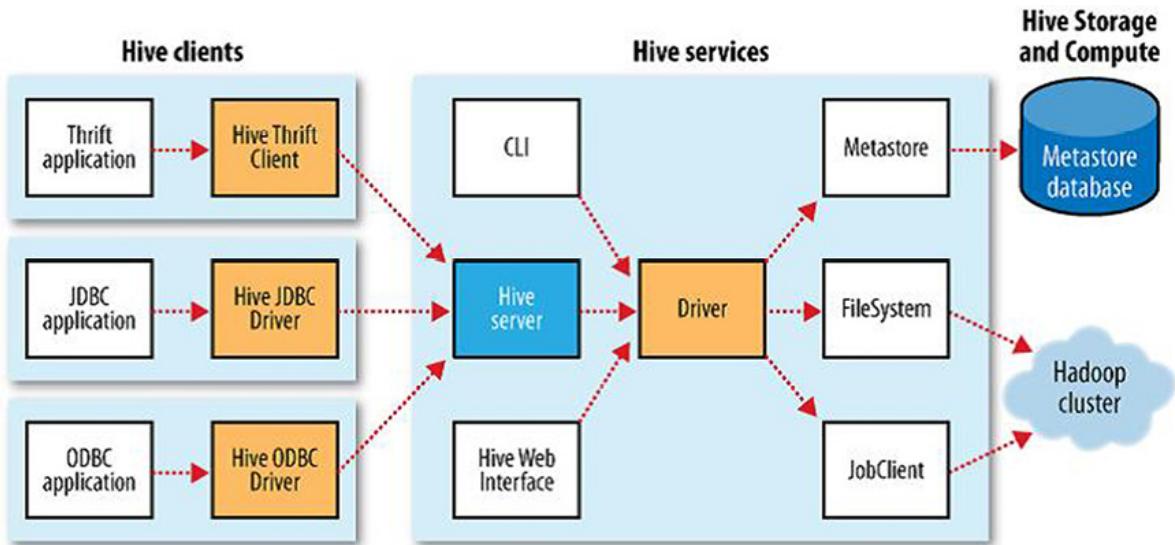
```
<name>datanucleus.autoCreateSchema</name>
<value>false</value>
<description>Creates necessary schema on a
startup if one doesn't exist
</description>
</property>
</configuration>
```

7. Start the hive console now.

## Hive architecture

The structures depicted in [Figure 8.23](#) are used to establish the architecture of Hive. Hive architecture contains the following parameters:

- Hive **SET** command
- command line **-hiveconf** option
- **hive-site.xml**
- **hive-default.xml**
- **hadoop-site.xml**
- **hadoop-default.xml**



**Figure 8.23: Hive architecture**

## Hive services

With hive shell, there are some services that can be started. These can be accessed using **hive -service help**. These services are discussed as follows:

- **clr:** This is a command line interface for default services.
- **Hiveserver:** It runs hive as a server with a thrift service that enables clients from many servers with different languages. Applications using Thrift, JDBC, and ODBC connectors need to run the hive server to communicate with the hive.
- **hwi:** It is an alternative to hive shell. It is also possible to run Web interfaces as shared services.
- **Jar:** Hive is equal to the Jar that runs Java applications, which includes Hadoop and hive classes on the classpath.
- **Metastore:** It runs in the same process as hive services. Using this, it is possible to run metastore as a remote process, and it also sets the **METASTORE\_PORT** environment variable to specify the port. Metastore is the repository of hive metadata that is divided into two parts: a service and a backlog of data. Metastore builds with Java on the same server-embedded derby database. This embedded database is used to access the database files on disk at any time, but only one session is allowed to open to make a conversation. For multiple session support, it is referred to as a

local metastore. A hive service is configured to use this local server by setting **hive.metastore.local** to **false**.

## Data type and file format

In Apache Hive, data types and formats are important concepts when working with data. Here is an overview of Hive data types and file formats. Hive supports a wide range of data types, both primitive and complex, similar to those found in SQL databases. Some common Hive data types include the following:

- **Primitive data types:**
  - **INT:** Integer data type.
  - **BIGINT:** A 64-bit integer.
  - **FLOAT:** Single-precision floating-point number.
  - **DOUBLE:** Double-precision floating-point number.
  - **STRING:** Variable-length character string.
  - **BOOLEAN:** Boolean (true/false) value.
  - **TIMESTAMP:** Timestamp with date and time information.
  - **DATE:** Date data type.
  - **CHAR:** Fixed-length character string.
  - **VARCHAR:** Variable-length character string.
- **Complex data types:**
  - **ARRAY:** An ordered collection of elements, all of the same type.
  - **MAP:** A collection of key-value pairs.
  - **STRUCT:** A complex data structure containing named fields with different data types.
  - **UNION:** Represents multiple possible data types for a single column.

In the following table, Hive datatypes are stated:

Category	Type

Primitive	TINYINT (1 byte)
	SMALLINT (2 byte)
	INT (4 byte)
	BIGINT (8 byte)
	FLOAT (4 byte)
	DOUBLE (8 byte)
	BOOLEAN (true/false)
	STRING (character string)
	BINARY (byte array)
	TIMESTAMP (timestamp with nanosecond)
Complex	ARRAY (ordered collection of field)
	MAP (unordered collection of key-value)
	STRUCT (collection of named fields)

**Table 8.7:** Hive datatypes

Hive is required to manage large databases with ease. With file format, it can predict the information stored in the computer and the kind of data that can be fetched. These formats vary with respect to data encoding, compression, usage of space, and disk I/O. The following file formats support hive in its execution:

- **Text file:** This is the most used format in Hadoop, and it is also frequently used in Hive with default case. From text data, it can be loaded from **Comma Separated Values (CSV)** delimited by tabs, commas, spaces, and JSON data.

Here is the command by which we can create the table in hive:

```
CREATE TABLE table_name (
    column1 INT,
    column2 STRING,
```

```
    column3 DOUBLE  
)  
ROW FORMAT DELIMITED  
FIELDS TERMINATED BY ','  
STORED AS TEXTFILE;
```

Text file I/O format present in the package as follows:

```
org.apache.hadoop.mapred.TextInputFormat  
org.apache.hadoop.mapred.TextOutputFormat
```

#### **Creating text file:**

Using the preceding command following are the steps to create a table and upload data on it:

```
CREATE TABLE study_text (  
    subject STRING,  
    rollno INT,  
    batch STRING,  
    year STRING,  
    admission STRING,  
    address STRING,  
    marks INT  
)  
ROW FORMAT DELIMITED  
FIELDS TERMINATED BY '\t'  
STORED AS TEXTFILE;
```

The preceding example explains how to create a table name study, which is stored as a text file in the Hive database. Data can be easily loaded using:

```
LOAD DATA LOCAL INPATH 'path_of_your_file' INTO  
TABLE study_text;
```

Data can be transferred into the study table, which is already in the database that is stored as a text file.

- **Sequence file:** Hadoop performance decreases while working with a small number of large files compared to files with a large amount of data. This will increase the size of metadata that acts as an overhead to **NameNode**. Sequence files act as containers to store small files. These are flat files that consist of key–value pairs. At the time of conversion of queries to MapReduce, appropriate key–value pairs are used as records in Hive processing. Sequence files are in binary format that can be split and can be used to club two or more smaller files. Sequence files can be categorized as follows:
  - Uncompressed key/value records
  - Record compressed key/value records
  - Block compressed key/value records

Following is the format by which we can create the table in a sequence file:

```
CREATE TABLE table_name (  
    column1 INT,  
    column2 STRING,  
    column3 DOUBLE  
)  
ROW FORMAT DELIMITED  
FIELDS TERMINATED BY ','  
STORED AS SEQUENCEFILE;
```

Sequence file I/O format present in package as follows:

```
org.apache.hadoop.mapred.SequenceFileInputFormat  
org.apache.hadoop.hive.ql.io.HiveSequenceFileOutputFormat
```

### **Creating sequence file**

The following example will show creating the table in sequence file and uploading data into it:

```
CREATE TABLE study_sequencefile (  
    subject STRING,  
    rollno INT,  
    batch STRING,  
    year STRING,  
    admission STRING,  
    address STRING,  
    marks INT  
)  
ROW FORMAT DELIMITED  
FIELDS TERMINATED BY '\t'  
STORED AS SEQUENCEFILE;
```

The preceding query can create the table in which data can be loaded using Hive:

```
INSERT OVERWRITE TABLE study_sequencefile  
SELECT * FROM study_sequencefile;
```

- **RC file:** It stands for **Record Columnar (RC)** file, which is also a binary file format that promotes a high compression rate on rows. It is mostly

used when there is a need to perform multiple rows at a time.

It has many similarities to Sequence files, and it also behaves like flat files that consist of binary key-value pairs. It is used to save columns as records. It processes records by first splitting rows horizontally and then partitioning each row vertically in a columnar way. RC file metadata split rows with key part and its data as value part. This is most effective when it is used for analysis purposes, so Hive can use it better than other techniques.

The following example will show the creation of a table in hive and use it as an RCfile:

```
CREATE TABLE table_name (
    column1 INT,
    column2 STRING,
    column3 DOUBLE
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS RCFIELD;
```

RC file I/O format present in the package as follows:

```
org.apache.hadoop.hive.ql.io.RCFileInputFormat
org.apache.hadoop.hive.ql.io.RCFileOutputFormat
```

### **Creating RC file:**

Let us create a table name **study\_rcfile** and store data in it:

```
CREATE TABLE study_rcfile (
    subject STRING,
    rollno INT,
```

```

batch STRING,
year STRING,
admission STRING,
address STRING,
marks INT
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
STORED AS RCFILE;

```

After creating a table, data cannot be transferred directly into the file. First, it needs to load into the table, and that table needs to overwrite as follows:

```

INSERT OVERWRITE TABLE study_rcfile
SELECT * FROM study_rcfile;

```

It means, like in the sequence files, in the RC file, there is a need to load the data to a table that is already built with a text file, after which overwrite can occur.

- **ORC file:** ORC stands for **Optimized Row Columnar**, which stores data in an optimized manner as per its name, unlike another format. Being an optimization, it reduces the size of the original data by up to 75%. This is caused by better performance than text, sequence and RC file formats.

Following is the syntax to create a table with an ORC file:

```

org. CREATE TABLE table_name (
    column1 INT,
    column2 STRING,
    column3 DOUBLE

```

```
)  
ROW FORMAT DELIMITED  
FIELDS TERMINATED BY ','  
STORED AS ORC;
```

ORC file I/O format present in package as follows:

```
apache.hadoop.hive.io.orc
```

### **Creating ORC file:**

Following is the example to use with hive:

```
create table study_rcfile (subject STRING, rollno  
INT, batch STRING, year STRING, admission STRING,  
address STRING, marks INT) row format delimited  
fields terminated by '\t' stored as orcfile;
```

Again, it cannot be filled directly, but it can be overwritten. **INSERT OVERWRITE** as it will permanently replace the data in the target table, and the original data will be lost:

```
INSERT OVERWRITE TABLE study_orcfile  
SELECT * FROM study_orcfile;
```

This is the structural representation of files. All files discussed have different scenarios for using this as discussed as follows:

- **TEXT** file format data is used when data is delimited by a parameter.
- If data is of a small size, less than the block itself, then **SEQUENCE** file format is used.
- For efficiency purposes, there is analytics used on data, and then RC file format used.
- For better efficiency and optimum results with respect to space and time, there is the use of ORC file format.

## **Comparison of HiveQL with traditional database**

Comparison between both the databases can be observed with respect to two major points, which are discussed in the upcoming sections.

## Schema on read versus write

At the time of reading and writing data for traditional and hive databases following points can reflect its differences:

- In a traditional database, the schema of the table needs to be built at a loading time; if the schema is not defined, then data is rejected.
- Traditional data design is called **schema on write** since data is checked at the time of writing it.
- Hive does not verify data at the time of loading, but it checks schema at the time of query, which is called **schema on read**.
- **Schema on read** effective for initial load is parsed and serialized to disk.
- **Schema on write** makes the performance faster with respect to index columns and compression of data.
- Hive data is also useful when schema is not known, so no indexing can be done.

## Update, transactions and indexes

In Hive, which is a data warehousing and SQL-like querying tool built on top of Hadoop, the concepts of update operations, transactions, and indexes are somewhat different from traditional relational databases. Here is an overview of these concepts in Hive:

- **Update operations:** Hive predominantly follows a schema-on-read approach, which means that data is typically ingested into Hive tables, but updates to existing records are less common. Hive does not support direct updates to individual records like traditional relational databases. Instead, updates are typically done through a multi-step process:
  - **Insert-overwrite:** You can use the **INSERT OVERWRITE** statement to overwrite the contents of an entire Hive table with new data. However, this is usually used for batch operations, not individual record updates.
  - **Historical data:** To maintain historical data, it is common to append new data into Hive tables and use partitioning or versioning to

distinguish different versions of records.

- Transactions: Hive, traditionally, did not support **Atomicity, Consistency, Isolation, and Durability (ACID)** transactions like many traditional databases. Each query in Hive was treated as a single transaction. However, recent versions of Hive (Hive ACID) introduced support for ACID transactions. With this feature, you can perform traditional transactions like inserts, updates, deletes, and rollbacks.

To use Hive ACID transactions, you need to enable it and define tables with transactional properties [for example, **STORED AS ORC** with **tblproperties ('transactional'='true')**]. This allows you to perform ACID-compliant operations.

- **Indexes:** Hive does support indexes to improve query performance. However, these indexes work differently than traditional relational database indexes:
  - **Bitmap indexes:** Hive supports bitmap indexes. These indexes map values to their corresponding rows. Bitmap indexes can improve query performance for certain types of queries.
  - **B-tree indexes:** Hive also supports B-tree indexes, primarily for timestamp columns. B-tree indexes are less commonly used compared to bitmap indexes.

It is important to note that Hive indexes are optional and must be explicitly defined on columns. Also, their use and effectiveness depend on the query patterns.

Hive offers different ways to handle updates, including insert-overwrite for batch operations and ACID transactions for more complex transactional use cases. Hive supports both bitmap and B-tree indexes, but their use and performance impact should be carefully considered based on your specific use case and query patterns.

## HiveQL

This section will be dedicated to performing data definition and data manipulation language with HiveQL.

## Data definition language

**Data Definition Language (DDL)** is the part of HiveQL which is used for creating, altering and dropping databases:

- **Creating/alter databases:** Hive maintains the database for creating tables. This is considered as a catalogue or namespace of tables. This is effective for avoiding collisions. If a user does not specify a database, then it is considered as a default. The **Database** directory is created under **hive.metastore.warehouse.dir**, which can be configured in either mode of Hadoop.

```
hive> CREATE DATABASE college;
```

This is the simplest one to create a database, which is needed at the time of starting of play for queries.

```
hive> CREATE DATABASE IF NOT EXISTS college;
```

This is useful when the user wants to be warned if the database of the same name already exists. **SCHEMA** keyword can also be used instead of **DATABASE**.

```
hive> SHOW DATABASES;
```

```
default
```

```
college
```

```
hive> CREATE DATABASE department;
```

```
hive> SHOW DATABASES;
```

```
default
```

```
college
```

```
department
```

It shows database details on the screen; default is also one of them which cannot be deleted:

```
hive> SHOW DATABASES LIKE 'd.*';
```

```
department
```

```
hive> ...
```

It can also search with the **LIKE** keyword:

```
hive> CREATE DATABASE college
```

```
> LOCATION '/mayank/disk/directory';
```

The default location of storage can be edited as shown preceding:

```
hive> CREATE DATABASE college
```

```
> COMMENT 'Define branch ';
```

```
hive> DESCRIBE DATABASE college;
```

```
college Define branch
```

```
    hdfs://master-
server/user/hive/warehouse/college.db
```

Descriptive comments can be added through previously mentioned query.

**DESCRIBE DATABASE** also shows the location for the database:

```
hive> CREATE DATABASE college
```

```
> WITH DBPROPERTIES ('creator' = 'Mayank
Bhushan', 'date' = '2017-12-01');
```

```
hive> DESCRIBE DATABASE college;
```

```
college hdfs://master-
server/user/hive/warehouse/college.db
```

```
hive> DESCRIBE DATABASE EXTENDED college;
```

```
college    hdfs://master-
server/user/hive/warehouse/college.db
          {date=2017-12-01, creator=Mayank Bhushan};
```

**DESCRIBE DATABASE** can provide master-server indication with port number. **Key-value** property can be added to the database with **DESCRIBE DATABASE EXTENDED <database>** keyword:

```
hive> USE financials;
```

**USE** keyword to specify which database is being used currently. There is no concept of nesting of database. There is also no command to show which database is using currently, so it is safe to use the **USE** command.

```
hive> show tables;
```

It provides all the information about tables in the current database:

```
hive> DROP DATABASE IF EXISTS college;
```

```
hive> DROP DATABASE IF EXISTS college CASCADE;
```

**IF EXIST** is optional, the preceding command drops a database. By default, Hive will not permit you to drop a database if it contains tables. So, either drop the tables first or append the **CASCADE** keyword to the command, which will cause the Hive to drop the tables in the database first.

```
hive> ALTER DATABASE college SET DBPROPERTIES
('edited-by' = 'Nitin');
```

Using the **DBPROPERTIES** database can be altered. There is no method to delete or unset **DBPROPERTY**.

- **Creating tables:** Tables are the main assets for any database, as in hive. There are two kinds of tables in hive: internal and external. The internal table can be said to be a managed table. Both of table has its own advantages.

In the following example, the branch is the name of the table which are to be created:

```
hive> CREATE TABLE IF NOT EXISTS branch ( sid  
int, name String, branch String, section String)  
COMMENT 'Student details'  
ROW FORMAT DELIMITED  
FIELDS TERMINATED BY '\t'  
LINES TERMINATED BY '\n'  
STORED AS TEXTFILE;
```

Although creating tables is the same as SQL conventions, Hive is already having expanding flexibility with its format. The preceding query defines the name of the table with its attributes and types:

```
CREATE TABLE IF NOT EXISTS mydb.college (  
    name          STRING COMMENT 'Student name',  
    marks         FLOAT  COMMENT 'Student marks',  
    subordinates ARRAY<STRING> COMMENT 'Names of  
    subordinates',  
    deductions   MAP<STRING, FLOAT>  
                  COMMENT 'Keys are deductions  
    names, values are percentages',  
    address       STRUCT<street:STRING, city:STRING,  
    state:STRING, zip:INT>  
                  COMMENT 'Permananet address')  
COMMENT 'College description'  
TBLPROPERTIES ('creator'='mayank',  
'created_at'='2017-12-01 10:07:00', ...)  
LOCATION '/usr/hive/warehouse/mydb.db/college';
```

If the target database is not currently in use, you should use the prefix **mydb** for its name. Note that, Hive will not warn you if the specified schema differs from the existing one. If you want to create a new schema, you must first delete the old table. Additionally, comments can be added to any column:

```
CREATE TABLE IF NOT EXISTS mydb.college2  
LIKE mydb.employees;
```

Copy of the previous schema can be created, but not data from the preceding example. It means all schema of database **mydb.college2** will be created, but not its contents, which were loaded.

All of the preceding pertains to managed or internal tables. However, there may be cases where data is of utmost importance, such that if a table is dropped, the data should remain protected. For instance, if an external table named **sample\_test** is created in Hive using HiveQL and linked to a file named **flat\_file.txt**, then deleting **sample\_test** from Hive will not delete **flat\_file.txt** from HDFS.

```
CREATE EXTERNAL TABLE IF NOT EXISTS school (  
    location STRING,  
    state STRING,  
    country STRING,  
    fees FLOAT  
)  
ROW FORMAT DELIMITED  
FIELDS TERMINATED BY ','  
LOCATION '/data/school';
```

The keyword **EXTERNAL** indicates to Hive that the table has an **external** property, and its location is on disk. Being an external table, deleting it

will only remove the metadata of the table but not the data itself, as it resides outside the control of Hive.

Create an external table named **school2** in the database **mydb**, with the same schema as the external table **school** in the same database. The **LOCATION** parameter specifies the path to the data files for the new table, which must be located at **/path/to/data**. Since it is an **external** table, deleting it will not delete the data; only the metadata of the table will be deleted.

```
CREATE EXTERNAL TABLE IF NOT EXIST mydb.school2
LIKE mydb.school
LOCATION '/path/to/data'
```

Like the internal table, schema can be copied from an existing table.

Partitioning of tables is a concept that involves distributing the load horizontally by physically moving the data closer to frequent users. Partitioning data is necessary when there is a need to separate the data into smaller, more manageable parts, such as by country and state.

```
CREATE TABLE organization (
    name      STRING,
    profit     FLOAT,
    subordinates ARRAY<STRING>,
    deductions MAP<STRING, FLOAT>,
    address    STRUCT<street:STRING, city:STRING,
                state:STRING, zip:INT>
)
PARTITIONED BY (country STRING, state STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
```

Partitioning of tables changes the hive structure of data storage. Country and state are the partition keys which behave like regular columns. Hive is by default used to put into a strict mode which prohibits queries of partitioned tables without **where** clause that is responsible for the filter. Restriction mode can be changed by the following method:

```
set hive.exec.dynamic.partition.mode=nonstrict;  
set hive.exec.dynamic.partition=true;  
set hive.enforce.bucketing=true;
```

```
create table office (IDno INT, date STRING,  
officerno INT, salary DOUBLE,  
project STRING, city STRING, state STRING)  
partitioned by (team STRING)  
clustered by (state) INTO 10 buckets  
row format delimited  
fields terminated by ','  
stored as textfile;
```

This is for creating a partitioned table.

```
from officerecord ofc INSERT OVERWRITE TABLE  
office PARTITION(team)  
  
select ofc.IDno, ofc.date, ofc.salary,  
ofc.project, ofc.city, ofc.state, DISTRIBUTE BY  
team;
```

This is for loading data into a partitioned table.

```
CREATE TABLE student (  
    name          STRING,
```

```

rollno      FLOAT,
address      STRUCTURE <street:STRING,
city:STRING, state:STRING, zip:INT>
)

ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\001'
COLLECTION ITEMS TERMINATED BY '\002'
MAP KEYS TERMINATED BY '\003'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE;

```

This is stored as a text file that is encoded using alphanumeric characters. It treats each line as a separate record. Other file formats can be replaced instead of text files.

- **Dropping table:** Dropping tables is similar to SQL. For internal tables, metadata and its data are deleted, while this is not the case in external table dropping:

```
Drop table IF EXISTS College;
```

This will delete the table with all its data and metadata.

- **Alter table:** Most tables require alteration, including renaming, adding, dropping, and modifying. Which can be done in this database as follows:

```
ALTER TABLE college RENAME TO school;
```

This will change the name of the table from **college** to **school**:

```
ALTER TABLE college ADD IF NOT EXISTS
PARTITION (year = 2017, month = 12, day = 1
LOCATION '/logs/2017/12/01'
```

```
PARTITION (year = 2017, month = 12, day = 2
LOCATION '/logs/2017/12/02'

PARTITION (year = 2017, month = 12, day = 3
LOCATION '/logs/2017/12/03'

...;
```

From the previously mentioned query, multiple partitions can be added:

```
ALTER TABLE college

CHANGE COLUMN bsc branch_section_year INT

COMMENT 'Branch, section and year is part of
academic'

AFTER school;
```

**Column** keyword is optional; this column will move after **school**:

```
ALTER TABLE school ADD COLUMNS (
Section_name    STRING COMMENT 'school name',
    school_id LONG    COMMENT 'The current school
is');
```

**Comment** clause is optional, and alter column can change the statement with its position.

```
ALTER TABLE school

CLUSTERED BY (location, state)

SORTED BY (location)

INTO 70 BUCKETS;
```

**Sorted By** clause is optional, other parameters can edit its value.

## Data manipulation language

Delete, Update, Insert, Merge, and more are the **Data Manipulation Language (DML)** statements that can be used in HiveQL as SQL language. The following table contains all DML commands used in Hive:

Command	Statements
INSERT	<pre> INSERT OVERWRITE TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...) [IF NOT EXISTS]] select_statement1 FROM from_statement;  INSERT INTO TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...)] select_statement1 FROM from_statement;  Hive extension (multiple inserts):  FROM from_statement  INSERT OVERWRITE TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...) [IF NOT EXISTS]] select_statement1  [INSERT OVERWRITE TABLE tablename2 [PARTITION ... [IF NOT EXISTS]] select_statement2]  [INSERT INTO TABLE tablename2 [PARTITION ...] select_statement2] ...  FROM from_statement  INSERT INTO TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...)] select_statement1  [INSERT INTO TABLE tablename2 [PARTITION ...] select_statement2]  [INSERT OVERWRITE TABLE tablename2 [PARTITION ... [IF NOT EXISTS]] select_statement2] ...  Hive extension (dynamic partition inserts):  INSERT OVERWRITE TABLE tablename PARTITION (partcol1[=val1], partcol2[=val2] ...) select_statement FROM from_statement; </pre>

	INSERT INTO TABLE tablename PARTITION (partcol1[=val1], partcol2[=val2] ...) select_statement FROM from_statement;
DELETE	DELETE FROM tablename [WHERE expression]
MERGE	MERGE INTO <target table> AS T USING <source expression/table> AS S  ON <boolean expression1>  WHEN MATCHED [AND <boolean expression2>] THEN UPDATE SET <set clause list>  WHEN MATCHED [AND <boolean expression3>] THEN DELETE  WHEN NOT MATCHED [AND <boolean expression4>] THEN INSERT VALUES<value list>
UPDATE	UPDATE tablename SET column = value [, column = value ...] [WHERE expression]

**Table 8.8:** Hive DML commands

### Example for practice

Following are the practice sets for hive queries. These are the steps for any dummy data, which consists of four-column customer data and seven-column transaction data.

#### Example 1:

##### 1. Create database

```
create database retail;
retail is the name of database.
```

##### 2. Select database

```
use retail;
```

##### 3. Create table for storing transactional records

```
create table records (txnno INT, txndate STRING,
custno INT, amount DOUBLE,
category STRING, product STRING, city STRING,
state STRING, spend STRING)
row format delimited
fields terminated by ','
stored as textfile;
```

#### 4. Load the data into the table

```
LOAD DATA LOCAL INPATH 'txns1.txt' OVERWRITE INTO
TABLE records;
```

#### 5. Describing metadata or schema of the table

```
describe records;
```

#### 6. Counting no of records

```
select count (*) from records;
```

#### 7. Counting total spending by category of products

```
select category, sum(amount) from records group by
category;
```

#### 8. 10 customers

```
select custno, sum(amount) from records group by
custno limit 10;
```

#### Example 2:

Following are two data which are **engineer.txt** and other is **email.txt**, join operation needs to be performed:

```
// engineer.txt
```

```
John, 550000, Mumbai
```

```
George, 400000, Newyork
```

Kaleny,300000,Washington

Mario,250000,Tokyo

// email.txt

John,john@gmail.com

George,george@yahoo.co.in

Kaleny,kaleny@yahoo.com

Mario,mario@gmail.com

1.create table engineer(name string, salary float,  
city string)  
row format delimited  
fields terminated by ',';

2.load data local inpath 'engineer.txt' into table  
engineer;

3.select \* from engineer where name='John';

4.create table email (name string, email string)  
row format delimited  
fields terminated by ',';

5.load data local inpath 'email.txt' into table  
email;

6.select a.name, a.city, a.salary, b.email from  
engineer a join email b on a.name = b.name;

7.select a.name,a.city,a.salary,b.email from  
engineer a left outer join email b on a.name =  
b.name;

8.select a.name, a.city, a.salary, b.email from

```
engineer a right outer join email b on a.name =
b.name;

9. select a.name, a.city, a.salary, b.email from
engineer a full outer join email b on a.name =
b.name;
```

**Example 3:**

```
1.create table person (pno string, firstname string,
lastname string, age int, profession string)
row format delimited
fields terminated by ',';
load data local inpath
'/home/mayank/hive/practicals' into table person;

2.create table result1 (pno int, firstname string,
age int, profession string, amount double, product
string)
row format delimited
fields terminated by ',';

3.insert overwrite table result

4.select a.pno, a.firstname, a.age, a.profession,
b.amount, b.product
from person a JOIN result1 b ON a.pno = b.pno;

5.select * from result1 limit 100;

6.create table result2 (pno int, firstname string,
age int, profession string, amount double, product
string, level string)
row format delimited
fields terminated by ',';
```

```

insert overwrite table result2
select * , case
when age<30 then 'low'
when age>=30 and age < 50 then 'middle'
when age>=50 then 'old'
else 'others'
end
from result1;

select * from result2 limit 100;
create table result3 (level string, amount double)
row format delimited
fields terminated by ',';
7.insert overwrite table result3
8.select level, sum(amount) from result2 group by
level;

```

## Conclusion

Pig and Hive are both powerful tools in the Hadoop ecosystem that enable data processing and analysis. They each have their strengths and use cases, making them valuable tools for big data analytics and processing. Both Pig and Hive have their place in the big data landscape and can be used together to handle various data processing and analysis challenges in a Hadoop ecosystem. The choice often depends on your familiarity with the tools, your specific use case, and the requirements of your data processing tasks.

In the upcoming chapter, we can now see real practical-based examples, which were developed to provide scenarios with the use of all technologies used in Hadoop.

## **Join our book's Discord space**

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

[https://discord\(bpbonline.com](https://discord(bpbonline.com)



# CHAPTER 9

## Practical and Research-based Topics

### Introduction

This chapter explains research and practical-oriented topics that are useful for project purposes. Several tools that can be used in Hadoop will be introduced in this chapter. We will explore the dynamic realms of data analysis through Twitter, delve into the efficiency of Bloom filters within MapReduce, harness the power of Amazon Web Services, unlock historical insights with archived documents from the NY Times, navigate the intricacies of data mining on mobile devices.

### Structure

This chapter includes the following topics:

- Data analysis with Twitter
- Use of Bloom filter in MapReduce
- Amazon Web Services
- Document archived from NY Times
- Data mining in mobiles
- Hadoop diagnosis

### Objectives

Studying a research-based topic in Big Data offers several compelling objectives for both researchers and the broader field of data science. First, delving into research on Big Data enables us to uncover innovative techniques and methodologies for handling and analyzing vast and complex datasets. By advancing our understanding of Big Data, researchers can develop more efficient

algorithms, improved data management strategies, and novel data visualization techniques. These advancements are crucial for addressing the ever-growing challenges posed by the rapid expansion of data in various domains, from healthcare and finance to social media and beyond. Research in Big Data contributes to the development of real-world solutions and applications. By conducting research in this field, scholars can identify practical applications of Big Data analytics in solving complex problems, enhancing decision-making processes, and driving business intelligence. This chapter will give examples to use analysis tools for innovation and progress, offering the potential to reshape how we understand and use data in our rapidly evolving digital world.

## **Data analysis with X**

For data analysis, there is a requirement to have bulk data. In the present scenario, there are many sources available for analysis purposes that can make data more sensitive. Twitter is the most famous and useful source because it is one of the main news sources nowadays. Extracting data through X can be performed by two methods: using flume and MapReduce.

Let us discuss these methods in detail.

### **Using flume**

Flume is a tool by which users can extract fixed amounts of data from Twitter using pre-build functions of flume. This tool needs to be configured with a certain amount of information. This is a reliable service that is effective for collecting data and moving it to HDFS smoothly. Flume is useful for the following reasons:

- It streams data from many sources and takes it for analysis purposes.
- It collects large volumes of data in real-time and stores it directly in HDFS, which can be filtered later.
- It also prevents data loss as it acts as a buffer of data when the data rate exceeds from which data is written.
- It guarantees on-time data delivery.
- It reflects some features of Hadoop since it supports horizontal scaling of data, which can handle it additionally.

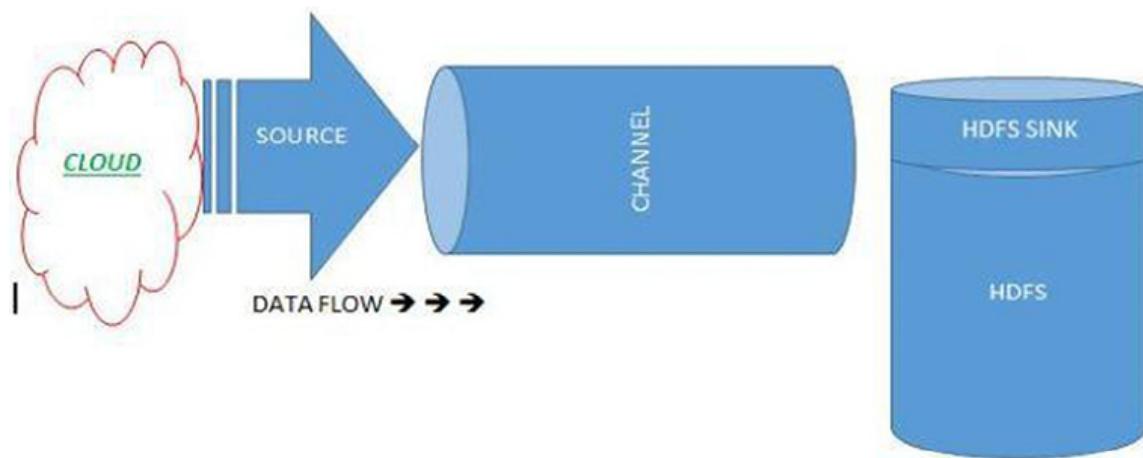
Flume supports major components, which are as follows:

- **Event:** It is a single unit of data that is transported by flume.

- **Source:** A variety of data sources, for example, **log4j** logs and **syslogs** allow data to be collected. This source gives entry of data through entity. External source sends data to flume in a format that should be recognized at the source end, like Avro data from Avro clients.
- **Sink:** It is used to deliver data to the destination with various sinks that stream data like HDFS.
- **Channel:** It is like a bridge between the source and the sink. The source ingests the event into the channel, and the sink drains the channel.
- **Agent:** It is considered as any machine that uses JVM in it. It needs to have sources, sinks, and channels.
- **Client:** It produces and transmits events to sources operating within an agent.

Apart from the installation of Hadoop, the user must have a developer X account, which needs to be used as a source from which information will be extracted.

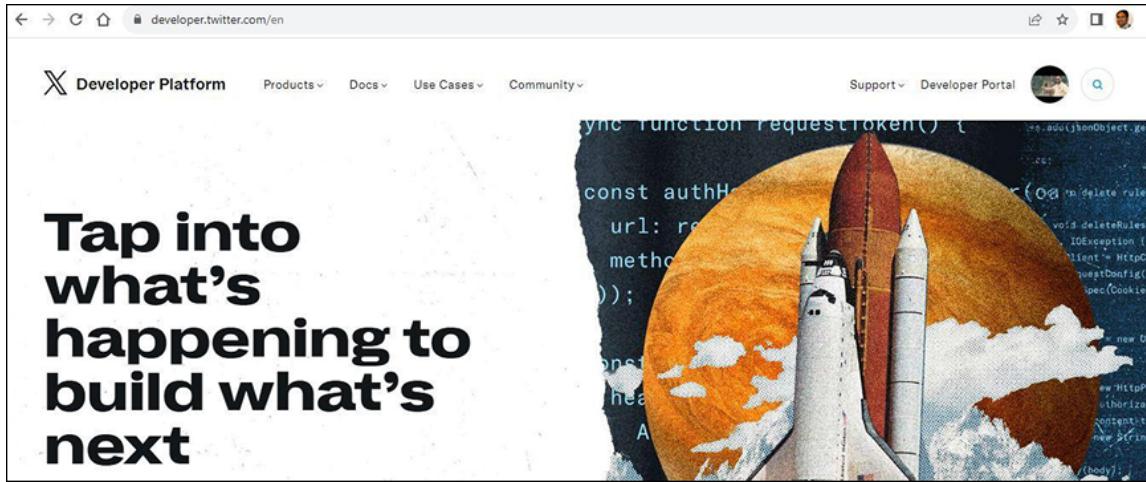
Refer to the following *Figure 9.1*:



*Figure 9.1: Data flow of flume*

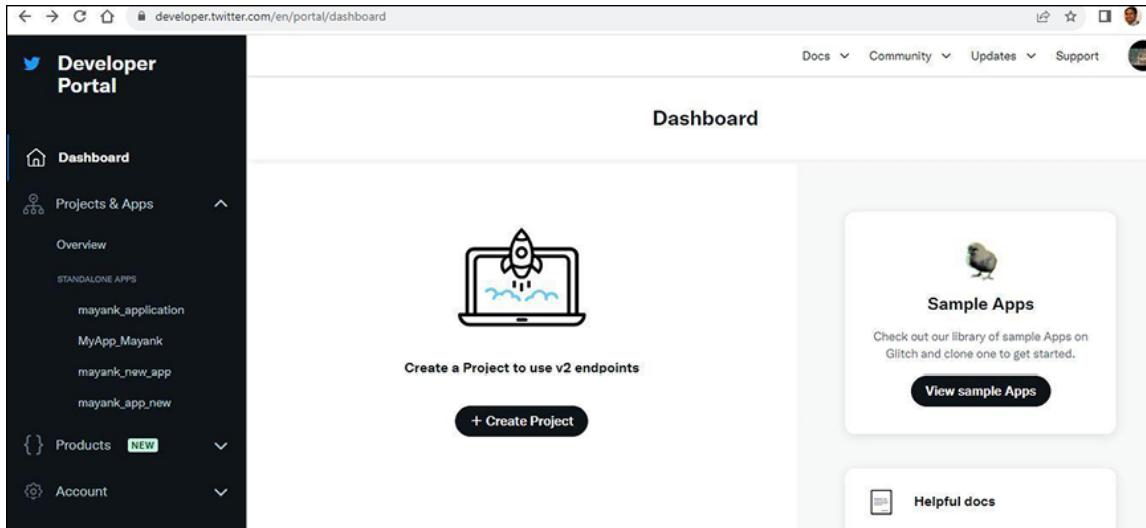
The following steps are required for processing data using flume:

1. Log in to **X** account, and this account should be for the developer. After login to the account need to click on the **Developer** portal (<https://developer.twitter.com/en>), which is in to right corner of X. Refer to *Figure 9.2*:



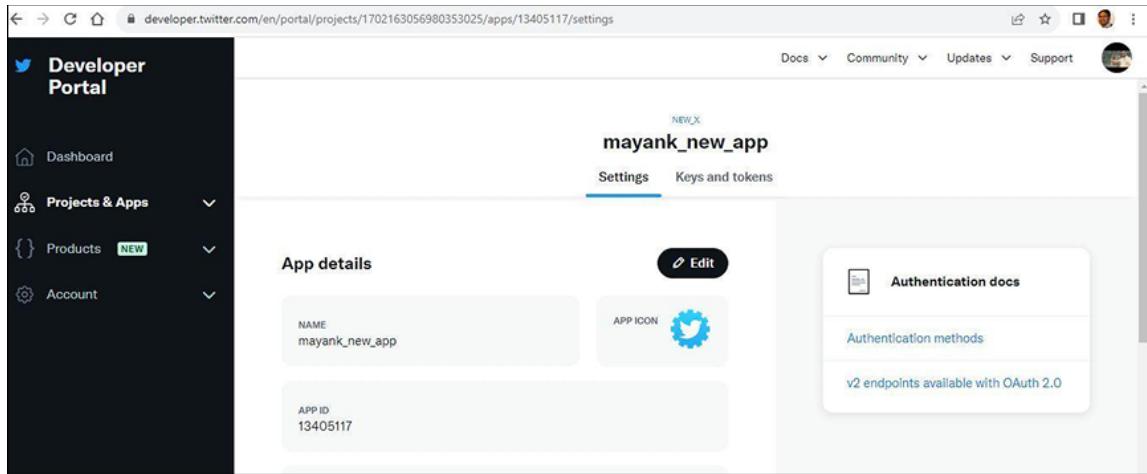
*Figure 9.2: Login to Twitter developer account*

Click on **Create new project** after logging in to X, which needs some information such as **Name**, **usecase**, **project description**, and **app setup**, refer to *Figure 9.3*. After creating a new project, X will ask to create a new one or add it to an existing project. It is the user's choice to choose from.



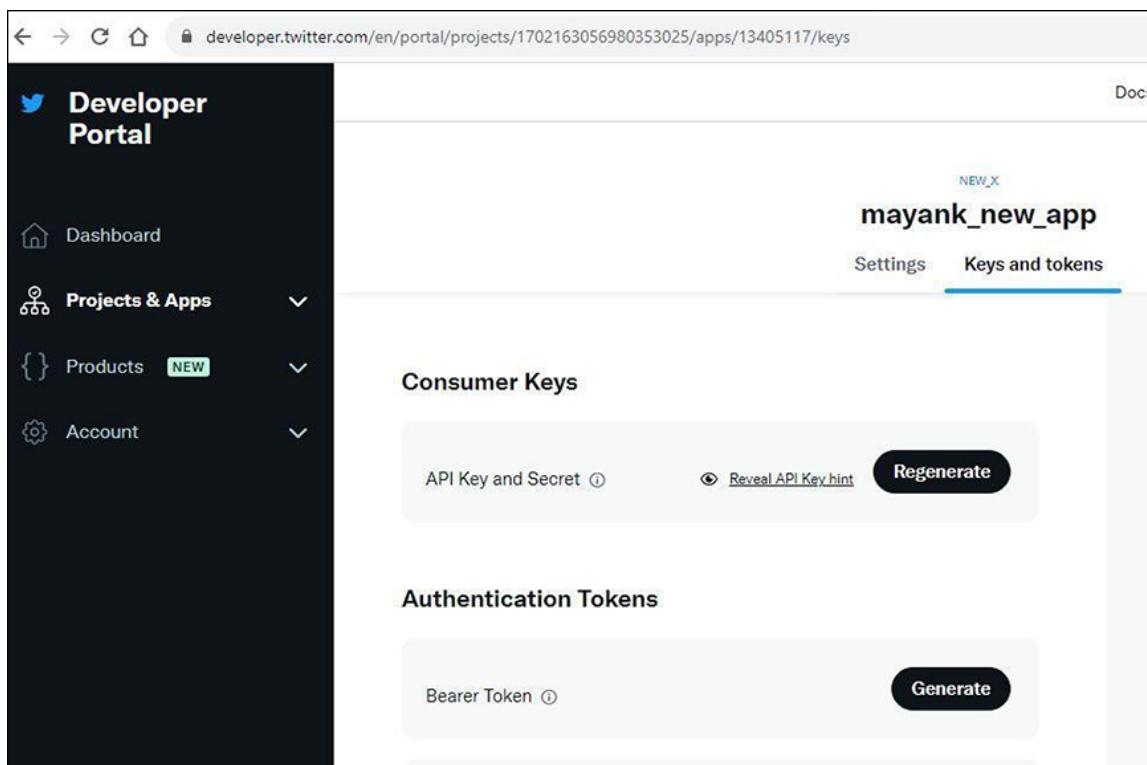
*Figure 9.3: Create a new project*

2. After clicking on Create New project and filling in all the required fields, this project with the given name reflects on the developer portal with **Setting** and **keys and token** option. Refer to *Figure 9.4*:



*Figure 9.4: Application creation*

The **Keys and Tokens** section provides the information needed for synchronization and downloading data from Twitter to HDFS. Refer to [Figure 9.5](#):



*Figure 9.5: Keys and Access Token*

**Consumer Key**, **Consumer Secret**, **Access Token**, and **Access Token Secret** are needed for further use.

3. Download flume from the command prompt:

```
wget http://apache.mirrors.hoobly.com/flume/1.7.0/apache-flume-1.7.0-bin.tar.gz
```

Check the existence of flume in the folder and create a separate directory for flume:

```
sudo mkdir /usr/lib/flume-ng
```

4. Copy flume tar file that was downloaded to the flume-ng directory:

```
sudo cp -r apache-flume-1.7.0-bin.tar.gz /usr/lib/flume-ng/
```

5. After checking if flume is copied, change the directory to flume-ng using:

```
cd /usr/lib/flume-ng/
```

6. Extract the file to **flume-ng** directory:

```
sudo tar -xvf /usr/lib/flume-ng/apache-flume-1.7.0-bin.tar.gz
```

7. Create **flume.env.sh** file in the **conf** directory of Apache flume:

```
sudo cp /usr/lib/flume-ng/apache-flume-1.7.0-bin/conf/flume-env.sh.template /usr/lib/flume-ng/apache-flume-1.7.0-bin/conf/flume-env.sh
```

8. Now open **flume-env.sh** using **gedit** command; refer to *Figure 9.6*:

```
sudo gedit /usr/lib/flume-ng/apache-flume-1.7.0-bin/conf/flume-env.sh
```

```

*flume-env.sh (/usr/lib/flume-ng/apache-flume-1.4.0-bin/conf) - gedit
File Edit View Search Tools Documents Help
Open Save Undo Cut Copy Paste Find Replace
*flume-env.sh x
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# If this file is placed at FLUME_CONF_DIR/flume-env.sh, it will be sourced
# during Flume startup.

# Environment variables can be set here.

JAVA_HOME=/usr/lib/jvm/java-6-sun-1.6.0.24

# Give Flume more memory and pre-allocate, enable remote monitoring via JMX
#JAVA_OPTS="-Xms100m -Xmx200m -Dcom.sun.management.jmxremote"

# Note that the Flume conf directory is always included in the classpath.
FLUME_CLASSPATH="/usr/lib/flume-ng/apache-flume-1.4.0-bin/lib/flume-
sources-1.0-SNAPSHOT.jar"

```

**Figure 9.6:** Flume configuration

*Figure 9.6* reflects the configuration of flume that has information about Java and its classpath that needs to be recognized with the flume and the system. **Flume-source-1.0.SNAPSHOT.jar** can be downloaded from an external source if it is not reflected in the system.

9. Edit **flume.conf** file, which needs information about the **Consumer Key**, **Consumer Secret**, **Access Token**, and **Access Token Secret**:

```

# Define the agent's name

agent.sources = mySource
agent.sinks = mySink
agent.channels = myChannel

# Configure the source

agent.sources.mySource.type = [Source_Type]

```

```
agent.sources.mySource.property1 = value1
agent.sources.mySource.property2 = value2
# Configure the sink
agent.sinks.mySink.type = [Sink_Type]
agent.sinks.mySink.property1 = value1
agent.sinks.mySink.property2 = value2
# Configure the channel
agent.channels.myChannel.type = [Channel_Type]
agent.channels.myChannel.property1 = value1
agent.channels.myChannel.property2 = value2
# Bind the source and sink to the channel
agent.sources.mySource.channels = myChannel
agent.sinks.mySink.channel = myChannel
```

*Figures 9.7* and *9.8* show Twitter's needed data. *Figure 9.7* has space for keywords that need to be mentioned, which will be downloaded limited from Twitter:

```

*flume.conf (/usr/lib/flume-ng/apache-flume-1.4.0-bin/conf) - gedit
File Edit View Search Tools Documents Help
Open Save Undo Cut Copy Paste Find Replace
*flume.conf

TwitterAgent.sources = Twitter
TwitterAgent.channels = MemChannel
TwitterAgent.sinks = HDFS
TwitterAgent.sources.Twitter.type = com.cloudera.flume.source.TwitterSource
TwitterAgent.sources.Twitter.channels = MemChannel

TwitterAgent.sources.Twitter.consumerKey =[Required-Copy from twitter App]
TwitterAgent.sources.Twitter.consumerSecret =[Required-Copy from twitter App]
TwitterAgent.sources.Twitter.accessToken =[Required-Copy from twitter App]
TwitterAgent.sources.Twitter.accessTokenSecret =[Required-Copy from twitter App]

TwitterAgent.sources.Twitter.keywords = vineet, hisham, hadoop, big data, analytics, bigdata, cloudera, data science, data
scientist, business intelligence, mapreduce, data warehouse, data warehousing, mahout, hbase, nosql, newsql,
businessintelligence, cloudcomputing

TwitterAgent.sinks.HDFS.channel = MemChannel
TwitterAgent.sinks.HDFS.type = hdfs
TwitterAgent.sinks.HDFS.hdfs.path = hdfs://localhost:8020/user/flume/tweets/
TwitterAgent.sinks.HDFS.hdfs.fileType = DataStream
TwitterAgent.sinks.HDFS.hdfs.writeFormat = Text
TwitterAgent.sinks.HDFS.hdfs.batchSize = 1000
TwitterAgent.sinks.HDFS.hdfs.rollsize = 0
TwitterAgent.sinks.HDFS.hdfs.rollCount = 10000
TwitterAgent.sinks.HDFS.hdfs.rollInterval = 600
TwitterAgent.channels.MemChannel.type = memory
TwitterAgent.channels.MemChannel.capacity = 10000
TwitterAgent.channels.MemChannel.transactionCapacity = 100

```

**Figure 9.7:** Flume configuration

**Figure 9.8** shows the flume fie configured:

```

*flume.conf (/usr/lib/flume-ng/apache-flume-1.4.0-bin/conf) - gedit
File Edit View Search Tools Documents Help
Open Save Undo Cut Copy Paste Find Replace
*flume.conf

TwitterAgent.sources = Twitter
TwitterAgent.channels = MemChannel
TwitterAgent.sinks = HDFS
TwitterAgent.sources.Twitter.type = com.cloudera.flume.source.TwitterSource
TwitterAgent.sources.Twitter.channels = MemChannel

TwitterAgent.sources.Twitter.consumerKey =@p8dkYUhQmn8uRt7C71w
TwitterAgent.sources.Twitter.consumerSecret =G04dAE50ahieqUeh6vZoF0MAC2YIcgkgu8vpwaJamb
TwitterAgent.sources.Twitter.accessToken =1635433267-HvStt9EggeUhrxNTTJhc4DKrDymxfHGi3oroJpc
TwitterAgent.sources.Twitter.accessTokenSecret =qBX9kFeM1K7zSr0IMhUUYfQmvLeuWv7ms0EgyhLls

TwitterAgent.sources.Twitter.keywords = vineet, hisham, hadoop, big data, analytics, bigdata, cloudera, data science, data
scientist, business intelligence, mapreduce, data warehouse, data warehousing, mahout, hbase, nosql, newsql,
businessintelligence, cloudcomputing

TwitterAgent.sinks.HDFS.channel = MemChannel
TwitterAgent.sinks.HDFS.type = hdfs
TwitterAgent.sinks.HDFS.hdfs.path = hdfs://localhost:8020/user/flume/tweets/
TwitterAgent.sinks.HDFS.hdfs.fileType = DataStream
TwitterAgent.sinks.HDFS.hdfs.writeFormat = Text
TwitterAgent.sinks.HDFS.hdfs.batchSize = 1000
TwitterAgent.sinks.HDFS.hdfs.rollSize = 0
TwitterAgent.sinks.HDFS.hdfs.rollCount = 10000
TwitterAgent.sinks.HDFS.hdfs.rollInterval = 600
TwitterAgent.channels.MemChannel.type = memory
TwitterAgent.channels.MemChannel.capacity = 10000
TwitterAgent.channels.MemChannel.transactionCapacity = 100

```

**Figure 9.8:** Flume file configured

10. Change the directory to the **bin** folder of Apache flume using the following:

```
cd /usr/lib/flume-ng/apache-flume-1.7.0-bin/bin/
```

11. Start fetching the data from X after configuring it in the proper manner:

```
./flume-ng agent -n TwitterAgent -c conf -f /usr/lib/flume-
ng/apache-flume-1.7.0-bin/conf/flume.conf
```

12. The previous step will download all the tweets in HDFS. Now user needs to login with **NameNode** and download all the tweets for processing. Under **namenode**, **/user/flume** directory over cluster will show all tweets as given in *Figure 9.9*:

```
url='https://t.co/6Y4TTJSE81', isProtected=false, followersCount=
8707, status=null, profileBackgroundColor='0099B9',
profileTextColor='3C3940', profileLinkColor='E81C4F',
profileSidebarFillColor='95E8EC',
profileSidebarBorderColor='FFFFFF',
profileUseBackgroundImage=true, isDefaultProfile=false,
showAllInlineMedia=false, friendsCount=1656, createdAt=Wed Nov 14
15:28:09 IST 2012, favouritesCount=80682, utcOffset=19800,
timeZone='New Delhi',
profileBackgroundImageUrl='http://abs.twimg.com/images/themes/the
me4/bg.gif',
profileBackgroundImageUrlHttps='https://abs.twimg.com/images/themes/theme4/bg.gif', profileBackgroundTiled=false, lang='en',
statusesCount=381, isGeoEnabled=true, isVerified=false,
translator=false, listedCount=10, isFollowRequestSent=false,
withheldInCountries=null}, withheldInCountries=null,
quotedStatusId=-1, quotedStatus=null}, userMentionEntities=
[UserMentionEntityJSONImpl{name='Er P K Prajapati â ',
screenName='ErPKP', id=947446562}], urlEntities=[],
hashtagEntities=[HashtagEntityJSONImpl
{text='InternationalYogaDay'}], mediaEntities=[], symbolEntities=
[], currentUserRetweetId=-1, user=UserJSONImpl{id=4734335114,
name='â'•â'•sonaaâ'•â'•', screenName='sunitasharma202',
location='BHOPAL , India', description='be positive n think
positive. â'•married â'•spreading entertainment. shairy
writing me hard sunna n sunana trust on and *tweets like
```

*Figure 9.9: Tweets*

## Using MapReduce

Another method is about downloading tweets with the help of MapReduce. This method reflects the program which we used and configured in downloading. The following is the program that can be used directly to download files from X with specified keywords:

```
import java.io.*;
```

```
import java.util.*;
import twitter4j.*;
import twitter4j.conf.ConfigurationBuilder;
public class TweetsDownload
{
    private static final String Filename =
"/home/mayank/tweetshashnew.txt";
    private static final String Filenametotal =
"/home/mayank/tweetshashtotalnew.txt";
    public static void main(String[] args) throws
Exception
    {
        // Authentication
        module.....//  

        ConfigurationBuilder cb = new
ConfigurationBuilder();
        cb.setDebugEnabled(true)
        .setOAuthConsumerKey(" ")
        .setOAuthConsumerSecret(" ")
        .setOAuthAccessToken(" ")
        .setOAuthAccessTokenSecret(" ");
        // Authenticated....//
        Twitter twitter = new
TwitterFactory(cb.build()).getInstance();
        Query query = new Query("#InternationalYogaDay");
        String hashtag = query.getQuery();
```

```
int numberoftweets = 1000;
long lastID = Long.MAX_VALUE;
ArrayList<Status> tweets = new ArrayList<Status>();
while (tweets.size () < numberoftweets) {
    if (numberoftweets - tweets.size() > 100)
        query.setCount(100);
    else
        query.setCount(numberoftweets - tweets.size());
try {
    QueryResult result = twitter.search(query);
    tweets.addAll(result.getTweets());
    System.out.println("Downloaded " +
tweets.size() + " tweets from twitter related to
"+hashtag +"\n");
    for (Status t: tweets)
        if(t.getId() < lastID)
            lastID = t.getId();
}
catch (TwitterException te) {
    System.out.println("Couldn't connect: " + te);
};
query.setMaxId(lastID-1);
}
for (int i = 0; i < tweets.size(); i++) {
```

```
    Status stat = (Status) tweets.get(i);

    String totalinfo = stat.toString();

    String user = stat.getUser().getScreenName();

    String msg = stat.getText();

    String language = stat.getLang();

    FileWriter fw = new FileWriter(Filename, true);

//    FileWriter fw = new FileWriter(Filename);

    BufferedWriter bw = new BufferedWriter(fw);

    String content = user + ";" + msg + ";" +
language + ";" + "\n";

    FileWriter fw1 = new FileWriter(Filenametotal,
true);

//    FileWriter fw1 = new FileWriter(Filenametotal);

    BufferedWriter bw1 = new BufferedWriter(fw1);

    bw.write(content);

    bw1.write(totalinfo);

    //String time = "";

    //if (loc!=null) {

        //Double lat =
t.getLocation().getLatitude();

        //Double lon =
t.getLocation().getLongitude(); */

//        System.out. println(i + " USER: " + user + "
wrote: " + msg + "\n");

//        System.out. println(content);
```

```
        bw.close();
        bw1.close();
        fw.close();
        fw1.close();

    }

//else

    //System.out.println(i + " USER: " + user + "
wrote: " + msg+"\n");

}

}
```

The preceding program will run as other MapReduce programs run with reference to the MapReduce chapter. The result is shown in *Figure 9.10*:

**Figure 9.10:** Format of downloaded Twitter data

## Use of Bloom filter in MapReduce

Bloom filter is used to speed up answers in a storage system by using multiple hash functions to quickly test whether an element is a member of a set. Bloom filters are efficient data structures for testing set membership, offering advantages such as constant-time lookups and low memory usage. They are particularly useful in scenarios where false positives are acceptable, such as caching and filtering operations, as they can quickly eliminate non-members. However, they come with trade-offs, including the possibility of false positives, lack of support for element deletion, and the need to set an optimal filter size and number of hash functions, which can impact their accuracy and memory efficiency, making them less suitable for applications where false positives are unacceptable or when dynamic data modification is required. This provides the facility of a probabilistic approach for finding solutions for large databases for faster results. It is more suitable for distributed applications.

The key idea of the Bloom filter is to store any bit with any number of hash functions and then fetch it in a manner that will read only a few numbers of bits to provide an exact result. It can convert a result like **q 2 S** to **q!S**. Nowadays, millions of people share their files in a network, so that it creates popularity for **peer-to-peer (P2P)** computing. In 2004, 70 million users used the file-sharing concept. Recent research in Sweden showed that 75% of youngsters use the file-sharing concept. Bloom filter concept can be used in P2P computing because it has bandwidth-efficient technology that functions with a reduced amount of data to communicate with. In many places, proxy servers are used to store data on Web pages. Proxy servers work to provide faster access to fetch data for users. So, the number of pages are stored with this server with a specified address. Bloom filter can provide functions to make it faster to provide quick results. It can also be used in Web-based searching because there are 29.2% of data found common when 150 million pages of similar topics were researched. There was a repetition of information.

## **The function of the bloom filter**

Bloom filter is a probabilistic data structure that is space efficient with a little error allowable when there is a test performed. Its data structure was developed by *Burton H. Bloom* in 1970. Bloom filter stores elements in an array using hash functions. Let us say that a set  $S = x_1; x_2; \dots; x_n$  constructs a data structure to answer queries about the presence of element existence like, Is  $y$  in  $S$ ? This query does not provide a direct result; it just provides an idea about the data's presence. It also deals with some allowable errors with the bloom filter. Bloom filter sets all bits of the array to initially 0. Then, a bloom filter uses  $k$  independent hashing functions  $h_i$  from  $(1 \dots k)$ . Each hash function maps an item  $a$  to one of the  $m$ -array positions with uniform random distribution. To check whether  $a$  is member, we need to check the  $h_i$  value of that position. If this value is 1, that means the item is stored in the position; otherwise, it is not. With this, there is no need to check the full data structure. When checking data presence, there is a possibility that FP may arise in the Bloom filter. It may be that a situation comes that element  $x$  does not belong to the array but is providing a positive result of its presence. [\*\*Figure 9.11\*\*](#) shows the basic Bloom filter. The functioning of an accurate Bloom filter depends on its size:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Hash each item  $x_j$  in the S dataset k times. If  $Hi(xj) = a$ , set  $B[a] = 1$

0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

To check if  $y$  is in  $S$ , check  $B$  at  $Hi(y)$ . All  $k$  values must be 1.

0	1	0	<b>0</b>	1	0	<b>1</b>	0	0	1	<b>1</b>	1	0	1	1	0
---	---	---	----------	---	---	----------	---	---	---	----------	---	---	---	---	---

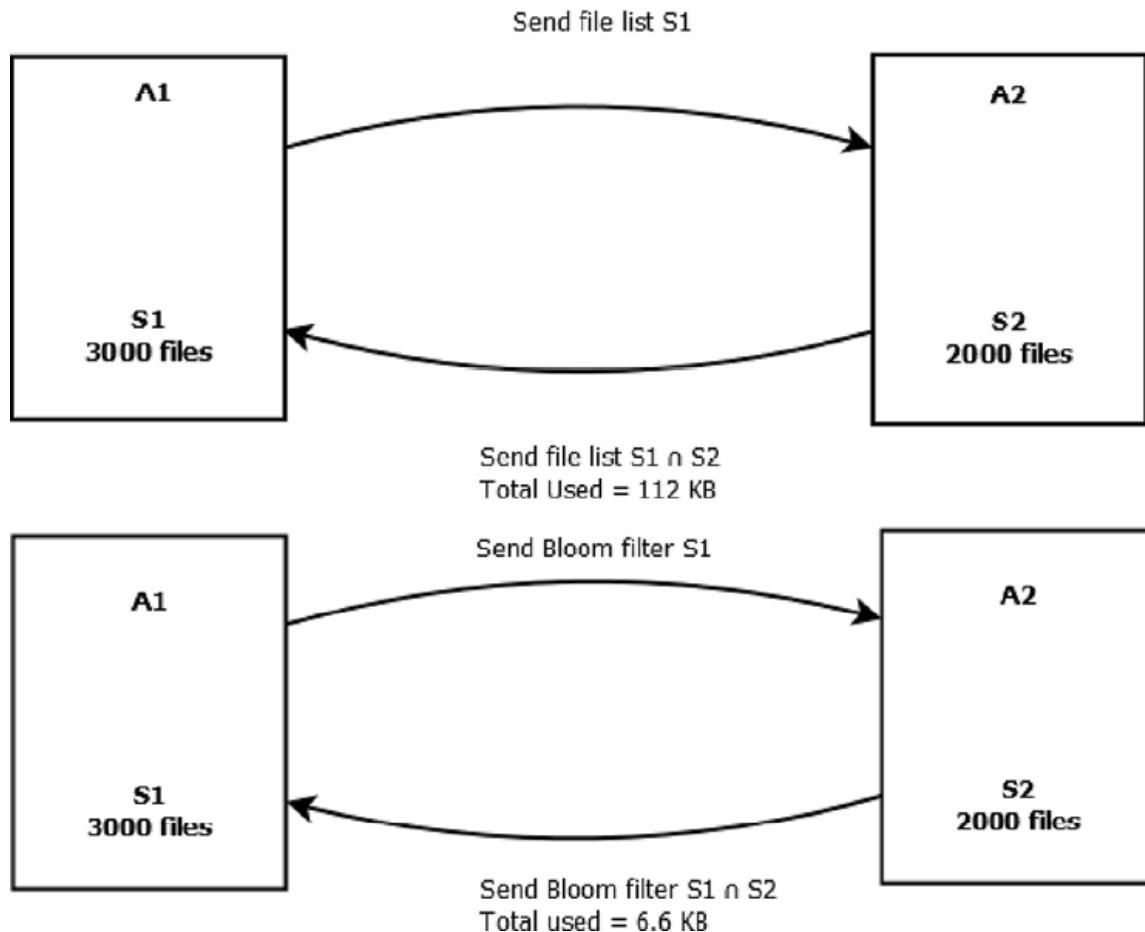
Possible to have a false positive; all  $k$  values are 1, but  $y$  is not in  $S$ .

0	1	0	0	<b>1</b>	0	<b>1</b>	0	0	1	<b>1</b>	1	0	1	1	0
---	---	---	---	----------	---	----------	---	---	---	----------	---	---	---	---	---

*Figure 9.11: Description of Bloom filter*

The array is considered as Bloom filter that consists of “ $n$ ” elements of “ $k$ ” bits. The probability to set any bit of array “1” by the hash function is “ $1/m$ ” and then probability not set to 1 will be “ $1-1/m$ ”, if it is not set 1 by any  $n$  member of the array, then probability “ $(1 - 1/m)^n$ ” since there are  $k$  bits in message then probability will become “ $(1-1=m)^{nk}$ .” If an element is found in an array, that is, 1, then it should be “ $1 - (1 - 1=m)^{nk}$ .” For the complete message found in the array, probability becomes “ $f$ ”, that is:

$$f = (1 - (1 - 1=m)^{nk})k \text{ equivalent to } (1 - e^{\{-kn/m\}} )^k$$



**Figure 9.12: Comparative study**

Bloom filter provides an overhead for data search, as shown in [Figure 9.12](#). With this overhead, data can be searched to find results by surfing a few bits. The result of the experiment is that for a 97% exact match, the number of bits for the size of the Bloom filter should be 8x the number of chunks in the document. If there are two documents of sizes 8 KB and 64 KB, respectively, then 3.2% and 0.4% overhead of size 256 bytes would be required. This means there would be no need to search through 8KB of data for any item. Only 256 bytes are required to determine whether an element is present or not, and the same applies to a 64 KB document. Bloom filters are beneficial in applications due to their small size. Let us say there are two applications, A1 and A2, which consist of S1 (3,000 files) and S2 (2,000 files), respectively. In conversation with both of them, 256 bits are used. Now, if there is a task to get  $S1 + (S1 \cap S2)$  in A1, then traditionally, A1 will send 96KB data of S1 to A2, and after performing the operation of  $(S1 \cap S2)$ , it will return 16 kB data to A1. A total of 112 kB of data is used in processing.

If the Bloom filter is used in both applications, then A1 will only send overhead of S1 to A2, that is, 5.6 kB data; after performing (S1\|S2), it will send data 938 bytes to A1 as shown in [Figure 9.4](#). So, in a transaction with a Bloom filter, 6.6KB of data is used. There is a reduction of 94% in data transfer. This can save bandwidth in the network.

## Working of Bloom filter

There are two main functions used in the Bloom filter: the first is storing a true value that returns from the function, and the second is a function for the testing value that was stored earlier. The first function accepts value to store it in a Bloom array according to its size, which should be well calculated. Second, it justifies whether the function is present there or not. It provides a Boolean value for the result. If it gives all true values that are stored, then the value is present in the database. If it returns any false value, then that value will not be found. A conventional Bloom filter is additive, so it can store extra Bloom positive values from function and cannot remove the values stored earlier.

For good results of Bloom filter, high quality hash functions like MD5 are needed. Its property to generate different hash values for every word means if there is a slight change in a word, it can produce a different value for it. MD5 function takes N bytes from an array and produces a 128-bit value. Bloom filter indexes are obtained by breaking the Bloom filter key into blocks. If there is a block size of 16 bits, then 128 bits are broken into 8-bit complete 16-bit segments.

[Figure 9.13](#) shows a string **Roonie is in park**, that is used to store an array using Bloom filter. It has an index size of 16 bits and an array size that is 64K bits long. Let us assume that for storing this, there is a need to hash this string using the MD5 hash function, which will generate a 128-bit value

**0x10027AB30001BF7877AB34D976A09667**. The first three segments of bits are used as index values to be stored in an array. The remaining bits in 128 bits will be ignored. These three sets of segments of bits will modify the value of the array from “0” to “1.” So, when checking if the string is present in the database, there is no need to check all bits of the index.

There will be a need to check three locations; if those locations present a true result, then the data is present, as shown in [Figure 9.13](#). As a property of MD5, if there is a change in even a single bit of string, then the hash value will dramatically change. There may be cases when data is not present in the array, yet the result is shown. For example, if there is a need to check **Roonie is in office**, it also

generates 128 bits of hash value, which checks 12 bits of it in the array. The array may represent 1 or 0. If it represents 1, that means the string is present. This is the chance of a false positive. *Figure 9.13* shows the case of a false positive. The first line gives a false result in an array, whereas the third string produces a false result. A case may arise when all checking strings represent 1, although the element is not present in it. To avoid that situation, there is a need to calculate the error probability.

Bloom filter is created with an array to give information on whether an element is stored in it. Following is the algorithm for the creation of the Bloom filter:

```
{  
    BloomFilter  
        (set A, hash functions, integer m) returns filter  
        filter = allocate m bits  
        foreach ai in A:  
            foreach hash function hj:  
                filter [hj(ai)] = 1  
            end foreach  
        end foreach  
        return filter  
}
```

Members can be added in the Bloom filter; the following is the algorithm for adding elements in an array of Bloom filters. At the time of adding elements, there is a need to hash that element for the following reasons:

Hash functions are a crucial component of a Bloom filter for several reasons:

- **Uniform distribution:** Hash functions are used to transform input data (for example, an element you want to check for membership) into a fixed-size array index. Well-designed hash functions provide a uniform distribution of hash values across the array, reducing the likelihood of collisions (multiple

elements mapping to the same array index). This uniform distribution is important for minimizing false positives.

- **Multiple hash functions:** Bloom filters typically use multiple hash functions. When you insert an element into the filter, you apply each of these hash functions to map the element to multiple positions in the filter array. This multiple hashing helps to spread the bits in the filter more uniformly, reducing the chances of false positives.
- **Independence:** Hash functions should be independent of each other. By using different hash functions, you reduce the likelihood of different elements having the same set of positions in the filter. This further helps in reducing the probability of false positives.

```
{  
    for (i=1....k)  
        do  
            hash of all object (x) with hash function;  
            if (array [position] == 0 then)  
                array [position] = 1;  
}
```

Take a look at [Figure 9.13](#):

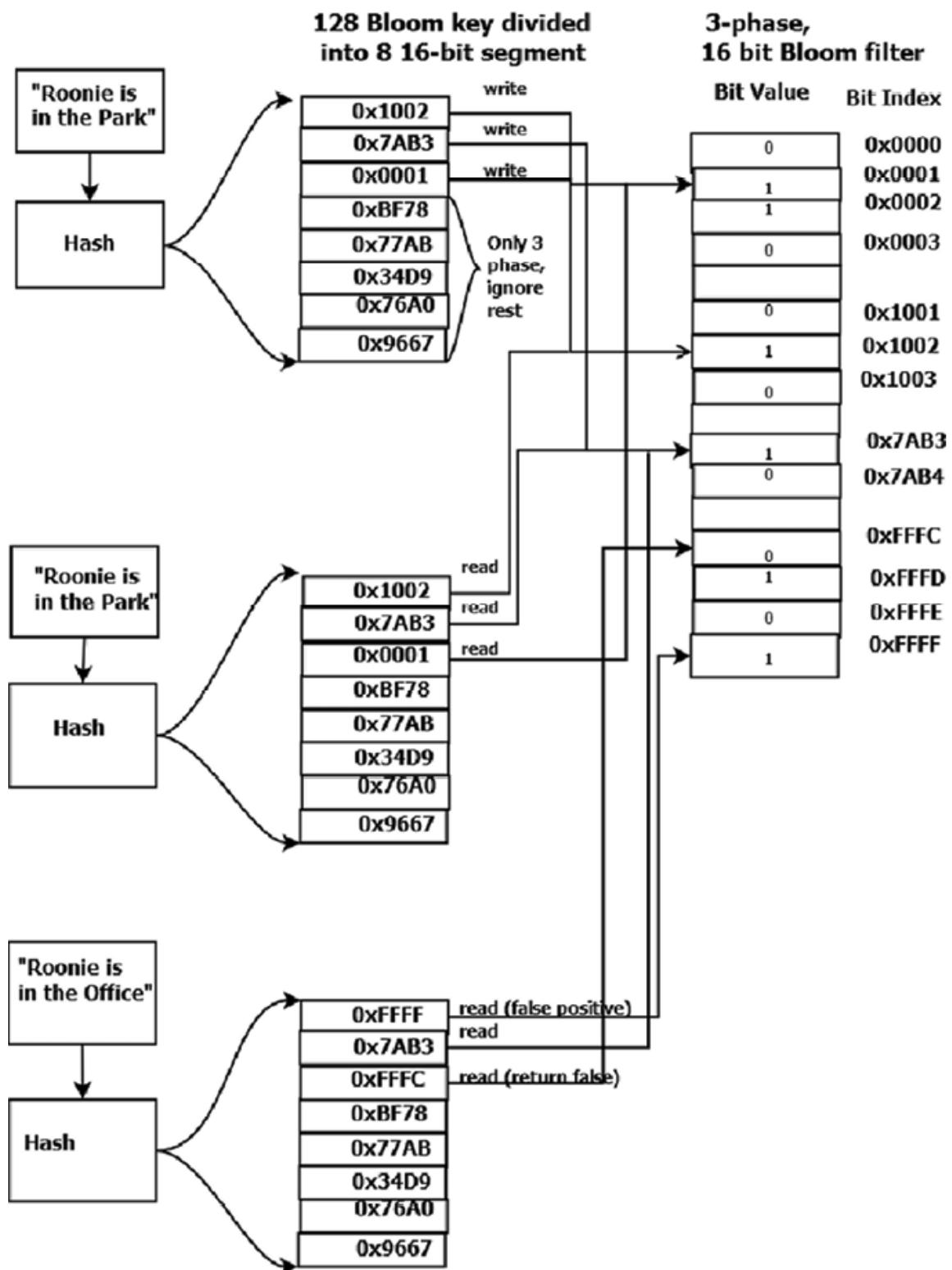


Figure 9.13: Working of Bloom filter

At the time of checking the element presence, the array position is checked. Following is an algorithm for checking the presence of an element:

```

{
    while array[position] == 1 and i <k;
    do
        x is member of list;
        if array[position] == 0 then
            x is not member of list;
        increment position;
}

```

## **Application of Bloom filter**

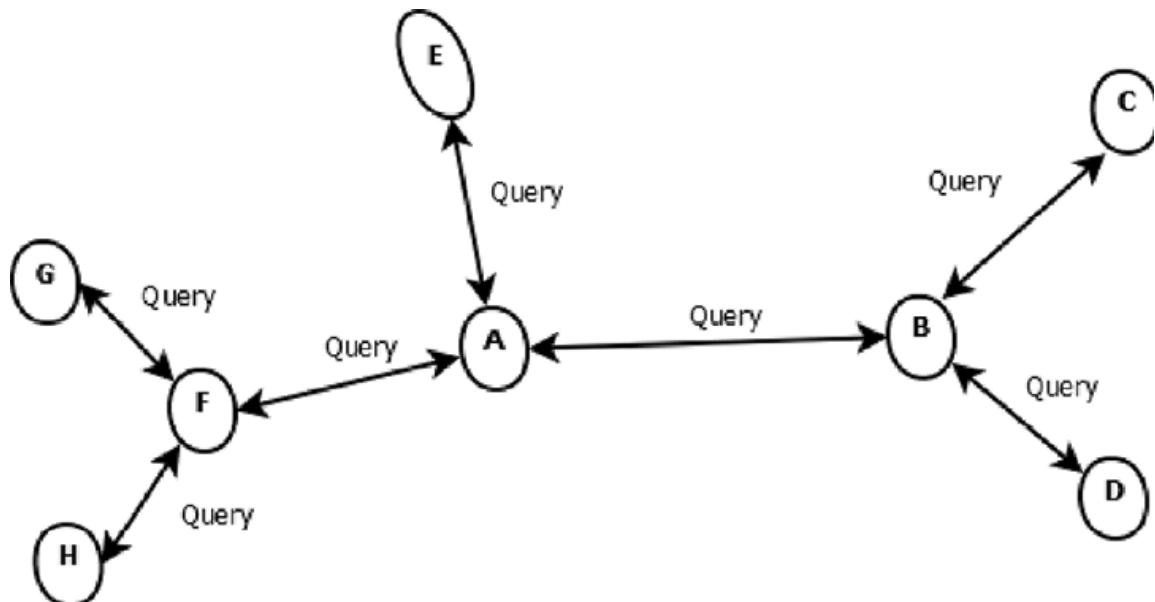
As the Bloom filter is more appropriate for its working with space and time efficiency, it is used in many areas of computer technology. There are a lot of usages in network and data storage fields:

- **Spell checking:** When there is a problem with storing all data in memory, a bloom filter can be used effectively, as seen in the UNIX spell checker. By using it to store spelling, it reduces the amount to access the hard disk. With this, the speed of the system will be faster for accessing data.
- **Password protection:** For providing security to passwords, frequently used words are banned. For matching these types of words, the Bloom filter is used for its implementation. Frequently used words such as the name of the person, city name, date of birth, and so on can be stored in an array which can be matched by words that are entered by the user at the time of creating the password.
- **Attack checking:** Bloom filter is used by Google in its browser, Google Chrome, to save it from attacks. Viruses are a serious problem for any Internet activity. If there is a list of suspicious attacks and every site entered by the user is checked, then it will be helpful for users to get rid of that problem. Bloom filter provides this functionality and proposed an embedded Bloom filter to scan the network.
- **Metadata finding:** When there is a lot of data in files, problems may occur to find out the server is holding data. A simple Bloom filter uses an array for

storing a set of files for a look-up test. Lots of memory is consumed when there is a requirement to store large amounts of data in servers.

- **Spam mail control:** Bloom filter used in spam to make it easier to use email lists as a safe list of friends. Sometimes, when the mail goes to spam, it is shifted back to the inbox. Once done, next time it will not go to spam, and there is no need to change it from time to time.
- **Web caching: Internet Cache Protocol (ICP)** is for web cache sharing, which cooperates with many proxy servers. Queries are sent to other servers to get responses about the data. ICP introduces a summary cache for communication between proxy servers. These summary caches consist of counting the Bloom filter of the URL that is stored in the servers. All proxy servers make their communication by sending the Bloom filter to each other.
- **Distributed systems:** Bloom filter is largely used in a distributed environment with storing keys for searching, for example, p2p, in which Skype and BitTorrent fall.

In a p2p network, keys can be exchanged for finding data. The problem with p2p is that users can automatically increase and decrease anytime. It is also useful in finding metadata information for p2p networks. However, it does not guarantee exact results for accurate matching. **Gnutella** protocol is a file-sharing system based on an unstructured P2P overlay that allows for decentralized, scalable, reliable, and anonymous sharing of files between participating nodes. The distributed system also used the Bloom filter to arrange the list of data systematically for easy search. *Figure 9.14* shows an example of a distributed system through Gnutella. The system can forward their query to other systems to get results. Through this method, a p2p system can make a single system that can search for any item and store it. Bloom filter can make it easier to search by storing items with an array:



**Figure 9.14: Gnutella**

Attenuated Bloom Filter is used for developing a probabilistic algorithm. It gives the result of a file near the requesting system. This Bloom filter can only find files within a range, for example, **R** hopes. This attenuated Bloom filter consists of an array, in which the  $i^{\text{th}}$  filter keeps a record of files that are stored in filters and can be reached in  $i^{\text{th}}$  hopes. The functioning of the Bloom filter in exponentially decaying Bloom filter is probabilistically encoded routing tables for efficient information of unstructured p2p networks. This application may be applied for approximate data synchronization by sending compressed data structure to another peer.

**Packet routing:** IP route lookup and classification of packets can be easily solved by Bloom filters. Routing in the network can be improved as packet stores using the Bloom filter can efficiently work to find routes in the network. For that purpose, the counting Bloom filter is used to optimize the hash table in network processing. It also can increase the speed of packets to find a route fast as it needs to visit only a few bits for making the decision. Protocol, IP address, and Port no. can work to populate the Bloom filter, which can be matched with other pairs of its signature. This can resolve the problem of collision and provide a high-speed network. Longest Prefix Matching is an example of implementing a Bloom filter that can speed up network applications. This algorithm performs parallel queries on the Bloom filter for efficient data structure. It is determined by lookups for dependent memory access. **Classless Inter Domain Routing (CIDR)** requires searching variable addresses for finding destination IPs. The basic idea behind Longest Prefix Matching is to have different regular Bloom filters for different

address prefixes. The Bloom filter is updated from time to time to keep track of records.

## Implementation of bloom filter in MapReduce

Bloom filter is used to perform search results in a more effective and efficient manner because it passes all the data through its filter, which creates an array so that at the time of searching, that array can be searched instead of whole data.

The following is the program that is used for the same purpose:

```
import java.io.Serializable;
import java.nio.charset.Charset;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.BitSet;
import java.util.Collection;

public class BloomFilter<E> implements Serializable {
    private BitSet bitset;
    private int bitsetSize;
    private double bitsPerElement;
    private int expectedNumberOfFilterElements; // expected (maximum) number of elements to be added
    private int numberOfAddedElements; // number of elements actually added to the Bloom filter
    private int k; // number of hash functions

    static final Charset charset = Charset.forName("UTF-8"); // encoding used for storing hash values as
```

```
strings
```

```
    static final String hashName = "MD5"; // MD5 gives
    good enough accuracy in most circumstances. Change to
    SHA1 if it's needed

    static final MessageDigest digestFunction;

    static {
        // The digest method is reused between instances
        MessageDigest tmp;
        try {
            tmp =
                java.security.MessageDigest.getInstance(hashName);
        } catch (NoSuchAlgorithmException e) {
            tmp = null;
        }
        digestFunction = tmp;
    }

    /*
    Constructs an empty Bloom filter. The total length of the Bloom
    filter will be
    */
    public BloomFilter(double c, int n, int k) {
        this.expectedNumberOfFilterElements = n;
        this.k = k;
        this.bitsPerElement = c;
```

```

        this.bitsetSize = (int)Math.ceil(c * n);
        numberofAddedElements = 0;
        this.bitmap = new BitSet(bitmapSize);
    }

/*
Constructs an empty Bloom filter. The optimal number of
hash functions (k) is estimated from the total size of the
Bloom and the number of expected elements.

*/
public BloomFilter(int bitmapSize, int
expectedNumberofElements) {

    this(bitmapSize / (double)expectedNumberofElements,
         expectedNumberofElements,
         (int) Math.round((bitmapSize /
(double)expectedNumberofElements) * Math.log(2.0)));
}

/*
Constructs an empty Bloom filter with a given false
positive probability. The number of bits per element and the
number of hash functions is estimated to match the false
positive probability.

*/
public BloomFilter(double falsePositiveProbability,
int expectedNumberofElements) {

    this(Math.ceil(-(Math.log(falsePositiveProbability)
/ Math.log(2))) / Math.log(2), // c = k / ln(2)

```

```
    expectedNumberOfElements,  
        (int)Math.ceil(-  
(Math.log(falsePositiveProbability) / Math.log(2))));  
    // k = ceil(-log_2(false prob.))  
}  
  
/*  
Construct a new Bloom filter based on existing Bloom filter  
data.  
*/  
public BloomFilter(int bitsetSize, int  
expectedNumberOfFilterElements, int  
actualNumberOfFilterElements, BitSet filterData) {  
    this(bitsetSize, expectedNumberOfFilterElements);  
    this.bitset = filterData;  
    this.numberOfAddedElements =  
actualNumberOfFilterElements;  
}  
  
/*  
Generates a digest based on the contents of a String.  
*/  
public static int createHash(String val, Charset  
charset) {  
    return createHash(val.getBytes(charset));  
}
```

```
/*
Generates a digest based on the contents of a String.

*/
public static int createHash(String val) {
    return createHash(val, charset);
}

/*
Generates a digest based on the contents of an array of bytes.

*/
public static int createHash(byte[] data) {
    return createHashes(data, 1)[0];
}

/* Generates digests based on the contents of an array of
bytes and splits the result into 4-byte int's and store them in
an array.

*/
public static int[] createHashes(byte[] data, int
hashes) {
    int[] result = new int[hashes];

    int k = 0;
    byte salt = 0;
```

```
while (k < hashes) {  
    byte[] digest;  
    synchronized (digestFunction) {  
        digestFunction.update(salt);  
        salt++;  
        digest = digestFunction.digest(data);  
    }  
  
    for (int i = 0; i < digest.length/4 && k < hashes; i++) {  
        int h = 0;  
        for (int j = (i*4); j < (i*4)+4; j++) {  
            h <= 8;  
            h |= ((int) digest[j]) & 0xFF;  
        }  
        result[k] = h;  
        k++;  
    }  
}  
return result;  
}  
  
/*
```

**Compares the contents of two instances to see if they are equal.**

```
/*
 * Compare the contents of two instances to see if they are
 * equal.
 */
@Override
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final BloomFilter<E> other = (BloomFilter<E>) obj;
    if (this.expectedNumberOfFilterElements !=
other.expectedNumberOfFilterElements) {
        return false;
    }
    if (this.k != other.k) {
        return false;
    }
    if (this.bitsetSize != other.bitsetSize) {
        return false;
    }
    if (this.bitmap != other.bitmap && (this.bitmap ==
null || !this.bitmap.equals(other.bitmap))) {
        return false;
    }
}
```

```
        }

        return true;
    }

/*
Calculates a hash code for this class.

*/
@Override
public int hashCode() {
    int hash = 7;

    hash = 61 * hash + (this.bitset != null ?
this.bitset.hashCode() : 0);

    hash = 61 * hash +
this.expectedNumberOfFilterElements;

    hash = 61 * hash + this.bitsetSize;

    hash = 61 * hash + this.k;

    return hash;
}

/*
Calculates the expected probability of false positives based on
the number of expected filter elements and the size of the
Bloom filter.

*/
public double expectedFalsePositiveProbability() {
```

```

        return
getFalsePositiveProbability(expectedNumberOfFilterEleme
nts);

    }

/*
Calculate the probability of a false positive given the
specified number of inserted elements.

*/
public double getFalsePositiveProbability(double
numberOfElements) {

    //  $(1 - e^{-k * n / m})^k$ 
    return Math.pow((1 - Math.exp(-k * (double)
numberOfElements

        / (double) bitsetSize)), k);

}

/*
Get the current probability of a false positive. The
probability is calculated from the size of the Bloom filter
and the current number of elements added to it.

*/
public double getFalsePositiveProbability() {

    return
getFalsePositiveProbability(numberOfAddedElements);

}

```

```
/*
     Returns the value chosen for K.<br />
*/
public int getK() {
    return k;
}

/*
Sets all bits to false in the Bloom filter.
*/
public void clear() {
    bitset.clear();
    numberOfAddedElements = 0;
}

/*
Adds an object to the Bloom filter. The output from the
object's
toString() method is used as input to the hash functions.
*/
public void add(E element) {
    add(element.toString().getBytes(charset));
}
```

```
/*
 * Adds an array of bytes to the Bloom filter.
 */
public void add(byte[] bytes) {
    int[] hashes = createHashes(bytes, k);
    for (int hash : hashes)
        bitset.set(Math.abs(hash % bitsetSize), true);
    numberOfAddedElements++;
}

/*
 * Adds all elements from a Collection to the Bloom filter.
 */
public void addAll(Collection<? extends E> c) {
    for (E element : c)
        add(element);
}

/*
 * Returns true if the element could have been inserted into
 * the Bloom filter.
 */
public boolean contains(E element) {
```

```
        return
contains(element.toString().getBytes(charset));
    }

/*
     * Returns true if the array of bytes could have been
inserted into the Bloom filter.
*/
public boolean contains(byte[] bytes) {
    int[] hashes = createHashes(bytes, k);
    for (int hash : hashes) {
        if (!bitset.get(Math.abs(hash % bitSetSize))) {
            return false;
        }
    }
    return true;
}

/*
     * Returns true if all the elements of a Collection could
have been inserted
into the Bloom filter.
*/
public boolean containsAll(Collection<? extends E> c)
{
```

```
        for (E element : c)
            if (!contains(element))
                return false;
        return true;
    }

/*
Read a single bit from the Bloom filter.
*/
public boolean getBit(int bit) {
    return bitset.get(bit);
}

/*
Set a single bit in the Bloom filter.
*/
public void setBit(int bit, boolean value) {
    bitset.set(bit, value);
}

/*
Return the bit set used to store the Bloom filter.
*/
public BitSet getBitSet() {
```

```
        return bitset;
    }

/*
Returns the number of bits in the Bloom filter. Use count() to
retrieve
the number of inserted elements.
*/
public int size() {
    return this.bitsetSize;
}

/*
Returns the number of elements added to the Bloom filter
after it was constructed or after clear() was called.
*/
public int count() {
    return this.numberOfAddedElements;
}

/*
Returns the expected number of elements to be inserted into
the filter.
*/
public int getExpectedNumberOfElements() {
    return expectedNumberOfFilterElements;
```

```

    }

/*
    Get expected number of bits per element when the Bloom
filter is full. This value is set by the constructor
*/
public double getExpectedBitsPerElement() {
    return this.bitsPerElement;
}

/*
    Get actual number of bits per element based on the number
of elements that have currently been inserted and the length of
the Bloom filter.
*/
public double getBitsPerElement() {
    return this.bitsetSize /
(double)numberOfAddedElements;
}
}

```

## [Amazon Web Service](#)

Hadoop runs on large amount of data, which can be unstructured or structured. This kind of data needs to get storage first before its execution. Cloud computing can effectively address storage issues when users avail of cloud services because it provides a scalable and flexible storage solution. With cloud services, users can leverage the virtually limitless storage capacity offered by cloud providers. They can easily scale their storage resources up or down based on their needs, ensuring

they never run out of space. Additionally, cloud storage is accessible from anywhere with an internet connection, providing seamless data access and backup, which is especially valuable for businesses with remote teams. Furthermore, cloud providers manage data redundancy and backups, reducing the risk of data loss due to hardware failures or disasters. Cloud computing not only solves storage problems but also simplifies management and enhances accessibility, making it an attractive solution for users facing storage challenges. Amazon is a major market for providing all services.

Amazon itself has a lot to learn from. There are two core services offered by Amazon, which are **Elastic Computing Cloud (EC2)** and **Simple Storage Service (S3)**. EC2 services are required for computations on Hadoop nodes. It can also be considered as virtual machine. Instances of EC2 can be referred to as a terminology in AWS. All services of EC2 can be opted for rent on an hourly basis:

- If all data from EC2 is deleted once, the services taken by the user will discontinue.
- If any user wants to retain data for future reference, then the user needs to take persistent storage.
- EC2 functions with commodity machines, so there is no issue with the user.
- An instance can boot up using **Amazon Machine Image (AMI)** or an image.
- Most of the users can well serve one of many preconfigured services at a time.
- Linux, Windows Server, OpenSolaris, and so on, with its six variants, are easily supported by AWS.
- Hadoop running on Linux can provide in-built support that works for both EC2 and S3.

## Setting up AWS

Two authentication features given by AWS are the AWS Access Key Identifier and the X.509 client certificate. The following are the points that can be taken into consideration while setting up:

- AWS Access Key Identifier consists of a Key ID and Secret Access Key. Access key ID is a 20-character alphanumeric representation, whereas Secret Access Key is a 40-character representation.

- These are unique keys but can be regenerated if anyone compromises with security.
- These keys need to be configured with Hadoop to access it through S3 with formatted URI.
- X.509 has different scenarios for security, public, and private keys, and that certificate is hundreds of characters long. They have to be stored and managed, as shown in *Figure 9.15*:

**X.509 Certificate**

**Certificate File**

An X.509 Certificate consists of Public Key and a Private Key. The file containing the public key, the certificate file, must contain a base64-encoded DER certificate body. The file containing the private key, the Private Key file, must contain a base64-encoded PKCS#8 private key. The Private Key is used to authenticate requests to AWS.

AWS accepts any syntactically and cryptographically valid X.509 certificates. They do not need to be from a formal Certificate Authority (CA).

To learn more about how certificates are used to authenticate requests, please see the [Developer Guide](#) for services that support X.509 authentication.

**Your X.509 Certificate:**  
cert-ZF34WZELIV2DZOTL2V600KAD2PNDJHRK.pem

**Create New** Create a New X.509 Certificate  
**Download** Download Your X.509 Certificate  
**Upload** Upload Your Own X.509 Certificate  
**Delete** Delete Your Current X.509 Certificate from AWS

*Figure 9.15: X.509 certificate*

- The public key is also called a certificate file. The private key cannot be shared with anyone, although Amazon cannot store a copy of it.

The public key is meant to be shared openly with anyone. It is used to encrypt data that is meant to be kept confidential. When someone wants to send you an encrypted message, they use your public key to scramble the data in such a way that only your corresponding private key can unscramble it. This means that even if the public key is widely available, only the holder of the private key can decrypt and access the original message. Public keys are also used in digital certificates for secure website connections, verifying the authenticity of the website.

On the other hand, the private key is meant to be kept secret and known only to the key's owner. It is used to decrypt data that was encrypted with the corresponding public key. Private keys are used for decrypting received messages, proving the authenticity of a digital signature (as only the private key can produce a signature that corresponds to a given public key), and for secure access to sensitive

resources. Safeguarding the private key is of utmost importance since its compromise could lead to unauthorized access and breaches of confidentiality or data integrity. In summary, public keys are for encryption and verifying, whereas private keys are for decryption and digital signing in the context of secure communication and data protection.

Here is the summary of different kinds of IDs used in AWS: 20-character, alphanumeric Access Key ID, 40-character Secret Access Key, X.509 Certificate file under the `.ec2` directory, X.509 private key file under the `.ec2` directory, and 12-digit (unhyphenated) AWS Account ID.

## Setting up Hadoop on EC2

EC2 can be set up on a Hadoop cluster, which requires the following parameters:

- Configure the single initialization script at `src/contrib/ec2/bin/hadoopec2-env.sh`, which provides information about three variables that are `AWS_Account-ID`, `AWS_Access_Key-ID`, `AWS_Secret_Access_Key`.
- EC2 must know the instance type and image which is used to boot up. This type provides the physical configuration of virtual machines such as CPU, RAM, disk space, and so on.
- Amazon's S3 storage services can store disk images that are used for booting EC2. Amazon S3 to store the disk images or templates that are needed to create and initialize EC2 instances. These images typically contain the operating system, software, and configurations required for specific tasks or applications. Storing them in S3 allows for easy and reliable access to these images when launching new EC2 instances.

Take a look at *Figure 9.16*:

```
Terminal — bash — 164x10
chuck-loms-computer:~/Projects/Hadoop/aws/ec2-api-tools-1.3-30349 chuck$ ec2-describe-images -x all | grep hadoop-images
IMAGE ami-65997c0c hadoop-images/hadoop-0.17.1-x86_64.manifest.xml 914733919441 available public i386 machine oki-071cf9dc arn-051cf9dc
IMAGE ami-4b997c22 hadoop-images/hadoop-0.17.1-x86_64.manifest.xml 914733919441 available public x86_64machine oki-b51cf9dc arn-b31cf9dc
IMAGE ami-b9fe1cd9 hadoop-images/hadoop-0.18.0-1366.manifest.xml 914733919441 available public i386 machine oki-071cf9ce arn-051cf9cc
IMAGE ami-90fe1cf9 hadoop-images/hadoop-0.18.0-x86_64.manifest.xml 914733919441 available public x86_64machine oki-b51cf9dc arn-b31cf9da
IMAGE ami-ea36d203 hadoop-images/hadoop-0.18.1-1366.manifest.xml 914733919441 available public i386 machine oki-071cf9ce arn-051cf9cc
IMAGE ami-f37d397 hadoop-images/hadoop-0.18.1-x86_64.manifest.xml 914733919441 available public x86_64machine oki-b51cf9dc arn-b31cf9da
IMAGE ami-fa6a6e93 hadoop-images/hadoop-0.19.0-1366.manifest.xml 914733919441 available public i386 machine oki-071cf9ce arn-051cf9cc
IMAGE ami-cd6a6ea4 hadoop-images/hadoop-0.19.0-x86_64.manifest.xml 914733919441 available public x86_64machine oki-b51cf9dc arn-b31cf9dc
chuck-loms-computer:~/Projects/Hadoop/aws/ec2-api-tools-1.3-30349 chuck$
```

*Figure 9.16: Hadoop image in AWS*

- Figure 9.16* shows a Hadoop image in AWS, each image having eleven properties. One of the properties specifies the manifest location of the image

that is expressed in two-level hierarchies.

- Manifest location includes several pieces of information like Hadoop version number image's instances 32-bit or 64-bit.
- Application code to master in Hadoop EC2 uses the following script for moving code to cluster:

```
source hadoop-ec2-env.sh  
scp $SSH_OPTS <localfilepath> root@$MASTER_HOST:  
<masterfilepath>
```

## **Examples of data analysis**

Following are the examples of data analysis:

### **Document archived from NY Times**

In 2007, the *New York Times* set out to make all their public domain articles between 1851 and 1922 freely on their website. Doing this required a scalable image conversion system. As a result, *The Days* had to keep its older articles as scanned run-in pictures, and they required image processing to mix different items of every article into one file in the required PDF format.

Previously, these articles were behind a paid wall and did not receive much traffic. The days may use a real-time approach to scale, glue, and convert the run-in pictures. Though that worked well enough for a low volume of requests, it would not be scalable to handle the traffic increase expected from the articles' free availability. The Days needed a stronger design to handle the gap in its archive.

The solution was to pre-generate all the articles as PDF files and serve them different static content. The New York Times had the code to convert the run-in pictures to PDF files. It appeared like an easy task to execute all the articles in one setting rather than addressing every individual article as a request came in. The difficult part of this answer came when one accomplished that the archive had 11 million articles consisting of 4TB of data.

A software system programmer at The Days thought this was a perfect chance to use AWS and Hadoop. Storing and serving the final set of PDFs from Amazon's straightforward S3 was already deemed a less expensive approach than scaling up the storage back-end of the website.

The 4 TB of pictures were run-in into S3. He *started writing code to drag all the components that structure an article out of S3, generate a PDF from them, and store the PDF back in S3. This was straightforward enough using the JetS3t—Open supply Java toolkit for S3, iText PDF Library, and putting in the Java Advanced Image Extension 1.* When tweaking his code to figure among the Hadoop framework, Derek deployed it to Hadoop running on 100 nodes in Amazon's **Elastic Compute Cloud (EC2)**. The duty ran for 24 hours and generated another 1.5TB of data to be held on in S3.

At 10 cents per instance per hour, the full job was completed with cost accounting of only \$240 ( $100 \text{ instances} \times 24 \text{ hours} \times \$0.10$ ) in computation. The storage value for S3 was additional; however, because The Times had set to archive its files in S3 anyway, that value was already amortized. With data transfer between S3 and EC2 being free, the Hadoop job did not incur any bandwidth value in the least.

## **Data mining in mobiles**

From earlier efforts at IBM, we have learned that these issues are troublesome to beating the mistreatment of ancient data retrieval techniques. Afterward, we have a tendency to project an approach consisting of elaborated offline analyzes to pre-identify guidance pages and, therefore, the use of a special-purpose direction index. Demonstrate the viability of this approach through experiments over a 5.5-million-page corpus from the IBM intranet. The system uses a mixture of proprietary platforms and relational databases. Since then, it crawled a way larger portion of the IBM intranet, having discovered over 100 million URLs and indexed over 16 million documents. So as to tackle these sizes and beyond, and having learned from previous efforts, we have developed ES2, a scalable, high-quality search engine for the IBM intranet. ES2 relies on the analytics described, but it leverages a variety of open supply platforms and tools, like Hadoop, Nutch, Lucene, and Jaql.

In principle, the Nutch crawler, the Hadoop MapReduce framework, and the Lucene indexing engine provide a full suite of software elements for building a whole search engine. But, to address the challenges described earlier, it is not sufficient to simply stitch these systems together. The way we use refined analyzes, mining of the crawled pages, and special-purpose direction indexes in conjunction with an intelligent question process to ensure effective search quality. However, to understand how these components move, we tend to examine some illustrative search queries and their corresponding results.

It shows the results of running the query idp on ES2. The idp is a descriptor for Individual Development set up, a Web-based HR application in IBM to help in pursuit of worker career development. The first two results returned by ES2 represent two completely different URLs that each enable the user to launch the idp Web application. The third result entry is, in fact, a group of pages describing the idping IDP automatic data processing ADP process, one per country, that are classified along (indicated visually through indentation and, therefore, the presence of a globe icon).

## Hadoop diagnosis

This section gives learning about some diagnosis systems that can help a user in recovering a system:

### System's health

Hadoop's file system can be checked by **fsck**. It checks the file path as well as its health. It is shown in *Figure 9.17*:

```
bin/hadoop fsck /
Status: HEALTHY
Total size: 143106109768 B
Total dirs: 9726
Total files: 41532
Total blocks (validated): 42419 (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: 8
Number of racks: 1
```

*Figure 9.17: Checking system health*

Most of the information should be self-explanatory. By default, **fsck** will ignore files still open for writing by a client. As **fsck** checks the file system, it will print out a dot for each file it found healthy (not shown in the preceding output). It will print out a message for each file that is less than healthy, including the ones that have over-replicated blocks, under-replicated blocks, mis-replicated blocks,

corrupt blocks, and missing replicas. Over-replicated blocks, under-replicated blocks, and mis-replicated blocks are not too alarming as HDFS is self-healing. But the corrupt blocks and missing replicas mean that data has been permanently lost. By default, **fsck** does not act on those corrupt files, but you can run **fsck** with the **-delete** option to remove them. A better way is to run **fsck** with the **-move** option, which moves the corrupted files into the **/lost +found** directory for salvaging. *Figure 9.18* suggests the location of racks, blocks, and so on:

```
bin/hadoop fsck /user/hadoop/test -files -blocks -locations -racks
/usr/hadoop/test/part-00000 35792 bytes, 1 block(s):  OK
0. blk_-4630072455652803568_97605 len=35792 repl=3
  [ /default-rack/10.130.164.71:50010, /default-rack/10.130.164.177:50010,
  /default-rack/10.130.164.186:50010]

Status: HEALTHY
Total size: 35792 B
Total dirs: 0
Total files: 1
Total blocks (validated): 1 (avg. block size 35792 B)
Minimally replicated blocks: 1 (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: 8
Number of racks: 1
```

*Figure 9.18: Report about location*

## Setting permission

Hadoop has the same permission settings as the file system in Linux. That means it has read (**r**), write (**w**), and execute (**x**) permission. Permission settings for directories also closely follow the POSIX model. The **r** permission allows for the listing of the directory. The **w** permission allows for the creation or deletion of files or directories. The **x** permission allows one to access the children of the directory.

## Managing quotas

By default, Hadoop does not force to put any kind of quota, but the user can enable quotas on names on specific directories so that it can easily prevent generating too many files.

The `hdfs dfsadmin -setSpaceQuota <N> <directory>...<directory>` command takes an argument for the number of bytes as each directory's quota. The argument can have a suffix to represent a unit. For example, `20g` would mean 20 gigabytes and `5t` would mean 5 terabytes. All replicas count towards the quota.

## Enabling trash

Its purpose is to prevent you from unintentionally deleting something. In addition to file permissions, an additional safeguard against accidental deletion of files in HDFS is the trash feature. By default, this feature is disabled. When this feature is enabled, the command line utilities for deleting files do not delete the files immediately. Instead, they move the files temporarily to a `.Trash/` folder under the user's working directory. The files are not permanently removed until after a user-configurable time delay. As long as a file is in the `.Trash/` folder, you can restore it by moving it back to its original location.

To enable the trash feature and set the time delay for the trash removal, set the `fs.trash.interval` property in `core-site.xml` to the delay (in minutes). For example, if you want users to have 24 hours (1,440 minutes) to restore a deleted file, you should have in `core-site.xml`:

```
<property>
  <name>fs.trash.interval</name>
  <value>1440</value>
</property>
```

Setting the value to 0 will disable the trash feature.

## Removing DataNode

In order to use this feature, you must create an (initially empty) exclude file in the NameNode's local file system and the configuration parameter `dfs.hosts.exclude` must point to this file during NameNode's startup. When you want to retire **DataNodes**, list them in the exclude file, one node per line. You have to specify the nodes using the full `hostname`, `IP`, or `IP:port` format. Execute the following code:

```
bin/hadoop dfsadmin -refreshNodes
```

To force the **NameNode** to reread the exclude file and start the decommissioning process. Messages like Decommission complete for node **172.16.1.55:50010** will appear in the **NameNode** log files when it finishes decommissioning, at which point you can remove the nodes from the cluster.

If you have started HDFS without setting **dfs.hosts.exclude** to point to an exclude file, the proper way to decommission **DataNodes** is this:

1. Shut down the **NameNode**.
2. Set **dfs.hosts.exclude** to point to an empty exclude file.
3. Restart **NameNode**.
4. After **NameNode** has successfully restarted, follow the preceding procedure.

**Note:** If you list the retiring DataNodes in the exclude file before restarting NameNode, the NameNode will be confused and throw messages like **ProcessReport from unregistered node: node055:50010** in its logs.

The NameNode thinks that it is being contacted by a DataNode outside the system rather than a node to be decommissioned.

## Conclusion

Data analysis plays a pivotal role in today's data-driven world, enabling organizations and individuals to extract valuable insights from vast and complex datasets. It empowers data-driven decision-making, enhances business intelligence, and fosters innovation across various industries. By using advanced techniques and tools, data analysis helps uncover hidden patterns, trends, and correlations, ultimately leading to more informed choices and improved outcomes. As the volume of data continues to grow, the importance of data analysis in extracting actionable knowledge from this wealth of information will only become more pronounced, making it an indispensable discipline for the modern era.

In the upcoming chapter, we will delve into Spark techniques, a subject currently in high demand.

# CHAPTER 10

## Spark

### Introduction

Apache Spark is an open-source, distributed computing system designed for big data processing and analytics. It was developed to address the limitations of the Hadoop MapReduce model by providing a faster, more flexible, and user-friendly framework for handling large-scale data processing tasks. Apache Spark was first introduced in 2009 by *Matei Zaharia* as part of his PhD research at the *University of California, Berkeley*. The project was later open-sourced under the Apache Software Foundation in 2010. Spark was created to address the limitations of the Hadoop MapReduce framework, providing a faster and more flexible platform for big data processing and analytics.

Apache Spark quickly gained popularity in the big data and analytics community due to its in-memory processing capabilities, ease of use, and support for various data processing tasks, such as batch processing, interactive queries, streaming data, and machine learning. It has since become one of the most widely used and adopted frameworks for big data processing and analytics. One of the key innovations of Apache Spark is its ability to execute a **Directed Acyclic Graph (DAG)** of operators for data processing. This is a departure from the strict MapReduce model employed by Hadoop, which requires intermediate data to be written to disk between map and reduce stages. In contrast, Spark allows for in-memory data processing, which can significantly speed up data processing tasks by minimizing data movement and reducing the need to write data to disk. By using a DAG-based execution model, Spark can optimize the order of

operations and perform operations on intermediate data without persisting it to storage. This makes Spark more efficient for iterative algorithms and interactive data analysis, and it can be particularly beneficial for machine learning workloads, where data needs to be iteratively processed and modified.

The ability to work with data in memory and perform operations on intermediate results in a flexible and optimized manner is one of the reasons why Spark has gained popularity in the big data processing and analytics domain. Spark has a rich transformation that enables users to express computation, which means it can present complex sets of code in an easy manner. The beauty of its feature is the **Resilient Distributed Dataset (RDD)**, which does not force you to upload all data again if data is added in the future. RDD enables developers to materialize in the processing pipeline into memory, which does not force it to recompute. Spark also stated a challenge is not CPU productivity but analytics analysis speed. More quickly, analytics provide results quicker, which will lead to easier and faster processing. Spark strives to be closer to Python than MATLAB for big data.

## Structure

In this chapter, we will be covering the following topics:

- Spark programming model
- Record linkage
- Spark shell
- Scala programming model
- Working with Scala
- Resilient distributed dataset
- Spark methods
- Examples

## Objectives

The objective of this chapter on Apache Spark in a learning resource or course typically revolves around providing learners with a comprehensive understanding of this powerful big data processing framework. In this chapter, the primary aim is to introduce learners to the fundamental concepts and capabilities of Spark. This includes explaining its distributed computing model, in-memory processing, and its various APIs for batch processing, real-time streaming, and machine learning. Additionally, the chapter should aim to illustrate how Spark can be integrated into real-world data processing pipelines and used to solve practical data challenges across different industries. Ultimately, the goal is to equip learners with the knowledge and skills to effectively use Spark for big data analysis and processing tasks.

## Spark programming model

The Spark system stores datasets in a distributed fashion using persistent storage systems such as **Hadoop Distributed File System (HDFS)**. It requires the following steps:

1. Defining a set of transformations on the input dataset.
2. Action to invoke transformed dataset output into local memory.
3. Calculates results in a distributed manner by running local computation. This also helps in deciding the next computation.

Spark provides an understanding of the storage and execution framework. Spark pairs these abstractions in an elegant way that essentially allows any intermediate step in a data processing pipeline to be cached in memory for later use.

## Record linkage

As all we know, problems with Big Data are a large collection of records and simultaneously access with functioning from different sources. All

records need attention because they have different data types and attributes such as name, birthday, SSN no., location, and so on. All records also cannot be predicted while stored on clusters, and this will lead to problems with data cleansing. This issue can be referred to as record linkage; the following are the challenges of record lineage:

Name	Address	City	State
John's shop	Kingwood street	Rio de Janeiro	Brazil
John's shop	Pinewood enclave	Sydney	Australia
Paris office	Rosewood palace	Tokyo	Japan
Head office	Knight palace	Delhi	New Delhi

**Table 10.1: Challenge of record linkage**

*Table 10.1* refers to the structure of records that need to get attention for data cleaning. Records 1 and 2 have the same data but in different entries, same with Records 3 and 4 as Rosewood Palace can also be written as Knight Palace, and Delhi can also be referred to as New Delhi. These entries can be identified easily by the human side but are difficult for computers to learn.

The dataset is used for analysis and prediction, and it assigns a numeric value between 0.0 and 1.0 based on the similarity of the strings being compared. Each pair of strings, whether they match or not, will have a label assigned to them for use in linkage research.

From shell, it can be **re\$ mkdir** linkage:

```
$ cd linkage/
$ curl -o donation.zip http://bit.ly/1Aoywaq
$ unzip donation.zip
$ unzip 'block_*.zip'
```

In Hadoop cluster it can be loaded as:

```
$ hadoop fs -mkdir linkage  
$ hadoop fs -put block_*.csv linkage
```

Spark context has a list of methods as follows. The Scala section in this chapter shows important and frequently used RDDs, which are abstract methods for representing the following objects:

Accumable	accumableCollection
Accumulator	addFile
addJar	addSparkListener
appName	asInstanceOf
broadcast	cancelAllJobs
cancelJobGroup	clearCallSite
clearFiles	clearJars
clearJobGroup	defaultMinPartitions
defaultMinSplits	defaultParallelism
emptyRDD	files
getAllPools	getCheckpointDir
getConf	getExecutorMemoryStatus
getExecutorStorageStatus	getLocalProperty
getPersistentRDDs	getPoolForName
getRDDStorageInfo	getSchedulingMode
hadoopConfiguration	hadoopFile
hadoopRDD	initLocalProperties

isInstanceOf	isLocal
jars	makeRDD
master	newAPIHadoopFile
newAPIHadoopRDD	objectFile
parallelize	runApproximateJob
runJob	sequenceFile
setCallSite	setCheckpointDir
setJobDescription	setJobGroup
startTime	stop
submitJob	tachyonFolderName
textFile	toString
union	version
wholeTextFiles	

Like Python, Scala has a built-in tuple type for collecting large values of different types of data to represent records. Each data is parsed with the type of records, and based on scores, it is decided whether fields are matched or not like:

```
val line = head(5)
val pieces = line.split(',')
...
pieces: Array[String] = Array(36950, 42116, 1,
?, ...)
```

In Scala, each array element is a function call, not a special operator with a special function named apply, so **head(10)** will perform the same as **head.apply(10)**.

In Apache Spark, you can persist (or cache) an RDD to either memory or disk so that subsequent actions on that RDD can be performed more efficiently without having to recompute the RDD from the original data source. Caching an RDD can be particularly useful when you have multiple actions or transformations that depend on the same intermediate result.

The **cache()** method in Spark is used to persist an RDD in memory by default. You can also specify other storage levels, including caching on disk, in a serialized form, or a combination of these. Here is how your example code would work:

```
# Assuming 'cached' is an RDD  
cached.cache() # This persists 'cached' in memory (the  
default storage level).
```

```
# Subsequent actions on 'cached' can benefit from  
the data being cached.
```

```
count_result = cached.count() # The 'count' action  
is faster because data is cached.
```

```
take_result = cached.take(10) # The 'take' action is  
faster too.
```

Caching RDDs in Spark is a valuable optimization technique to avoid redundant computations, especially when you have a series of operations or multiple actions that need the same intermediate data. It helps improve the overall performance of your Spark application.

A call to cache indicates that an RDD should be stored as soon as it is computed. When you make a call to compute it initially, the take method is used to retrieve the first 10 elements of the RDD as a local array. This

approach for accessing elements is optimized because it avoids recomputation by using the cached data from its dependencies.

Spark defines:

`StorageLevel` for persisting RDDs

`Rdd.cache()` stores RDD as unserialized java objects

In Apache Spark, the **StorageLevel** is a parameter that allows you to specify how an RDD should be stored in memory or on disk. It provides flexibility in choosing where and how the RDD should be cached based on factors such as memory usage, performance, and fault tolerance requirements. The **StorageLevel** defines characteristics such as:

- **Memory versus disk:** You can specify whether the RDD should be stored in memory RAM or on disk. Storing in memory is faster, but it may consume a significant amount of memory space. Storing on disk is slower but allows you to handle larger datasets.
- **Serialization:** You can specify whether the data should be stored in serialized form to reduce memory usage. Serialization is useful when memory is a constraint, but it comes at the cost of CPU overhead for serialization and deserialization.
- **Replication:** You can define the level of replication for fault tolerance. For example, you can choose to replicate the data on multiple nodes to ensure data availability in case of node failures.

The following are some common **StorageLevel** options in Spark:

- **MEMORY\_ONLY:** Store the data in deserialized form in memory. This provides the best performance but uses more memory.
- **MEMORY\_ONLY\_SER:** Store the data in serialized form in memory. This reduces memory usage but may require deserialization when data is accessed.
- **MEMORY\_AND\_DISK:** Store the data in memory and spill it to disk if the memory is not sufficient.

- **DISK\_ONLY**: Store the data on disk, not in memory. This is the slowest option but may be necessary for very large datasets.
- **MEMORY\_ONLY\_2, MEMORY\_ONLY\_SER\_2, and so on**: These options replicate data on two nodes for fault tolerance.

You can use the **`persist()`** method on an RDD and pass a **StorageLevel** as an argument to specify how the RDD should be cached. For example:

```
cachedRDD.persist(StorageLevel.MEMORY_ONLY_SER_2)
```

By choosing the appropriate **StorageLevel**, you can optimize the performance and resource usage of your Spark application based on your specific requirements and constraints.

```
rdd.persist(storageLevel=StorageLevel.DISK_ONLY)
```

When you call **RDD.cache()**, it stores the RDD in memory, but by default, it is stored as unserialized Java objects. This means that the data is stored in its native, deserialized form, which can consume more memory than storing it in serialized form. You can specify a different storage level if you want to change the default behavior and store the RDD differently, such as in a serialized form to save memory at the cost of CPU overhead during deserialization.

If the partition does not fit into memory, then it will not store data while the next computation takes place. Because this level avoids any serialization overhead, it requires low-latency access. This can affect slowness and put pressure on Java's garbage collection. **MEMORY\_SER** storage level allocates large byte collection memory and serializes RDD contents into them. **MEMORY\_AND\_DISK** and **MEMORY\_AND\_DISK\_SER** spill partitions that do not fit into memory.

## Spark shell

Shell is the centric point on which Spark goes around and is its strongest feature. Shell allows the user to execute commands like the terminal on Unix. Scala is mostly preferred language for this purpose because Spark is

implemented in Scala. Following are the commands (see *Table 10.2*) by which configuration can be done on Spark:

export SPARK_MEM =1g (max memory allotted 1 GB)	For the amount of memory that Spark may use for executing queries, you have to set the following environment prior to starting the shell.
export SPARK_WORKER_INSTANCES =4	The environment variable controls the number of worker threads that Spark uses.
export SPARK_LOG_DIR =/ home / cloudera / Documents / mylog sh spark - shell scala > var logger = new org.apache . spark . scheduler . JobLogger (" cloudera ", " someLogDirName ") scala > sc.addSparkListener ( logger )	Spark-Shell is able to print logs automatically.

**Table 10.2:** *Spark shell*

## SCALA programming model

Scala, the programming language, was indeed developed by *Martin Odersky*. It was first released in 2003. The name *Scala* is a portmanteau of *Scalable* and *Language*, reflecting its design goal of being a language that can scale from small scripts to large systems. Scala combines features from both object-oriented and functional programming paradigms, making it a versatile and expressive language for a wide range of software development tasks. Its concise and expressive syntax, along with strong support for concurrent and distributed programming, has made it popular in various domains, including Web development, data analysis, and more:

- Scala has a strict type system, which means you cannot create a value in Scala without initializing it with some valid value other than null

and nothing.

- Scala is a superset of Java means Scala runs on a **Java Virtual Machine (JVM)** and compiles with byte code to provide compatibility with Java programs.
- All Java programs can be ported to Scala, but not all Scala programs can be ported to Java.
- Scala consists of features of imperative programming, object-oriented programming, and functional programming, which include the beauty of crossing paradigms within a single program if needed.
- Scala is inspired by mathematics and uses math structure in syntax.
- Scala is implemented with optimization decisions aimed at reducing or eliminating potential sources of distributed processing bugs. These optimizations are designed to hide or prevent common issues that can arise when working with distributed systems and parallel processing in order to make code more robust and efficient.
- Scala enables the programmer to take control when required by an application to exchange control of scalability.
- Scala comes with existing pure functions, immutable data, implicit looping, and iteration over collection.
- Scala is more expressive than Java means 10 lines of code in Java might be equivalent to 2 lines in Scala.
- In Scala, code blocks return a value even if the value is nothing.
- While statement does not return any value.

Scala is based on a functional programming paradigm, which is often referred to as declarative programming. This approach emphasizes expressing what the program should do rather than specifying how it should be done, making it different from iterative programming, which is more focused on the step-by-step implementation details. Functional programming languages like Scala encourage developers to define operations in a more abstract and mathematical manner, which can lead to more concise and maintainable code. This shift in viewpoint allows

developers to express complex operations and transformations in a more declarative and readable fashion.

Iterative language programming focus on result, so it issues commands. If any programmer is supposed to use more than 1,000 CPUs, then he needs to require rewrite programs. However, functional programs like Scala declare their requirements without going in-depth about their structure. It does not care about how. Scala is very useful for production applications as it allows programmers to move between **Functional programming (FP)**, **Iterative Programming (IP)**, and **Object-oriented (OO)** paradigms where programmers can take control with imperatives and release control to the platform for scale.

## Features of Scala

The following are the features of Scala:

- **Inspired by math:** Functional programming languages like Scala excel at modeling computations as pure functions, where the output (result) solely depends on the input (arguments) and does not rely on external state or mutable data. This functional approach is in contrast to imperative or iterative programming, where the focus is often on changing state or executing a sequence of steps to achieve a desired outcome. Functional programming's mathematical foundation provides clarity, predictability, and a basis for reasoning about code behavior, which can lead to more robust and maintainable software. In functional programming, computations are expressed as mathematical functions, often represented as  $Y = f(X)$ . In this paradigm, for every given value of  $X$ , there is exactly one corresponding  $Y$ , and the value of  $Y$  for a specific  $X$  remains constant over time. This immutability and predictability in function evaluation are fundamental characteristics of functional programming languages.
- **Immutable and mutable:** Scala distinguishes between mutable and immutable variables and data types. Parallel collection types can be in different packages with **scala.collection.mutable** and **scala.collection.immutable**. Immutable is usually preferred

unless mutability is specifically required or Scala controls scope to limit changes to mutable data. In distributed computing, mutable data is the main cause of problems. A combination of immutable data and pure functions have side effect results in programs.

- **Implicit iteration:** Scala creates implicit variables to pass data between parts of expression and eliminate the need for counters and state variables to keep records of iterable state. It removes the main source of distributed bugs that cannot be addressed or changed because of implicit characteristics.
- **Functional programming:** Scala is implied by chaining expression over members of the collection. For example,

```
Source.fromFile (".../shops.log").foreach  
(print)
```

## Work on Scala

Scala is a versatile programming language that combines functional and object-oriented programming paradigms. To work with Scala, you will typically need to set up a development environment, write Scala code, and run it.

Work on Scala can be done in two ways:

- **Scala shell:** It can be started with the command line typing Scala. **Real Evaluate Print Loop (REPL)** immediately evaluates it.

```
scala> val a: Int = 123  
a:Int = 123  
scala> a  
res0:Int =123  
scala>println(res0)  
123
```

`:help`, `:history`, `:h? string`, `: quit`, `:sh` command, and `:load path` are examples of directive commands of Scala.

- **Scala compilation:** The main program file must contain an object, and the main file of Scala has an extension as `*.scala`, which compiles to `.class` file with `scalac program.scala` then executes the class file with Scala `ObjectName`.

For example, see *Table 10.3*:

Object ListSerial  Def main(args: Array[String]} {  println("Titanic")  println("Discovery")  println("Christmas")  }  }	ListSerial.scala  ListSerial.class
\$ scala ListSerial  >Titanic  >Discovery  >Christmas	
The preceding code is an example of a sample program.	

val s = "we are student of Big Data"  print (s)  > we are student of Big Data	Basic printing
import scala.io.Source	Basic file I/O

```

val filename: String = ".../bigdata.log"
val buffer = Source.fromFile(filename)
buffer.foreach(println)
Source.fromFile(filename).foreach(println)

```

**Table 10.3:** Scala compilation

**Program using file:** Create a new file with Scala extension in the file name in any directory. For example, **HelloScalaFirst.scala** in directory **~/scalamaterial/jes/**.

Now enter the following code:

```

object HelloScalaFirst {
    def main (args: Array[String]) {
        println ("Hello, Scala!")
    }
}

```

Save the file with the file name **HelloScalaFirst.scala** and compile it as follows:

**scalac HelloScalaFirst.scala**

This compilation code will generate **.class** file with the name of the object. Now, run the object name as follows:

**Scala HelloScalaFirst**

**Word count problem:** As always, the word count problem is like **HelloWorld** in Java since it clears all doubts of beginners. This is a code snippet in Scala that creates a Spark streaming application that listens for incoming text data on a socket, counts the occurrence of each word, and prints the results to the console.

However, if the input is in a file and needs to be loaded into the program, the following code can be used to execute the program with the desired output. First, create a directory in Hadoop and put the input file into it. Following is the sample code in Scala:

```
hadoop fs -mkdir /class  
hadoop fs -put texts /class/  
hadoop fs -ls /class
```

Then, the Spark shell is launched, and the input file is loaded:

```
spark-shell  
val file = sc.textFile("/class/texts")
```

Next, map each word to a tuple of the word and the number **1**, and then count the occurrence of each word:

```
val wc = file.flatMap(l => l.split(" ")).map(word  
=> (word,1)).cache()  
  
val w = wc.reduceByKey(_ + _)
```

Print the results to the console and save the output to a file:

```
w.collect  
val s = w.saveAsTextFile("/class/output")  
  
wc.reduceByKey(_ +  
_).saveAsTextFile("/home/Mayank_pc/output")
```

This will save the output in two different directories: **/class/output** and **/home/Mayank\_pc/output**.

## Resilient Distributed Dataset

**Resilient Distributed Dataset (RDD)** is a fundamental concept in Spark, serving as a foundational layer on top of the Spark system. RDD is responsible for managing and handling data distributed across various nodes in a cluster. This distributed data management plays a crucial role in ensuring fault tolerance within Spark. If a node fails or data becomes corrupted, Spark can recover the lost or affected data thanks to RDD, alleviating concerns about node failures. Linkage information, which is a sequence of transformations, is key to this data recovery process.

RDD is an abstract class in the Spark library, accessible both directly and indirectly. One of the reasons behind Spark's growing popularity for big data processing is its flexibility in handling data within RDD partitions. RDDs do not impose limitations on the type of data that can be stored, making them versatile for various data processing needs. While the RDD API includes many useful operations, Spark also extends this API to support key–value pairs, a preferred format for many users. This extension enhances the convenience and usability of RDDs in Spark:

- **DoubleRDDFunctions:** For aggregating numeric values, this extension is useful. The RDD can be used if its data items can be implicitly converted to the Scala data-type double.
- **PairRDDFunctions:** When the data items in an RDD have a two-component tuple structure, the extension for key–value pairs become available in Spark. In this case, Spark interprets the first tuple item as the key and the second item as its associated value.
- **OrderedRDDFunctions:** When the key is implicitly sorted, this extension is available.
- **SequenceFileRDDFunctions:** There are additional requirements for the convertibility of the tuple components to Writable types when using this extension, which contains several methods that allow users to create Hadoop sequence files from RDDs. The data items must be two-component key–value tuples as required by the **PairRDDFunctions**.

## Spark methods for data processing

Spark has automatically available methods that fulfill the requirements of users. Spark provides a wide range of methods and functions that can be used to perform various data processing and analysis tasks. These methods are available through different APIs and libraries within Spark, such as Spark Core, Spark SQL, Spark Streaming, **machine learning library (MLlib)**, and **graph processing library (GraphX)**. The following sections are the methods of Spark and its usage:

## Aggregate

This method provides an interface for customized reduction with RDD. Before its execution following points must be considered:

- Reduction and combine functions should be commutative and associative.
- Because of execution as a chain format of Spark, the output of the combiner must be equal to its input.
- There should not be any assumption execution order of partition computations or combining partitions.
- Zero value is applied at the beginning of each sequence of reduction within individual partitions.

### Example:

In the code, Apache Spark's **parallelize** and **aggregate** functions to perform an operation on an RDD. The following is a step-by-step explanation:

1. **val z = sc.parallelize(List(1, 2, 3, 4, 5, 6), 2)**: This line creates an RDD named **z** using the **parallelize** method. It takes a list of numbers **[1, 2, 3, 4, 5, 6]** and distributes them into two partitions. The second argument, **2**, specifies the number of partitions.
2. **z.aggregate(0)(math.max(\_, \_), \_ + \_)**: The **aggregate** function is used to perform an aggregation on the RDD **z**. The **aggregate** function takes three arguments:

- The initial value for aggregation, in this case, **0**.
- An aggregation function, which is **math.max(\_ , \_)**. This function computes the maximum of two values.
- A combining function, which is **\_ + \_**. This function combines the results of the aggregation.

The aggregation starts with the initial value of **0**, and then it applies the **math.max** function to find the maximum value in each partition of the RDD. Finally, it uses **\_ + \_** to combine the maximum values from different partitions.

The result of this code will be the maximum value of the numbers in the RDD **z**, which is **6**.

```
val z = sc.parallelize(List(1, 2, 3, 4, 5, 6), 2)
z.aggregate(0)(math.max(_, _), _ + _)
// Result: 9
```

```
val z = sc.parallelize(List("a", "b", "c", "d", "e", "f"),
2)
z.aggregate("")(_ + _, _ + _)
// Result: "abcdef"
```

```
z.aggregate("x")(_ + _, _ + _)
// Result: "xxdefxabc"
```

```
val z = sc.parallelize(List("12", "23", "345", "4567"), 2)
```

```
z.aggregate("")( (x, y) => math.max(x.length,  
y.length).toString, (x, y) => x + y)
```

```
// Result: "42"
```

```
z.aggregate("")( (x, y) => math.min(x.length,  
y.length).toString, (x, y) => x + y)
```

```
// Result: "11"
```

```
val z = sc.parallelize(List("12", "23", "345", ""), 2)
```

```
z.aggregate("")( (x, y) => math.min(x.length,  
y.length).toString, (x, y) => x + y)
```

```
// Result: "10"
```

## Cartesian

Each item of sequence in RDD is joined with every item of the second RDD returning with a new RDD. The use of this method may lead to memory consumption issues.

### Example:

The code creates two RDDs **x** and **y** containing lists of integers and performs a cartesian product of the two RDDs using the cartesian method. The result is an RDD containing tuples of all possible combinations of elements from the two RDDs.

```
val x = sc.parallelize(List(1, 2, 3, 4, 5))
```

```
val y = sc.parallelize(List(6, 7, 8, 9, 10))
```

```
val cartesianProduct = x.cartesian(y).collect()
```

Code, **cartesianProduct** will be an array containing all possible pairs of elements from the RDDs **x** and **y**. The size of this array can be quite large, especially if **x** and **y** have many elements, so you should be cautious when using collect on large RDDs, as it can potentially lead to out-of-memory issues if the result is too large to fit into memory.

The cartesian method performs the cartesian product of the two RDDs **x** and **y**, and the collect method returns the result as an array of tuples. The resulting RDD contains all possible combinations of elements from **x** and **y**.

## Checkpoint

Checkpoint will create a directory to store binary files using spark context.

### Example:

Checkpointing is a technique used to improve the fault tolerance of **Resilient Distributed Datasets (RDDs)** and can also be used to reduce recomputation in complex Spark workflows.

The following is a breakdown of the steps described in your code:

1. **sc.setCheckpointDir("my\_directory\_name")**: This line sets the checkpoint directory for Spark to **my\_directory\_name**. This is the location where Spark will store checkpointed data.
2. **val a = sc.parallelize(1 to 4)**: This line creates an RDD named **a** containing the values from 1 to 4 using **sc.parallelize**.
3. **a.checkpoint**: The **checkpoint** method is called on RDD **a**: This triggers the checkpointing process, which involves creating a checkpoint file to store the data. Checkpointing is an optimization for performance and fault tolerance.
4. **a.count()**: This line counts the number of elements in RDD **a**. In this case, it returns **4**, as you have described.
5. **Log output**: The log output indicates that the checkpointing was successful and that RDD 11 was checkpointed to a specific file path,

which includes the **checkpoint** directory specified earlier.

The checkpointing process in Spark helps in recovering lost data partitions in case of node failures and can also optimize execution plans by reducing the need to recompute certain RDDs. It is particularly useful for complex workflows or iterative algorithms to improve both fault tolerance and performance. The log output shows that the checkpointing was successful and that RDD 11 was checkpointed to file

```
/home/mayank/Documents/spark-0.9.0-incubating-bin-cdh4/bin/my_directory/65407913-fdc6-4ec1-82c9-48a1656b95d6/rdd-11.
```

## Repartition

In Apache Spark, the repartition method is used to change the number of partitions in RDD. Partitions are the basic units of parallelism in Spark, and the number of partitions can impact the performance and efficiency of your Spark job. The repartition method allows you to control the distribution and organization of data across partitions by specifying the desired number of partitions. It is particularly useful when you want to increase or decrease the level of parallelism or redistribute data for better load balancing.

In Apache Spark, repartitioning refers to the process of changing the number of partitions in an RDD. Repartitioning is useful for optimizing data distribution, balancing workloads, and improving performance in various Spark operations. You can use the repartition method to achieve this.

The following is how you can use repartition in Spark:

- 1. Increase the number of partitions:** If you want to increase the number of partitions in an RDD, you can use the repartition method with the desired number of partitions. This typically involves shuffling the data across the new partitions:

```
val newRDD =  
oldRDD.repartition(newNumPartitions)
```

**2. Decrease the number of partitions:** To decrease the number of partitions, you can use the **coalesce** method, which is more efficient than **repartition** when reducing the number of partitions because it avoids a full shuffle.

```
val newRDD = oldRDD.coalesce(newNumPartitions)
```

If you set the second argument of coalesce to true (**default**), it will also shuffle the data for an even distribution across partitions. Repartitioning can be useful in various scenarios, such as optimizing data for join operations, balancing workloads, and ensuring that data fits well with available resources. It is important to choose the right number of partitions based on your data size, available resources, and the specific operations you plan to perform, as an inappropriate number of partitions can impact performance.

### Example:

Following is the demonstration of how to use the coalesce method in Apache Spark to reduce the number of partitions of an RDD without shuffling the data.

```
val y = sc.parallelize(1 to 10, 10) val z =  
y.coalesce(2, false) z.partitions.length res9:Int=2
```

It reduces the number of partitions from 10 to 2. The following is a breakdown of what is happening in your code:

**1. `val y = sc.parallelize(1 to 10, 10)`:** This line creates an RDD named y containing integers from 1 to 10, and you explicitly specify that it should have 10 partitions by passing 10 as the second argument to parallelize. This means the data is initially distributed into 10 partitions.

**2. `val z = y.coalesce(2, false)`:** Here, you use the coalesce method on RDD y to reduce the number of partitions to 2 without shuffling the data (**false** as the second argument). The **coalesce** method can be used to either reduce or increase the number of

partitions in an RDD and by setting the shuffle flag to false, you ensure that it does not trigger data shuffling.

3. **`z.partitions.length`**: This line retrieves the number of partitions in RDD `z` after the coalesce operation, and it correctly returns **2**.

The coalesce method is useful for optimizing the number of partitions in an RDD to match the actual needs of your computation. It can be used to reduce the overhead of managing a large number of partitions or to repartition data for more efficient processing.

## Cogroup

In Apache Spark, the **cogroup** transformation is used to group elements from multiple RDDs together based on a common key, much like a SQL **join** operation. It is used to combine data from two or more RDDs that have key–value pairs, where the keys are used to perform the grouping.

The cogroup transformation takes multiple RDDs as input and groups the elements with the same key into a tuple. The resulting RDD contains key–value pairs, where the key is the common key, and the value is a tuple containing all the values from the input RDDs associated with that key.

The following is a basic example of how to use cogroup:

```
val rdd1 = sc.parallelize(Seq(("A", 1), ("B", 2),  
("A", 3)))
```

```
val rdd2 = sc.parallelize(Seq(("A", "apple"),  
("B", "banana")))
```

```
val result = rdd1.cogroup(rdd2)
```

```
result.collect().foreach(println)
```

The **result** RDD in this example will contain key–value pairs like the following:

```
("A", (ArrayBuffer(1, 3), ArrayBuffer("apple")))
("B", (ArrayBuffer(2), ArrayBuffer("banana")))
```

The values are grouped as arrays within the tuple, with each array containing values from the respective RDDs associated with the common key. You can then perform further operations on this resulting RDD to process or join the grouped data.

**cogroup** is particularly useful when you need to combine data from different RDDs based on a common key, such as performing joins or aggregations on datasets with related information.

### Example:

```
1. val a = sc.parallelize(List(1, 2, 1, 3), 1)
  val b = a.map((_, "b"))
  val c = a.map((_, "c"))
  b.cogroup(c).collect
  res7: Array[(Int, (Seq[String], Seq[String]))]
= Array((2,(ArrayBuffer(b),ArrayBuffer(c))), 
  (3,(ArrayBuffer(b),ArrayBuffer(c))), (1,
  (ArrayBuffer(b, b),ArrayBuffer(c, c))))
```

```
2. val d = a.map((_, "d"))
  b.cogroup(c, d).collect
  res9: Array[(Int, (Seq[String], Seq[String],
  Seq[String]))] = Array((2,
  (ArrayBuffer(b),ArrayBuffer(c),ArrayBuffer(d))),
  (3,
  (ArrayBuffer(b),ArrayBuffer(c),ArrayBuffer(d))))
```

```

, (1,(ArrayBuffer(b, b),ArrayBuffer(c,
c),ArrayBuffer(d, d)))))

3. val x = sc.parallelize(List((1, "apple"), (2,
"banana"), (3, "orange"), (4, "kiwi")), 2)
val y = sc.parallelize(List((5, "computer"),
(1, "laptop"), (1, "desktop"), (4, "iPad")), 2)
x.cogroup(y).collect
res23: Array[(Int, (Seq[String], Seq[String]))]
= Array((4,
(ArrayBuffer(kiwi),ArrayBuffer(iPad))), (2,
(ArrayBuffer(banana),ArrayBuffer())), (3,
(ArrayBuffer(orange),ArrayBuffer()))), (1,
(ArrayBuffer(apple),ArrayBuffer(laptop,
desktop))), (5,
(ArrayBuffer(),ArrayBuffer(computer))))

```

## Collect

This is applied to values before inserting in the result array.

### Example:

```

val c = sc.parallelize(List("Gnu", "Cat", "Rat",
"Dog", "Gnu", "Rat"), 2)
c.collect

```

The output is as follows:

```
res29: Array[String] = Array(Gnu, Cat, Rat, Dog, Gnu, Rat)
```

## CollectAsMap

It works similarly to the **collect** method, but it preserves key-value structure as Scala maps.

### **Example:**

```
val data = List(1, 2, 1, 3)  
val rdd = sc.parallelize(data, 1)  
val pairedRDD = rdd.zip(rdd)  
val resultMap = pairedRDD.collectAsMap()  
  
println(resultMap)
```

This code first defines a list of integers `data` and then creates an RDD `rdd` from it with a single partition. The `zip` transformation is applied to `rdd` to create a new RDD `pairedRDD` containing tuples of corresponding elements from `rdd`. Finally, the `collectAsMap` action is applied on `pairedRDD`, which returns a map where the first element of each tuple is used as a key and the second element as a value. The map is then printed to the console.

## **CombineByKey**

It combines the values of RDD consisting of two component tuples.

### **Example:**

```
val a = sc.parallelize(List("dog", "cat", "gnu",  
"salmon", "rabbit", "turkey", "wolf", "bear",  
"bee"), 3)  
  
val b = sc.parallelize(List(1, 1, 2, 2, 2, 1, 2,  
2, 2), 3)  
  
val c = b.zip(a)  
  
val d = c.combineByKey(  
    List(_),
```

```
(x: List[String], y: String) => y :: x,  
(x: List[String], y: List[String]) => x :: y  
)  
d.collect
```

The following will be the result:

```
res16: Array[(Int, List[String])] = Array(  
  (1, List("cat", "dog", "turkey")),  
  (2, List("gnu", "rabbit", "salmon", "bee",  
"bear", "wolf"))  
)
```

## Compute

It is not called directly by the user, and it executes the actual representation of RDD.

### Example:

```
def compute ( split : Partition , context :  
TaskContext ): Iterator [T]
```

## Count

It returns stored items within RDD.

### Example:

```
val c = sc. parallelize ( List (" Gnu ", "Cat ",  
"Rat ", "Dog ") , 2)  
  
c. count  
  
res2 : Long = 4
```

## CountByKey

Similar to count, it also counts values of RDD consisting of two components for each distinct key.

### Example:

```
val c = sc.parallelize ( List ((6 , "Gnu ") , (7, "Yak ") , (8, " Mouse ") , (9, " Dog ")), 1)  
c. countByKey res3 : scala . collection . Map[Int , Long ] = Map (4 -> 4, 6 -> 2)
```

This code snippet creates a Spark RDD called **c** using the **parallelize** method. The RDD c is created by passing a list of tuples containing an **integer** and a **string**, and a number indicating the number of partitions in which to split the RDD.

The **countByKey** method is then called on the RDD to count the occurrences of each key (the integers in the tuples). The result of **countByKey** is a map with the key–value pairs, where the key is the integer from the original tuples, and the value is the count of occurrences of that key in the RDD.

To answer your question, the phrase it is applied before inserting values into the result array is not applicable in this context, as there is no result array being created or modified in this code snippet. The **countByKey** method returns a map, not an array.

## CountByValue

Return a map that contains all unique values of RDD.

### Example:

```
val b = sc.parallelize(List(1, 2, 3, 4, 5, 6, 7, 8, 2, 4, 2, 1, 1, 1, 1))  
val countByValue = b.countByValue()
```

```
countByValue.foreach(println)
```

The following will be the result:

(5,1)

(8,1)

(3,1)

(6,1)

(1,6)

(2,3)

(4,2)

(7,1)

## countApproxDistinct

It computes distinct values of approximate numbers.

### Example:

These commands compute the approximate number of distinct elements in an RDD b using the **HyperLogLog** algorithm with different error tolerances (that is, relative standard error):

- **b.countApproxDistinct(0.1)** returns an approximate count with a 10% error tolerance, which is **10784**.
- **b.countApproxDistinct(0.05)** returns an approximate count with a 5% error tolerance, which is **11055**.
- **b.countApproxDistinct(0.01)** returns an approximate count with a 1% error tolerance, which is **10040**.
- **b.countApproxDistinct(0.001)** returns an approximate count with a 0.1% error tolerance, which is **10001**.

Note that the approximate count may not be exact, and the error tolerance depends on the given parameter to the **countApproxDistinct** method.

```
val data = sc.parallelize(1 to 10000, 20)

val dataRepeated = data ++ data ++ data ++ data ++
data

dataRepeated.countApproxDistinct(0.1)

res14: Long = 10784

dataRepeated.countApproxDistinct(0.05)

res15: Long = 11055

dataRepeated.countApproxDistinct(0.01)

res16: Long = 10040

dataRepeated.countApproxDistinct(0.001)

res0: Long = 10001
```

## Dependencies

It returns the resilient distributed dataset on which it depends.

### Example:

```
// Create an RDD b

val b = sc.parallelize(List(1, 2, 3, 4, 5, 6, 7,
8, 2, 4, 2, 1, 1, 1, 1))

// Check the number of dependencies of b

b.dependencies.length
```

```
res40: Int = 0

// Create a new RDD by mapping each element of b
to itself and check the number of dependencies
b.map(a => a).dependencies.length

res41: Int = 1

// Create a cartesian product of b and another RDD
and check the number of dependencies and the
dependencies themselves
b.cartesian(a).dependencies.length

res42: Int = 2

b.cartesian(a).dependencies

res43: Seq[org.apache.spark.Dependency[_]] =
List(org.apache.spark.rdd.CartesianRDD$$anon$1@e62
be3f,
org.apache.spark.rdd.CartesianRDD$$anon$2@4bdfc1ef
)
```

## Distinct

It returns a new RDD that contains a unique value once.

### Example:

```
val c = sc.parallelize(List("Gnu", "Cat", "Rat",
"Dog", "Gnu", "Rat"), 2)

val distinctC = c.distinct.collect
```

```
// distinctC: Array[String] = Array(Dog, Gnu, Cat,  
Rat)  
  
val a = sc.parallelize(List(1, 2, 3, 4, 5, 6, 7,  
8, 9, 10))  
  
val distinctPartitions2 =  
a.distinct(2).partitions.length  
// distinctPartitions2: Int = 2  
  
val distinctPartitions3 =  
a.distinct(3).partitions.length  
// distinctPartitions3: Int = 3
```

## first

It returns the first data item of RDD.

### Example:

```
val c = sc.parallelize(List("Gnu", "Cat", "Rat",  
"Dog"), 2)  
  
c.first()
```

The following will be the output:

```
res1: String = Gnu
```

## filter and filterWith

In Apache Spark, there is no built-in **filterWith** method. It appears you might be referring to the **filter** operation, which is a common transformation operation in Spark.

## filter Transformation

The **filter** transformation is used to create a new RDD by selecting elements from the original RDD that satisfy a given condition. It takes a function (usually a lambda function) as an argument, and this function should return a Boolean value to indicate whether an element should be included in the new RDD.

The following is the basic syntax of the **filter** operation:

```
val filteredRDD = originalRDD.filter(element =>  
    condition(element))
```

Example of using filter to create a new RDD with even numbers:

```
val numbersRDD = sc.parallelize(1 to 10)  
  
val evenNumbersRDD = numbersRDD.filter(num => num  
    % 2 == 0)
```

In this example, **evenNumbersRDD** will contain only the even numbers from the **numbersRDD**.

## fold

The aggregate values of each partition with its initialization as value zero.

### Example:

```
val a = sc.parallelize ( List (1 ,2 ,3) , 3)  
  
a. fold (0)(_ + _)  
  
res59 : Int = 6
```

## foreach

It performs function for each data item.

### Example:

```
val c = sc.parallelize(List("cat", "dog", "tiger",
"lion", "gnu", "crocodile", "ant", "whale",
"dolphin", "spider"), 3)

c.foreach(x => println(x.capitalize + "s are
yummy"))
```

The following will be the result:

Cats are yummy

Dogs are yummy

Tigers are yummy

Lions are yummy

Gnus are yummy

Crocodiles are yummy

Ants are yummy

Whales are yummy

Dolphins are yummy

Spiders are yummy

## **getStorageLevel**

It retrieves the current storage level of RDD. In case there is no such level assigned, then it can assign a new storage level.

### **Example:**

The code is attempting to change the storage level of an RDD after it has already been assigned a level, which is not allowed. The error message **java.lang.UnsupportedOperationException** cannot change the

storage level of an RDD after it was already assigned a level that indicates that the operation is not supported.

Note that calling the **persist()** method with **DISK\_ONLY** storage level will cache the RDD to disk but will not change its storage level. If you want to change the storage level of an RDD, you need to do it before caching it:

```
val a = sc.parallelize(1 to 200000, 3)

a.persist(org.apache.spark.storage.StorageLevel.DI
SK_ONLY)

println(a.getStorageLevel.description)

// Output: Disk Serialized 1x Replicated

// This will raise an exception because the
// storage level of `a` has already been set
// and cannot be changed

// a.cache()

// java.lang.UnsupportedOperationException: Cannot
// change storage level of an RDD after it was
// already assigned a level
```

## groupBy

It collects all required items as one record.

### Example:

- Grouping RDD elements by even and odd numbers:

```
val a = sc.parallelize(1 to 9, 3)

a.groupBy(x => if (x % 2 == 0) "even" else
"odd").collect
```

```
// res: Array[(String, Seq[Int])] =  
Array(("even", ArrayBuffer(2, 4, 6, 8)),  
("odd", ArrayBuffer(1, 3, 5, 7, 9)))
```

- Grouping RDD elements by the result of a user-defined function:

```
val a = sc.parallelize(1 to 9, 3)  
  
def myfunc(a: Int): Int = a % 2  
  
a.groupBy(myfunc).collect  
  
// res: Array[(Int, Seq[Int])] = Array((0,  
ArrayBuffer(2, 4, 6, 8)), (1, ArrayBuffer(1,  
3, 5, 7, 9)))
```

- Grouping RDD elements by the result of a user-defined function using a custom partitioner:

```
val a = sc.parallelize(1 to 9, 3)  
  
def myfunc(a: Int): Int = a % 2  
  
a.groupBy(x => myfunc(x), new  
MyPartitioner()).collect  
  
// res: Array[(Int, Seq[Int])] = Array((0,  
ArrayBuffer(2, 4, 6, 8)), (1, ArrayBuffer(1,  
3, 5, 7, 9)))
```

- Grouping RDD elements by their keys using a custom partitioner and then swapping the key–value pairs:

```
import org.apache.spark.Partitioner  
  
class MyPartitioner extends Partitioner {  
  
    def numPartitions: Int = 2
```

```
def getPartition(key: Any): Int = key match
{
    case null => 0
    case key: Int => key % numPartitions
    case _ => key.hashCode % numPartitions
}

override def equals(other: Any): Boolean =
other match {
    case h: MyPartitioner => true
    case _ => false
}

}

val a = sc.parallelize(1 to 9, 3)
val p = new MyPartitioner()
val b = a.groupBy((x: Int) => x, p)
val c = b.mapWith(i => i)((a, b) => (b, a))
c.collect

// res: Array[(Int, (Int, Seq[Int]))] =
Array((0, (4, ArrayBuffer(4))), (0, (2,
ArrayBuffer(2))), (0, (6, ArrayBuffer(6))),
(0, (8, ArrayBuffer(8))), (1, (9,
ArrayBuffer(9))), (1, (3, ArrayBuffer(3))))
```

```
(1, (1, ArrayBuffer(1))), (1, (7, ArrayBuffer(7))), (1, (5, ArrayBuffer(5))))
```

## Histogram

This function creates a histogram based on bucket boundaries supplied by users with an array of double values. It can provide histograms with even spacing and arbitrary spacing.

### Example:

```
val a = sc.parallelize(List(1.1, 1.2, 1.3, 2.0, 2.1, 7.4, 7.5, 7.6, 8.8, 9.0), 3)

val numBuckets = 5

a.histogram(numBuckets)

res11: (Array[Double], Array[Long]) = (Array(1.1, 2.68, 4.26, 5.84, 7.42, 9.0), Array(5, 0, 0, 1, 4))
```

```
val a = sc.parallelize(List(9.1, 1.0, 1.2, 2.1, 1.3, 5.0, 2.0, 2.1, 7.4, 7.5, 7.6, 8.8, 10.0, 8.9, 5.5), 3)

val numBuckets = 6

a.histogram(numBuckets)

res18: (Array[Double], Array[Long]) = (Array(1.0, 2.5, 4.0, 5.5, 7.0, 8.5, 10.0), Array(6, 0, 1, 1, 3, 4))
```

## id

It retrieves ID, which is assigned to RDD.

**Example:**

```
val y = sc.parallelize (1 to 10, 10)  
y.id  
res16 : int = 19
```

**join**

Using key–value pair, it performs inner join RDDs.

**Example:**

```
val a = sc.parallelize(List("dog", "salmon",  
"salmon", "rat", "elephant"), 3)  
  
val b = a.keyBy(_.length)  
  
val c = sc.parallelize(List("dog", "cat", "gnu",  
"salmon", "rabbit", "turkey", "wolf", "bear",  
"bee"), 3)  
  
val d = c.keyBy(_.length)  
  
b.join(d).collect  
  
res17: Array[(Int, (String, String))] = Array((6,  
("salmon", "salmon")), (6, ("salmon", "rabbit")),  
(6, ("salmon", "turkey")), (6, ("rabbit",  
"salmon")), (6, ("rabbit", "rabbit")), (6,  
("rabbit", "turkey")), (6, ("turkey", "salmon")),  
(6, ("turkey", "rabbit")), (6, ("turkey",  
"turkey")), (3, ("dog", "dog")), (3, ("dog",  
"cat")), (3, ("dog", "gnu")), (3, ("dog", "bee")),  
(3, ("cat", "dog")), (3, ("cat", "cat")), (3,  
("cat", "gnu")), (3, ("cat", "bee")), (3, ("gnu",  
"dog")), (3, ("gnu", "cat")), (3, ("gnu", "gnu"))),
```

```
(3, ("gnu", "bee")), (3, ("bee", "dog")), (3, ("bee", "cat")), (3, ("bee", "gnu")), (3, ("bee", "bee")), (4, ("wolf", "wolf")), (4, ("wolf", "bear")), (4, ("bear", "wolf")), (4, ("bear", "bear")))
```

## leftOuterJoin

Using two key–value, it performs left outer join.

### Example:

```
val animals = sc.parallelize(List("dog", "salmon", "salmon", "rat", "elephant"), 3)

val animalsByLength = animals.keyBy(_.length)

val fruits = sc.parallelize(List("dog", "cat", "gnu", "salmon", "rabbit", "turkey", "wolf", "bear", "bee"), 3)

val fruitsByLength = fruits.keyBy(_.length)

animalsByLength.leftOuterJoin(fruitsByLength).collect
```

The code takes two RDDs, **animals** and **fruits**, key them by the length of their strings using **keyBy**, and then performs a left outer join using **leftOuterJoin**. The resulting RDD contains tuples of the form **(Int, (String, Option[String]))**, where the first element is the length of the string, the second element is the string itself from the **animals** RDD, and the third element is an **Option** containing the matching string from the **fruits** RDD, or none if no match was found. The **collect** method is used to return the results to the driver program.

## Example of programs using Scala

The following are a few simple examples of programs written in Scala:

- **Hello, World! Program:**

```
object HelloWorld {  
    def main(args: Array[String]): Unit = {  
        println("Hello, World!")  
    }  
}
```

This is the classic **Hello, World!** program in Scala. It defines an object called **HelloWorld** with a **main** method that prints **Hello, World!** to the console.

- **Basic arithmetic operations:**

```
object ArithmeticOperations {  
    def main(args: Array[String]): Unit = {  
        val a = 10  
        val b = 5  
        val sum = a + b  
        val difference = a - b  
        val product = a * b  
        val quotient = a / b  
  
        println(s"Sum: $sum")  
        println(s"Difference: $difference")  
        println(s"Product: $product")  
    }  
}
```

```
    println(s"Quotient: $quotient")  
}  
}
```

This program performs basic arithmetic operations (addition, subtraction, multiplication, and division) and prints the results.

- **Fibonacci sequence:**

```
object Fibonacci {  
  
  def main(args: Array[String]): Unit = {  
  
    val n = 10  
  
    var a = 0  
  
    var b = 1  
  
    print(s"Fibonacci Series (first $n terms):  
$a, $b")  
  
  
    for (_ <- 3 to n) {  
  
      val next = a + b  
  
      print(s", $next")  
  
      a = b  
  
      b = next  
    }  
  
    println()  
  }  
}
```

```
    }  
}  
}
```

This program generates and prints the Fibonacci sequence for the first “n” terms.

- **Factorial calculation:**

```
object Factorial {  
  
    def factorial(n: Int): BigInt = {  
  
        if (n <= 1) 1  
        else n * factorial(n - 1)  
  
    }  
  
  
    def main(args: Array[String]): Unit = {  
  
        val n = 10  
        val result = factorial(n)  
        println(s"Factorial of $n is $result")  
  
    }  
}
```

This program calculates the factorial of a given number using a recursive function.

These are simple examples to get you started with Scala programming. Scala is a versatile language used for various applications, from Web development to big data processing. You can explore more complex projects and libraries as you become more familiar with Scala’s features and capabilities.

An implementation of the PageRank algorithm in Scala using Apache Spark. PageRank is an algorithm used by search engines to rank Web pages in their search results. It measures the importance of Web pages based on the structure of hyperlinks between them.

A list of pages and their ranks are given, and there is a need to calculate page contribution to its neighbors:

```
def computeContribs(neighbors: Iterable[String],  
rank: Double): Iterable[(String,Double)] = {
```

```
    for (neighbor <- neighbors) yield(neighbor,  
rank/neighbors.size)
```

```
}
```

```
// read in a file of link pairs (format: url1 url2)
```

```
val linkfile =  
"file:/home/scalametricals/data/pagelinks.txt"
```

```
val links = sc.textFile(linkfile).map(_.split("")).
```

```
map(pages => (pages(0),pages(1))).
```

```
distinct().groupByKey().persist()
```

```
// create initial page ranges of 1.0 for each
```

```
var ranks=links.map(pair => (pair._1,1.0))
```

```
// number of iterations
```

```

val n = 10

// for n iterations, calculate new page ranks based on
neighbor contributions

for (x <- 1 to n) {
    var contribs = links.join(ranks).flatMap(pair =>
computeContribs(pair._2._1,pair._2._2))

ranks=contribs.

    reduceByKey(_+_).

    map(pair => (pair._1,pair._2 * 0.85 + 0.15))

    println("Iteration " + x)

    for (rankpair <- ranks.take(10))
    println(rankpair)
}

}

```

**Reduce** and **Reducebykey** with an example:

```

val mylist = List(1,2,3,4,5,10)

mylist.reduce((a,b)=> a + B)

```

**o/p: 25**

Reduce gives the commutative output of the list, whereas **reducebykey** gives commutative output using the key:

```

val mydata = sc.textFile("mayank/files/codeblock")

mydata.collect

```

**o/p:**

```
Array[String] = Array(APACHE SPARK is a  
replacement of map reduce PROCESSING., PACHE SPARK  
is a replacement of map reduce PROCESSING., ACHE  
SPARK is a replacement of map reduce PROCESSING.,  
HE SPARK is a replacement of map reduce  
PROCESSING.)
```

```
mydata.flatMap(_.split(" "  
)).map((_,1)).reduceByKey(_+_).collect()
```

```
Array[(String, Int)] = Array((is,4),  
(replacement,4), (a,4), (SPARK,4), (HE,1),  
(PROCESSING.,4), (ACHE,1), (map,4), (of,4),  
(APACHE,1), (PACHE,1), (reduce,4))
```

The user interacts with the Spark shell, which includes in driver process of Spark applications, and executor processes are distributed across nodes on the cluster. The driver needs to take care of all applications that need to be executed as high-level control flow, whereas the executor process is responsible for executing work in the form of a task and also responsible for choosing data for cache. Driver and executor both run continuously till the application keeps executing. An executor can run many tasks concurrently throughout its lifespan.

## Shuffling

Shuffling in Scala, particularly in the context of Apache Spark, refers to the process of redistributing and reorganizing data across different partitions. It often occurs when data needs to be grouped, sorted, or joined in a way that

requires reorganizing the data in a new way or across different partitions. Shuffling is an expensive operation in terms of time and network I/O, and it should be minimized when designing Spark applications to improve performance.

Shuffling typically happens in the following scenarios:

- **Grouping by key:** When you use operations like **groupByKey** or **reduceByKey** in Spark, it can trigger shuffling because it needs to move data with the same key to the same partition. Shuffling can be minimized using **reduceByKey** when possible, as it performs a local reduction before shuffling.
- **Joins:** When you perform joins using operations such as **join**, **cogroup**, or **joinWith**, Spark may need to shuffle data to ensure that records with matching keys are on the same partitions.
- **Aggregations:** Operations like **reduce** and **aggregateByKey** can also cause shuffling when aggregating data across partitions.

Minimizing shuffling is important for optimizing the performance of Spark applications. To reduce shuffling, you can consider the following strategies:

- Use operations such as **reduceByKey**, **combineByKey**, and **aggregateByKey** instead of **groupByKey** to perform local aggregation before shuffling.
- Use broadcast joins when one of the **DataFrames** being joined is small enough to fit in memory across all nodes.
- Use operations such as **map**, **filter**, and **flatMap** before performing shuffling operations to reduce the amount of data to be shuffled.
- Cache or persist intermediate RDDs to avoid recomputation and shuffling.

Understanding and optimizing shuffling in your Spark application is crucial for achieving good performance and efficient resource utilization.

Jobs are on top of the execution model. Spark application executes the launch of the Spark job by invoking it using triggers. Graph of RDDs

decides what this job looks like that action depends on and formulates an action plan with computing farthest back RDDs and cumulate in RDDs to produce a result. Execution plans consist of several stages that correspond to a collection of tasks that execute the same code with different partitions of data. Every stage consists of a sequence of transformations that can be completed without shuffling full data.

The following code would execute in a single stage because none of the outputs of these three operations depend on data that comes from different partitions than their inputs:

```
sc.textFile("someFile.txt")
map(mapFunc)
map(mapFunc)
flatMap(flatMapFunc)
filter(filterFunc)
count()
```

If there is a need to **count** character in appears in a text file, then the following code will provide the desired output by breaking a file into stages because computing their output repartitioning data by keys:

```
val tokenized =
sc.textFile(args(0)).flatMap(_.split(' '))
val wordCounts = tokenized.map(word => (word,
1)).reduceByKey(_ + _)
val filtered = wordCounts.filter { case (_, count) => count >= 1000 }
val charCounts = filtered.flatMap { case (word, _) => word.toCharArray.map(char => (char, 1)) }
.reduceByKey(_ + _)
```

## `charCounts.collect()`

The code reads in a text file, splits each line into words, and counts the occurrence of each word. It then filters out any words that appear less than 1,000 times and counts the occurrence of each character in the remaining words. Finally, it collects and returns the character counts.

There may be differences in data partition in the parent stage and child stage, so stage boundaries can be avoided when possible. **numPartitions** argument determines how many partitions to split the data into child stages. Counting the number of reducers can be useful to get to know about MapReduce jobs, which can break application performance. The amount of data decides the slowness of the system; if too much data is imposed, it results in affecting the performance of the system. A large number of partitions will decide to increase overhead in tasks on the parent side when records are being sorted.

## Common Spark memory issues

Memory-related issues are common in Apache Spark applications, as Spark heavily relies on memory for data processing and caching. The following are some common memory-related issues in Spark and ways to mitigate them:

- **OutOfMemoryError:** This is a common issue in Spark when tasks or executors run out of memory. It can occur during data shuffling, caching, or when working with large datasets. To mitigate this issue:
  - Monitor and adjust the memory configurations for Spark, such as **spark.driver.memory** and **spark.executor.memory**.
  - Tune the memory fractions for on-heap and off-heap memory (for example, **spark.memory.fraction** and **spark.memory.offHeap.size**).
  - Reduce the dataset size by filtering or sampling if possible.
  - Optimize data storage and retrieval with strategies like repartitioning, caching, and avoiding unnecessary transformations.

- **Garbage Collection (GC) overhead:** Frequent garbage collection can lead to performance degradation. To address this issue:
  - Increase the memory allocated for the Java heap (for example, `spark.driver.memory`, `spark.executor.memory`).
  - Configure garbage collection settings (for example, `spark.executor.extraJavaOptions`) to optimize GC for your workload.
  - Use off-heap storage (if available) to reduce GC overhead.
- **Data serialization and deserialization:** Inefficient data serialization, especially when working with non-native formats, can lead to memory and performance issues. You can:
  - Choose appropriate serialization formats like Kryo or Avro.
  - Use columnar storage formats like Parquet for optimized data storage and retrieval.
- **Skewed data:** Skewed data distributions can cause memory and processing imbalances across partitions. Solutions include the following:
  - Preprocessing or resampling data to reduce skew.
  - Using techniques like data skew join handling, custom partitioning, and bucketing.
- **Memory leaks:** Long-running Spark applications can sometimes experience memory leaks, especially when using external libraries or user-defined functions. To avoid memory leaks:
  - Regularly profile and monitor memory usage.
  - Ensure that resources are released after use in UDFs or custom code.
  - Restart the application periodically to free up resources.
- **Distributed cache:** Loading too much data into memory can lead to issues. Consider:

- Managing data sizes and evicting least-used data from the cache.
- Using cache invalidation strategies to remove outdated or unneeded data from memory.
- **Data serialization and deserialization:** Inefficient data serialization, especially when working with non-native formats, can lead to memory and performance issues. You can:
  - Choose appropriate serialization formats like Kryo or Avro.
  - Use columnar storage formats like Parquet for optimized data storage and retrieval.
- **Driver memory:** In certain cases, insufficient driver memory can lead to issues when collecting or running actions. Ensure that **spark.driver.memory** is appropriately configured to handle the driver's memory requirements.

Monitoring, profiling, and careful tuning of memory settings are essential to address and prevent these common memory-related issues in Spark applications. Additionally, understanding your specific workload and data characteristics is key to effective memory management in Spark.

## Conclusion

The combination of Apache Spark and the Scala programming language offers a powerful and versatile platform for large-scale data processing and analytics. Spark's distributed computing framework provides speed, scalability, and fault tolerance, making it a preferred choice for organizations dealing with big data. Scala, as the language of choice for Spark development, brings its functional programming capabilities, concise syntax, and compatibility with the Java ecosystem, allowing developers to write efficient and expressive code.

Together, Spark and Scala form a dynamic duo for tackling complex data challenges, from batch processing and real-time streaming to machine learning and graph analytics. With the ability to seamlessly integrate with various data sources and libraries, this pairing provides a comprehensive

solution for data professionals and engineers. As the field of big data continues to evolve, Spark and Scala remain essential tools for harnessing the potential of large datasets and driving innovation in diverse industries, reaffirming their significance in the modern data landscape.

## **Join our book's Discord space**

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



# Index

## A

---

Amazon Machine Image (AMI) 395  
Amazon Prime Video 25  
Amazon Web Service (AWS) 394, 395  
    Hadoop, setting up on EC2 397  
    setting up 395, 396  
Apache Avro 145, 146  
Apache CouchDB 41, 42  
    distribution and replication 44-46  
    functioning 42  
    versioning 44  
    views 43, 44  
Apache Hadoop 10  
Apache PigLatin 308, 309  
    execution types 311  
    installation 309, 310  
    local mode 312  
    MapReduce mode 312  
Apache Spark 403  
Application Programming Interfaces (APIs) 14  
Avro 14  
    Avro file-based data structure 145  
    considerations 146  
    data type and schemas 146-148  
    deserialization 149, 150  
    serialization 149, 150  
Avro MapReduce 150, 154

## B

---

.bashrc file 177  
big data 3, 4  
    3V's 6, 7  
    characteristics 5  
    features 7-10  
    industry examples 20, 21  
    versus, traditional techniques of databases 5  
big data convergence  
    big data, versus other techniques 16-18  
    into business 15, 16

key trends 15  
big data tools 20  
big data, usages  
  algorithm trading 31-33  
  big data and marketing 27  
  entertainment fields 25, 26  
  fraud detection 28-30  
  health care 33  
  machine learning tools 24, 25  
  risk management, with credit card 30, 31  
  web analytics 26, 27  
Bigtable 57-59  
  client library 60  
  components 59  
  master server lifecycle 61  
  master servers 60  
  Tablet lifecycle 60  
BigTable derivatives  
  Cassandra 63  
  Facebook 63  
  HBase 63  
  Hypertable 62, 63  
bloom filter 375  
  application 381-383  
  function 376, 377  
  implementation, in MapReduce 383, 394  
  using, in MapReduce 375, 376  
  working 378-381  
business intelligence software 20  
business-to-consumer (B2C) 4

## C

---

CAP theorem 72  
Cassandra 285, 286  
  CAP theorem 286, 287  
  characteristics 288  
  CLI commands 290, 291  
  Hadoop integration 303  
  installing 288-290  
  in terms of intersection points 287  
  use cases 303  
Cassandra client 299, 300  
  Avro 302  
  Hector 303  
  Thrift 300, 302

Cassandra data model 291, 292  
clusters 293  
column families 294  
keyspaces 293  
super column family 292, 293  
super columns 294, 295  
Cassandra examples 295  
batch executing 298  
current cluster, describing 298  
entire row, deleting 298  
keyspace, altering 296  
keyspace, creating 296  
keyspace, dropping 296  
primary key 297  
table, altering 297  
table, creating 296  
table, truncating 298  
Classless Inter Domain Routing (CIDR) 383  
cloud computing  
versus big data 18  
column-family database 56-59  
components 60, 61  
combiner 96, 97  
situation 102  
uses 97, 98, 102  
Comma Separated Values (CSV) 345  
compression 139, 140  
and input splits 141-143  
codecs 140, 141  
map output 143  
consistency issues, in NoSQL 69, 70  
ACID, versus BASE 70, 71  
relaxing consistency 72, 73  
Content Delivery Networks (CDNs) 26

## D

---

data  
analyzing, with Hadoop 112, 115  
data analysis  
document archived from NY Times 398  
examples 397  
data analysis, with X 364  
flume, using 364-371  
MapReduce, using 372, 375  
databases, in NoSQL

column-family database 56-59  
document database 40, 41  
graph database 64  
key-value database 37-40  
Data Definition Language (DDL) 351  
data distribution 109  
data flow 132  
    cluster balance 137  
    coherency model 136  
    file read 133, 134  
    file write 134, 135  
data format 110, 111  
data integration tools 20  
data locality 17  
data mining  
    in mobiles 399  
DataNodes 201  
denormalizing 67  
Describe operator 335  
Directed Acyclic Graph (DAG) 14, 231, 403  
document database 40, 41  
    Apache CouchDB 41-46  
    MongoDB 46-49  
document management systems 20  
Dump operator 335  
Dynamo 37, 38

## E

---

Eclipse 326  
Elastic Compute Cloud (EC2) 398  
enterprise content management systems 20  
Explanation operator 336

## F

---

file formats, Hive  
    ORC file 349  
    RC file 347, 348  
    sequence file 346, 347  
    text file 345, 346  
Filesystem in Userspace (FUSE) 131  
flume 15, 364  
    components 365  
    using 364-371  
fully distributed mode, Hadoop 201, 202  
    multi-node cluster, configuring 202-210

multi-node cluster, installing 202  
Functional programming (FP) 411

## G

---

Geographic Information Systems (GIS) 3  
Global Positioning Systems (GPS) 3  
Google Bigtable 57  
    table representation 61, 62  
Google File System (GFS) 55, 61  
graph database 64  
    Neo4j 65  
graphics process unit (GPU) analytics 31  
graph processing library (GraphX) 415  
GraphQL 65  
    characteristics 65  
grid computing  
    versus big data 17  
GridFS 49

## H

---

Hadoop 10  
    data, analyzing with 112-115  
    driver class, with no reducer 119  
    history 11, 12  
    name 12  
    reducers 117, 118  
    scale-in, versus scale-out 116, 117  
    streaming 120  
        streaming, in Java 122  
        streaming, in Python 121  
        streaming, in Ruby 120, 121  
Hadoop Archives (HAR) 137  
Hadoop diagnosis 399  
    DataNode, removing 402  
    quotas, managing 401  
    setting permission 401  
    system health 399, 400  
    trash, enabling 401  
Hadoop Distributed File System (HDFS) 13, 109, 125, 405  
    all-time availability 128  
    blocks 126, 127  
    DataNode 127  
    group 128  
    Hadoop file system 129  
    NameNode 127

Hadoop ecosystem 12, 13  
Avro 14  
Flume 15  
HBase 13  
HCatalog 14  
HDFS 13  
Hive 14  
Mahout 14  
Oozie 14  
Pig 14  
ZooKeeper 14

Hadoop installation  
fully distributed mode 201, 202  
on Ubuntu 16.04 183-200  
Oracle VirtualBox, using 168-182  
pseudo-distributed mode 159, 160  
standalone (local) mode, using 158-160  
Ubuntu, using 160-168  
VmWare 160

Hadoop I/O  
data integrity 138  
local file system 138, 139

Hadoop pipes 123, 124

HBase 13, 73, 253  
challenges 283-285  
components 80-83  
conceptual architecture 256  
data structure 77  
examples and commands 264-270  
features 259  
HBase shell command 83  
history 77  
implementation 258  
installation 73-77, 255  
locking 257, 258  
locking mechanism 87, 88  
Master Server 257  
physical storage 78-80  
region 87, 256  
RegionServers 257  
scan command, using 86  
stopping 282, 283  
terminologies 86  
version stamp 86, 87  
versus RDBMS 259, 260

HBase client 261

Class Delete 263  
Class Get 262  
Class HTable 261  
Class Put 262  
Class Result 263  
HBase shell command 83-86  
HBase, with Java APIs  
  column family, adding 276, 277  
  column family, deleting 277, 278  
  table, creating 271-273  
  table, deleting 280  
  table, disabling 275-281  
  table existence, verifying 279  
  table, listing 273, 274  
HCatalog 14  
HDFS design  
  arbitrary file modifications 126  
  commodity hardware 126  
  data access streaming 125  
  large files 125  
  low-latency data access 126  
  small files 126  
Hfile 81  
high-performance computing (HPC) 71  
Hive 14, 337  
  architecture 343  
  complex data types 344, 345  
  data types and formats 344  
  installing 338-340  
  installing, with MySQL 341, 342  
  primitive data types 344  
  services 343, 344  
HiveQL 351  
  Data Definition Language (DDL) 351-357  
  Data Manipulation Language (DML) 357-359  
HiveQL, versus traditional database  
  indexes 351  
  schema on read versus write 350  
  transactions 350  
  update operations 350, 351  
HLog 82  
Hmaster 80  
horizontal partitioning 67

Illustration operator 337  
industry examples, big data  
  Hadoop at Facebook 23, 24  
  Hadoop at Yahoo 21  
  RackSpace, for log processing 21-23  
information management solutions 20  
Integrated Development Environment (IDE) 326  
internet-business-to-business (B2B) transactions 4  
Internet Cache Protocol (ICP) 382  
Iterative Programming (IP) 411

## J

---

Java interface 130  
  APIs, in C language 131  
  data reading 132  
  Filesystem in Userspace (FUSE) 131  
  HTTP 130  
Java MapReduce 93-96  
Java Virtual Machine (JVM) 231, 410  
JobConf object 115  
JSON (JavaScript Object Notation) 40

## K

---

key-value database 37  
  Amazon's dynamo 37, 38  
  Project Voldemort 38-40  
  Tokyo cabinet 40  
  Tokyo tyrant 40

## L

---

locks in HBase  
  Multi-Version Concurrency Control (MVCC) 257  
  row-level locks 257

## M

---

machine learning library (MLlib) 415  
Mahout 14  
MapReduce 23, 89, 214  
  anatomy of file read 233, 234  
  anatomy of file write 232  
  anatomy of job 231  
  architecture 91, 92  
  calculations, composing 102-104

example 98  
job chaining 230, 231  
job control 230, 231  
job, debugging 229, 230  
map side 220, 221  
reduce side 221, 222  
sample program, using for 222-228  
traditional way 216  
using 372, 375  
Web User Interface (Web UI) 229  
word count program process 215, 216  
working 217-220

MapReduce 1 236  
job initialization 237  
job submission 237  
task assignment 237, 238  
task completion 238, 239  
task execution 238

MapReduce2 YARN 239, 240  
job initialization 241  
job submission 241  
task assignment 241, 242  
task execution 242

MapReduce datatype 92  
file input format 93  
Java MapReduce 93

MapReduce job run 236  
Classic MapReduce 236  
failure in MapReduce 2 242-250  
failure of MapReduce 1 238, 239

MapReduce 1 236-238  
MapReduce2 YARN 239-242

memstore 80

MongoDB 46  
examples 49-56  
features 46, 47  
operations 48  
scalar types 48

## N

---

NameNode 201  
National Security Agency (NSA) 10  
Neo4j 65  
Netflix 25  
network-attached storage (NAS) 71

NodeManagers 201

NoSQL

database 37

terminology 36

## O

---

Oozie 14

operations, in MongoDB

aggregation 48

CRUD 48

data manipulation 48

GridFS 49

indexing 48

querying 48

text search 49

transactions 49

Oracle VirtualBox

for Hadoop installation 168-182

## P

---

packet routing 383

partitioner 96, 97

situation 102

Pig data model 317

Complex 318, 319

scalar 318

Pig framework 14

PigLatin 319, 320

examples 324, 325

input and output 320, 321

relational operations 322-324

store 321

User Defined Functions (UDF) 325-334

PigLatin script

Describe operator 335

developing 335

Dump operator 335

Explanation operator 336

Illustration operator 337

testing 335

PigPig 314

platform, for Pig programs 313

embedded 314

example 314-316

Grunt 314

grunt commands 317  
Grunt Shell 314  
script 313  
Principal Component Analysis (PCA) 25  
Project Voldemort 38, 39  
pseudo-distributed mode, Hadoop  
implementing 159, 160

## Q

---

Quality of Service (QoS) 26

## R

---

rebalancing 67  
Record Columnar (RC) file 347  
record linkage 405-409  
region server 80  
relational data 19  
Relational Database Management Systems (RDBMS) 35, 261  
terminology 36  
versus big data 17  
Remote Procedure Call (RPC) 144  
representational state transfer (REST) 14  
Resilient Distributed Dataset (RDD) 404, 414, 415  
ResourceManager 201

## S

---

Scala  
examples of programs 432-435  
features 411, 412  
working on 412-414  
search and classification tools 20  
Secondary NameNode 201  
serialization 144  
shuffling 436, 437  
Simple Storage Service (S3) 395  
Social Network Analysis (SNA) 29  
Spark 403  
memory issues 438, 439  
programming model 405  
Spark methods, for data processing 415  
aggregate 415, 416  
cartesian 417  
checkpoint 418  
cogroup 420

collect 421  
CollectAsMap 421, 422  
CombineByKey 422  
compute 423  
count 423  
countApproxDistinct 424, 425  
CountByKey 423  
CountByValue 424  
dependencies 425  
distinct 426  
filter 426  
filter transformation 427  
filterWith 426  
first 426  
fold 427  
foreach 427  
getStorageLevel 428  
groupBy 429  
histogram 430  
id 431  
join 431  
leftOuterJoin 431, 432  
repartition 418, 419  
Spark shell 409  
SCALA programming model 410, 411  
SQL, versus NoSQL 66  
data distribution 67  
data durability 69  
denormalization 67  
Master/Slave replication 68  
peer-to-peer replication 68  
replication 68  
sharding 67  
Standalone mode, Hadoop 158, 159  
storage area network (SAN) 71  
symmetric multiprocessing (SMP) 71

## T

---

Tokyo cabinet 40  
Tokyo tyrant 40

## U

---

Ubuntu  
for Hadoop installation 160-168  
unstructured data 18, 19

and large data 19, 20  
managing 20  
mining 19  
use cases, Cassandra 303  
eBay 304  
Hulu 304  
User Defined Functions (UDF), PigLatin 325-330  
aggregate functions 331  
Eval functions 331  
FILTER functions 334  
User-Defined Functions (UDFs) 14

## V

---

Visual Networking Index (VNI) 4

## W

---

web analytics 26, 27  
Web UI, MapReduce 229  
Write Ahead Log (WAL) 80, 81

## Y

---

Yet Another Resource Negotiator (YARN) 239  
YouTube 25

## Z

---

Zookeeper 82  
ZooKeeper 14