Modern Big Data Processing with Hadoop

Expert techniques for architecting end-to-end big data solutions to get valuable insights

www.packt.com

By V. Naresh Kumar and Prashant Shindgikar

Modern Big Data Processing with Hadoop

Expert techniques for architecting end-to-end big data solutions to get valuable insights

V. Naresh Kumar
Prashant Shindgikar

Packt

BIRMINGHAM - MUMBAI

Modern Big Data Processing with Hadoop

Copyright © 2018 Packt Publishing

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(s), nor Packt Publishing 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.

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

Commissioning Editor: Amey Varangaonkar
Acquisition Editor: Varsha Shetty
Content Development Editor: Cheryl Dsa
Technical Editor: Sagar Sawant
Copy Editors: Vikrant Phadke, Safis Editing
Project Coordinator: Nidhi Joshi
Proofreader: Safis Editing
Indexer: Pratik Shirodkar
Graphics: Tania Dutta
Production Coordinator: Arvindkumar Gupta

First published: March 2018

Production reference: 1280318

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

ISBN 978-1-78712-276-5

`www.packtpub.com`

Get
80%
off any Packt tech eBook or Video!
Full-Stack Web Development with Vue.js and Node
Frank Kane
Hands-On Data Science and Python Machine Learning
Perform data mining and machine learning efficiently using Python and Spark
The Modern C++ Challenge
Go to www.packtpub.com and use this code in the checkout:
HBBI80OFF
Packt

mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

PacktPub.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.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 Packt books and eBooks.

Contributors

About the authors

V. Naresh Kumar has more than a decade of professional experience in designing, implementing, and running very-large-scale Internet applications in Fortune 500 Companies. He is a full-stack architect with hands-on experience in e-commerce, web hosting, healthcare, big data, analytics, data streaming, advertising, and databases. He admires open source and contributes to it actively. He keeps himself updated with emerging technologies from Linux systems internals to frontend technologies. He studied in BITS- Pilani, Rajasthan, with a dual degree in computer science and economics.

Prashant Shindgikar is an accomplished big data Architect with over 20 years of experience in data analytics. He specializes in data innovation and resolving data challenges for major retail brands. He is a hands-on architect having an innovative approach to solving data problems. He provides thought leadership and pursues strategies for engagements with the senior executives on innovation in data processing and analytics. He presently works for a large USA-based retail company.

About the reviewers

Sumit Pal is a published author with Apress. He has 22+ years of experience in software from startups to enterprises and is an independent consultant working with big data, data visualization, and data science. He builds end-to-end data-driven analytic systems.

He has worked for Microsoft (SQLServer), Oracle (OLAP Kernel), and Verizon. He advises clients on their data architectures and builds solutions in Spark and Scala. He has spoken at many conferences in North America and Europe and has developed a big data analyst training for Experfy. He has an MS and BS in computer science.

Manoj R. Patil is a big data architect at TatvaSoft—an IT services and consulting firm. He has a bachelor's degree in engineering from COEP, Pune. He is a proven and highly skilled business intelligence professional with 18 years of experience in IT. He is a seasoned BI and big data consultant with exposure to all the leading platforms.

Earlier, he has served for organizations such as Tech Mahindra and Persistent Systems. Apart from authoring a book on Pentaho and big data, he has been an avid reviewer for different titles in the respective fields from Packt and other leading publishers.

> *Manoj would like to thank his entire family, especially his two beautiful angels Ayushee and Ananyaa for understanding him during the review process. He would also like to thank the Packt publication for giving this opportunity, the project co-ordinator and the author.*

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight 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.

Table of Contents

Preface

The complex structure of data these days requires sophisticated solutions for data transformation and its semantic representation to make information more accessible to users. Apache Hadoop, along with a host of other big data tools, empowers you to build such solutions with relative ease. This book lists some unique ideas and techniques that enable you to conquer different data processing and analytics challenges on your path to becoming an expert big data architect.

The book begins by quickly laying down the principles of enterprise data architecture and showing how they are related to the Apache Hadoop ecosystem. You will get a complete understanding of data life cycle management with Hadoop, followed by modeling structured and unstructured data in Hadoop. The book will also show you how to design real-time streaming pipelines by leveraging tools such as Apache Spark, as well as building efficient enterprise search solutions using tools such as Elasticsearch. You will build enterprise-grade analytics solutions on Hadoop and learn how to visualize your data using tools such as Tableau and Python.

This book also covers techniques for deploying your big data solutions on-premise and on the cloud, as well as expert techniques for managing and administering your Hadoop cluster.

By the end of this book, you will have all the knowledge you need to build expert big data systems that cater to any data or insight requirements, leveraging the full suite of modern big data frameworks and tools. You will have the necessary skills and know-how to become a true big data expert.

Who this book is for

This book is for big data professionals who want to fast-track their career in the Hadoop industry and become expert big data architects. Project managers and mainframe professionals looking forward to build a career in big data and Hadoop will also find this book useful. Some understanding of Hadoop is required to get the best out of this book.

What this book covers

`Chapter 1`, *Enterprise Data Architecture Principles*, shows how to store and model data in Hadoop clusters.

`Chapter 2`, *Hadoop Life Cycle Management*, covers various data life cycle stages, including when the data is created, shared, maintained, archived, retained, and deleted. It also further details data security tools and patterns.

`Chapter 3`, *Hadoop Design Considerations*, covers key data architecture principles and practices. The reader will learn how modern data architects adapt to big data architect use cases.

`Chapter 4`, *Data Movement Techniques*, covers different methods to transfer data to and from our Hadoop cluster to utilize its real power.

`Chapter 5`, *Data Modeling in Hadoop*, shows how to build enterprise applications using cloud infrastructure.

`Chapter 6`, *Designing Real-Time Streaming Data Pipelines*, covers different tools and techniques of designing real-time data analytics.

`Chapter 7`, *Large-Scale Data Processing Frameworks*, describes the architecture principles of enterprise data and the importance of governing and securing that data.

`Chapter 8`, *Building an Enterprise Search Platform*, gives a detailed architecture design to build search solutions using Elasticsearch.

`Chapter 9`, *Designing Data Visualization Solutions*, shows how to deploy your Hadoop cluster using Apache Ambari.

`Chapter 10`, *Developing Applications Using the Cloud*, covers different ways to visualize your data and the factors involved in choosing the correct visualization method.

`Chapter 11`, *Production Hadoop Cluster Deployment*, covers different data processing solutions to derive value out of our data.

To get the most out of this book

It would be great if proper installation of Hadoop is done as explained in the earlier set of chapters. Detailed or even little knowledge of Hadoop will serve as an added advantage.

Download the example code files

You can download the example code files for this book from your account at `www.packtpub.com`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packtpub.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Modern-Big-Data-Processing-with-Hadoop`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `http://www.packtpub.com/sites/default/files/downloads/ModernBigDataProcessingwithHadoop_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

```
export HADOOP_CONF_DIR="${HADOOP_CONF_DIR:-$YARN_HOME/etc/hadoop}"
export HADOOP_COMMON_HOME="${HADOOP_COMMON_HOME:-$YARN_HOME}"
export HADOOP_HDFS_HOME="${HADOOP_HDFS_HOME:-$YARN_HOME}"
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
$ hadoop fs -cat /tmp/output-7/part*
 NewDelhi, 440
 Kolkata, 390
 Bangalore, 270
```

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

```
useradd hadoop
passwd hadoop1
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

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 `copyright@packtpub.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 `authors.packtpub.com`.

Reviews

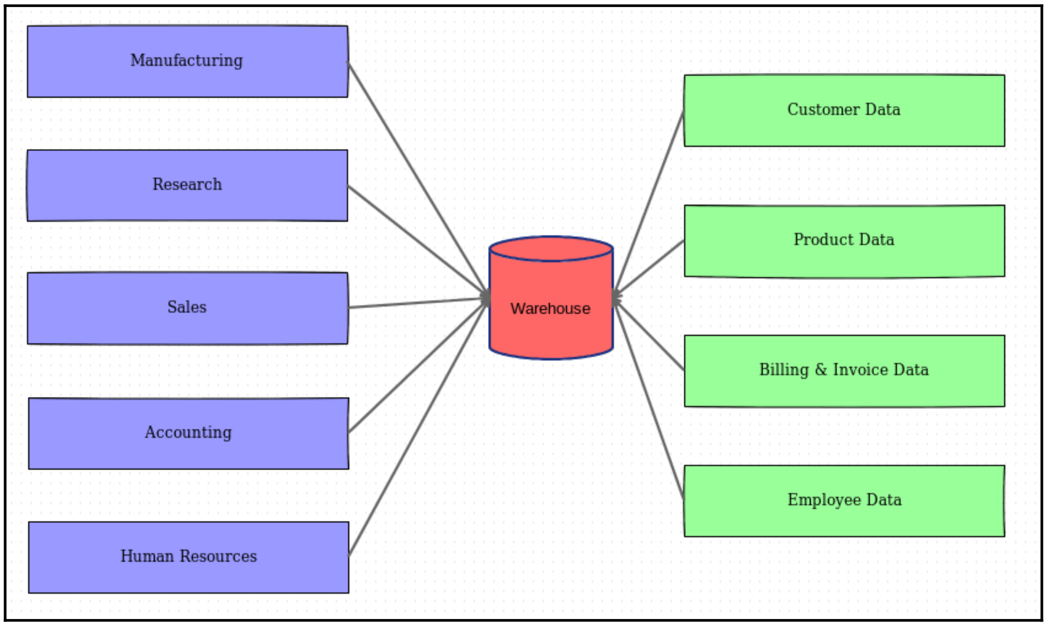
Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your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

For more information about Packt, please visit `packtpub.com`.

1 Enterprise Data Architecture Principles

Traditionally, enterprises have embraced data warehouses to store, process, and access large volumes of data. These warehouses are typically large RDBMS databases capable of storing a very-large-scale variety of datasets. As the data complexity, volume, and access patterns have increased, many enterprises have started adopting big data as a model to redesign their data organization and define the necessary policies around it.

This figure depicts how a typical data warehouse looks in an Enterprise:



As Enterprises have many different departments, organizations, and geographies, each one tends to own a warehouse of their own and presents a variety of challenges to the Enterprise as a whole. For example:

- Multiple sources and destinations of data
- Data duplication and redundancy
- Data access regulatory issues
- Non-standard data definitions across the Enterprise.
- Software and hardware scalability and reliability issues
- Data movement and auditing
- Integration between various warehouses

It is becoming very easy to build very-large-scale systems at less costs compared to what it was a few decades ago due to several advancements in technology, such as:

- Cost per terabyte
- Computation power per nanometer
- Gigabits of network bandwidth
- Cloud

With globalization, markets have gone global and the consumers are also global. This has increased the reach manifold. These advancements also pose several challenges to the Enterprises in terms of:

- Human capital management
- Warehouse management
- Logistics management
- Data privacy and security
- Sales and billing management
- Understanding demand and supply

In order to stay on top of the demands of the market, Enterprises have started collecting more and more metrics about themselves; thereby, there is an increase in the dimensions data is playing with in the current situation.

In this chapter, we will learn:

- Data architecture principles
- The importance of metadata
- Data governance
- Data security
- Data as a Service
- Data architecture evolution with Hadoop

Data architecture principles

Data at the current state can be defined in the following four dimensions (four Vs).

Volume

The volume of data is an important measure needed to design a big data system. This is an important factor that decides the investment an Enterprise has to make to cater to the present and future storage requirements.

Different types of data in an enterprise need different capacities to store, archive, and process. Petabyte storage systems are a very common in the industry today, which was almost impossible to reach a few decades ago.

Velocity

This is another dimension of the data that decides the mobility of data. There exist varieties of data within organizations that fall under the following categories:

- Streaming data:
    - Real-time/near-real-time data
- Data at rest:
    - Immutable data
    - Mutable data

This dimension has some impact on the network architecture that Enterprise uses to consume and process data.

Variety

This dimension talks about the form and shape of the data. We can further classify this into the following categories:

- Streaming data:
  - On-wire data format (for example, JSON, MPEG, and Avro)
- Data At Rest:
  - Immutable data (for example, media files and customer invoices)
  - Mutable data (for example, customer details, product inventory, and employee data)
- Application data:
  - Configuration files, secrets, passwords, and so on

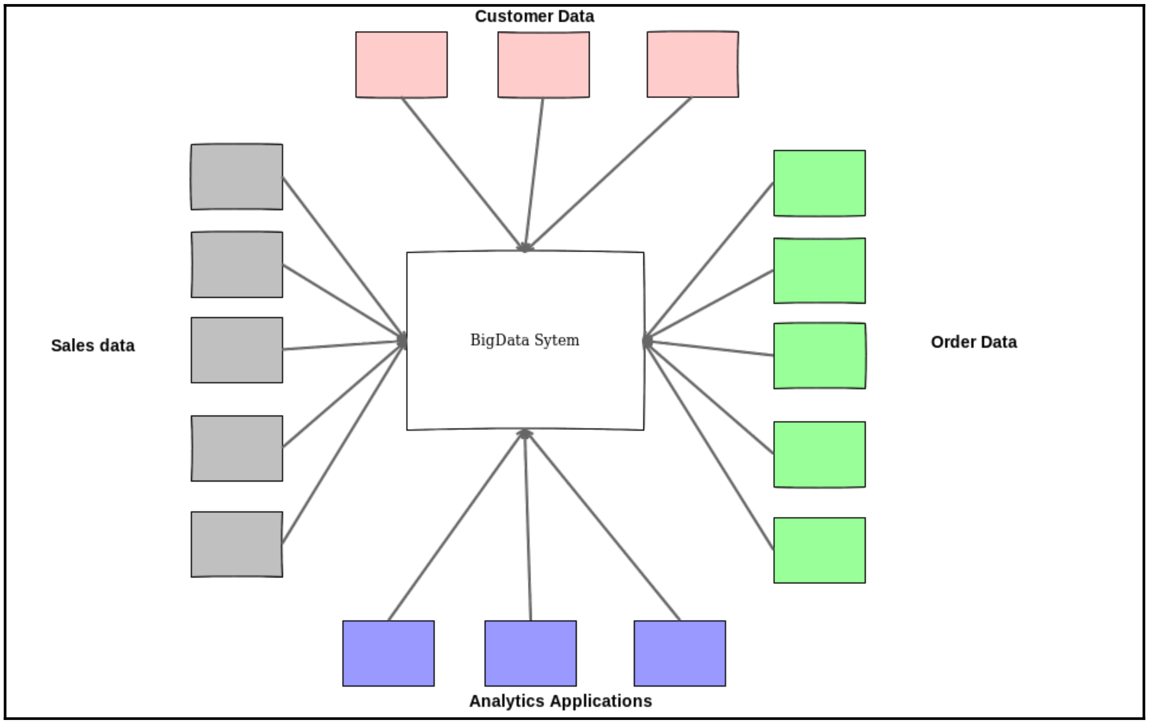
As an organization, it's very important to embrace very few technologies to reduce the variety of data. Having many different types of data poses a very big challenge to an Enterprise in terms of managing and consuming it all.

Veracity

This dimension talks about the accuracy of the data. Without having a solid understanding of the guarantee that each system within an Enterprise provides to keep the data safe, available, and reliable, it becomes very difficult to understand the Analytics generated out of this data and to further generate insights.

Necessary auditing should be in place to make sure that the data that flows through the system passes all the quality checks and finally goes through the big data system.

Let's see how a typical big data system looks:



As you can see, many different types of applications are interacting with the big data system to store, process, and generate analytics.

The importance of metadata

Before we try to understand the importance of Metadata, let's try to understand what metadata is. Metadata is simply **data about data**. This sounds confusing as we are defining the definition in a recursive way.

In a typical big data system, we have these three levels of verticals:

- Applications writing data to a big data system
- Organizing data within the big data system
- Applications consuming data from the big data system

This brings up a few challenges as we are talking about millions (even billions) of data files/segments that are stored in the big data system. We should be able to correctly identify the ownership, usage of these data files across the Enterprise.

Let's take an example of a TV broadcasting company that owns a TV channel; it creates television shows and broadcasts it to all the target audience over wired cable networks, satellite networks, the internet, and so on. If we look carefully, the source of content is only one. But it's traveling through all possible mediums and finally reaching the user's Location for viewing on TV, mobile phone, tablets, and so on.

Since the viewers are accessing this TV content on a variety of devices, the applications running on these devices can generate several messages to indicate various user actions and preferences, and send them back to the application server. This data is pretty huge and is stored in a big data system.

Depending on how the data is organized within the big data system, it's almost impossible for outside applications or peer applications to know about the different types of data being stored within the system. In order to make this process easier, we need to describe and define how data organization takes place within the big data system. This will help us better understand the data organization and access within the big data system.

Let's extend this example even further and say there is another application that reads from the big data system to understand the best times to advertise in a given TV series. This application should have a better understanding of all other data that is available within the big data system. So, without having a well-defined metadata system, it's very difficult to do the following things:

- Understand the diversity of data that is stored, accessed, and processed
- Build interfaces across different types of datasets
- Correctly tag the data from a security perspective as highly sensitive or insensitive data
- Connect the dots between the given sets of systems in the big data ecosystem
- Audit and troubleshoot issues that might arise because of data inconsistency

Data governance

Having very large volumes of data is not enough to make very good decisions that have a positive impact on the success of a business. It's very important to make sure that only quality data should be collected, preserved, and maintained. The data collection process also goes through evolution as new types of data are required to be collected. During this process, we might break a few interfaces that read from the previous generation of data. Without having a well-defined process and people, handling data becomes a big challenge for all sizes of organization.

To excel in managing data, we should consider the following qualities:

- Good policies and processes
- Accountability
- Formal decision structures
- Enforcement of rules in management

The implementation of these types of qualities is called **data governance**. At a high level, we'll define data governance as data that is managed well. This definition also helps us to clarify that data management and data governance are not the same thing. Managing data is concerned with the use of data to make good business decisions and ultimately run organizations. Data governance is concerned with the degree to which we use disciplined behavior across our entire organization in how we manage that data.

It's an important distinction. So what's the bottom line? Most organizations manage data, but far fewer govern those management techniques well.

Fundamentals of data governance

Let's try to understand the fundamentals of data governance:

- Accountability
- Standardization
- Transparency

Transparency ensures that all the employees within an organization and outside the organization understand their role when interacting with the data that is related to the organization. This will ensure the following things:

- Building trust
- Avoiding surprises

Accountability makes sure that teams and employees who have access to data describe what they can do and cannot do with the data.

Standardization deals with how the data is properly labeled, describe, and categorized. One example is how to generate email address to the employees within the organization. One way is to use `firstname-lastname@company.com`, or any other combination of these. This will ensure that everyone who has access to these email address understands which one is first and which one is last, without anybody explaining those in person.

Standardization improves the quality of data and brings order to multiple data dimensions.

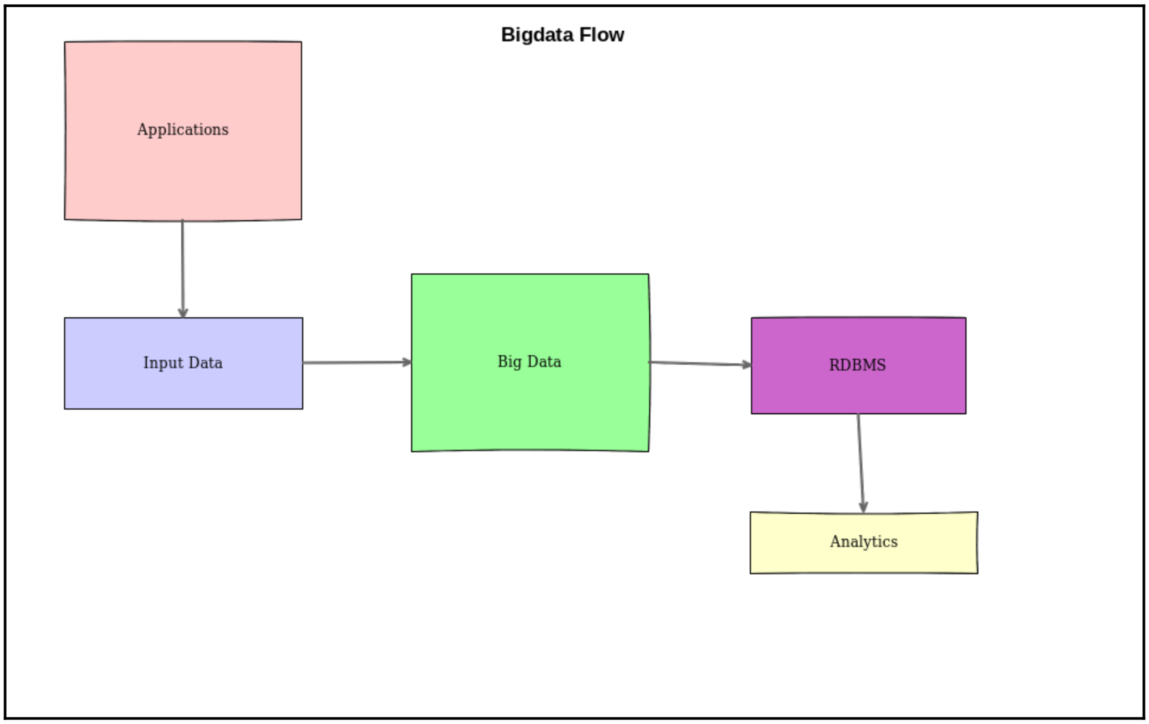
Data security

Security is not a new concept. It's been adopted since the early UNIX time-sharing operating system design. In the recent past, security awareness has increased among individuals and organizations on this security front due to the widespread data breaches that led to a lot of revenue loss to organizations.

Security, as a general concept, can be applied to many different things. When it comes to data security, we need to understand the following fundamental questions:

- *What types of data exist?*
- *Who owns the data?*
- *Who has access to the data?*
- *When does the data exit the system?*
- *Is the data physically secured?*

Let's have a look at a simple big data system and try to understand these questions in more detail. The scale of the systems makes security a nightmare for everyone. So, we should have proper policies in place to keep everyone on the same page:



In this example, we have the following components:

- Heterogeneous applications running across the globe in multiple geographical regions.
- Large volume and variety of input data is generated by the applications.
- All the data is ingested into a big data system.
- ETL/ELT applications consume the data from a big data system and put the consumable results into RDBMS (this is optional).
- Business intelligence applications read from this storage and further generate insights into the data. These are the ones that power the leadership team's decisions.

You can imagine the scale and volume of data that flows through this system. We can also see that the number of servers, applications, and employees that participate in this whole ecosystem is very large in number. If we do not have proper policies in place, its not a very easy task to secure such a complicated system.

Also, if an attacker uses social engineering to gain access to the system, we should make sure that the data access is limited only to the lowest possible level. When poor security implementations are in place, attackers can have access to virtually all the business secrets, which could be a serious loss to the business.

Just to think of an example, a start-up is building a next-generation computing device to host all its data on the cloud and does not have proper security policies in place. When an attacker compromises the security of the servers that are on the cloud, they can easily figure out what is being built by this start-up and can steal the intelligence. Once the intelligence is stolen, we can imagine how hackers use this for their personal benefit.

With this understanding of security's importance, let's define what needs to be secured.

Application security

Applications are the front line of product-based organizations, since consumers use these applications to interact with the products and services provided by the applications. We have to ensure that proper security standards are followed while programming these application interfaces.

Since these applications generate data to the backend system, we should make sure only proper access mechanisms are allowed in terms of firewalls.

Also, these applications interact with many other backend systems, we have to ensure that the correct data related to the user is shown. This boils down to implementing proper authentication and authorization, not only for the user but also for the application when accessing different types of an organization's resources.

Without proper auditing in place, it is very difficult to analyze the data access patterns by the applications. All the logs should be collected at a central place away from the application servers and can be further ingested into the big data system.

Input data

Once the applications generate several metrics, they can be temporarily stored locally that are further consumed by periodic processes or they are further pushed to streaming systems like Kafka.

In this case, we should carefully think through and design where the data is stores and which uses can have access to this data. If we are further writing this data to systems like Kafka or MQ, we have to make sure that further authentication, authorization, and access controls are in place.

Here we can leverage the operating-system-provided security measures such as process user ID, process group ID, filesystem user ID, group ID, and also advanced systems (such as SELinux) to further restrict access to the input data.

Big data security

Depending on which data warehouse solution is chosen, we have to ensure that authorized applications and users can write to and read from the data warehouse. Proper security policies and auditing should be in place to make sure that this large scale of data is not easily accessible to everyone.

In order to implement all these access policies, we can use the operating system provided mechanisms like file access controls and use access controls. Since we're talking about geographically distributed big data systems, we have to think and design centralized authentication systems to provide a seamless experience for employees when interacting with these big data systems.

RDBMS security

Many RDBMSes are highly secure and can provide the following access levels to users:

- Database
- Table
- Usage pattern

They also have built-in auditing mechanisms to tell which users have accessed what types of data and when. This data is vital to keeping the systems secure, and proper monitoring should be in place to keep a watch on these system's health and safety.

BI security

These can be applications built in-house for specific needs of the company, or external applications that can power the insights that business teams are looking for. These applications should also be properly secured by practicing single sign-on, role-based access control, and network-based access control.

Since the amount of insights these applications provide is very much crucial to the success of the organization, proper security measures should be taken to protect them.

So far, we have seen the different parts of an enterprise system and understood what things can be followed to improve the security of the overall enterprise data design. Let's talk about some of the common things that can be applied everywhere in the data design.

Physical security

This deals with physical device access, data center access, server access, and network access. If an unauthorized person gains access to the equipment owned by an Enterprise, they can gain access to all the data that is present in it.

As we have seen in the previous sections, when an operating system is running, we are able to protect the resources by leveraging the security features of the operating system. When an intruder gains physical access to the devices (or even decommissioned servers), they can connect these devices to another operating system that's in their control and access all the data that is present on our servers.

Care must be taken when we decommission servers, as there are ways in which data that's written to these devices (even after formatting) can be recovered. So we should follow industry-standard device erasing techniques to properly clean all of the data that is owned by enterprises.

In order to prevent those, we should consider encrypting data.

Data encryption

Encrypting data will ensure that even when authorized persons gain access to the devices, they will not be able to recover the data. This is a standard practice that is followed nowadays due to the increase in mobility of data and employees. Many big Enterprises encrypt hard disks on laptops and mobile phones.

Secure key management

If you have worked with any applications that need authentication, you will have used a combination of username and password to access the services. Typically these secrets are stored within the source code itself. This poses a challenge for programs which are non-compiled, as attackers can easily access the username and password to gain access to our resources.

Many enterprises started adopting centralized key management, using which applications can query these services to gain access to the resources that are authentication protected. All these access patterns are properly audited by the KMS

Employees should also access these systems with their own credentials to access the resources. This makes sure that secret keys are protected and accessible only to the authorized applications.

Data as a Service

Data as a Service (**DaaS**) is a concept that has become popular in recent times due to the increase in adoption of cloud. When it comes to data. It might some a little confusing that how can data be added to as a service model?

DaaS offers great flexibility to users of the service in terms of not worrying about the scale, performance, and maintenance of the underlying infrastructure that the service is being run on. The infrastructure automatically takes care of it for us, but given that we are dealing with a cloud model, we have all the benefits of the cloud such as *pay as you go*, capacity planning, and so on. This will reduce the burden of data management.

If we try to understand this carefully we are taking out the data management part alone. But data governance should be well-defined here as well or else we will lose all the benefits of the service model.

So far, we are talking about the *Service in the cloud* concept. *Does it mean that we cannot use this within the Enterprise or even smaller organizations?* The answer is *No*. Because this is a generic concept that tells us the following things.

When we are talking about a service model, we should keep in mind the following things, or else chaos will ensue:

- Authentication
- Authorization
- Auditing

This will guarantee that only well-defined users, IP addresses, and services can access the data exposed as a service.

Let's take an example of an organization that has the following data:

- Employees
- Servers and data centers
- Applications
- Intranet documentation sites

As you can see, all these are independent datasets. But, as a whole when we want the organization to succeed. There is lot of overlap and we should try to embrace the DaaS model here so that all these applications that are authoritative for the data will still manage the data. But for other applications, they are exposed as a simple service using REST API; therefore, this increases collaboration and fosters innovation within the organization.

Let's take further examples of how this is possible:

- The team that manages all the employee data in the form of a database can provide a simple **Data Service**. All other applications can use this dataset without worrying about the underlying infrastructure on which this employee data is stored:
  - This will free the consumers of the data services in such a way that the consumers:
    - Need not worry about the underlying infrastructure
    - Need not worry about the protocols that are used to communicate with these data servers
    - Can just focus on the REST model to design the application

  - Typical examples of this would be:
    - Storing the employee data in a database like `LDAP` or the `Microsoft Active` directory
- The team that manages the infrastructure for the entire organization can design their own system to keep off the entire hardware inventory of the organization, and can provide a simple data service. The rest of the organization can use this to build applications that are of interest to them:
  - This will make the Enterprise more agile
  - It ensures there is a single source of truth for the data about the entire hardware of the organization
  - It improves trust in the data and increases confidence in the applications that are built on top of this data
- Every team in the organization might use different technology to build and deploy their applications on to the servers. Following this, they also need to build a data store that keeps track of the active versions of software that are deployed on the servers. Having a data source like this helps the organization in the following ways:
  - Services that are built using this data can constantly monitor and see where the software deployments are happening more often
  - The services can also figure out which applications are vulnerable and are actively deployed in production so that further action can be taken to fix the loopholes, either by upgrading the OS or the software
  - Understanding the challenges in the overall software deployment life cycle
  - Provides a single platform for the entire organization to do things in a standard way, which promotes a sense of ownership
- Documentation is one of the very important things for an organization. Instead of running their own infrastructure, with the DaaS model, organizations and teams can focus on the documents that are related to their company and pay only for those. Here, services such as Google Docs and Microsoft Office Online are very popular as they give us flexibility to pay as we go and, most importantly, not worry about the technology required to build these.
  - Having such a service model for data will help us do the following:
    - Pay only for the service that is used
    - Increase or decrease the scale of storage as needed

- Access the data from anywhere if the service is on the Cloud and connected to the internet
- Access corporate resources when connected via VPN as decided by the Enterprise policy

In the preceding examples, we have seen a variety of applications that are used in Enterprises and how data as a model can help Enterprises in variety of ways to bring collaboration, innovation, and trust.

But, *when it comes to big data, what can DaaS Do?*

Just like all other data pieces, big data can also be fit into a DaaS model and provides the same flexibility as we saw previously:

- No worry about the underlying hardware and technology
- Scale the infrastructure as needed
- Pay only for the data that is owned by the Enterprise
- Operational and maintenance challenges are taken away
- Data can be made geographically available for high availability
- Integrated backup and recovery for DR requirements

With these few advantages, enterprises can be more agile and build applications that can leverage this data as service.

Evolution data architecture with Hadoop

Hadoop is a software that helps in scalable and distributed computing. Before Hadoop came into existence, there were many technologies that were used by the industry to take care of their data needs. Let's classify these storage mechanisms:

- Hierarchical database
- Network database
- Relational database

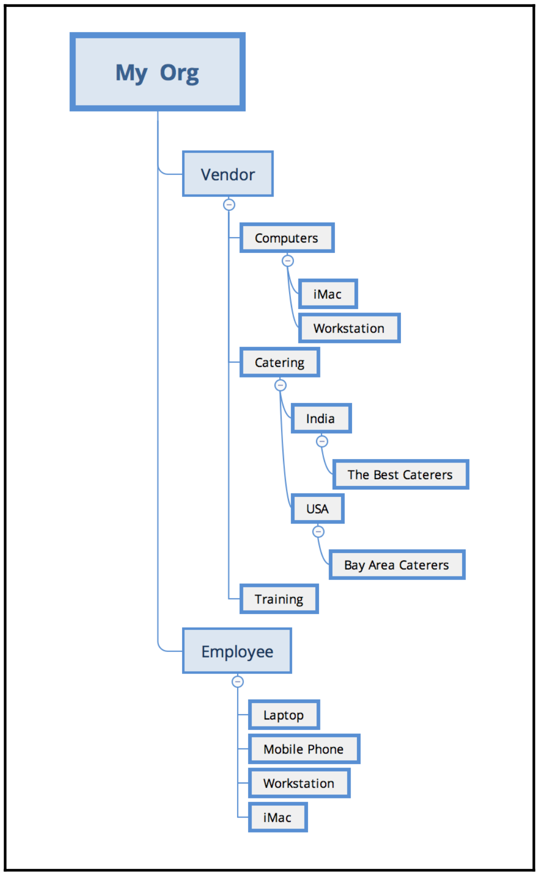
Let's understand what these data architectures are.

Hierarchical database architecture

This model of storing Enterprise data was invented by IBM in the early 60s and was used in their applications. The basic concept of hierarchical databases is that the data is organized in the form of a **rooted tree**. The root node is the beginning of the tree and then all the children are linked only to one of its parent nodes. This is a very unique way of storing and retrieving things.

If you have some background in computer science, trees are one of the unique ways of storing data so that it has some relation with each other (like a parent and child relationship).

This picture illustrates how data is organized in a typical HDBMS:



As we can see, the root node is the organization itself and all the data associated with the organization follows a tree structure which depicts several relationships. These relationships can be understood like this:

- **Employee** owns **Laptop**, **Mobile phone**, **Workstation**, and **iMac**
- **Employee** belongs to **organization**
- Many vendors supply different requirements:
    - Computer vendors supply **iMac** and **Workstation**
- Catering is in both India and USA; two vendors, *The Best Caterers* and *Bay Area Caterers,* serve these

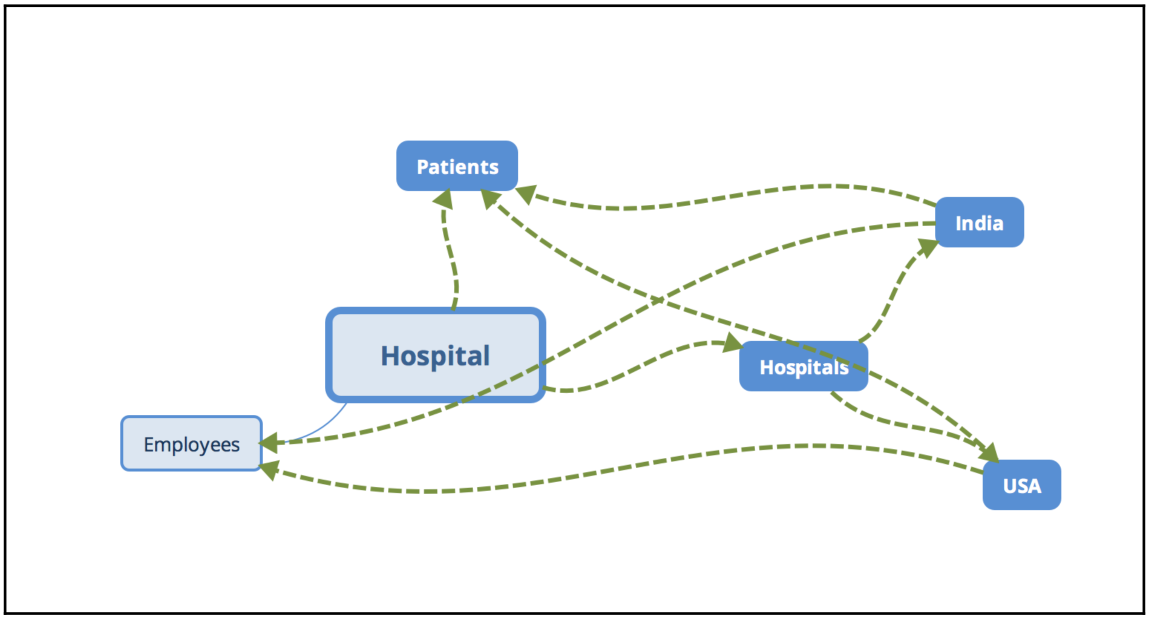
Even though we have expressed multiple types of relationships in this one gigantic data store, we can see that the data gets duplicated and also querying data for different types of needs becomes a challenge.

Let's take a simple question like: *Which vendor supplied the iMac owned by Employee-391?*

In order to do this, we need to traverse the tree and find information from two different sub-trees.

Network database architecture

The network database management system also has its roots in computer science: graph theory, where there are a vast and different types of nodes and relationships connect them together. There is no specific root node in this structure. It was invented in the early 70s:



As we can see, in this structure, there are a few core datasets and there are other datasets linked with the core datasets.

This is how we can understand it:

- The main hospital is defined
- It has many subhospitals
- Subhospitals are in India and USA
- The Indian hospital uses the data in patients
- The USA hospital uses the data in patients
- The patients store is linked to the main hospital
- Employees belong to the hospital and are linked with other organizations

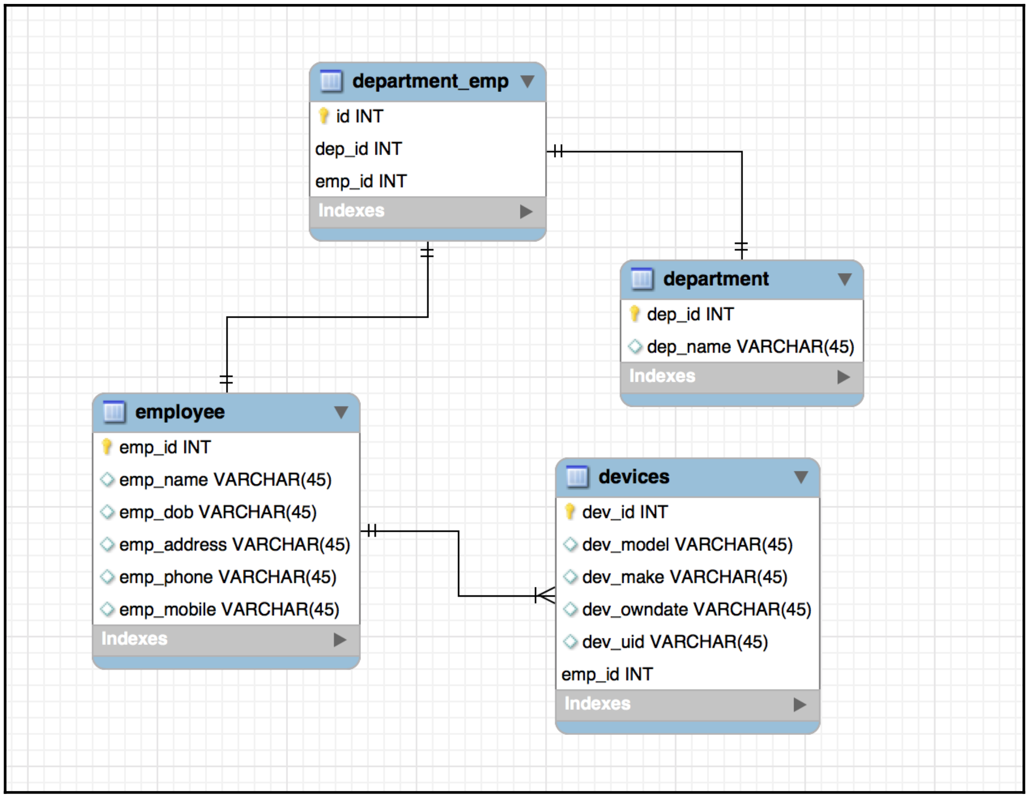
In this structure, depending upon the design we come up with, the data is represented as a network of elements.

Relational database architecture

This system was developed again in IBM in the early 80s and is considered one of the most reputed database systems to date. A few notable examples of the software that adopted this style are Oracle and MySQL.

In this model, data is stored in the form of records where each record in turn has several attributes. All the record collections are stored in a table. Relationships exist between the data attributes across tables. Sets of related tables are stored in a database.

Let's see a typical example of how this RDBMS table looks:



We are defining the following types of tables and relationships

Employees

- The table consists of all the employee records
- Each record is defined in terms of:
    - Employee unique identifier
    - Employee name
    - Employee date of birth
    - Employee address
    - Employee phone
    - Employee mobile

Devices

- The table consists of all the devices that are owned by employees
- Each ownership record is defined in terms of the following:
    - Device ownership identifier
    - Device model
    - Device manufacturer
    - Device ownership date
    - Device unique number
    - Employee ID

Department

A table consisting of all the departments in the organization:

- Unique department ID
- Unique department name

Department and employee mapping table

This is a special table that consists of only the relationships between the department and employee using their unique identifiers:

- Unique department ID
- Unique employee ID

Hadoop data architecture

So far, we have explored several types of data architectures that have been in use by Enterprises. In this section, we will understand how the data architecture is made in Hadoop.

Just to give a quick introduction, Hadoop has multiple components:

- Data
- Data management
- Platform to run jobs on data

Data layer

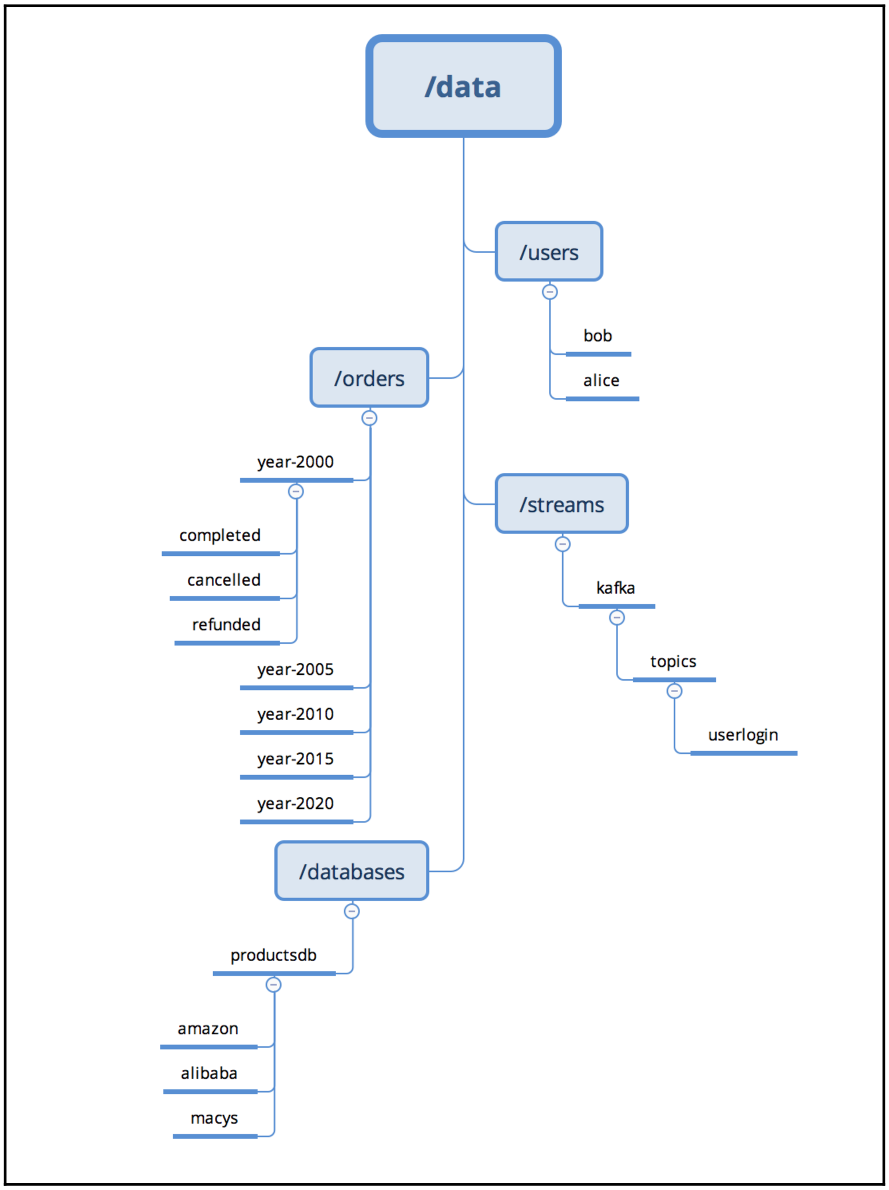
This is the layer where all of the data is stored in the form of files. These files are internally split by the Hadoop system into multiple parts and replicated across the servers for high availability.

Since we are talking about the data stored in terms of files, it is very important to understand how these files are organized for better governance.

The next diagram shows how the data can be organized in one of the Hadoop storage layers. The content of the data can be in any form as Hadoop does not enforce them to be in a specific structure. So, we can safely store Blu-Ray™ Movies, **CSV** (**Comma Separated Value)** Files, AVRO Encoded Files, and so on inside this data layer.

TIP

You might be wondering why we are not using the word **HDFS** (**Hadoop Distributed File System**) here. It's because Hadoop is designed to run on top of any distributed file system.


/data
/users
bob
alice
/orders
year-2000
completed
cancelled
refunded
year-2005
year-2010
year-2015
year-2020
/streams
kafka
topics
userlogin
/databases
productsdb
amazon
alibaba
macys

Data management layer

This layer is responsible for keeping track of where the data is stored for a given file or path (in terms of servers, offsets, and so on). Since this is just a bookkeeping layer, it's very important that the contents of this layer are protected with high reliability and durability. Any corruption of the data in this layer will cause the entire data files to be lost forever.

In Hadoop terminology, this is also called **NameNode**.

Job execution layer

Once we have the data problem sorted out, next come the programs that read and write data. When we talk about data on a single server or a laptop, we are well aware where the data is and accordingly we can write programs that read and write data to the corresponding locations.

In a similar fashion, the Hadoop storage layer has made it very easy for applications to give file paths to read and write data to the storage as part of the computation. This is a very big win for the programming community as they need not worry about the underlying semantics about where the data is physically stored across the distributed Hadoop cluster.

Since Hadoop promotes the *compute near the data model*, which gives very high performance and throughput, the programs that were run can be scheduled and executed by the Hadoop engine closer to where the data is in the entire cluster. The entire transport of data and movement of the software execution is all taken care of by Hadoop.

So, end users of Hadoop see the system as a simple one with massive computing power and storage. This abstraction has won everyone's requirements and has become the standard in big data computing today.

Summary

In this chapter, we have seen how many organizations have adopted data warehouses to store, process, and access large volumes of data they possess. We learned about data architecture principles, their governance, and security. In the next chapter, we will take a look at some concepts of data pre-processing.

2
Hadoop Life Cycle Management

In this chapter, we will understand the following topics:

- Data wrangling
- Data masking
- Data security

Data wrangling

If you have some experience working on data of some sort, you will recollect that most of the time data needs to be preprocessed so that we can further use it as part of a bigger analysis. This process is called **data wrangling**.

Let's see what the typical flow in this process looks like:

- Data acquisition
- Data structure analysis
- Information extraction
- Unwanted data removal
- Data transformation
- Data standardization

Let's try to understand these in detail.

Data acquisition

Even though not a part of data wrangling, this phase deals with the process of acquiring data from somewhere. Typically, all data is generated and stored in a central location or is available in files located on some shared storage.

Having an understanding of this step helps us to build an interface or use existing libraries to pull data from the acquired data source location.

Data structure analysis

Once data is acquired, we have to understand the structure of the data. Remember that the data we are getting can be in any of the following forms:

- Text data:
    - Structured data
    - Unstructured data
- Binary data

This is where we need certain tools to help us understand the structure of the data.

Once we have a thorough understanding of the data we are dealing with, the next task is to understand the bits and pieces we need to extract from this structure. Sometimes, depending on the complexity and size of the data we are dealing with, it might take time for us to really find and extract the information we are looking for.

Once we know what we are looking for and also have a solid understanding of the structure of the data, it becomes easier for us to come up with simple algorithms to extract the required information from the input data.

Information extraction

In this phase, we are interested in extracting the necessary details from the input data. In the previous phase, we already identified the necessary pieces that are of interest to us. Here is where we can adopt the following techniques for information extraction:

- Identify and locate where the text is present

- Analyze and come up with the best method of information extraction:
- Tokenize and extract information
- Go to offset and extract information
- Regular expression-based information extraction
- Complex algorithm-based information extraction

Depending on the complexity of the data, we might have to adopt one or more of the aforementioned techniques to extract the information from the target data.

Unwanted data removal

This phase can occur before the information extraction step or after the information extraction step. It depends on which one is easier (shortening the text or the extraction of information). This is a design choice the analyst can make.

In this phase, we are removing unwanted data from the information or input data so that the data is further distilled and can easily be consumed for our business needs.

Data transformation

This is also a very important phase, where we enforce the standards defined by the enterprise to define the final data output. For example, an organization can suggest that all the country codes should be in ISO 3166-1 alpha-2 format. In order to adhere to this standard, we might have to transform the input data, which can contain countries with their full names. So a mapping and transformation has to be done.

Many other transformations can be performed on the input data to make the final data consumable by anyone in the organization in a well-defined form and as per the organizations standards.

This step also gives some importance to having an enterprise level standard to improve collaboration.

Data standardization

Once the information extraction is complete and any necessary cleanup is done, we need to decide how we are going to save the outcome of this process. Typically, we can use a simple **CSV** (**comma separated value**) format for this data. If we are dealing with a complicated output format, we can choose **XML** (**Extensible Markup Language**) or **JSON** (**javascript object notation**) formats.

These formats are very much standard and almost all the technologies that we have today understand these very easily. But to keep things simple at first, it's good to start with CSV format.

Data masking

Businesses that deal with customer data have to make sure that the **PII** (**personally identifiable information**) of these customers is not moving freely around the entire data pipeline. This criterion is applicable not only to customer data but also to any other type of data that is considered classified, as per standards such as GDPR, SOX, and so on. In order to make sure that we protect the privacy of customers, employees, contractors, and vendors, we need to take the necessary precautions to ensure that when the data goes through several pipelines, users of the data see only anonymized data. The level of anonymization we do depends upon the standards the company adheres to and also the prevailing country standards.

So, data masking can be called the process of hiding/transforming portions of original data with other data without losing the meaning or context.

In this section, we will understand various techniques that are available to accomplish this:

- Substitution:
    - Static
    - Dynamic:
        - Encryption
        - Hashing
- Hiding
- Erasing
- Truncation
- Variance
- Shuffling

Substitution

Substitution is the process of replacing portions of data with computed data. It can be mathematically be defined as:

$$f(x) = y$$

Where x is the source and y is the output from this function.

In order to choose the correct substitution mechanism, we need to understand how this data is going to be used, the target audience, and the data flow environment as well. Let's look at the various available substitution mechanisms.

Static

In this method, we have a Lookup table; it consists of all possible substitutions for a given set of inputs. This Lookup table can be visualized like this:

Source Text (y)	**Substituted Text (y)**
Steve Jobs	AAPL-1
Cat	123456789
Tennis	Cricket

This table illustrates how a Lookup table can be constructed for substituting source text with a different text. This method scales well when there is a predefined quantity of substitutions available.

Another example of this Lookup table-based substitution is when we follow a naming standard for country codes, for example, ISO-8661:

Source Text (x)	**Substituted Text (y)**
Egypt	EG
India	IN
Saint Vincent and Grenadines	VN
United Kingdom	GB
United States of America	US

Dynamic

These substitution techniques are useful when there are a large number of possibilities and we want to change the data using some algorithms. These methods can be classified into two types.

Encryption

This is the process of changing a given text to some other form by using some form of secret. These are mathematically defined functions:

$$f(input, secret) = output$$

As you can see, these functions take an input and a secret and generate data that can be decrypted using the same secret and the output:

$$f(output, secret) = input$$

If we observe carefully, it is the secret that is playing an important role here. In cryptography, there are two types of algorithms that are available based on this secret. The usage of these depends on the situation and the secret transportation challenges.

Without going too deep into cryptography, let's try to understand what these methods are:

- Symmetric key encryption
- Asymmetric key encryption

The basic difference between the two is that in the first one, we use the same secret for both encryption and decryption. But in the latter, we use two different keys for encryption and decryption.

Let's take a look at a few examples of symmetric key encryption in action:

Algorithm	**Input Data**	**Output Data**	**Method**
ROT13	`hello`	`uryyb`	Encryption
	`uryyb`	`hello`	Decryption
DES	`hello`	`yOYffF4rl8lxCQ4HS2fpMg==`	Encryption (secret is `hello`)

	`yOYffF4rl8lxCQ4HS2fpMg==`	`hello`	Decryption (secret is `hello`)
RIJNDAEL-256	`hello`	`v8QbYPszQX/TFeYKbSfPL/ rNJDywBIQKtxzOzWhBm16/ VSNN4EtlgZi3/ iPqJZpCiXXzDu0sKmKSl6IxbBKhYw==`	Encryption (secret is `hello`)
	`v8QbYPszQX/TFeYKbSfPL/ rNJDywBIQKtxzOzWhBm16/ VSNN4EtlgZi3/ iPqJZpCiXXzDu0sKmKSl6IxbBKhYw==`	`hello`	Encryption (secret is `hello`)

As you can see, the data that is generated varies in both complexity and length depending on the encryption algorithm we use. It also depends on the secret key that is used for encryption.

Encryption poses a challenge of more computational requirements and storage space. We need to plan our system accordingly if we want to use encryption as one of the methods in the masking process.

Hashing

This is also a cryptography-based technique where the original data is converted to an irreversible form. Let's see the mathematical form for this:

$$f(input) = output$$

Here, unlike in the case of encryption, we cannot use the output to discover what the input is.

Let's see a few examples to understand this better:

Input	**Output**	**Method**
`10-point`	`7d862a9dc7b743737e39dd0ea3522e9f`	MD5
`10th`	`8d9407b7f819b7f25b9cfab0fe20d5b3`	MD5
`10-point`	`c10154e1bdb6ea88e5c424ee63185d2c1541efe1bc3d4656a4c3c99122ba9256`	SHA256
`10th`	`5b6e8e1fcd052d6a73f3f0f99ced4bd54b5b22fd4f13892eaa3013ca65f4e2b5`	SHA256

We can see that depending upon the encryption algorithm we have used, the output size varies. Another thing to note is that a given hash function produces the same output size irrespective of the input size.

Hiding

In this approach, the data is considered too sensitive even to reveal it to the original owners. So, to protect the confidentiality of the data, certain portions of the text are masked with a predefined character, say X (or anything), so that only the person with complete knowledge about those pieces can extract the necessary information.

Examples: Credit card information is considered highly confidential and should never be revealed to anyone. If you have some experience of purchasing online on websites such as Amazon and so on, you would have seen that your full credit card information is not shown; only the last four digits are shown. Since I am the genuine owner of such a credit card, I can easily identify it and continue with the transaction.

Similarly, when there is a need for portions of data to be seen by analysts, it's important to mask significant pieces of it so that the end users will not get the complete picture but will use this data at the same time for any analysis that they are doing.

Let's see a few examples to understand this better:

Data type	**Input**	**Output**	**Network**
Creditcard	4485 **4769 3682** 9843	4485 **XXXX XXXX** 9843	Visa
Creditcard	5402 **1324 5087** 3314	5402 **XXXX XXXX** 3314	Mastercard
Creditcard	3772 **951960** 72673	3772 **XXXXXX** 72673	American Express

In the preceding examples, these numbers follow a predefined algorithm and size. So a simple technique of masking digits at fixed locations can work better.

Let's take up another example of hiding out portions of email addresses which vary in both size and complexity. In this case we have to follow different techniques to hide the characters to not reveal complete information:

Data type	**Input**	**Output**	**Method**
Email	`hello@world.com`	`h.l.o@w.r.d.com`	Even Hide
	`simple@book.com`	`.i.p.e@.o.k.c.m`	Odd Hide
	`something@something.com`	`s...th.ng@..me...com`	Complex Hide

The techniques can be as simple as:

- **Even Hide**: In this technique, we hide the every character that is in the even position
- **Odd Hide**: We hide every odd character in the input data
- **Complex Hide**: In this technique, we understand the data we are dealing with using NLP and then try to apply an algorithm that doesn't reveal too much information that would allow any intelligent person to decode

Erasing

As the name suggests, this causes data loss when applied to the input data. Depending on the significance of the data we are dealing with, we need to apply this technique. Typical examples of this technique is to set a `NULL` value for all the records in a column. Since this null data cannot be used to infer anything that is meaningful, this technique helps in making sure that confidential data is not sent to the other phases of data processing.

Let's take few examples of erasing:

Input Data	**Output Data**	**What's erased**
NULL earns 1000 INR per month	Ravi earns NULL per month	Salary and name
NULL mobile number is 0123456789	Ravi's mobile number is NULL	Mobile number and name

From the examples, you might be wondering: why do we nullify these values? This technique is useful when we are not really interested in the PII but interested in a summary of how many salary records or mobile number records are there in our database/input.

This concept can be extended to other use cases as well.

Truncation

Another variant of erasing is truncation, where we make all the input data a uniform size. This is useful when we are pretty sure that information loss is accepted in the further processing of the pipelines.

This can also be an intelligent truncation where we are aware of the data we are dealing with. Let's see this example of email addresses:

Input	**Output**	**What's truncated**
`alice@localhost.com`	`alice`	`@localhost.com`
`bob@localhost.com`	`bob`	`@localhost.com`
`rob@localhost.com`	`rob`	`@localhost.com`

From the preceding examples, we can see that all the domain portions from the email are truncated as all of them belong to the same domain. This technique saves storage space.

Variance

This technique is useful for data types that are numeric in nature. It can also be applied to Date/Time values.

This follows a statistical approach where we try to algorithmically vary the input data by a factor of +/- X percent. The value of X purely depends on the analysis we are doing and shouldn't have an overall impact on understanding the business figures.

Let's see a few examples:

Input Data	**Output Data**	**Method**	**Explanation**
100	110	Fixed variance	Increase by 10%
-100	90	Fixed variance	Decrease by 10%
1-Jan-2000	1-Feb-2000	Fixed variance	Add 1 month
1-Aug-2000	1-Jul-2000	Fixed variance	Reduce by 1 month
100	101	Dynamic variance	1% to 5% increase or decrease
100	105	Dynamic	1% to 5% increase or decrease

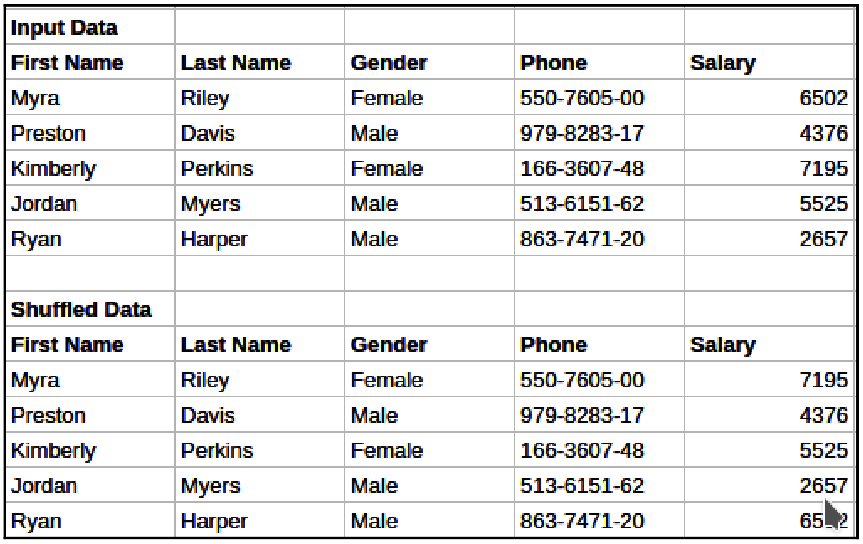
Shuffling

This is also considered one of the standard techniques of achieving anonymity of data. This process is more applicable where we have records of data with several attributes (columns in database terminology). In this technique, the data in the records is shuffled around a column so as to make sure that the record-level information is changed. But statistically, the data value remains the same in that column.

Example: When doing an analysis on the salary ranges of an organization, we can actually do a shuffle of the entire salary column, where the salaries of all the employees never match reality. But we can use this data to do an analysis on the ranges.

Complex methods can also be employed in this case, where we can do a shuffle based on other fields such as seniority, geography, and so on. The ultimate objective of this technique is to preserve the meaning of the data and, at the same time, make it impossible to discover the original owners of these attributes.

Let's see this with some example data:



Input Data				
First Name	**Last Name**	**Gender**	**Phone**	**Salary**
Myra	Riley	Female	550-7605-00	6502
Preston	Davis	Male	979-8283-17	4376
Kimberly	Perkins	Female	166-3607-48	7195
Jordan	Myers	Male	513-6151-62	5525
Ryan	Harper	Male	863-7471-20	2657
Shuffled Data				
First Name	**Last Name**	**Gender**	**Phone**	**Salary**
Myra	Riley	Female	550-7605-00	7195
Preston	Davis	Male	979-8283-17	4376
Kimberly	Perkins	Female	166-3607-48	5525
Jordan	Myers	Male	513-6151-62	2657
Ryan	Harper	Male	863-7471-20	65[illegible]2

There are five sample employee records with their salary information. The top table has original salary details and the table below has shuffled salary records. Look at the data carefully and you will understand. Remember that while shuffling, a random algorithm can be applied to increase the complexity of discovering the truth.

Data security

Data has become a very important asset for businesses when making very critical decisions. As the complexity of the infrastructure that generates and uses this data, its very important to have some control over the access patterns of this data. In the Hadoop ecosystem, we have Apache Ranger, which is another open source project that helps in managing the security of big data.

What is Apache Ranger?

Apache Ranger is an application that enables data architects to implement security policies on a big data ecosystem. The goal of this project is to provide a unified way for all Hadoop applications to adhere to the security guidelines that are defined.

Here are some of the features of Apache Ranger:

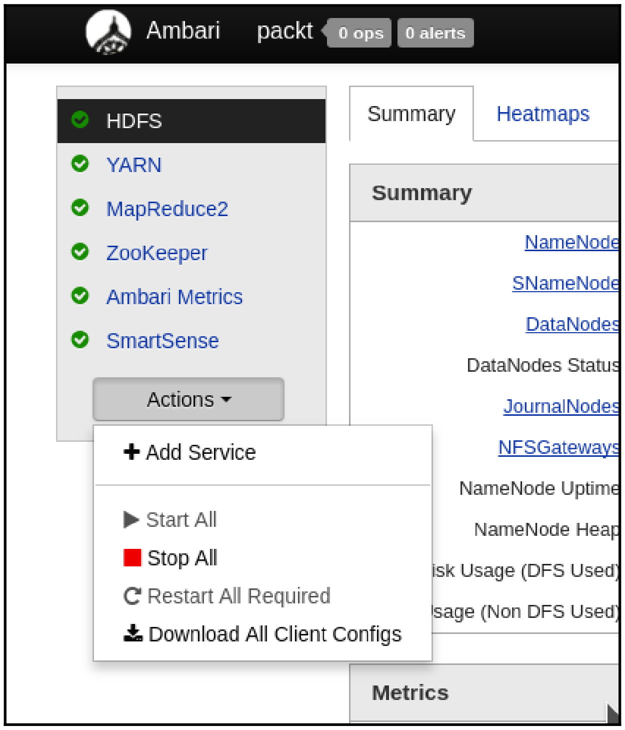
- Centralized administration
- Fine grained authorization
- Standardized authorization
- Multiple authorization methods
- Centralized auditing

Apache Ranger installation using Ambari

In this section, we will install Ranger using Apache Ambari. This section assumes that there is already a running Ambari instance.

Ambari admin UI

Open the Ambari web interface running on master node; then click on **Add Service**, as shown in the screenshot:

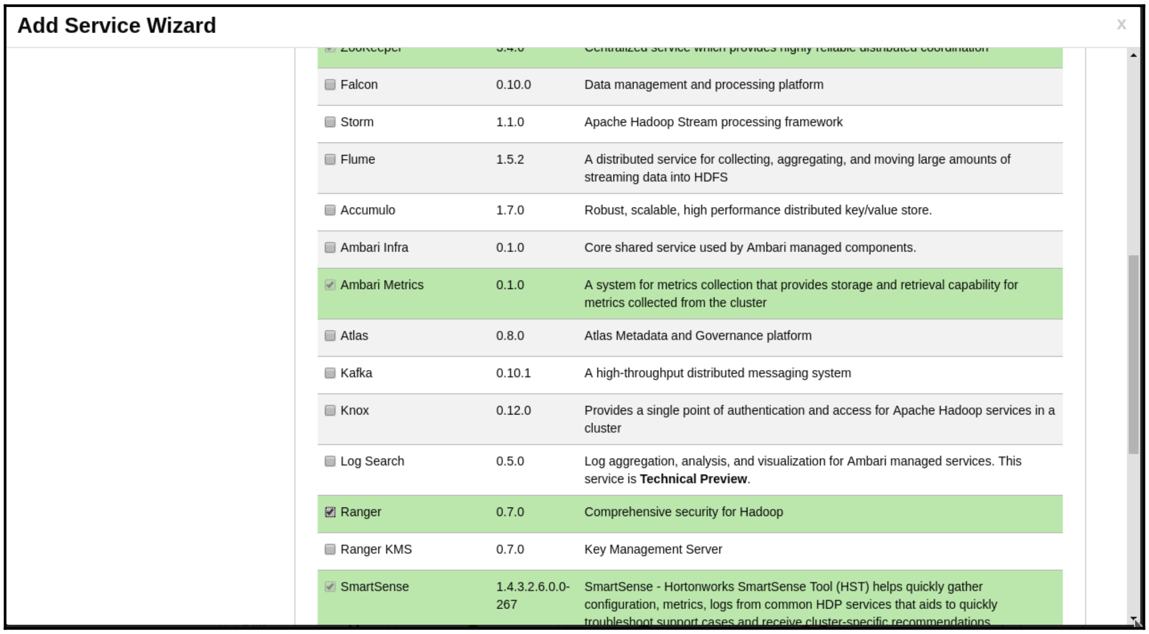


This will open a modal window, **Add Service Wizard**, which will take us through several steps for a complete installation of Apache Ambari.

Add service

Once the modal window is in view, select the **Apache Ranger** service from the list and click on **Next** on the screen.

This is shown in the following screenshot:



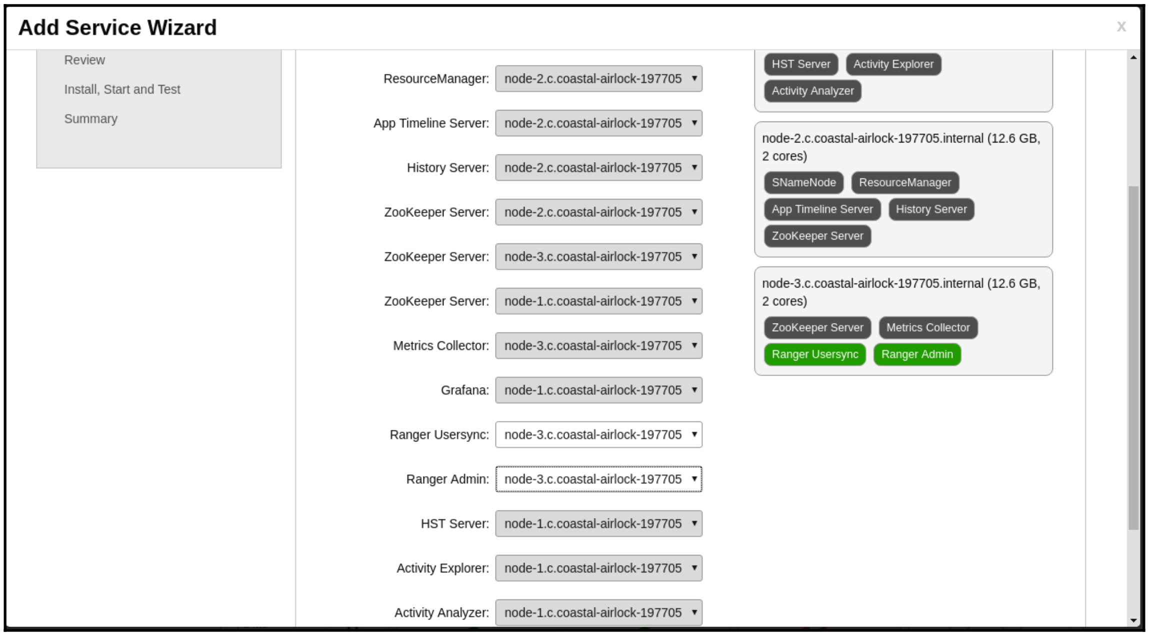
Add Service Wizard

✓ ZooKeeper	3.4.6	Centralized service which provides highly reliable distributed coordination
☐ Falcon	0.10.0	Data management and processing platform
☐ Storm	1.1.0	Apache Hadoop Stream processing framework
☐ Flume	1.5.2	A distributed service for collecting, aggregating, and moving large amounts of streaming data into HDFS
☐ Accumulo	1.7.0	Robust, scalable, high performance distributed key/value store.
☐ Ambari Infra	0.1.0	Core shared service used by Ambari managed components.
✓ Ambari Metrics	0.1.0	A system for metrics collection that provides storage and retrieval capability for metrics collected from the cluster
☐ Atlas	0.8.0	Atlas Metadata and Governance platform
☐ Kafka	0.10.1	A high-throughput distributed messaging system
☐ Knox	0.12.0	Provides a single point of authentication and access for Apache Hadoop services in a cluster
☐ Log Search	0.5.0	Log aggregation, analysis, and visualization for Ambari managed services. This service is **Technical Preview**.
✓ Ranger	0.7.0	Comprehensive security for Hadoop
☐ Ranger KMS	0.7.0	Key Management Server
✓ SmartSense	1.4.3.2.6.0.0-267	SmartSense - Hortonworks SmartSense Tool (HST) helps quickly gather configuration, metrics, logs from common HDP services that aids to quickly troubleshoot support cases and receive cluster-specific recommendations.

Service placement

Once the service is selected, we are presented the next step in the UI where we need to chose the servers on which this service is going to be installed and run.

I have selected **node-3** for Ranger (see the green labels):



ResourceManager: node-2.c.coastal-airlock-197705
App Timeline Server: node-2.c.coastal-airlock-197705
History Server: node-2.c.coastal-airlock-197705
ZooKeeper Server: node-2.c.coastal-airlock-197705
ZooKeeper Server: node-3.c.coastal-airlock-197705
ZooKeeper Server: node-1.c.coastal-airlock-197705
Metrics Collector: node-3.c.coastal-airlock-197705
Grafana: node-1.c.coastal-airlock-197705
Ranger Usersync: node-3.c.coastal-airlock-197705
Ranger Admin: node-3.c.coastal-airlock-197705
HST Server: node-1.c.coastal-airlock-197705
Activity Explorer: node-1.c.coastal-airlock-197705
Activity Analyzer: node-1.c.coastal-airlock-197705

HST Server, Activity Explorer, Activity Analyzer

node-2.c.coastal-airlock-197705.internal (12.6 GB, 2 cores)
SNameNode, ResourceManager, App Timeline Server, History Server, ZooKeeper Server

node-3.c.coastal-airlock-197705.internal (12.6 GB, 2 cores)
ZooKeeper Server, Metrics Collector, Ranger Usersync, Ranger Admin

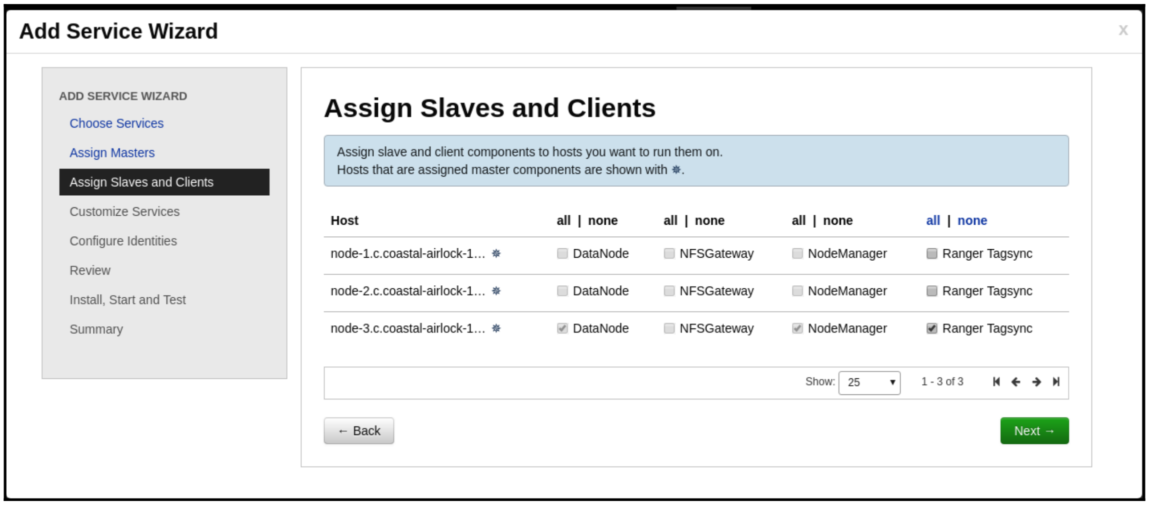
Screenshot showing how to choose servers on which this services is going to be installed and run

After this, select **Next**, which is at the bottom of the page.

Service client placement

In this step, we can choose where the clients for this service can be installed. Use the checkboxes to mark your preferences.

They look something like this:



Add Service Wizard

Assign Slaves and Clients

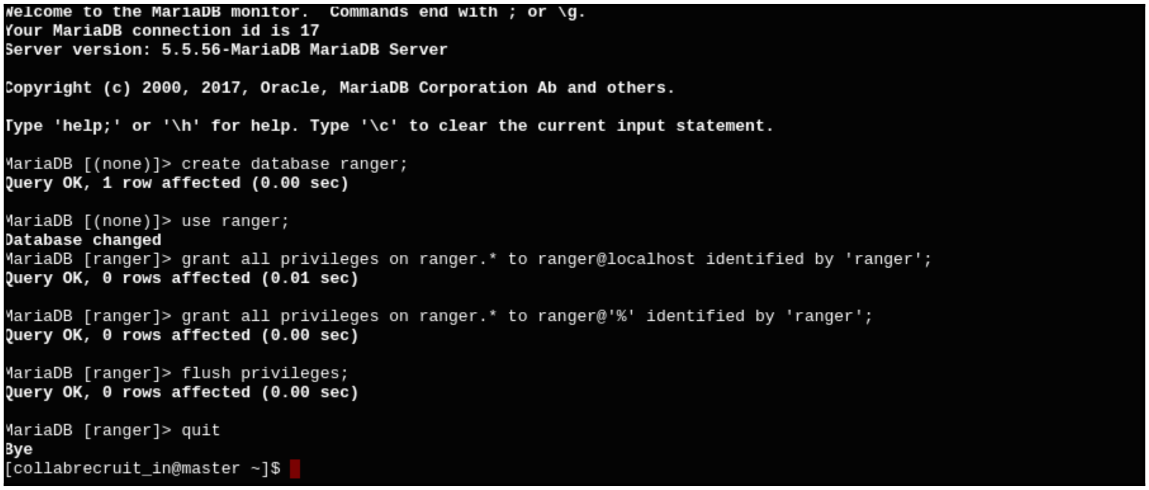
Assign slave and client components to hosts you want to run them on.
Hosts that are assigned master components are shown with ✱.

Host	all | none	all | none	all | none	all | none
node-1.c.coastal-airlock-1... ✱	☐ DataNode	☐ NFSGateway	☐ NodeManager	☐ Ranger Tagsync
node-2.c.coastal-airlock-1... ✱	☐ DataNode	☐ NFSGateway	☐ NodeManager	☐ Ranger Tagsync
node-3.c.coastal-airlock-1... ✱	☑ DataNode	☐ NFSGateway	☑ NodeManager	☑ Ranger Tagsync

Click on **Next** when your choices are made.

Database creation on master

We have installed the MySQL database server on the master node. Before we continue to the next step in the Ambari wizard, we have to create a new database and assign a few privileges:



```
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 17
Server version: 5.5.56-MariaDB MariaDB Server

Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> create database ranger;
Query OK, 1 row affected (0.00 sec)

MariaDB [(none)]> use ranger;
Database changed
MariaDB [ranger]> grant all privileges on ranger.* to ranger@localhost identified by 'ranger';
Query OK, 0 rows affected (0.01 sec)

MariaDB [ranger]> grant all privileges on ranger.* to ranger@'%' identified by 'ranger';
Query OK, 0 rows affected (0.00 sec)

MariaDB [ranger]> flush privileges;
Query OK, 0 rows affected (0.00 sec)

MariaDB [ranger]> quit
Bye
[collabrecruit_in@master ~]$
```

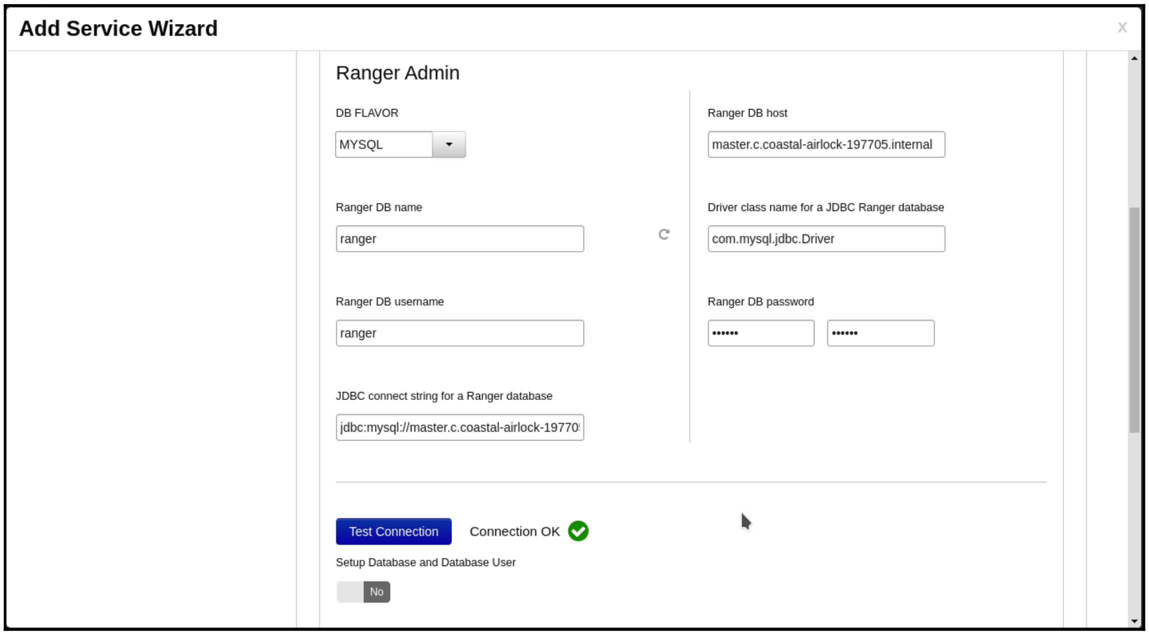
We also have to register the JDBC driver using the `ambari-server setup` command:

```
bash-$ sudo ambari-server setup --jdbc-db=mysql --jdbc-
driver=/usr/share/java/mysql-connector-java.jar
Using python /usr/bin/python
Setup ambari-server
Copying /usr/share/java/mysql-connector-java.jar to /var/lib/ambari-
server/resources
If you are updating existing jdbc driver jar for mysql with mysql-
connector-java.jar. Please remove the old driver jar, from all hosts.
Restarting services that need the driver, will automatically copy the new
jar to the hosts.
JDBC driver was successfully initialized.
Ambari Server 'setup' completed successfully.
```

After this step, we can go back to the Ambari wizard.

Ranger database configuration

In the wizard, we are prompted with the database name, username, and password. Please fill them according to the choices we made in the previous step:



Add Service Wizard

Ranger Admin

DB FLAVOR: MYSQL

Ranger DB host: master.c.coastal-airlock-197705.internal

Ranger DB name: ranger

Driver class name for a JDBC Ranger database: com.mysql.jdbc.Driver

Ranger DB username: ranger

Ranger DB password: •••••• ••••••

JDBC connect string for a Ranger database: jdbc:mysql://master.c.coastal-airlock-19770

Setup Database and Database User: No

Once the settings are added, please click on **Test Connection**. This will save lot of time later.

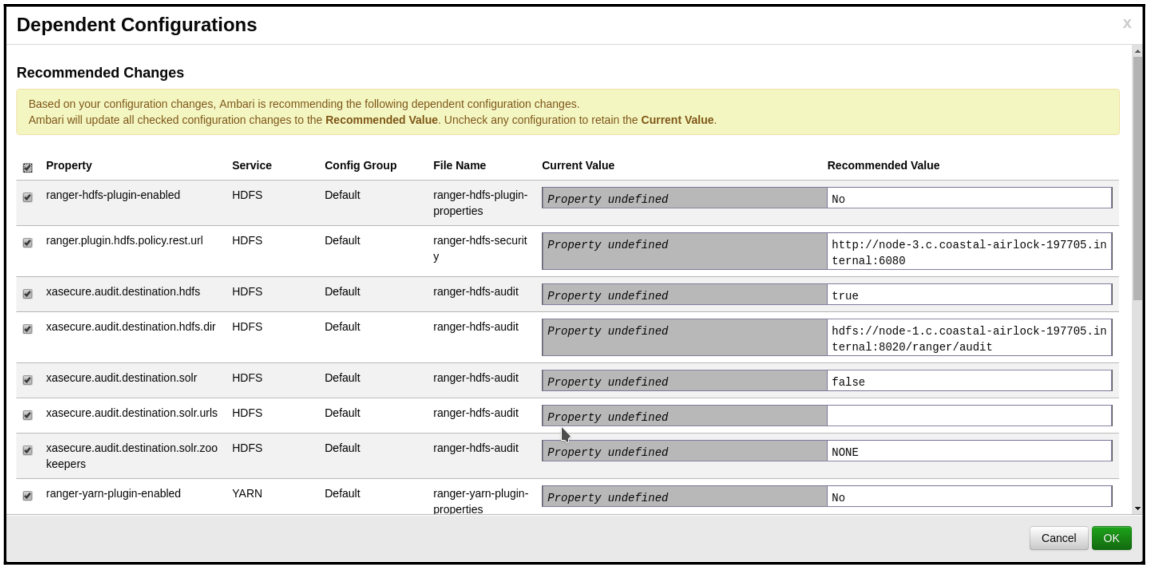
If there are any errors, please go back to the previous step; see whether there are any spelling mistakes and rerun those.

Click on **Next** when done with the changes.

Configuration changes

Since we are adding Ranger a service, Ambari shows a list of configuration changes that are required for Ranger to work correctly. Mostly leave these on default.

These changes look like the following screenshot. Once the changes look good, click on **OK** to continue:



Dependent Configurations

Recommended Changes

Based on your configuration changes, Ambari is recommending the following dependent configuration changes.
Ambari will update all checked configuration changes to the **Recommended Value**. Uncheck any configuration to retain the **Current Value**.

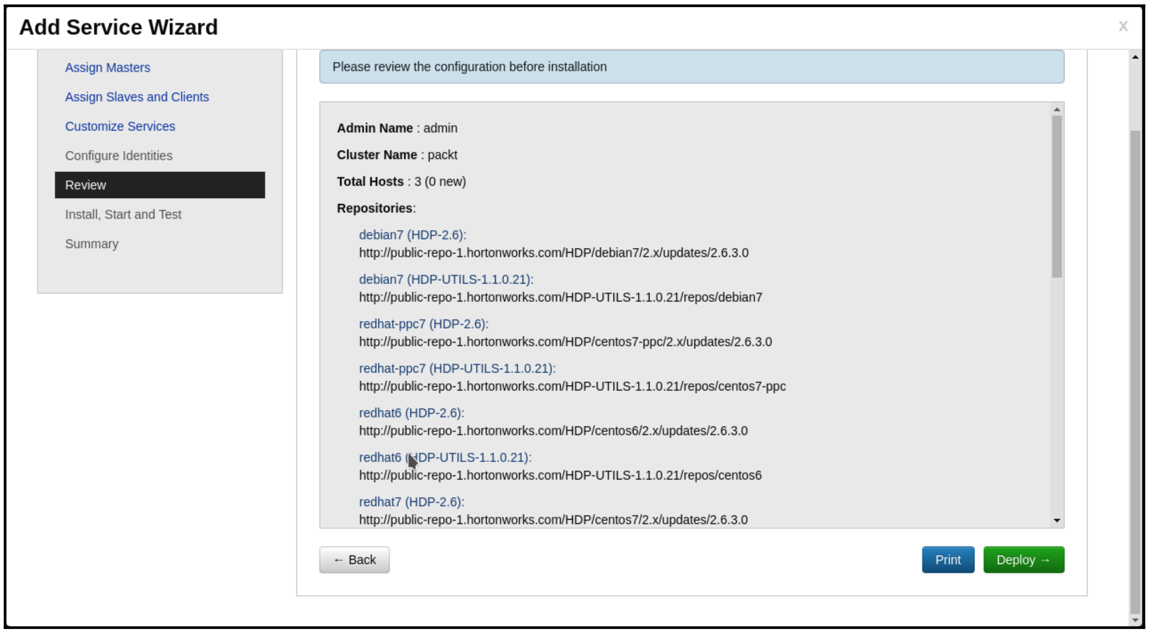
✓	Property	Service	Config Group	File Name	Current Value	Recommended Value
✓	ranger-hdfs-plugin-enabled	HDFS	Default	ranger-hdfs-plugin-properties	*Property undefined*	No
✓	ranger.plugin.hdfs.policy.rest.url	HDFS	Default	ranger-hdfs-security	*Property undefined*	http://node-3.c.coastal-airlock-197705.internal:6080
✓	xasecure.audit.destination.hdfs	HDFS	Default	ranger-hdfs-audit	*Property undefined*	true
✓	xasecure.audit.destination.hdfs.dir	HDFS	Default	ranger-hdfs-audit	*Property undefined*	hdfs://node-1.c.coastal-airlock-197705.internal:8020/ranger/audit
✓	xasecure.audit.destination.solr	HDFS	Default	ranger-hdfs-audit	*Property undefined*	false
✓	xasecure.audit.destination.solr.urls	HDFS	Default	ranger-hdfs-audit	*Property undefined*	
✓	xasecure.audit.destination.solr.zookeepers	HDFS	Default	ranger-hdfs-audit	*Property undefined*	NONE
✓	ranger-yarn-plugin-enabled	YARN	Default	ranger-yarn-plugin-properties	*Property undefined*	No

Configuration review

In this step, we are shown the list of changes that we have made so far in the wizard, and are shown choices to print a changes summary and deploy Ranger.

Only when we click on **Deploy** will the Ranger software get installed. Until then, it is all kept in the browser cache.

The screen looks like this:



Please review the configuration before installation

Admin Name : admin

Cluster Name : packt

Total Hosts : 3 (0 new)

Repositories:

debian7 (HDP-2.6):
http://public-repo-1.hortonworks.com/HDP/debian7/2.x/updates/2.6.3.0

debian7 (HDP-UTILS-1.1.0.21):
http://public-repo-1.hortonworks.com/HDP-UTILS-1.1.0.21/repos/debian7

redhat-ppc7 (HDP-2.6):
http://public-repo-1.hortonworks.com/HDP/centos7-ppc/2.x/updates/2.6.3.0

redhat-ppc7 (HDP-UTILS-1.1.0.21):
http://public-repo-1.hortonworks.com/HDP-UTILS-1.1.0.21/repos/centos7-ppc

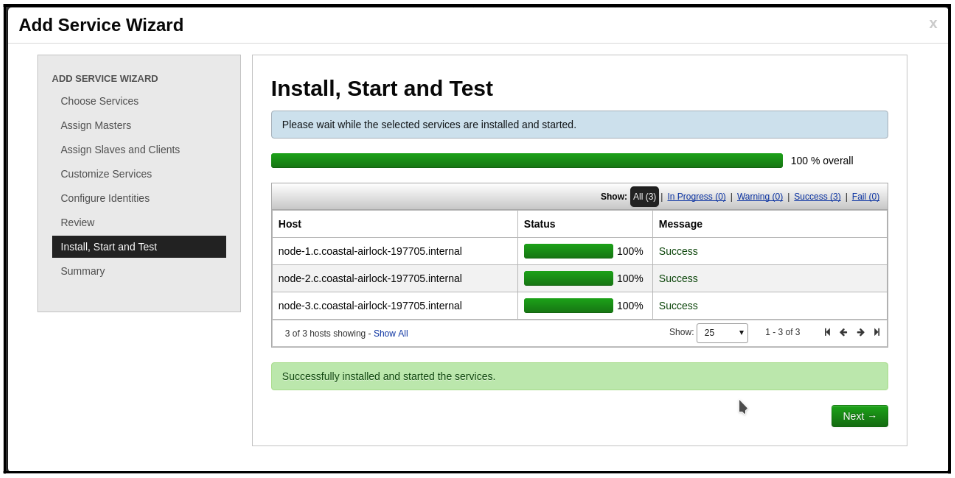
redhat6 (HDP-2.6):
http://public-repo-1.hortonworks.com/HDP/centos6/2.x/updates/2.6.3.0

redhat6 (HDP-UTILS-1.1.0.21):
http://public-repo-1.hortonworks.com/HDP-UTILS-1.1.0.21/repos/centos6

redhat7 (HDP-2.6):
http://public-repo-1.hortonworks.com/HDP/centos7/2.x/updates/2.6.3.0

Deployment progress

Once the installation of Ranger starts, it should look something like the one in the screenshot. There should not be any failures as we have set up all the configurations correctly. If there is any failure, check the logs and review the configuration by clicking on the **Back** button:


Add Service Wizard

ADD SERVICE WIZARD
- Choose Services
- Assign Masters
- Assign Slaves and Clients
- Customize Services
- Configure Identities
- Review
- Install, Start and Test
- Summary

Install, Start and Test

Please wait while the selected services are installed and started.

100 % overall

Show: All (3) | In Progress (0) | Warning (0) | Success (3) | Fail (0)

Host	Status	Message
node-1.c.coastal-airlock-197705.internal	100%	Success
node-2.c.coastal-airlock-197705.internal	100%	Success
node-3.c.coastal-airlock-197705.internal	100%	Success

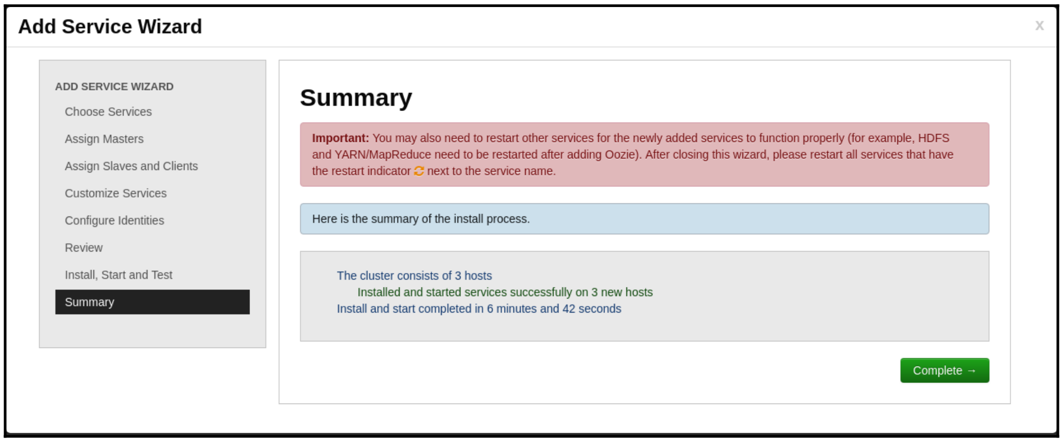
3 of 3 hosts showing - Show All   Show: 25   1 - 3 of 3

Successfully installed and started the services.

Next →

Application restart

Once the deployment is complete, we need to restart all the affected Hadoop components, as shown in the following screenshot:


Add Service Wizard

ADD SERVICE WIZARD
- Choose Services
- Assign Masters
- Assign Slaves and Clients
- Customize Services
- Configure Identities
- Review
- Install, Start and Test
- Summary

Summary

Important: You may also need to restart other services for the newly added services to function properly (for example, HDFS and YARN/MapReduce need to be restarted after adding Oozie). After closing this wizard, please restart all services that have the restart indicator next to the service name.

Here is the summary of the install process.

The cluster consists of 3 hosts
    Installed and started services successfully on 3 new hosts
Install and start completed in 6 minutes and 42 seconds

Complete →

Once all the components are restarted, the Ambari dashboard looks pretty healthy and we are done with the Apache Ranger installation.

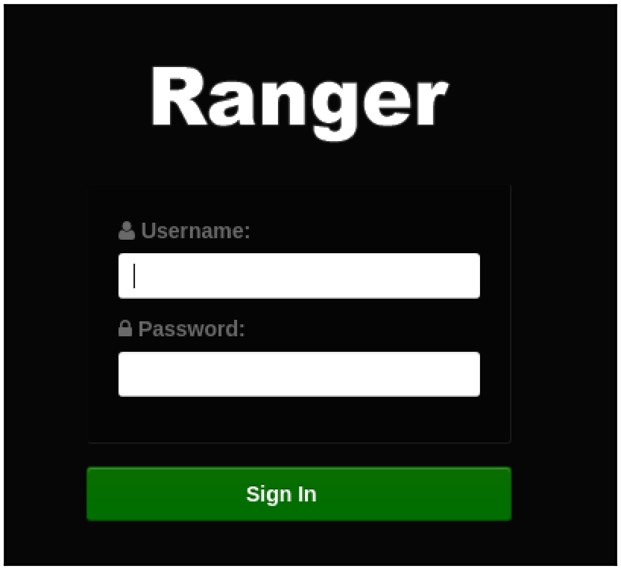
In the next step, we will see how to use Apache Ranger for handling our data security.

Apache Ranger user guide

Once the deployment of Apache Ranger is complete, we can manage our entire Hadoop infrastructure security using the web interface provided by Apache Ranger.

Login to UI

If you have not changed the default settings, Ranger runs on port `6080` by default in non-SSL Mode. Open up a web browser on the server where its installed on port `6080` (`http://<server-ip>:6080`) and you will be prompted with a screen like this:

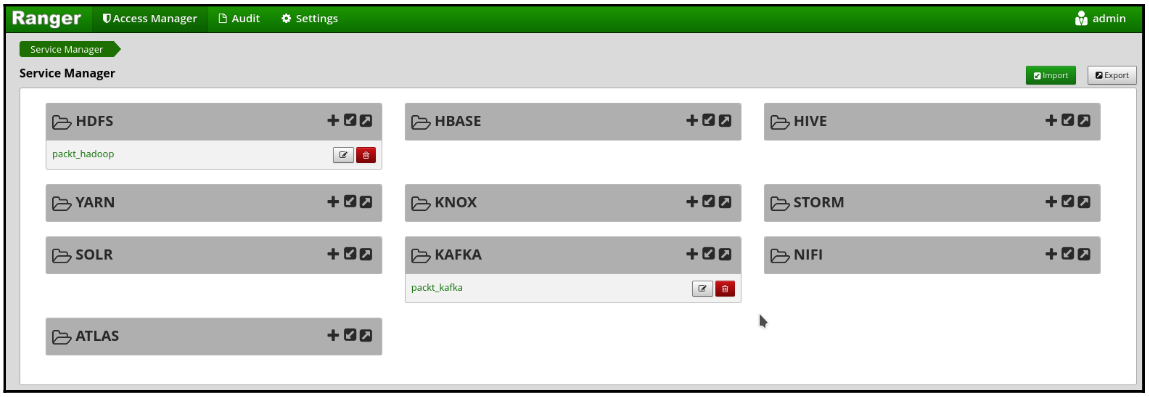


Log in with the default username `admin` and password `admin` (please change the password after you log in for the first time, for security reasons).

Once the login is successful, we are taken to the *Access manager* section.

Access manager

Access manager lets us define policies based on services and tags. This screenshot shows the default list of services and the configured policies:



As you can see, there is already a policy defined for **HDFS** service and **KAFKA** service as they are already installed in the Ambari setup.

When we want to define a new service, we can click on the **+** icon and define the service details.

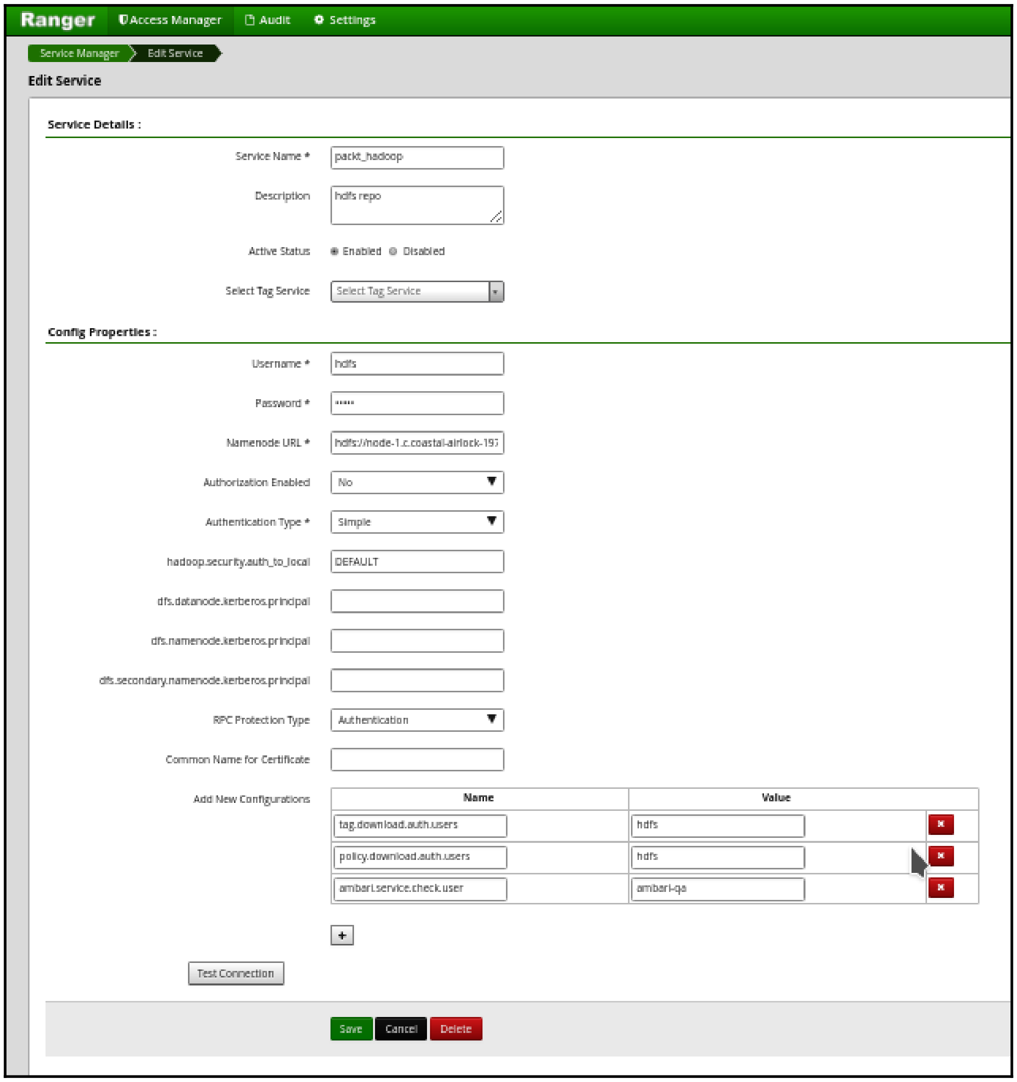
Service details

Before we start defining the authorization rules for the service, we need to define a service and then add authorization policies to the service. These are the mandatory properties which are needed to define a service from the UI:

UI Element name	Description
Service Name	Name of the service as defined in agent configuration
Username	Name of the service user
Password	Password for the service user
Namenode URL	URL to the namenode

A new service can be defined by clicking on the **+** icon below the application (for example, `HDFS`, `Kafka`, and so on)

After that, the service definition screen looks like this:



Edit Service

Service Details :

Service Name *	packt_hadoop
Description	hdfs repo
Active Status	[✓ Enabled] Disabled
Select Tag Service	Select Tag Service

Config Properties :

Username *	hdfs
Password *	*****
Namenode URL *	hdfs://node-1.c.coastal-airlock-197
Authorization Enabled	No
Authentication Type *	Simple
hadoop.security.auth_to_local	DEFAULT
dfs.datanode.kerberos.principal	
dfs.namenode.kerberos.principal	
dfs.secondary.namenode.kerberos.principal	
RPC Protection Type	Authentication
Common Name for Certificate	

Add New Configurations

Name	Value
tag.download.auth.users	hdfs
policy.download.auth.users	hdfs
ambari.service.check.user	ambari-qa

Screenshot of the service definition screen after defining new services

We need to fill in all the necessary values for our service definition and hit save. Later, we need to add policies to this service to access enforcement and auditing.

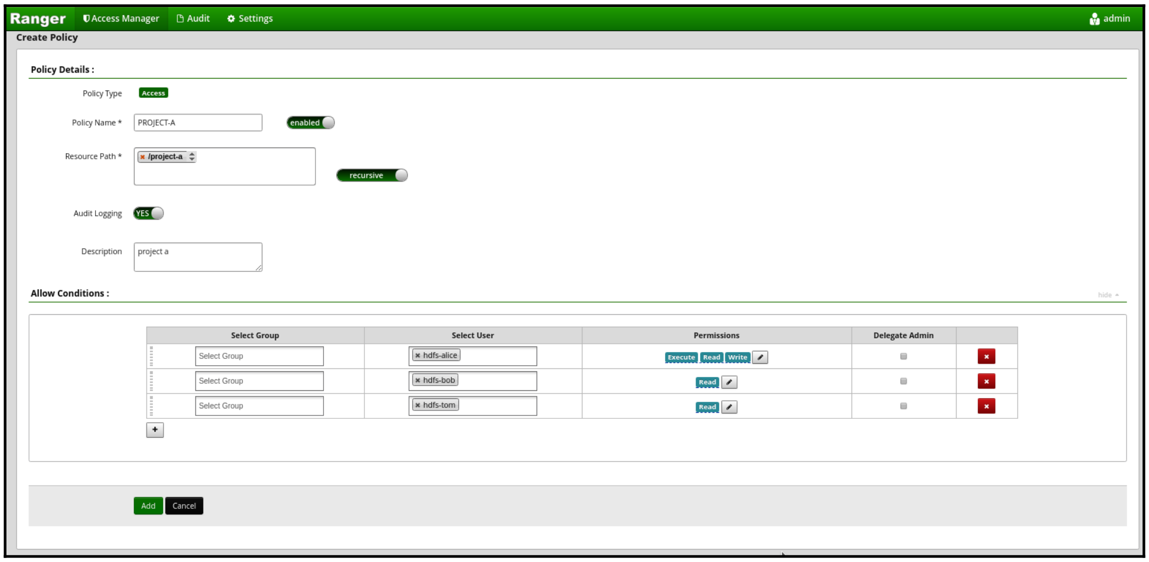
Policy definition and auditing for HDFS

For every service in Ranger, we can associate different policies to the resources in the service. In case of HDFS, the resources will be the file/directory paths.

In this section, we will define a new policy for an HDFS path called projects for three users: `hdfs-alice`, `hdfs-bob`, and `hdfs-tom`. Where only `hdfs-alice` is allowed all permissions and rest of the users have only read access.

We will see how Ranger enforces access restrictions once the policy is in place.

Let's see the screen for the policy creation:



Create Policy

Policy Details :

Policy Type: Access

Policy Name *: PROJECT-A (enabled)

Resource Path *: /project-a (recursive)

Audit Logging: YES

Description: project a

Allow Conditions :

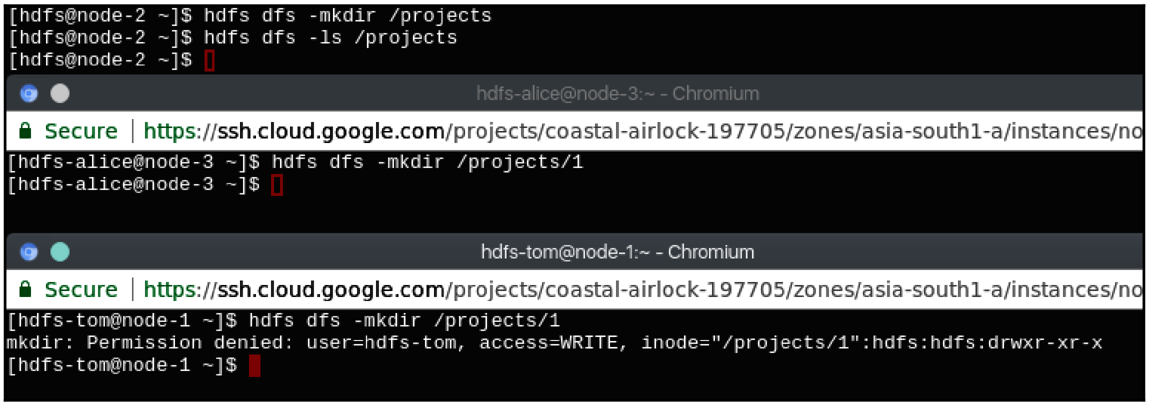
Select Group	Select User	Permissions	Delegate Admin
Select Group	hdfs-alice	Execute Read Write	☐
Select Group	hdfs-bob	Read	☐
Select Group	hdfs-tom	Read	☐

Screenshot showing how Ranger enforces access restrictions

Once we hit the **Add** button, this policy is registered and added under the current service.

Now, let's get back to the Unix terminal and see how Ranger enforces the policies.

This screen shows how `hdfs` and `hdfs-alice` users are allowed to create directories `/projects` and `/projects/1`, but how this is denied for `hdfs-tom`:



```
[hdfs@node-2 ~]$ hdfs dfs -mkdir /projects
[hdfs@node-2 ~]$ hdfs dfs -ls /projects
[hdfs@node-2 ~]$
```

hdfs-alice@node-3:~ - Chromium
Secure | https://ssh.cloud.google.com/projects/coastal-airlock-197705/zones/asia-south1-a/instances/no

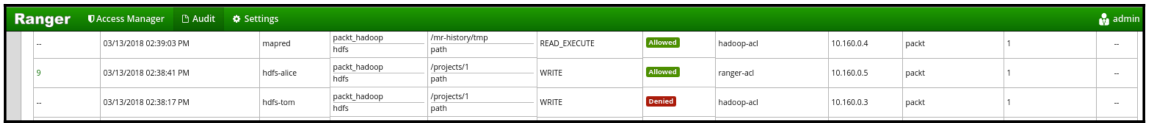
```
[hdfs-alice@node-3 ~]$ hdfs dfs -mkdir /projects/1
[hdfs-alice@node-3 ~]$
```

hdfs-tom@node-1:~ - Chromium
Secure | https://ssh.cloud.google.com/projects/coastal-airlock-197705/zones/asia-south1-a/instances/no

```
[hdfs-tom@node-1 ~]$ hdfs dfs -mkdir /projects/1
mkdir: Permission denied: user=hdfs-tom, access=WRITE, inode="/projects/1":hdfs:hdfs:drwxr-xr-x
[hdfs-tom@node-1 ~]$
```

Apache Ranger also has an audit section in the web interface, where we can see these access patterns.

This screen shows that `hdfs-tom` is denied and `hdfs-alice` is granted access by the policy:



--	03/13/2018 02:39:03 PM	mapred	packt_hadoop hdfs	/mr-history/tmp path	READ_EXECUTE	Allowed	hadoop-acl	10.160.0.4	packt	1	--
9	03/13/2018 02:38:41 PM	hdfs-alice	packt_hadoop hdfs	/projects/1 path	WRITE	Allowed	ranger-acl	10.160.0.5	packt	1	--
--	03/13/2018 02:38:17 PM	hdfs-tom	packt_hadoop hdfs	/projects/1 path	WRITE	Denied	hadoop-acl	10.160.0.3	packt	1	--

Screenshot showing access denied to hdfs-tom and access granted to hdfs-alice by the policy

Like this, we can define our own policies and customize how `hdfs` should allow/deny access to several resources.

The power and flexibility of Ranger comes from the its configurability. There is no need for any configuration files and restarts of applications for the access control to play a significant role.

Summary

In this chapter, we learned about the different data life cycle stages, including when data is created, shared, maintained, archived, retained, and deleted.

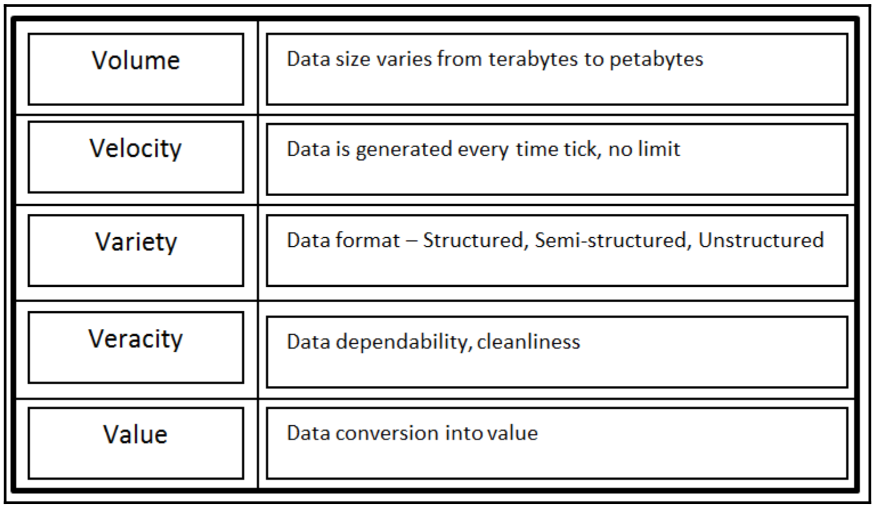
This chapter gave you a detailed understanding of how big data is managed, considering the fact that it is either unstructured or semi-structured and it has a fast arrival rate and large volume.

As the complexity of the infrastructure that generates and uses data in business organizations has increased drastically, it has become imperative to secure your data properly. This chapter further covered data security tools, such as Apache Ranger, and patterns to help us learn how to have control over the access patterns of data.

In the next chapter, we will take a look at Hadoop installation, its architecture and key components.

3
Hadoop Design Consideration

Big data does not necessarily mean huge data. If a dataset is small, it's very easy to analyze it. We can load it on to an Excel spreadsheet and do the required calculations. But, as the volume of data gets bigger, we have to find other alternatives to process it. We may have to load it to an RDMBS table and run a SQL query to find the trend and patterns on the given structure. Further, if the dataset format changes to something like email, then loading to RDBMS becomes a huge challenge. To add more complexity to it, if the data speed changes to something like real time, it becomes almost impossible to analyze the given dataset with traditional RDBMS-based tools. In the modern world, the term *big data* can be expressed using the five most famous *V*s. Following is the explanation of each *V* in a nutshell.



Volume	Data size varies from terabytes to petabytes
Velocity	Data is generated every time tick, no limit
Variety	Data format – Structured, Semi-structured, Unstructured
Veracity	Data dependability, cleanliness
Value	Data conversion into value

In this chapter, we will cover the following topics:

- Data structure principles
- Installing Hadoop cluster
- Exploring Hadoop architecture
- Introducing YARN
- Hadoop cluster composition
- Hadoop file formats

Understanding data structure principles

Let's go through some important data architecture principles:

- **Data is an asset to an enterprise**: Data has a measurable value. It provides some real value to the enterprise. In modern times, data is treated like real gold.
- **Data is shared enterprise-wide**: Data is captured only once and then used and analyzed many times. Multiple users access the same data for different uses cases and requirements.
- **Data governance**: Data is governed to ensure data quality.
- **Data management**: Data needs to be managed to attain enterprise objectives.
- **Data access**: All users should have access to data.
- **Data security**: Data should be properly secured and protected.
- **Data definition**: Each attribute of the data needs to be consistently defined enterprise-wide.

Now that we know the basics of big data and its principles, let's get into some real action.

Installing Hadoop cluster

The following steps need to be performed in order to install Hadoop cluster. As the time of writing this book, Hadoop Version 2.7.3 is a stable release. We will install it.

1. Check the Java version using the following command:

```
Java -version
Java(TM) SE Runtime Environment (build 1.8.0_144-b01)
Java HotSpot(TM) 64-Bit Server VM (build 25.144-b01, mixed mode)
You need to have Java 1.6 onwards
```

2. Create a Hadoop user account on all the servers, including all NameNodes and DataNodes with the help of the following commands:

   ```
   useradd hadoop
   passwd hadoop1
   ```

   Assume that we have four servers and we have to create a Hadoop cluster using all four servers. The IPs of these four servers are as follows: `192.168.11.1`, `192.168.11.2`, `192.168.11.3`, and `192.168.11.4`. Out of these four servers, we will first use a server as a master server (NameNode) and all remaining servers will be used as slaves (DataNodes).

3. On both servers, NameNode and DataNodes, change the `/etc/hosts` file using the following command:

   ```
   vi /etc/hosts--
   ```

4. Then add the following to all files on all servers:

   ```
   NameNode 192.168.11.1
   DataNode1 192.168.11.2
   DataNode2 192.168.11.3
   DataNode3 192.168.11.4
   ```

5. Now, set up SSH on NamesNodes and DataNodes:

   ```
   su - hadoop
   ssh-keygen -t rsa
   ssh-copy-id -i ~/.ssh/id_ras.pub hadoop@namenode
   ssh-copy-id -i ~/.ssh/id_ras.pub hadoop@datanode1
   ssh-copy-id -i ~/.ssh/id_ras.pub hadoop@datanode2
   ssh-copy-id -i ~/.ssh/id_ras.pub hadoop@datanode3
   chmod 0600 ~/.ssh/authorized_keys
   exit
   ```

6. Download and install Hadoop on NameNode and all DataNodes:

   ```
   mkdir /opt/hadoop
   cd /opt/hadoop
   wget
   http://www-eu.apache.org/dist/hadoop/common/hadoop-2.7.3/hadoop-2.7
   .3.tar.gz
   tar -xvf hadoop-2.7.3.tar.gz
   mv Hadoop-2.7.3 hadoop
   chown -R hadoop /opt/hadoop
   cd /opt/hadoop/Hadoop
   ```

Configuring Hadoop on NameNode

Log in to NameNode:

```
cd /opt/Hadoop/conf
vi core-site.xml
```

Find and change the following properties with these values:

Filename	**Property name**	**Property value**
`core-site.xml`	`fs.default.name`	`hdfs://namenode:9000/`
	`dfs.permissions`	`False`
`hdfs-site.xml`	`dfs.data.dir`	`/opt/hadoop/hadoop/dfs/namenode/data`
	`dfs.name.dir`	`/opt/hadoop/hadoop/dfs/namenode`
	`dfs.replication`	`1`
`mapred-site.xml`	`mapred.job.tracker`	`namenode:9001`

```
vi masters
namenode
vi slaves
datanode1
datanode2
datanode3
```

Format NameNode

The following code is used to format the NameNode:

```
cd /opt/Hadoop/Hadoop/bin
hadoop -namenode  -format
```

Start all services

We start all the services with the following line of code:

```
./start-all.sh
```

For details about how to set up a Hadoop single-node and multi-node cluster, please use the following link: `https://hadoop.apache.org/docs/r2.7.0/hadoop-project-dist/hadoop-common/ClusterSetup.html`.

Exploring HDFS architecture

The HDFS architecture is based on master and slave patterns. NameNode is a master node and all DataNodes are SlaveNodes. Following are some important points to be noted about these two nodes.

Defining NameNode

The NameNode is a master node of all DataNodes in the Hadoop cluster. It stores only the metadata of files and directories stored in the form of a tree. The important point is NameNode never stores any other data other than metadata. NameNode keeps track of all data written to DataNodes in the form of blocks. The default block size is 256 MB (which is configurable). Without the NameNode, the data on the DataNodes filesystem cannot be read. The metadata is stored locally on the NameNode using two files—filesystem namespace image file, FSImage, and edit logs. FSImage is the snapshot of the filesystem from the start of the NameNode edit logs—all the changes of the filesystem since the NameNode started, when the NameNode starts, it reads FSImage file and edits log files. All the transactions (edits) are merged into the FSImage file. The FSImage file is written to disk and a new, empty edits log file is created to log all the edits. Since NameNode is not restarted very often, the edits log file becomes very large and unmanageable. When NameNode is restarted, it takes a very long time to restart it as all the edits need to be applied to the FSImage file. In the event of NameNode crashing, all the metadata in the edits log file will not be written to the FSImage file and will be lost.

Secondary NameNode

The name secondary NameNode is confusing. It does not act as a NameNode. Its main function is to get the filesystem changes from the NameNode and merge it to NameNode FSImage at regular intervals. Writing edits log file changes to FSImage are called **commits**. Regular commits help to reduce the NameNode start time. The secondary NameNode is also known as the commit node.

NameNode safe mode

It is a read-only mode for the HDFS cluster. Clients are not allowed any modifications to the filesystem or blocks. During startup, NameNode automatically starts in safe mode, applies edits to FSImage, disables safe mode automatically, and restarts in normal mode.

DataNode

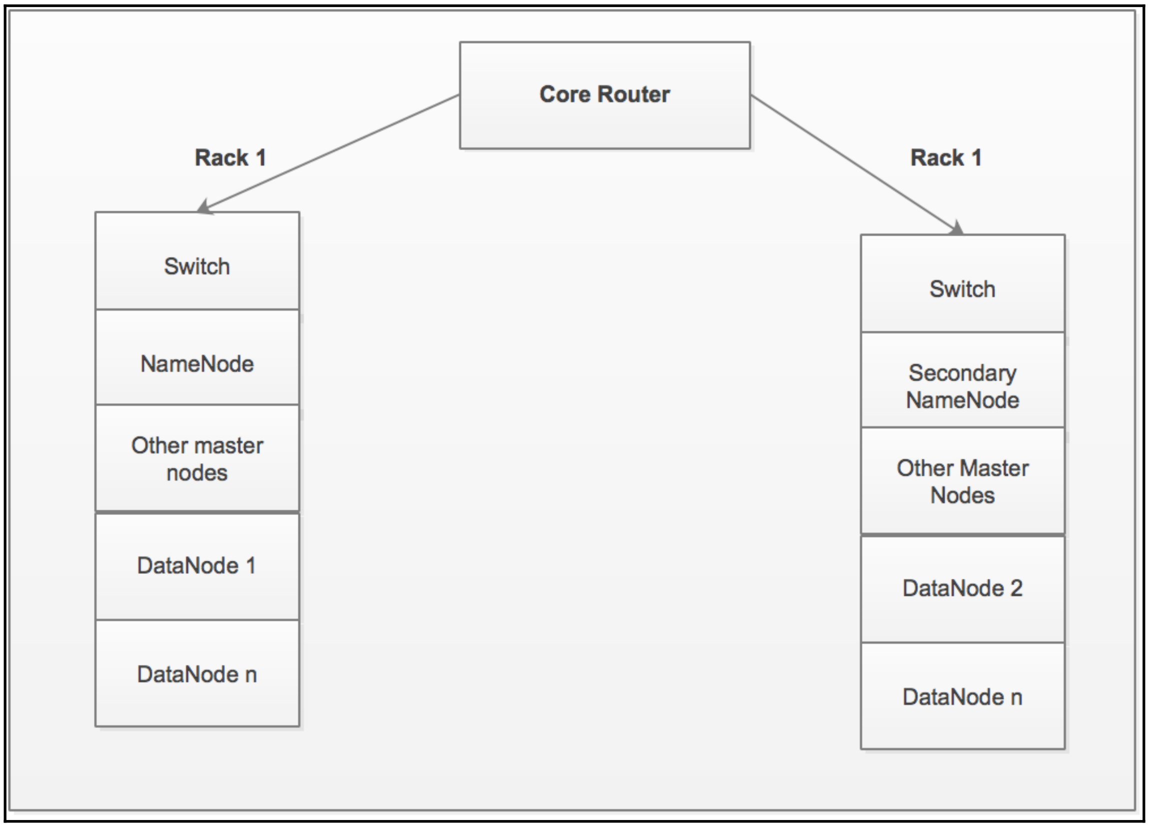
DataNodes are the workhorses of the Hadoop cluster. Their main function is to store and retrieve data in the form of blocks. They always communicate their status to the NameNode in the form of heartbeats. That's how NameNode keeps track of any DataNodes, whether they are alive or dead. DataNodes keep three copies of the blocks known and the replication factor. DataNodes communicate with other DataNodes to copy data blocks to maintain data replication.

Data replication

HDFS architecture supports placing very large files across the machines in a cluster. Each file is stored as a series of blocks. In order to ensure fault tolerance, each block is replicated three times to three different machines. It is known as a replication factor, which can be changed at the cluster level or at the individual file level. It is a NameNode that makes all the decisions related to block replication. NameNode gets heartbeat and block reports from each DataNode. Heartbeat makes sure that the DataNode is alive. A block report contains a list of all blocks on a DataNode.

Rack awareness

HDFS block placement will use rack awareness for fault tolerance by placing one block replica on a different rack, as shown in the following diagram:



Let's understand the figure in detail:

- The first replica is placed on the same rack as the initiating request DataNode, for example, Rack 1 and DataNode 1
- The second replica is placed on any DataNode of another rack, for example, Rack 2, DataNode 2
- The third replica is placed on any DataNode of the same rack, for example, Rack 2, DataNode 3

A custom rack topology script, which contains an algorithm to select appropriate DataNodes, can be developed using a Unix shell, Java, or Python. It can be activated on the cluster by changing the `topology.script.file.name` parameter in `Core-site.xml` file.

HDFS WebUI

The following table shows the services in the HDFS WebUI:

Service	Protocol	Port	URL
NameNode WebUI	HTTP	`50070`	`http://namenode:50070/`
DataNode WebUI	HTTP	`50075`	`http://datanode:50075/`
Secondary NameNode	HTTP	`50090`	`http://Snamenode:50090/`

Introducing YARN

The **Yet Another Resource Negotiator** (**YARN**) separates the resource management, scheduling, and processing components. It helps to achieve 100% resource utilization of the cluster resources. YARN manages the CPU and memory of the cluster based on the Hadoop scheduler policy. YARN supports any type of application and is not restricted to just MapReduce. It supports applications written in any type of language, provided binaries can be installed on the Hadoop cluster.

YARN architecture

Let's understand the YARN architecture in detail in the following sections.

Resource manager

The resource manager is responsible for tracking the resources in a cluster and scheduling applications. The resource manager has two main components: the scheduler and the applications manager.

Node manager

The node manager is responsible for launching and managing containers on a node. Containers execute tasks as specified by the application master. It acts as a slave for the resource manager. Each node manager tracks the available data processing resources on its SlaveNode and sends regular reports to the resource manager. The processing resources in a Hadoop cluster are consumed in byte-size pieces called **containers**.

Configuration of YARN

You can perform the following steps for the configuration of YARN:

1. Start Hadoop NameNode, secondary NameNode, and DataNode
2. Alter `yarn-env.sh`.

TIP: Find corresponding XML files based on your Hadoop installation.

3. Add the following under the definition of `YARN_CONF_DIR`:

```
export HADOOP_CONF_DIR="${HADOOP_CONF_DIR:-$YARN_HOME/etc/hadoop}"
export HADOOP_COMMON_HOME="${HADOOP_COMMON_HOME:-$YARN_HOME}"
export HADOOP_HDFS_HOME="${HADOOP_HDFS_HOME:-$YARN_HOME}"
```

4. Alter `yarn-site.xml`:

```
<?xml version="1.0"?>
<configuration>
  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce.shuffle</value>
  </property>
  <property>
    <name>yarn.nodemanager.aux-
services.mapreduce.shuffle.class</name>
    <value>org.apache.hadoop.mapred.ShuffleHandler</value>
  </property>
</configuration>
```

5. Alter `mapred-site.xml`:

```
<?xml version="1.0"?>
<?xml-stylesheet href="configuration.xsl"?>
<configuration>
  <property>
    <name>mapreduce.framework.name </name>
    <value>yarn</value>
  </property>
</configuration>
```

6. Start the YARN services:

```
yarn resourcemanager
yarn nodemanager
```

Configuring HDFS high availability

Let's take a look at the changes brought about in Hadoop over time.

During Hadoop 1.x

Hadoop 1.x started with the architecture of a single NameNode. All DataNodes used to send their block reports to that single NameNode. There was a secondary NameNode in the architecture, but its sole responsibility was to merge all edits to FSImage. With this architecture, the NameNode became the **single point of failure** (**SPOF**). Since it has all the metadata of all the DataNodes of the Hadoop cluster, in the event of NameNode crash, the Hadoop cluster becomes unavailable till the next restart of NameNode repair. If the NameNode cannot be recovered, then all the data in all the DataNodes would be completely lost. In the event of shutting down NameNode for planned maintenance, the HDFS becomes unavailable for normal use. Hence, it was necessary to protect the existing NameNode by taking frequent backups of the NameNode filesystem to minimize data loss.

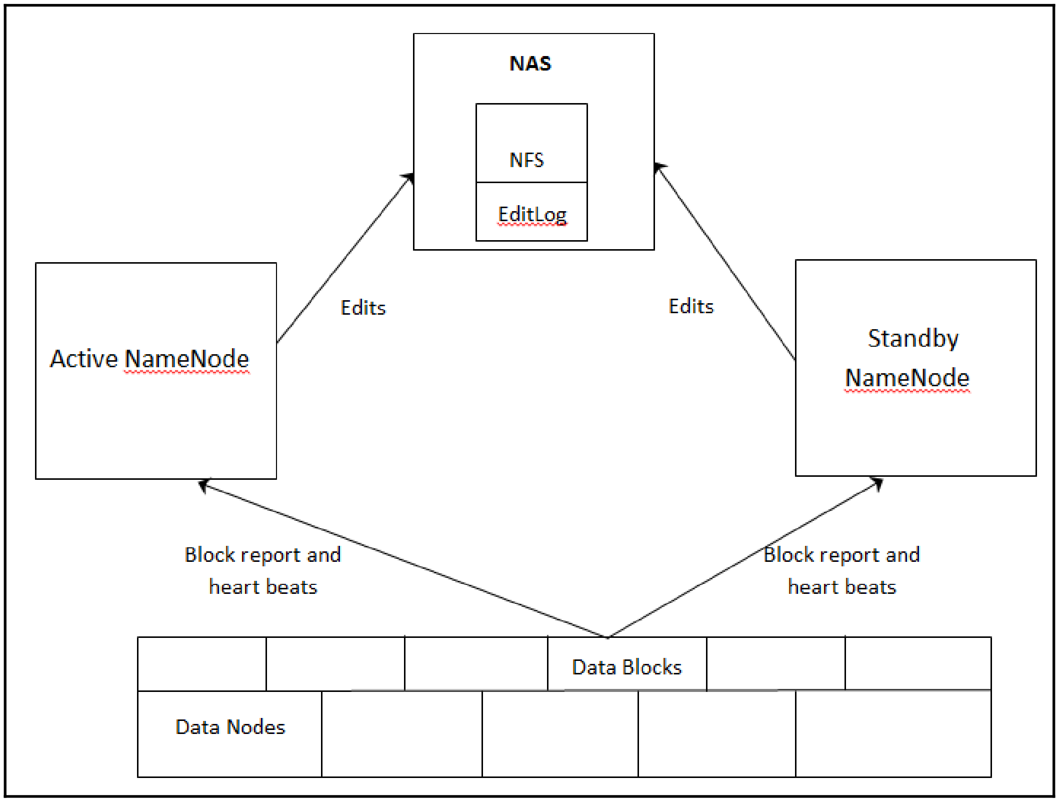
During Hadoop 2.x and onwards

In order to overcome HDFS **high availability** (**HA**) problems and make NameNode a SPOF, the architecture has changed. The new architecture provides a running of two redundant NameNodes in the same cluster in an active/passive configuration with a hot standby. This allows a fast failover to a new NameNode in the event of a machine crashing, or a graceful administrator-initiated failover for the purpose of planned maintenance. The following two architectural options are provided for HDFS HA:

- Using shared storage
- Using quorum journal manager

HDFS HA cluster using NFS

The following diagram depicts the HDFS HA cluster using NFS for shared storage required by the NameNodes architecture:



Important architecture points

Following are some important points to remember about the HDFS HA using shared storage architecture:

- In the cluster, there are two separate machines: active state NameNode and standby state NameNode.
- At any given point in time, one-and-only, one of the NameNodes is in the active state, and the other is in the standby state.

- The active NameNode manages the requests from all client DataNodes in the cluster, while the standby remains a slave.
- All the DataNodes are configured in such a way that they send their block reports and heartbeats to both the active and standby NameNodes.
- The standby NameNode keeps its state synchronized with the active NameNode.
- Active and standby nodes both have access to a filesystem on a shared storage device (for example, an NFS mount from a NAS)
- When a client makes any filesystem change, the active NameNode makes the corresponding change (edits) to the edit log file residing on the network shared directory.
- The standby NameNode makes all the corresponding changes to its own namespace. That way, it remains in sync with the active NameNode.
- In the event of the active NameNode being unavailable, the standby NameNode makes sure that it absorbs all the changes (edits) from the shared network directory and promotes itself to an active NameNode.
- The Hadoop administrator should apply the fencing method to the shared storage to avoid a scenario that makes both the NameNodes active at a given time. In the event of failover, the fencing method cuts the access to the previous active NameNode to make any changes to the shared storage to ensure smooth failover to standby NameNode. After that, the standby NameNode becomes the active NameNode.

Configuration of HA NameNodes with shared storage

Add the following properties to the `hdfs-site.xml`:

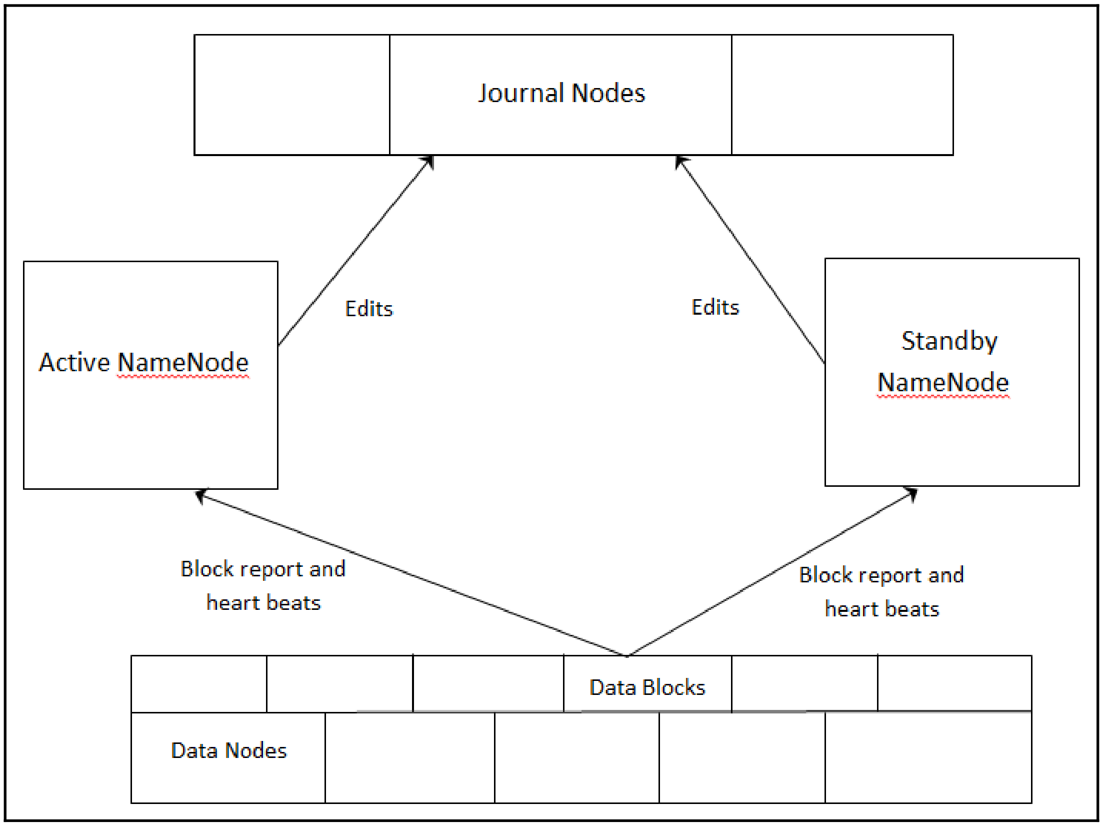
Property	**Value**
`dfs.nameservices`	`cluster_name`
`dfs.ha.namenodes.cluster_name`	`NN1,NN2`
`dfs.namenode.rpc-address.cluster_name.NN1`	`machine1:8020`
`dfs.namenode.rpc-address.cluster_name.NN2`	`machine2:8020`
`dfs.namenode.http-address.cluster_name.NN1`	`machine1:50070`
`dfs.namenode.http-address.cluster_name.NN2`	`machine2:50070`
`dfs.namenode.shared.edits.dir`	`file:///mnt/filer1/dfs/ha-name-dir-shared`
`dfs.client.failover.proxy.provider.cluster_name`	`org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider`
`dfs.ha.fencing.methods`	`sshfence`
`dfs.ha.fencing.ssh.private-key-files`	`/home/myuser/.ssh/id_rsa`
`dfs.ha.fencing.methods`	`sshfence([[username][:port]])`
`dfs.ha.fencing.ssh.connect-timeout`	`30000`

Add the following properties to `core-site.xml`:

Property	**Value**
`fs.defaultFS`	`hdfs://cluster_name`

HDFS HA cluster using the quorum journal manager

The following diagram depicts the **quorum journal manager** (**QJM**) architecture to share edit logs between the active and standby NameNodes:



Important architecture points

Following are some important points to remember about the HDFS HA using the QJM architecture:

- In the cluster, there are two separate machines—the active state NameNode and standby state NameNode.
- At any point in time, exactly one of the NameNodes is in an active state, and the other is in a standby state.
- The active NameNode manages the requests from all client DataNodes in the cluster, while the standby remains a slave.
- All the DataNodes are configured in such a way that they send their block reports and heartbeats to both active and standby NameNodes.
- Both NameNodes, active and standby, remain synchronized with each other by communicating with a group of separate daemons called **JournalNodes** (**JNs**).
- When a client makes any filesystem change, the active NameNode durably logs a record of the modification to the majority of these JNs.
- The standby node immediately applies those changes to its own namespace by communicating with JNs.
- In the event of the active NameNode being unavailable, the standby NameNode makes sure that it absorbs all the changes (edits) from JNs and promotes itself as an active NameNode.
- To avoid a scenario that makes both the NameNodes active at a given time, the JNs will only ever allow a single NameNode to be a writer at a time. This allows the new active NameNode to safely proceed with failover.

Configuration of HA NameNodes with QJM

Add the following properties to `hdfs-site.xml`:

Property	**Value**
`dfs.nameservices`	`cluster_name`
`dfs.ha.namenodes.cluster_name`	`NN1,NN2`
`dfs.namenode.rpc-address.cluster_name.NN1`	`machine1:8020`
`dfs.namenode.rpc-address.cluster_name.NN2`	`machine2:8020`
`dfs.namenode.http-address.cluster_name.NN1`	`machine1:50070`
`dfs.namenode.http-address.cluster_name.NN2`	`machine2:50070`
`dfs.namenode.shared.edits.dir`	`qjournal://node1:8485;node2:8485;node3:8485/cluster_name`
`dfs.client.failover.proxy.provider.cluster_name`	`org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider`
`dfs.ha.fencing.methods`	`sshfence`

`dfs.ha.fencing.ssh.private-key-files`	`/home/myuser/.ssh/id_rsa`
`dfs.ha.fencing.methods`	`sshfence([[username][:port]])`
`dfs.ha.fencing.ssh.connect-timeout`	`30000`

Add the following properties to `core-site.xml`:

Property	**Value**
`fs.defaultFS`	`hdfs://cluster_name`
`dfs.journalnode.edits.dir`	`/path/to/journal/node/local/datat`

Automatic failover

It's very important to know that the above two architectures support only manual failover. In order to do automatic failover, we have to introduce two more components a ZooKeeper quorum, and the **ZKFailoverController** (**ZKFC**) process, and more configuration changes.

Important architecture points

- Each NameNode, active and standby, runs the ZKFC process.
- The state of the NameNode is monitored and managed by the ZKFC.
- The ZKFC pings its local NameNode periodically to make sure that that the NameNode is alive. If it doesn't get the ping back, it will mark that NameNode unhealthy.
- The healthy NameNode holds a special lock. If the NameNode becomes unhealthy, that lock will be automatically deleted.
- If the local NameNode is healthy, and the ZKFC sees the lock is not currently held by any other NameNode, it will try to acquire the lock. If it is successful in acquiring the lock, then it has won the election. It is now the responsibility of this NameNode to run a failover to make its local NameNode active.

Configuring automatic failover

Add the following properties to `hdfs-site.xml` to configure automatic failover:

Property	**Value**
`dfs.ha.automatic-failover.enabled`	`true`
`ha.zookeeper.quorum`	`zk1:2181,zk2:2181,zk3:2181`

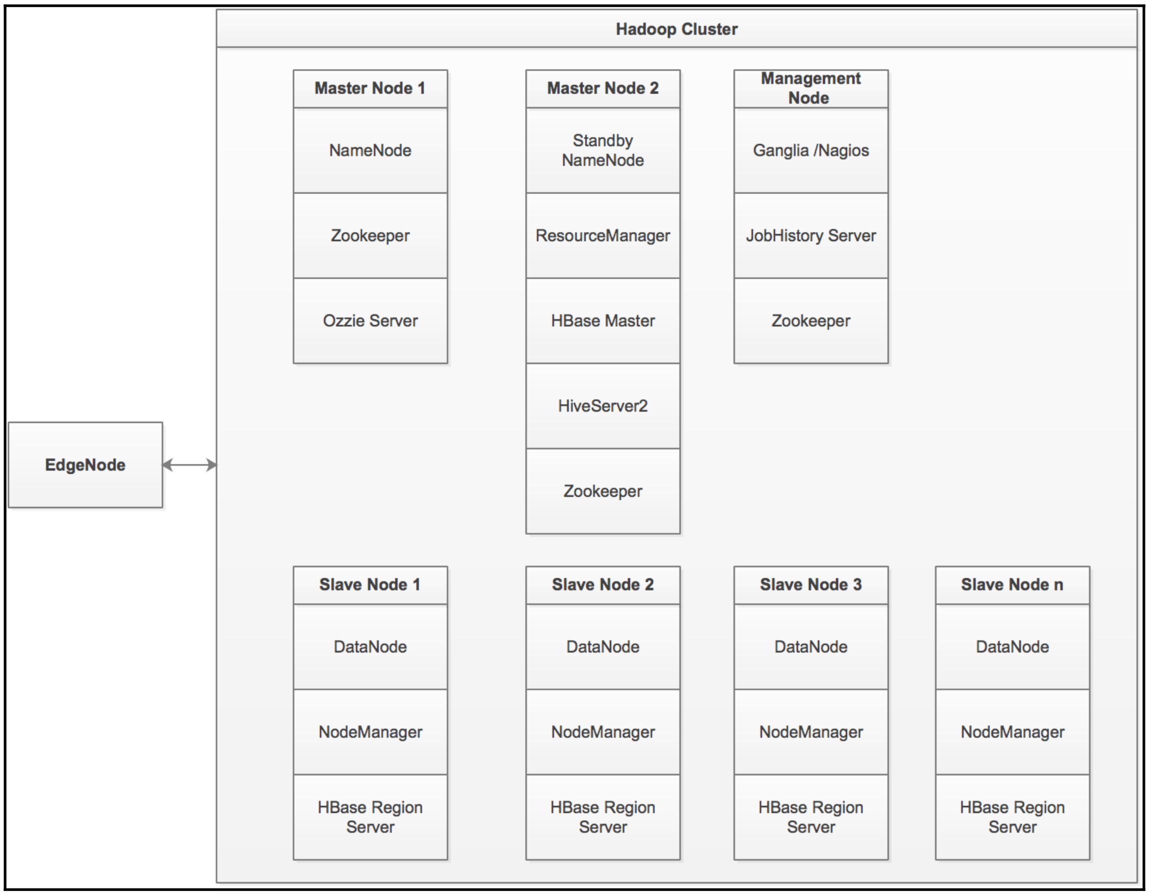
Hadoop cluster composition

As we know, a Hadoop cluster consists of master and slave servers: MasterNodes—to manage the infrastructure, and SlaveNodes—distributed data store and data processing. EdgeNodes are not a part of the Hadoop cluster. This machine is used to interact with the Hadoop cluster. Users are not given any permission to directly log in to any of the MasterNodes and DataNodes, but they can log in to the EdgeNode to run any jobs on the Hadoop cluster. No application data is stored on the EdgeNode. The data is always stored on the DataNodes on the Hadoop cluster. There can be more than one EdgeNode, depending on the number of users running jobs on the Hadoop cluster. If enough hardware is available, it's always better to host each master and DataNode on a separate machine. But, in a typical Hadoop cluster, there are three MasterNodes.

Please note that it is assumed that we are using HBase as a NoSQL datastore in our cluster.

Typical Hadoop cluster

The Hadoop cluster composition will look like the following:



The following are some hardware specifications to be taken into account:

- NameNode and standby NameNodes.
- The memory requirement depends on the number of files and block replicas to be created. Typically, at least 64 GB - 96 GB memory is recommended for NameNodes.
- NameNodes need reliable storage to host FSImage and edit logs. It is recommended that these MasterNodes should have at least 4 TB - 6 TB SAS storage. It is a good idea to have RAID 5 - 6 storage for NameNodes. If the cluster is a HA cluster, then plan your Hadoop cluster in such a way that JNs should be configured on the master node.

As far as processors are concerned, it is recommended to have at least 2 quad core CPUs running at 2 GHz, to handle messaging traffic for the MasterNodes.

- DataNodes/SlaveNodes should have at least 64 GB RAM per node. It is recommended that, typically, 2 GB - 3 GB memory is required for each Hadoop daemon, such as DataNode, node manager ZooKeeper, and so on; 5 GB for OS and other services; and 5 GB - 8 GB for each MapReduce task.
- DataNodes may have commodity storage with at least 8 TB - 10 TB disk storage with 7,200 RPM SATA drives. Hard disk configuration should be in **Just a Bunch Of Disks** (**JBOD**).
- It is recommended to have at least 8 processors—2.5 GHz cores and 24 cores CPUs for all DataNodes.
- It is recommended to have 1 GbE to 10 GbE network connectivity within each RACK. For all slaves, 1 GB network bandwidth, and for MasterNodes, 10 GB bandwidth is recommended.
- If you plan to expand your Hadoop cluster in future, you can also add additional machines.

Please read the following articles from Hortonworks and Cloudera for additional reference:

- `http://docs.hortonworks.com/HDPDocuments/HDP1/HDP-1.3.3/bk_cluster-planning-guide/content/conclusion.html`
- `http://blog.cloudera.com/blog/2013/08/how-to-select-the-right-hardware-for-your-new-hadoop-cluster/`

Best practices Hadoop deployment

Following are some best practices to be followed for Hadoop deployment:

- **Start small**: Like other software projects, an implementation Hadoop also involves risks and cost. It's always better to set up a small Hadoop cluster of four nodes. This small cluster can be set up as **proof of concept** (**POC**). Before using any Hadoop component, it can be added to the existing Hadoop POC cluster as **proof of technology** (**POT**). It allows the infrastructure and development team to understand big data project requirements. After successful completion of POC and POT, additional nodes can be added to the existing cluster.

- **Hadoop cluster monitoring**: Proper monitoring of the NameNode and all DataNodes is required to understand the health of the cluster. It helps to take corrective actions in the event of node problems. If a service goes down, timely action can help avoid big problems in the future. Setting up Gangalia and Nagios are popular choices to configure alerts and monitoring. In the case of the Hortonworks cluster, Ambari monitoring, and the Cloudera cluster, Cloudera (CDH) manager monitoring can be an easy setup.
- **Automated deployment**: Use of tools like Puppet or Chef is essential for Hadoop deployment. It becomes super easy and productive to deploy the Hadoop cluster with automated tools instead of manual deployment. Give importance to data analysis and data processing using available tools/components. Give preference to using Hive or Pig scripts for problem solving rather than writing heavy, custom MapReduce code. The goal should be to develop less and analyze more.
- **Implementation of HA**: While deciding about HA infrastructure and architecture, careful consideration should be given to any increase in demand and data growth. In the event of any failure or crash, the system should be able to recover itself or failover to another data center/site.
- **Security**: Data needs to be protected by creating users and groups, and mapping users to the groups. Setting appropriate permissions and enforcing strong passwords should lock each user group down.
- **Data protection**: The identification of sensitive data is critical before moving it to the Hadoop cluster. It's very important to understand privacy policies and government regulations for the better identification and mitigation of compliance exposure risks.

Hadoop file formats

In Hadoop, there are many file formats available. A user can select any format based on the use case. Each format has special features in terms of storage and performance. Let's discuss each file format in detail.

Text/CSV file

Text and CSV files are very common in Hadoop data processing algorithms. Each line in the file is treated as a new record. Typically, each line ends with the *n* character. These files do not support column headers. Hence, while processing, an extra line of the code is always required to remove column headings. CSV files are typically compressed using GZIP codec because they do not support block level compression; it adds to more processing costs. Needless to mention they do not support schema evolution.

JSON

The JSON format is becoming very popular in all modern programming languages. These files are collection name/value pairs. The JSON format is typically used in data exchange applications and it is treated as an object, record, struct, or an array. These files are text files and support schema evolutions. It's very easy to add or delete attributes from a JSON file. Like text/CSV files, JSON files do not support block-level compression.

Sequence file

A sequence file is a flat file consisting of binary key/value pairs. They are extensively used in MapReduce (`https://wiki.apache.org/hadoop/MapReduce`) as input/output formats. They are mostly used for intermediate data storage within a sequence of MapReduce jobs. Sequence files work well as containers for small files. If there are too many small files in HDFS, they can be packed in a sequence file to make file processing efficient. There are three formats of sequence files: uncompressed, record compressed, and block compressed key/value records. Sequence files support block-level compression but do not support schema evolution.

Avro

Avro is a widely used file type within the Hadoop community. It is popular because it helps schema evolution. It contains serialized data with a binary format. An Avro file is splittable and supports block compression. It contains data and metadata. It uses a separate JSON file to define the schema format. When Avro data is stored in a file, its schema is stored with it so that files may be processed later by any program. If the program reading the data expects a different schema, this can be easily resolved, since both schemas are present.

Parquet

Parquet stores nested data structures in a flat columnar format. Parquet is more efficient in terms of storage and performance than any row-level file formats. Parquet stores binary data in a column-oriented way. In the Parquet format, new columns are added at the end of the structure. Cloudera mainly supports this format for Impala implementation but is aggressively becoming popular recently. This format is good for SQL queries, which read particular columns from a wide table having many columns because only selective columns are read to reduce I/O cost.

ORC

ORC files are optimized record columnar file format and are the extended version of RC files. These are great for compression and are best suited for Hive SQL performance when Hive is reading, writing, and processing data to reduce access time and the storage space. These files do not support true schema evolution. They are mainly supported by Hortonworks and are not suitable for Impala SQL processing.

Which file format is better?

The answer is: it depends on your use cases. Generally, the criteria for selecting a file format is based on query-read and query-write performance. Also, it depends on which Hadoop distribution you are using. The ORC file format is the best for Hive and Tez using the Hortonworks distribution and a parquet file is recommended for Cloudera Impala implementations. For a use case involving schema evolution, Avro files are best suited. If you want to import data from RDBMS using Sqoop, text/CSV file format is the better choice. For storing map intermediate output, a sequence file is the ultimate choice.

Summary

In this chapter, the main objective was to learn about various Hadoop design alternatives. We've learned a lot when it comes to the Hadoop cluster and its best practices for deployment in a typical production environment. We started with a basic understanding about Hadoop and we proceeded to Hadoop configuration, installation, and HDFS architecture. We also learned about various techniques for achieving HDFS high availability. We also looked into YARN architecture. Finally, we looked at various file formats and how to choose one based on your use case.

In the next chapter, we will see how to ingest data into a newly created Hadoop cluster.

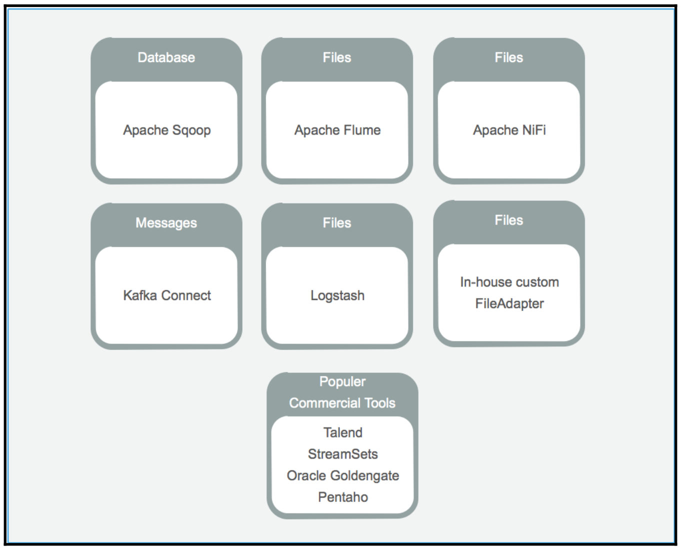
4
Data Movement Techniques

In the last chapter, we learned about how to create and configure a Hadoop cluster, HDFS architecture, various file formats, and the best practices for a Hadoop cluster. We also learned about Hadoop high availability techniques.

Since we now know how to create and configure a Hadoop cluster, in this chapter, we will learn about various techniques of data ingestion into a Hadoop cluster. We know about the advantages of Hadoop, but now, we need data in our Hadoop cluster to utilize its real power.

Data ingestion is considered the very first step in the Hadoop data life cycle. Data can be ingested into Hadoop as either a batch or a (real-time) stream of records. Hadoop is a complete ecosystem, and MapReduce is a batch ecosystem of Hadoop.

The following diagram shows various data ingestion tools:



We will learn about each tool in detail in the next few sections.

In this chapter, we will cover the following methods of transferring data to and from our Hadoop cluster:

- Apache Sqoop
- Apache Flume
- Apache NiFi
- Apache Kafka Connect

Batch processing versus real-time processing

Before we dive deep into different data ingestion techniques, let's discuss the difference between batch and real-time (stream) processing. The following explains the difference between these two ecosystems.

Batch processing

The following points describe the batch processing system:

- Very efficient in processing a high volume of data.
- All data processing steps (that is, data collection, data ingestion, data processing, and results presentation) are done as one single batch job.
- Throughput carries more importance than latency. Latency is always more than a single minute.
- Throughput directly depends on the size of the data and available computational system resources.
- Available tools include Apache Sqoop, MapReduce jobs, Spark jobs, Hadoop DistCp utility, and so on.

Real-time processing

The following points describe how real-time processing is different from batch processing:

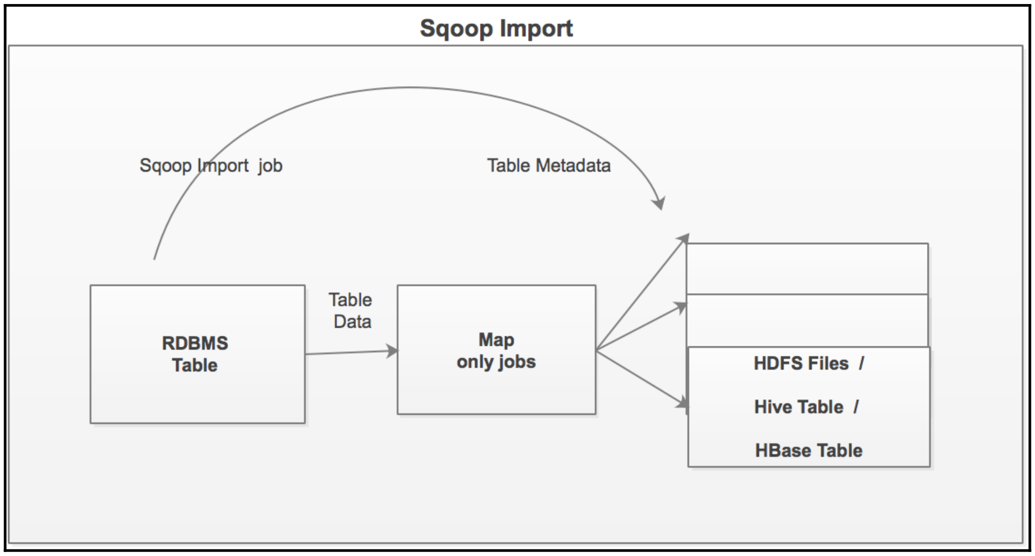
- Latency is extremely important, for example, less than one second
- Computation is relatively simple
- Data is processed as an independent unit
- Available tools include Apache Storm, Spark Streaming, Apache Fink, Apache Kafka, and so on

Apache Sqoop

Apache Sqoop is a tool designed for efficiently transferring bulk data between a Hadoop cluster and structured data stores, such as relational databases. In a typical use case, such as a data lake, there is always a need to import data from RDBMS-based data warehouse stores into the Hadoop cluster. After data import and data aggregation, the data needs to be exported back to RDBMS. Sqoop allows easy import and export of data from structured data stores like RDBMS, enterprise data warehouses, and NoSQL systems. With the help of Sqoop, data can be provisioned from external systems into a Hadoop cluster and populate tables in Hive and HBase. Sqoop uses a connector-based architecture, which supports plugins that provide connectivity to external systems. Internally, Sqoop uses MapReduce algorithms to import and export data. By default, all Sqoop jobs run four map jobs. We will see Sqoop import and export functions in detail in the next few sections.

Sqoop Import

The following diagram shows the **Sqoop Import** function to import data from an RDBMS table into a Hadoop cluster:



Import into HDFS

The following is a sample command to import data into HDFS:

```
$sqoop import -connect jdbc:mysql://localhost/dbname -table <table_name>
--username <username> --password >password> -m 4
```

The import is done in two steps, which are as follows.

1. Sqoop scans the database and collects the table metadata to be imported
2. Sqoop submits a map-only job and transfers the actual data using necessary metadata

The imported data is saved in HDFS folders. The user can specify alternative folders. The imported data is saved in a directory on HDFS, based on the table being imported. As is the case with most aspects of a Sqoop operation, the user can specify any alternative directory where the files should be populated. You can easily override the format in which data is copied over by explicitly specifying the field separator and record terminator characters. The user can use different formats, like Avro, ORC, Parquet, sequence files, text files, and so on, to store files onto HDFS, for example, importing a MySQL table to HDFS. The following is an example to import a MySQL table to HDFS:

```
$ mysql>  create database  sales;
$ mysql>  use sales;
$  mysql>   create table customer
    (cust_num int not null,cust_fname  varchar(30),cust_lname varchar
(30),cust_address  varchar (30),cust_city varchar (20),cust_state
varchar (3), cust_zip  varchar (6),primary key (cust_num));
$ ctrl-C   -- to exit from MySQL
```

On the Command Prompt, run the following `sqoop` command to import the MySQL sales database table, `customer`:

```
$  sqoop import --connect jdbc:mysql://127.0.0.1:3306/sales --username root
--password hadoop --table customer  --fields-terminated-by ","  --driver
com.mysql.jdbc.Driver --target-dir /user/data/customer
```

Verify the `customer` folder on HDFS as follows:

```
$ hadoop fs -ls /user/data/customerFound 5 items-rw-r--r--   1 root hdfs
0 2017-04-28 23:35 /user/data/customer/_SUCCESS-rw-r--r--   1 root hdfs
154 2017-04-28 23:35 /user/data/customer/part-m-00000-rw-r--r--   1 root
hdfs         95 2017-04-28 23:35 /user/data/customer/part-m-00001-rw-r--r--
1 root hdfs         96 2017-04-28 23:35 /user/data/customer/part-m-00002-
rw-r--r--   1 root hdfs        161 2017-04-28 23:35
/user/data/customer/part-m-00003
```

Let's create an external Hive table to verify the records, as shown in the following snippet:

```
$ hive$hive > CREATE EXTERNAL TABLE customer_H
(cust_num int,cust_fname  string,cust_lname  string,cust_address string,
cust_city  string,cust_state  string,cust_zip   string)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','LINES TERMINATED BY
'n'LOCATION '/user/data/customer';
$hive> select * from customer_H;
```

Custnum	Cust Fname	Cust Lname	Cust address	City	State	Zip
1	James	Butt	6649 N Blue Gum St	New Orleans	LA	70116
2	Art	Venere 8	W Cerritos Ave #54	Bridgeport	NJ	8014
3	Lenna	Paprocki	639 Main St	Anchorage	AK	99501
4	Donette	Foller	34 Center St	Hamilton	OH	45011
5	Simona	Morasca	3 Mcauley Dr	Ashland	OH	44805
6	Mitsue	Tollner	7 Eads St	Chicago	IL	60632
7	Leota	Dilliard	7 W Jackson Blvd	San Jose	CA	95111
8	Sage	Wieser	5 Boston Ave #88	Sioux Falls	SD	57105
9	Kris	Marrier	228 Runamuck Pl #2808	Baltimore	MD	21224
10	Minna	Amigon	2371 Jerrold Ave	Kulpsville	PA	19443

The following is an example of importing a MySQL table to Hive:

```
$ sqoop import --connect jdbc:mysql://127.0.0.1:3306/sales --username root
--password hadoop --table customer  --driver com.mysql.jdbc.Driver --m 1 --
hive-import  --hive-table customor_H
```

Verify the table Hive:

```
$hive$use default;
$ show tables;
```

You will see that the customer_H table is created under a default database. If you want to create the customer_H table under a different database, for example, a sales database, you have to create the sales database in advance. Also, you have to change the -hive-table parameter to the --hive-table sales cutomer_H incremental load (insert only). It's a typical data load requirement of loading only the incremental changes happening in the source table. Let's assume that a new customer, 11, is inserted into the source customer MySQL table:

```
insert into customer values (11,'Abel','Maclead','25 E 75th St #69','Los
Angeles','CA','90034');
```

To accommodate only the new record (that is, customer 11), we have to add a few additional parameters to our original sqoop command. The new sqoop command is as follows:

```
sqoop import --connect jdbc:mysql://127.0.0.1:3306/sales --username root --
password hadoop --table customer  --driver com.mysql.jdbc.Driver --
incremental append --check-column cust_num
      --last-value 10
    --m 1 --split-by cust_state --target-dir /user/data/customer
```

After running this command, Sqoop will pick up only the new row (that is, cust_num, which is 11):

```
$hive> select * from  customer_H;
```

Custnum	Cust Fname	Cust Lname	Cust address	City	State	Zip
1	James	Butt	6649 N Blue Gum St	New Orleans	LA	70116
2	Art	Venere 8	W Cerritos Ave #54	Bridgeport	NJ	8014
3	Lenna	Paprocki	639 Main St	Anchorage	AK	99501
4	Donette	Foller	34 Center St	Hamilton	OH	45011
5	Simona	Morasca	3 Mcauley Dr	Ashland	OH	44805
6	Mitsue	Tollner	7 Eads St	Chicago	IL	60632
7	Leota	Dilliard	7 W Jackson Blvd	San Jose	CA	95111
8	Sage	Wieser	5 Boston Ave #88	Sioux Falls	SD	57105
9	Kris	Marrier	228 Runamuck Pl #2808	Baltimore	MD	21224
10	Minna	Amigon	2371 Jerrold Ave	Kulpsville	PA	19443

11	Abel	Maclead	25 E 75th St #69	Los Angeles	CA	90034

For incremental load we cannot update the data directly using Sqoop import.

Please follow the steps in the given link for row-level updates: `http://hortonworks.com/blog/four-step-strategy-incremental-updates-hive/`.
Now, let's look at an example of importing a subset of a MySQL table into Hive. The following command shows how to import only a subset of a `customer` table in MySQL into Hive. For example, we have import-only customer data of `State = "OH"`:

```
$ sqoop import --connect jdbc:mysql://127.0.0.1:3306/sales --username root
--password hadoop --table sales.customer  --driver com.mysql.jdbc.Driver --
m 1 --where "city = 'OH' --hive-import  --hive-table customer_H_1$ hive>
select * from customer_H_1;
```

Custnum	Cust Fname	Cust Lname	Cust address	City	State	Zip
4	Donette	Foller	34 Center St	Hamilton	OH	45011
5	Simona	Morasca	3 Mcauley Dr	Ashland	OH	44805

Import a MySQL table into an HBase table

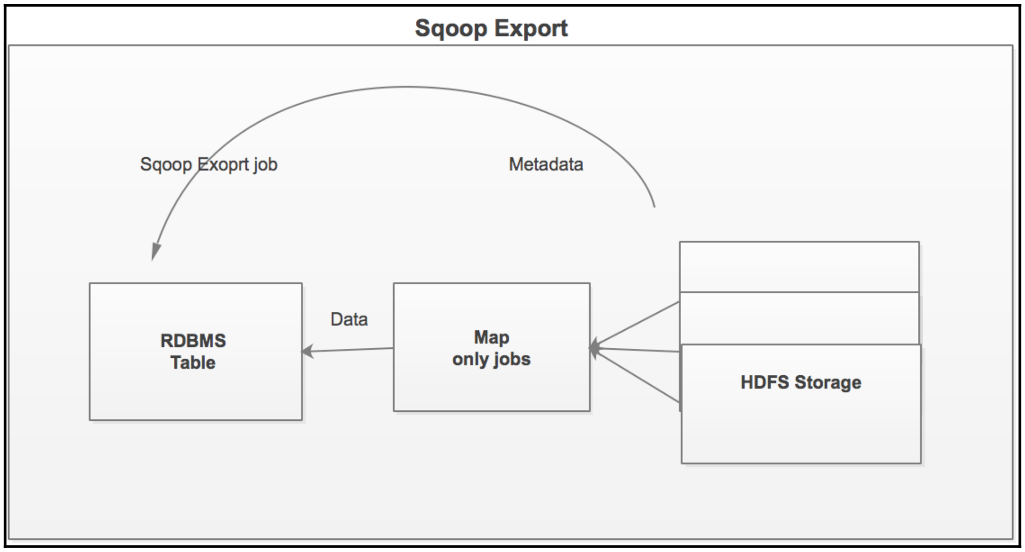
The following is a sample command to import data into an HBase table:

```
$sqoop import -connect jdbc:mysql://localhost/dbname -table <table_name>  -
-username <username> --password >password>  --hive-import -m 4
--hbase-create-table --hbase-table <table_name>--column-family <col family
name>
```

Sqoop imports data into the HBase table column family. The data is converted and inserted as a UTF-8 bytes format.

Sqoop export

The following diagram shows the **Sqoop Export** function to export data from a Hadoop cluster:



Data processed in a data lake-like use case may be needed for additional business functions. Sqoop can be used to export that data back to RDBMS from HDFS or from a Hive table. In the case of exporting data back to an RDBMS table, the target table must exist in a MySQL database. The rows in HDFS files or records from a Hive table are given as input to the `sqoop` command and are called rows in a target table. Those records are read and parsed into a set of records and delimited with a user-specified delimiter.

The following are the commands to export data from HDFS to a MySQL table. Let's create a table in MySQL to store data exported from HDFS:

```
$ mysql>  use sales;$  mysql>   create table customer_export (
cust_num int not null,      cust_fname  varchar(30),      cust_lname
varchar (30),      cust_address  varchar (30),      cust_city varchar (20),
cust_state  varchar (3),      cust_zip  varchar (6),      primary key
(cust_num));

$  sqoop export --connect jdbc:mysql://127.0.0.1:3306/sales --driver
com.mysql.jdbc.Driver --username root --password hadoop --table
customer_exported  --export-dir /user/data/customer
```

The `--table` parameter specifies the table which will be populated. Sqoop splits the data and uses individual map tasks to push the splits into the database. Each map task does the actual data transfer. The `--export-dir <directory h>` is the directory from which data will be exported:

```
$ mysql>  use sales;$ mysql>  select * from customer_exported;
```

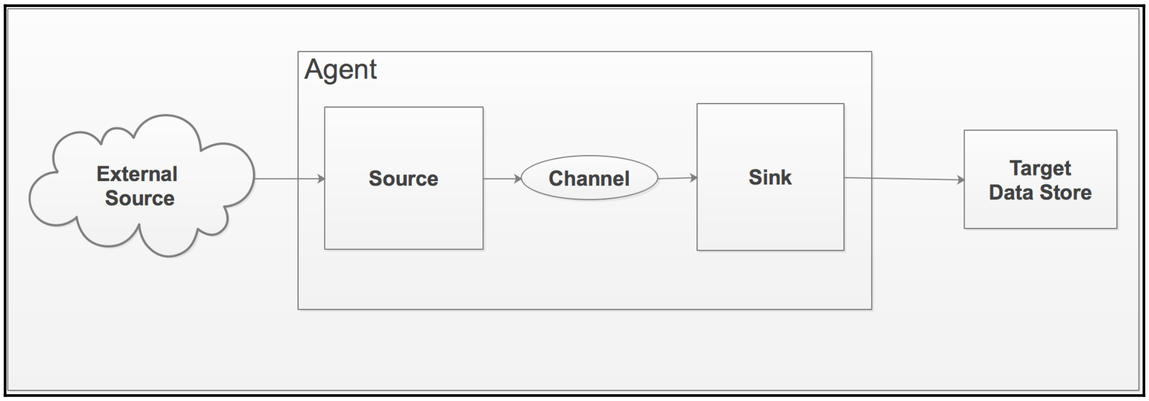
Custnum	Cust Fname	Cust Lname	Cust Address	City	State	Zip
1	James	Butt	6649 N Blue Gum St	New Orleans	LA	70116
2	Art	Venere 8	W Cerritos Ave #54	Bridgeport	NJ	8014
3	Lenna	Paprocki	639 Main St	Anchorage	AK	99501
4	Donette	Foller	34 Center St	Hamilton	OH	45011
5	Simona	Morasca	3 Mcauley Dr	Ashland	OH	44805
6	Mitsue	Tollner	7 Eads St	Chicago	IL	60632
7	Leota	Dilliard	7 W Jackson Blvd	San Jose	CA	95111
8	Sage	Wieser	5 Boston Ave #88	Sioux Falls	SD	57105
9	Kris	Marrier	228 Runamuck Pl #2808	Baltimore	MD	21224
10	Minna	Amigon	2371 Jerrold Ave	Kulpsville	PA	19443

Flume

Flume is a reliable, available and distributed service to efficiently collect, aggregate, and transport large amounts of log data. It has a flexible and simple architecture that is based on streaming data flows. The current version of Apache Flume is 1.7.0, which was released in October 2016.

Apache Flume architecture

The following diagram depicts the architecture of Apache Flume:



Let's take a closer look at the components of the Apache Flume architecture:

- **Event**: An event is a byte payload with optional string headers. It represents the unit of data that Flume can carry from its source to destination.
- **Flow**: The transport of events from source to destination is considered a data flow, or just flow.
- **Agent**: It is an independent process that hosts the components of Flume, such as sources, channels, and sinks. It thus has the ability to receive, store, and forward events to its next-hop destination.
- **Source**: The source is an interface implementation. It has the ability to consume events that are delivered to it with the help of a specific mechanism.
- **Channel**: It is a store where events are delivered to the channel through sources that operate within the agent. An event placed in a channel remains there until a sink takes it out for further transport. Channels play an important role in ensuring this.
- **Sink**: It is an interface implementation, just like the source. It can remove events from a channel and transport them to the next agent in the flow, or to its final destination.
- **Interceptors**: They help to change an event in transit. An event can be removed or modified on the basis of the chosen criteria. Interceptors are the classes, which implement the `org.apache.flume.interceptor.Interceptor` interface.

Data flow using Flume

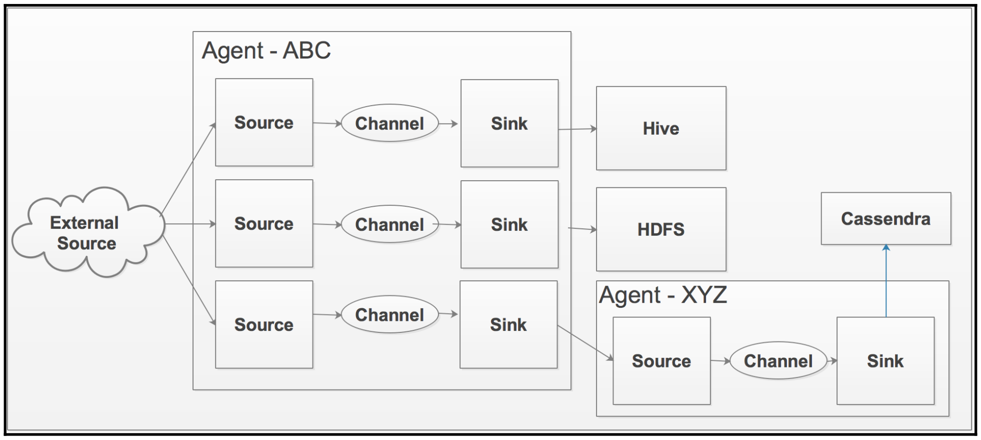
The entire Flume agent runs in a JVM process, which includes all the components (source, channel, and sink). The Flume source receives events from the external sources, like a web server, external files, and so on. The source pushes events to the channel, which stores it until picked up by the sink. The channel stores the payload (message stream) in either the local filesystem or in a memory, depending on the type of the source. For example, if the source is a file, the payload is stored locally. The sink picks up the payload from the channel and pushes it to external data stores. The source and sink within the agent run asynchronously. Sometimes, it may be possible for the sink to push the payload to yet another Flume agent. We will talk about that scenario in the next section.

Flume complex data flow architecture

In the following architecture, there are three sources (servers). In order to pull data from the log files stored on these servers, we have to install Flume software on each of these servers. After installation, the filenames need to be added to the `flume.conf` file. Flume collects all the data from files and pushes it to the corresponding sink through channels. There are multiple sinks in the above architecture; Hive HDFS, and another sink, which is connected to another installation of the Flume agent installed on another server. It pushes data from sink to source and writes data to the Cassendra data store.

Please note that this is not a good architecture, but I have mentioned it to explain how a Flume sink and Flume sources can be connected.

The following diagram shows complex data flow involving multiple agents:



Flume setup

Flume agent configuration is stored in a local text file. Please refer to the sample Flume agent configuration file in the code repository of this book. Flume 1.7.0 supports various sources and sinks. Widely used Flume sources (a summary) are as follows:

Source	**Description**
Avro source	Listens on Avro port and receives events from external Avro client streams
Exec source	Runs a given Unix command and expects that process to continuously produce data on standard out
Spooling directory source	Ingests data from files on disk
Taildir source	Tails files in near real-time after new lines are detected in the files
Kafka source	Reads messages from Kafka topics
Syslog source	Reads syslog data (supports syslog-TCP and syslog-UDP)
HTTP source	Accepts Flume events by HTTP `POST` and `GET`

The widely used Flume sinks can be summarized as follows:

Sink	**Description**
Avro sink	Events are turned into Avro events and sent to the configured hostname/port pair
HDFS sink	Writes events into the HDFS
Hive sink	Writes text or JSON data into a Hive table
HBase sink	Writes data to HBase
Morphline Solr sink	Loads it in near real-time into Apache Solr servers
Elasticsearch sink	Writes data to an Elasticsearch cluster
Kafka sink	Writes data to a Kafka topic

The widely used Flume channels (a summary) are as follows:

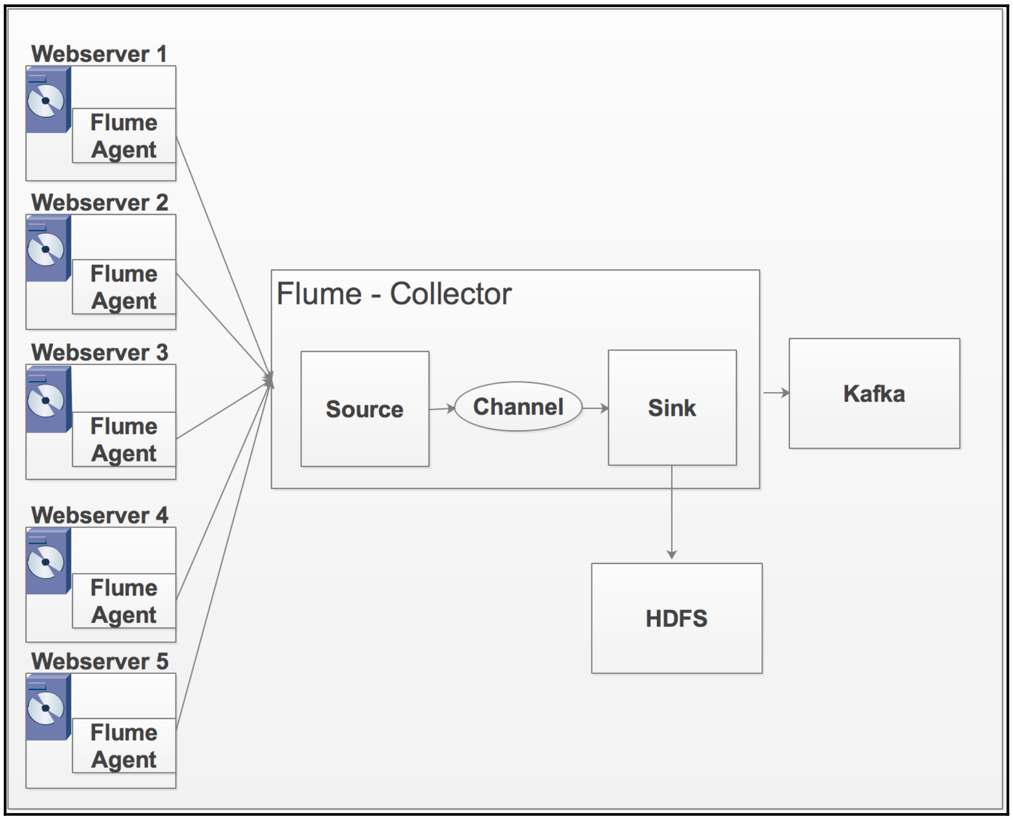
Channel	**Description**
JDBC channel	Events are stored in storage supported by database
Kafka channel	Events are stored in a Kafka cluster
File channel	Events are stored in files
Spillable memory channel	Events are stored in memory; if memory gets full, then stored on disk

The widely used Flume interceptors can be summarized as follows:

Interceptor	**Description**
Timestamp interceptor	Adds the processing time of an event into event headers
Host interceptor	Adds hostname of agent
Search and replace interceptor	Supports Java regular expressions
Regex filtering interceptor	Filters the events against RegEx
Regex extractor interceptor	Extracts and appends the match RegEx groups as headers on the event

Log aggregation use case

In day-to-day business scenarios, we always find the need to get log files and make sense out of them. For example, we always find the need to get logs from different applications and servers and merge them together to find trends and patterns. Let me extend this example further. Let's assume that we have five web servers deployed on five different servers. We want to get all five web server logs and merge/aggregate them together to analyze them further by storing one copy on HDFS and another copy to be shipped on to a Kafka topic for real-time analytics. The question is how we design Flume-based log aggregation architecture. The following is the Flume architecture for our web server log aggregation scenario:



Let us walk through the architecture in detail: There are a total of five web servers. Each web server generates a log file and stores it locally. The Flume agent is installed on each web server. The Flume agent is nothing but a (JVM) process that hosts the components through which events flow from an external source to the next destination (hop). Each Flume agent accesses log files based on local configuration of `flume.conf`. Each Flume agent reads the log files and pushes data to the Flume collector. Each line of the log file is treated as one message (a payload). The Flume collector gets messages from all web servers, fitters and aggregates all messages, and pushes these messages to the data store. The following is the sample `flume.conf` of the Flume agent and the collectors agent `flume.conf`:

```
## Sample Flume Agent Configuration
## This conf file should deploy on each webserver
##
a1.sources = apache
a1.sources.apache.type = exec
```

```
a1.sources.apache.command = gtail -F /var/log/httpd/access_log
a1.sources.apache.batchSize = 1
a1.sources.apache.channels = memoryChannel

a1.channels = memoryChannel
a1.channels.memoryChannel.type = memory
a1.channels.memoryChannel.capacity = 100

## Collector Details

a1.sinks = AvroSink
a1.sinks.AvroSink.type = avro
a1.sinks.AvroSink.channel = memoryChannel
a1.sinks.AvroSink.hostname = 10.0.0.10
a1.sinks.AvroSink.port = 6565
```

The collector `flume.conf` file is as follows:

```
## Collector get data from all agents

collector.sources = AvroIn
collector.sources.AvroIn.type = avro
collector.sources.AvroIn.bind = 0.0.0.0
collector.sources.AvroIn.port = 4545
collector.sources.AvroIn.channels = mc1 mc2


collector.channels = mc1 mc2
collector.channels.mc1.type = memory
collector.channels.mc1.capacity = 100

collector.channels.mc2.type = memory
collector.channels.mc2.capacity = 100


## Write copy to Local Filesystem (Debugging)
# http://flume.apache.org/FlumeUserGuide.html#file-roll-sink
collector.sinks.LocalOut.type = file_roll
collector.sinks.LocalOut.sink.directory = /var/log/flume
collector.sinks.LocalOut.sink.rollInterval = 0
collector.sinks.LocalOut.channel = mc1

## Write to HDFS
collector.sinks.HadoopOut.type = hdfs
collector.sinks.HadoopOut.channel = mc2
collector.sinks.HadoopOut.hdfs.path = /flume/events/%{log_type}/%{host}/%y-
%m-%d
```

```
collector.sinks.HadoopOut.hdfs.fileType = DataStream
collector.sinks.HadoopOut.hdfs.writeFormat = Text
collector.sinks.HadoopOut.hdfs.rollSize = 0
collector.sinks.HadoopOut.hdfs.rollCount = 10000
collector.sinks.HadoopOut.hdfs.rollInterval = 600
```

Apache NiFi

What is Apache NiFi? In any organization, we know that there is a variety of systems. Some systems generate the data and other systems consume that data. Apache NiFi is built to automate that data flow from one system to another. Apache NiFi is a data flow management system that comes with a web UI that helps to build data flows in real time. It supports flow-based programming. The graph programming includes a series of nodes and edges through which data moves. In NiFi, these nodes are translated into processors, and the edges into connectors. The data is stored in a packet of information called a **FlowFile**. This FlowFile includes content, attributes, and edges. As a user, you connect processors together using connectors to define how the data should be handled.

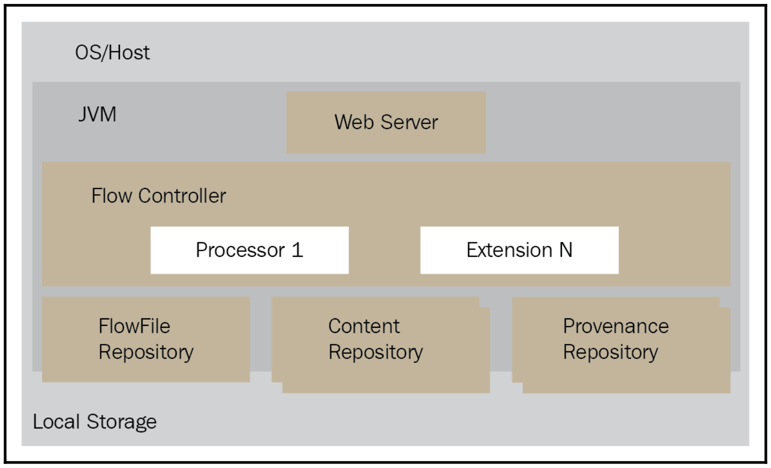
Main concepts of Apache NiFi

The following table describes the main components of Apache NiFi:

Component name	**Description**
FlowFile	Data packet running through the system
FlowFile processor	Performs the actual work of data routing, transformation, and data movement
Connetion	Actual data linkage between processors
Flow controller	Facilitates the exchange of FlowFiles between the processors
Process group	Specific group of data inputs and data output processors

Apache NiFi architecture

The following diagram shows the components of the Apache NiFi architecture (source: `https://nifi.apache.org/docs.html`):



The components are as follows:

- **Web server**: This hosts NiFi's HTTP-based UI
- **File controller**: This provides threads and manages the schedule for the extensions to run on
- **Extensions**: The extensions operate and execute within the JVM
- **FileFlow repository**: This keeps track of the state of what it knows about a given FlowFile that is presently active in the flow
- **Content repository**: This is where the actual content bytes of a given FlowFile live
- **Provenance repository**: This is where all provenance event data is stored

Key features

The following are the key features of Apache NiFi:

- **Guaranteed delivery**: In the event of increased volume of data, power failures, and network and system failures in NiFi, it becomes necessary to have a robust guaranteed delivery of the data. NiFi ensures, within the dataflow system itself, the transactional communication between NiFi and the data where it is coming to the points to which it is delivered to.
- **Data buffering with back pressure and pressure release**: In any dataflow, it may be possible that there are some issues with the systems involved; some might be down or some might be slow. In that case, data buffering becomes very essential to coping with the data coming into or going out of the dataflow.

  NiFi supports the buffering of all queues with back pressure when it reaches specific limits and age of the data. NiFi does it with a maximum possible throughput rate, while maintaining a good response time.

- **Prioritized queuing**: In general, the data queues maintain natural order or insertion order. But, many times, when the rate of data insertion is faster than the bandwidth, you have to prioritize your data retrieval from the queue. The default is the oldest data first. But NiFi supports prioritization of queues to pull data out based on size, time, and so on that is, largest first or newest first.
- **Flow-specific quality of service (QoS)**: There are some situations where we have to process the data in a specific time period, for example, within a second and so on, otherwise the data loses its value. The fine-grained flow of specific configuration of these concerns is enabled by Apache NiFi.
- **Data provenance**: NiFi automatically records, indexes, and makes available provenance data as objects flow through the system—even across fan-in, fan-out, transformations, and more. This information becomes extremely critical in supporting compliance, troubleshooting, optimization, and other scenarios.
- **Visual command and control**: Apache NiFi allows users to have interactive management of dataflow. It provides immediate feedback to each and every change to the dataflow. Hence, users understand and immediately correct any problems, mistakes, or issues in their dataflows. Based on analytical results of the dataflows, users can make changes to their dataflow, prioritize of queues, add more data flows, and so on.
- **Flow templates**: Data flows can be developed, designed, and shared. Templates allow subject matter experts to build and publish their flow designs and for others to benefit and collaborate on them.

- **Extension**: NiFi allows us to extend its key components.
- **Points of extension:** Processors, controller services, reporting tasks, prioritizers, and customer UIs.
- **Multi-role security**: Multi-grained, multi-role security can be applied to each component, which allows the admin user to have a fine-grained level of access control.
- **Clustering**: NiFi is designed to scale-out through the use of clustering many nodes together. That way, it can handle more data by adding more nodes to the cluster.

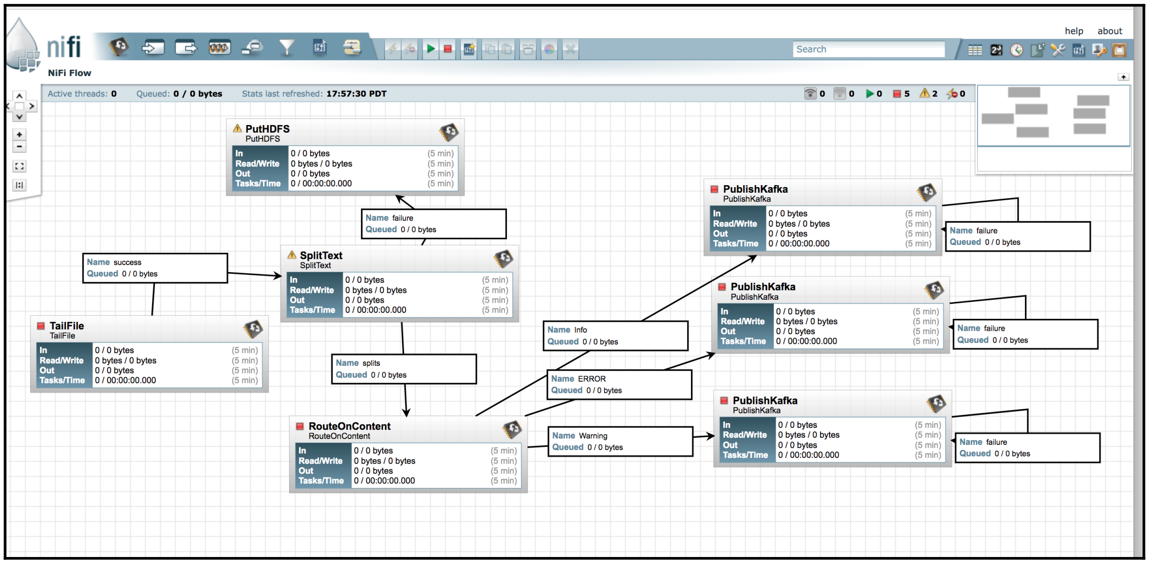
For getting started with Apache NiFi, please use this link: `https://nifi.apache.org/docs/nifi-docs/html/getting-started.html`.

Let's imagine a scenario. I have a running log file. It is updated on the fly. I want to capture and monitor each line in that file, based on its contents. I want to send it to my Kafka brokers. I also want to deliver all my error records to HDFS for archival and further analysis. Different line types will be sent to different Kafka brokers. For example, error, info, and success types will be sent to three different Kafka topics, namely error, info, and success. I have developed the following NiFi workflow for that. The following table gives the detailed explanation of each processor:

Processor	**Purpose**	**Property**	**Value**
TailFile	To tail log files	File to tail	`/var/log/apache.log`
SplitText	To split the log entries to the lines	Line split count	`1`
RouteOnContent	To make a routing decision		
PutHDFS	To send errors to HDFS		HDFS details
PublishKafka	To deliver data to Kafka topic	Broker and topic name	Hostname: port, topic pane

Real-time log capture dataflow

The following example workflow shows how log file data can be pushed to HDFS and then to moved to Kafka brokers:



Kafka Connect

Kafka Connect is a part of Apache Kafka. It is a framework to ingest data from one to another system using connectors. There are two types of connectors: source connectors and sink connectors. The sink connectors import data from source systems and write to Kafka topics. The sink connectors read data from the Kafka topic and export it to target systems. Kafka Connect provides various source and sink connectors out of the box.

Kafka Connect – a brief history

Kafka Connect was mainly introduced in November 2015 in Kafka 0.9.x. In addition to the various features of Kafka 0.9.x, Connect APIs was a brand new feature. Then, in May 2016, the new version Kafka 0.10.0 was released. In that version, Kafka Streams API was a new and exciting feature. But, in March 2017, it was Kafka Version 0.10.2 where Kafka Connect got its real momentum. As a part of Kafka 0.10.2, improved simplified Connect APIs and single message transform APIs were released.

Why Kafka Connect?

Kafka Connect helps to simplify getting data in and out of Kafka. It provides a lot of connectors to do that out of the box. In my opinion, that's the best incentive to a developer like me, because I do not have to develop a separate code to develop my own connector to import and export data; I can always reuse the out-of-the-box connector for that. Also, if I want, I can always develop my own unique connector using Kafka Connect APIs. Also, all the connectors are configuration-based. The common sources and targets are databases, search engines, NoSQL data stores, and applications like SAP, GoldenGate, Salesforce, HDFS, Elasticsearch, and so on. For a detailed listing of all available sources and connectors, please refer to `https://www.confluent.io/product/connectors/`.

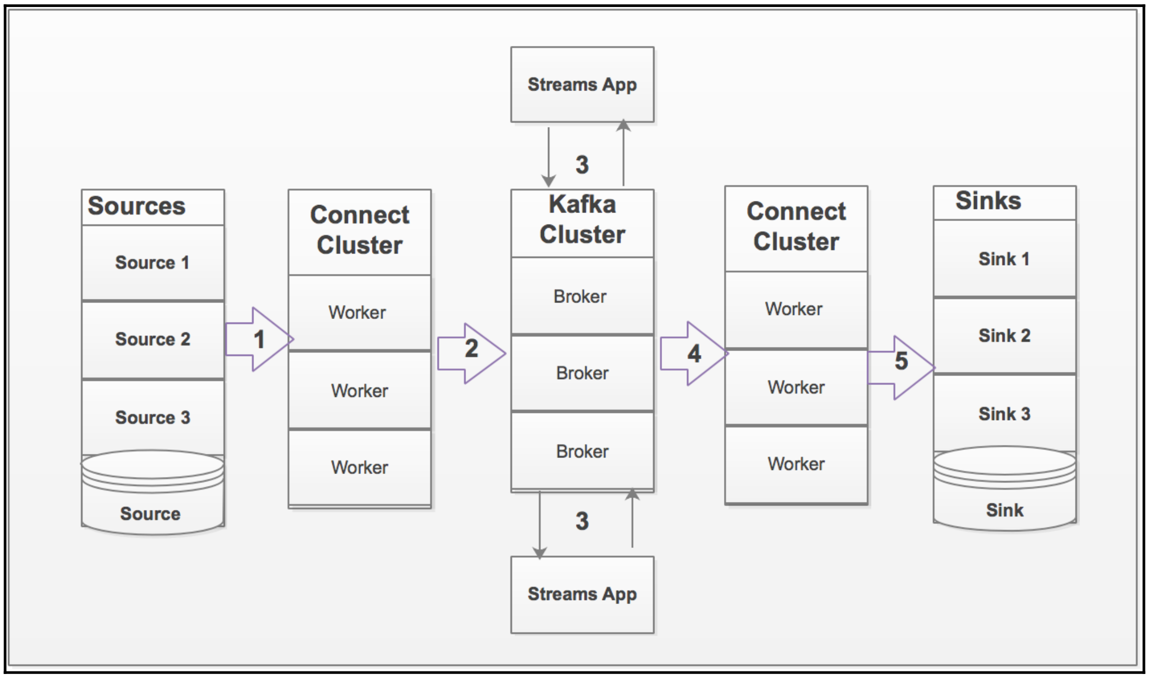
Kafka Connect features

The following are some features of Kafka connect:

- **Scalable**: This is a framework for scalable and reliable streaming data between Apache Kafka and other systems
- **Simple**: This makes it simple to define connectors that move large collections of data into and out of Kafka
- **Offset management**: The framework does most of the hard work of properly recording the offsets of the connectors
- **Easy operation**: This has a service that has a RESTful API for managing and deploying connectors
- **Distributed**: The framework can be clustered and will automatically distribute the connectors across the cluster, ensuring that the connector is always running
- **Out-of-the-box connectors**: For a detailed listing of all available sources and connectors, please refer to `https://www.confluent.io/product/connectors/`

Kafka Connect architecture

The following diagram represents the Kafka Connect architecture:



The Kafka cluster is made of Kafka brokers: three brokers, as shown in the diagram. Sources can be of any type, for example, databases, NoSQL, Twitter, and so on. In between the source and Kafka cluster, there is a Kafka Connect cluster, which is made up of workers. The working of Kafka Connect comprises the following steps:

1. Workers, based on configuration, pull data from sources
2. After getting data, the connector pushes data to the Kafka cluster
3. If data needs to be transformed, filtered, joined, or aggregated using stream applications such as Spark, Storm, and so on, stream APIs will change data in and out of Kafka
4. Based on the configuration, the connector will pull data out of Kafka and write it to the sink

Some Kafka Connect concepts are as follows:

- Source connectors get data from common data sources.
- Sink connectors publish data to common data sources.
- Kafka Connect makes it easy to quickly get data reliably into Kafka.
- It is a part of the ETL pipeline.
- Scaling from a small pipeline to a company-wide pipeline is very easy.
- The code is reusable.
- The Kafka Connect cluster has multiple loaded connectors. Each connector is a reusable piece of code, `(Java JARs)`. There are a lot of open source connectors available, which can be leveraged.
- Each connector task is a combination of connector class and configuration. A task is linked to a connector configuration. A job creation may create multiple tasks. So, if you have one connector and one configuration, then two or more tasks can be created.
- Kafka Connect workers and servers execute tasks. A worker is a single java process. A worker can be standalone or distributed.

Kafka Connect workers modes

There are two modes of Kafka Connect workers:

- Standalone mode
- Distributed mode

Standalone mode

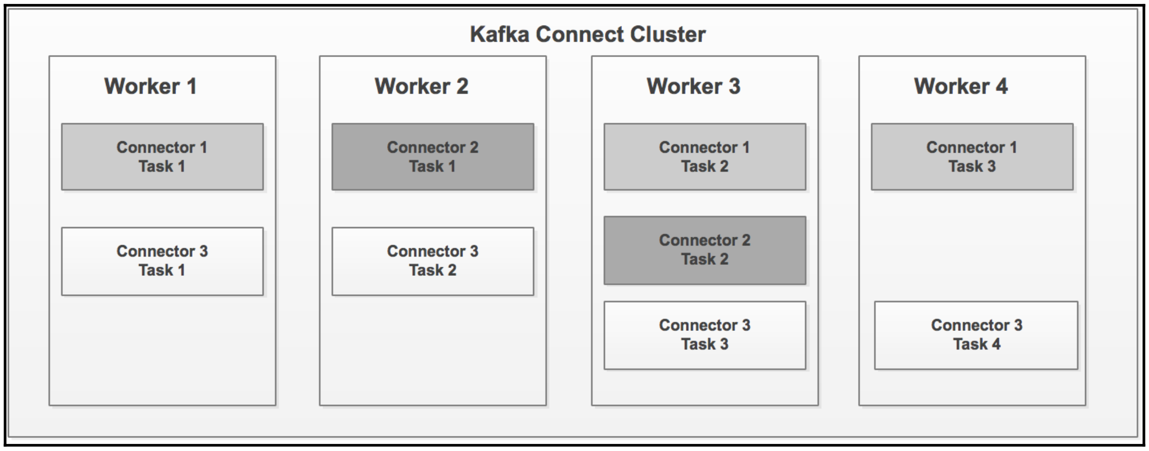
Standalone mode is a single process (worker) that runs all the connectors and tasks. The configuration is bundled in a single process. It is not fault tolerant or scalable, and it is very difficult to monitor. Since it is easy to set up, it is mainly used during development and testing.

Distributed mode

With distributed mode, multiple workers (processes) run your connectors and tasks. The configuration is submitted using the REST API. It is scalable and fault tolerant. It automatically rebalances all the tasks on the cluster if any worker dies. Since it is scalable and fault tolerant, it is mainly used in a production environment.

Kafka Connect cluster distributed architecture

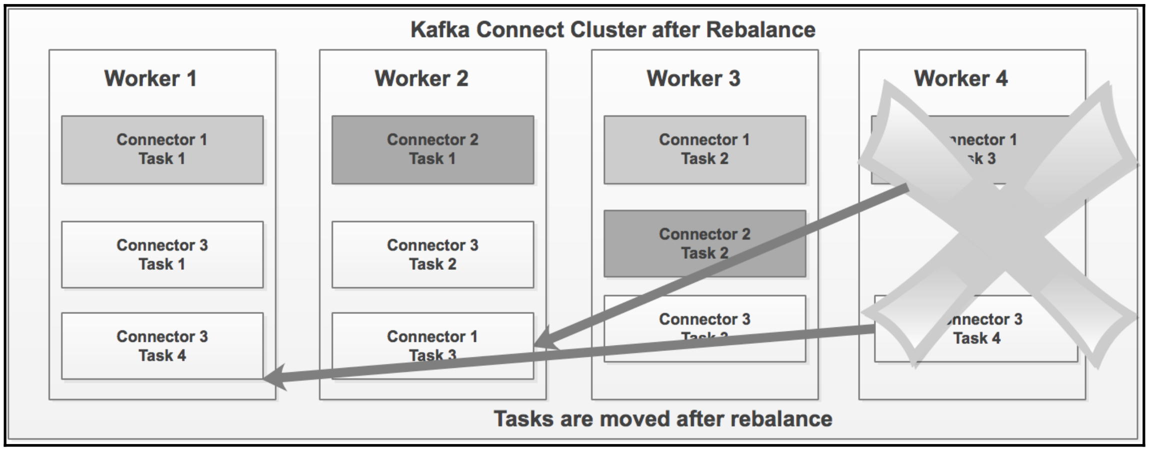
The following is the representation of the Kafka Connect cluster distributed architecture details:



In the preceding diagram, we can see the following details:

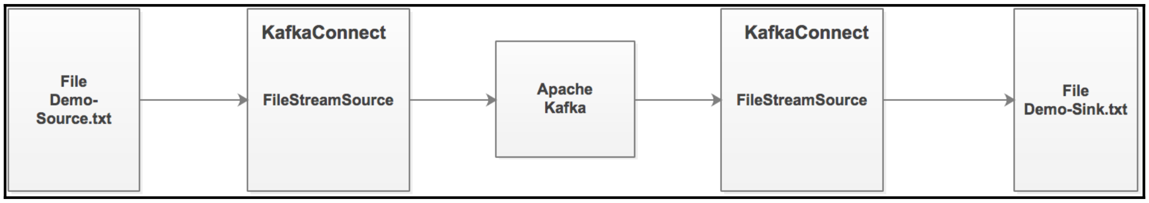
- We have source **Connector 1**, with three tasks: **Task 1**, **Task 2**, and **Task 3**. These three tasks are spread out among four workers: **Worker 1**, **Worker 3**, and **Worker 4**.
- We also have source **Connector 2**, with two tasks: **Task 1** and **Task 2**. These two tasks are spread out between two workers: **Worker 2** and **Worker 3**.
- We also have sink **Connector 3** with four tasks: **Task 1**, **Task 2**, **Task 3**, and **Task 4**. These four tasks are spread out among four workers: **Worker 1**, **Worker 2**, **Worker 3**, and **Worker 4**.
- Now, something happens and **Worker 4** dies and we lose that worker completely.
- As a part of fault tolerance, the rebalance activity kicks off. **Connector 1** and **Task 3** move from **Worker 4** to **Worker 2**. Similarly, **Connector 3** and **Task 4** move from **Connector 4** to **Connector 1**.

The following diagram represents the Kafka Connect cluster after rebalance:



Example 1

Streamed data from source file `Demo-Source.txt` is moved to destination file `Demo-Sink.txt` in standalone mode, as shown in the following diagram:



In order to stream data from source file `Demo-Source.txt` to destination file `Demo-Sink.txt` in standalone mode, we need to perform the following steps:

1. Start Kafka:

```
$ /bin/kafka-server-start.sh config/server.properties
```

2. Create topic:

```
$ .bin/kafka-topics --create --topic demo-1-standalone --partitions
3 --replication-factor 1 --zookeeper 127.0.0.1:2181
```

3. Configure the `source-file-stream-standalone.properties` file:

   ```
   name=source-file-stream-standalone
   connector.class=org.apache.kafka.connect.file.FileStreamSourceConne
   ctor
   tasks.max=1
   file=demo-source.txt
   topic=file-source-topic
   ```

4. Configure `file-stream-standalone.properties` file:

   ```
   name=sinkfile-stream-standalone
   connector.class=org.apache.kafka.file.FileStreamSourceConnector
   tasks.max=1
   file=demo-sink.txt
   topics=file-source-topic
   ```

5. Configure `file-worker.properties` file:

   ```
   bootstrap.servers=127.0.0.1:9092
   key.converter=org.apache.kafka.connect.json.JsonConverter
   key.converter.schemas.enable=false
   value.converter=org.apache.kafka.connect.json.JsonConverter
   value.converter.schemas.enable=false
   # we always leave the internal key to JsonConverter
   internal.key.converter=org.apache.kafka.connect.json.JsonConverter
   internal.key.converter.schemas.enable=false
   internal.value.converter=org.apache.kafka.connect.json.JsonConverte
   r
   internal.value.converter.schemas.enable=false
   rest.port=8086
   rest.host.name=127.0.0.1
   # this config is only for standalone workers
   offset.storage.file.filename=standalone.offsets
   offset.flush.interval.ms=10000
   ```

6. Start Kafka Connect. Open another terminal and run the following command:

   ```
   $ .bin/connect-standalone config/file-worker.properties
   config/source-file-stream-standalone.properties config/ sink-file-
   stream-standalone.properties
   ```

7. Add data to the `demo-source.txt` file. Open another terminal and run the following command:

   ```
   $ touch demo-source.txt

   $ echo "Test Line 1 " >>  demo-source.txt

   $ echo "Test Line 2 " >>  demo-source.txt

   $ echo "Test Line 2 " >>  demo-source.txt
   ```

8. Read the `demo-sink.txt` file:

   ```
   $ cat demo-sink.file
   ```

Example 2

Streamed data from source file `Demo-Source.txt` is moved to destination file `Demo-Sink.txt` in distributed mode. If you want to run the previous example using distributed mode, you have to add the following parameter to `source-file-stream` and `sink-file-stream` in *steps 3* and *4*:

```
key.converter=org.apache.kafka.connect.json.JsonConverter
key.converter.schemas.enable=true
value.converter=org.apache.kafka.connect.json.JsonConverter
value.converter.schemas.enable=true
```

Summary

In this chapter, we have learned all the popular data ingestion tools used in production environments. Sqoop is mainly used to import and export data in and out of RDBMS data stores. Apache Flume is used in real-time systems to import data, mainly from files sources. It supports a wide variety of sources and sinks. Apache NiFi is a fairly new tool and getting very popular these days. It also supports GUI-based ETL development. Hortonworks has started supporting this tool since their HDP 2.4 release. Apache Kafka Connect is another popular tool in the market. It is also a part of the Confluent Data Platform. Kafka Connect can ingest entire databases or collect metrics from all your application servers into Kafka topics, making the data available for stream processing with low latency.

Since we so far know how to build Hadoop clusters and how to ingest data in them, we will learn data modeling techniques in the next chapter.

5
Data Modeling in Hadoop

So far, we've learned how to create a Hadoop cluster and how to load data into it. In the previous chapter, we learned about various data ingestion tools and techniques. As we know by now, there are various open source tools available in the market, but there is a single silver bullet tool that can take on all our use cases. Each data ingestion tool has certain unique features; they can prove to be very productive and useful in typical use cases. For example, Sqoop is more useful when used to import and export Hadoop data from and to an RDBMS.

In this chapter, we will learn how to store and model data in Hadoop clusters. Like data ingestion tools, there are various data stores available. These data stores support different data models—that is, columnar data storage, key value pairs, and so on; and they support various file formats, such as ORC, Parquet, and AVRO, and so on. There are very popular data stores, widely used in production these days, for example, Hive, HBase, Cassandra, and so on. We will learn more about the following two data stores and data modeling techniques:

- Apache Hive
- Apache HBase

First, we will start with basic concepts and then we will learn how we can apply modern data modeling techniques for faster data access. In a nutshell, we will cover the following topics in this chapter:

- Apache Hive and RDBMS
- Supported datatypes
- Hive architecture and how it works

Apache Hive

Hive is a data processing tool in Hadoop. As we have learned in the previous chapter, data ingestion tools load data and generate HDFS files in Hadoop; we need to query that data based on our business requirements. We can access the data using MapReduce programming. But data access with MapReduce is extremely slow. To access a few lines of HDFS files, we have to write separate mapper, reducer, and driver code. So, in order to avoid this complexity, Apache introduced Hive. Hive supports an SQL-like interface that helps access the same lines of HDFS files using SQL commands. Hive was initially developed by Facebook but was later taken over by Apache.

Apache Hive and RDBMS

I mentioned that Hive provides an SQL-like interface. Bearing this in mind, the question that arises is: *is Hive the same as RDBMS on Hadoop?* The answer is *no*. Hive is not a database. Hive does not store any data. Hive stores table information as a part of metadata, which is called schema, and points to files on HDFS. Hive accesses data stored on HDFS files using an SQL-like interface called **HiveQL** (**HQL**). Hive supports SQL commands to access and modify data in HDFS. Hive is not a tool for OLTP. It does not provide any row-level insert, update, or delete. The current version of Hive (version 0.14), does support insert, update, and delete with full ACID properties, but that feature is not efficient. Also, this feature does not support all file formats. For example, the update supports only ORC file format. Basically, Hive is designed for batch processing and does not support transaction processing like RDBMS does. Hence, Hive is better suited for data warehouse applications for providing data summarization, query, and analysis. Internally, Hive SQL queries are converted into MapReduce by its compiler. Users need not worry about writing any complex mapper and reducer code. Hive supports query structured data only. It is very complex to access unstructured data using Hive SQL. You may have to write your own custom functions for that. Hive supports various file formats such as text files, sequence files, ORC, and Parquet, which provide significant data compression.

Supported datatypes

The following datatypes are supported by Hive version 0.14:

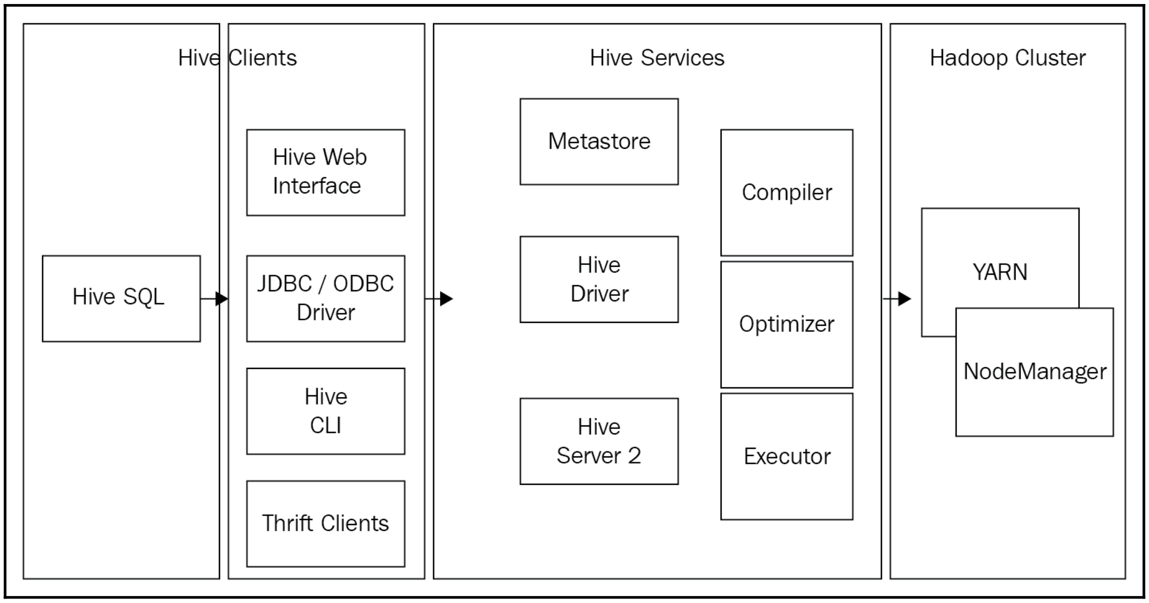
Datatype group	Datatype	Format
String	STRING	column_name STRING
	VARCHAR	column_name VARCHAR(max_length)
	CHAR	column_name CHAR(length)
Numeric	TINYINT	column_name TINYINT
	SMALLINT	column_name SMALLINT
	INT	column_name INT
	BIGINT	column_name BIGINT
	FLOAT	column_name FLOAT
	DOUBLE	column_name DOUBLE
	DECIMAL	column_name DECIMAL[(precision[,scale])]
Date/time type	TIMESTAMP	column_name TIMESTAMP
	DATE	column_name DATE
	INTERVAL	column_name INTERVAL year to month
Miscellaneous type	BOOLEAN	column_name BOOLEAN
	BINARY	column_name BINARY
Complex type	ARRAY	column_name ARRAY < type >
	MAPS	column_name MAP < primitive_type, type >
	STRUCT	column_name STRUCT < name : type [COMMENT 'comment_string'] >
	UNION	column_name UNIONTYPE <int, double, array, string>

How Hive works

Hive databases are comprised of tables which are made up of partitions. Data can be accessed via a simple query language and Hive supports overwriting or appending of data. Within a particular database, data in tables is serialized and each table has a corresponding HDFS directory. Each table can be sub-divided into partitions that determine how data is distributed within subdirectories of the table directory. Data within partitions can be further broken down into buckets.

Hive architecture

The following is a representation of Hive architecture:



The preceding diagram shows that Hive architecture is divided into three parts—that is, clients, services, and metastore. The Hive SQL is executed as follows:

- **Hive SQL query**: A Hive query can be submitted to the Hive server using one of these ways: WebUI, JDBC/ODBC application, and Hive CLI. For a thrift-based application, it will provide a thrift client for communication.

- **Query execution**: Once the Hive server receives the query, it is compiled, converted into an optimized query plan for better performance, and converted into a MapReduce job. During this process, the Hive Server interacts with the metastore for query metadata.
- **Job execution**: The MapReduce job is executed on the Hadoop cluster.

Hive data model management

Hive handles data in the following four ways:

- Hive tables
- Hive table partition
- Hive partition bucketing
- Hive views

We will see each one of them in detail in the following sections.

Hive tables

A Hive table is very similar to any RDBMS table. The table is divided into rows and columns. Each column (field) is defined with a proper name and datatype. We have already seen all the available datatypes in Hive in the *Supported datatypes* section. A Hive table is divided into two types:

- Managed tables
- External tables

We will learn about both of these types in the following sections.

Managed tables

The following is a sample command to define a Hive managed table:

```
Create Table < managed_table_name>
   Column1 <data type>,
   Column2 <data type>,
   Column3 <data type>
Row format delimited Fields Terminated by "t";
```

When the preceding query is executed, Hive creates the table and the metadata is updated in the metastore accordingly. But the table is empty. So, data can be loaded into this table by executing the following command:

```
Load data inpath <hdfs_folder_name> into table <managed_table_name>;
```

After executing the previous command, the data is moved from `<hdfs_folder_name>` to the Hive table's default location `/user/hive/warehouse/<managed_table_name`. This default folder, `/user/hive/warehouse`, is defined in `hive-site.xml` and can be changed to any folder. Now, if we decide to drop the table, we can do so by issuing the following command:

```
Drop table <managed_table_name>;
```

The `/user/hive/warehouse/<managed_table_name` folder will be dropped and the metadata stored in the metastore will be deleted.

External tables

The following is a sample command to define a Hive external table:

```
Create Table < external_table_name>
   Column1 <data type>,
   Column2 <data type>,
   Column3 <data type>
Row format delimited Fields Terminated by "t"
Location <hdfs_folder_name>;
```

When the preceding query is executed, Hive creates the table and the metadata is updated in the metastore accordingly. But, again, the table is empty. So, data can be loaded into this table by executing the following command:

```
Load data inpath <hdfs_folder_name> into table <external_table_name>;
```

This command will not move any file to any folder but, instead, creates a pointer to the folder location, and it is updated in the metadata in the metastore. The file remains at the same location (`<hdfs_folder_name>`) of the query. Now, if we decide to drop the table, we can do so by issuing the following command:

```
Drop table <managed_table_name>;
```

The folder /user/hive/warehouse/<managed_table_name will not be dropped and only the metadata stored in the metastore will be deleted. The file remains in the same location—<hdfs_folder_name>.

Hive table partition

Partitioning a table means dividing a table into different parts based on a value of a partition key. A partition key can be any column, for example, date, department, country, and so on. As data is stored in parts, the query response time becomes faster. Instead of scanning the whole table, partition creates subfolders within the main table folders. Hive will scan only a specific part or parts of the table based on the query's WHERE clause. Hive table partition is similar to any RDBMS table partition. The purpose is also the same. As we keep inserting data into a table, the table becomes bigger in data size. Let's say we create an ORDERS table as follows:

```
hive> create database if not exists ORDERS;
OK
Time taken: 0.036 seconds

hive> use orders;
OK
Time taken: 0.262 seconds

hive> CREATE TABLE if not exists ORDEERS_DATA
    > (Ord_id INT,
    > Ord_month INT,
    > Ord_customer_id INT,
    > Ord_city  STRING,
    > Ord_zip   STRING,
    > ORD_amt   FLOAT
    > )
    > ROW FORMAT DELIMITED
    > FIELDS TERMINATED BY  ','
    > ;
OK
Time taken: 0.426 seconds
hive>
```

We will load the following sample file ORDERS_DATA table as follows:

```
101,1,100,'Los Angeles','90001',1200
102,2,200,'Los Angeles','90002',1800
103,3,300,'Austin','78701',6500
104,4,400,'Phoenix','85001',7800
```

```
105,5,500,'Beverly Hills','90209',7822
106,6,600,'Gaylord','49734',8900
107,7,700,'Los Angeles','90001',7002
108,8,800,'Los Angeles','90002',8088
109,9,900,'Reno','89501',6700
110,10,1000,'Los Angeles','90001',8500
111,10,1000,'Logan','84321',2300
112,10,1000,'Fremont','94539',9500
113,10,1000,'Omaha','96245',7500
114,11,2000,'New York','10001',6700
115,12,3000,'Los Angeles','90003',1000
```

Then we load `orders.txt` to the `/tmp` HDFS folder:

```
[root@sandbox order_data]# hadoop fs -put /root/order_data/orders.txt /tmp

[root@sandbox order_data]# hadoop fs -ls /tmp
Found 3 items
-rw-r--r--   1 root      hdfs        530 2017-09-02 18:06 /tmp/orders.txt
```

Load the `ORDERS_DATA` table as follows:

```
hive> load data inpath '/tmp/orders.txt' into table ORDERS_DATA;
Loading data to table orders.orders_data
Table orders.orders_data stats: [numFiles=1, numRows=0, totalSize=530,
rawDataSize=0]
OK
Time taken: 0.913 seconds

hive> select * from ORDERS_DATA;
OK
101      1     100   'Los Angeles'     '90001'     1200.0
102      2     200   'Los Angeles'     '90002'     1800.0
103      3     300   'Austin'    '78701'     6500.0
104      4     400   'Phoenix'   '85001'     7800.0
105      5     500   'Beverly Hills'   '90209'     7822.0
106      6     600   'Gaylord'   '49734'     8900.0
107      7     700   'Los Angeles'     '90001'     7002.0
108      8     800   'Los Angeles'     '90002'     8088.0
109      9     900   'Reno'      '89501'     6700.0
110      10    1000  'Los Angeles'     '90001'     8500.0
111      10    1000  'Logan'     '84321'     2300.0
112      10    1000  'Fremont'   '94539'     9500.0
113      10    1000  'Omaha'     '96245'     7500.0
114      11    2000  'New York'  '10001'     6700.0
115      12    3000  'Los Angeles'     '90003'     1000.0
Time taken: 0.331 seconds, Fetched: 15 row(s)
```

Let's assume we want to insert cities data in an `ORDERS_DATA` table. Each city orders data is of 1 TB in size. So the total data size of the `ORDERS_DATA` table will be 15 TB (there are 15 cities in the table). Now, if we write the following query to get all orders booked in `Los Angeles`:

```
hive>  select * from ORDERS where Ord_city = 'Los Angeles' ;
```

The query will run very slowly as it has to scan the entire table. The obvious idea is that we can create 10 different `orders` tables for each city and store `orders` data in the corresponding city of the `ORDERS_DATA` table. But instead of that, we can partition the `ORDERS_PART` table as follows:

```
hive> use orders;

hive> CREATE TABLE orders_part
    > (Ord_id INT,
    > Ord_month INT,
    > Ord_customer_id INT,
    > Ord_zip   STRING,
    > ORD_amt   FLOAT
    > )
    > PARTITIONED BY  (Ord_city INT)
    > ROW FORMAT DELIMITED
    > FIELDS TERMINATED BY  ','
    > ;
OK
Time taken: 0.305 seconds
hive>
```

Now, Hive organizes the tables into partitions for grouping similar types of data together based on a column or partition key. Let's assume that we have 10 `orders` files for each city, that is, `Orders1.txt` to `Orders10.txt`. The following example shows how to load each monthly file to each corresponding partition:

```
load data inpath '/tmp/orders.txt' into table orders_part
partition(Ord_city='Los Angeles');
load data inpath '/tmp/orders.txt' into table orders_part
partition(Ord_city='Austin');
load data inpath '/tmp/orders.txt' into table orders_part
partition(Ord_city='Phoenix');
load data inpath '/tmp/orders.txt' into table orders_part
partition(Ord_city='Beverly Hills');
load data inpath '/tmp/orders.txt' into table orders_part
partition(Ord_city='Gaylord');
load data inpath '/tmp/orders.txt' into table orders_part
partition(Ord_city=Reno');
```

```
load data inpath '/tmp/orders.txt' into table orders_part
partition(Ord_city='Fremont');
load data inpath '/tmp/orders.txt' into table orders_part
partition(Ord_city='Omaha');
load data inpath '/tmp/orders.txt' into table orders_part
partition(Ord_city='New York');
load data inpath '/tmp/orders.txt' into table orders_part
partition(Ord_city='Logan');

[root@sandbox order_data]# hadoop fs -ls
/apps/hive/warehouse/orders.db/orders_part
Found 10 items
drwxrwxrwx   - root hdfs          0 2017-09-02 18:32
/apps/hive/warehouse/orders.db/orders_part/ord_city=Austin
drwxrwxrwx   - root hdfs          0 2017-09-02 18:32
/apps/hive/warehouse/orders.db/orders_part/ord_city=Beverly Hills
drwxrwxrwx   - root hdfs          0 2017-09-02 18:32
/apps/hive/warehouse/orders.db/orders_part/ord_city=Fremont
drwxrwxrwx   - root hdfs          0 2017-09-02 18:32
/apps/hive/warehouse/orders.db/orders_part/ord_city=Gaylord
drwxrwxrwx   - root hdfs          0 2017-09-02 18:33
/apps/hive/warehouse/orders.db/orders_part/ord_city=Logan
drwxrwxrwx   - root hdfs          0 2017-09-02 18:32
/apps/hive/warehouse/orders.db/orders_part/ord_city=Los Angeles
drwxrwxrwx   - root hdfs          0 2017-09-02 18:32
/apps/hive/warehouse/orders.db/orders_part/ord_city=New York
drwxrwxrwx   - root hdfs          0 2017-09-02 18:32
/apps/hive/warehouse/orders.db/orders_part/ord_city=Omaha
drwxrwxrwx   - root hdfs          0 2017-09-02 18:32
/apps/hive/warehouse/orders.db/orders_part/ord_city=Phoenix
drwxrwxrwx   - root hdfs          0 2017-09-02 18:33
/apps/hive/warehouse/orders.db/orders_part/ord_city=Reno
[root@sandbox order_data]
```

Partitioning the data can greatly improve the performance of queries because the data is already separated into files based on the column value, which can decrease the number of mappers and greatly decrease the amount of shuffling and sorting of data in the resulting MapReduce job.

Hive static partitions and dynamic partitions

If you want to use a static partition in Hive, you should set the property as follows:

```
set hive.mapred.mode = strict;
```

In the preceding example, we have seen that we have to insert each monthly order file to each static partition individually. Static partition saves time in loading data compared to dynamic partition. We have to individually add a partition to the table and move the file into the partition of the table. If we have a lot partitions, writing a query to load data in each partition may become cumbersome. We can overcome this with a dynamic partition. In dynamic partitions, we can insert data into a partition table with a single SQL statement but still load data in each partition. Dynamic partition takes more time in loading data compared to static partition. When you have large data stored in a table, dynamic partition is suitable. If you want to partition a number of columns but you don't know how many columns they are, then dynamic partition is also suitable. Here are the hive dynamic partition properties you should allow:

```
SET hive.exec.dynamic.partition = true;
SET hive.exec.dynamic.partition.mode = nonstrict;
```

The following is an example of dynamic partition. Let's say we want to load data from the `ORDERS_PART` table to a new table called `ORDERS_NEW`:

```
hive> use orders;
OK
Time taken: 1.595 seconds
hive> drop table orders_New;
OK
Time taken: 0.05 seconds
hive> CREATE TABLE orders_New
    > (Ord_id INT,
    > Ord_month INT,
    > Ord_customer_id INT,
    > Ord_city  STRING,
    > Ord_zip   STRING,
    > ORD_amt   FLOAT
    > )
    > )
    > PARTITIONED BY  (Ord_city STRING)
    > ROW FORMAT DELIMITED
    > FIELDS TERMINATED BY  ','
    > ;
OK
Time taken: 0.458 seconds
hive>
```

Load data into the `ORDER_NEW` table from the `ORDERS_PART` table. Here, Hive will load all partitions of the `ORDERS_NEW` table dynamically:

```
hive> SET hive.exec.dynamic.partition = true;
hive> SET hive.exec.dynamic.partition.mode = nonstrict;
hive>
    > insert into table orders_new  partition(Ord_city) select * from
orders_part;
Query ID = root_20170902184354_2d409a56-7bfc-416e-913a-2323ea3b339a
Total jobs = 1
Launching Job 1 out of 1
Status: Running (Executing on YARN cluster with App id
application_1504299625945_0013)

--------------------------------------------------------------------------------
-----
        VERTICES      STATUS  TOTAL  COMPLETED  RUNNING  PENDING  FAILED
KILLED
--------------------------------------------------------------------------------
-----
Map 1 ..........   SUCCEEDED      1          1        0        0       0
0
--------------------------------------------------------------------------------
-----
VERTICES: 01/01  [==========================>>] 100%  ELAPSED TIME: 3.66 s
--------------------------------------------------------------------------------
-----
Loading data to table orders.orders_new partition (ord_city=null)
    Time taken to load dynamic partitions: 2.69 seconds
   Loading partition {ord_city=Logan}
   Loading partition {ord_city=Los Angeles}
   Loading partition {ord_city=Beverly Hills}
   Loading partition {ord_city=Reno}
   Loading partition {ord_city=Fremont}
   Loading partition {ord_city=Gaylord}
   Loading partition {ord_city=Omaha}
   Loading partition {ord_city=Austin}
   Loading partition {ord_city=New York}
   Loading partition {ord_city=Phoenix}
    Time taken for adding to write entity : 3
Partition orders.orders_new{ord_city=Austin} stats: [numFiles=1, numRows=1,
totalSize=13, rawDataSize=12]
Partition orders.orders_new{ord_city=Beverly Hills} stats: [numFiles=1,
numRows=1, totalSize=13, rawDataSize=12]
Partition orders.orders_new{ord_city=Fremont} stats: [numFiles=1,
numRows=1, totalSize=15, rawDataSize=14]
Partition orders.orders_new{ord_city=Gaylord} stats: [numFiles=1,
numRows=1, totalSize=13, rawDataSize=12]
Partition orders.orders_new{ord_city=Logan} stats: [numFiles=1, numRows=1,
totalSize=15, rawDataSize=14]
Partition orders.orders_new{ord_city=Los Angeles} stats: [numFiles=1,
numRows=6, totalSize=82, rawDataSize=76]
Partition orders.orders_new{ord_city=New York} stats: [numFiles=1,
numRows=1, totalSize=15, rawDataSize=14]
Partition orders.orders_new{ord_city=Omaha} stats: [numFiles=1, numRows=1,
totalSize=15, rawDataSize=14]
Partition orders.orders_new{ord_city=Phoenix} stats: [numFiles=1,
numRows=1, totalSize=13, rawDataSize=12]
Partition orders.orders_new{ord_city=Reno} stats: [numFiles=1, numRows=1,
totalSize=13, rawDataSize=12]
OK
Time taken: 10.493 seconds
hive>
```

Let's see how many partitions are created in `ORDERS_NEW`:

```
hive> show partitions ORDERS_NEW;
OK
ord_city=Austin
ord_city=Beverly Hills
ord_city=Fremont
ord_city=Gaylord
ord_city=Logan
ord_city=Los Angeles
ord_city=New York
ord_city=Omaha
ord_city=Phoenix
ord_city=Reno
Time taken: 0.59 seconds, Fetched: 10 row(s)
hive>
```

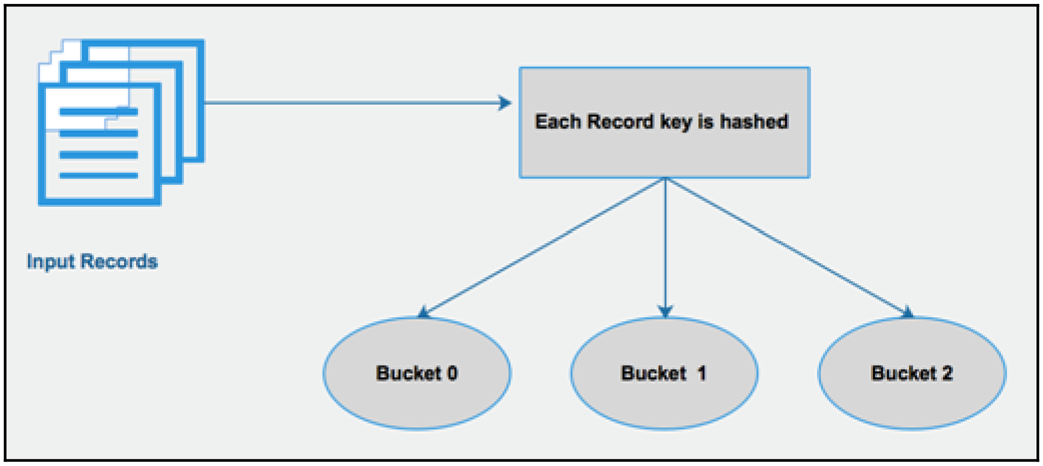
Now it is very clear when to use static and dynamic partitions. Static partitioning can be used when the partition column values are known well in advance before loading data into a hive table. In the case of dynamic partitions, partition column values are known only during loading of the data into the hive table.

Hive partition bucketing

Bucketing is a technique of decomposing a large dataset into more manageable groups. Bucketing is based on the hashing function. When a table is bucketed, all the table records with the same column value will go into the same bucket. Physically, each bucket is a file in a table folder just like a partition. In a partitioned table, Hive can group the data in multiple folders. But partitions prove effective when they are of a limited number and when the data is distributed equally among all of them. If there are a large number of partitions, then their use becomes less effective. So in that case, we can use bucketing. We can create a number of buckets explicitly during table creation.

How Hive bucketing works

The following diagram shows the working of Hive bucketing in detail:



If we decide to have three buckets in a table for a column, (`Ord_city`) in our example, then Hive will create three buckets with numbers 0-2 (*n-1*). During record insertion time, Hive will apply the Hash function to the `Ord_city` column of each record to decide the hash key. Then Hive will apply a modulo operator to each hash value. We can use bucketing in non-partitioned tables also. But we will get the best performance when the bucketing feature is used with a partitioned table. Bucketing has two key benefits:

- **Improved query performance**: During joins on the same bucketed columns, we can specify the number of buckets explicitly. Since each bucket is of equal size of data, map-side joins perform better on a bucketed table than a non-bucketed table. In a map-side join, the left-hand side table bucket will exactly know the dataset in the right-hand side bucket to perform a table join efficiently.
- **Improved sampling**: Because the data is already split up into smaller chunks.

Let's consider our `ORDERS_DATA` table example. It is partitioned in the `CITY` column. It may be possible that all of the cities do not have an equal distribution of orders. Some cities may have more orders than others. In that case, we will have lopsided partitions. This will affect query performance. Queries with cities that have more orders will be slower than for cities with fewer orders. We can solve this problem by bucketing the table. Buckets in the table are defined by the `CLUSTER` clause in the table DDL. The following examples explain the bucketing feature in detail.

Creating buckets in a non-partitioned table

First, we will create a `ORDERS_BUCK_non_partition` table:

```
SET hive.exec.dynamic.partition = true;
SET hive.exec.dynamic.partition.mode = nonstrict;
SET hive.exec.mx_dynamic.partition=20000;
SET hive.exec.mx_dynamic.partition.pernode=20000;
SET hive.enforce.bucketing = true;

hive> use orders;
OK
Time taken: 0.221 seconds
hive>
    > CREATE TABLE ORDERS_BUCKT_non_partition
    > (Ord_id INT,
    > Ord_month INT,
    > Ord_customer_id INT,
    > Ord_city  STRING,
    > Ord_zip   STRING,
    > ORD_amt   FLOAT
    > )
    > CLUSTERED BY (Ord_city) into 4 buckets stored as textfile;
OK
Time taken: 0.269 seconds
hive>
```

To refer to all Hive `SET` configuration parameters, please use this URL: `https://cwiki.apache.org/confluence/display/Hive/Configuration+Properties`.

Load the newly created non-partitioned bucket table:

```
hive> insert into ORDERS_BUCKT_non_partition select * from orders_data;
Query ID = root_20170902190615_1f557644-48d6-4fa1-891d-2deb7729fa2a
Total jobs = 1
Launching Job 1 out of 1
Tez session was closed. Reopening...
Session re-established.
Status: Running (Executing on YARN cluster with App id
application_1504299625945_0014)

---------------------------------------------------------------------------
-----
        VERTICES      STATUS  TOTAL  COMPLETED  RUNNING  PENDING  FAILED
KILLED
```

```
--------------------------------------------------------------------------------
-----
Map 1 ..........   SUCCEEDED      1          1        0        0       0
0
Reducer 2 ......   SUCCEEDED      4          4        0        0       0
0
--------------------------------------------------------------------------------
-----
VERTICES: 02/02  [==========================>>] 100%  ELAPSED TIME: 9.58 s
--------------------------------------------------------------------------------
-----
Loading data to table orders.orders_buckt_non_partition
Table orders.orders_buckt_non_partition stats: [numFiles=4, numRows=15,
totalSize=560, rawDataSize=545]
OK
Time taken: 15.55 seconds
hive>
```

The following command shows that Hive has created four buckets (folders), `00000[0-3]_0`, in the table:

```
[root@sandbox order_data]# hadoop fs -ls
/apps/hive/warehouse/orders.db/orders_buckt_non_partition
Found 4 items
-rwxrwxrwx   1 root hdfs         32 2017-09-02 19:06
/apps/hive/warehouse/orders.db/orders_buckt_non_partition/000000_0
-rwxrwxrwx   1 root hdfs        110 2017-09-02 19:06
/apps/hive/warehouse/orders.db/orders_buckt_non_partition/000001_0
-rwxrwxrwx   1 root hdfs        104 2017-09-02 19:06
/apps/hive/warehouse/orders.db/orders_buckt_non_partition/000002_0
-rwxrwxrwx   1 root hdfs        314 2017-09-02 19:06
/apps/hive/warehouse/orders.db/orders_buckt_non_partition/000003_0
[root@sandbox order_data]#
```

Creating buckets in a partitioned table

First, we will create a bucketed partition table. Here, the table is partitioned into four buckets on the `Ord_city` column, but subdivided into `Ord_zip` columns:

```
SET hive.exec.dynamic.partition = true;
SET hive.exec.dynamic.partition.mode = nonstrict;
SET hive.exec.mx_dynamic.partition=20000;
SET hive.exec.mx_dynamic.partition.pernode=20000;
SET hive.enforce.bucketing = true;
```

```
hive> CREATE TABLE ORDERS_BUCKT_partition
    > (Ord_id INT,
    > Ord_month INT,
    > Ord_customer_id INT,
    > Ord_zip   STRING,
    > ORD_amt   FLOAT
    > )
    > PARTITIONED BY  (Ord_city STRING)
    > CLUSTERED BY (Ord_zip) into 4 buckets stored as textfile;
OK
Time taken: 0.379 seconds
```

Load the bucketed partitioned table with another partitioned table (`ORDERS_PART`) with a dynamic partition:

```
hive> SET hive.exec.dynamic.partition = true;
hive> SET hive.exec.dynamic.partition.mode = nonstrict;
hive> SET hive.exec.mx_dynamic.partition=20000;
Query returned non-zero code: 1, cause: hive configuration
hive.exec.mx_dynamic.partition does not exists.
hive> SET hive.exec.mx_dynamic.partition.pernode=20000;
Query returned non-zero code: 1, cause: hive configuration
hive.exec.mx_dynamic.partition.pernode does not exists.
hive> SET hive.enforce.bucketing = true;
hive> insert into ORDERS_BUCKT_partition partition(Ord_city) select * from
orders_part;
Query ID = root_20170902194343_dd6a2938-6aa1-49f8-a31e-54dafbe8d62b
Total jobs = 1
Launching Job 1 out of 1
Status: Running (Executing on YARN cluster with App id
application_1504299625945_0017)

----------------------------------------------------------------------------
-----
        VERTICES      STATUS  TOTAL  COMPLETED  RUNNING  PENDING  FAILED
KILLED
----------------------------------------------------------------------------
-----
Map 1 ..........   SUCCEEDED      1          1        0        0       0
0
Reducer 2 ......   SUCCEEDED      4          4        0        0       0
0
----------------------------------------------------------------------------
-----
VERTICES: 02/02  [==========================>>] 100%  ELAPSED TIME: 7.13 s
----------------------------------------------------------------------------
-----
```

```
Loading data to table orders.orders_buckt_partition partition
(ord_city=null)
    Time taken to load dynamic partitions: 2.568 seconds
   Loading partition {ord_city=Phoenix}
   Loading partition {ord_city=Logan}
   Loading partition {ord_city=Austin}
   Loading partition {ord_city=Fremont}
   Loading partition {ord_city=Beverly Hills}
   Loading partition {ord_city=Los Angeles}
   Loading partition {ord_city=New York}
   Loading partition {ord_city=Omaha}
   Loading partition {ord_city=Reno}
   Loading partition {ord_city=Gaylord}
    Time taken for adding to write entity : 3
Partition orders.orders_buckt_partition{ord_city=Austin} stats:
[numFiles=1, numRows=1, totalSize=22, rawDataSize=21]
Partition orders.orders_buckt_partition{ord_city=Beverly Hills} stats:
[numFiles=1, numRows=1, totalSize=29, rawDataSize=28]
Partition orders.orders_buckt_partition{ord_city=Fremont} stats:
[numFiles=1, numRows=1, totalSize=23, rawDataSize=22]
Partition orders.orders_buckt_partition{ord_city=Gaylord} stats:
[numFiles=1, numRows=1, totalSize=23, rawDataSize=22]
Partition orders.orders_buckt_partition{ord_city=Logan} stats: [numFiles=1,
numRows=1, totalSize=26, rawDataSize=25]
Partition orders.orders_buckt_partition{ord_city=Los Angeles} stats:
[numFiles=1, numRows=6, totalSize=166, rawDataSize=160]
Partition orders.orders_buckt_partition{ord_city=New York} stats:
[numFiles=1, numRows=1, totalSize=23, rawDataSize=22]
Partition orders.orders_buckt_partition{ord_city=Omaha} stats: [numFiles=1,
numRows=1, totalSize=25, rawDataSize=24]
Partition orders.orders_buckt_partition{ord_city=Phoenix} stats:
[numFiles=1, numRows=1, totalSize=23, rawDataSize=22]
Partition orders.orders_buckt_partition{ord_city=Reno} stats: [numFiles=1,
numRows=1, totalSize=20, rawDataSize=19]
OK
Time taken: 13.672 seconds
hive>
```

Hive views

A Hive view is a logical table. It is just like any RDBMS view. The concept is the same. When a view is created, Hive will not store any data into it. When a view is created, Hive freezes the metadata. Hive does not support the materialized view concept of any RDBMS. The basic purpose of a view is to hide the query complexity. At times, HQL contains complex joins, subqueries, or filters. With the help of view, the entire query can be flattened out in a virtual table.

When a view is created on an underlying table, any changes to that table, or even adding or deleting the table, are invalidated in the view. Also, when a view is created, it only changes the metadata. But when that view is accessed by a query, it triggers the MapReduce job. A view is a purely logical object with no associated storage (no support for materialized views is currently available in Hive). When a query references a view, the view's definition is evaluated in order to produce a set of rows for further processing by the query. (This is a conceptual description. In fact, as part of query optimization, Hive may combine the view's definition with the queries, for example, pushing filters from the query down into the view.)

A view's schema is frozen at the time the view is created; subsequent changes to underlying tables (for example, adding a column) will not be reflected in the view's schema. If an underlying table is dropped or changed in an incompatible fashion, subsequent attempts to query the invalid view will fail. Views are read-only and may not be used as the target of `LOAD`/`INSERT`/`ALTER` for changing metadata. A view may contain `ORDER BY` and `LIMIT` clauses. If a referencing query also contains these clauses, the query-level clauses are evaluated after the view clauses (and after any other operations in the query). For example, if a view specifies `LIMIT 5` and a referencing query is executed as (`select * from v LIMIT 10`), then at most five rows will be returned.

Syntax of a view

Let's see a few examples of views:

```
CREATE VIEW [IF NOT EXISTS] [db_name.]view_name [(column_name [COMMENT
column_comment], ...) ]
  [COMMENT view_comment]
  [TBLPROPERTIES (property_name = property_value, ...)]
  AS SELECT ...;
```

I will demonstrate the advantages of views using the following few examples. Let's assume we have two tables, `Table_X` and `Table_Y`, with the following schema: `Table_XXCol_1` string, `XCol_2` string, `XCol_3` string, `Table_YYCol_1` string, `YCol_2` string, `YCol_3` string, and `YCol_4` string. To create a view exactly like the base tables, use the following code:

```
Create view table_x_view as select * from Table_X;
```

To create a view on selective columns of base tables, use the following:

```
Create view table_x_view as select xcol_1,xcol_3  from Table_X;
```

To create a view to filter values of columns of base tables, we can use:

```
Create view table_x_view as select * from Table_X where XCol_3 > 40 and
XCol_2 is not null;
```

To create a view to hide query complexities:

```
create view table_union_view  as select XCol_1, XCol_2, XCol_3,Null from
Table_X
   where XCol_2  = "AAA"
   union all
   select YCol_1, YCol_2, YCol_3, YCol_4 from Table_Y
   where YCol_3 = "BBB";
   create view table_join_view as select * from Table_X
   join Table_Y on Table_X. XCol_1 = Table_Y. YCol_1;
```

Hive indexes

The main purpose of the indexing is to search through the records easily and speed up the query. The goal of Hive indexing is to improve the speed of query lookup on certain columns of a table. Without an index, queries with predicates like `WHERE tab1.col1 = 10` load the entire table or partition and process all the rows. But if an index exists for `col1`, then only a portion of the file needs to be loaded and processed. The improvement in query speed that an index can provide comes at the cost of additional processing to create the index and disk space to store the index. There are two types of indexes:

- Compact index
- Bitmap index

The main difference is in storing mapped values of the rows in the different blocks.

Compact index

In HDFS, the data is stored in blocks. But scanning which data is stored in which block is time consuming. Compact indexing stores the indexed column's value and its `blockId`. So the query will not go to the table. Instead, the query will directly go to the compact index, where the column value and `blockId` are stored. No need to scan all the blocks to find data! So, while performing a query, it will first check the index and then go directly into that block.

Bitmap index

Bitmap indexing stores the combination of indexed column value and list of rows as a bitmap. Bitmap indexing is commonly used for columns with distinct values. Let's review a few examples: Base table, `Table_XXCol_1` Integer, `XCol_2` string, `XCol_3` integer, and `XCol_4` string. Create an index:

```
CREATE INDEX table_x_idx_1 ON TABLE table_x (xcol_1) AS 'COMPACT';
SHOW INDEX ON table_x_idx;
DROP INDEX table_x_idx ON table_x;

CREATE INDEX table_x_idx_2 ON TABLE table_x (xcol_1) AS 'COMPACT' WITH
DEFERRED REBUILD;
ALTER INDEX table_x_idx_2 ON table_x REBUILD;
SHOW FORMATTED INDEX ON table_x;
```

The preceding index is empty because it is created with the `DEFERRED REBUILD` clause, regardless of whether or not the table contains any data. After this index is created, the `REBUILD` command needs to be used to build the index structure. After creation of the index, if the data in the underlying table changes, the `REBUILD` command must be used to bring the index up to date. Create the index and store it in a text file:

```
CREATE INDEX table_x_idx_3 ON TABLE table_x (table_x) AS 'COMPACT' ROW
FORMAT DELIMITED
FIELDS TERMINATED BY 't'
STORED AS TEXTFILE;
```

Create a bitmap index:

```
CREATE INDEX table_x_bitmap_idx_4 ON TABLE table_x (table_x) AS 'BITMAP'
WITH DEFERRED REBUILD;
ALTER INDEX table_x_bitmap_idx_4 ON table03 REBUILD;
SHOW FORMATTED INDEX ON table_x;
DROP INDEX table_x_bitmap_idx_4 ON table_x;
```

JSON documents using Hive

JSON, is a minimal readable format for structuring data. It is used primarily to transmit data between a server and web application as an alternative to XML. JSON is built on two structures:

- A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

Please read more on JSON at the following URL: `http://www.json.org/`.

Example 1 – Accessing simple JSON documents with Hive (Hive 0.14 and later versions)

In this example, we will see how to query simple JSON documents using HiveQL. Let's assume we want to access the following `Sample-Json-simple.json` file in `HiveSample-Json-simple.json`:

```
{"username":"abc","tweet":"Sun shine is bright.","time1": "1366150681" }
{"username":"xyz","tweet":"Moon light is mild .","time1": "1366154481" }
```

View the `Sample-Json-simple.json` file:

```
[root@sandbox ~]# cat Sample-Json-simple.json
{"username":"abc","tweet":"Sun shine is bright.","timestamp": 1366150681 }
{"username":"xyz","tweet":"Moon light is mild .","timestamp": 1366154481 }
[root@sandbox ~]#
```

Load `Sample-Json-simple.json` into HDFS:

```
[root@sandbox ~]# hadoop fs -mkdir  /user/hive-simple-data/
[root@sandbox ~]# hadoop fs -put Sample-Json-simple.json /user/hive-simple-
data/
```

Create an external Hive table, `simple_json_table`:

```
hive> use orders;
OK
Time taken: 1.147 seconds
hive>
CREATE EXTERNAL TABLE simple_json_table (
username string,
tweet string,
time1 string)
ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe'
LOCATION '/user/hive-simple-data/';
OK
Time taken: 0.433 seconds
hive>
```

Now verify the records:

```
hive> select * from simple_json_table ;
OK
abc      Sun shine is bright.    1366150681
xyz      Moon light is mild .    1366154481
Time taken: 0.146 seconds, Fetched: 2 row(s)
hive>
```

Example 2 – Accessing nested JSON documents with Hive (Hive 0.14 and later versions)

We will see how to query Nested JSON documents using HiveQL. Let's assume we want to access the following `Sample-Json-complex.json` file in `HiveSample-Json-complex.json`:

```
{"DocId":"Doc1","User1":{"Id":9192,"Username":"u2452","ShippingAddress":{"A
ddress1":"6373 Sun Street","Address2":"apt 12","City":"Foster
City","State":"CA"},"Orders":[{"ItemId":5343,"OrderDate":"12/23/2017"},{"It
emId":7362,"OrderDate":"12/24/2017"}]}}
```

Load `Sample-Json-simple.json` into HDFS:

```
[root@sandbox ~]# hadoop fs -mkdir  /user/hive-complex-data/
[root@sandbox ~]# hadoop fs -put Sample-Json-complex.json /user/hive-
complex-data/
```

Create an external Hive table, `json_nested_table`:

```
hive>
CREATE EXTERNAL TABLE json_nested_table(
DocId string,
user1 struct<Id: int, username: string,
shippingaddress:struct<address1:string,address2:string,city:string,state:st
ring>, orders:array<struct<ItemId:int,orderdate:string>>>
)
```

```
ROW FORMAT SERDE
'org.apache.hive.hcatalog.data.JsonSerDe'
LOCATION
'/user/hive-complex-data/';
OK
Time taken: 0.535 seconds
hive>
```

Verify the records:

```
hive> select DocId,user1.username,user1.orders FROM json_nested_table;
OK
Doc1     u2452
[{"itemid":5343,"orderdate":"12/23/2017"},{"itemid":7362,"orderdate":"12/24
/2017"}]
Time taken: 0.598 seconds, Fetched: 1 row(s)
hive>
```

Example 3 – Schema evolution with Hive and Avro (Hive 0.14 and later versions)

In production, we have to change the table structure to address new business requirements. The table schema has to change to add/delete/rename table columns. Any of these changes affect downstream ETL jobs adversely. In order avoid these, we have to make corresponding changes to ETL jobs and target tables.

Schema evolution allows you to update the schema used to write new data while maintaining backwards compatibility with the schemas of your old data. Then you can read it all together as if all of the data has one schema. Please read more on Avro serialization at the following URL: `https://avro.apache.org/`. In the following example, I will demonstrate how Avro and Hive tables absorb the changes of source table's schema changes without ETL job failure. We will create a customer table in the MySQL database and load it to the target Hive external table using Avro files. Then we will add one more column to the source tables to see how a Hive table absorbs that change without any errors. Connect to MySQL to create a source table (`customer`):

```
mysql -u root -p

GRANT ALL PRIVILEGES ON *.* TO 'sales'@'localhost' IDENTIFIED BY 'xxx';

mysql -u sales  -p

mysql> create database orders;
mysql> use orders;

CREATE TABLE customer(
cust_id INT ,
cust_name  VARCHAR(20) NOT NULL,
cust_city VARCHAR(20) NOT NULL,
PRIMARY KEY ( cust_id )
);
```

Insert records into the `customer` table:

```
INSERT into customer (cust_id,cust_name,cust_city) values (1,'Sam
James','Austin');
INSERT into customer (cust_id,cust_name,cust_city) values (2,'Peter
Carter','Denver');
INSERT into customer (cust_id,cust_name,cust_city) values (3,'Doug
Smith','Sunnyvale');
INSERT into customer (cust_id,cust_name,cust_city) values (4,'Harry
Warner','Palo Alto');
```

On Hadoop, run the following `sqoop` command to import the `customer` table and store data in Avro files into HDFS:

```
hadoop fs -rmr /user/sqoop_data/avro
sqoop import -Dmapreduce.job.user.classpath.first=true
--connect jdbc:mysql://localhost:3306/orders
--driver com.mysql.jdbc.Driver
--username sales --password xxx
--target-dir /user/sqoop_data/avro
--table customer
--as-avrodatafile
```

Verify the target HDFS folder:

```
[root@sandbox ~]# hadoop fs -ls /user/sqoop_data/avro
Found 7 items
-rw-r--r--   1 root hdfs          0 2017-09-09 08:57
/user/sqoop_data/avro/_SUCCESS
-rw-r--r--   1 root hdfs        472 2017-09-09 08:57
/user/sqoop_data/avro/part-m-00000.avro
-rw-r--r--   1 root hdfs        475 2017-09-09 08:57
/user/sqoop_data/avro/part-m-00001.avro
-rw-r--r--   1 root hdfs        476 2017-09-09 08:57
/user/sqoop_data/avro/part-m-00002.avro
-rw-r--r--   1 root hdfs        478 2017-09-09 08:57
/user/sqoop_data/avro/part-m-00003.avro
```

Create a Hive external table to access Avro files:

```
use orders;
drop table customer ;
CREATE EXTERNAL TABLE customer
(
cust_id INT ,
cust_name  STRING ,
cust_city STRING
)
STORED AS AVRO
location '/user/sqoop_data/avro/';
```

Verify the Hive `customer` table:

```
hive> select * from customer;
OK
1  Sam James   Austin
2  Peter Carter      Denver
3  Doug Smith  Sunnyvale
4  Harry Warner      Palo Alto
Time taken: 0.143 seconds, Fetched: 4 row(s)
hive>
```

Perfect! We have no errors. We successfully imported the source `customer` table to the target Hive table using Avro serialization. Now, we add one column to the source table and import it again to verify that we can access the target Hive table without any schema changes. Connect to MySQL and add one more column:

```
mysql -u sales  -p

mysql>
ALTER TABLE customer
ADD COLUMN cust_state VARCHAR(15) NOT NULL;

mysql> desc customer;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| cust_id    | int(11)     | NO   | PRI | 0       |       |
| cust_name  | varchar(20) | NO   |     | NULL    |       |
| cust_city  | varchar(20) | NO   |     | NULL    |       |
| CUST_STATE | varchar(15) | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
4 rows in set (0.01 sec)

mysql>
```

Now insert rows:

```
INSERT into customer (cust_id,cust_name,cust_city,cust_state) values
(5,'Mark Slogan','Huston','TX');
INSERT into customer (cust_id,cust_name,cust_city,cust_state) values
(6,'Jane Miller','Foster City','CA');
```

On Hadoop, run the following `sqoop` command to import the `customer` table so as to append the new address column and data. I have used the `append` and `where "cust_id > 4"` parameters to import only the new rows:

```
sqoop import -Dmapreduce.job.user.classpath.first=true
--connect jdbc:mysql://localhost:3306/orders
--driver com.mysql.jdbc.Driver
--username sales --password xxx
--table customer
--append
--target-dir /user/sqoop_data/avro
--as-avrodatafile
--where "cust_id > 4"
```

Verify the HDFS folder:

```
[root@sandbox ~]# hadoop fs -ls /user/sqoop_data/avro
Found 7 items
-rw-r--r--   1 root hdfs          0 2017-09-09 08:57
/user/sqoop_data/avro/_SUCCESS
-rw-r--r--   1 root hdfs        472 2017-09-09 08:57
/user/sqoop_data/avro/part-m-00000.avro
-rw-r--r--   1 root hdfs        475 2017-09-09 08:57
/user/sqoop_data/avro/part-m-00001.avro
-rw-r--r--   1 root hdfs        476 2017-09-09 08:57
/user/sqoop_data/avro/part-m-00002.avro
-rw-r--r--   1 root hdfs        478 2017-09-09 08:57
/user/sqoop_data/avro/part-m-00003.avro
-rw-r--r--   1 root hdfs        581 2017-09-09 09:00
/user/sqoop_data/avro/part-m-00004.avro
-rw-r--r--   1 root hdfs        586 2017-09-09 09:00
/user/sqoop_data/avro/part-m-00005.avro
```

Now, let's verify that our target Hive table is still able to access old and new Avro files:

```
hive> select * from customer;
OK
1  Sam James   Austin
2  Peter Carter      Denver
3  Doug Smith  Sunnyvale
4  Harry Warner      Palo Alto
Time taken: 0.143 seconds, Fetched: 4 row(s
```

Great! No errors. Still, it's business as usual; now we will add one new column to the Hive table to see the newly added Avro files:

```
hive> use orders;
hive> ALTER TABLE customer ADD COLUMNS (cust_state STRING);
hive> desc customer;
OK
cust_id             int
cust_name           string
cust_city           string
cust_state          string
Time taken: 0.488 seconds, Fetched: 4 row(s
```

Verify the Hive table for new data:

```
hive> select * from customer;
OK
1  Sam James   Austin      NULL
2  Peter Carter      Denver      NULL
3  Doug Smith  Sunnyvale   NULL
4  Harry Warner      Palo Alto   NULL
5  Mark Slogan Huston      TX
6  Jane Miller Foster City CA
Time taken: 0.144 seconds, Fetched: 6 row(s)
hive>
```

Awesome! Take a look at customer IDs `5` and `6`. We can see the newly added column (`cust_state`) with values. You can experiment the delete column and replace column feature with the same technique. Now we have a fairly good idea about how to access data using Apache Hive. In the next section, we will learn about accessing data using HBase, which is a NoSQL data store.

Apache HBase

We have just learned about Hive, which is a database where users can access data using SQL commands. But there are certain databases where users cannot use SQL commands. Those databases are known as **NoSQL data stores**. HBase is a NoSQL database. So, what is actually meant by NoSQL? NoSQL means not only SQL. In NoSQL data stores like HBase, the main features of RDBMS, such as validation and consistency, are relaxed. Also, another important difference between RDBMS or SQL databases and NoSQL databases is schema on write versus schema on read. In schema on write, the data is validated at the time of writing to the table, whereas schema on read supports validation of data at the time of reading it. In this way, NoSQL data stores support storage of huge data velocity due to the relaxation of basic data validation at the time of writing data. There are about 150 NoSQL data stores in the market today. Each of these NoSQL data stores has some unique features to offer. Some popular NoSQL data stores are HBase, Cassandra, MongoDB, Druid, Apache Kudu, and Accumulo, and so on.

You can get a detailed list of all types of NoSQL databases at `http://nosql-database.org/`.

HBase is a popular NoSQL database used by many big companies such as Facebook, Google, and so on.

Differences between HDFS and HBase

The following explains the key difference between HDFS and HBase. Hadoop is built on top of HDFS, which has support for storing large volumes (petabytes) of datasets. These datasets are accessed using batch jobs, by using MapReduce algorithms. In order to find a data element in such a huge dataset, the entire dataset needs to be scanned. HBase, on the other hand, is built on top of HDFS and provides fast record lookups (and updates) for large tables. HBase internally puts your data in indexed StoreFiles that exist on HDFS for high-speed lookup.

Differences between Hive and HBase

HBase is a database management system; it supports both transaction processing and analytical processing. Hive is a data warehouse system, which can be used only for analytical processing. HBase supports low latency and random data access operations. Hive only supports batch processing, which leads to high latency. HBase does not support any SQL interface to interact with the table data. You may have to write Java code to read and write data to HBase tables. At times, Java code becomes very complex to process data sets involving joins of multiple data sets. But Hive supports very easy access with SQL, which makes it very easy to read and write data to its tables. In HBase, data modeling involves flexible data models and column-oriented data storage, which must support data denormalization. The columns of HBase tables are decided at the time of writing data into the tables. In Hive, the data model involves tables with a fixed schema like an RDBMS data model.

Key features of HBase

The following are a few key features of HBase:

- **Sorted rowkeys**: In HBase, data processing is down with three basic operations/APIs: get, put, and scan. All three of these APIs access data using rowkeys to ensure smooth data access. As scans are done over a range of rows, HBase lexicographically orders rows according to their rowkeys. Using these sorted rowkeys, a scan can be defined simply from its start and stop rowkeys. This is extremely powerful to get all relevant data in a single database call. The application developer can design a system to access recent datasets by querying recent rows based on their timestamp as all rows are stored in a table in sorted order based on the latest timestamp.
- **Control data sharding**: HBase Table rowkey strongly influences data sharding. Table data is sorted in ascending order by rowkey, column families, and column key. A solid rowkey design is very important to ensure data is evenly distributed across the Hadoop cluster. As rowkeys determine the sort order of a table's row, each region in the table ends up being responsible for the physical storage of a part of the row key space.
- **Strong consistency**: HBase favors consistency over availability. It also supports ACID-level semantics on a per row basis. It, of course, impacts the write performance, which will tend to be slower. Overall, the trade-off plays in favor of the application developer, who will have the guarantee that the data store always the right value of the data.

- **Low latency processing**: HBase supports fast, random reads and writes to all data stored.
- **Flexibility**: HBase supports any type—structured, semi-structured, unstructured.
- **Reliability**: HBase table data block is replicated multiple times to ensure protection against data loss. HBase also supports fault tolerance. The table data is always available for processing even in case of failure of any regional server.

HBase data model

These are the key components of an HBase data model:

- **Table**: In HBase, data is stored in a logical object, called **table**, that has multiple rows.
- **Row**: A row in HBase consists of a row key and one or more columns. The row key sorts rows. The goal is to store data in such a way that related rows are near each other. The row key can be a combination of one of more columns. The row key is like the primary key of the table, which must be unique. HBase uses row keys to find data in a column. For example, `customer_id` can be a row key for the `customer` table.
- **Column**: A column in HBase consists of a column family and a column qualifier.
- **Column qualifier**: It is the column name of a table.
- **Cell**: This is a combination of row, column family, and column qualifier, and contains a value and a timestamp which represents the value's version.
- **Column family**: It is a collection of columns that are co-located and stored together, often for performance reasons. Each column family has a set of storage properties, such as cached, compression, and data encodation.

Difference between RDBMS table and column - oriented data store

We all know how data is stored in any RDBMS table. It looks like this:

ID	`Column_1`	`Column_2`	`Column_3`	`Column_4`
1	A	11	P	XX
2	B	12	Q	YY

3	C	13	R	ZZ
4	D	14	S	XX1

The column ID is used as a unique/primary key of the table to access data from other columns of the table. But in a column-oriented data store like HBase, the same table is divided into key and value and is stored like this:

Key		**Value**
Row	**Column**	**Column Value**
1	`Column_1`	`A`
1	`Column_2`	`11`
1	`Column_3`	`P`
1	`Column_4`	`XX`
2	`Column_1`	`B`
2	`Column_2`	`12`
2	`Column_3`	`Q`
2	`Column_4`	`YY`
3	`Column_1`	`C`
3	`Column_2`	`13`
3	`Column_3`	`R`
3	`Column_4`	`ZZ`

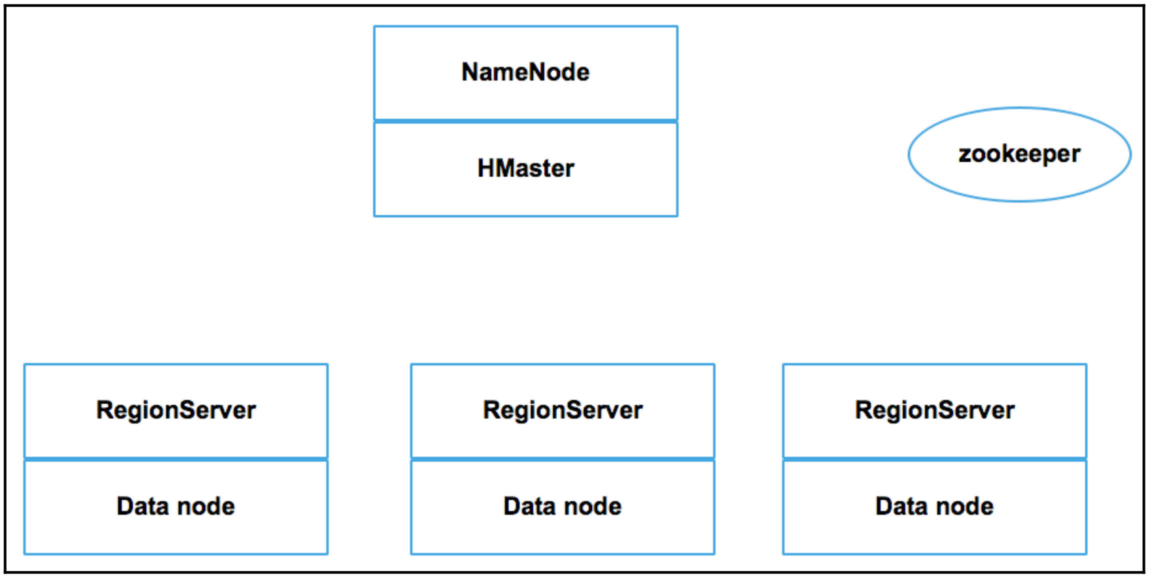
In HBase, each table is a sorted map format, where each key is sorted in ascending order. Internally, each key and value is serialized and stored on the disk in byte array format. Each column value is accessed by its corresponding key. So, in the preceding table, we define a key, which is a combination of two columns, *row + column*. For example, in order to access the `Column_1` data element of row 1, we have to use a key, row 1 + `column_1`. That's the reason the row key design is very crucial in HBase. Before creating the HBase table, we have to decide a column family for each column. A column family is a collection of columns, which are co-located and stored together, often for performance reasons. Each column family has a set of storage properties, such as cached, compression, and data encodation. For example, in a typical `CUSTOMER` table, we can define two column families, namely `cust_profile` and `cust_address`. All columns related to the customer address are assigned to the column family `cust_address`; all other columns, namely `cust_id`, `cust_name`, and `cust_age`, are assigned to the column family `cust_profile`. After assigning the column families, our sample table will look like the following:

Key		**Value**		
Row	**Column**	**Column family**	**Value**	**Timestamp**
1	`Column_1`	`cf_1`	`A`	`1407755430`
1	`Column_2`	`cf_1`	`11`	`1407755430`
1	`Column_3`	`cf_1`	`P`	`1407755430`
1	`Column_4`	`cf_2`	`XX`	`1407755432`
2	`Column_1`	`cf_1`	`B`	`1407755430`
2	`Column_2`	`cf_1`	`12`	`1407755430`
2	`Column_3`	`cf_1`	`Q`	`1407755430`
2	`Column_4`	`cf_2`	`YY`	`1407755432`
3	`Column_1`	`cf_1`	`C`	`1407755430`
3	`Column_2`	`cf_1`	`13`	`1407755430`
3	`Column_3`	`cf_1`	`R`	`1407755430`
3	`Column_4`	`cf_2`	`ZZ`	`1407755432`

While inserting data into a table, HBase will automatically add a timestamp for each version of the cell.

HBase architecture

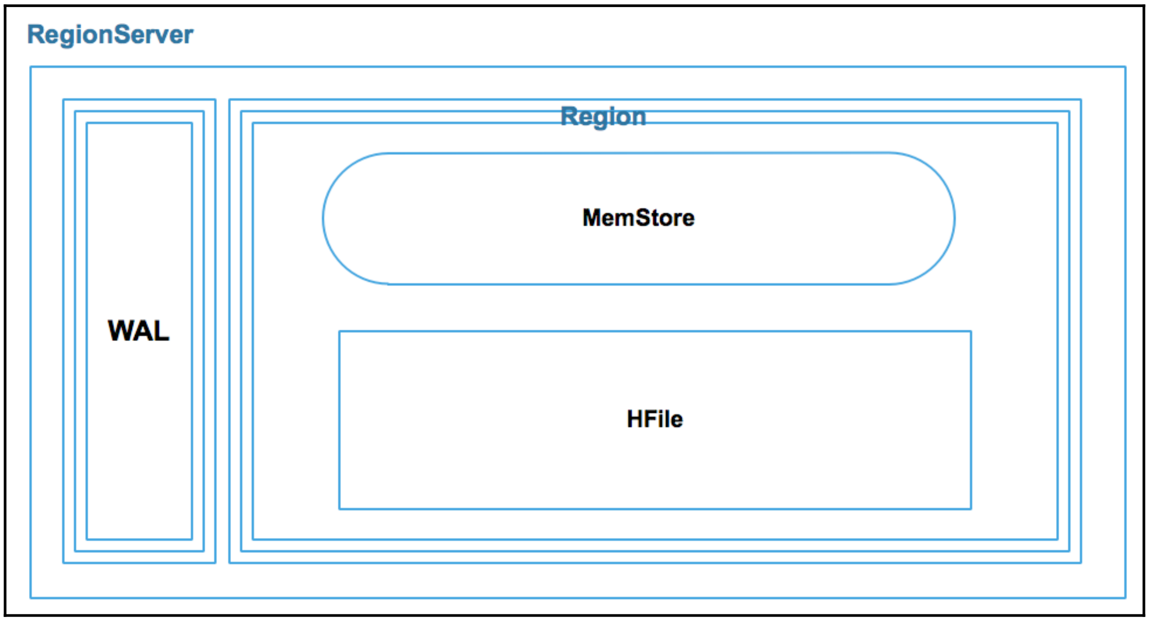
If we want to read data from an HBase table, we have to give an appropriate row ID, and HBase will perform a lookup based on the given row ID. HBase uses the following sorted nested map to return the column value of the row ID: row ID a column family, a column at timestamp, and value. HBase is always deployed on Hadoop. The following is a typical installation:



It is a master server of the HBase cluster and is responsible for the administration, monitoring, and management of RegionServers, such as assignment of regions to RegionServer, region splits, and so on. In a distributed cluster, the HMaster typically runs on the Hadoop NameNode.

ZooKeeper is a coordinator of HBase cluster. HBase uses ZooKeeper as a distributed coordination service to maintain server state in the cluster. ZooKeeper maintains which servers are alive and available, and provides server failure notification. RegionServer is responsible for management of regions. RegionServer is deployed on DataNode. It serves data for reads and writes. RegionServer is comprised of the following additional components:

- Regions: HBase tables are divided horizontally by row key range into regions. A region contains all rows in the table between the region's start key and end key. **Write-ahead logging** (**WAL**) is used to store new data that has not yet stored on disk.
- MemStore is a write cache. It stores new data that has not yet been written to disk. It is sorted before writing to disk. There is one MemStore per column family per region. Hfile stores the rows as sorted key/values on disk/HDFS:



HBase architecture in a nutshell

- The HBase cluster is comprised of one active master and one or more backup master servers
- The cluster has multiple RegionServers
- The HBase table is always large and rows are divided into partitions/shards called **regions**
- Each RegionServer hosts one or many regions
- The HBase catalog is known as META table, which stores the locations of table regions
- ZooKeeper stores the locations of the META table
- During a write, the client sends the put request to the HRegionServer
- Data is written to WAL
- Then data is pushed into MemStore and an acknowledgement is sent to the client
- Once enough data is accumulated in MemStore, it flushes data to the Hfile on HDFS
- The HBase compaction process activates periodically to merge multiple HFiles into one Hfile (called **compaction**)

HBase rowkey design

Rowkey design is a very crucial part of HBase table design. During key design, proper care must be taken to avoid hotspotting. In case of poorly designed keys, all of the data will be ingested into just a few nodes, leading to cluster imbalance. Then, all the reads have to be pointed to those few nodes, resulting in slower data reads. We have to design a key that will help load data equally to all nodes of the cluster. Hotspotting can be avoided by the following techniques:

- **Key salting**: It means adding an arbitrary value at the beginning of the key to make sure that the rows are distributed equally among all the table regions. Examples are `aa-customer_id`, `bb-customer_id`, and so on.
- **Key hashing**: The key can be hashed and the hashing value can be used as a rowkey, for example, `HASH(customer_id)`.
- **Key with reverse timestamp**: In this technique, you have to define a regular key and then attach a reverse timestamp to it. The timestamp has to be reversed by subtracting it from any arbitrary maximum value and then attached to the key. For example, if `customer_id` is your row ID, the new key will be `customer_id` + reverse timestamp.

The following are the guidelines while designing a HBase table:

- Define no more than two column families per table
- Keep column family names as small as possible
- Keep column names as small as possible
- Keep the rowkey length as small as possible
- Do not set the row version at a high level
- The table should not have more than 100 regions

Example 4 – loading data from MySQL table to HBase table

We will use the same `customer` table that we created earlier:

```
mysql -u sales -p
mysql> use orders;
mysql> select * from customer;
+---------+-------------+-------------+-----------+
| cust_id | cust_name | cust_city | InsUpd_on |
+---------+-------------+-------------+-----------+
| 1 | Sam James | Austin | 1505095030 |
| 2 | Peter Carter | Denver | 1505095030 |
| 3 | Doug Smith | Sunnyvale | 1505095030 |
| 4 | Harry Warner | Palo Alto | 1505095032 |
| 5 | Jen Turner | Cupertino | 1505095036 |
| 6 | Emily Stone | Walnut Creek | 1505095038 |
| 7 | Bill Carter | Greenville | 1505095040 |
| 8 | Jeff Holder | Dallas | 1505095042 |
| 10 | Mark Fisher | Mil Valley | 1505095044 |
| 11 | Mark Fisher | Mil Valley | 1505095044 |
+---------+-------------+-------------+-----------+
10 rows in set (0.00 sec)
```

Start HBase and create a `customer` table in HBase:

```
hbase shell
create 'customer','cf1'
```

Load MySQL `customer` table data in HBase using Sqoop:

```
hbase
sqoop import --connect jdbc:mysql://localhost:3306/orders --driver
com.mysql.jdbc.Driver --username sales --password sales1 --table customer -
-hbase-table customer --column-family cf1 --hbase-row-key cust_id
```

Verify the HBase table:

```
hbase shell
scan 'customer'
```

You must see all 11 rows in the HBase table.

Example 5 – incrementally loading data from MySQL table to HBase table

```
mysql -u sales -p
mysql> use orders;
```

Insert a new customer and update the existing one:

```
mysql> Update customer set cust_city = 'Dublin', InsUpd_on = '1505095065'
where cust_id = 4;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0

mysql> INSERT into customer (cust_id,cust_name,cust_city,InsUpd_on) values
(12,'Jane Turner','Glen Park',1505095075);
Query OK, 1 row affected (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from customer;
+---------+--------------+--------------+------------+
| cust_id | cust_name | cust_city | InsUpd_on |
+---------+--------------+--------------+------------+
| 1 | Sam James | Austin | 1505095030 |
| 2 | Peter Carter | Denver | 1505095030 |
| 3 | Doug Smith | Sunnyvale | 1505095030 |
| 4 | Harry Warner | Dublin | 1505095065 |
| 5 | Jen Turner | Cupertino | 1505095036 |
| 6 | Emily Stone | Walnut Creek | 1505095038 |
| 7 | Bill Carter | Greenville | 1505095040 |
| 8 | Jeff Holder | Dallas | 1505095042 |
| 10 | Mark Fisher | Mil Valley | 1505095044 |
| 11 | Mark Fisher | Mil Valley | 1505095044 |
| 12 | Jane Turner | Glen Park | 1505095075 |
+---------+-------------+-------------+------------+
11 rows in set (0.00 sec)
mysql>
```

Example 6 – Load the MySQL customer changed data into the HBase table

Here, we have used the InsUpd_on column as our ETL date:

```
sqoop import --connect jdbc:mysql://localhost:3306/orders --driver
com.mysql.jdbc.Driver --username sales --password sales1 --table customer -
-hbase-table customer --column-family cf1 --hbase-row-key cust_id --append
-- -m 1 --where "InsUpd_on > 1505095060"

hbase shell
hbase(main):010:0> get 'customer', '4'
COLUMN          CELL
cf1:InsUpd_on   timestamp=1511509774123, value=1505095065
cf1:cust_city   timestamp=1511509774123, value=Dublin
cf1:cust_name   timestamp=1511509774123, value=Harry Warner
3 row(s) in 0.0200 seconds

hbase(main):011:0> get 'customer', '12'
COLUMN           CELL
cf1:InsUpd_on    timestamp=1511509776158, value=1505095075
cf1:cust_city    timestamp=1511509776158, value=Glen Park
cf1:cust_name    timestamp=1511509776158, value=Jane Turner
3 row(s) in 0.0050 seconds

hbase(main):012:0>
```

Example 7 – Hive HBase integration

Now, we will access the HBase `customer` table using the Hive external table:

```
create external table customer_hbase(cust_id string, cust_name string,
cust_city string, InsUpd_on string)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'with
serdeproperties
("hbase.columns.mapping"=":key,cf1:cust_name,cf1:cust_city,cf1:InsUpd_on")t
blproperties("hbase.table.name"="customer");

hive> select * from customer_hbase;
OK
1 Sam James Austin 1505095030
10 Mark Fisher Mil Valley 1505095044
11 Mark Fisher Mil Valley 1505095044
12 Jane Turner Glen Park 1505095075
2 Peter Carter Denver 1505095030
3 Doug Smith Sunnyvale 1505095030
4 Harry Warner Dublin 1505095065
5 Jen Turner Cupertino 1505095036
6 Emily Stone Walnut Creek 1505095038
7 Bill Carter Greenville 1505095040
8 Jeff Holder Dallas 1505095042
Time taken: 0.159 seconds, Fetched: 11 row(s)
hive>
```

Summary

In this chapter, we saw how data is stored and accessed using a Hadoop SQL interface called Hive. We studied various partitioning and indexing strategies in Hive. The working examples helped us to understand JSON data access and schema evolution using Avro in Hive. In the second section of the chapter, we studied a NoSQL data store called HBase and its difference with respect to RDBMS. The row design of the HBase table is very crucial to balancing reads and writes to avoid region hotspots. One has to keep in mind the HBase table design best practices discussed in this chapter. The working example shows the easier paths of data ingestions into an HBase table and its integration with Hive.

In the next chapter, we will take a look at tools and techniques for designing real-time data analytics.

6
Designing Real-Time Streaming Data Pipelines

The first three chapters of this book all dealt with batch data. Having learned about the installation of Hadoop, data ingestion tools and techniques, and data stores, let's turn to data streaming. Not only will we look at how we can handle real-time data streams, but also how to design pipelines around them.

In this chapter, we will cover the following topics:

- Real-time streaming concepts
- Real-time streaming components
- Apache Flink versus Spark
- Apache Spark versus Storm

Real-time streaming concepts

Let's understand a few key concepts relating to real-time streaming applications in the following sections.

Data stream

The data stream is a continuous flow of data from one end to another end, from sender to receiver, from producer to consumer. The speed and volume of the data may vary; it may be 1 GB of data per second or it may be 1 KB of data per second or per minute.

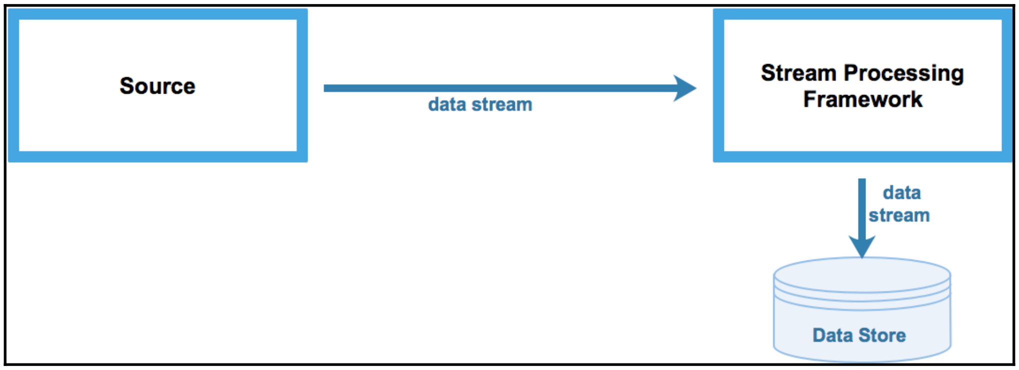
Batch processing versus real-time data processing

In batch processing, data is collected in batches and each batch is sent for processing. The batch interval can be anything from one day to one minute. In today's data analytics and business intelligence world, data will not be processed in a batch for more than one day. Otherwise, business teams will not have any insight about what's happening to the business in a day-to-day basis. For example, the enterprise data warehousing team may collect all the orders made during the last 24 hours and send all these collected orders to the analytics engine for reporting.

The batch can be of one minute too. In the Spark framework (we will learn Spark in `Chapter 7`, *Large-Scale Data Processing Frameworks*), data is processed in micro batches.

In real-time processing, data (event) is transferred (streamed) from the producer (sender) to the consumer (receiver) as soon as an event is produced at the source end. For example, on an e-commerce website, orders gets processed immediately in an analytics engine as soon as the customer places the same order on that website. The advantage is that the business team of that company gets full insights about its business in real time (within a few milliseconds or sub milliseconds). It will help them adjust their promotions to increase their revenue, all in real-time.

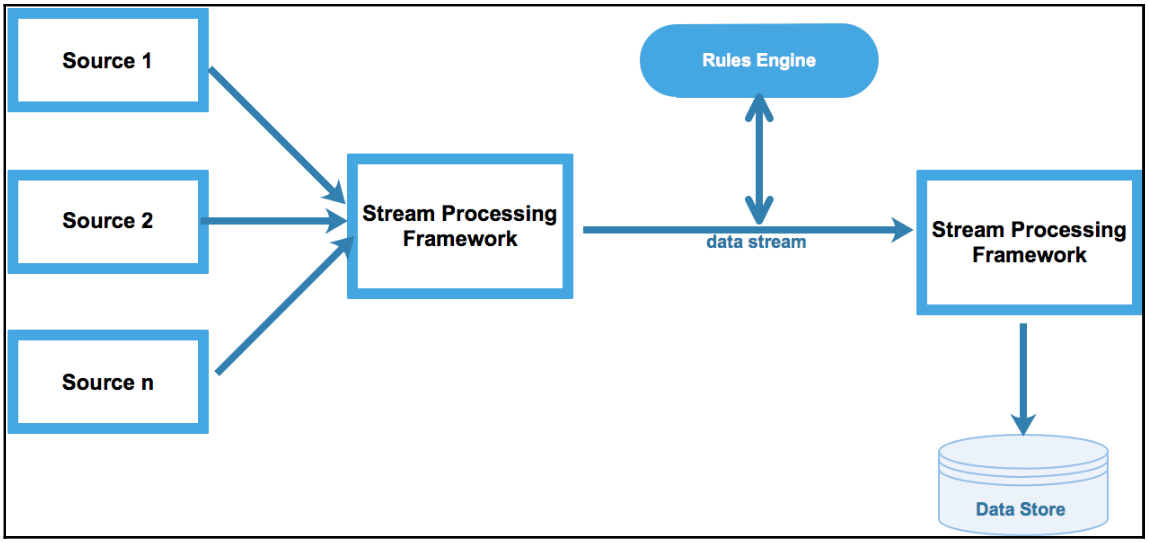
The following image explains stream-processing architecture:



Complex event processing

Complex event processing (**CEP**) is event processing that combines data from multiple sources to discover complex relationships or patterns. The goal of CEP is to identify meaningful events (such as opportunities or threats) and respond to them as quickly as possible. Fundamentally, CEP is about applying business rules to streaming event data. For example, CEP is used in use cases, such as stock trading, fraud detection, medical claim processing, and so on.

The following image explains stream-processing architecture:



Continuous availability

Any real-time application is expected to be available all the time with no stoppage whatsoever. The event collection, processing, and storage components should be configured with the underlined assumptions of high availability. Any failure to any components will cause major disruptions to the running of the business. For example, in a credit card fraud detection application, all the fraudulent transactions need to be declined. If the application stops midway and is unable to decline fraudulent transactions, then it will result in heavy losses.

Low latency

In any real-time application, the event should flow from source to target in a few milliseconds. The source collects the event, and a processing framework moves the event to its target data store where it can be analyzed further to find trends and patterns. All these should happen in real time, otherwise it may impact business decisions. For example, in a credit card fraud detection application, it is expected that all incoming transactions should be analyzed to find possible fraudulent transactions, if any. If the stream processing takes more than the desired period of time, it may be possible that these transactions may pass through the system, causing heavy losses to the business.

Scalable processing frameworks

Hardware failure may cause disruption to the stream processing application. To avoid this common scenario, we always need a processing framework that offers built-in APIs to support continuous computation, fault tolerant event state management, checkpoint features in the event of failures, in-flight aggregations, windowing, and so on. Fortunately, all the recent Apache projects such as Storm, Spark, Flink, and Kafka do support all and more of these features out of the box. The developer can use these APIs using Java, Python, and Scala.

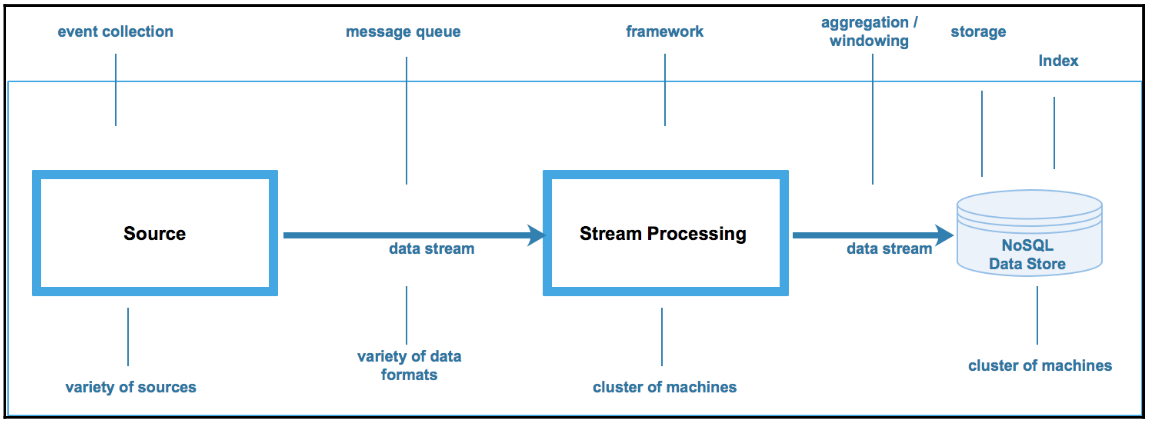
Horizontal scalability

The stream-processing platform should support horizontal scalability. That means adding more physical servers to the cluster in the event of a higher incoming data load to maintain throughput SLA. This way, the performance of processing can be increased by adding more nodes rather than adding more CPUs and memory to the existing servers; this is called **vertical scalability**.

Storage

The preferable format of a stream is key-value pair. This format is very well represented by the JSON and Avro formats. The preferred storage to persist key-value type data is NoSQL data stores such as HBase and Cassandra. There are in total 100 NoSQL open source databases in the market these days. It's very challenging to choose the right database, one which supports storage to real-time events, because all these databases offer some unique features for data persistence. A few examples are schema agnostic, highly distributable, commodity hardware support, data replication, and so on.

The following image explains all stream processing components:



In this chapter we will talk about message queue and stream processing frameworks in detail. In the next chapter, we will focus on data indexing techniques.

Real-time streaming components

In the following sections we will walk through some important real-time streaming components.

Message queue

The message queue lets you publish and subscribe to a stream of events/records. There are various alternatives we can use as a message queue in our real-time stream architecture. For example, there is RabbitMQ, ActiveMQ, and Kafka. Out of these, Kafka has gained tremendous popularity due to its various unique features. Hence, we will discuss the architecture of Kafka in detail. A discussion of RabbitMQ and ActiveMQ is beyond the scope of this book.

So what is Kafka?

Kafka is a fast, scalable, durable, and fault-tolerant publish-subscribe messaging system. Apache Kafka is an open-source stream-processing project. It provides a unified, high-throughput, and is a low-latency platform for handling real-time data streams. It provides a distributed storage layer, which supports massively scalable pub/sub message queues. Kafka Connect supports data import and export by connecting to external systems. Kafka Streams provides Java APIs for stream processing. Kafka works in combination with Apache Spark, Apache Cassandra, Apache HBase, Apache Spark, and more for real-time stream processing.

Apache Kafka was originally developed by LinkedIn, and was subsequently open sourced in early 2011. In November 2014, several engineers who worked on Kafka at LinkedIn created a new company named Confluent with a focus on Kafka. Please use this URL `https://www.confluent.io/` to learn more about the Confluent platform.

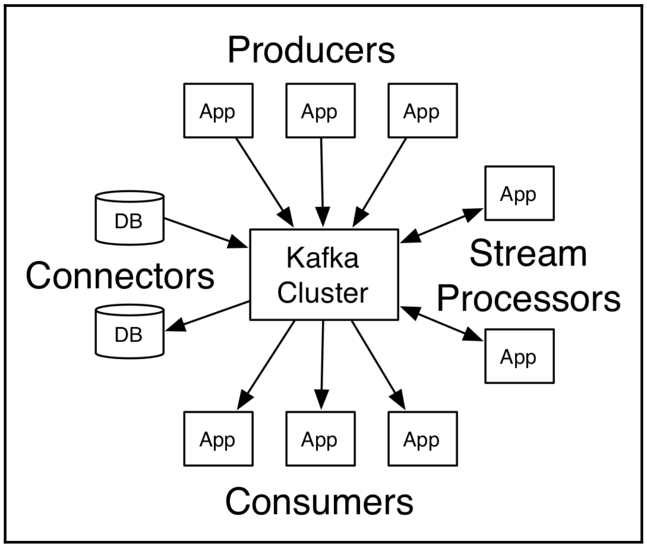
Kafka features

The following are features of Kafka:

- **Kafka is scalable**: Kafka Cluster consists of more than one physical server, which helps to distribute the data load. It is easily scalable in the event that additional throughputs are required as additional servers can be added to maintain the SLA.
- **Kafka is durable**: During stream processing, Kafka persists messages on the persistent storage. This storage can be server local disks or Hadoop Cluster. In the event of message processing failure, the message can be accessed from the disk and replayed to process the message again. By default, the message is stored for seven days; this can be configured further.
- **Kafka is reliable**: Kafka provides message reliability with the help of a feature called **data replication**. Each message is replicated at least three times (this is configurable) so that in the event of data loss, the copy of the message can be used for processing.
- **Kafka supports high performance throughput**: Due to its unique architecture, partitioning, message storage, and horizontal scalability, Kafka helps to process terabytes of data per second.

Kafka architecture

The following image shows Kafka's architecture:



Kafka architecture components

Let's take a look at each component in detail:

- **Producers**: Producers publish messages to a specific Kafka topic. Producers may attach a key to each message record. By default, producers publish messages to topic partitions in round robin fashion. Sometimes, producers can be configured to write messages to a particular topic partition based on the hash value of message key.
- **Topic**: All messages are stored in a topic. A topic is a category or feed name to which records are published. A topic can be compared to a table in a relational database. Multiple consumers can be subscribed to a single topic to consume message records.

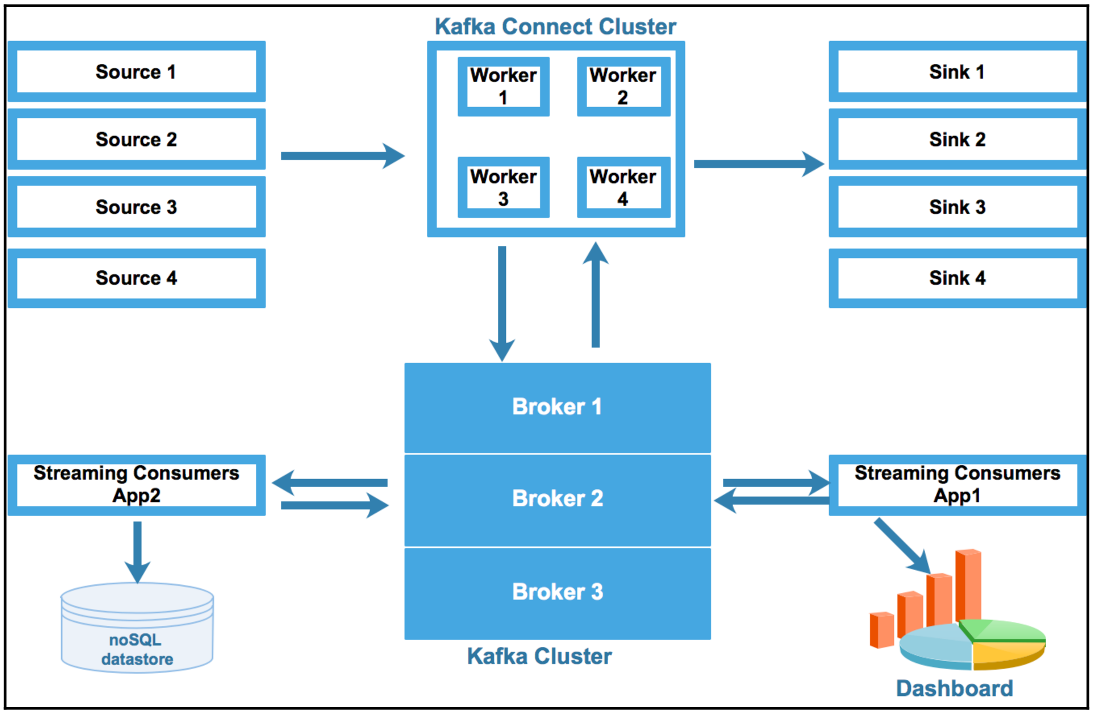
- **Partition**: A topic is divided into multiple partitions. Kafka offers topic parallelism by dividing topic into partitions and by placing each partition on a separate broker (server) of a Kafka Cluster. Each partition has a separate partition log on a disk where messages are stored. Each partition contains an ordered, immutable sequence of messages. Each message is assigned unique sequence numbers called **offset**. Consumers can read messages from any point from a partition—from the beginning or from any offset.
- **Consumers**: A consumer subscribes to a topic and consumes the messages. In order to increase the scalability, consumers of the same application can be grouped into a consumer group where each consumer can read messages from a unique partition.
- **Brokers**: Kafka is divided into multiple servers called **brokers**. All the brokers combined are called **Kafka Cluster**. Kafka brokers handle message writes from producers and message reads from consumers. Kafka Brokers stores all the messages coming from producers. The default period is of seven days. This period (retention period) can be configured based on the requirements. The retention period directly impacts the local storage of Kafka brokers. It takes more storage if a higher retention period is configured. After the retention period is over, the message is automatically discarded.
- **Kafka Connect**: According to Kafka documentation, Kafka Connect allows the building and running of reusable producers or consumers that connect Kafka topics to existing applications or data systems. For example, a connector to a relational database might capture every change to a table.
- **Kafka Streams**: Stream APIs allow an application to act as a stream processor, consuming an input stream from one or more topics and producing an output stream to one or more output topics, effectively transforming the input streams to output streams.

Kafka Connect deep dive

Kafka Connect is a part of the Confluent platform. It is integrated with Kafka. Using Kafka Connect, it's very easy to build data pipelines from multiple sources to multiple targets. **Source Connectors** import data from another system (for example, from a relational database into Kafka) and **Sink Connectors** export data (for example, the contents of a Kafka topic to an HDFS file).

Kafka Connect architecture

The following image shows Kafka Connect's architecture:



The data flow can be explained as follows:

- Various sources are connected to **Kafka Connect Cluster**. **Kafka Connect Cluster** pulls data from the sources.
- **Kafka Connect Cluster** consists of a set of worker processes that are containers that execute connectors, and tasks automatically coordinate with each other to distribute work and provide scalability and fault tolerance.
- **Kafka Connect Cluster** pushes data to **Kafka Cluster**.
- **Kafka Cluster** persists the data on to the broker local disk or on Hadoop.

- Streams applications such as Storm, Spark Streaming, and Flink pull the data from **Kafka Cluster** for stream transformation, aggregation, join, and so on. These applications can send back the data to Kafka or persist it to external data stores such as HBase, Cassandra, MongoDB, HDFS, and so on.
- **Kafka Connect Cluster** pulls data from **Kafka Cluster** and pushes it Sinks.
- Users can extend the existing Kafka connectors or develop brand new connectors.

Kafka Connect workers standalone versus distributed mode

Users can run Kafka Connect in two ways: standalone mode or distributed mode.

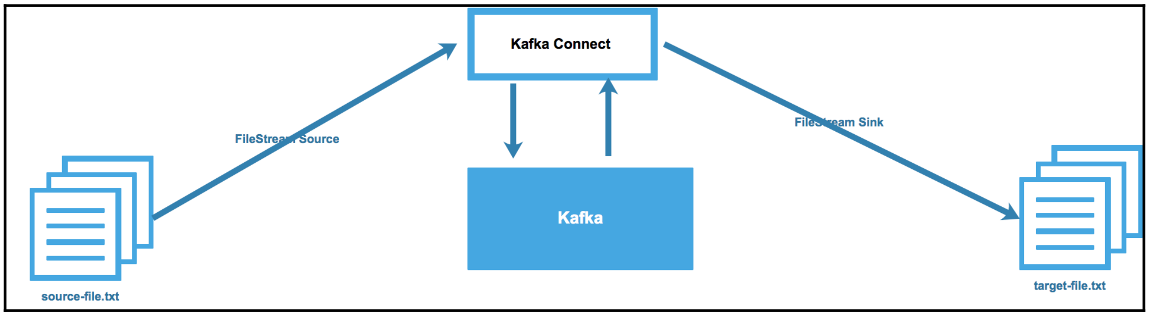
In standalone mode, a single process runs all the connectors. It is not fault tolerant. Since it uses only a single process, it is not scalable. Generally, it is useful for users for development and testing purposes.

In distributed mode, multiple workers run Kafka Connect. In this mode, Kafka Connect is scalable and fault tolerant, so it is used in production deployment.

Let's learn more about Kafka and Kafka Connect (standalone mode). In this example, we will do the following:

1. Install Kafka
2. Create a topic
3. Generate a few messages to verify the producer and consumer
4. Kafka Connect-File-source and file-sink
5. Kafka Connect-JDBC -Source

The following image shows a use case using **Kafka Connect**:



Let's see how Kafka and Kafka Connect works by running a few examples. For more details, use the following link for Kafka Confluent's documentation: `https://docs.confluent.io/current/`.

Install Kafka

Let's perform the following steps to install Kafka:

1. Download Confluent from `https://www.confluent.io/download/`
2. Click on **Confluent Open Source**
3. Download the file `confluent-oss-4.0.0-2.11.tar.gz` from `tar.gz` and perform the following:

   ```
   tar xvf confluent-oss-4.0.0-2.11.tar.gz
   cd /opt/confluent-4.0.0/etc/kafka
   vi server.properties
   ```

4. Uncomment `listeners=PLAINTEXT://:9092`
5. Start Confluent:

   ```
   $ ./bin/confluent start schema-registry
   ```

6. Start `zookeeper`:

   ```
   zookeeper is [UP]
   ```

7. Start `kafka`:

   ```
   kafka is [UP]
   ```

8. Start `schema-registry`:

   ```
   schema-registry is [UP]
   A4774045:confluent-4.0.0 m046277$
   ```

Create topics

Perform the following steps to create topics:

1. List the existing topics
2. Open another terminal and enter the following command:

```
/opt/confluent-4.0.0
bin/kafka-topics --list --zookeeper localhost:2181
_schemas
```

3. Create a topic:

```
bin/kafka-topics --create --zookeeper localhost:2181 --replication-
factor 1 --partitions 3 --topic my-first-topic

Created topic "my-first-topic"
```

4. Double check the newly created topic:

```
bin/kafka-topics --list --zookeeper localhost:2181
_schemas
my-first-topic
```

Generate messages to verify the producer and consumer

Perform the following steps to generate messages to verify the producer and consumer:

1. Send messages to Kafka `my-first-topic`:

```
bin/kafka-console-producer --broker-list localhost:9092 --topic my-
first-topic
test1
test2
test3
```

2. Start consumer to consume messages
3. Open another terminal and enter the following command:

```
$ bin/kafka-console-consumer --bootstrap-server localhost:9092 --
topic my-first-topic --from-beginning
test3
test2
test1
```

4. Go to the producer terminal and enter another message:

   ```
   test4
   ```

5. Verify the consumer terminal to check whether you can see the message `test4`

Kafka Connect using file Source and Sink

Let's take a look at how to create topics using file Source and Sink, with the help of the following:

```
cd /opt/confluent-4.0.0/etc/kafka
vi connect-file-test-source.properties
name=local-file-source
connector.class=FileStreamSource
tasks.max=1
file=/opt/kafka_2.10-0.10.2.1/source-file.txt
topic=my-first-topic
vi connect-file-test-sink.properties
name=local-file-sink
connector.class=FileStreamSink
tasks.max=1
file=/opt/kafka_2.10-0.10.2.1/target-file.txt
topics=my-first-topic
```

Perform the following steps:

1. Start the Source Connector and Sink Connector:

   ```
   cd /opt/confluent-4.0.0
   $ ./bin/connect-standalone config/connect-standalone.properties
   config/connect-file-test-source.properties config/connect-file-
   test-sink.properties

   echo 'test-kafka-connect-1' >> source-file.txt
   echo 'test-kafka-connect-2' >> source-file.txt
   echo 'test-kafka-connect-3' >> source-file.txt
   echo 'test-kafka-connect-4' >> source-file.txt
   ```

2. Double check whether the Kafka topic has received the messages:

   ```
   $ ./bin/kafka-console-consumer.sh --zookeeper localhost:2181 --
   from-beginning --topic my-first-topic

   test3
   test1
   test4
   ```

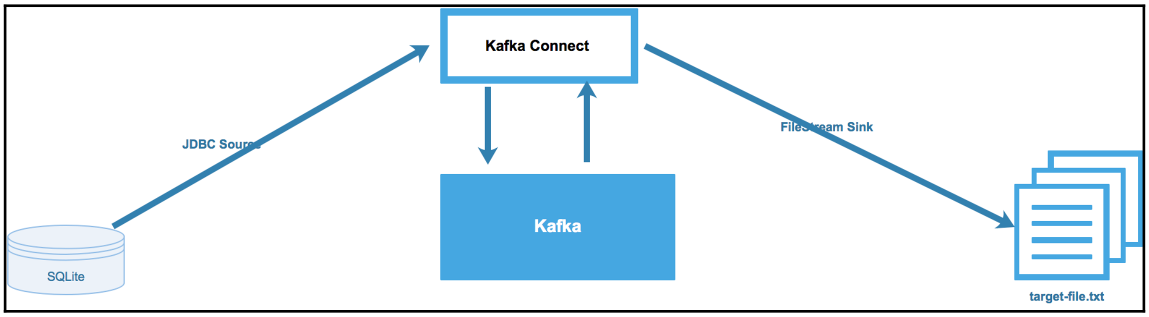
```
{"schema":{"type":"string","optional":false},"payload":"test-kafka-
connect-1"}
{"schema":{"type":"string","optional":false},"payload":"test-kafka-
connect-2"}
{"schema":{"type":"string","optional":false},"payload":"test-kafka-
connect-3"}
{"schema":{"type":"string","optional":false},"payload":"test-kafka-
connect-4"}

test2
```

3. Verify `target-file.txt`:

```
$ cat target-file.txt

{"schema":{"type":"string","optional":false},"payload":"test-kafka-
connect-1"}
{"schema":{"type":"string","optional":false},"payload":"test-kafka-
connect-2"}
{"schema":{"type":"string","optional":false},"payload":"test-kafka-
connect-3"}
{"schema":{"type":"string","optional":false},"payload":"test-kafka-
connect-4"}
```

Kafka Connect using JDBC and file Sink Connectors

The following image shows how we can push all records from the database table to a text file:



Let's implement the preceding example using Kafka Connect:

1. Install SQLite:

```
$ sqlite3 firstdb.db

SQLite version 3.16.0 2016-11-04 19:09:39
Enter ".help" for usage hints.

sqlite>
sqlite> CREATE TABLE customer(cust_id INTEGER PRIMARY KEY
AUTOINCREMENT NOT NULL, cust_name VARCHAR(255));
sqlite> INSERT INTO customer(cust_id,cust_name) VALUES(1,'Jon');
sqlite> INSERT INTO customer(cust_id,cust_name) VALUES(2,'Harry');
sqlite> INSERT INTO customer(cust_id,cust_name) VALUES(3,'James');
sqlite> select * from customer;

1|Jon
2|Harry
3|James
```

2. Configure the JDBC Source Connector:

```
cd /opt/confluent-4.0.0
vi ./etc/kafka-connect-jdbc/source-quickstart-sqlite.properties
name=test-sqlite-jdbc-autoincrement
connector.class=io.confluent.connect.jdbc.JdbcSourceConnector
tasks.max=1
connection.url=jdbc:sqlite:firstdb.db
mode=incrementing
incrementing.column.name=cust_id
topic.prefix=test-sqlite-jdbc-
```

3. Configure the file Sink Connector:

```
cd /opt/confluent-4.0.0
vi etc/kafka/connect-file-sink.properties
name=local-file-sink
connector.class=FileStreamSink
tasks.max=1
file=/opt/confluent-4.0.0/test.sink.txt
topics=test-sqlite-jdbc-customer
```

4. Start Kafka Connect (`.jdbs` source and file Sink):

```
./bin/connect-standalone ./etc/schema-registry/connect-avro-
standalone.properties ./etc/kafka-connect-jdbc/source-quickstart-
sqlite.properties ./etc/kafka/connect-file-sink.properties
```

5. Verify the consumer:

   ```
   $ ./bin/kafka-avro-console-consumer --new-consumer --bootstrap-
   server localhost:9092 --topic test-sqlite-jdbc-customer --from-
   beginning
   ```

   The `--new-consumer` option is deprecated and will be removed in a future major release. The new consumer is used by default if the `--bootstrap-server` option is provided:

   ```
   {"cust_id":1,"cust_name":{"string":"Jon"}}
   {"cust_id":2,"cust_name":{"string":"Harry"}}
   {"cust_id":3,"cust_name":{"string":"James"}}
   ```

6. Verify the target file:

   ```
   tail -f /opt/confluent-4.0.0/test.sink.txt

   Struct{cust_id=1,cust_name=Jon}
   Struct{cust_id=2,cust_name=Harry}
   Struct{cust_id=3,cust_name=James}
   ```

7. Insert a few more records in the customer table:

   ```
   sqlite> INSERT INTO customer(cust_id,cust_name) VALUES(4,'Susan');
   sqlite> INSERT INTO customer(cust_id,cust_name) VALUES(5,'Lisa');
   ```

8. Verify the target file:

   ```
   tail -f /opt/confluent-4.0.0/test.sink.txt
   ```

You will see all customer records (`cust_id`) in the target file. Using the preceding example, you can customize and experiment with any other Sink.

The following table presents the available Kafka connectors on the Confluent platform (developed and fully supported by Confluent):

Connector Name	Source/Sink
JDBC	Source and Sink
HDFS	Sink
Elasticsearch	Sink
Amazon S3	Sink

For more information on other certified Connectors by Confluent, please use this URL: `https://www.confluent.io/product/connectors/`.

You must have observed that Kafka Connect is a configuration-based stream-processing framework. It means we have to configure only the Source and Sink Connector files. We don't need to write any code using low-level languages like Java or Scala. But, now, let's turn to one more popular real-time stream processing framework called **Apache Storm**. Let's understand some cool features of Apache Storm.

Apache Storm

Apache Storm is a free and open source distributed real-time stream processing framework. At the time of writing this book, the stable release version of Apache Storm is 1.0.5. The Storm framework is predominantly written in the Clojure programming language. Originally, it was created and developed by Nathan Marz and the team at Backtype. The project was later acquired by Twitter.

During one of his talks on the Storm framework, Nathan Marz talked about stream processing applications using any framework, such as Storm. These applications involved queues and worker threads. Some of the data source threads write messages to queues and other threads pick up these messages and write to target data stores. The main drawback here is that source threads and targets threads do not match the data load of each other and this results in data pileup. It also results in data loss and additional thread maintenance.

To avoid the preceding challenges, Nathan Marz came up with a great architecture that abstracts source threads and worker threads into Spouts and Bolts. These Spouts and Bolts are submitted to the Topology framework, which takes care of entire stream processing.

Features of Apache Storm

Apache Storm is distributed. In case of an increase in a stream's workload, multiple nodes can be added to the Storm Cluster to add more workers and more processing power to process.

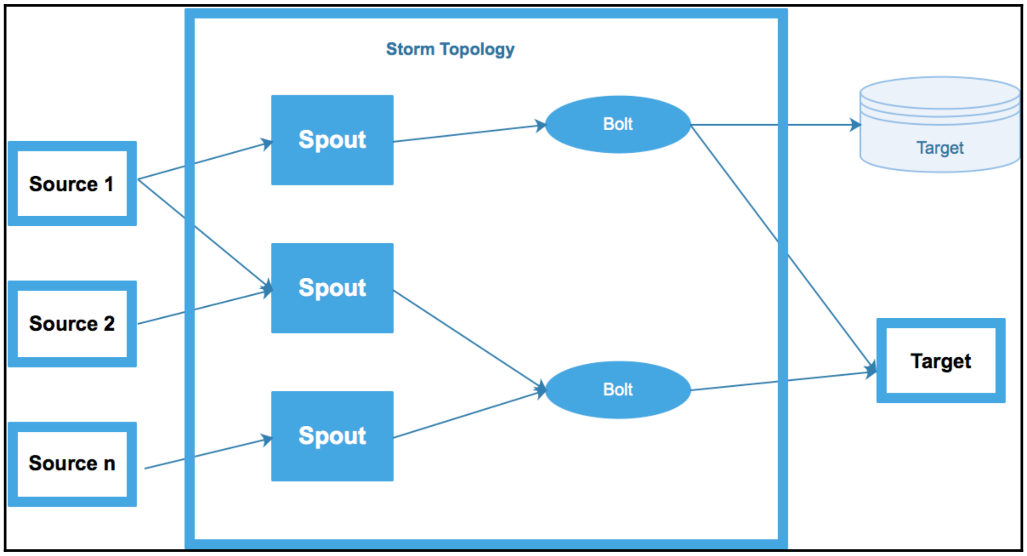
It is a truly real-time stream processing system and supports **low-latency**. The event can be reached from source to target in in milliseconds, seconds, or minutes depending on the use cases.

Storm framework supports **multiple programming languages**, but Java is the top preference. Storm is **fault-tolerant**. It continues to operate even though the failure of any node in the cluster. Storm is **reliable**. It supports at least once or exactly-once processing.

There is **no complexity** to using Storm framework. For more detail information, refer to the Storm documentation: `http://storm.apache.org/releases/1.0.4/index.html`.

Storm topology

The following image shows a typical **Storm Topology**:

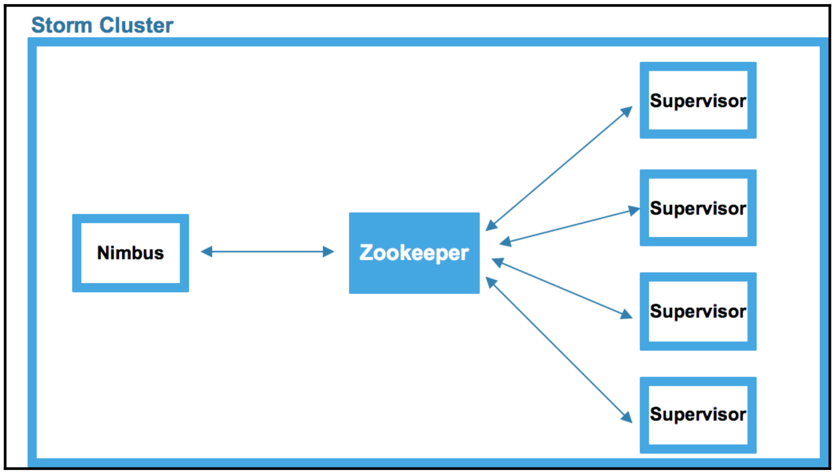


Storm topology components

The following sections explain all the components of a Storm topology:

- **Topology**: A topology is a **DAG** (**directed acyclic graph**) of spouts and bolts that are connected with stream groupings. A topology runs continuously untill it is killed.
- **Stream**: A stream is an unbounded sequence of tuples. A tuple can be of any data type. It supports all the Java data types.
- **Stream groupings**: Stream grouping decides which bolt receives a tuple from a spout. Basically, these are the strategies about how the stream will flow among different bolts. The following are the built-in stream groupings in Storm.
- **Shuffle grouping**: It is a default grouping strategy. Tuples are randomly distributed and each bolt gets an equal number of streams to process.
- **Field grouping**: In this strategy, the same value of a stream field will be sent to one bolt. For example, if all the tuples are grouped by `customer_id`, then all the tuples of the same `customer_id` will be sent one bolt task and all the tuples of another `customer_id` will be sent to another bolt task.
- **All grouping**: In all grouping, each tuple is sent to each bolt task. It can be used when two different functions have to be performed on the same set of data. In that case, the stream can be replicated and each function can be calculated on each copy of the data.
- **Direct grouping**: This is a special kind of grouping. Here, the developer can define the grouping logic within the component where tuple is emitted itself. The producer of the tuple decides which task of the consumer will receive this tuple.
- **Custom grouping**: The developer may decide to implement his/her own grouping strategy by implementing the `CustomGrouping` method.
- **Spout**: A spout connects to the data source and ingests streams into a Storm topology.
- **Bolt**: A spout emits a tuple to a bolt. A bolt is responsible for event transformation, joining events to other events, filtering, aggregation, and windowing. It emits the tuple to another bolt or persists it to a target. All processing in topologies is done in bolts. Bolts can do anything from filtering to functions, aggregations, joins, talking to databases, and more.

- **Storm Cluster**: The following image shows all the components of a **Storm Cluster**:



- **Storm Cluster nodes**: The three main nodes of a Storm Cluster are Nimbus, Supervisor, and Zookeeper. The following section explains all the components in detail.
- **Nimbus node**: In Storm, this is the master node of a Storm Cluster. It distributes code and launches the worker tasks across the cluster. Basically, it assigns tasks to each node in a cluster. It also monitors the status of each job submitted. In the case of any job failure, Nimbus reallocates the job to a different supervisor within a cluster. In the case of Nimbus being unavailable, the workers will still continue to function. However, without Nimbus, workers won't be reassigned to other machines when necessary. In the case of an unavailable node, the tasks assigned to that node will time-out and Nimbus will reassign those tasks to other machines. In the case of both Nimbus and Supervisor being unavailable, they need to be restarted like nothing happened and no worker processes will be affected.
- **Supervisor node**: In Storm, this is a slave node. It communicates with Nimbus through ZooKeeper. It starts and stops the worker processes within a supervisor itself. For example, if Supervisor finds that a particular worker process has died, then it immediately restarts that worker process. If Supervisor fails to restart the worker after trying few times, then it communicates this to Nimbus and Nimbus will restart that worker on a different Supervisor node.

- **Zookeeper node**: It acts as a coordinator between masters (Nimbus) and slaves (supervisors) within a Storm Cluster. In a production environment, it is typical to set up a Zookeeper cluster that has three instances (nodes) of Zookeeper.

Installing Storm on a single node cluster

The following are the steps to install Storm Cluster on a single machine:

1. Install `jdk`. Make sure you have installed 1.8:

   ```
   $ java -version
   ```

   You should see the following output:

   ```
   openjdk version "1.8.0_141"
   OpenJDK Runtime Environment (build 1.8.0_141-b16)
   OpenJDK 64-Bit Server VM (build 25.141-b16, mixed mod
   ```

2. Create a folder to download the `.tar` file of Storm:

   ```
   $ mkdir /opt/storm
   $ cd storm
   ```

3. Create a folder to persist Zookeeper and Storm data:

   ```
   $ mkdir /usr/local/zookeeper/data
   $ mkdir /usr/local/storm/data
   ```

4. Download Zookeeper and Storm:

   ```
   $ wget http://apache.osuosl.org/zookeeper/stable/zookeeper-3.4.10.
   tar.gz
   $ gunzip zookeeper-3.4.10.tar.gz
   $ tar -xvf zookeeper-3.4.10.tar
   $ wget http://mirrors.ibiblio.org/apache/storm/apache-storm-1.0.5/
   apache-storm-1.0.5.tar.gz
   $ gunzip apache-storm-1.0.5.tar.gz
   $ tar -xvf apache-storm-1.0.5.tar
   ```

5. Configure Zookeeper and set the following to Zookeeper (`zoo.cfg`):

   ```
   $ cd zookeeper-3.4.10
   $ vi con/zoo.cfg
   tickTime = 2000
   dataDir = /usr/local/zookeeper/data
   clientPort = 2181
   ```

6. Configure Storm as follows:

   ```
   $ cd /opt/ apache-storm-1.0.5
   $ vi conf/storm.yaml
   ```

7. Add the following:

   ```
   storm.zookeeper.servers:
    - "127.0.0.1"
    nimbus.host: "127.0.0.1"
    storm.local.dir: "/usr/local/storm/data"
    supervisor.slots.ports:
    - 6700
    - 6701
    - 6702
    - 6703
   ```

   (for additional workers, add more ports, such as 6704 and so on)

8. Start Zookeeper:

   ```
   $ cd /opt/zookeeper-3.4.10
   $ bin/zkServer.sh start &amp;amp;
   ```

9. Start Nimbus:

   ```
   $ cd /opt/ apache-storm-1.0.5
   $ bin/storm nimbus &amp;amp;
   ```

10. Start Supervisor:

    ```
    $ bin/storm supervisor &amp;amp;
    ```

11. Verify installation in the Storm UI:

    ```
    http://127.0.0.1:8080
    ```

Developing a real-time streaming pipeline with Storm

In this section, we will create the following three pipelines:

- Streaming pipeline with Kafka - Storm - MySQL
- Streaming pipeline with Kafka - Storm - HDFS - Hive

In this section, we will see how data streams flow from Kafka to Storm to MySQL table.

The whole pipeline will work as follows:

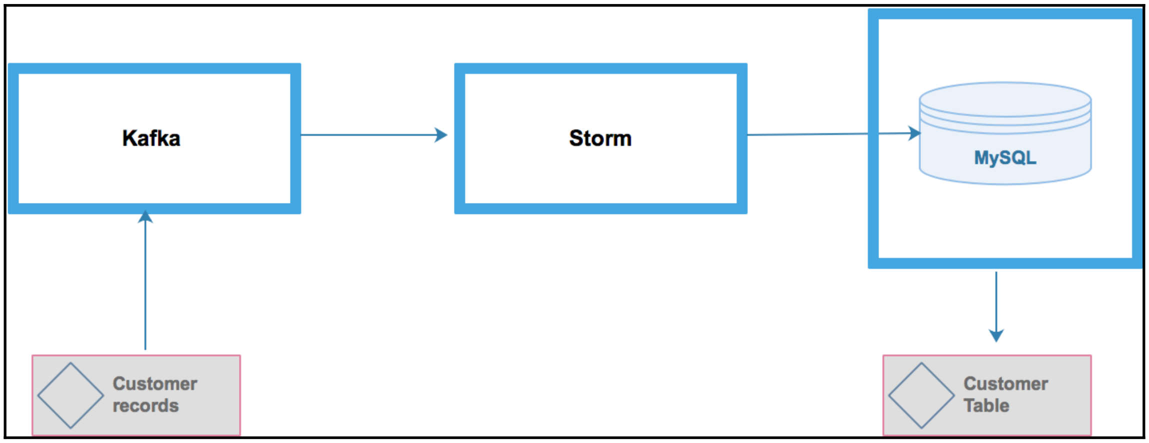
1. We will ingest customer records (`customer_firstname` and `customer_lastname`) in Kafka using the Kafka console-producer API.
2. After that, Storm will pull the messages from Kafka.
3. A connection to MySQL will be established.
4. Storm will use MySQL-Bolt to ingest records into MySQL table. MySQL will automatically generate `customer_id`.
5. The MySQL table data (`customer_id`, `customer_firstname`, and `customer_lastname`) will be accessed using SQL.

We will develop the following Java classes:

- `MysqlConnection.java`: This class will establish a connection with the local MySQL database.
- `MysqlPrepare.java`: This class will prepare the SQL statements to be inserted into the database.
- `MysqlBolt`: This class is a storm bolt framework to emit the tuple from Kafka to MySQL.
- `MySQLKafkaTopology`: This is a Storm Topology Framework that builds a workflow to bind spouts (Kafka) to Bolts (MySQL). Here, we are using a Local Storm Cluster.

Streaming a pipeline from Kafka to Storm to MySQL

The following image shows the components of the pipeline. In this pipeline, we will learn how the messages will flow from Kafka to Storm to MySQL in real-time:



The following is the complete Java code for `MysqlConnection.java`:

```
package com.StormMysql;
import java.sql.Connection;
import java.sql.DriverManager;
public class MysqlConnection {
private String server_name;
 private String database_name;
 private String user_name;
 private String password;
 private Connection connection;

public MysqlConnection(String server_name, String database_name, String
user_name, String password)
 {
 this.server_name=server_name;
 this.database_name=database_name;
 this.user_name=user_name;
 this.password=password;
 }

public Connection getConnection()
 {
 return connection;
 }
public boolean open()
 {
 boolean successful=true;
 try{
 Class.forName("com.mysql.jdbc.Driver");
 connection =
DriverManager.getConnection("jdbc:mysql://"+server_name+"/"+database_name+"
?"+"user="+user_name+"&amp;amp;password="+password);
 }catch(Exception ex)
 {
 successful=false;
 ex.printStackTrace();
 }
 return successful;
 }

public boolean close()
 {
 if(connection==null)
 {
 return false;
 }

boolean successful=true;
 try{
 connection.close();
 }catch(Exception ex)
 {
 successful=false;
 ex.printStackTrace();
 }

return successful;
 }
 }
```

The following is the complete code for `MySqlPrepare.java`:

```
package com.StormMysql;
import org.apache.storm.tuple.Tuple;
import java.sql.PreparedStatement;
public class MySqlPrepare {
 private MysqlConnection conn;

public MySqlPrepare(String server_name, String database_name, String
user_name, String password)
 {
 conn = new MysqlConnection(server_name, database_name, user_name,
password);
 conn.open();
 }

public void persist(Tuple tuple)
 {
 PreparedStatement statement=null;
 try{
 statement = conn.getConnection().prepareStatement("insert into customer
(cust_id,cust_firstname, cust_lastname) values (default, ?,?)");
 statement.setString(1, tuple.getString(0));

statement.executeUpdate();
 }catch(Exception ex)
 {
 ex.printStackTrace();
 }finally
{
 if(statement != null)
 {
 try{
 statement.close();
 }catch(Exception ex)
 {
 ex.printStackTrace();
 }
 }
 }
 }

public void close()
 {
 conn.close();
 }
 }
```

The following is the complete code for `MySqlBolt.java`:

```
package com.StormMysql;

import java.util.Map;

import org.apache.storm.topology.BasicOutputCollector;
 import org.apache.storm.topology.OutputFieldsDeclarer;
 import org.apache.storm.topology.base.BaseBasicBolt;
 import org.apache.storm.tuple.Fields;
 import org.apache.storm.tuple.Tuple;
 import org.apache.storm.tuple.Values;
 import org.apache.storm.task.TopologyContext;

import java.util.Map;

public class MySqlBolt extends BaseBasicBolt {

private static final long serialVersionUID = 1L;
 private MySqlPrepare mySqlPrepare;

@Override
 public void prepare(Map stormConf, TopologyContext context)
  {
 mySqlPrepare=new MySqlPrepare("localhost", "sales","root","");
  }

public void execute(Tuple input, BasicOutputCollector collector) {
 // TODO Auto-generated method stub
mySqlPrepare.persist(input);
 //System.out.println(input);
}
@Override
 public void cleanup() {
 mySqlPrepare.close();
  }
}
```

The following is the complete code for `KafkaMySQLTopology.java`:

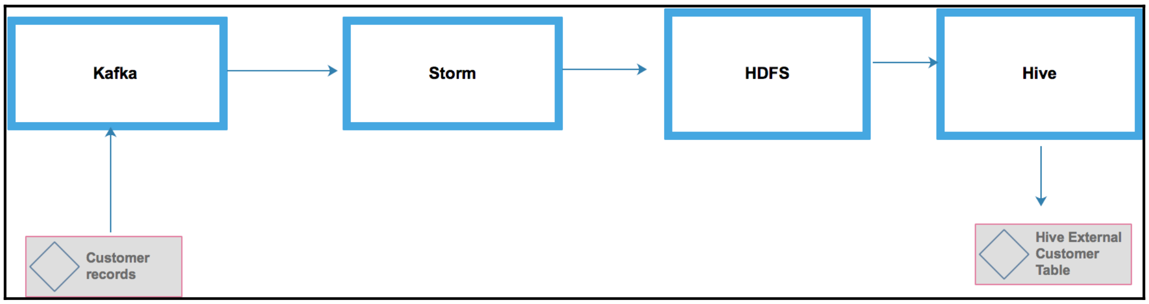
```
package com.StormMysql;

import org.apache.storm.Config;
 import org.apache.storm.spout.SchemeAsMultiScheme;
 import org.apache.storm.topology.TopologyBuilder;
 import org.apache.storm.kafka.*;
 import org.apache.storm.LocalCluster;
 import org.apache.storm.generated.AlreadyAliveException;
 import org.apache.storm.generated.InvalidTopologyException;

public class KafkaMySQLTopology
 {
 public static void main( String[] args ) throws AlreadyAliveException,
InvalidTopologyException
 {
 ZkHosts zkHosts=new ZkHosts("localhost:2181");

String topic="mysql-topic";
 String consumer_group_id="id7";

SpoutConfig kafkaConfig=new SpoutConfig(zkHosts, topic, "",
consumer_group_id);

kafkaConfig.scheme=new SchemeAsMultiScheme(new StringScheme());

KafkaSpout kafkaSpout=new KafkaSpout(kafkaConfig);

TopologyBuilder builder=new TopologyBuilder();
 builder.setSpout("KafkaSpout", kafkaSpout);
 builder.setBolt("MySqlBolt", new
MySqlBolt()).globalGrouping("KafkaSpout");

LocalCluster cluster=new LocalCluster();

Config config=new Config();

cluster.submitTopology("KafkaMySQLTopology", config,
builder.createTopology());

try{
 Thread.sleep(10000);
 }catch(InterruptedException ex)
 {
 ex.printStackTrace();
 }

// cluster.killTopology("KafkaMySQLTopology");
 // cluster.shutdown();
}
 }
```

Use the `pom.xml` file to build your project in IDE.

Streaming a pipeline with Kafka to Storm to HDFS

In this section, we will see how the data streams will flow from Kafka to Storm to HDFS and access them with a Hive external table.

The following image shows the components of the pipeline. In this pipeline, we will learn how the messages will flow from Kafka to Storm to HDFS in real-time:



The whole pipeline will work as follows:

1. We will ingest customer records (`customer_id`, `customer_firstname`, and `customer_lastname`) in Kafka using the Kafka console-producer API
2. After that, Storm will pull the messages from Kafka
3. A Connection to HDFS will be established
4. Storm will use HDFS-Bolt to ingest records into HDFS
5. Hive external table will be created to store (`customer_id`, `customer_firstname`, and `customer_lastname`)
6. The Hive table data (`customer_id`, `customer_firstname`, and `customer_lastname`) will be accessed using SQL

We will develop the following Java classes:

`KafkaTopology.java`: This is a Storm Topology framework that builds a workflow to bind spouts (Kafka) to Bolts (HDFS). Here we are using a Local Storm cluster.

In the previous example pipeline, multiple separate classes for data streams parsing and transformations can be developed to handle Kafka producers and consumers.

The following is the complete Java code for `KafkaToplogy.java`:

```
package com.stormhdfs;

import org.apache.storm.Config;
 import org.apache.storm.LocalCluster;
 import org.apache.storm.generated.AlreadyAliveException;
 import org.apache.storm.generated.InvalidTopologyException;
 import org.apache.storm.hdfs.bolt.HdfsBolt;
 import org.apache.storm.hdfs.bolt.format.DefaultFileNameFormat;
 import org.apache.storm.hdfs.bolt.format.DelimitedRecordFormat;
 import org.apache.storm.hdfs.bolt.format.RecordFormat;
 import org.apache.storm.hdfs.bolt.rotation.FileRotationPolicy;
 import org.apache.storm.hdfs.bolt.rotation.FileSizeRotationPolicy;
 import org.apache.storm.hdfs.bolt.sync.CountSyncPolicy;
 import org.apache.storm.hdfs.bolt.sync.SyncPolicy;
 import org.apache.storm.kafka.KafkaSpout;
 import org.apache.storm.kafka.SpoutConfig;
 import org.apache.storm.kafka.StringScheme;
 import org.apache.storm.kafka.ZkHosts;
 import org.apache.storm.spout.SchemeAsMultiScheme;
 import org.apache.storm.topology.TopologyBuilder;

public class KafkaTopology {
 public static void main(String[] args) throws
AlreadyAliveException, InvalidTopologyException {

// zookeeper hosts for the Kafka cluster
ZkHosts zkHosts = new ZkHosts("localhost:2181");

// Create the KafkaSpout configuartion
 // Second argument is the topic name
 // Third argument is the zookeeper root for Kafka
 // Fourth argument is consumer group id
SpoutConfig kafkaConfig = new SpoutConfig(zkHosts,
 "data-pipleline-topic", "", "id7");

// Specify that the kafka messages are String
kafkaConfig.scheme = new SchemeAsMultiScheme(new StringScheme());

// We want to consume all the first messages in the topic everytime
 // we run the topology to help in debugging. In production, this
 // property should be false
kafkaConfig.startOffsetTime = kafka.api.OffsetRequest.EarliestTime();

RecordFormat format = new DelimitedRecordFormat().withFieldDelimiter("|");
 SyncPolicy syncPolicy = new CountSyncPolicy(1000);

FileRotationPolicy rotationPolicy = new
```

```
FileSizeRotationPolicy(1.0f,FileSizeRotationPolicy.Units.MB);

DefaultFileNameFormat fileNameFormat = new DefaultFileNameFormat();

fileNameFormat.withPath("/user/storm-data");

fileNameFormat.withPrefix("records-");

fileNameFormat.withExtension(".txt");

HdfsBolt bolt =
 new HdfsBolt().withFsUrl("hdfs://127.0.0.1:8020")
 .withFileNameFormat(fileNameFormat)
 .withRecordFormat(format)
 .withRotationPolicy(rotationPolicy)
 .withSyncPolicy(syncPolicy);

// Now we create the topology
TopologyBuilder builder = new TopologyBuilder();

// set the kafka spout class
builder.setSpout("KafkaSpout", new KafkaSpout(kafkaConfig), 1);

// configure the bolts
 // builder.setBolt("SentenceBolt", new SentenceBolt(),
1).globalGrouping("KafkaSpout");
 // builder.setBolt("PrinterBolt", new PrinterBolt(),
1).globalGrouping("SentenceBolt");
builder.setBolt("HDFS-Bolt", bolt ).globalGrouping("KafkaSpout");

// create an instance of LocalCluster class for executing topology in local
mode.
LocalCluster cluster = new LocalCluster();
 Config conf = new Config();

// Submit topology for execution
cluster.submitTopology("KafkaTopology", conf, builder.createTopology());

try {
 // Wait for some time before exiting
System.out.println("Waiting to consume from kafka");
 Thread.sleep(10000);
 } catch (Exception exception) {
 System.out.println("Thread interrupted exception : " + exception);
 }

// kill the KafkaTopology
 //cluster.killTopology("KafkaTopology");
```

```
// shut down the storm test cluster
 // cluster.shutdown();
}
 }
```

The Hive table for the same is as follows:

```
CREATE EXTERNAL TABLE IF NOT EXISTS customer (
customer_id INT,
customer_firstname String,
customer_lastname String))
COMMENT 'customer table'
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '|'
STORED AS TEXTFILE
location '/user/storm-data';
$ hive > select * from customer;
```

Other popular real-time data streaming frameworks

Apart from Apache Storm, there are quite a few other open source real-time data streaming frameworks. In this section, I will discuss in brief only open source non-commercial frameworks. But, at the end of this section, I will provide a few URLs for a few commercial vendor products that offer some very interesting features.

Kafka Streams API

Kafka Streams is a library for building streaming applications. Kafka Streams is a client library for building applications and microservices, where the input and output data are stored in Kafka Clusters. The Kafka Streams API transforms and enriches the data.

The following are the important features of the Kafka Streams API:

- It is part of the open source Apache Kafka project.
- It supports per record streams processing with a very low latency (milliseconds). There is no micro- batching concept in the Kafka Streams API. Every record that comes into the stream is processed on its own.

- It supports stateless processing (filtering and mapping), stateful processing (joins and aggregations), and windowing operations (for example, counting the last minute, last 5 minutes, last 30 minutes, or last day's worth of data, and so on).
- To run the Kafka Streams API, there is no need to build a separate cluster that has multiple machines. Developers can use the Kafka Streams API in their Java applications or microservices to process real-time data.
- The Kafka Streams API is highly scalable and fault-tolerant.
- The Kafka Streams API is completely deployment agnostic. It can be deployed on a bare metal machine, VMs, Kubernetes containers, and on Cloud. There are no restrictions at all. Stream APIs are never deployed on Kafka Brokers. It is a separate application just like any other Java application, which is deployed outside of Kafka brokers.
- It uses the Kafka security model.
- It supports exactly-once semantics since version 0.11.0.

Let's review the earlier image again to find out the exact place of the Kafka Streams API in the overall Kafka architecture.

Here are a few useful URLs to understand Kafka Streams in detail:

- `https://kafka.apache.org/documentation/`
- `https://www.confluent.io/blog/`
- `https://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/`
- `https://docs.confluent.io/current/streams/index.html`

Spark Streaming

Please note that we will discuss Spark in `Chapter 7`, *Large-Scale Data Processing Frameworks,* which is fully dedicated to Spark. However, in this section, I will discuss some important features of Spark Streaming. For better understanding, readers are advised to study `Chapter 7`, *Large-Scale Data Processing Frameworks* first and come back to read this section further to understand more about Spark Streaming.

It is a general practice to use Hadoop MapReduce for batch processing and Apache Storm for real-time stream processing.

The use of these two different programming models causes an increase in code size, number of bugs to fix, and development effort; it also introduces a learning curve and causes other issues. Spark Streaming helps fix these issues and provides a scalable, efficient, resilient, and integrated (with batch processing) system.

The strength of Spark Streaming lies in its ability to combine with batch processing. It's possible to create a RDD using normal Spark programming and join it with a Spark stream. Moreover, the code base is similar and allows easy migration if required—and there is zero to no learning curve from Spark.

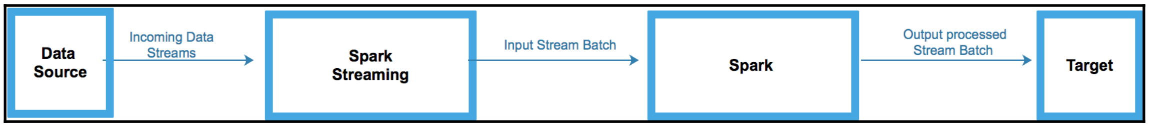
Spark Streaming is an extension of the core Spark API. It extends Spark for doing real-time stream processing. Spark Streaming has the following features:

- It's scalable—it scales on hundreds of nodes
- It provides high-throughput and achieves second level latency
- It's fault-tolerant and it efficiently receives from the failures
- It integrates with batch and interactive data processing

Spark Streaming processes data streams application as a series of very small, deterministic batch jobs.

Spark Streaming provides an API in Scala, Java, and Python. Spark Streaming divides live stream of data into multiple batches based on time. The time can range from one second to a few minutes/hours. In general, batches are divided into a few seconds. Spark treats each batch as a RDD and process each based on RDD operations (map, filter, join flatmap, distinct, reduceByKey, and so on). Lastly, the processed results of RDDs are returned in batches.

The following image depicts the Spark Streaming data flow:



Here are few useful URLs for understanding Spark Streaming in detail:

- `https://databricks.com/blog`
- `https://databricks.com/blog/category/engineering/streaming`
- `https://spark.apache.org/streaming/`

Apache Flink

Apache Flink's documentation describes Flink in the following way: Flink is an open-source framework for distributed stream processing.

Flink provides accurate results and supports out-of-order or late-arriving datasets. It is stateful and fault-tolerant and can seamlessly recover from failures while maintaining an exactly-once application state. It performs at a large scale, running on thousands of nodes with very good throughput and latency characteristics.

The following are the features of Apache Flink:

- Flink guarantees exactly-once semantics for stateful computations
- Flink supports stream processing and windowing with event time semantics
- Flink supports flexible windowing based on time, count, or sessions, in addition to data-driven windows
- Flink is capable of high throughput and low latency
- Flink's savepoints provide a state versioning mechanism, making it possible to update applications or reprocess historic data with no lost state and minimal downtime
- Flink is designed to run on large-scale clusters with many thousands of nodes, and, in addition to a standalone cluster mode, Flink provides support for YARN and Mesos

Flink's core is a distributed streaming dataflow engine. It supports processing one stream at a time rather than processing an entire batch of streams at a time.

Flink supports the following libraries:

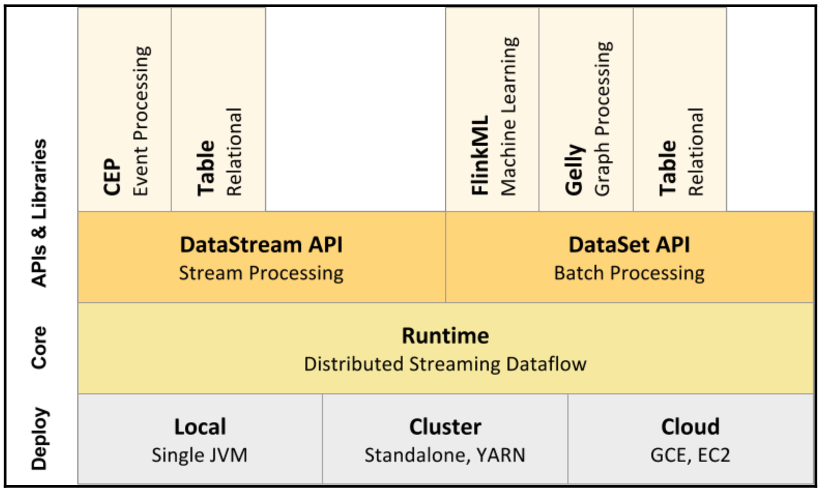
- CEP
- Machine learning
- Graph processing
- Apache Storm compatibility

Flink supports the following APIs:

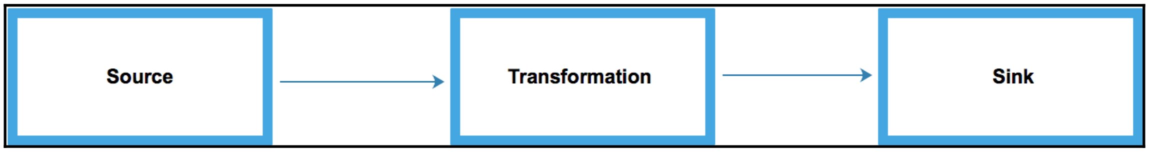
- **DataStream API**: This API helps all the streams, transformations, that is, filtering, aggregations, counting, and windowing
- **DataSet API**: This API helps all the batch data transformations, that is, join, group, map, and filter

- **Table API**: Supports SQL over relational data streams
- **Streaming SQL**: Supports SQL over batch and streaming tables

The following image describes the Flink programming model:



The following image describes the Flink architecture:



The following are the components of the Flink programming model:

- **Source**: A data source where data is collected and sent to the Flink engine
- **Transformation**: In this component the whole transformation takes place
- **Sink**: A target where processed streams are sent

Here are a few useful URLs to understand Spark Streaming in detail:

- `https://ci.apache.org/projects/flink/flink-docs-release-1.4/`
- `https://www.youtube.com/watch?v=ACS6OM1-xgEamp;amp;feature=youtu.be`

In the following sections, we will take a look at a comparison of various stream frameworks.

Apache Flink versus Spark

The main focus of Spark Streaming is stream-batching operation, called **micro-batching**. This programming model suits many use cases, but not all use cases require real-time stream processing with sub-second latency. For example, a use case such as credit card fraud prevention requires millisecond latency. Hence, the micro-batching programming model is not suited there. (But, the latest version of Spark, 2.4, supports millisecond data latency).

Apache Flink supports millisecond latency and is suited for use cases such as fraud detection and like.

Apache Spark versus Storm

Spark uses micro-batches to process events while Storm processes events one by one. It means that Spark has a latency of seconds while Storm provides a millisecond of latency. Spark Streaming provides a high-level abstraction called a **Discretized Stream** or **DStream**, which represents a continuous sequence of RDDs. (But, the latest version of Spark, 2.4 supports millisecond data latency.) The latest Spark version supports DataFrames.

Almost the same code (API) can be used for Spark Streaming and Spark batch jobs. That helps to reuse most of the code base for both programming models. Also, Spark supports Machine learning and the Graph API. So, again, the same codebase can be used for those use cases as well.

Summary

In this chapter, we started with a detailed understanding of real-time stream processing concepts, including data stream, batch vs. real-time processing, CEP, low latency, continuous availability, horizontal scalability, storage, and so on. Later, we learned about Apache Kafka, which is a very important component of modern real-time stream data pipelines. The main features of Kafka are scalability, durability, reliability, and high throughput.

We also learned about Kafka Connect; its architecture, data flow, sources, and connectors. We studied case studies to design a data pipeline with Kafka Connect using file source, file Sink, JDBC source, and file Sink Connectors.

In the later sections, we learned about various open source real-time stream-processing frameworks, such as the Apache Storm framework. We have seen a few practical examples, as well. Apache Storm is distributed and supports low-latency and multiple programming languages. Storm is fault-tolerant and reliable. It supports at least once or exactly-once processing.

Spark Streaming helps to fix these issues and provides a scalable, efficient, resilient, and integrated (with batch processing) system. The strength of Spark Streaming lies in its ability to combine with batch processing. Spark Streaming is scalable, and provides high-throughput. It supports micro-batching for second level latency, is fault-tolerant, and integrates with batch and interactive data processing.

Apache Flink guarantees exactly-once semantics, supports event time semantics, high throughput, and low latency. It is designed to run on large-scale clusters.

7
Large-Scale Data Processing Frameworks

As the volume and complexity of data sources are increasing, deriving value out of data is also becoming increasingly difficult. Ever since Hadoop was made, it has built a massively scalable filesystem, HDFS. It has adopted the MapReduce concepts from functional programming to approach the large-scale data processing challenges. As technology is constantly evolving to overcome the challenges posed by data mining, enterprises are also finding ways to embrace these changes to stay ahead.

In this chapter, we will focus on these data processing solutions:

- MapReduce
- Apache Spark
- Spark SQL
- Spark Streaming

MapReduce

MapReduce is a concept that is borrowed from functional programming. The data processing is broken down into a map phase, where data preparation occurs, and a reduce phase, where the actual results are computed. The reason MapReduce has played an important role is the massive parallelism we can achieve as the data is sharded into multiple distributed servers. Without this advantage, MapReduce cannot really perform well.

Let's take up a simple example to understand how MapReduce works in functional programming:

- The input data is processed using a mapper function of our choice
- The output from the mapper function should be in a state that is consumable by the reduce function
- The output from the mapper function is fed to the reduce function to generate the necessary results

Let's understand these steps using a simple program. This program uses the following text (randomly created) as input:

```
Bangalore,Onion,60
Bangalore,Chilli,10
Bangalore,Pizza,120
Bangalore,Burger,80
NewDelhi,Onion,80
NewDelhi,Chilli,30
NewDelhi,Pizza,150
NewDelhi,Burger,180
Kolkata,Onion,90
Kolkata,Chilli,20
Kolkata,Pizza,120
Kolkata,Burger,160
```

The input consists of data with the following fields: **City Name**, **Product Name**, and **Item Price** on that day.

We want to write a program that will show the total cost of all products in a given city. This can be done in many ways. But let's try to approach this using MapReduce and see how it works.

The mapper program is like this:

```
#!/usr/bin/env perl -wl

use strict;
use warnings;

while(<STDIN>) {
    chomp;
    my ($city, $product, $cost) = split(',');
    print "$city $cost";
}
```

The reduce program is:

```
#!/usr/bin/perl

use strict;
use warnings;

my %reduce;

while(<STDIN>) {
    chomp;
    my ($city, $cost) = split(/\s+/);
    $reduce{$city} = 0 if not defined $reduce{$city};
    $reduce{$city} += $cost;
}

print "-" x 24;
printf("%-10s : %s\n", "City", "Total Cost");
print "-" x 24;

foreach my $city (sort keys %reduce) {
    printf("%-10s : %d\n", $city, $reduce{$city});
}
```

We create a data pipeline using the UNIX terminal like this:

```
[user@node-1 ~]$ cat input.txt | perl map.pl | perl reduce.pl
------------------------
City : Total Cost
------------------------
Bangalore : 270
Kolkata : 390
NewDelhi : 440
```

As we can see, the result is as expected. This is a very simple case of MapReduce. Let's try to see what is happening:

- Each input line is processed by the `map.pl` program and prints the city and price
- The output from the `map.pl` program is fed to `reduce.pl`, which performs a `SUM()` operation for all records and categorizes them per city

Let's shuffle the `input.txt` and see if we get the desired results.

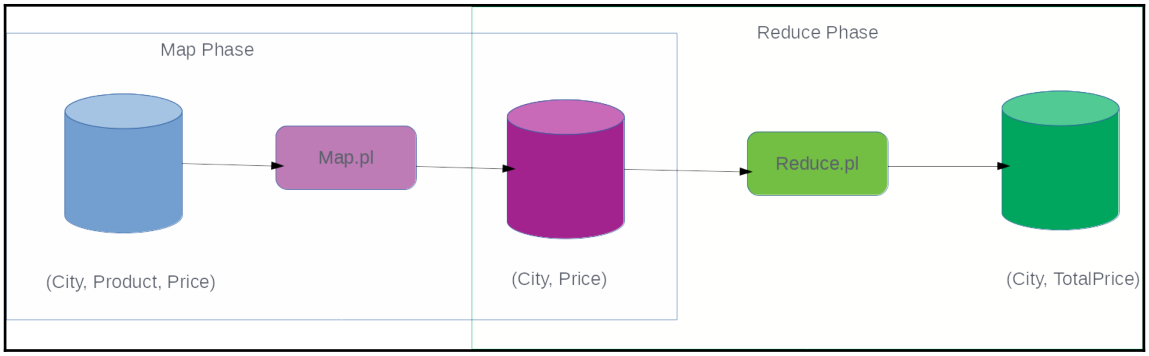
Here is the modified `input.txt`:

```
Bangalore,Onion,60
NewDelhi,Onion,80
Bangalore,Pizza,120
Bangalore,Burger,80
Kolkata,Onion,90
Kolkata,Pizza,120
Kolkata,Chilli,20
NewDelhi,Chilli,30
NewDelhi,Burger,180
Kolkata,Burger,160
NewDelhi,Pizza,150
Bangalore,Chilli,10
```

And the output from the MapReduce operation is:

```
[user@node-1 ~]$ cat input-shuffled.txt | perl map.pl | perl reduce.pl
------------------------
City : Total Cost
------------------------
Bangalore : 270
Kolkata : 390
NewDelhi : 440
```

There is no difference because both the map and reduce operations are being performed independently in one go. There is no data parallelism here. The entire process can be visualized in this diagram:



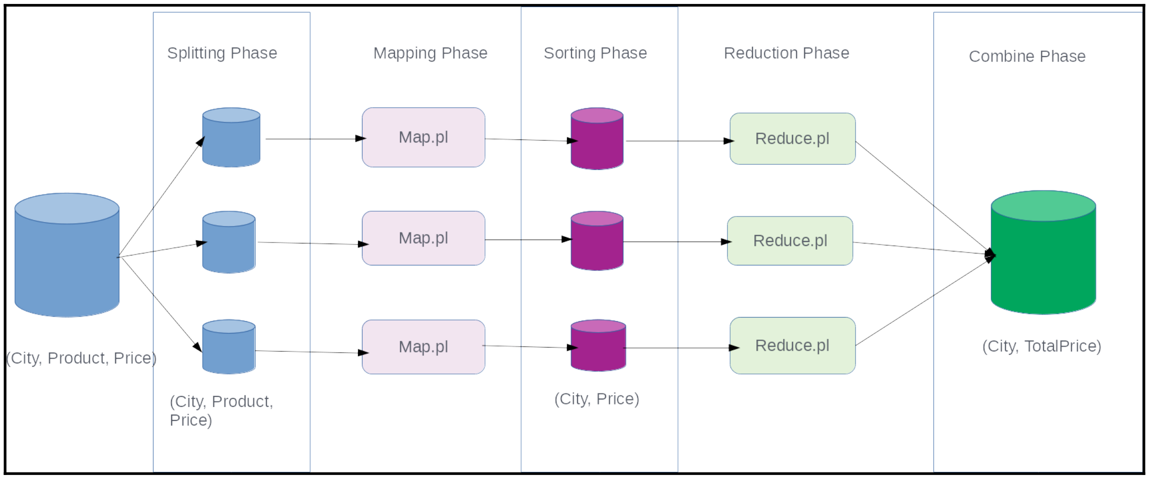
As we can see, there is one copy of the input data after the **Map Phase**, and the final output after **Reduce Phase** is what we are interested in.

Running a single-threaded process is useful and is needed when we don't have to deal with massive amounts of data. When the input sizes are unbounded and cannot be fit into a single server, we need to start thinking of distributed/parallel algorithms to attack the problem at hand.

Hadoop MapReduce

Apache MapReduce is a framework that makes it easier for us to run MapReduce operations on very large, distributed datasets. One of the advantages of Hadoop is a distributed file system that is rack-aware and scalable. The Hadoop job scheduler is intelligent enough to make sure that the computation happens on the nodes where the data is located. This is also a very important aspect as it reduces the amount of network IO.

Let's see how the framework makes it easier to run massively parallel computations with the help of this diagram:



This diagram looks a bit more complicated than the previous diagram, but most of the things are done by the Hadoop MapReduce framework itself for us. We still write the code for mapping and reducing our input data.

Let's see in detail what happens when we process our data with the Hadoop MapReduce framework from the preceding diagram:

- Our input data is broken down into pieces
- Each piece of the data is fed to a mapper program
- Outputs from all the mapper programs are collected, shuffled, and sorted
- Each sorted piece is fed to the reducer program
- Outputs from all the reducers are combined to generate the output data

Streaming MapReduce

Streaming MapReduce is one of the features that is available in the Hadoop MapReduce framework, where we can use any of the external programs to act as Mapper and Reducer. As long as these programs can be executed by the target operating system, they are accepted to run the Map and Reduce tasks.

Here are a few things to keep in mind while writing these programs:

- These programs should read the input from the `STDIN`
- They should be able to process infinite amount of data (stream) or else they crash
- The memory requirements of these programs should be known well ahead of time before they are used in the streaming MapReduce, or else we might see unpredictable behavior

In the previous section, we have written simple Perl scripts to do mapping and reduction. In the current scenario also, we will use the same programs to understand how they perform our task.

If you observe carefully, `map.pl` can process infinite amounts of data and will not have any memory overhead. But the `reduce.pl` program uses the Perl Hash data structure to perform the reduction operation. Here, we might face some memory pressure with real-world data.

In this exercise, we use randomized input data as shown here:

```
[user@node-3 ~]$ cat ./input.txt
 Bangalore,Onion,60
 NewDelhi,Onion,80
 Bangalore,Pizza,120
 Bangalore,Burger,80
 Kolkata,Onion,90
 Kolkata,Pizza,120
 Kolkata,Chilli,20
 NewDelhi,Chilli,30
 NewDelhi,Burger,180
 Kolkata,Burger,160
 NewDelhi,Pizza,150
 Bangalore,Chilli,10
```

Later, we need to copy the mapper and reducer scripts to all the Hadoop nodes:

We are using the same Hadoop cluster that's built as part of `Chapter 10`, *Production Hadoop Cluster Deployment* for this exercise. If you remember, the nodes are master, `node-1`, `node-2`, and `node-3`.

```
[user@master ~]$ scp *.pl node-1:~
[user@master ~]$ scp *.pl node-2:~
[user@master ~]$ scp *.pl node-3:~
```

In this step, we are copying the input to the `hadoop /tmp/ directory`.

Please use a sensible directory in your production environments as per your enterprise standards. Here the `/tmp` directory is used for illustration purposes only.

```
[user@node-3 ~]$ hadoop fs -put ./input.txt /tmp/
```

In this step, we are using the Hadoop streaming MapReduce framework to use our scripts for performing the computation:

The contents of the `map.pl` and `reduce.pl` are exactly the same as we have used in the previous examples.

```
[user@node-3 ~]$ hadoop jar \
    /usr/hdp/current/hadoop-mapreduce-client/hadoop-streaming.jar \
    -input hdfs:///tmp/input.txt \
    -output hdfs:///tmp/output-7 \
    -mapper $(pwd)/map.pl \
    -reducer $(pwd)/reduce.pl
```

The output is stored in HDFS, which we can view like this:

```
[user@node-3 ~]$ hadoop fs -cat /tmp/output-7/part*
 NewDelhi, 440
 Kolkata, 390
 Bangalore, 270
[user@node-3 ~]$
```

If we observe carefully, the results match exactly with our traditional program.

Java MapReduce

In the previous section, we have seen how to use any arbitrary programming language to run a MapReduce operation on Hadoop. But in most practical scenarios, it's good if we leverage the libraries provided by the Hadoop MapReduce infrastructure as they are powerful and take care of many requirements for us.

Let's try to write a simple Java program using the MapReduce libraries and see whether we can generate the same output as in the previous exercises. In this example, we will use the official MapReduce implementation from the official docs.

Documents at: `https://hadoop.apache.org/docs/r2.8.0/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html`

Since our input is very different from the example, and we also want to find the Total price of all products in a given city, we have to change the mapper program as per our CSV `input.txt` file. The reduce function is the same as the one in the official documents where our mapper function generates a `<City, Price>` pair. This can easily be consumed by the existing implementation.

We have called our program `TotalPrice.java`. Let's see how our source code looks:

```
[user@node-3 ~]$ cat TotalPrice.java
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.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.output.FileOutputFormat;

public class TotalPrice {
  public static class TokenizerMapper extends Mapper<Object, Text, Text,
IntWritable>{
    public void map(Object key, Text value, Context context) throws
IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString(), ",");
      Text city = new Text(itr.nextToken());
      itr.nextToken();
      IntWritable price = new
IntWritable(Integer.parseInt(itr.nextToken()));
      context.write(city, price);
    }
  }

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

  private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values, Context
context) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
```

```
        result.set(sum);
        context.write(key, result);
      }
    }

    public static void main(String[] args) throws Exception {
      Configuration conf = new Configuration();
      Job job = Job.getInstance(conf, "TotalPriceCalculator");
      job.setJarByClass(TotalPrice.class);
      job.setMapperClass(TokenizerMapper.class);
      job.setCombinerClass(IntSumReducer.class);
      job.setReducerClass(IntSumReducer.class);
      job.setOutputKeyClass(Text.class);
      job.setOutputValueClass(IntWritable.class);
      FileInputFormat.addInputPath(job, new Path(args[0]));
      FileOutputFormat.setOutputPath(job, new Path(args[1]));
      System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
  }
```

Once we have the source code, we need to compile it to create a **Java Archive** (**JAR**) file. It's done in the following manner:

```
[user@node-3 ~]$ javac -cp `hadoop classpath` TotalPrice.java
[user@node-3 ~]$ jar cf tp.jar TotalPrice*.class
```

Once we have the JAR file created, we can use the Hadoop command to submit the job to process the `input.txt`, and produce the output in the `/tmp/output-12` directory:

As in the case of streaming MapReduce, we need not copy the source to all the Hadoop servers.

```
 [user@node-3 ~]$ hadoop jar tp.jar TotalPrice /tmp/input.txt
/tmp/output-12
```

This run should go through fine and will produce the output files in the `/tmp/output-12` directory. We can see the contents of the output using this command:

```
[user@node-3 ~]$ hadoop fs -cat /tmp/output-12/part*
Bangalore       270
Kolkata 390
NewDelhi        440
```

This exactly matches with the previous runs as well.

As we can see, the Hadoop Mapreduce framework has taken all the necessary steps to make sure that the entire pipeline progress is kept within its control, giving us the desired result.

Even though we have used a very simple dataset for our computation, Hadoop Mapreduce makes sure that, regardless of the size of data we are dealing with, the same program we have written before yields the results we are looking for. This makes it a very powerful architecture for batch jobs.

Summary

So far, we have seen that Hadoop Mapreduce is a powerful framework that offers both streaming and batch modes of operation to process vast amounts of data with very simple instructions. Even though Mapreduce was originally the choice of computation framework in Hadoop, it has failed to meet the ever-changing demands of the market, and new architectures were developed to address those concerns. We will learn about one such framework called **Apache Spark** in the next section.

Apache Spark 2

Apache Spark is a general-purpose cluster computing system. It's very well suited for large-scale data processing. It performs 100 times better than Hadoop when run completely in-memory and 10 times better when run entirely from disk. It has a sophisticated directed acyclic graph execution engine that supports an acyclic data flow model.

Apache Spark has first-class support for writing programs in Java, Scala, Python, and R programming languages to cater to a wider audience. It offers more than 80 different operators to build parallel applications without worrying about the underlying infrastructure.

Apache Spark has libraries catering to **Structured Query Language**, known as Spark **SQL**; this supports writing queries in programs using ANSI SQL. It also has support for computing streaming data, which is very much needed in today's real-time data processing requirements such as powering dashboards for interactive user experience systems. Apache Spark also has **machine learning libraries** such as **Mlib**, which caters to running scientific programs. Then it has support for writing programs for data that follows graph data structures, known as **GraphX**. This makes it a really powerful framework that supports most advanced ways of computing.

Apache Spark runs not only on the Hadoop platform but also on a variety of systems, such as Apache Mesos, Kubernetes, Standalone, or the Cloud. This makes it a perfect choice for today's enterprise to chose the way it wants to leverage the power of this system.

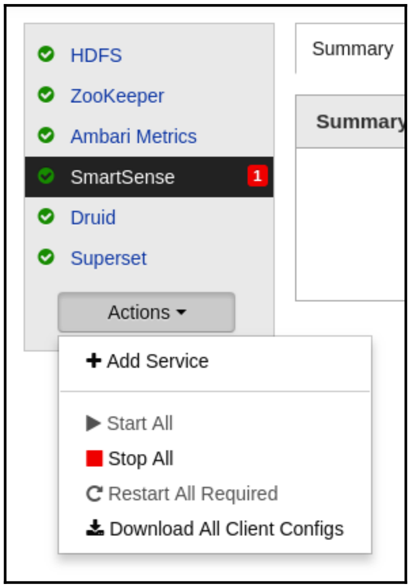
In the coming sections, we will learn more about Spark and its ecosystem. We are using Spark 2.2.0 for this exercise.

Installing Spark using Ambari

From the previous chapter, we have an existing Ambari installation that is running. We will leverage the same installation to add Spark support. Let's see how we can accomplish this.

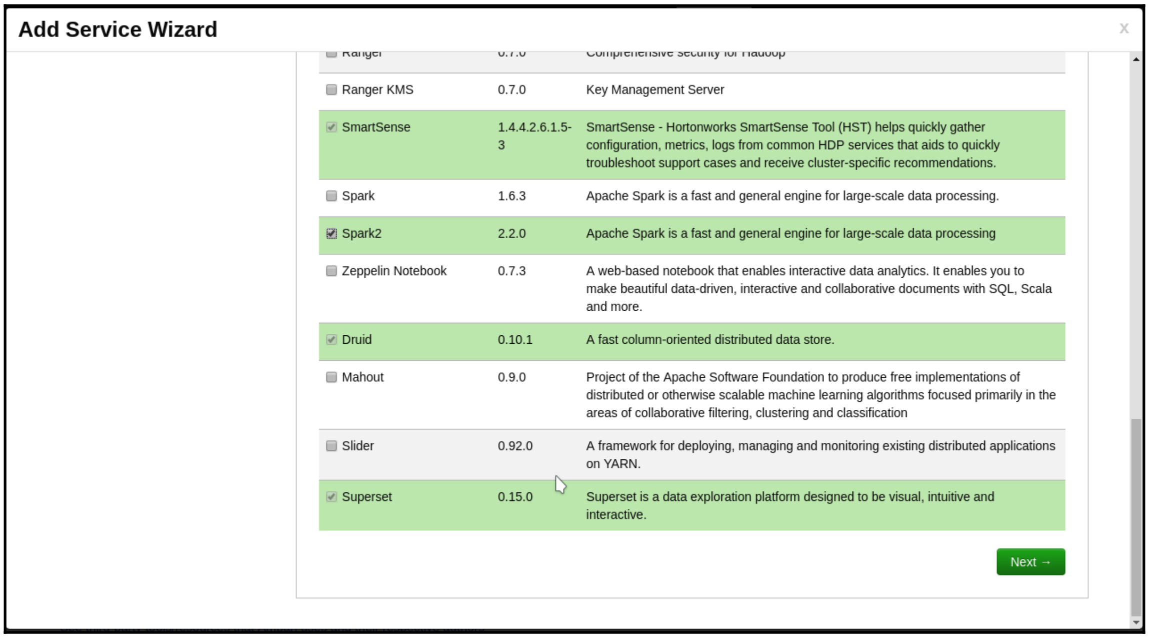
Service selection in Ambari Admin

Once we log in to the Ambari Admin interface, we see the main cluster that is created. On this page, we click on the **Actions** button on the left-hand-side menu. It shows a screen as follows. From this menu, we click on the **Add Service** option:



Add Service Wizard

Once we click on the **Add Service** menu item, we are shown a Wizard, where we have to select **Spark 2** from the list of all supported services in Ambari. The screen looks like this:



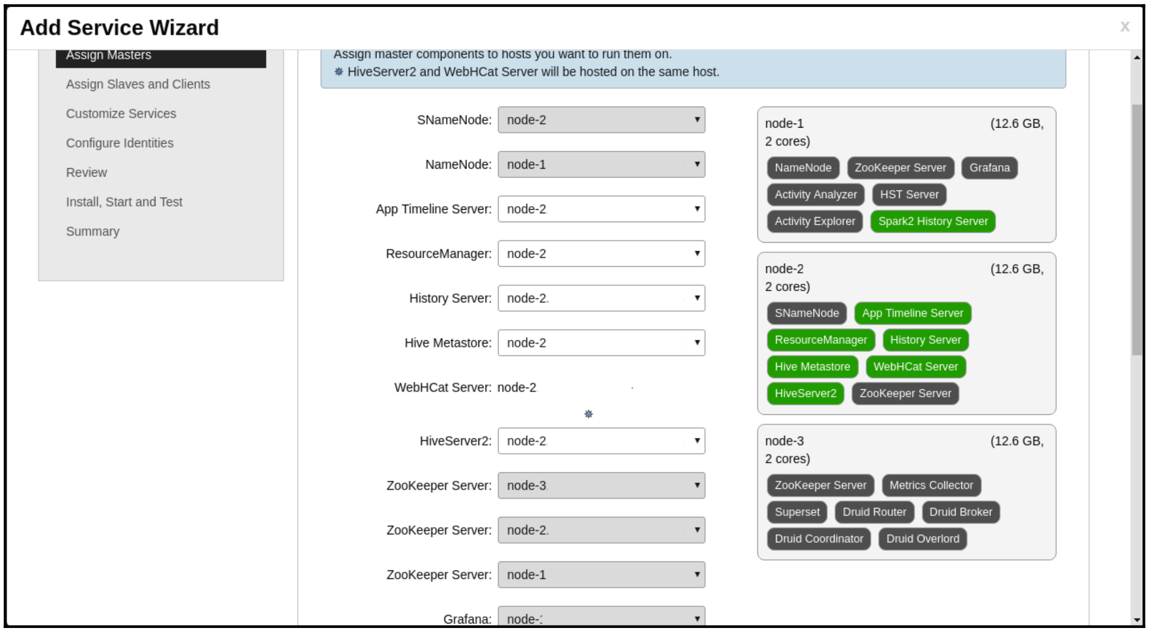
Add Service Wizard

☐	Ranger	0.7.0	Comprehensive security for Hadoop
☐	Ranger KMS	0.7.0	Key Management Server
☑	SmartSense	1.4.4.2.6.1.5-3	SmartSense - Hortonworks SmartSense Tool (HST) helps quickly gather configuration, metrics, logs from common HDP services that aids to quickly troubleshoot support cases and receive cluster-specific recommendations.
☐	Spark	1.6.3	Apache Spark is a fast and general engine for large-scale data processing.
☑	Spark2	2.2.0	Apache Spark is a fast and general engine for large-scale data processing
☐	Zeppelin Notebook	0.7.3	A web-based notebook that enables interactive data analytics. It enables you to make beautiful data-driven, interactive and collaborative documents with SQL, Scala and more.
☑	Druid	0.10.1	A fast column-oriented distributed data store.
☐	Mahout	0.9.0	Project of the Apache Software Foundation to produce free implementations of distributed or otherwise scalable machine learning algorithms focused primarily in the areas of collaborative filtering, clustering and classification
☐	Slider	0.92.0	A framework for deploying, managing and monitoring existing distributed applications on YARN.
☑	Superset	0.15.0	Superset is a data exploration platform designed to be visual, intuitive and interactive.

Click on the **Next** button when the service selection is complete.

Server placement

Once the **Spark 2** service is selected, other dependent services are also automatically selected for us and we are given a choice to select the placement of the master servers. I have left the default selection as is:


Add Service Wizard

Assign master components to hosts you want to run them on.
✻ HiveServer2 and WebHCat Server will be hosted on the same host.

SNameNode:	node-2
NameNode:	node-1
App Timeline Server:	node-2
ResourceManager:	node-2
History Server:	node-2
Hive Metastore:	node-2
WebHCat Server:	node-2
HiveServer2:	node-2
ZooKeeper Server:	node-3
ZooKeeper Server:	node-2
ZooKeeper Server:	node-1
Grafana:	node-1

node-1 (12.6 GB, 2 cores): NameNode, ZooKeeper Server, Grafana, Activity Analyzer, HST Server, Activity Explorer, Spark2 History Server

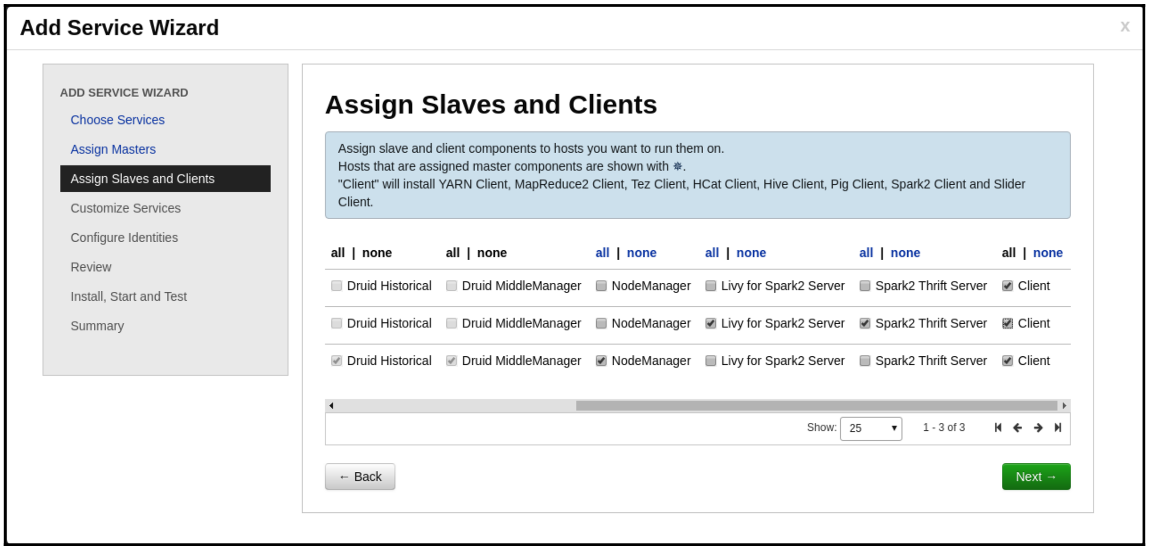
node-2 (12.6 GB, 2 cores): SNameNode, App Timeline Server, ResourceManager, History Server, Hive Metastore, WebHCat Server, HiveServer2, ZooKeeper Server

node-3 (12.6 GB, 2 cores): ZooKeeper Server, Metrics Collector, Superset, Druid Router, Druid Broker, Druid Coordinator, Druid Overlord

Click on the **Next** button when the changes look good.

Clients and Slaves selection

In this step, we are given a choice to select the list of nodes that act as clients for the masters we have selected in the previous step. We can also select the list of servers on which we can install the client utilities. Make the selection as per your choice:



Add Service Wizard

ADD SERVICE WIZARD
- Choose Services
- Assign Masters
- Assign Slaves and Clients
- Customize Services
- Configure Identities
- Review
- Install, Start and Test
- Summary

Assign Slaves and Clients

Assign slave and client components to hosts you want to run them on.
Hosts that are assigned master components are shown with ✱.
"Client" will install YARN Client, MapReduce2 Client, Tez Client, HCat Client, Hive Client, Pig Client, Spark2 Client and Slider Client.

all | none	all | none	all | none	all | none	all | none	all | none
☐ Druid Historical	☐ Druid MiddleManager	☐ NodeManager	☐ Livy for Spark2 Server	☐ Spark2 Thrift Server	☑ Client
☐ Druid Historical	☐ Druid MiddleManager	☐ NodeManager	☑ Livy for Spark2 Server	☑ Spark2 Thrift Server	☑ Client
☑ Druid Historical	☑ Druid MiddleManager	☑ NodeManager	☐ Livy for Spark2 Server	☐ Spark2 Thrift Server	☑ Client

Show: 25 1 - 3 of 3

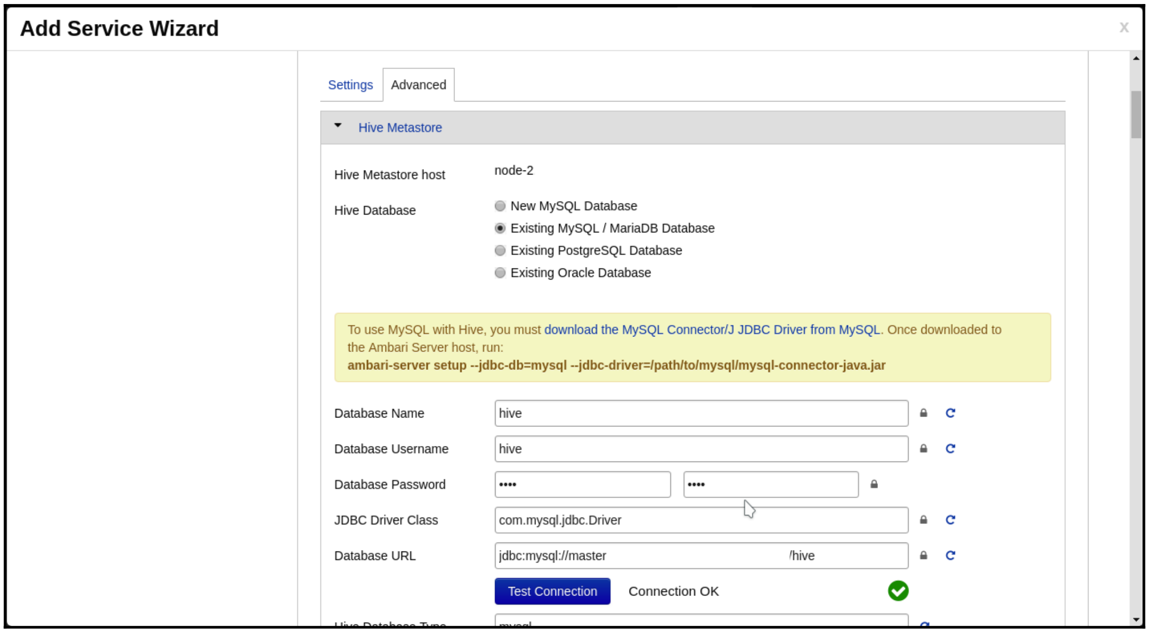
← Back Next →

Click on the **Next** button when the changes are done.

Service customization

Since Hive is also getting installed as part of the **Spark 2** selection, we are given a choice to customize the details of the Hive datasource. I have created the database on the master node with the username as `hive`, password as `hive`, and the database also as `hive`. Please choose a strong password while making changes in production.

The customization screen looks like this:



Hive Metastore host: node-2

Hive Database:
- New MySQL Database
- [✓ Existing MySQL / MariaDB Database]
- Existing PostgreSQL Database
- Existing Oracle Database

To use MySQL with Hive, you must download the MySQL Connector/J JDBC Driver from MySQL. Once downloaded to the Ambari Server host, run:
ambari-server setup --jdbc-db=mysql --jdbc-driver=/path/to/mysql/mysql-connector-java.jar

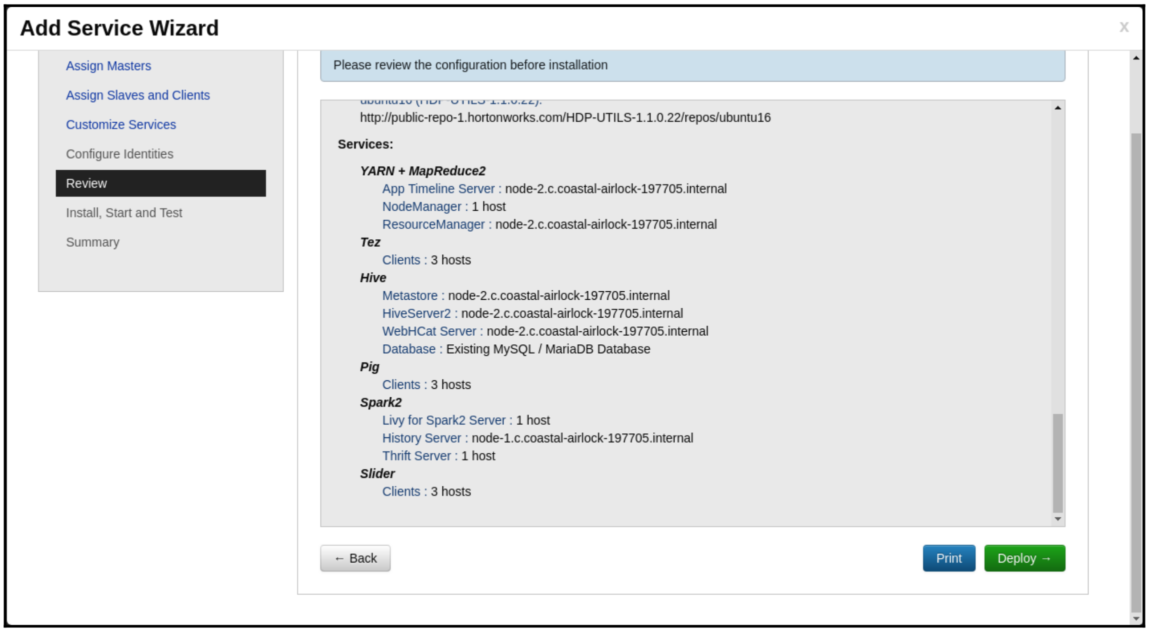
Database Name	hive
Database Username	hive
Database Password	•••• ••••
JDBC Driver Class	com.mysql.jdbc.Driver
Database URL	jdbc:mysql://master /hive

Connection OK

Click on **Next** once the changes are done correctly.

Software deployment

In this screen, we are shown a summary of the selections we have made so far. Click on **Deploy** to start deploying the Spark 2 software on the selected servers. We can always cancel the wizard and start over again in this step if we feel that we have missed any customization:



Please review the configuration before installation

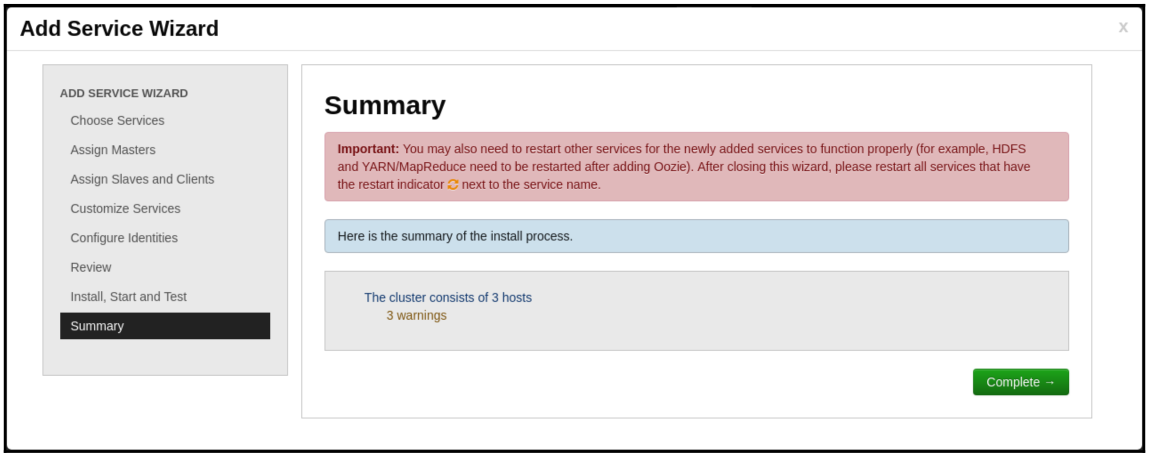
ubuntu16 (HDP-UTILS-1.1.0.22):
http://public-repo-1.hortonworks.com/HDP-UTILS-1.1.0.22/repos/ubuntu16

Services:

YARN + MapReduce2
App Timeline Server : node-2.c.coastal-airlock-197705.internal
NodeManager : 1 host
ResourceManager : node-2.c.coastal-airlock-197705.internal
Tez
Clients : 3 hosts
Hive
Metastore : node-2.c.coastal-airlock-197705.internal
HiveServer2 : node-2.c.coastal-airlock-197705.internal
WebHCat Server : node-2.c.coastal-airlock-197705.internal
Database : Existing MySQL / MariaDB Database
Pig
Clients : 3 hosts
Spark2
Livy for Spark2 Server : 1 host
History Server : node-1.c.coastal-airlock-197705.internal
Thrift Server : 1 host
Slider
Clients : 3 hosts

Spark installation progress

In this step, we are shown the progress of Spark software installation and its other dependencies. Once everything is deployed, we are shown a summary of any warnings and errors. As we can see from the following screen, there are some warnings encountered during the installation, which indicates that we need to restart a few services once the wizard is complete. Don't worry its pretty normal to see these errors. We will correct these in the coming steps to have a successfully running Spark system:



Summary

Important: You may also need to restart other services for the newly added services to function properly (for example, HDFS and YARN/MapReduce need to be restarted after adding Oozie). After closing this wizard, please restart all services that have the restart indicator next to the service name.

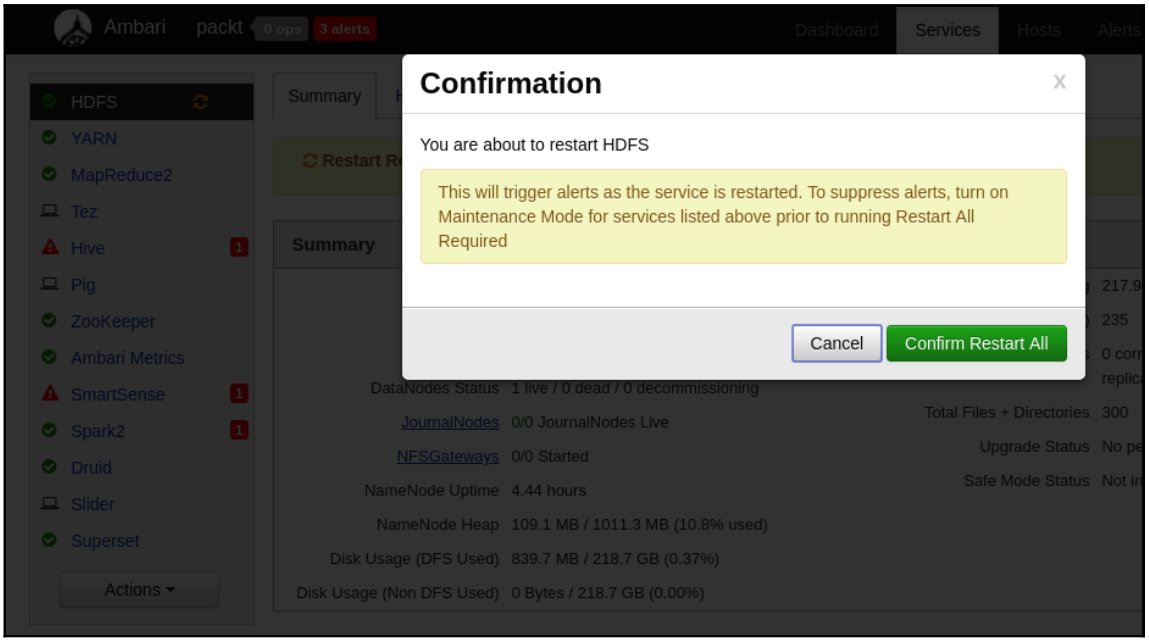
Here is the summary of the install process.

The cluster consists of 3 hosts
3 warnings

Clicking on **Complete** finishes the wizard.

Service restarts and cleanup

Since there were warnings during the installation process, we have to restart all the affected components. The restart process is shown in this screen:



Confirmation

You are about to restart HDFS

This will trigger alerts as the service is restarted. To suppress alerts, turn on Maintenance Mode for services listed above prior to running Restart All Required

DataNodes Status 1 live / 0 dead / 0 decommissioning
JournalNodes 0/0 JournalNodes Live
NFSGateways 0/0 Started
NameNode Uptime 4.44 hours
NameNode Heap 109.1 MB / 1011.3 MB (10.8% used)
Disk Usage (DFS Used) 839.7 MB / 218.7 GB (0.37%)
Disk Usage (Non DFS Used) 0 Bytes / 218.7 GB (0.00%)
Total Files + Directories 300

Once we give a confirmation, all the associated services will be restarted and we will have a successfully running system.

This finishes the installation of Spark 2 on an existing Hadoop cluster managed by Ambari. We will now learn more about various data structures and libraries in Spark in the coming sections.

Apache Spark data structures

Even though Mapreduce provides a powerful way to process large amounts of data, it is restricted due to several drawbacks:

- Lack of support for variety of operators
- Real-time data processing
- Caching the results of data for faster iterations

This is to name a few. Since Apache Spark was built from the ground up, it has approached the big data computation problem in a very generic way and has provided the developers with data structures that makes it easier to represent any type of data and use those to compute in a better way.

RDDs, DataFrames and datasets

At the core of Apache Spark are distributed datasets called **RDD**, also known as **Resilient Distributed Datasets**. These are immutable datasets that are present in the cluster, which are highly available and fault tolerant. The elements in the RDD can be operated in parallel, giving a lot of power to the Spark cluster.

Since the data is already present in storage systems, such as HDFS, RDBMS, S3, and so on, RDDs can easily be created from these external datasources. The API also provides us with the power to create RDDs from existing in-memory data elements.

These RDDs do not have any pre-defined structure. So, they can assume any form and, by leveraging the different operators in the Spark library, we can write powerful programs that give us necessary results without worrying too much about the data complexities.

In order to cater to the RDBMS needs, DataFrames come into play where a DataFrame can be compared with a table in a relational database system. As we know, tables have rows and columns and the structure of the data is known ahead of time. By knowing the structure of the data, several optimizations can be performed during data processing.

Spark datasets are somewhat similar to the DataFrames. But they extend the functionality of the DataFrames by supporting semi-structured data objects with native language objects (Java and Scala). DataFrames are an immutable collection of objects with semantics of a relational schema. Since we are dealing with semi-structured data and native language objects, there is an encoder/decoder system that takes care of automatically converting between the types.

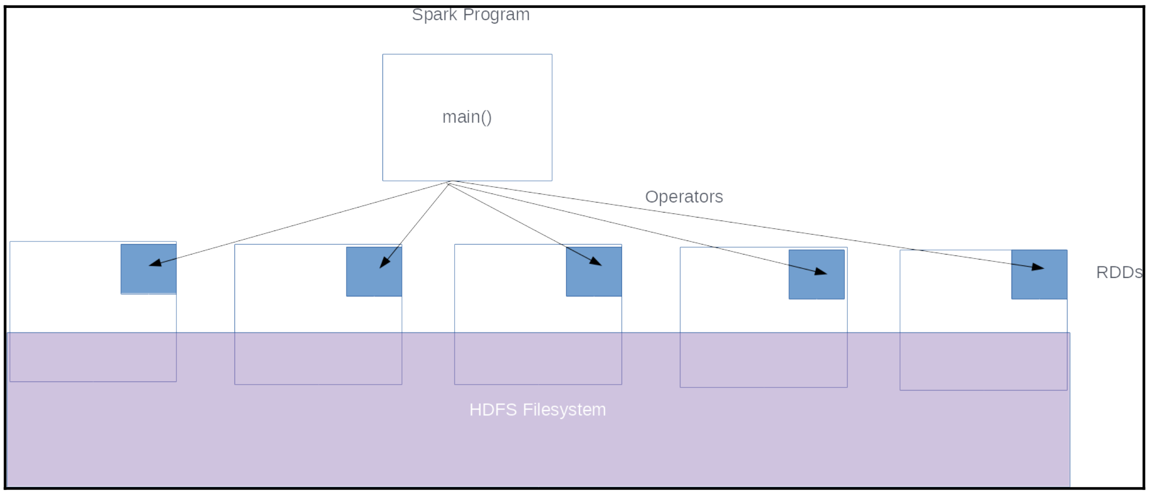
Here is a quick comparison chart:

Feature	**RDDs**	**DataFrame**	**Dataset**
Data type	Unstructured data	Structured data	Semi-structured data
Schema requirement	Completely free form	Strict datatypes	Loosely coupled
Optimization provided by Spark	Not needed as data is unstructured	Leverages optimizations as datatypes are known	Inferred datatypes provide some level of optimization
High level expressions/filters	Difficult as the data form is complex in nature	Can leverage these as we know the data we are dealing with	Can leverage here too

Apache Spark programming

Apache Spark has very good programming language support. It provides first-class support for Java, Scala, Python, and R programming languages. Even though the data structures and operators that are available with the programming languages are similar in nature, we have to use programming-language-specific constructs to achieve the desired logic. Throughout this chapter, we will use Python as the programming language of choice. However, Spark itself is agnostic to these programming languages and produces the same results regardless of the programming language used.

Apache Spark with Python can be used in two different ways. The first way is to launch the `pyspark` interactive shell, which helps us run Python instructions. The experience is similar to the Python shell utility. Another way is to write standalone programs that can be invoked using the spark-submit command. In order to use standalone Spark programs, we have to understand the basic structure of a Spark program:



The typical anatomy of a spark program consists of a main function that executes different operators on the RDDs to generate the desired result. There is support for more than 80 different types of operators in the Spark library. At a high level, we can classify these operators into two types: transformations and actions. Transformation operators convert data from one form to another. Actions generate the result from the data. In order to optimize the resources in the cluster for performance reasons, Apache Spark actually executes the programs in checkpoints. Each checkpoint is arrived at only when there is a action operator. This is one important thing to remember, especially if you are new to programming with Spark. Even the most advanced programmers sometimes get confused about why they don't see the desired result as they did not use any action operator on the data.

Coming back to the preceding diagram, we have a driver program that has main routine which performs several actions/transformations on the data thats stored in a filesystem like HDFS and gives us the desired result. We are aware that RDDs are the basic parallel datastore in the Spark programming language. Spark is intelligent enough to create these RDDs from the seed storage like HDFS and once they are created, it can cache the RDDs in Memory and also make these RDDs highly available by making them fault-tolerant. Even if the copy of the RDD goes offline due to a node crash, future access on the same RDDs will quickly be generated from the computation from which it was originally generated.

Sample data for analysis

In order to understand the programming API of spark, we should have a sample dataset on which we can perform some operations to gain confidence. In order to generate this dataset, we will import the sample table from the employees database from the previous chapter.

These are the instructions we follow to generate this dataset:

Log in to the server and switch to Hive user:

```
ssh user@node-3
[user@node-3 ~]$ sudo su - hive
```

This will put us in a remote shell, where we can dump the table from the MySQL database:

```
[hive@node-3 ~]$ mysql -usuperset -A -psuperset -h master employees -e
"select * from vw_employee_salaries" > vw_employee_salaries.tsv
[hive@node-3 ~]$ wc -l vw_employee_salaries.tsv
2844048 vw_employee_salaries.tsv
[hive@node-3 ~]$
```

Next, we should copy the file to Hadoop using the following command:

```
[hive@node-3 ~]$ hadoop fs -put ./vw_employee_salaries.tsv
/user/hive/employees.csv
```

Now, the data preparation is complete as we have successfully copied it to HDFS. We can start using this data with Spark.

Interactive data analysis with pyspark

Apache Spark distribution comes with an interactive shell called **pyspark**. Since we are dealing with interpreted programming languages like Python, we can write interactive programs while learning.

If you remember, we have installed Spark with Apache Ambari. So we have to follow the standard directory locations of Apache Ambari to access the Spark-related binaries:

```
[hive@node-3 ~]$ cd /usr/hdp/current/spark2-client/
[hive@node-3 spark2-client]$ ./bin/pyspark
Python 2.7.5 (default, Aug  4 2017, 00:39:18)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-16)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use
setLogLevel(newLevel).
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.2.0.2.6.4.0-91
      /_/

Using Python version 2.7.5 (default, Aug  4 2017 00:39:18)
SparkSession available as 'spark'.
>>>
```

The preceding steps launch the interactive Spark shell.

As a first step in understanding Spark's data structures, we will load the `employees.csv` file from the HDFS and count the total number of lines in the file using these instructions:

```
>>> ds = spark.read.text("employees.csv")
>>> ds.count()
2844048
>>>
```

As we can see, the count matches with the previous load operation on the Unix shell.

Now, let's try to load the first five records from the file and try to see the schema of the data structure object:

```
>>> ds.first()
Row(value=u'emp_no\tbirth_date\tfirst_name\tlast_name\tgender\thire_date\ts
alary\tfrom_date\tto_date')
>>> ds.head(5)
```

```
[Row(value=u'emp_no\tbirth_date\tfirst_name\tlast_name\tgender\thire_date\t
salary\tfrom_date\tto_date'),
Row(value=u'10001\t1953-09-02\tGeorgi\tFacello\tM\t1986-06-26\t60117\t1986-
06-26\t1987-06-26'),
Row(value=u'10001\t1953-09-02\tGeorgi\tFacello\tM\t1986-06-26\t62102\t1987-
06-26\t1988-06-25'),
Row(value=u'10001\t1953-09-02\tGeorgi\tFacello\tM\t1986-06-26\t66074\t1988-
06-25\t1989-06-25'),
Row(value=u'10001\t1953-09-02\tGeorgi\tFacello\tM\t1986-06-26\t66596\t1989-
06-25\t1990-06-25')]
>>> ds.printSchema()
root
 |-- value: string (nullable = true)

>>>
```

As we can see, even though we have a CSV (tab separated file), Spark has read the file as a normal text file separated by newlines and the schema contains only one value, which is of string datatype.

In this mode of operation, where we treat each record as a line, we can perform only a few types of operations, such as counting all occurrences of a given name:

```
>>> ds.filter(ds.value.contains("Georgi")).count()
2323
>>>
```

This mode of operations is somewhat similar to log processing. But the true power of Spark comes from the power of treating the data as a table with rows and columns, also known as **DataFrames**:

```
>>> ds = spark.read.format("csv").option("header",
"true").option("delimiter", "\t").load("employees.csv")
>>> ds.count()
2844047

>>> ds.show(5)
+------+----------+----------+---------+------+----------+------+----------
+----------+
|emp_no|birth_date|first_name|last_name|gender| hire_date|salary|
from_date|   to_date|
+------+----------+----------+---------+------+----------+------+----------
+----------+
| 10001|1953-09-02|    Georgi|  Facello|     M|1986-06-26|
60117|1986-06-26|1987-06-26|
| 10001|1953-09-02|    Georgi|  Facello|     M|1986-06-26|
62102|1987-06-26|1988-06-25|
| 10001|1953-09-02|    Georgi|  Facello|     M|1986-06-26|
66074|1988-06-25|1989-06-25|
| 10001|1953-09-02|    Georgi|  Facello|     M|1986-06-26|
66596|1989-06-25|1990-06-25|
| 10001|1953-09-02|    Georgi|  Facello|     M|1986-06-26|
66961|1990-06-25|1991-06-25|
+------+----------+----------+---------+------+----------+------+----------
+----------+
only showing top 5 rows

>>>

>>> ds.printSchema()
root
 |-- emp_no: string (nullable = true)
 |-- birth_date: string (nullable = true)
 |-- first_name: string (nullable = true)
 |-- last_name: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- hire_date: string (nullable = true)
 |-- salary: string (nullable = true)
 |-- from_date: string (nullable = true)
 |-- to_date: string (nullable = true)

>>>
```

Now, we can see that Spark has automatically converted the input CSV text into a DataFrame. But all the fields are treated as strings.

Let's try to use the schema inference feature of spark to automatically find the datatype of the fields:

```
>>> ds = spark.read.format("csv").option("header",
"true").option("delimiter", "\t").option("inferSchema",
"true").load("employees.csv")
18/03/25 19:21:15 WARN FileStreamSink: Error while looking for metadata
directory.
18/03/25 19:21:15 WARN FileStreamSink: Error while looking for metadata
directory.
>>> ds.count()
2844047
>>> ds.show(2)
+------+--------------------+----------+---------+------+--------------------
+------+--------------------+--------------------+
|emp_no|          birth_date|first_name|last_name|gender|
hire_date|salary|           from_date|             to_date|
+------+--------------------+----------+---------+------+--------------------
+------+--------------------+--------------------+
| 10001|1953-09-02 00:00:00|    Georgi|  Facello|     M|1986-06-26
00:00:00| 60117|1986-06-26 00:00:00|1987-06-26 00:00:00|
| 10001|1953-09-02 00:00:00|    Georgi|  Facello|     M|1986-06-26
00:00:00| 62102|1987-06-26 00:00:00|1988-06-25 00:00:00|
+------+-------------------+----------+---------+------+-------------------
+------+-------------------+-------------------+
only showing top 2 rows

>>> ds.printSchema()
root
 |-- emp_no: integer (nullable = true)
 |-- birth_date: timestamp (nullable = true)
 |-- first_name: string (nullable = true)
 |-- last_name: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- hire_date: timestamp (nullable = true)
 |-- salary: integer (nullable = true)
 |-- from_date: timestamp (nullable = true)
 |-- to_date: timestamp (nullable = true)

>>>
```

Now we can see that all the fields have a proper datatype that is closest to the MySQL table definition.

We can apply simple actions on the data to see the results. Let's try to find the total male records:

```
>>> ds.filter(ds.gender == "M").count()
1706321
```

Also, try to find the male records that have more than $100K of pay:

```
>>> ds.filter(ds.gender == "M").filter(ds.salary > 100000).count()
57317
```

It's so simple, right? There are many more operators that are available for exploration in the official Spark documentation.

Standalone application with Spark

In the previous section, we have seen how to use the interactive shell `pyspark` to learn the Spark Python API. In this section, we will write a simple Python program that we will run on the Spark cluster. In real-world scenarios, this is how we run our applications on the Spark cluster.

In order to do this, we will write a program called `MyFirstApp.py` with the following contents:

```
[hive@node-3 ~]$ cat MyFirstApp.py
from pyspark.sql import SparkSession

# Path to the file in HDFS
csvFile = "employees.csv"

# Create a session for this application
spark = SparkSession.builder.appName("MyFirstApp").getOrCreate()

# Read the CSV File
csvTable = spark.read.format("csv").option("header",
"true").option("delimiter", "\t").load(csvFile)

# Print the total number of records in this file
print "Total records in the input : {}".format(csvTable.count())

# Stop the application
spark.stop()
[hive@node-3 ~]$
```

In order to run this program on the Spark cluster, we have to use the spark-submit command, which does the needful in terms of scheduling and coordinating the complete application life cycle:

```
[hive@node-3 ~]$ /usr/hdp/current/spark2-client/bin/spark-submit
./MyFirstApp.py 2>&1 | grep -v -e INFO -e WARN
Total records in the input : 2844047
```

As expected, those are the total number of records in our input file (excluding the header line).

Spark streaming application

One of the powerful features of spark is building applications that process real-time streaming data and produce real-time results. In order to understand this more, we will write a simple application that tries to find duplicate messages in an input stream and prints all the unique messages.

This kind of application is helpful when we are dealing with an unreliable stream of data and we want to submit only the data that is unique.

The source code for this application is given here:

```
[hive@node-3 ~]$ cat StreamingDedup.py
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

context = SparkContext(appName="StreamingDedup")
stream = StreamingContext(context, 5)

records = stream.socketTextStream("localhost", 5000)
records
    .map(lambda record: (record, 1))
    .reduceByKey(lambda x,y: x + y)
    .pprint()

ssc.start()
ssc.awaitTermination()
```

In this application, we connect to a remote service on port `5000`, which emits the messages at its own page. The program summarizes the result of operation every 5 seconds as defined in the `StreamingContext` parameter.

Now, let's start a simple TCP server using the UNIX netcat command (`nc`) and a simple loop:

```
for i in $(seq 1 10)
do
  for j in $(seq 1 5)
  do
   sleep 1
   tail -n+$(($i * 3)) /usr/share/dict/words | head -3
  done
done | nc -l 5000
```

After this, submit our program to the spark cluster:

```
[hive@node-3 ~]$ /usr/hdp/current/spark2-client/bin/spark-submit
./StreamingDedup.py 2>&1 | grep -v -e INFO -e WARN
```

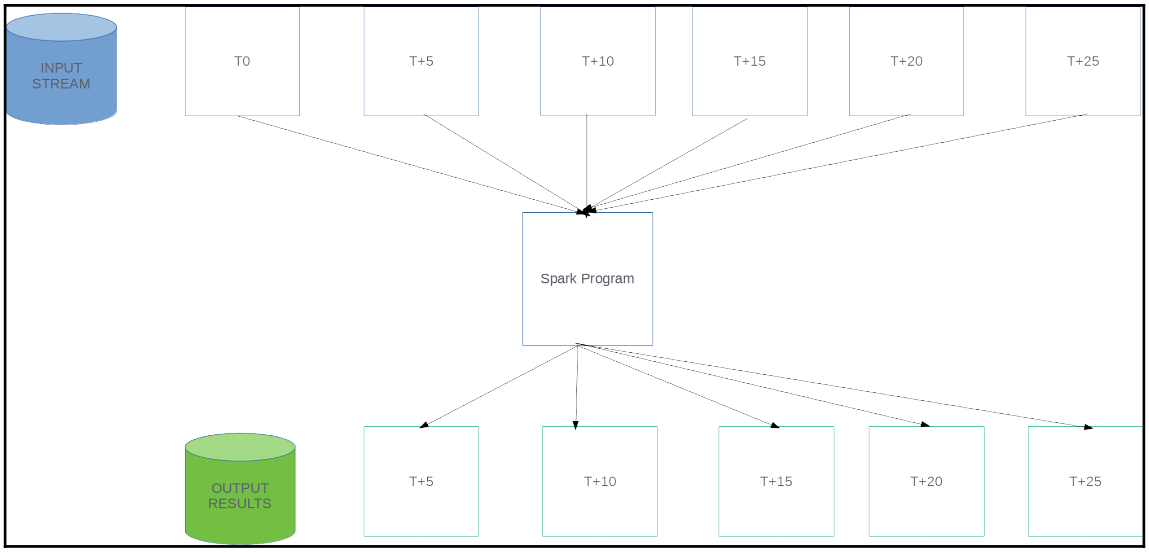
After the program starts, we see the following output:

```
-------------------------------------------
Time: 2018-03-26 04:33:45
-------------------------------------------
(u'16-point', 5)
(u'18-point', 5)
(u'1st', 5)
```

```
-------------------------------------------
Time: 2018-03-26 04:33:50
-------------------------------------------
(u'2', 5)
(u'20-point', 5)
(u'2,4,5-t', 5)
```

We see that every word has exactly 5 as the count, which is expected as we are printing it five times in the Unix command loop.

We can understand this with the help of this diagram:



INPUT STREAM produces a continuous stream of data, which is consumed in real time by the **Spark Program**. After that, the results are printed by eliminating the duplicates

If we see this in chronological order, the data from time zero to time five seconds (**T0** - **T5**) is processed and results are available in **T5** time. Same thing for all other time slots.

In this simple example, we have just learned the basics of how Spark Streaming can be used to build real-time applications.

Spark SQL application

When writing applications using Spark, developers have the option to use SQL on structured data to get the desired results. An example makes this easier for us to understand how to do this:

```
[hive@node-3 ~]$ cat SQLApp.py
from pyspark.sql import SparkSession

# Path to the file in HDFS
csvFile = "employees.csv"

# Create a session for this application
spark = SparkSession.builder.appName("SQLApp").getOrCreate()

# Read the CSV File
csvTable = spark.read.format("csv").option("header",
"true").option("delimiter", "\t").load(csvFile)
csvTable.show(3)

# Create a temporary view
csvView = csvTable.createOrReplaceTempView("employees")

# Find the total salary of employees and print the highest salary makers
highPay = spark.sql("SELECT first_name, last_name, emp_no, SUM(salary) AS
total FROM employees GROUP BY emp_no, first_name, last_name ORDER BY
SUM(salary)")

# Generate list of records
results = highPay.rdd.map(lambda rec: "Total: {}, Emp No: {}, Full Name: {}
{}".format(rec.total, rec.emp_no, rec.first_name, rec.last_name)).collect()

# Show the top 5 of them
for r in results[:5]:
    print(r)

# Stop the application
spark.stop()
[hive@node-3 ~]$
```

In this example, we build a DataFrame from `employees.csv` and then create a view in memory called **employees**. Later, we can use ANSI SQL to write and execute queries to generate the necessary results.

Since we are interested in finding the top paid employees, the results are shown as expected:

```
[hive@node-3 ~]$ /usr/hdp/current/spark2-client/bin/spark-submit
./SQLApp.py 2>&1 | grep -v -e INFO -e WARN
[rdd_10_0]
+------+----------+----------+---------+------+----------+------+----------
+----------+
|emp_no|birth_date|first_name|last_name|gender| hire_date|salary|
from_date|   to_date|
+------+----------+----------+---------+------+----------+------+----------
+----------+
| 10001|1953-09-02|    Georgi|  Facello|     M|1986-06-26|
60117|1986-06-26|1987-06-26|
| 10001|1953-09-02|    Georgi|  Facello|     M|1986-06-26|
62102|1987-06-26|1988-06-25|
| 10001|1953-09-02|    Georgi|  Facello|     M|1986-06-26|
66074|1988-06-25|1989-06-25|
+------+----------+----------+---------+------+----------+------+----------
+----------+
only showing top 3 rows

Total: 40000.0, Emp No: 15084, Full Name: Aloke Birke
Total: 40000.0, Emp No: 24529, Full Name: Mario Antonakopoulos
Total: 40000.0, Emp No: 30311, Full Name: Tomofumi Coombs
Total: 40000.0, Emp No: 55527, Full Name: Kellyn Ouhyoung
Total: 40000.0, Emp No: 284677, Full Name: Richara Eastman
```

As we can see, the simplified API provided by Apache Spark makes it easier to write SQL Queries on top of CSV data (without the need for an RDBMS) to get what we are looking for.

Summary

In this chapter, you looked at the basic concepts of large-scale data processing frameworks and also learned that one of the powerful features of spark is building applications that process real-time streaming data and produce real-time results.

In the next few chapters, we will discuss how to build real-time data search pipelines with Elasticsearch stack.

8 Building Enterprise Search Platform

After learning data ingestions and data persistence approaches, let's learn about searching the data. In this chapter, we will learn about the following important things:

- Data search techniques
- Building real-time search engines.
- Searching real-time, full-text data
- Data indexing techniques
- Building a real-time data search pipeline

The data search concept

In our everyday life, we always keep on searching something. In the morning, we search for a toothbrush, newspaper, search stock prices, bus schedule, office bag, and so on. The list goes on and on. This search activity stops when we go to bed at the end of the day. We use a lot of tools and techniques to search these things to minimize the actual search time. We use Google to search most of the things such as news, stock prices, bus schedule, and anything and everything we need. To search a particular page of a book, we use the book's index. So, the point is that search is a very important activity of our life. There are two important concepts can be surfaced out of this, that is, search tool and search time. Just think of a situation where you want to know about a particular stock price of a company and it takes a few minutes to load that page. You will definitely get very annoyed. It is because the Search Time in this case is not acceptable to you. So then the question is, *How to reduce this search time?* We will learn that in this chapter.

The need for an enterprise search engine

Just like we all need a tool to search our own things, every company also needs a search engine to build so that internal and external entities can find what they want.

For example, an employee has to search for his/her PTO balance, paystub of a particular month, and so on. The HR department may search for employees who are in finance group or so. In an e-commerce company, a product catalog is the most searchable object. It is a very sensitive object because it directly impacts the revenue of the company. If a customer wants to buy a pair of shoes, the first thing he/she can do is search the company product catalog. If the search time is more than a few seconds, the customer may lose interest in the product. It may also be possible that the same customer goes to another website to buy a pair of shoes, resulting in a loss of revenue.

It appears that even with all the tech and data in the world, we can't do much without two crucial components:

- Data search
- Data index

Companies such as Google, Amazon, and Apple have changed the word's expectations of search. We all expect them to search anything, anytime, and using any tool such as website, mobile, and voice-activated tools like Google Echo, Alexa, and HomePad. We expect these tools to answer all our questions, from *How's the weather today?* to give me a list of gas stations near me.

As these expectations are growing, the need to index more and more data is also growing.

Tools for building an enterprise search engine

The following are some popular tools/products/technologies available:

- Apache Lucene
- Elasticsearch
- Apache Solr
- Custom (in-house) search engine

In this chapter, I will focus on Elasticsearch in detail. I will discuss Apache Solr on a conceptual level only.

Elasticsearch

Elasticsearch is an open source search engine. It is based on Apache Lucene. It is distributed and supports multi-tenant capability. It uses schema-free JSON documents and has a built-in, HTTP-based web interface. It also supports analytical RESTful query workloads. It is a Java-based database server. Its main protocol is HTTP/JSON.

Why Elasticsearch?

Elasticsearch is the most popular data indexing tool as of today. It is because of its the following features:

- It is **fast**. Data is indexed at a real-time speed.
- It is **scalable**. It scales horizontally.
- It is **flexible**. It supports any data format, structured, semi-structured, or unstructured.
- It is **distributed**. If one node fails, the cluster is still available for business.
- It supports data search query in any language: Java, Python Ruby, C#, and so on.
- It has a **Hadoop connector,** which facilitates smooth communication between Elasticsearch and Hadoop.
- It supports robust data **aggregation** on huge datasets to find trends and patterns.
- The **Elastic stack** (Beats, Logstash, Elasticsearch, and Kibana) and X-Pack offers out-of-the-box support for data ingestion, data indexing, data visualization, data security, and monitoring.

Elasticsearch components

Before we take a deep dive, let's understand a few important components of Elasticsearch.

Index

Elasticsearch index is a collection of JSON documents. Elasticsearch is a data store that may contain multiple indices. Each index may be divided into one or many types. A type is a group of similar documents. A type may contain multiple documents. In terms of database analogy, an index is a database and each of its types is a table. Each JSON document is a row in that table.

Indices created in Elasticsearch 6.0.0 or later may only contain a single mapping type.

Mapping types will be completely removed in Elasticsearch 7.0.0.

Document

Document in Elasticsearch means a JSON document. It is a basic unit of data to be stored in an index. An index comprises multiple documents. In the RDBMS world, a document is nothing but a row in a table. For example, a customer document may look like the following:

```
{
"name": "Sam Taylor",
"birthdate": "1995-08-11",
"address":
{
"street": "155 rabbit Street",
"city": "San Francisco",
"state": "ca",
"postalCode": "94107"
},
"contactPhone":
[
{
"type": "home",
"number": "510-415-8929"
},
{
"type": "cell",
"number": "408-171-8187"
}
]
}
```

Mapping

Mapping is schema definition of an index. Just like a database, we have to define a data structure of a table. We have to create a table, its columns, and column data types. In Elasticsearch, we have define a structure of an index during its creation. We may have to define which field can be indexed, searchable, and storable.

The good news is that, Elasticsearch supports **dynamic mapping**. It means that mapping is not mandatory at index creation time. An index can be created without mapping. When a document is sent to Elasticsearch for indexing, Elasticsearch automatically defines the data structure of each field and make each field a searchable field.

Cluster

Elasticsearch is a collection of nodes (servers). Each node may store part of the data in index and provides federated indexing and search capabilities across all nodes. Each cluster has a unique name, `elasticsearch`, by default. A cluster is divided into multiple types of nodes, namely Master Node and Data Node. But an Elasticsearch cluster can be created using just one node having both Master and Data nodes installed on the same node:

- **Master node**: This controls the entire cluster. There can be more than one master node in a cluster (three are recommended). Its main function is index creation or deletion and allocation of shards (partitions) to data nodes.
- **Data node**: This stores the actual index data in shards. They support all data-related operations such as aggregations, index search, and so on.

Type

Documents are divided into various logical types for example, order document, product document, customer document, and so on. Instead of creating three separate order, product, and customer indices, a single index can be logically divided into order, product and customer types. In RDBMS analogy, a type is nothing but a Table in a database. So, a type is a logical partition of an index.

Type is deprecated in Elasticsearch version 6.0.

How to index documents in Elasticsearch?

Let's learn how Elasticsearch actually works by indexing these three sample documents. While learning this, we will touch upon a few important functions/concepts of Elasticsearch.

These are the three sample JSON documents to be indexed:

```
{
"name": "Angela Martin",
"birthdate": "1997-11-02",
"street": "63542 Times Square",
"city": "New York",
"state": "NY",
"zip": "10036",
"homePhone": "212-415-8929",
"cellPhone": "212-171-8187"
} ,
{
"name": "Sam Taylor",
"birthdate": "1995-08-11",
"street": "155 rabbit Street",
"city": "San Francisco",
"state": "ca",
"zip": "94107",
"homePhone": "510-415-8929",
"cellPhone": "408-171-8187"
} ,
{
"name": "Dan Lee",
"birthdate": "1970-01-25",
"street": "76336 1st Street",
"city": "Los Angeles",
"state": "ca",
"zip": "90010",
"homePhone": "323-892-5363",
"cellPhone": "213-978-1320"
}
```

Elasticsearch installation

First things first. Let's install Elasticsearch.

Please do the following steps to install Elasticsearch on your server. It is assumed that you are installing Elasticsearch using CentOS 7 on your server.

What minimum Hardware is required?

- **RAM**: 4 GB
- **CPU**: 2

Which JDK needs to be installed? We need JDK 8. If you don't have JDK 8 installed on your server, do the following steps to install JDK 8:

1. Change to home folder:

   ```
   $ cd ~
   ```

2. Download JDK RPM:

   ```
   $ 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/8u73-b02/jdk-8u73-linux
   -x64.rpm
   ```

3. Install RMP using YUM (it is assumed that you have `sudo` access):

   ```
   $ sudo yum -y localinstall jdk-8u73-linux-x64.rpm
   ```

Since, we have installed JDK 8 on our server successfully, let's start installing Elasticsearch.

Installation of Elasticsearch

For detailed installation steps, please refer to the following URL:

https://www.elastic.co/guide/en/elasticsearch/reference/current/rpm.html

1. The RPM for Elasticsearch v6.2.3 can be downloaded from the website and installed as follows:

   ```
   $ wget
   https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-
   6.1.2.rpm
   $ sudo rpm --install elasticsearch-6.1.2.rpm
   ```

2. To configure Elasticsearch to start automatically when the system boots up, run the following commands.

   ```
   sudo /bin/systemctl daemon-reload
   sudo /bin/systemctl enable elasticsearch.service
   ```

3. Elasticsearch can be started and stopped as follows:

   ```
   sudo systemctl start elasticsearch.service
   sudo systemctl stop elasticsearch.service
   ```

The main configuration file is located in the config folder called `elasticsearch.yml`.

Let's do the following initial config changes in `elasticsearch.yml`. Find and replace the following parameters:

```
cluster.name: my-elaticsearch
path.data: /opt/data
path.logs: /opt/logs
network.host: 0.0.0.0
http.port: 9200
```

Now start Elasticsearch:

```
sudo systemctl start elasticsearch.service
```

Check whether Elasticsearch is running using the following URL:

```
http://localhost:9200
```

We will get the following response:

```
// 20180320161034
// http://localhost:9200/
{
    "name": "o7NVnfX",
"cluster_name": "my-elasticsearch",
"cluster_uuid": "jmB-_FEuTb6N_OFokwxF1A",
"version": {
"number": "6.1.2",
"build_hash": "5b1fea5",
"build_date": "2017-01-10T02:35:59.208Z",
"build_snapshot": false,
"lucene_version": "7.1.0",
"minimum_wire_compatibility_version": "5.6.0",
"minimum_index_compatibility_version": "5.0.0"
},
"tagline": "You Know, for Search"
}
```

Now, our Elasticsearch is working fine. Let's create an index to store our documents.

Create index

We will use the following `curl` command to create our first index named `my_index`:

```
curl -XPUT 'localhost:9200/my_index?pretty' -H 'Content-Type:
application/json' -d'
{
"settings" : {
"index" : {
"number_of_shards" : 2,
"number_of_replicas" : 1
}
}
}
'
```

We will get this response:

```
{
"acknowledged" : true,
"shards_acknowledged" : true,
"index" : "my_index"
}
```

In the index creation URL, we used settings, shards, and replica. Let's understand what is meant by shard and replica.

Primary shard

We have created index with three shards. It means Elasticsearch will divide index into three partitions. Each partition is called a **shard**. Each shard is a full-fledged, independent Lucene index. The basic idea is that Elasticsearch will store each shard on a separate data node to increase the scalability. We have to mention how many shards we want at the time of index creation. Then, Elasticsearch will take care of it automatically. During document search, Elasticsearch will aggregate all documents from all available shards to consolidate the results so as to fulfill a user search request. It is totally transparent to the user. So the concept is that index can be divided into multiple shards and each shard can be hosted on each data node. The placement of shards will be taken care of by Elasticsearch itself. If we don't specify the number of shards in the index creation URL, Elasticsearch will create five shards per index by default.

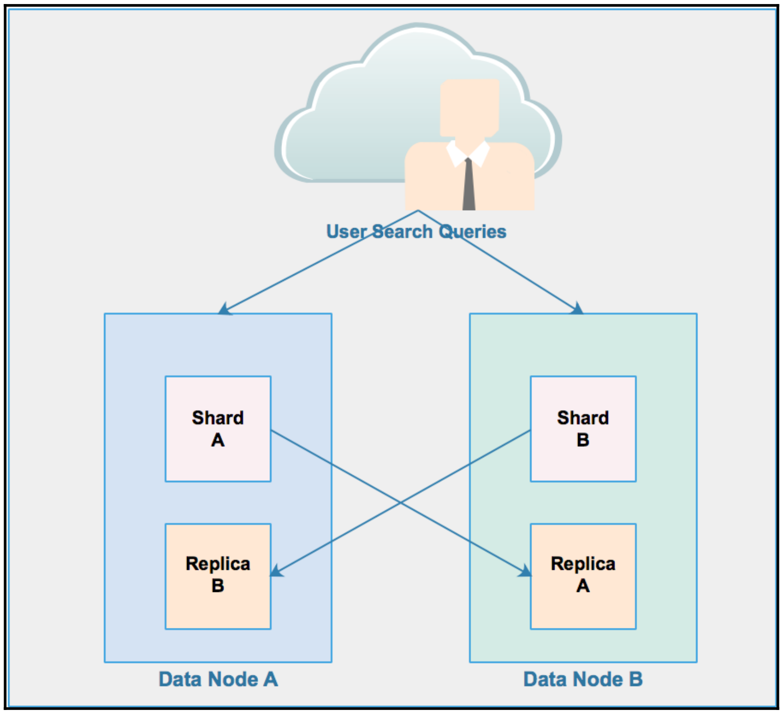
Replica shard

We have created index with one replica. It means Elasticsearch will create one copy (replica) of each shard and place each replica on separate data node other than the shard from which it is copied. So, now there are two shards, primary shard (the original shard) and replica shard (the copy of the primary shard). During a high volume of search activity, Elasticsearch can provide query results either from primary shards or from replica shards placed on different data nodes. This is how Elasticsearch increases the query throughput because each search query may go to different data nodes.

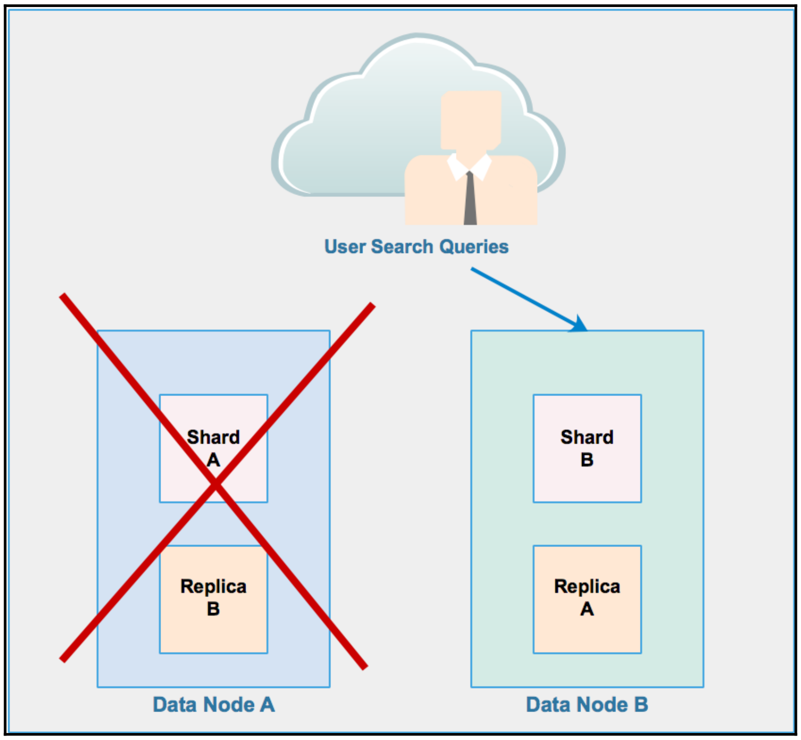
In the summary, both, primary shards and replica shards provide horizontal scalability and throughput. It scales out your search volume/throughput since searches can be executed on all replicas in parallel.

Elasticsearch is a distributed data store. It means data can be divided into multiple data nodes. For example, assume if we have just one data node and we keep on ingesting and indexing documents on the same data node, it may possible that after reaching out the hardware capacity of that node, we will not be to ingest documents. Hence, in order to accommodate more documents, we have to add another data node to the existing Elasticsearch cluster. If we add another data node, Elasticsearch will re-balance the shards to the newly created data node. So now, user search queries can be accommodated to both the data nodes. If we created one replica shard, then two replicas per shard will be created and placed on these two data nodes. Now, if one of the data nodes goes down, then still user search queries will be executed using just one data node.

This picture shows how user search queries are executed from both the data nodes:



The following picture shows that even if data nodes **A** goes down, still, user queries are executed from data node **B**:



Let's verify the newly created index:

```
curl -XGET 'localhost:9200/_cat/indices?v&amp;amp;amp;pretty'
```

We will get the following response:

```
health status index uuid pri rep docs.count docs.deleted store.size
pri.store.size
yellow open my_index 2MXqDHedSUqoV8Zyo0l-Lw 5 1 1 0 6.9kb 6.9kb
```

Let's understand the response:

- **Health:** This means the overall cluster health is yellow. There are three statuses: green, yellow, and red. The status `Green`" means the cluster is fully functional and everything looks good. The status "Yellow" means cluster is fully available but some of the replicas are not allocated yet. In our example, since we are using just one node and 5 shards and 1 replica each, Elasticsearch will not allocate all the replicas of all the shards on just one data node. The cluster status "Red" means cluster is partially available and some datasets are not available. The reason may be that the data node is down or something else.
- **Status**: `Open`. It means the cluster is open for business.
- **Index** : Index name. In our example, the index name is `my_index`.
- **Uuid** : This is unique index ID.
- **Pri** : Number of primary shards.
- **Rep** : Number of replica shards.
- **docs.count** : Total number of documents in an index.
- **docs.deleted** : Total number of documents deleted so far from an index.
- **store.size** : The store size taken by primary and replica shards.
- **pri.store.size** : The store size taken only by primary shards.

Ingest documents into index

The following `curl` command can be used to ingest a single document in the `my_index` index:

```
curl -X PUT 'localhost:9200/my_index/customer/1' -H 'Content-Type:
application/json' -d '
{
"name": "Angela Martin",
"birthdate": "1997-11-02",
"street": "63542 Times Square",
"city": "New York",
"state": "NY",
"zip": "10036",
"homePhone": "212-415-8929",
"cellPhone": "212-171-8187"
}'
```

In the previous command, we use a type called `customer`, which is a logical partition of an index. In a RDBMS analogy, a type is like a table in Elasticsearch.

Also, we used the number `1` after the type customer. It is an ID of a customer. If we omit it, then Elasticsearch will generate an arbitrary ID for the document.

We have multiple documents to be inserted into the `my_index` index. Inserting documents one by one in the command line is very tedious and time consuming. Hence, we can include all the documents in a file and do a bulk insert into `my_index`.

Create a `sample.json` file and include all the three documents:

```
{"index":{"_id":"1"}}

{"name": "Sam Taylor","birthdate": "1995-08-11","address":{"street": "155
rabbit Street","city": "San Francisco","state": "CA","zip":
"94107"},"contactPhone":[{"type": "home","number": "510-415-8929"},{"type":
"cell","number": "408-171-8187"}]}

{"index":{"_id":"2"}}
{"name": "Dan Lee","birthdate": "1970-01-25","address":{"street": "76336
1st Street","city": "Los Angeles","state": "CA","zip":
"90010"},"contactPhone":[{"type": "home","number": "323-892-5363"},{"type":
"cell","number": "213-978-1320"}]}

{"index":{"_id":"3"}}

{"name": "Angela Martin","birthdate": "1997-11-02","address":{"street":
"63542 Times Square","city": "New York","state": "NY","zip":
"10036"},"contactPhone":[{"type": "home","number": "212-415-8929"},{"type":
"cell","number": "212-171-8187"}]}
```

Bulk Insert

Let's ingest all the documents in the file `sample.json` at once using the following command:

```
curl -H 'Content-Type: application/json' -XPUT
'localhost:9200/my_index/customer/_bulk?pretty&amp;amp;amp;refresh' --data-
binary "@sample.json"
```

Let's verify all the records using our favorite browser. It will show all the three records:

```
http://localhost:9200/my_index/_search
```

Document search

Since we have documents in our `my_index` index, we can search these documents:

Find out a document where `city = " Los Angeles?` and query is as follows:

```
curl -XGET 'http://localhost:9200/my_index2/_search?pretty' -H 'Content-
Type: application/json' -d' {
"query": {
"match": {
"city": "Los Angeles" }
}
}'
```

Response:

```
{
"took" : 3,
"timed_out" : false,
"_shards" : {
"total" : 3,"successful" : 3,
"skipped" : 0,
"failed" : 0
},
"hits" : {
"total" : 1,
"max_score" : 1.3862944,
"hits" : [
{
"_index" : "my_index",
"_type" : "customer",
"_id" : "3",
"_score" : 1.3862944,
"_source" : {
"name" : "Dan Lee",
"birthdate" : "1970-01-25",
"street" : "76336 1st Street",
"city" : "Los Angeles",
"state" : "ca",
"postalCode" : "90010",
"homePhone" : "323-892-5363",
"cellPhone" : "213-978-1320"
}
}
]
}
}
```

If we analyze the response, we can see that the source section gives a back the document we were looking for. The document is in the index `my_index`, `"_type" : "customer"`, `"_id" : "3"`. Elasticsearch searches all `three _shards` successfully.

Under the `hits` section, there is a field called `_score`. Elasticsearch calculates the relevance frequency of each field within a document and stores it in index. It is called the weight of the document. This weight is calculated based on four important factors: term frequency, inverse frequency, document frequency, and field length frequency. This brings up another question, *How does Elasticsearch index a document?*

For example, we have the following four documents to index in Elasticsearch:

- I love Elasticsearch
- Elasticsearch is a document store
- HBase is key value data store
- I love HBase

Term	**Frequency**	**Document No.**
a	2	2
index	1	2
Elasticsearch	2	1,2
HBase	2	1
I	2	1,4
is	2	2,3
Key	1	3
love	2	1,4
store	2	2,3
value	1	3

When we ingest three documents in Elasticsearch, an Inverted Index is created, like the following.

Now, if we want to query term Elasticsearch, then only two documents need to be searched: 1 and 2. If we run another query to find *love Elasticsearch*, then three documents need to be searched (documents 1,2, and 4) before sending the results from only the first document.

Also, there is one more important concept we need to understand.

Meta fields

When we ingest a document into index, Elasticsearch adds a few meta fields to each index document. The following is the list of meta fields with reference to our sample `my_index`:

- `_index`: Name of the index. `my_index`.
- `_type`: Mapping type. "customer" (deprecated in version 6.0).
- `_uid`: `_type + _id` (deprecated in version 6.0).
- `_id`: `document_id` (1).
- `_all`: This concatenates all the fields of an index into a searchable string (deprecated in version 6.0).
- `_ttl`: Life a document before it can be automatically deleted.
- `_timestamp`: Provides a timestamp for a document.
- `_source`: This is an actual document, which is automatically indexed by default.

Mapping

In RDBMS analogy, mapping means defining a table schema. We always define a table structure, that is, column data types. In Elasticsearch, we also need to define the data type for each field. But then comes another question. Why did we not define it before when we ingested three documents into the `my_index` index? The answer is simple. Elasticsearch doesn't care. It is claimed that *Elasticsearch is a schema-less data model*.

If we don't define a mapping, Elasticsearch dynamically creates a mapping for us by defining all fields as text. Elasticsearch is intelligent enough to find out date fields to assign the `date` data type to them.

Let's find the existing dynamic mapping of index `my_index`:

```
curl -XGET 'localhost:9200/my_index2/_mapping/?pretty'
```

Response:

```
{
"my_index" : {
"mappings" : {
customer" : {
"properties" : {
"birthdate" : {
"type" : "date"
},
"cellPhone" : {
"type" : "text",
"fields" : {
"keyword" : {
"type" : "keyword",
"ignore_above" : 256
}
}
},
"city" : {
"type" : "text",
"fields" : {
"keyword" : {
"type" : "keyword",
"ignore_above" : 256
}
}
},
"homePhone" : {
"type" : "text",
"fields" : {
"keyword" : {
"type" : "keyword",
"ignore_above" : 256
}
}
},
"name" : {
"type" : "text",
"fields" : {
"keyword" : {
type" : "keyword",
"ignore_above" : 256
}
}
},
"postalCode" : {
"type" : "text",
"fields" : {
"keyword" : {
"type" : "keyword",
"ignore_above" : 256
}
}
},
"state" : {
"type" : "text",
"fields" : {
"keyword" : {
"type" : "keyword",
"ignore_above" : 256
}
}
},
"street" : {
"type" : "text",
"fields" : {
"keyword" : {
"type" : "keyword",
"ignore_above" : 256
}
}
}
}
}
}
}
}
```

Elasticsearch supports two mapping types as follows:

- Static mapping
- Dynamic mapping

Static mapping

In static mapping, we always know our data and we define the appropriate data type for each field. Static mapping has to be defined at the time of index creation.

Dynamic mapping

We have already used dynamic mapping for our documents in our example. Basically, we did not define any data type for any field. But when we ingested documents using `_Bulk` load, Elasticsearch transparently defined `text` and `date` data types appropriately for each field. Elasticsearch intelligently found our `Birthdate` as a date field and assigned the `date` data type to it.

Elasticsearch-supported data types

The following spreadsheet summarizes the available data types in Elasticsearch:

Common	**Complex**	**Geo**	**Specialized**
String	Array	Geo_Point	`ip`
Keyword	Object (single Json)	Geo_Shape	`completion`
Date	Nested (Json array)		`token_count`
Long			`join`
Short			`percolator`
Byte			`murmur3`
Double			
Float			
Boolean			
Binary			
Integer_range			
Float_range			
Long_range			
Double_range			
Date_range			

Most of the data types need no explanation. But the following are a few explanations for specific data types:

- **Geo-Point**: You can define latitude and longitude points here
- **Geo-Shape**: This is for defining shapes
- **Completion**: This data type is for defining auto completion of words.
- **Join**: To define parent/child relationships
- **Percolator**: This is for query-dsl
- **Murmur3**: During index time, it is for calculations hash value and store it into index

Mapping example

Let's re-create another index, `second_index`, which is similar to our `first_index` with static mapping, where we will define the data type of each field separately:

```
curl -XPUT localhost:9200/second_index -d '{
"mappings": {
"customer": {
"_source": {
"enabled": false
},
"properties": {
"name": {"type": "string", "store": true},
"birthdate": {"type": "string"},
"street": {"type": "string"},
"city": {"type": "date"},
"state": {"type": "string", "index": "no", "store": true}
"zip": {"type": "string", "index": "no", "store": true}}
}
}
}
```

Let's understand the preceding mapping. We disable the `_source` field for the customer type. It means, we get rid of the default behavior, where Elasticsearch stores and indexes the document by default. Now, since we have disabled it, we will deal with each and every field separately to decide whether that field should be indexed stored or both.

So, in the preceding example, we want to store only three fields, `name`, `state` and `zip`. Also, we don't want to index the `state` and `zip` fields. It means `state` and `zip` fields are not searchable.

Analyzer

We have already learned about an inverted index. We know that Elasticsearch stores a document into an inverted index. This transformation is known as analysis. This is required for a successful response of the index search query.

Also, many of the times, we need to use some kind of transformation before sending that document to Elasticsearch index. We may need to change the document to lowercase, stripping off HTML tags if any from the document, remove white space between two words, tokenize the fields based on delimiters, and so on.

Elasticsearch offers the following built-in analyzers:

- **Standard analyzer**: It is a default analyzer. This uses standard tokenizer to divide text. It normalizes tokens, lowercases tokens, and also removes unwanted tokens.
- **Simple analyzer**: This analyzer is composed of lowercase tokenizer.
- **Whitespace analyzer**: This uses the whitespace tokenizer to divide text at spaces.
- **Language analyzers**: Elasticsearch provides many language-specific analyzers such as English, and so on.
- **Fingerprint analyzer**: The fingerprint analyzer is a specialist analyzer. It creates a fingerprint, which can be used for duplicate detection.
- **Pattern analyzer**: The pattern analyzer uses a regular expression to split the text into terms.
- **Stop analyzer**: This uses letter tokenizer to divide text. It removes stop words from token streams. for example, all stop words like a, an, the, is and so on.
- **Keyword analyzer**: This analyzer tokenizes an entire stream as a single token. It can be used for zip code.
- **Character filter**: Prepare a string before it is tokenize. Example: remove html tags.
- **Tokenizer**: MUST have a single tokenizer. It's used to break up the string into individual terms or tokens.
- **Token filter**: Change, add or remove tokens. Stemmer is a token filter, it is used to get base of word, for example: learned, learning => learn

Example of atandard analyzer:

```
curl -XPOST 'localhost:9200/_analyze?pretty' -H 'Content-Type:
application/json' -d'
{
"analyzer": "standard",
"text": " 1. Today it's a Sunny-day, very Bright."
}'
```

Response:

```
[today, it's , a, sunny, day, very, bright ]
```

Example of simple analyzer:

```
curl -XPOST 'localhost:9200/_analyze?pretty' -H 'Content-Type:
application/json' -d'
{
"analyzer": "simple",
"text": " 1. Today it's a Sunny-day, very Bright."
}'
```

Response:

```
[today, it's , a, sunny, day, very, bright ]
```

Elasticsearch stack components

The Elasticsearch stack consists of following

- Beats
- Logstash
- Elasticsearch
- Kibana

Let's study them in brief.

Beats

Please refer to the following URL to know more about beats: `https://www.elastic.co/products/beats`.

Beats are lightweight data shippers. Beats are installed on to servers as agents. Their main function is collect the data and send it to either Logstash or Elasticsearch. We can configure beats to send data to Kafka topics also.

There are multiple beats. Each beat is meant for collecting specific datasets and metrics. The following are various types of Beats:

- **Filebeat**: For collection of log files. They simplify the collection, parsing, and visualization of common log formats down to a single command. Filebeat comes with internal modules (auditd, Apache, nginx, system, and MySQL).
- **Metricbeat**: For collection of metrics. They collect metrics from any systems and services, for example, memory, COU, and disk. Metricbeat is a lightweight way to send system and service statistics.
- **Packetbeat**: This is for collection of network data. Packetbeat is a lightweight network packet analyzer that sends data to Logstash or Elasticsearch.
- **Winlogbeat**: For collection of Windows event data. Winlogbeat live-streams Windows event logs to Elasticsearch and Logstash.
- **Auditbeat**: For collection of audit data. Auditbeat collects audit framework data.
- **Heartbeat**: For collection of uptime monitoring data. Heartbeat ships this information and response time to Elasticsearch.

Installation of Filebeat:

```
$wget
https://artifacts.elastic.co/downloads/beats/filebeat/filebeat-6.1.2-x86_64
.rpm
$ sudo rpm --install filebeat-6.1.2-x86_64.rpm
sudo /bin/systemctl daemon-reload
sudo /bin/systemctl enable filebeat.service
```

Logstash

Logstash is a lightweight, open source data processing pipeline. It allows collecting data from a wide variety of sources, transforming it on the fly, and sending it to any desired destination.

It is most often used as a data pipeline for Elasticsearch, a popular analytics and search engine. Logstash is a popular choice for loading data into Elasticsearch because of its tight integration, powerful log processing capabilities, and over 200 prebuilt open source plugins that can help you get your data indexed the way you want it.

The following is a structure of `Logstash.conf`:

```
input {
...
}
filter {
...
}
output {
..
}
```

Installation of Logstash:

```
$ wget https://artifacts.elastic.co/downloads/logstash/logstash-6.1.2.rpm
$ sudo rpm --install logstash-6.1.2.rpm
$ sudo /bin/systemctl daemon-reload
$ sudo systemctl start logstash.service
```

Kibana

Kibana is an open-source data visualization and exploration tool used for log and time series analytics, application monitoring, and operational intelligence use cases. Kibana offers tight integration with Elasticsearch, a popular analytics and search engine, which makes Kibana the default choice for visualizing data stored in Elasticsearch. Kibana is also popular due to its powerful and easy-to-use features such as histograms, line graphs, pie charts, heat maps, and built-in geospatial support.

Installation of Kibana:

```
$wget https://artifacts.elastic.co/downloads/kibana/kibana-6.1.2-x86_64.rpm
$ sudo rpm --install kibana-6.1.2-x86_64.rpm
sudo /bin/systemctl daemon-reload
sudo /bin/systemctl enable kibana.service
```

Use case

Let's assume that we have an application deployed on an application server. That application is logging on to an access log. Then how can we analyze this access log using a dashboard? We would like to create a real-time visualization of the following info:

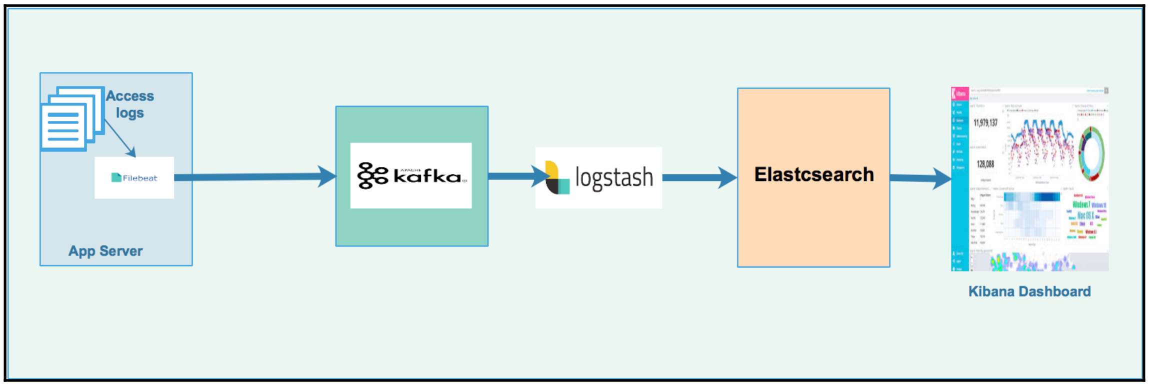
- Number of various response codes
- Total number of responses
- List of IPs

Proposed technology stack:

- **Filebeat**: To read access log and write to Kafka topic
- **Kafka:** Message queues and o buffer message
- **Logstash:** To pull messages from Kafka and write to Elasticsearch index
- **Elasticsearch**: For indexing messages
- **Kibana**: Dashboard visualization

In order to solve this problem, we install filebeat on Appserver. Filebeat will read each line from the access log and write to the kafka topic in real time. Messages will be buffered in Kafka. Logstash will pull messages from the Kafka topic and write to Elasticsearch.

Kibana will create real-time streaming dashboard by reading messages from Elasticsearch index. The following is the architecture of our use case:



Here is the step-by-step code sample, `Acccss.log`:

```
127.0.0.1 - - [21/Mar/2017:13:52:29 -0400] "GET /web-
portal/performance/js/common-functions.js HTTP/1.1" 200 3558
127.0.0.1 - - [21/Mar/2017:13:52:30 -0400] "GET /web-
portal/performance/js/sitespeed-functions.js HTTP/1.1" 200 13068
127.0.0.1 - - [21/Mar/2017:13:52:34 -0400] "GET /web-portal/img/app2-icon-
dark.png HTTP/1.1" 200 4939
127.0.0.1 - - [21/Mar/2017:13:52:43 -0400] "GET /web-search-
service/service/performanceTest/release/list HTTP/1.1" 200 186
127.0.0.1 - - [21/Mar/2017:13:52:44 -0400] "GET /web-
portal/performance/fonts/opan-sans/OpenSans-Light-webfont.woff HTTP/1.1"
200 22248
127.0.0.1 - - [21/Mar/2017:13:52:44 -0400] "GET /web-
portal/performance/img/icon/tile-actions.png HTTP/1.1" 200 100
127.0.0.1 - - [21/Mar/2017:13:52:44 -0400] "GET /web-
portal/performance/fonts/fontawesome/fontawesome-webfont.woff?v=4.0.3
HTTP/1.1" 200 44432
```

The following is the complete `Filebeat.ymal`:

In the Kafka output section, we have mentioned Kafka broker details. `output.kafka`:

```
# initial brokers for reading cluster metadata
hosts: ["localhost:6667"]
```

The following is the complete `Filebeat.ymal`:

```
###################### Filebeat Configuration Example
#########################
# This file is an example configuration file highlighting only the most
common
# options. The filebeat.reference.yml file from the same directory contains
all the
# supported options with more comments. You can use it as a reference.
#
# You can find the full configuration reference here:
# https://www.elastic.co/guide/en/beats/filebeat/index.html
# For more available modules and options, please see the
filebeat.reference.yml sample
# configuration file.
#====================== Filebeat prospectors=======================
filebeat.prospectors:
# Each - is a prospector. Most options can be set at the prospector level,
so
# you can use different prospectors for various configurations.
# Below are the prospector specific configurations.
- type: log
```

```
# Change to true to enable this prospector configuration.
enabled: true
# Paths that should be crawled and fetched. Glob based paths.
paths:
- /var/log/myapp/*.log
#- c:programdataelasticsearchlogs*
#json.keys_under_root: true
#json.add_error_key: true
# Exclude lines. A list of regular expressions to match. It drops the lines
that are
# matching any regular expression from the list.
#exclude_lines: ['^DBG']
# Include lines. A list of regular expressions to match. It exports the
lines that are
# matching any regular expression from the list.
#include_lines: ['^ERR', '^WARN']
# Exclude files. A list of regular expressions to match. Filebeat drops the
files that
# are matching any regular expression from the list. By default, no files
are dropped.
#exclude_files: ['.gz$']
# Optional additional fields. These fields can be freely picked
# to add additional information to the crawled log files for filtering
#fields:
# level: debug
# review: 1
fields:
app: myapp
env: dev
dc: gce
### Multiline options
# Mutiline can be used for log messages spanning multiple lines. This is
common
# for Java Stack Traces or C-Line Continuation
# The regexp Pattern that has to be matched. The example pattern matches
all lines starting with [#multiline.pattern: ^[
# Defines if the pattern set under pattern should be negated or not.
Default is false.
#multiline.negate: false
# Match can be set to "after" or "before". It is used to define if lines
should be append to a pattern
# that was (not) matched before or after or as long as a pattern is not
matched based on negate.
# Note: After is the equivalent to previous and before is the equivalent to
to next in Logstash
#multiline.match: after
#============================= Filebeat modules
===============================
filebeat.config.modules:
# Glob pattern for configuration loading
path: ${path.config}/modules.d/*.yml
# Set to true to enable config reloading
reload.enabled: false
# Period on which files under path should be checked for changes
#reload.period: 10s
#==================== Elasticsearch template setting
==========================
setup.template.settings:
index.number_of_shards: 3
#index.codec: best_compression
#_source.enabled: false
#================================ General
=====================================
# The name of the shipper that publishes the network data. It can be used
to group
# all the transactions sent by a single shipper in the web interface.
#name:
# The tags of the shipper are included in their own field with each
# transaction published.
#tags: ["service-X", "web-tier"]
# Optional fields that you can specify to add additional information to the
# output.
#fields:
# env: staging
#============================== Dashboards
=====================================
# These settings control loading the sample dashboards to the Kibana index.
Loading
# the dashboards is disabled by default and can be enabled either by
setting the
# options here, or by using the `-setup` CLI flag or the `setup` command.
#setup.dashboards.enabled: false
# The URL from where to download the dashboards archive. By default this
URL
# has a value which is computed based on the Beat name and version. For
released
# versions, this URL points to the dashboard archive on the
artifacts.elastic.co
# website.
#setup.dashboards.url:
#============================== Kibana
=====================================
# Starting with Beats version 6.0.0, the dashboards are loaded via the
Kibana API.
# This requires a Kibana endpoint configuration.
setup.kibana:
```

```
# Kibana Host
# Scheme and port can be left out and will be set to the default (http and
5601)
# In case you specify and additional path, the scheme is required:
http://localhost:5601/path
# IPv6 addresses should always be defined as: https://[2001:db8::1]:5601
#host: "localhost:5601"
#============================== Elastic Cloud
==================================
# These settings simplify using filebeat with the Elastic Cloud
(https://cloud.elastic.co/).
# The cloud.id setting overwrites the `output.elasticsearch.hosts` and
# `setup.kibana.host` options.
# You can find the `cloud.id` in the Elastic Cloud web UI.
#cloud.id:
# The cloud.auth setting overwrites the `output.elasticsearch.username` and
# `output.elasticsearch.password` settings. The format is `<user>:<pass>`.
#cloud.auth:
#================================= Outputs
=====================================
# Configure what output to use when sending the data collected by the beat.
#-----------------------------------Kafka Output---------------------------
----
output.kafka:
# initial brokers for reading cluster metadata
hosts: ["localhost:6667"]
# message topic selection + partitioning
topic: logs-topic
partition.round_robin:
reachable_only: false
required_acks: 1
compression: gzip
max_message_bytes: 1000000
#-------------------------- Elasticsearch output --------------------------
----
#output.elasticsearch:
# Array of hosts to connect to.
#hosts: ["localhost:9200"]
# Optional protocol and basic auth credentials.
#protocol: "https"
#username: "elastic"
#password: "changeme"
#----------------------------- Logstash output ----------------------------
----#output.logstash:
# The Logstash hosts
#hosts: ["localhost:5044"]
# Optional SSL. By default is off.
# List of root certificates for HTTPS server verifications
```

```
#ssl.certificate_authorities: ["/etc/pki/root/ca.pem"]
# Certificate for SSL client authentication
#ssl.certificate: "/etc/pki/client/cert.pem"
# Client Certificate Key
#ssl.key: "/etc/pki/client/cert.key"
#================================ Logging
=====================================
# Sets log level. The default log level is info.
# Available log levels are: error, warning, info, debug
logging.level: debug
# At debug level, you can selectively enable logging only for some
components.
# To enable all selectors use ["*"]. Examples of other selectors are
"beat",
# "publish", "service".
#logging.selectors: ["*"]
#============================== Xpack Monitoring
===============================
# filebeat can export internal metrics to a central Elasticsearch
monitoring
# cluster. This requires xpack monitoring to be enabled in Elasticsearch.
The
# reporting is disabled by default.
# Set to true to enable the monitoring reporter.
#xpack.monitoring.enabled: false
# Uncomment to send the metrics to Elasticsearch. Most settings from the
# Elasticsearch output are accepted here as well. Any setting that is not
set is
# automatically inherited from the Elasticsearch output configuration, so
if you
# have the Elasticsearch output configured, you can simply uncomment the
# the following line.
#xpack.monitoring.elasticsearch:
```

We have to create a `logs-topic` topic in Kafka before we start ingesting messages into it. It is assumed that we have already installed Kafka on the server. Please refer to `Chapter 2`, *Hadoop Life Cycle Management* to read more about Kafka.

Create logs-topic:

```
bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-
factor 1 --partitions 1 --topic logs-topic
```

The following is the `Logstash.conf` (to read messages from Kafka and push them to Elasticseach):

```
input
{
kafka
{
bootstrap_servers => "127.0.0.1:6667"
group_id => "logstash_logs"
topics => ["logs-topic"]
consumer_threads => 1
type => "kafka_logs"
}
}
filter {
if [type] == "kafka_logs"
{
json {
source => "message"
}
grok {
match => { "message" => "%{IP:ip} - - [%{GREEDYDATA:log_timestamp}]
%{GREEDYDATA:middle} %{NUMBER:status} %{NUMBER:bytes}" }
}
mutate {
add_field => {
"App" => "%{[fields][app]}"
}
}
}
}
output {
if [App] == "myapp"
{
elasticsearch
{
action => "index"
codec => "plain"
hosts => ["http://127.0.0.1:9200"]
index => "log_index-%{+YYYY-MM-dd}"
}
}
}
```

In the Kafka section, we've mentioned the following things:

```
Kafka bootstrap_servers => "127.0.0.1:6667"
Kafka topics => ["logs-topic"]
```

In the filter section, we are converting each message into JSON format. After that, we are parsing each message and dividing it into multiple fields such as `ip`, `timestamp`, and `status`. Also, we add the application name `myapp` field to each message.

In the output section, we are writing each message to Elasticsearch. The index name is `log_index-YYYY-MM-dd`.

Summary

In this chapter, you looked at the basic concepts and components of an Elasticsearch cluster.

After this, we discussed how Elasticsearch indexes a document using inverted index. We also discussed mapping and analysis techniques. We learned how we can denormalize an event before ingesting into Elasticsearch. We discussed how Elasticsearch uses horizontal scalability and throughput. After learning about Elasticstack components such as Beats, Logstash, and Kibana, we handled a live use case, where we demonstrated how access log events can be ingested into Kafka using Filebeat. We developed a code to pull messages from Kafka and ingest into Elasticsearch using Logstash. At the end, we learned data visualization using Kibana.

In the next chapter, we will see how to build analytics to design data visualization solutions that drive business decisions.

9
Designing Data Visualization Solutions

Once we have the data living in the Hadoop ecosystem and it's been processed, the next logical step is to build the analytics that drive the business decisions.

In this chapter, we will learn the following topics:

- Data visualization
- Apache Druid
- Apache Superset

Data visualization

Data visualization is the process of understanding the relationships between various entities in the raw data via graphical means. This is a very powerful technique because it enables end users to get the message in a very easy form without even knowing anything about the underlying data.

Data visualization plays a very important role in visual communication of insights from big data. Its both an art and a science, and requires some effort in terms of understanding the data; at the same time we need some understanding of the target audience as well.

So far, we have seen that any type of data can be stored in the **Hadoop filesystem** (**HDFS**). In order to convert complex data structures into a visual form, we need to understand the standard techniques that are used to represent the data.

In data visualization, the message is conveyed to the end users in the form of graphics which can be in 1D, 2D, 3D, or even higher dimensions. This purely depends on the meaning we are trying to convey.

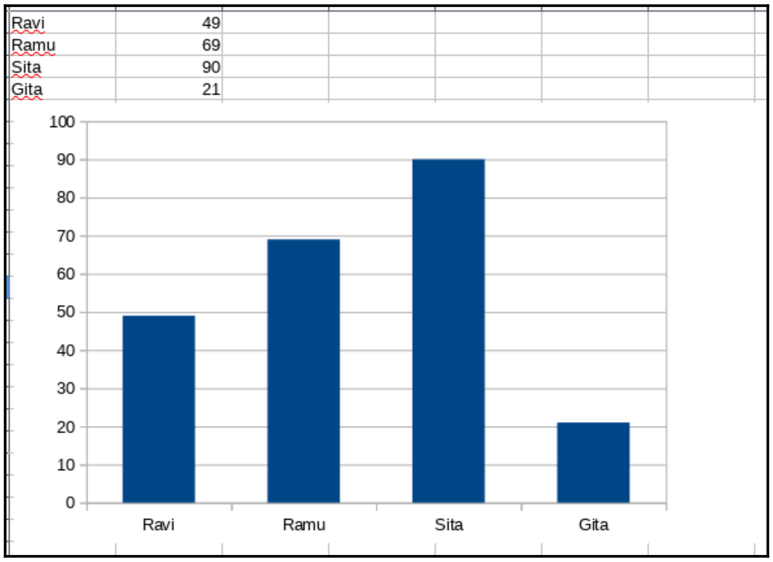
Let's take a look at the standard graphics that are used to convey visual messages to users:

- Bar/column chart
- Line/area chart
- Pie chart
- Radar chart
- Scatter/bubble chart
- Tag Cloud
- Bubble chart

Bar/column chart

This is a 2D graphical representation of data where the data points are shown as vertical/horizontal bars. Each bar represents one data point. When there is no time dimension involved with reference to the data points, the order in which these points are shown might not make any difference. When we deal with time series data for representing bar charts, we generally follow the chronological order of display along the X (horizontal) axis.

Let's take a look at a sample chart that is generated with four data points. The data represents the amount each user has:



Ravi	49
Ramu	69
Sita	90
Gita	21

Interpretation: The graph has both text data in rows and columns, and also visuals. If you observe carefully, the textual data is smaller in size and has only four records. But the visual graphic conveys the message straightaway without knowing anything about the data.

The message the graph conveys is that:

- Sita has more money than everyone
- Gita has the least money

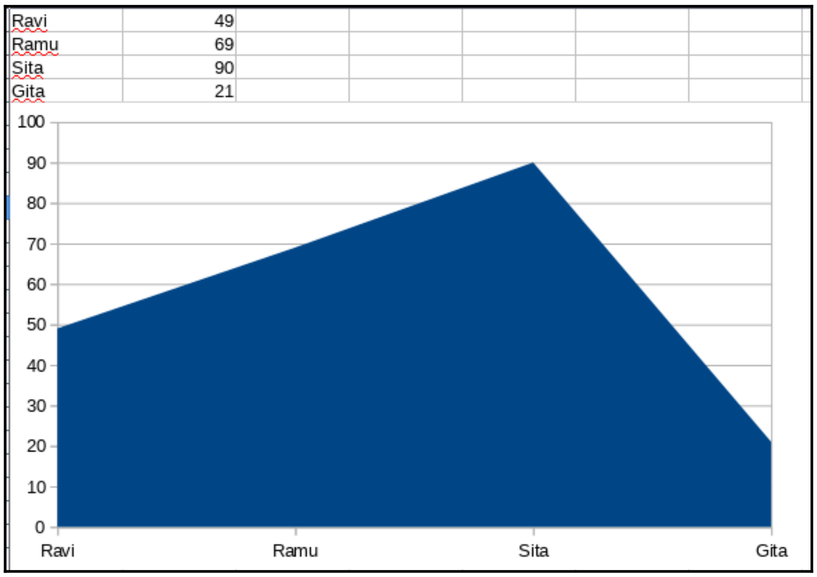
Other interpretations are also possible. They are left to the reader.

Line/area chart

This is also typically a 2D chart where each data point is represented as a point on canvas and all these points belonging to the same dataset are connected using a line. This chart becomes an area chart when the region from the horizontal/vertical axis is completely covered up to the line.

There can be more than one line in the same graph, which indicates multiple series of data for the same entities.

Let's take a look at the sample of this area chart based on the same data as before:



Ravi	49
Ramu	69
Sita	90
Gita	21

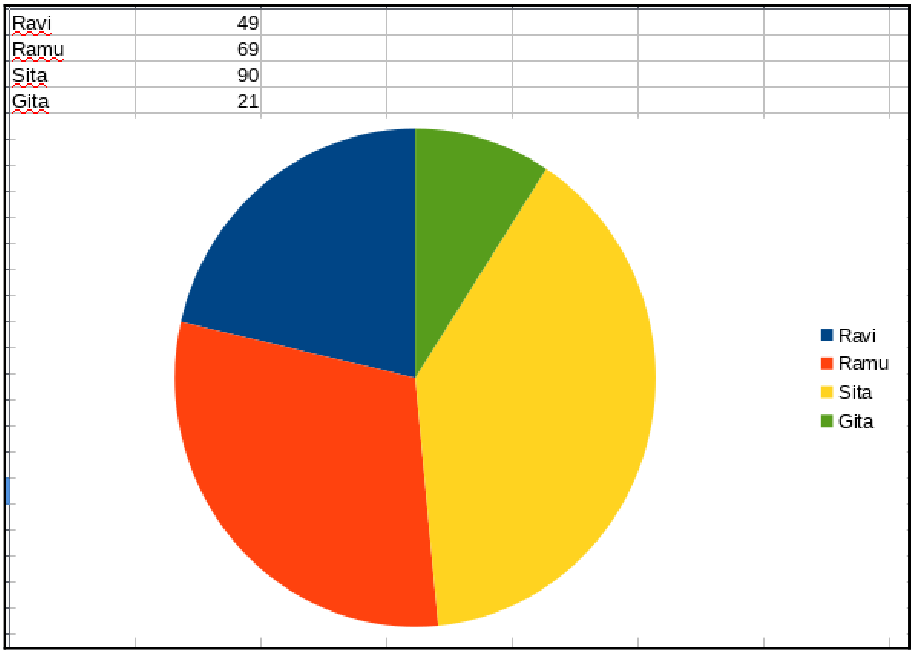
These are the properties of the chart:

- The x axis has the list of all the people
- The y axis indicates the amount from **0** to **100**
- Points are drawn on the graph at four places, corresponding to the values in tabular form
- Points are connected with straight lines
- The area is filled below the line to make it an area chart

Pie chart

This is also a 2D chart drawn as multiple sectors in a circle. This chart is useful when we want to highlight the relative importance of all the data points.

Let's take a look at the example chart that is drawn with the same dataset as before to understand it better:



Ravi	49
Ramu	69
Sita	90
Gita	21

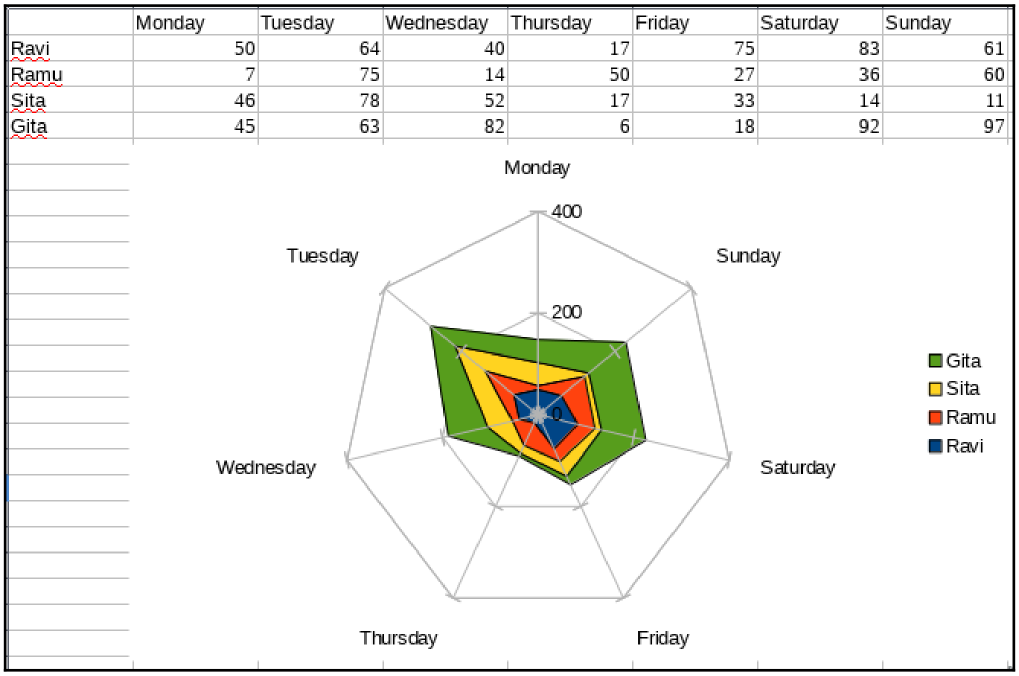
As you can see, it's easy to understand the relative importance of the amounts owned by each of the persons using this chart.

The conclusions that are drawn are similar to the previous charts. But the graph is a simple circle and there are no multiple dimensions here to burden the user.

Radar chart

This is also a 2D graphic where the data axes are the edges of equidistant sectors (like a pie chart's edges). This graph is useful when there are multiple dimensions in which we want to understand the relative significance of each data point.

To understand this graph better, let's take a look at this sample data and the graphic:



	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
Ravi	50	64	40	17	75	83	61
Ramu	7	75	14	50	27	36	60
Sita	46	78	52	17	33	14	11
Gita	45	63	82	6	18	92	97

The data consists of eight columns:

- **First column**: List of all users
- **Second to Eighth column**: Days in a week and the dollars owned by each person on that day

We want to draw a graph that shows us the following things:

- Total dollars per day
- Dollars owned by every person every day

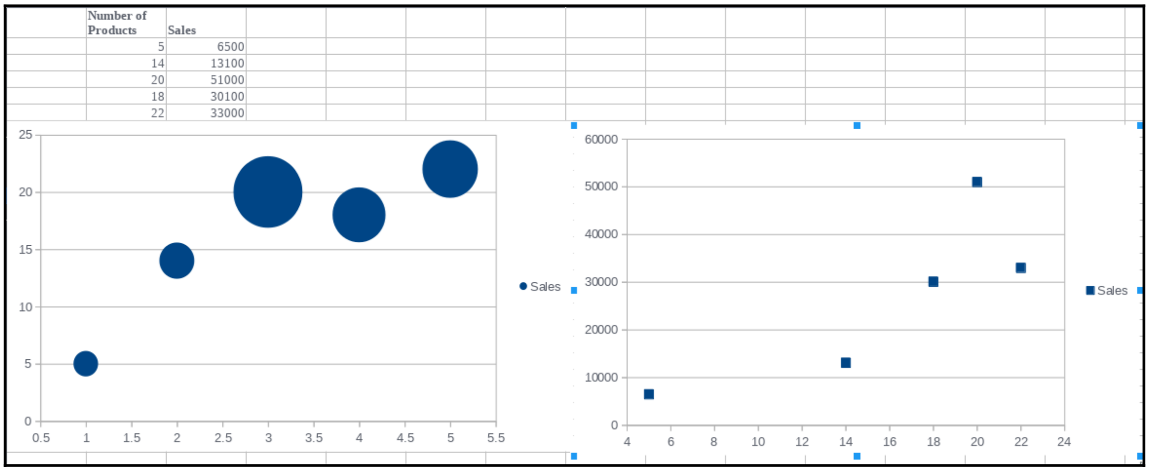
We have drawn all this information in the radar chart, where the axes are the sectors (days) and are capped at a maximum value of **400**. Each user's worth is drawn one on top of another so that we will know the total worth instead of relative worth (this is similar to area stacking).

Scatter/bubble chart

A scatter chart can be a multi-dimensional graphic. This is one of the simpler graphics to understand as we render each data point on the canvas corresponding to the numeric value along the axis. This graph is useful to understand the relative importance of each point along the axes.

A bubble chart is a variant of a scatter chart, where the points on the canvas show the values as big bubbles (to signify their importance).

Let's take a look at both these graphics with this example:



Number of Products	Sales
5	6500
14	13100
20	51000
18	30100
22	33000

The graphic on the left-hand side is a bubble chart and the right one is a scatter plot.

Let's take a look at the data and the charts that are generated.

The input data:

- Consists of five rows, whereas we have **Sales** and **Number of Products** in columns

With a bubble chart:

- The *y* axis shows the number of products
- The *x* axis is just positional and doesn't reflect the value from the input data
- Each point on the canvas shows the sales corresponding to the number of products

With the scatter chart:

- The *y* axis shows the sales done
- The *x* axis shows the products sold
- Each point on the canvas shows each row in the input

Other charts

There are many other types of graphics possible that are not covered in this section but are worth exploring on the `https://d3js.org` website. This will give you an understanding of how data can be represented to convey a very good message to the users.

Practical data visualization in Hadoop

Hadoop has a rich ecosystem of data sources and applications that help us build rich visualizations. In the coming sections, we will understand two such applications:

- Apache Druid
- Apache Superset

We will also learn how to use Apache Superset with data in RDBMSes such as MySQL.

Apache Druid

Apache Druid is a distributed, high-performance columnar store. Its official website is `https://druid.io`.

Druid allows us to store both real-time and historical data that is time series in nature. It also provides fast data aggregation and flexible data exploration. The architecture supports storing trillions of data points on petabyte sizes.

In order to understand more about the Druid architecture, please refer to this white paper at `http://static.druid.io/docs/druid.pdf`.

Druid components

Let's take a quick look at the different components of the Druid cluster:

Component	**Description**
Druid Broker	These are the nodes that are aware of where the data lies in the cluster. These nodes are contacted by the applications/clients to get the data within Druid.
Druid Coordinator	These nodes manage the data (they load, drop, and load-balance it) on the historical nodes.
Druid Overlord	This component is responsible for accepting tasks and returning the statuses of the tasks.
Druid Router	These nodes are needed when the data volume is in terabytes or higher range. These nodes route the requests to the brokers.
Druid Historical	These nodes store immutable segments and are the backbone of the Druid cluster. They serve load segments, drop segments, and serve queries on segments' requests.

Other required components

The following table presents a couple of other required components:

Component	Description
Zookeeper	Apache Zookeeper is a highly reliable distributed coordination service
Metadata Storage	MySQL and PostgreSQL are the popular RDBMSes used to keep track of all segments, supervisors, tasks, and configurations

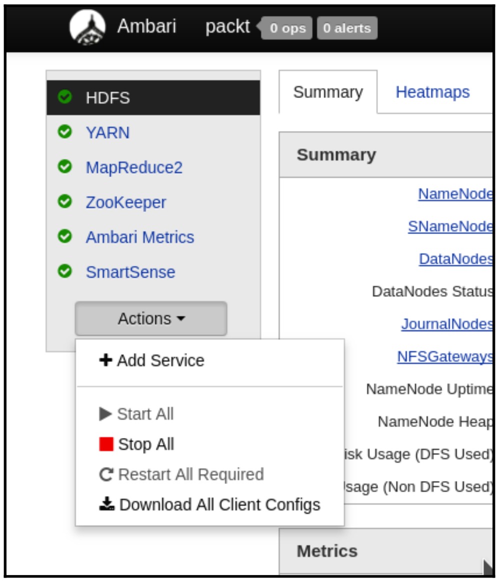
Apache Druid installation

Apache Druid can be installed either in standalone mode or as part of a Hadoop cluster. In this section, we will see how to install Druid via Apache Ambari.

Add service

First, we invoke the **Actions** drop-down below the list of services in the Hadoop cluster.

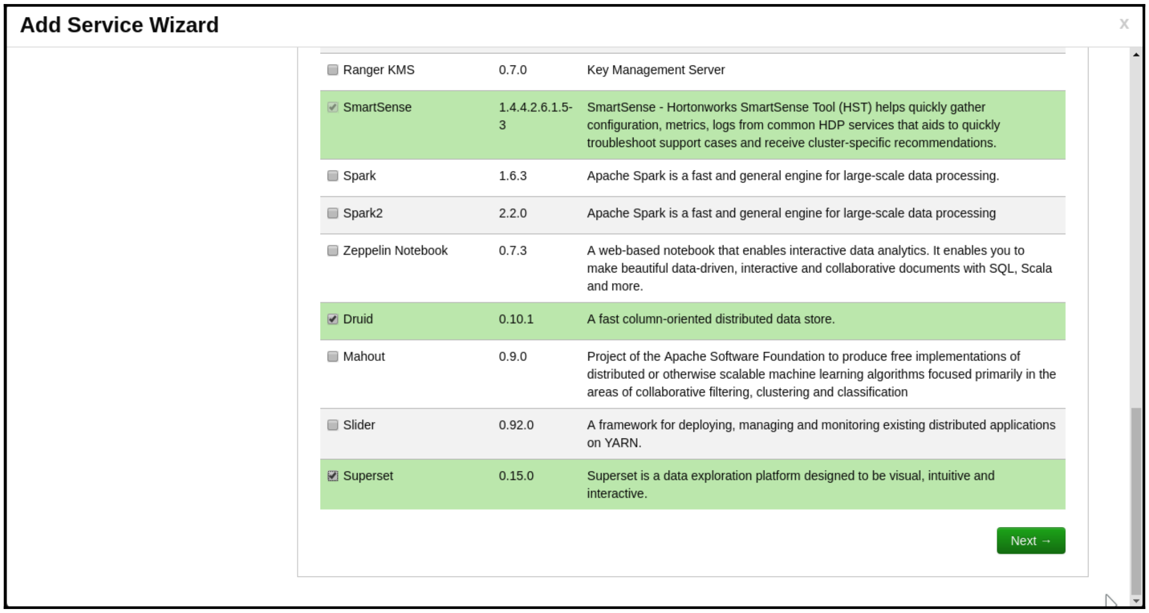
The screen looks like this:



Select Druid and Superset

In this setup, we will install both Druid and Superset at the same time. Superset is the visualization application that we will learn about in the next step.

The selection screen looks like this:



Add Service Wizard

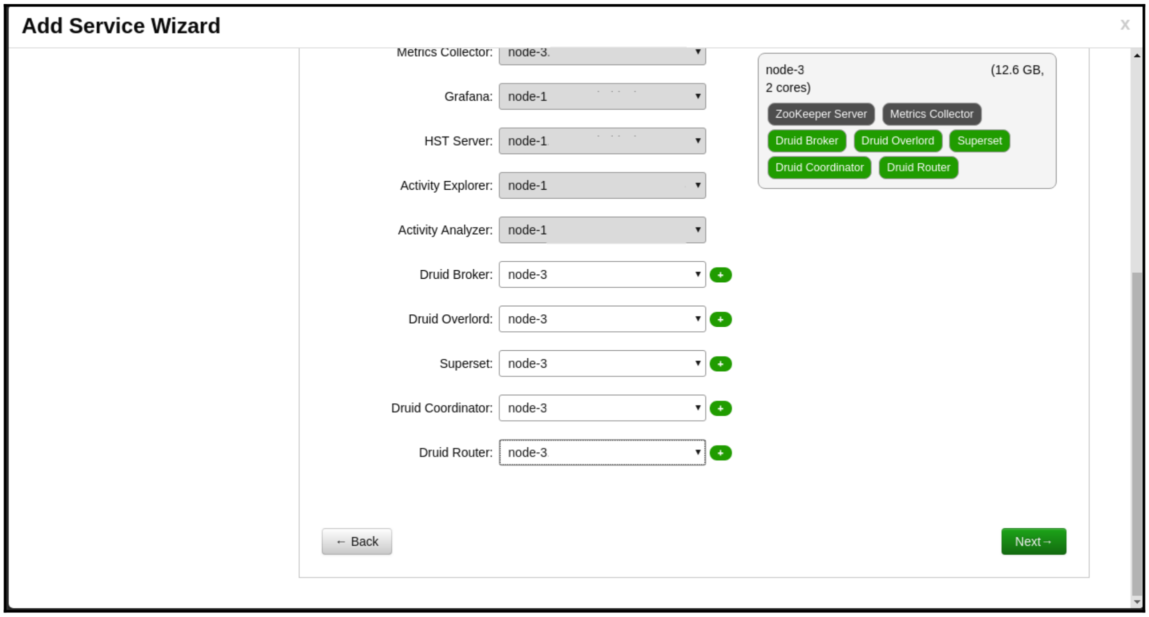
☐ Ranger KMS	0.7.0	Key Management Server
✓ SmartSense	1.4.4.2.6.1.5-3	SmartSense - Hortonworks SmartSense Tool (HST) helps quickly gather configuration, metrics, logs from common HDP services that aids to quickly troubleshoot support cases and receive cluster-specific recommendations.
☐ Spark	1.6.3	Apache Spark is a fast and general engine for large-scale data processing.
☐ Spark2	2.2.0	Apache Spark is a fast and general engine for large-scale data processing
☐ Zeppelin Notebook	0.7.3	A web-based notebook that enables interactive data analytics. It enables you to make beautiful data-driven, interactive and collaborative documents with SQL, Scala and more.
✓ Druid	0.10.1	A fast column-oriented distributed data store.
☐ Mahout	0.9.0	Project of the Apache Software Foundation to produce free implementations of distributed or otherwise scalable machine learning algorithms focused primarily in the areas of collaborative filtering, clustering and classification
☐ Slider	0.92.0	A framework for deploying, managing and monitoring existing distributed applications on YARN.
✓ Superset	0.15.0	Superset is a data exploration platform designed to be visual, intuitive and interactive.

Click on **Next** when both the services are selected.

Service placement on servers

In this step, we will be given a choice to select the servers on which the application has to be installed. I have selected node 3 for this purpose. You can select any node you wish.

The screen looks something like this:



Add Service Wizard

Metrics Collector: node-3

Grafana: node-1

HST Server: node-1

Activity Explorer: node-1

Activity Analyzer: node-1

Druid Broker: node-3

Druid Overlord: node-3

Superset: node-3

Druid Coordinator: node-3

Druid Router: node-3

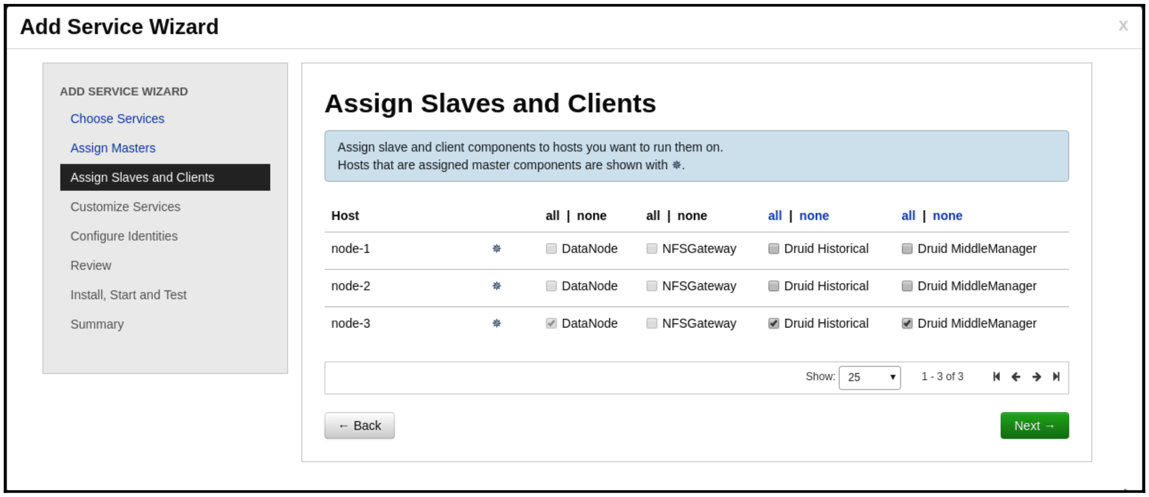
node-3 (12.6 GB, 2 cores)

ZooKeeper Server, Metrics Collector, Druid Broker, Druid Overlord, Superset, Druid Coordinator, Druid Router

Click on **Next** when when the changes are done.

Choose Slaves and Clients

Here, we are given a choice to select the nodes on which we need the Slaves and Clients for the installed components. I have left the options that are already selected for me:



Add Service Wizard

ADD SERVICE WIZARD
- Choose Services
- Assign Masters
- Assign Slaves and Clients
- Customize Services
- Configure Identities
- Review
- Install, Start and Test
- Summary

Assign Slaves and Clients

Assign slave and client components to hosts you want to run them on.
Hosts that are assigned master components are shown with ✱.

Host		all | none	all | none	all | none	all | none
node-1	✱	☐ DataNode	☐ NFSGateway	☐ Druid Historical	☐ Druid MiddleManager
node-2	✱	☐ DataNode	☐ NFSGateway	☐ Druid Historical	☐ Druid MiddleManager
node-3	✱	✓ DataNode	☐ NFSGateway	✓ Druid Historical	✓ Druid MiddleManager

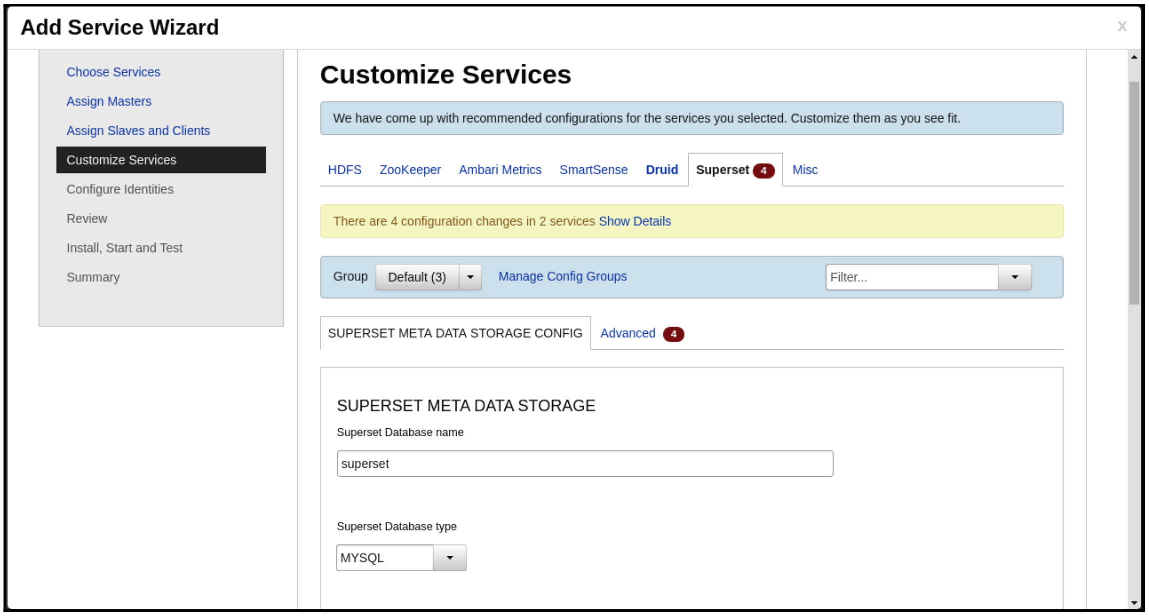
Show: 25 1 - 3 of 3

← Back Next →

Service configurations

In this step, we need to select the databases, usernames, and passwords for the metadata store used by the Druid and Superset applications. Feel free to choose the default ones. I have given MySQL as the backend store for both of them.

The screen looks like this:



Customize Services

We have come up with recommended configurations for the services you selected. Customize them as you see fit.

There are 4 configuration changes in 2 services Show Details

SUPERSET META DATA STORAGE

Superset Database name: superset

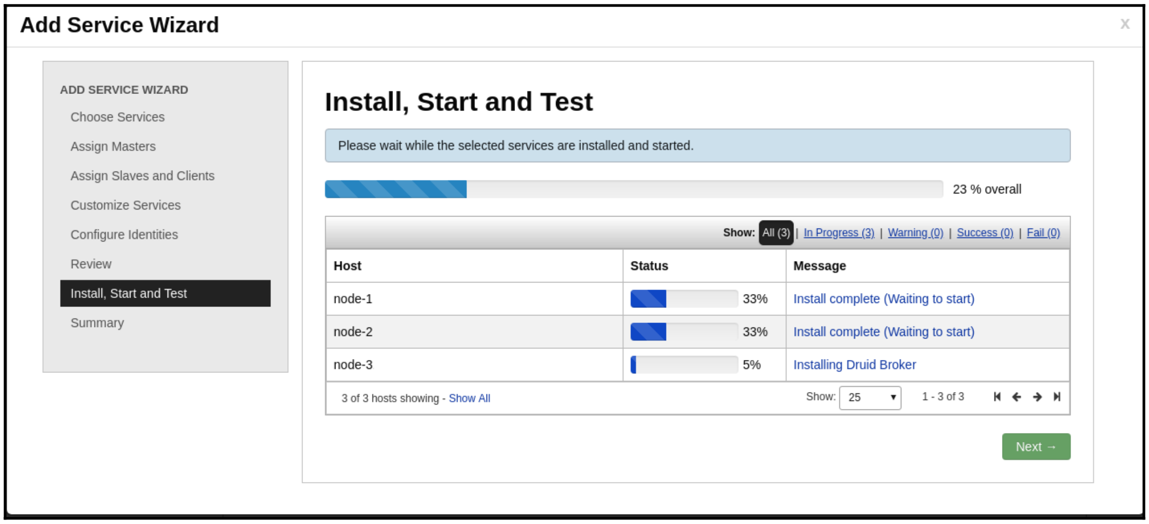
Superset Database type: MYSQL

Once the changes look good, click on the **Next** button at the bottom of the screen.

Service installation

In this step, the applications will be installed automatically and the status will be shown at the end of the plan.

Click on **Next** once the installation is complete. Changes to the current screen look like this:



Install, Start and Test

Please wait while the selected services are installed and started.

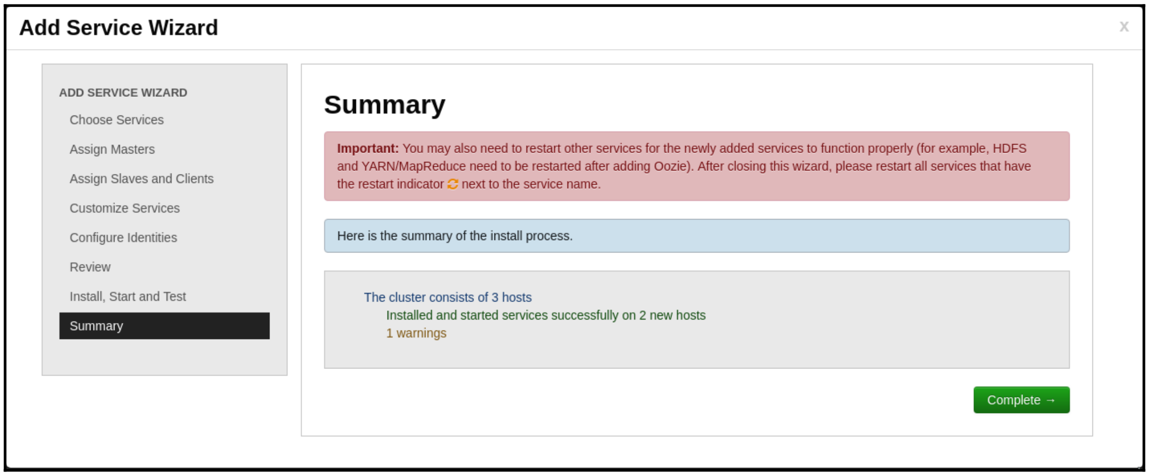
23 % overall

Host	Status	Message
node-1	33%	Install complete (Waiting to start)
node-2	33%	Install complete (Waiting to start)
node-3	5%	Installing Druid Broker

3 of 3 hosts showing - Show All

Installation summary

Once everything is successfully completed, we are shown a summary of what has been done. Click on **Complete** when done:



Summary

Important: You may also need to restart other services for the newly added services to function properly (for example, HDFS and YARN/MapReduce need to be restarted after adding Oozie). After closing this wizard, please restart all services that have the restart indicator next to the service name.

Here is the summary of the install process.

The cluster consists of 3 hosts
Installed and started services successfully on 2 new hosts
1 warnings

Sample data ingestion into Druid

Once we have all the Druid-related applications running in our Hadoop cluster, we need a sample dataset that we must load in order to run some analytics tasks.

Let's see how to load sample data. Download the Druid archive from the internet:

```
[druid@node-3 ~$ curl -O
http://static.druid.io/artifacts/releases/druid-0.12.0-bin.tar.gz
% Total % Received % Xferd Average Speed Time Time Time Current
                               Dload Upload Total Spent Left Speed
100 222M 100 222M 0 0 1500k 0 0:02:32 0:02:32 --:--:-- 594k
```

Extract the archive:

```
[druid@node-3 ~$ tar -xzf druid-0.12.0-bin.tar.gz
```

Copy the sample Wikipedia data to Hadoop:

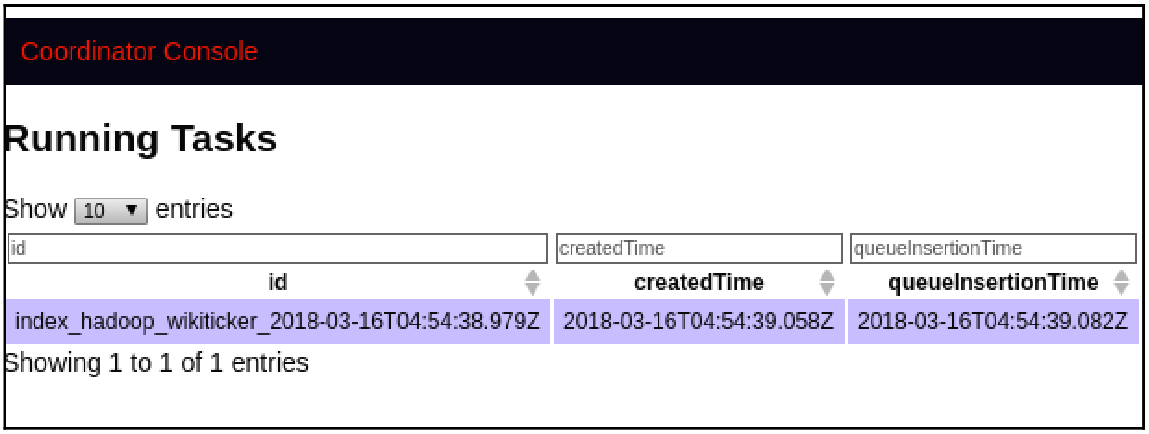
```
[druid@node-3 ~]$ cd druid-0.12.0
[druid@node-3 ~/druid-0.12.0]$ hadoop fs -mkdir /user/druid/quickstart
[druid@node-3 ~/druid-0.12.0]$ hadoop fs -put
quickstart/wikiticker-2015-09-12-sampled.json.gz /user/druid/quickstart/
```

Submit the import request:

```
[druid@node-3 druid-0.12.0]$ curl -X 'POST' -H 'Content-
Type:application/json' -d @quickstart/wikiticker-index.json
localhost:8090/druid/indexer/v1/task;echo
{"task":"index_hadoop_wikiticker_2018-03-16T04:54:38.979Z"}
```

After this step, Druid will automatically import the data into the Druid cluster and the progress can be seen in the overlord console.

The interface is accessible via `http://<overlord-ip>:8090/console.html`. The screen looks like this:


Coordinator Console

Running Tasks

Show 10 entries

id	createdTime	queueInsertionTime
index_hadoop_wikiticker_2018-03-16T04:54:38.979Z	2018-03-16T04:54:39.058Z	2018-03-16T04:54:39.082Z

Showing 1 to 1 of 1 entries

Once the ingestion is complete, we will see the status of the job as **SUCCESS**.

In case of `FAILED` imports, please make sure that the backend that is configured to store the Metadata for the Druid cluster is up and running.

Even though Druid works well with the OpenJDK installation, I have faced a problem with a few classes not being available at runtime. In order to overcome this, I have had to use Oracle Java version 1.8 to run all Druid applications.

Now we are ready to start using Druid for our visualization tasks.

MySQL database

Apache Superset also allows us to read the data present in an RDBMS system such as MySQL. We will also create a sample database in this section, which we can use later with Superset to create visualizations.

Sample database

The employees database is a standard dataset that has a sample organization and their employee, salary, and department data. We will see how to set it up for our tasks.

This section assumes that the MySQL database is already configured and running.

Download the sample dataset

Download the sample dataset from GitHub with the following command on any server that has access to the MySQL database:

```
[user@master ~]$ sudo yum install git -y

[user@master ~]$ git clone https://github.com/datacharmer/test_db
Cloning into 'test_db'...
remote: Counting objects: 98, done.
remote: Total 98 (delta 0), reused 0 (delta 0), pack-reused 98
Unpacking objects: 100% (98/98), done.
```

Copy the data to MySQL

In this step, we will import the contents of the data in the files to the MySQL database:

```
[user@master test_db]$ mysql -u root < employees.sql
INFO
CREATING DATABASE STRUCTURE
INFO
storage engine: InnoDB
INFO
LOADING departments
INFO
LOADING employees
INFO
LOADING dept_emp
INFO
LOADING dept_manager
INFO
LOADING titles
INFO
LOADING salaries
data_load_time_diff
NULL
```

Verify integrity of the tables

This is an important step, just to make sure that all of the data we have imported is correctly stored in the database. The summary of the integrity check is shown as the verification happens:

```
[user@master test_db]$ mysql -u root -t < test_employees_sha.sql
+----------------------+
| INFO                 |
+----------------------+
| TESTING INSTALLATION |
+----------------------+
+--------------+------------------+------------------------------------------
--+
| table_name   | expected_records | expected_crc
|
+--------------+------------------+------------------------------------------
--+
| employees    | 300024 | 4d4aa689914d8fd41db7e45c2168e7dcb9697359 |
| departments  |  9 | 4b315afa0e35ca6649df897b958345bcb3d2b764 |
| dept_manager |              24 |
9687a7d6f93ca8847388a42a6d8d93982a841c6c |
| dept_emp     | 331603 | d95ab9fe07df0865f592574b3b33b9c741d9fd1b |
| titles       | 443308 | d12d5f746b88f07e69b9e36675b6067abb01b60e |
| salaries     | 2844047 | b5a1785c27d75e33a4173aaa22ccf41ebd7d4a9f |
+--------------+------------------+------------------------------------------
--+
+--------------+------------------+------------------------------------------
--+
| table_name   | found_records    | found_crc                               |
+--------------+------------------+------------------------------------------
--+
| employees    | 300024 | 4d4aa689914d8fd41db7e45c2168e7dcb9697359 |
| departments  |  9 | 4b315afa0e35ca6649df897b958345bcb3d2b764 |
| dept_manager |              24 |
9687a7d6f93ca8847388a42a6d8d93982a841c6c |
| dept_emp     | 331603 | d95ab9fe07df0865f592574b3b33b9c741d9fd1b |
| titles       | 443308 | d12d5f746b88f07e69b9e36675b6067abb01b60e |
| salaries     | 2844047 | b5a1785c27d75e33a4173aaa22ccf41ebd7d4a9f |
+--------------+------------------+------------------------------------------
--+

+--------------+---------------+-----------+
| table_name   | records_match | crc_match |
+--------------+---------------+-----------+
| employees    | OK | ok         |
| departments  | OK | ok         |
| dept_manager | OK            | ok |
| dept_emp     | OK | ok         |
| titles       | OK | ok         |
| salaries     | OK | ok         |
+--------------+---------------+-----------+
+------------------+
| computation_time |
+------------------+
| 00:00:11         |
+------------------+
+---------+--------+
| summary | result |
+---------+--------+
| CRC     | OK |
| count   | OK |
+---------+--------+
```

Now the data is correctly loaded in the MySQL database called **employees**.

Single Normalized Table

In data warehouses, its a standard practice to have normalized tables when compared to many small related tables. Lets create a single normalized table that contains details of employees, salaries, departments

```
MariaDB [employees]> create table employee_norm as select e.emp_no,
e.birth_date, CONCAT_WS(' ', e.first_name, e.last_name) full_name ,
e.gender, e.hire_date, s.salary, s.from_date, s.to_date, d.dept_name,
t.title from employees e, salaries s, departments d, dept_emp de, titles t
where e.emp_no = t.emp_no and e.emp_no = s.emp_no and d.dept_no =
de.dept_no and e.emp_no = de.emp_no and s.to_date < de.to_date and
s.to_date < t.to_date order by emp_no, s.from_date;
Query OK, 3721923 rows affected (1 min 7.14 sec)
Records: 3721923  Duplicates: 0  Warnings: 0

MariaDB [employees]> select * from employee_norm limit 1\G
************************** 1. row **************************
    emp_no: 10001
birth_date: 1953-09-02
 full_name: Georgi Facello
    gender: M
 hire_date: 1986-06-26
    salary: 60117
 from_date: 1986-06-26
   to_date: 1987-06-26
 dept_name: Development
     title: Senior Engineer
1 row in set (0.00 sec)
```

```
MariaDB [employees]>
```

Once we have normalized data, we will see how to use the data from this table to generate rich visualisations.

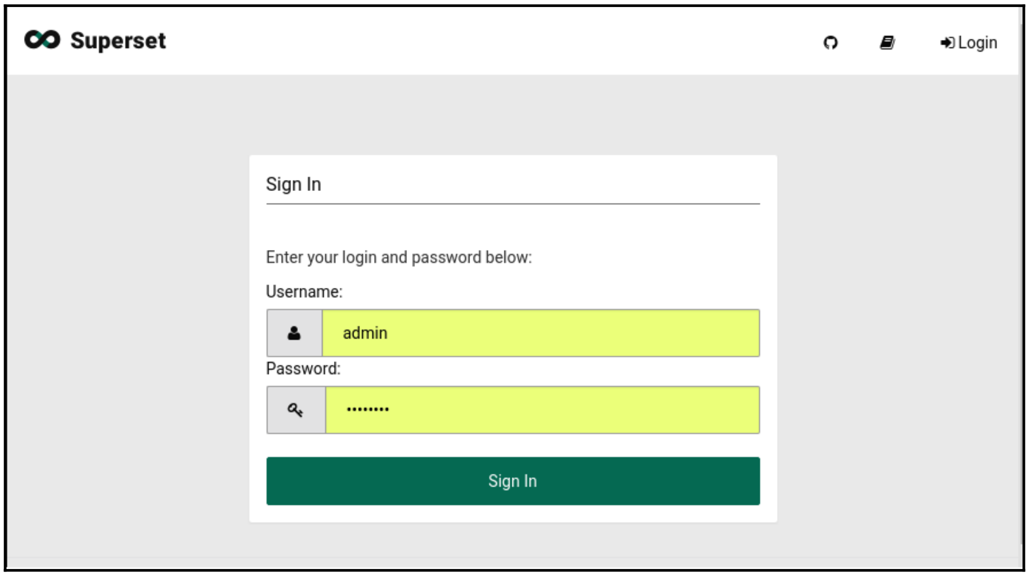
Apache Superset

Superset is a modern, enterprise-grade business intelligence application. The important feature of this application is that we can run all analyses directly from the browser. There is no need to install any special software for this.

If you remember, we have already installed Superset along with Druid in the previous sections. Now we need to learn how to use Superset to build rich visualizations.

Accessing the Superset application

Open `http://<SERVER-IP>:9088/` in your web browser. If everything is running fine, we will see a login screen like this:

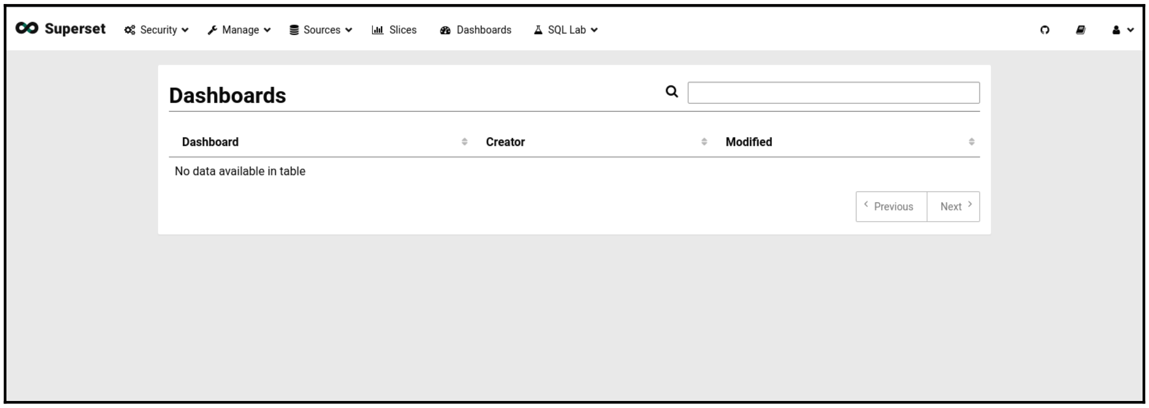


Enter `admin` as the username and the password as chosen during the installation.

Superset dashboards

Dashboards are important pieces of the Superset application. They let us showcase the results of the analytics computation in a graphical form. Dashboards are created from Slices, which in turn are built from the various data sources configured in the Superset application.

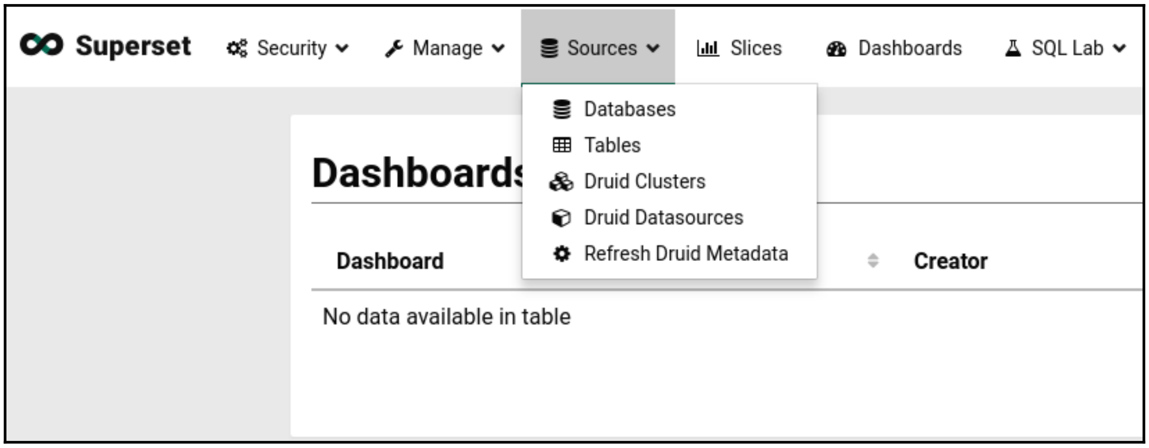
After successful login, there won't be any dashboards created automatically for us. We will see a blank list of dashboards, like this:



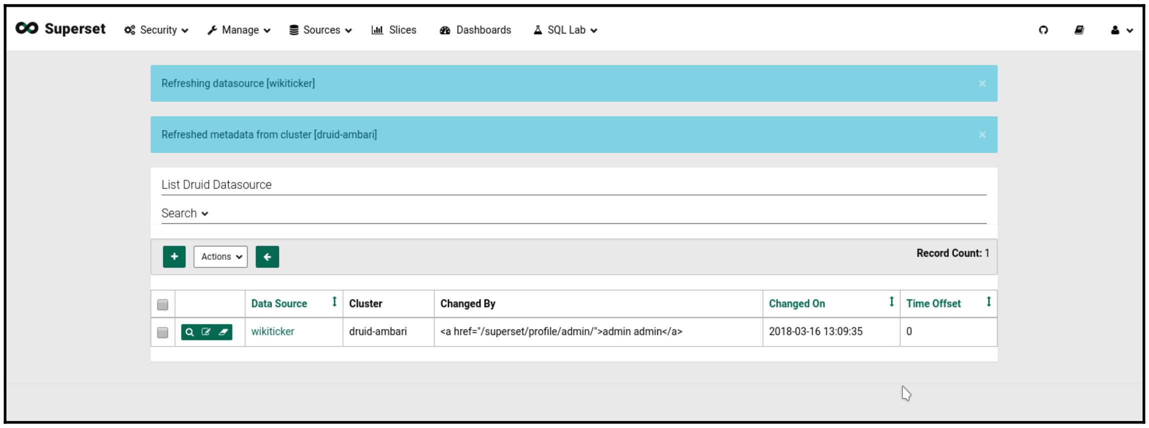
Dashboards

Dashboard	Creator	Modified
No data available in table		

In order to build dashboards, we first need to configure the data sources. So, let's click on the **Sources** menu from the top navigation and click on **Refresh Druid Metadata**:



After this step, we are taken to the data sources page and a new data source automatically appears here. Remember we uploaded this dataset to Druid before?



Refreshing datasource [wikiticker]

Refreshed metadata from cluster [druid-ambari]

List Druid Datasource

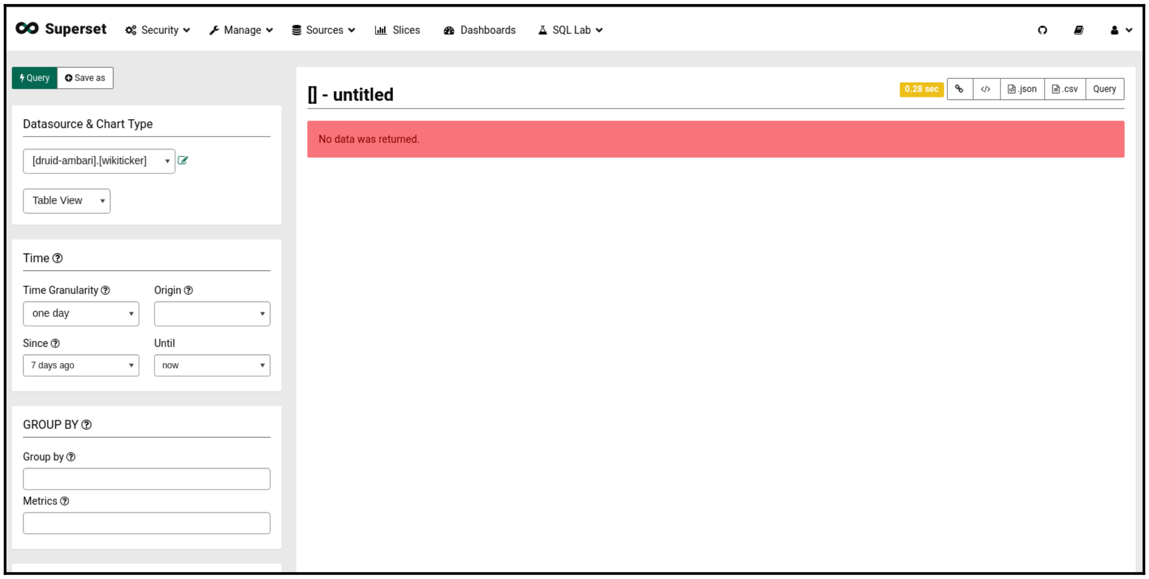
Search

Actions

Record Count: 1

		Data Source	Cluster	Changed By	Changed On	Time Offset
		wikiticker	druid-ambari	<a href="/superset/profile/admin/">admin admin</a>	2018-03-16 13:09:35	0

Now we can click on the data source name (in green), which will take us to the data source exploration page:



[] - untitled

No data was returned.

Datasource & Chart Type

[druid-ambari].[wikiticker]

Table View

Time

Time Granularity: one day

Origin:

Since: 7 days ago

Until: now

GROUP BY

Group by:

Metrics:

As we can see, this page is divided into multiple sections.

- **Left Side UI**:
    - **Datasource and Chart Type**: In this column, we can select the data source that we need to use and also the type of graphic we want to see on the right.
    - **Time**: This is the column where we can restrict the data from the data source to a given time range. Beginners tend to make a mistake with this column as they won't see any data on the right side. So, choose a start time value (a relative value like 100 years ago is recommended for better results).
    - **Group By**: This column is used to group data based on the dimensions of the input data.
    - **Other Options**: There are other options that are available below Group By, which we will explore in the coming steps.
- **Right Side UI**:
    - This UI contains the results of the options that we have selected on the left-hand side.

Understanding Wikipedia edits data

Before we jump into building visualizations. Let's take a closer look at the data we have ingested into Druid and what types of graphics we can render from that data:

Metric/Dimension	**Datatype**	**Description**
`delta`	`LONG`	Change represented in numeric form
`deleted`	`LONG`	Deleted data from the article in numeric form
`added`	`LONG`	Added data, measured in numeric form
`isMinor`	`STRING`	Boolean, indicating whether this is a minor edit or not
`page`	`STRING`	The page where the change has happened in Wikipedia
`isRobot`	`STRING`	Is the change done by a robot (not a human but some form of program)
`channel`	`STRING`	Wikipedia channel where the change has happened
`regionName`	`STRING`	Geographical region name from which the change has been done

`cityName`	`STRING`	City name from which the change has been done
`countryIsoCode`	`STRING`	ISO code of the country from which the change has been done
`user`	`STRING`	Wikipedia user or IP address that has made the change
`countryName`	`STRING`	Name of the country from which the change has been made
`isAnonymous`	`STRING`	Has the change been done by a anonymous user (not logged-in state)?
`regionIsoCode`	`STRING`	ISO code of the geographical region from which the change has been done
`metroCode`	`STRING`	This is similar to ZIP code in the United States (see `http://www.nlsinfo.org/usersvc/NLSY97/NLSY97Rnd9geocodeCodebookSupplement/gatt101.html`)
`namespace`	`STRING`	Wikipedia article/page namespace
`comment`	`STRING`	Comment that was added for this change
`isNew`	`STRING`	`true` if this is a new page (see `https://en.wikipedia.org/wiki/Wikipedia:Glossary#N`)
`isUnpatrolled`	`STRING`	`true` if the change is not a patrolled one (see `https://en.wikipedia.org/wiki/Wikipedia:New_pages_patrol`)

So, we have listed all the attributes of the data. Let's take a look at the sample one to get a better understanding of what we are talking about:

```
{
  "time": "2015-09-12T00:47:21.578Z",
  "channel": "#en.wikipedia",
  "cityName": null,
  "comment": "Copying assessment table to wiki",
  "countryIsoCode": null,
  "countryName": null,
  "isAnonymous": false,
  "isMinor": false,
  "isNew": false,
  "isRobot": true,
  "isUnpatrolled": false,
  "metroCode": null,
  "namespace": "User",
  "page": "User:WP 1.0 bot/Tables/Project/Pubs",
  "regionIsoCode": null,
  "regionName": null,
  "user": "WP 1.0 bot",
  "delta": 121,
  "added": 121,
  "deleted": 0
}
```

Once we have some understanding of the data dimensions, we need to see what types of questions we can answer from this data. These questions are the insights that are readily available to us. Later, we can represent these in the graphical form that best suits us.

So let's see some of the questions we can answer from this data.

Uni-dimensional insights:

- Which are the cities from which changes were made?
- Which pages were changed?
- Which are the countries from which changes were made?
- How many new pages were created?

Counts along the dimension:

- How many changes were made from each city?
- Which are the top cities from which changes were made?
- Which are the top users who have contributed to the changes?
- What are the namespaces that were changed frequently?

Multi-dimensional insights:

- How many changes were made between 9.00 am to 10.00 am across all countries?
- What are the wall clock hours when the edits are made by robots?
- Which country has the most origin of changes that are targeted by robots and at what times?

Looks interesting, right? Why don't we try to use Apache Superset to create a dashboard with some of these insights?

In order to do this we need to follow this simple workflow in the Superset application:

1. Data sources:
    - Define new data sources from supported databases
    - Refresh the Apache Druid data sources
2. Create Slices
3. Use the Slices to make a dashboard

If we recollect, we have already done *Step 1* in previous sections. So, we can go right away to the second and third steps.

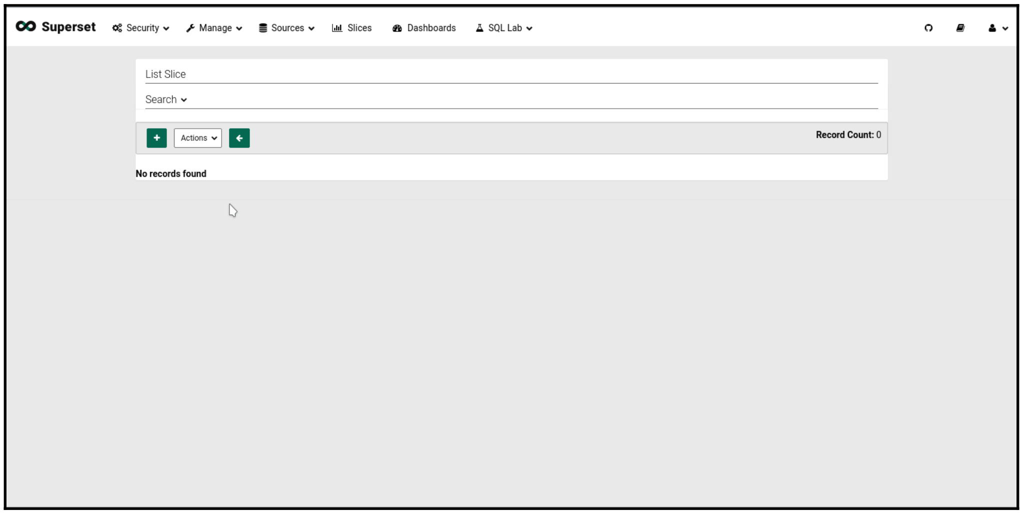
Create Superset Slices using Wikipedia data

Let's see what types of graphics we can generate using the Slices feature in the Superset application.

Unique users count

In this Slice, we see how to generate a graphic to find unique users who have contributed to the edits in the dataset.

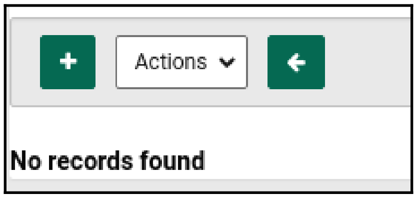
First, we need to go to the Slices page from the top navigation. After this, the screen looks like this:



Record Count: 0

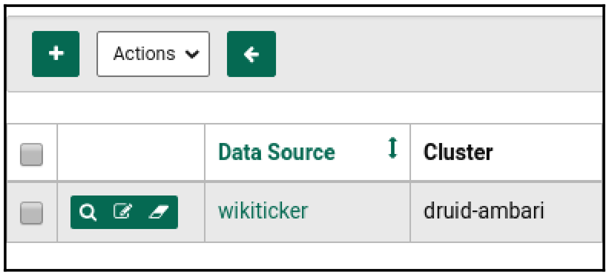
No records found

From this page, click on the plus icon (**+**) to add a new slice:



No records found

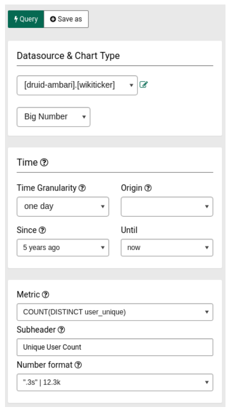
After this, we see a list of data sources that are configured in the system. We have to click on the data source name:



		Data Source	Cluster
		wikiticker	druid-ambari

After we click on **wikiticker**, we are taken to the visualization page, where we define the dimensions that we want to render as a graphic.

For the current use case, let's choose the following options from the UI:

UI Location	Graphic	Explanation
Sidebar		Choose the **Datasource** as **[druid-ambari].[wikiticker]** and the graphic type as **Big Number**. In the **Time** section, choose the value for since as **5 years ago** and leave the rest of the values to their defaults. In the **Metric** section. Choose **COUNT(DISTINCT user_unique)** from the autocomplete. In the **Subheader** Section, add **Unique User Count**, which is displayed on the screen. After this, click on the **Query** button at the top.

Datasource & Chart Type

[druid-ambari].[wikiticker]

Big Number

Time

Time Granularity: one day

Origin:

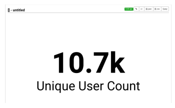
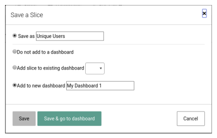
Since: 5 years ago

Until: now

Metric: COUNT(DISTINCT user_unique)

Subheader: Unique User Count

Number format: ".3s" | 12.3k

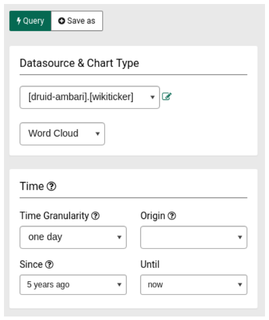
Graphic Output	 10.7k Unique User Count	We see the result of the query in this graphic.
Save Slice	 Save a Slice ● Save as Unique Users ○ Do not add to a dashboard ○ Add slice to existing dashboard ● Add to new dashboard My Dashboard 1 Save · Save & go to dashboard · Cancel	Clicking on the **Save As** button on top will show a pop-up window like this, where we need to add the corresponding values. Save the slice as `Unique Users` and add it to a new dashboard with the name `My Dashboard 1`.

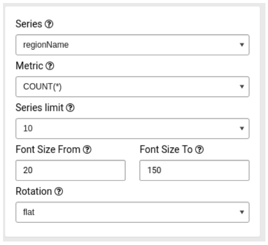
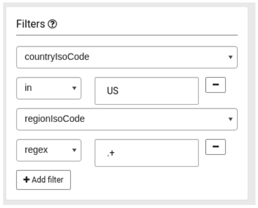
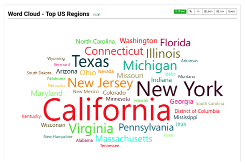
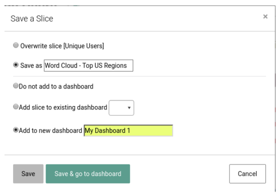
Sounds so simple, right? Let's not hurry to see the dashboard yet. Let's create some more analytics from the data in the coming sections.

Word Cloud for top US regions

In this section, we will learn how to build a word cloud for the top US regions that have contributed to the Wikipedia edits in the datasource we have in Druid. We can continue editing the same Slice from the previous section or go to a blank Slice, as mentioned in the previous section.

Let's concentrate on the values that we need to select for generating a word cloud:

UI Location	Graphic	Explanation
Sidebar	 Query · Save as Datasource & Chart Type [druid-ambari].[wikiticker] Word Cloud Time Time Granularity: one day Origin: Since: 5 years ago Until: now	Choose the **Datasource** as **[druid-ambari].[wikiticker]** and the graphic type as **Word Cloud**. In the **Time** section, choose the value for **Since** as **5 years ago** and leave the rest of the values to their defaults.

	 Series regionName Metric COUNT(*) Series limit 10 Font Size From 20 Font Size To 150 Rotation flat	In the **Series** section. Choose the **regionName** from the dropdown. In **Metric**, choose **COUNT(*)**, which is the total edit count.
	 Filters countryIsoCode in US regionIsoCode regex .+ + Add filter	In the **Filters** section, choose **countryIsoCode**; it should be in **US**. Add another filter to select only valid regions (skip null codes). Add the values as shown here in the graphic.
Graphic Output		After clicking on **Query**, we see this beautiful word cloud.
Save Slice	 Save a Slice Overwrite slice [Unique Users] Save as Word Cloud - Top US Regions Do not add to a dashboard Add slice to existing dashboard Add to new dashboard My Dashboard 1 Save / Save & go to dashboard / Cancel	Clicking on the **Save As** button at the top will show a pop-up window like this, where we need to add the corresponding values. Save the Slice as `Word Cloud - Top US Regions` and add it to a new dashboard named `My Dashboard 1`.

The significance of the word cloud is that we can see the top words according to their relative sizes. This type of visualization is helpful when there are fewer words for which we want to see the relative significance.

Let's try to generate another graphic from the data.

Sunburst chart – top 10 cities

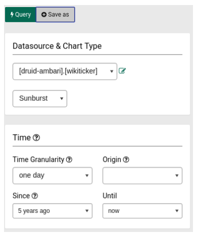
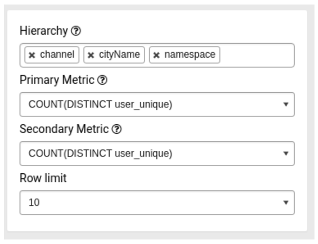
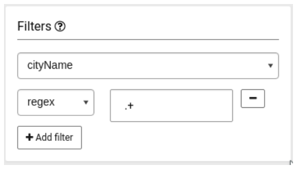
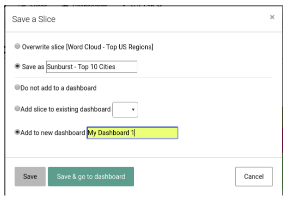
In this section, we will learn about a different type of chart that we have not seen so far in this chapter. But first, let's put forward the use case.

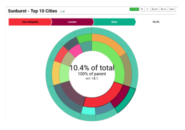
We want to find the unique users per channel, city name, and namespace at all three levels; that is, the graphic should be able to show us the:

- Unique users per channel
- Unique users per channel/city name
- Unique users per channel/city name/namespace

In order to show this kind of hierarchical data, we can use a sunburst chart.

Let's check out what type of values we need to select to render this type of chart:

UI Location	**Graphic**	**Explanation**
Sidebar	 Query · Save as Datasource & Chart Type [druid-ambari].[wikiticker] Sunburst Time Time Granularity: one day Origin: Since: 5 years ago Until: now	Choose the **Datasource** as **[druid-ambari].[wikiticker]** and the graphic type as **Sunburst**. In the **Time** section, choose the value for **Since** as **5 years ago** and leave the rest of the values to their defaults.
	 Hierarchy: channel, cityName, namespace Primary Metric: COUNT(DISTINCT user_unique) Secondary Metric: COUNT(DISTINCT user_unique) Row limit: 10	In the **Hierarchy** section, choose the `channel`, `cityName`, and `namespace` from the dropdown. In the **Primary Metric** and **Secondary Metric**, choose **COUNT(DISTINCT user_unique)**, which is the total user count.
	 Filters cityName regex .+ Add filter	In the **Filters** section, choose **cityName** and add the not null condition using **regex** matching
	 Save a Slice Overwrite slice [Word Cloud - Top US Regions] [✓ Save as] Sunburst - Top 10 Cities Do not add to a dashboard Add slice to existing dashboard [✓ Add to new dashboard] My Dashboard 1 Save · Save & go to dashboard · Cancel	Clicking on the **Save As** button at the top will show a pop-up window like this. We need to add the corresponding values here. Save the Slice as `Sunburst - Top 10 Cities` and add it to a new dashboard named `My Dashboard 1`.

Graphic Output		After clicking on **Query**, we see this beautiful graphic.

As we can see there are three concentric rings in the graphic:

- The innermost ring is the `channel` dimension
- The middle ring shows the `cityName` dimension
- The outermost ring is the `namespace` dimension

When we hover over the innermost ring, we can see how it spreads out into the outermost circles. The same thing happens with the other rings as well.

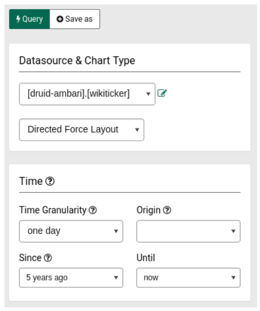
This type of graphic is very helpful when we want to do funnel analysis on our data. Let's take a look at another type of analysis in the next section.

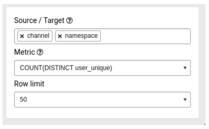
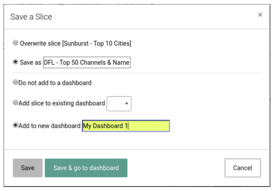
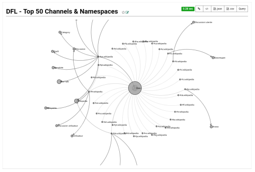
Top 50 channels and namespaces via directed force layout

Directed force layout (**DFL**) is a network layout with points that are interconnected to each other. Since it's a force layout, we can see the points moving on the screen as `d3.js` applies the physics engine.

In this network graph, we want to understand the connectivity between the namespace and channel over the unique users count metric. Since this is a network graph, we will see the nodes getting repeated in different paths.

Let's see how we can arrive at this graph:

UI Location	**Graphic**	**Explanation**
Sidebar	 Datasource & Chart Type [druid-ambari].[wikiticker] Directed Force Layout Time Time Granularity: one day Origin: Since: 5 years ago Until: now	Choose the **Datasource** as **[druid-ambari].[wikiticker]** and the **Graphic** type as **Directed Force Layout**. In the **Time** section, choose the value for since as **5 years ago** and leave the rest of the values to their defaults.

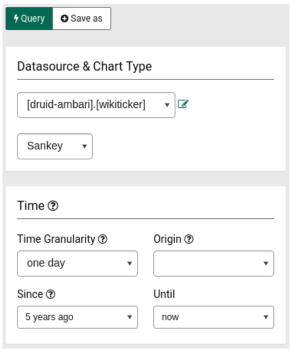
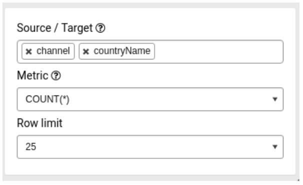
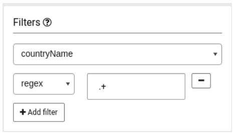
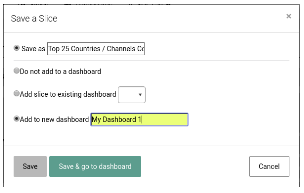
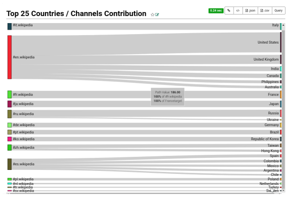
	 Source / Target channel namespace Metric COUNT(DISTINCT user_unique) Row limit 50	In the **Source / Target** section, choose the `channel` and `namespace` from the dropdown. In the **Metric** section, choose **COUNT(DISTINCT user_unique)** which is the total user count. We keep the **Row limit** at **50** so that we will see only the top 50.
	 Save a Slice Overwrite slice [Sunburst - Top 10 Cities] [✓ Save as] DFL - Top 50 Channels & Name Do not add to a dashboard Add slice to existing dashboard [✓ Add to new dashboard] My Dashboard 1 Save · Save & go to dashboard · Cancel	Clicking on the **Save As** button at the top will show a pop up window like this, where we need to add the corresponding values. Save the Slice as `DFL - Top 50 Channels & Namespaces`. Add it to a new dashboard with the name `My Dashboard 1`.
Graphic Output		After clicking on **Query**, we see this beautiful graphic.

Feel free to drag the nodes in the graphic to learn more about how they are interconnected to each other. The size of the nodes indicates the unique user count and its breakdown (similar to a sunburst chart).

Let's spend some time learning another visualization and business use case in the next section.

Top 25 countries/channels distribution

Now we will learn the Sankey chart, a waterfall-like way of representing of breakdown and interconnectivity between data. In this case, we want to find out how the **channelName** and **countryName** dimensions are related when it comes to the unique users metric:

UI Location	Graphic	Explanation
Sidebar	 Query Save as Datasource & Chart Type [druid-ambari].[wikiticker] Sankey Time Time Granularity: one day Origin: Since: 5 years ago Until: now	Choose the **Datasource** as **[druid-ambari].[wikiticker]** and the **Graphic Type** as **Sankey**. In the **Time** section, choose the value for **Since** as **5 years ago** and leave the rest as default.
	 Source / Target: channel, countryName Metric: COUNT(*) Row limit: 25	In the **Source / Target** section, choose `channel` and `countryName` from the drop-down. In the **Metric**, choose **COUNT(*)**, which is the total edit count. Keep the row limit at **25**; so we will see only the top 25 items.
	 Filters countryName regex .+ Add filter	In the **Filters** section, choose **countryName** and enable the **regex** filter so as to choose only those records that have a valid country name.
	 Save a Slice ● Save as Top 25 Countries / Channels Cc ○ Do not add to a dashboard ○ Add slice to existing dashboard ● Add to new dashboard My Dashboard 1 Save / Save & go to dashboard / Cancel	Clicking on the **Save As** button at the top will show a pop-up window. We need to add the corresponding values here. Save the Slice as `Top 25 Countries/Channels Distribution` and add it to a new dashboard with the name `My Dashboard 1`.
Graphic Output		After clicking on **Query**, we see this beautiful graphic.

This completes the list of all the analytics that we can generate so far. Now in the next section, we will see how to use this in the dashboard (which was our original goal anyway).

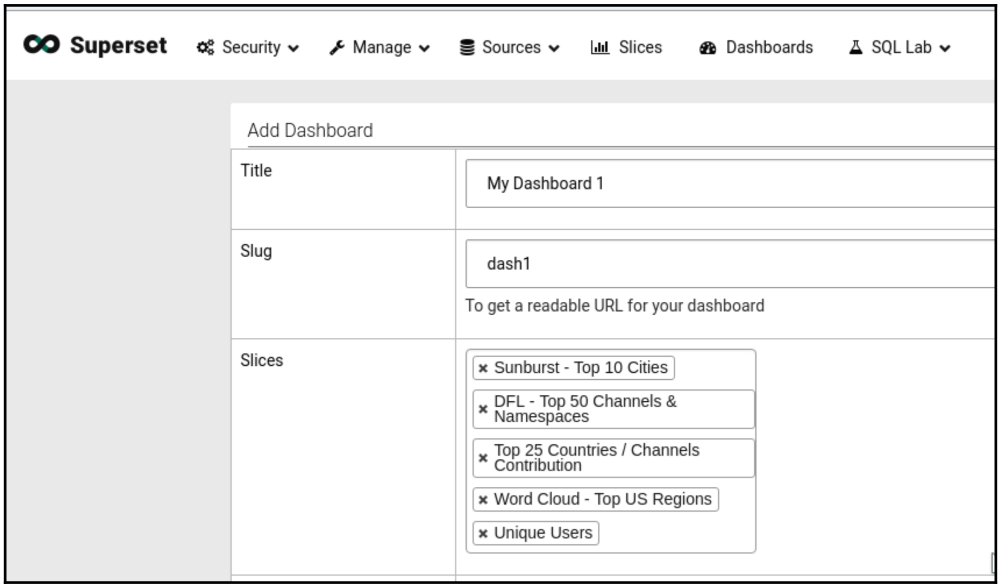
Creating wikipedia edits dashboard from Slices

So far we have seen how to create slices in the Apache Superset application for the wikipedia edits data, that is stored in the Apache Druid database. Its now time for us to see how to create a dashboard so that we can share it with the Business Teams or any other teams for which we want to share the insights.

In this process, the first step would be to click on the **Dashboard** menu on the top navigation bar. Which will take us to **Add New Dashboard** Page, where we need to fill the following details.

Element	**Description**	**Value**
Title	This is the name of the dashboard that we want to create	My Dashboard 1
Slug	Short alias for the dashboard	dash1
Slices	List of Slices that we want to add to the dashboard.	• Sunburst - Top 10 Cities • DFL - Top 50 Channels & Namespaces • Top 25 Countries / Channels Contribution • Word Cloud - Top US Regions • Unique Users
Other Fields	We can leave the other fields as empty as they are not mandatory to create the dashboard	

Here is the graphic for this page:

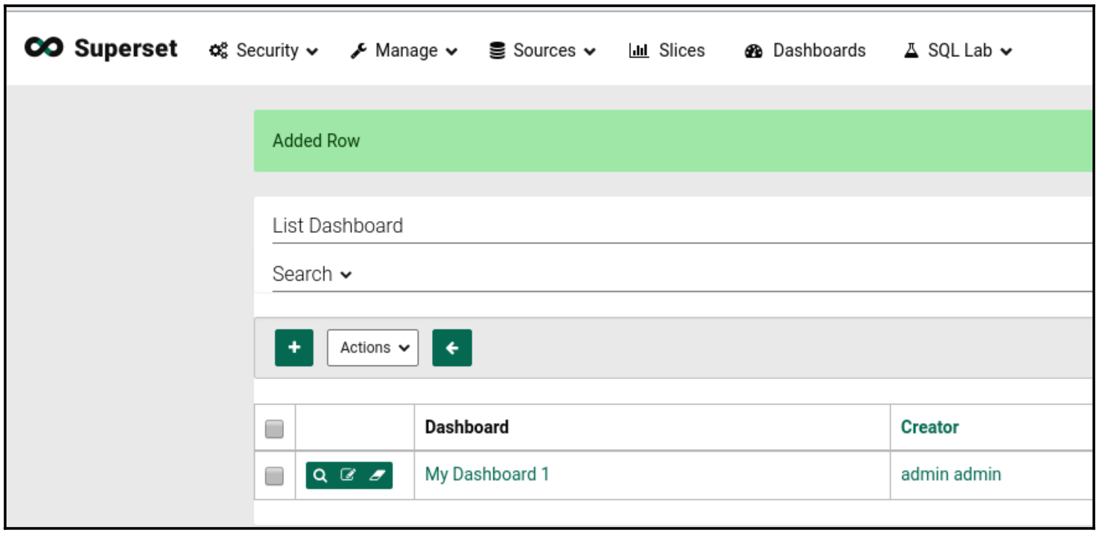


Add Dashboard

Title	My Dashboard 1
Slug	dash1 To get a readable URL for your dashboard
Slices	× Sunburst - Top 10 Cities × DFL - Top 50 Channels & Namespaces × Top 25 Countries / Channels Contribution × Word Cloud - Top US Regions × Unique Users

Click on **Save** button at the bottom of the screen once the changes look good.

This will take us to the next step where we can see that the dashboard is successfully created:

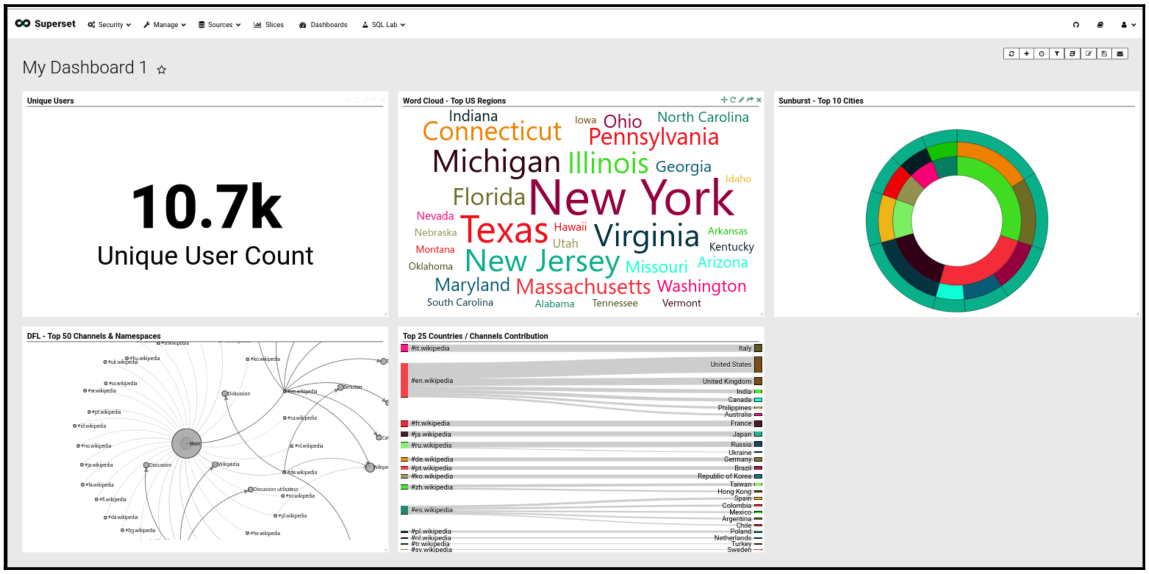


Added Row

List Dashboard

		Dashboard	Creator
		My Dashboard 1	admin admin

We can see the **My Dashboard 1** in the list of dashboards. In order to access this dashboard click on it, Where we are taken to the dashboard screen:



As we can see we have a very powerful way of representing all the raw data. This will definitely have an impact on the end users in making sure that the message is conveyed.

So far we have learned how to create slices and Dashboards from the data that is stored in the Apache Druid Columnar Database. In the next section we will see how to connect to RDBMS and generate slices and dashboards from that data.

Apache Superset with RDBMS

Apache Superset is built using Python programming language and supports many relational databases as it uses SQLAlchemy as the database driver. The installation of these drivers are out of scope in this section. But, it should be very easy to install those. Most of the time the Operating system vendors package them for us. So, we need not worry about the manual installation of these.

Supported databases

Here are some of the database that are supported by Apache Superset:

Database Name	**Python Package Name**	**Driver URI Prefix**	**Details**
MySQL	`mysqlclient`	`mysql://`	Oracle MySQL Database
PostgreSQL	`psycopg2`	`postgresql+psycopg2://`	The worlds most advanced opensource database
Presto	`pyhive`	`presto://`	Opensource distributed query Engine
Oracle	`cx_Oracle`	`oracle://`	Multi-model Database management system created by Oracle Corporation.
Sqlite		`sqlite://`	Fast, Scalable Embedded Database Library
Redshift	`sqlalchemy-redshift`	`postgresql+psycopg2://`	Amazon Redshift is Columnar database built on PostgreSQL
MSSQL	`pymssql`	`mssql://`	Microsoft SQL Server

Impala	`impyla`	`impala://`	Apache Impala is Massively Parallel Processing SQL Engine that runs on Hadoop
SparkSQL	`pyhive`	`jdbc+hive://`	Apache Spark Module for writing SQL in Spark Programs.
Greenplum	`psycopg2`	`postgresql+psycopg2://`	Greenplum is advanced , fully featured opensource data platform
Athena	`PyAthenaJDBC`	`awsathena+jdbc://`	Amazon Athena is Serverless Interactive Query Service
Vertica	`sqlalchemy-vertica-python`	`vertica+vertica_python://`	Vertica is Bigdata analytics software
ClickHouse	`sqlalchemy-clickhouse`	`clickhouse://`	Opensource distributed, columnar datastore

Portions of the above table is extracted from the official documentation of Apache Superset (`https://superset.incubator.apache.org/installation.html#database-dependencies`)

Understanding employee database

If you remember, in the previous sections we have imported a sample database called Employees and loaded it into the MySQL Database. We will dig further into this sample datastore so that we will learn what types of analytics we can generate from this.

Employees table

The `employees` table contains details of Employees (randomly generated data) with the following properties

Column	Datatype	Description
`emp_no`	`INTEGER`	Employee Number
`birth_date`	`DATE`	Employee Date Of Birth
`first_name`	`STRING`	First Name of Employee
`last_name`	`STRING`	Last Name of Employee
`gender`	`STRING`	Gender of Employee, M if Male, F if Female
`hire_date`	`STRING`	Latest Joining date of Employee

Departments table

The `departments` table consists of basic details of every department in the organisation. This is further understood with this table:

Table Column	Datatype	Description
`dept_no`	`STRING`	Department Number
`dept_name`	`STRING`	Department Name

Department manager table

The `dept_manager` table has records about Employee acting as manager for a given department. More details are in this table:

Table Column	Dataype	Description
`emp_no`	`INT`	Employee ID who is acting as manager for this department
`dept_no`	`STRING`	Department ID
`from_date`	`DATE`	Starting date from which Employee is acting as Manager for this department.
`to_date`	`DATE`	Ending date till where the Employee has acted as Manager for this department.

Department Employees Table

The `dept_emp` table consists of all the records which show how long each employee belonged to a department.

Table Column	Datatype	Description
`emp_no`	`INT`	Employee ID
`dept_no`	`STRING`	Department ID
`from_date`	`DATE`	Starting date from which employee belongs to this department
`to_date`	`DATE`	Last date of employee in this department

Titles table

The **titles** table consists of all the roles of employees from a given date to end date. More details are shown as follows:

Table Column	Datatype	Description
`emp_no`	`INT`	Employee Id
`title`	`STRING`	Designation of the employee
`from_date`	`DATE`	Starting date from which employee has assumed this role
`to_date`	`DATE`	Last date where the employee has performed this role

Salaries table

The `salaries` table consists of salary history of a given employee. More details are explained in the following table:

Table Column	Datatype	Description
`emp_no`	`INT`	Employee Id
`salary`	`INT`	Salary of Employee
`from_date`	`DATE`	Starting day for which salary is calculated
`to_date`	`DATE`	Last day for which salary is calculated.

Normalized employees table

The `employee_norm` table consists of data from employees, salaries, departments, `dept_emp` and titles table. Lets look at this table in detail:

Table Column	Datatype	Description
`emp_no`	`INT`	Employee ID
`birth_date`	`DATE`	Date of Birth of Employee
`full_name`	`STRING`	Employee Full Name
`gender`	`STRING`	Gender of Employee
`hire_date`	`DATE`	Joining date of Employee
`salary`	`INT`	Salary of Employee for the period
`from_date`	`DATE`	Salary period start
`to_date`	`DATE`	Salary period end
`dept_name`	`STRING`	Department where the employee is working during this salary period
`title`	`STRING`	Designation of the employee during this time period

With this knowledge of various tables in the Employee database we now have some understanding of the data we have so far. Now, the next task is to find out what types of analytics we can generate from this data. We will learn this in the next section.

Superset Slices for employees database

Once we have some basic understanding of the type of data that is stored in the MySQL database. We will now see what types of we can answer from this data.

Uni-dimensional insights:

- How many employees are there in the organisation?
- What is the total salary paid for all employees in the organisation?
- How many departments are there?

Multi dimensional insights

- What is the total salary paid for every year?
- What is the total salary per department?
- Who is the top paid employee for every year?

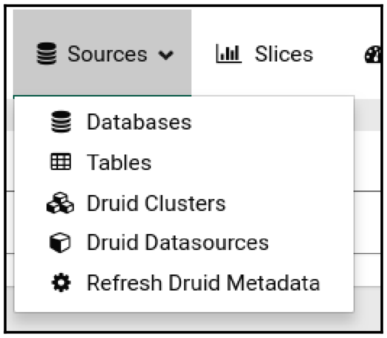
If we think along these lines we should be able to answer very important questions regarding the data and should be able generate nice graphics.

Lets take few examples of what types of visualisations we can generate in the coming sections.

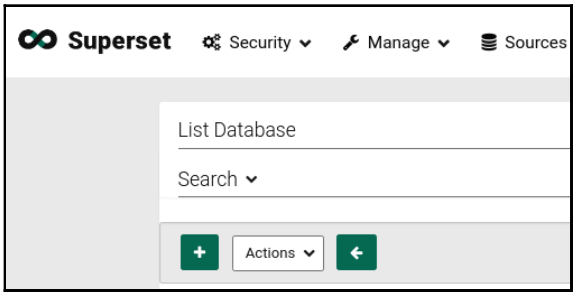
Register MySQL database/table

Before we start generating Slices for the employee tables, We should first register it. The registration process includes the following steps.

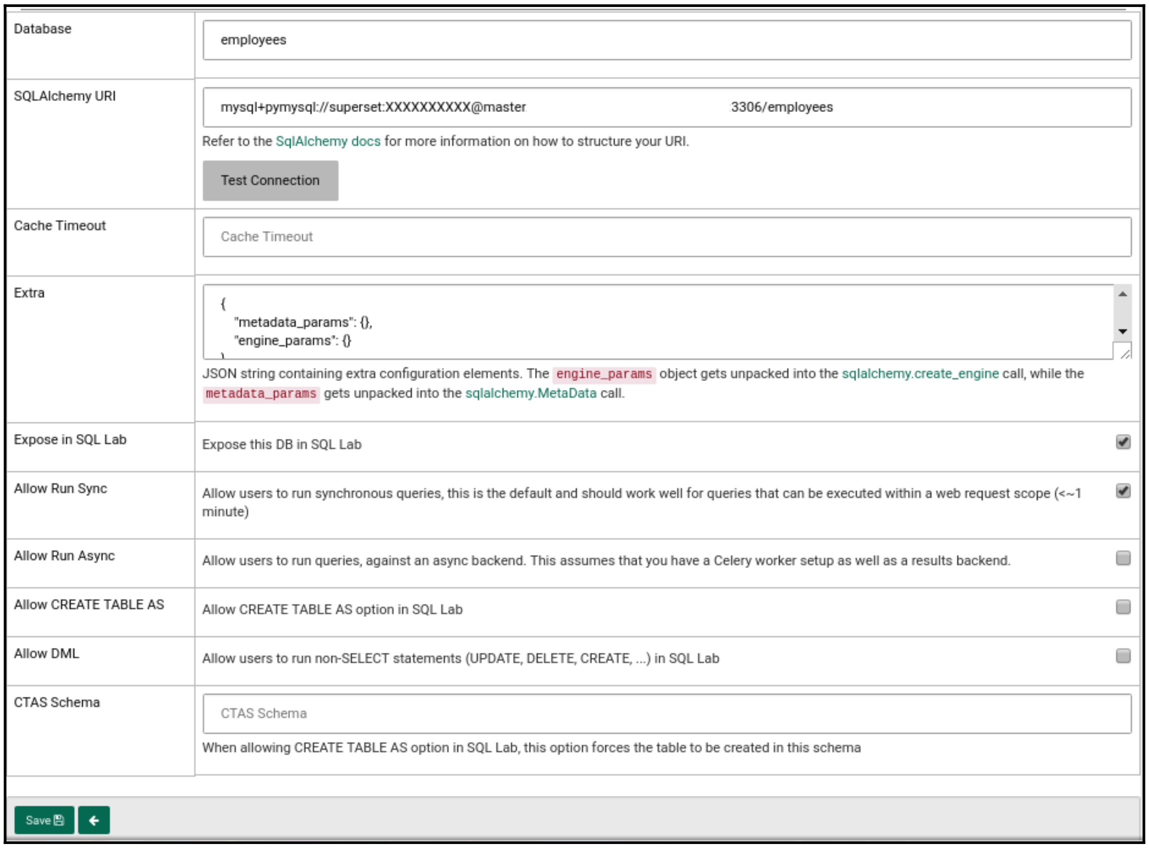
Open the Databases by clicking on the Databases dropdown from the **Sources** menu in the top navigation bar as shown here:



After this we need to click on the plus (+) icon from the page:



This will take us to a page where we can register the new database. The screen looks like this:



Database	employees
SQLAlchemy URI	mysql+pymysql://superset:XXXXXXXXXX@master 3306/employees Refer to the SqlAlchemy docs for more information on how to structure your URI. Test Connection
Cache Timeout	Cache Timeout

Extra	{
	"metadata_params": {},
	"engine_params": {}
	}
	JSON string containing extra configuration elements. The `engine_params` object gets unpacked into the sqlalchemy.create_engine call, while the `metadata_params` gets unpacked into the sqlalchemy.MetaData call.

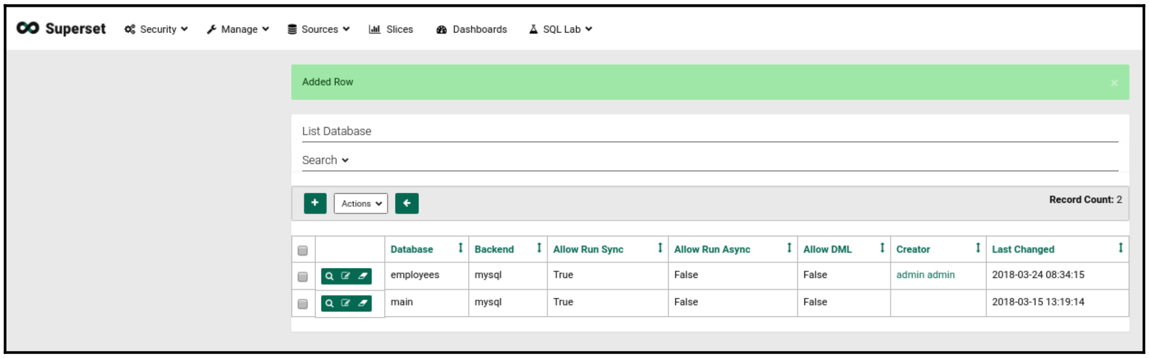
Expose in SQL Lab	Expose this DB in SQL Lab ✓
Allow Run Sync	Allow users to run synchronous queries, this is the default and should work well for queries that can be executed within a web request scope (<~1 minute) ✓
Allow Run Async	Allow users to run queries, against an async backend. This assumes that you have a Celery worker setup as well as a results backend. ☐
Allow CREATE TABLE AS	Allow CREATE TABLE AS option in SQL Lab ☐
Allow DML	Allow users to run non-SELECT statements (UPDATE, DELETE, CREATE, ...) in SQL Lab ☐
CTAS Schema	CTAS Schema When allowing CREATE TABLE AS option in SQL Lab, this option forces the table to be created in this schema

Save

We will fill the following details as shown here.

Field Name	**Value**	**Description**
Database	`employees`	Name of the database that we want to register. (Enter the same name as its in the MySQL Database)
SQLAlchemy URI	`mysql+pymysql://superset:superset@master:3306/employees`	URI to access this database programatically. This will include the protocol/driver, username, password, hostname & dbname
Other Fields		Keep them as default

After this click on **Save** Button, which will save the database details with Apache Superset. We are taken to the list of tables page which looks like this:



Added Row

List Database

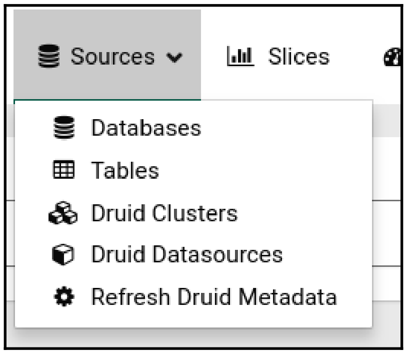
Search

Record Count: 2

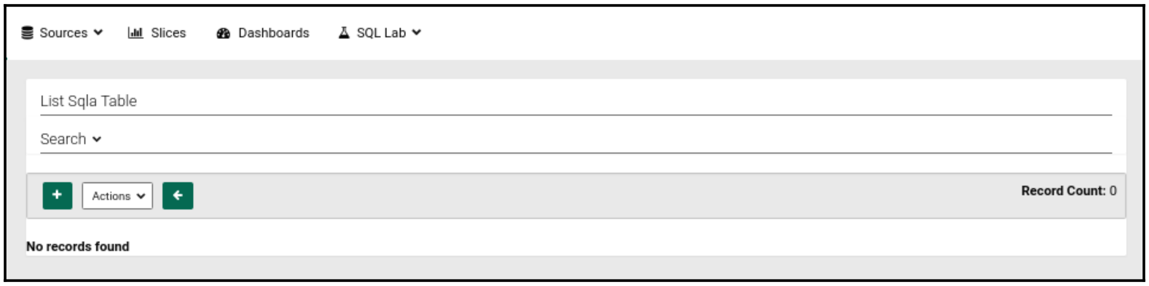
	Database	Backend	Allow Run Sync	Allow Run Async	Allow DML	Creator	Last Changed
	employees	mysql	True	False	False	admin admin	2018-03-24 08:34:15
	main	mysql	True	False	False		2018-03-15 13:19:14

As we can see, we have the employees database registered with MySQL backend.

In the next step we need to chose the tables from the top menu:



Since we do not have any tables registered, we will see a empty page like this:

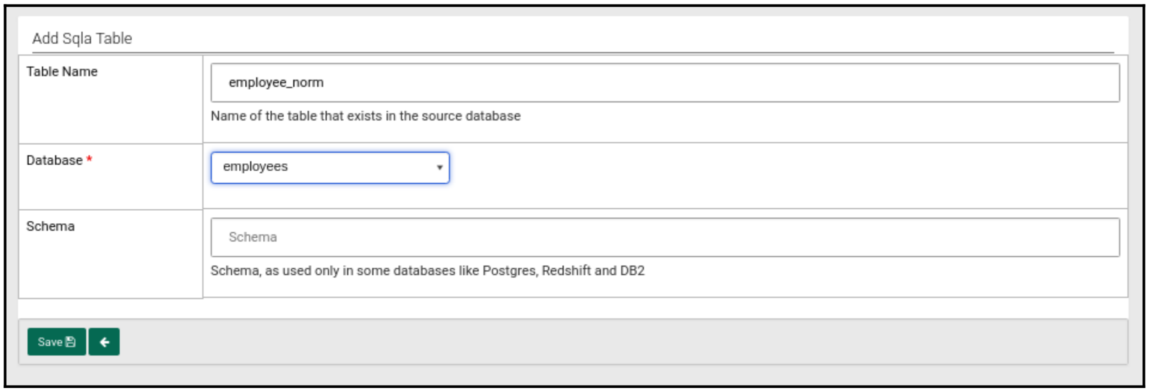


List Sqla Table

Record Count: 0

No records found

In order to register a new table we have to click on the plus (icon) in the UI, Which takes us to the following page:



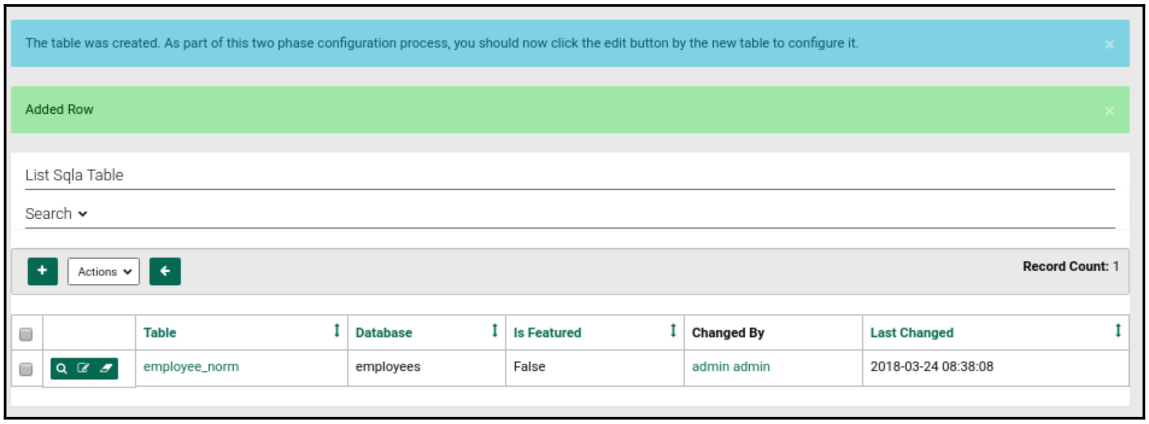
Add Sqla Table

Table Name	employee_norm Name of the table that exists in the source database
Database *	employees
Schema	Schema Schema, as used only in some databases like Postgres, Redshift and DB2

Enter the values for the fields as shown below and click **Save** once done:

Field Name	**Value**	**Description**
Table name	`employee_norm`	Name of the table that we want to register.
Database	`employees`	Select the database that is already registered with Superset.

Now we can see that the table is successfully registered as shown in the screen in the following screenshot:



The table was created. As part of this two phase configuration process, you should now click the edit button by the new table to configure it.

Added Row

List Sqla Table

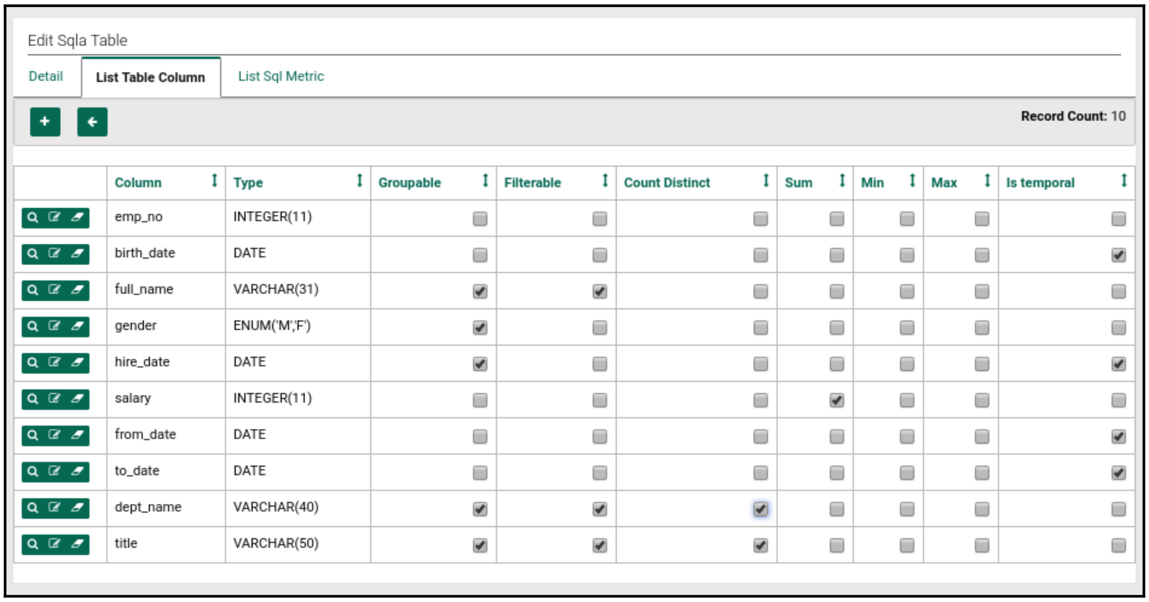
Search

Record Count: 1

		Table	Database	Is Featured	Changed By	Last Changed
		employee_norm	employees	False	admin admin	2018-03-24 08:38:08

One of the important features of Superset is that it will automatically select the different types of operations that we can perform on the columns of the table according to the datatype. This drives what types of dimensions, metrics we are shown in the rest of the UI.

In order to select these options, we need to edit the table by clicking on the edit icon and we are shown this page:



Edit Sqla Table

Detail | **List Table Column** | List Sql Metric

Record Count: 10

	Column	Type	Groupable	Filterable	Count Distinct	Sum	Min	Max	Is temporal
	emp_no	INTEGER(11)	☐	☐	☐	☐	☐	☐	☐
	birth_date	DATE	☐	☐	☐	☐	☐	☐	✓
	full_name	VARCHAR(31)	✓	✓	☐	☐	☐	☐	☐
	gender	ENUM('M','F')	✓	☐	☐	☐	☐	☐	☐
	hire_date	DATE	✓	☐	☐	☐	☐	☐	✓
	salary	INTEGER(11)	☐	☐	☐	✓	☐	☐	☐
	from_date	DATE	☐	☐	☐	☐	☐	☐	✓
	to_date	DATE	☐	☐	☐	☐	☐	☐	✓
	dept_name	VARCHAR(40)	✓	✓	✓	☐	☐	☐	☐
	title	VARCHAR(50)	✓	✓	✓	☐	☐	☐	☐

As we can see, Apache Superset has automatically recognized the datatype of each and every field and it also provided us with an option to chose these dimensions for various activities. These activities are listed in the following table:

Activity	**Description**
Groupable	If the checkbox is selected, then the field can be used as part of Grouping operations (`GROUP BY` in SQL).
Filterable	If the checkbox is selected, then the field can be used as part of Conditional operations (`WHERE` clause).
Count Distinct	If the checkbox is selected, then the field can be used as part of count (`DISTINCT`) operation on the field.
Sum	If the checkbox is selected, then the field can be used as part of `SUM()` function.
Min/Max	Indicates that the field can be used as part of finding minimum and maximum value.
Is Temporal	Indicates the field is a time dimension.

Make changes as shown above and click on Save button.

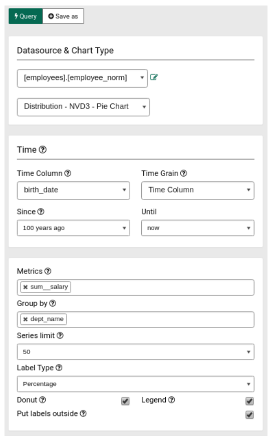
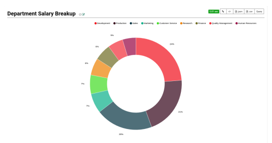
Now we are ready to start creating slices and dashboard in the next steps.

Slices and Dashboard creation

As we have seen in the previous sections, In order to create Dashboards we first need to create slices. In this section we will learn to create few slices.

Department salary breakup

In this slice we will learn how to create a visualization that will show the percentage of salary breakdown per department:

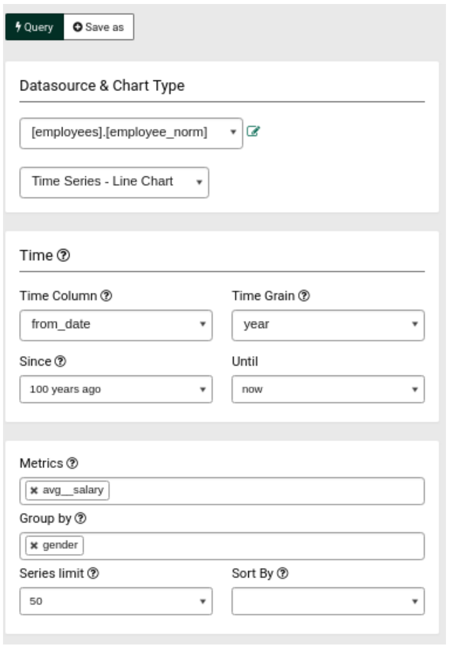
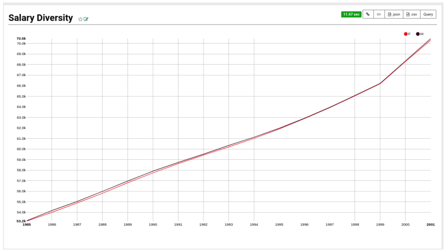
UI Location	Graphic	Description
Sidebar	 Query Save as Datasource & Chart Type [employees].[employee_norm] Distribution - NVD3 - Pie Chart Time Time Column: birth_date Time Grain: Time Column Since: 100 years ago Until: now Metrics: sum__salary Group by: dept_name Series limit: 50 Label Type: Percentage Donut ✓ Legend ✓ Put labels outside ✓	**Datasource & Chart Type**: select **[employees].[employee_norm]** as the **datasource** and **Distribution - NVD3 - Pie Chart** as chart type In the **Time** section, select **birth_date** as **Time Column** and **100 years ago** as **Since** column. In the **Metrics** section, select **sum_salary** as the value from dropdown and **dept_name** as **Group By.**
Graphic Output		Clicking on **Query** button will render this good liking chart. Save it with the name **Department Salary Breakup**.

Just like in the previous section, See how easy it is to create good looking graphic without any programming knowledge.

In the next section we will learn about another type of graphic from the same employees database.

Salary Diversity

This is a important graphic, where we identify how the salary diversity is between genders across the history of organisation. Here we use average salary as a basis for the analysis.

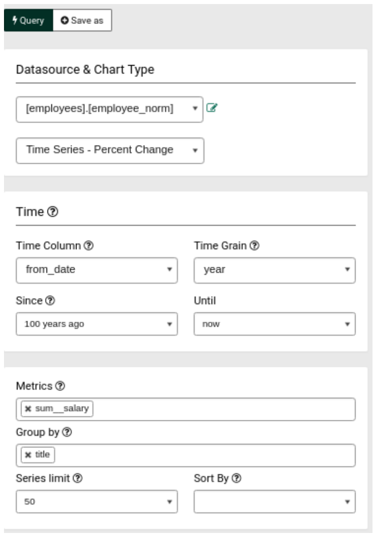
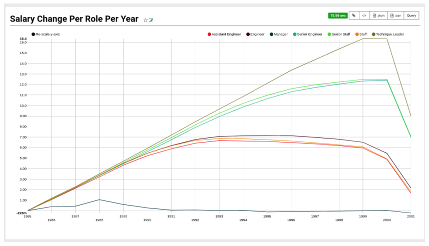
UI Location	Graphic	Description
Sidebar	 Query · Save as Datasource & Chart Type [employees].[employee_norm] Time Series - Line Chart Time Time Column: from_date Time Grain: year Since: 100 years ago Until: now Metrics: avg__salary Group by: gender Series limit: 50 Sort By:	**Datasource & Chart Type**: select **[employees].[employee_norm]** as the datasource and Time Series - line chart as chart type In the **Time** section, select **birth_date** as **Time Column** & **100 years ago** as since column. In the **Metrics** section, select **avg_salary** as the **Metric** and gender as `Group By`.
Output		Graphic showing the average salary per Gender for every Year. Save this with the title **Salary Diversity**

As we can see from the graphic, the salary breakup is even between genders and are very close. There is also a similar increase in the average salary over the period.

In the next section we will learn to generate another type of graphic that will give us different insight into the data.

Salary Change Per Role Per Year

This is a important statistic where we want to find out how much salary change is there for different Titles in the organisation across years.

UI Location	Graphic	Description
Sidebar	 Query / Save as; Datasource & Chart Type: [employees].[employee_norm]; Time Series - Percent Change; Time: Time Column: from_date; Time Grain: year; Since: 100 years ago; Until: now; Metrics: sum__salary; Group by: title; Series limit: 50; Sort By:	**Datasource & Chart Type**: select **[employees].[employee_norm]** as the datasource and **Time Series - Percentage Change** as chart type In the **Time** section, Select **from_date** as **Time** column , **Year** as **Time Granularity** & **100 years ago** as **Since** column. In the **Metrics** Section, select **sum_salary** as the **Metric** and **title** as **Group** By.
Output		Clicking on **Query**, yields us the following graphic. Save this with the name **Salary Change Per Role Per Year**.

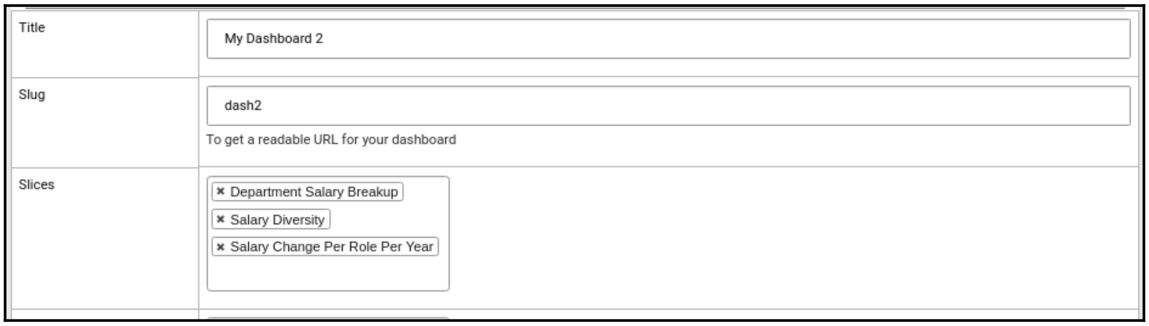
From this graphic we can find out that few roles have very large difference in the total salary within the organisation.

So far we have created three slices, we will create a new dashboard with the slices created so far.

Dashboard creation

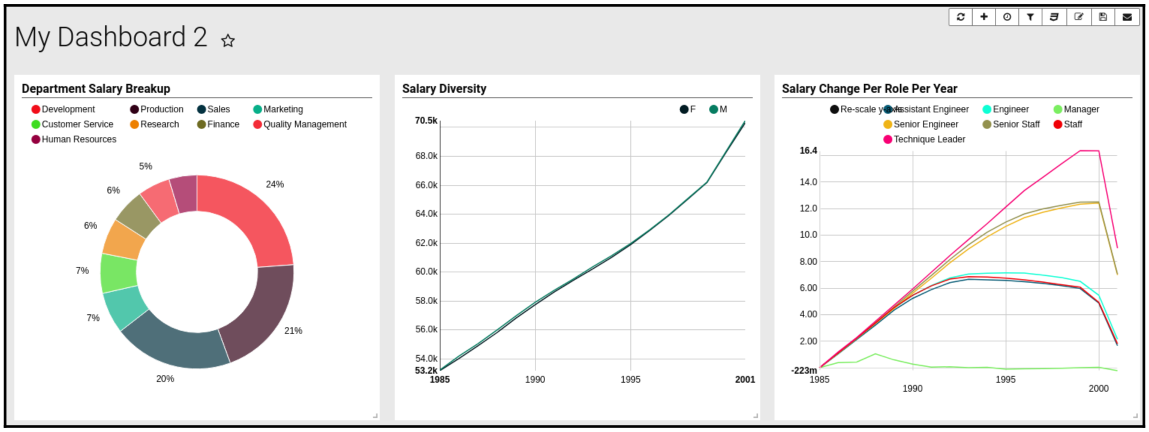
In this step we will create a new dashboard by going to the dashboards page and clicking on the Add Dashboard icon (as shown in previous sections).

We are presented with the following screen where we select the three slices we have created so far and click **Save**:



Title	My Dashboard 2
Slug	dash2 To get a readable URL for your dashboard
Slices	× Department Salary Breakup × Salary Diversity × Salary Change Per Role Per Year

Once the dashboard is saved successfully we can see it like this:



As we can see, Dashboards are very powerful way to express large amounts of data in a simple fashion.

Summary

In this chapter, we learned about data visualization and how it helps the users to receive the required message without any knowledge of the underlying data. We then saw the different ways to visualize our data graphically.

We walked through Hadoop applications such as Apache Druid and Apache Superset that are used to visualize data and learned how to use them with RDBMses such as MySQL. We also saw a sample database to help us understand the application better.

In the next chapter, we will learn how to build our Hadoop cluster on the cloud.

10
Developing Applications Using the Cloud

In the early days of computing, CPU power and storage were very scarce, and so the cost of purchasing relevant equipment was very high. With the advances in the development of personal computing in the early 80s by Apple and Microsoft, more and more individuals and organizations have gained access to these computing devices. As the industry has developed the way chips are made and billions if not trillions of transistors are now put on single chips, the size of these computing devices has drastically reduced, from taking up entire rooms to comprising a single unit of a rack in the data center. When the computation speed and storage device capacity started increasing, individuals and enterprises started to realize that efficiently managing their computing resources was becoming a challenge.

The widespread use of the internet has also made a significant contribution to how individuals can access resources.

In this chapter, we will cover the following topics:

- What is the Cloud?
- Available technologies in the Cloud
- Planning Cloud infrastructure
    - High availability in the Cloud
    - Business continuity planning in the Cloud
    - Security in the Cloud
- Building a Hadoop cluster in the Cloud
- Cloud and in-house applications
- Data access in the Cloud

What is the Cloud?

Cloud computing, or simply the Cloud, is a simple way to rent and use resources such as electronic storage space, computing power, network bandwidth, IP addresses, databases, web servers, and so on, on the internet. The Cloud has promoted the *pay per use* paradigm, where customers are only billed for the use of these resources, in the same way that a power grid bills its customers for their power consumption.

Cloud computing has transformed the way individuals and organizations access and manage their servers and applications on the internet. Before Cloud computing, everyone used to manage their servers and applications on their own premises or in dedicated data centers. The increase in the raw computing power of computing (CPU and GPU) of multiple-cores on a single chip and the increase in the storage space (HDD and SSD) present challenges in efficiently utilizing the available computing resources.

Available technologies in the Cloud

With the increased adoption of Cloud computing, enterprises have started building a variety of technologies and making them available to consumers. We will go through the list of organizations that have pioneered Cloud offerings, and also the different types of technologies they offer.

Here is a list of companies that offer Cloud services:

- Microsoft Azure (Azure)
- Amazon Web Services
- Google Cloud Platform
- IBM
- Salesforce
- SAP
- Oracle
- VMware

Various types of resources are being offered to the consumers in the form of:

- Platform as a Service
- Infrastructure as a Service
- Software as a Service
- Backend as a Service
- Network as a Service

With the increase in offerings such as these, many organizations need not focus on the infrastructure such as real estate, servers, firewalls, load balancers, switches, power supply, and so on. But they can instead just purchase these services from Cloud providers, and then just focus on what applications they are building.

Now, let's see what technologies are provided by the top providers, Microsoft, Amazon, and Google:

Technology	**Azure**	**Amazon Web Services**	**Google Cloud**	**Description**
Servers	Azure compute	Amazon EC2	**Google Compute Engine** (**GCE**)	This technology deals with providing servers that are on demand and that can be virtualized or dedicated/baremetal in nature.
Storage	Azure storage	Amazon EBS	Google storage	This is on-demand storage that can be attached to the compute nodes as needed. Some vendors provide the ability to scale the size of these storage devices on demand.
Network	Azure networking	Yes	Google network services	Providers supply network bandwidth from 100 Mbps to 10 Gbps, depending on the network requirements of the applications.

Databases	Azure databases	Amazon RDS	Google Cloud SQL	With managed databases, we need not worry about the maintenance of the database servers as the vendors take care of the support for these automatically. Bear in mind that, in some cases, we need to plan the high availability for ourselves.
Content delivery	Azure CDN	Amazon CloudFront	Google Cloud CDN	This is very helpful if we want to push our static assets to our users by leveraging the delivery network as it brings down the latency significantly. We can also use this as a private store to store all the files such as backups, conference recordings, and so on.
Domain Name System (**DNS**)	Azure DNS	Amazon Route S3	Google Cloud DNS	DNS is critical in running our applications on the internet. This service makes our life easier by taking care of making our servers accessible to the rest of the infrastructure, without having to run our own DNS servers.
Business mail	Microsoft o365	Amazon WorkMail	Google Mail	This is a must-have for organizations that demand access to emails and calendaring in a secure and scalable fashion.

Machine learning	Azure AI + machine learning	Amazon machine learning	Google ML Engine	Machine learning technology has become the buzzword these days. Vendors are offering several technologies that are related to machine learning, as we just have to focus on what we need to do, rather than worrying about the infrastructure that needs to run these algorithms.
Distributed Denial of Service (**DDoS**) Protection	Azure DDoS Protection	AWS Shield	–	This is a very important thing to have for organizations that cannot afford to have downtime for their services and when large-scale denial of service attacks happen that impact regular visitors of their sites.
Monitoring	Azure Monitor	Amazon CloudWatch	Google monitoring	Without monitoring our applications and infrastructure, we can fail to see how we are performing. These services help us keep our business on track and to respond to events that trigger downtime of our applications, and infrastructure that runs on the Cloud.
Containers	**Azure Container Service** (**AKS**)	**Amazon Elastic Container Service For Kubernetes** (**Amazon EKS**)	Google Kubernetes Engine	This is infrastructure that allows you to run applications as containers, rather than owning full compute environments to run them.

Planning the Cloud infrastructure

Traditional organizations have their own IT/infrastructure teams to manage their dedicated servers and network. When planning migration to the Cloud, we have to keep the following things in mind for better operability of the infrastructure.

Planning the Cloud infrastructure deals with:

- Dedicated or shared servers
- High availability
- Business continuity planning
- Security
- Network architecture

Dedicated servers versus shared servers

Cloud providers give you the option of renting servers that completely own the physical hardware or share the physical hardware with other Cloud users like us. In order to reach a decision on this, we need to understand the advantages and disadvantages of each of these models.

Dedicated servers

These are the type of servers where the type of ownership belongs to a single user or an organization and is not shared with any other user. There are several advantages to this setup, as follows:

- We completely own the physical server and any further servers that we allocate will be provisioned on the same hardware
- We might be billed more for this kind of setup
- With Spectre and Meltdown, we are better protected as the hardware is not shared with anyone
- We are not impacted by the neighbors as we completely own the hardware

Shared servers

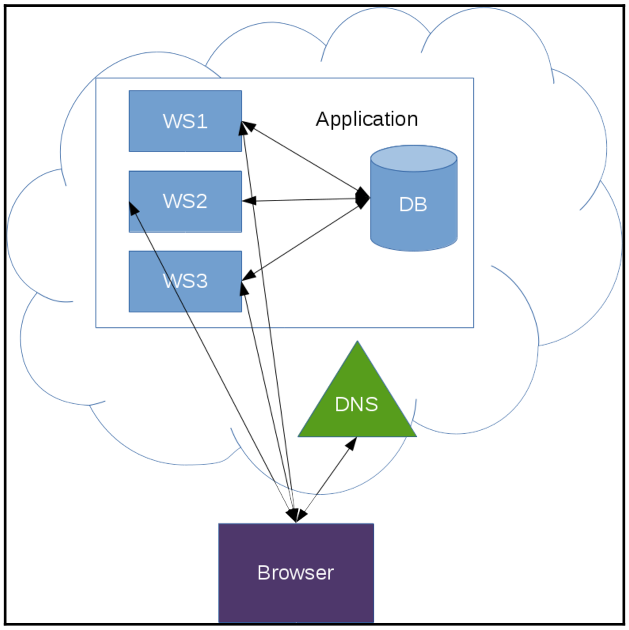
Owning a complete server is expensive for simple experimentation. In this scenario, we can go for a shared setup where we rent a few resources in a given physical hardware. Some advantages of shared servers are as follows:

- We are billed only for the virtual servers that we rent on demand.
- Even though Cloud vendors provide absolute isolation, with Spectre and Meltdown, we need to be a bit careful.
- Easier to provision than dedicated servers.

High availability

Depending on the type of applications we are planning to run, we have to understand the **service-level agreement** (**SLA**) that is provided by the vendors for these applications in terms of uptime, and we need to plan our applications accordingly.

Let's look at a simple way of using DNS to achieve high availability of our application:



In this design, the following things happen:

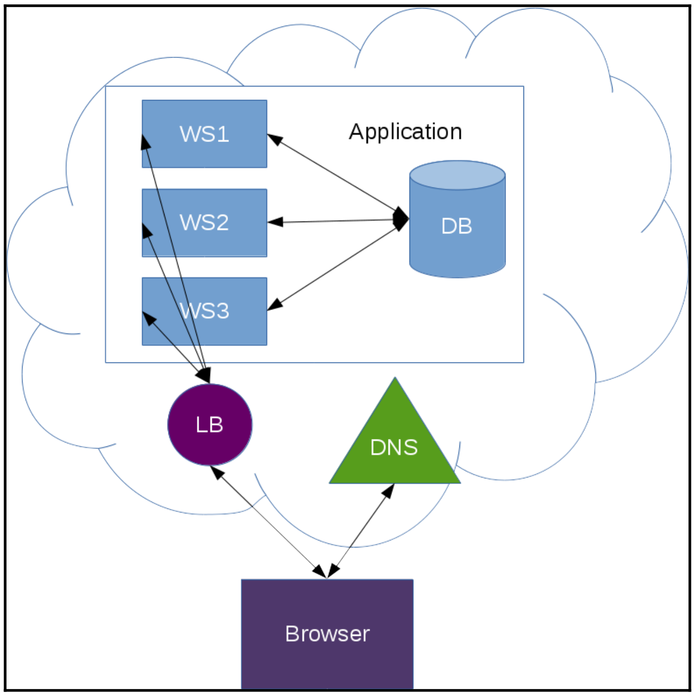
- When the user tries to connect to our website using a web browser such as Google Chrome or Firefox, it first tries to contact the DNS server
- The DNS server is aware of our frontend servers and returns a list of all the servers
- The browser will connect to the frontend server directly
- The frontend server connects to the database and returns the requested resource

In this design, we need to keep the following things in mind:

- Frontend servers are directly exposed to the internet, so we should have proper security measures in place such as a firewall or DDos protection to protect our servers
- These frontend servers should also be patched with the latest OS software so that any attacks can be prevented
- A database server should not be visible to the outside world, so an appropriate firewall should be in place to allow requests from the frontend servers

Cloud providers provide a private IP address. In order to minimize the risk of DB servers being accidentally exposed to the internet, we should block the public internet access to these servers.

Let's look at another design that also keeps our web servers secure from attacks on the internet:



In this design, we have made the following changes when compared to the previous one:

- When the **Browser** contacts the **DNS** server to connect to our website, the **DNS** server supplies the IP address of the **Load Balancer** (**LB**)/proxy server
- The browser connects to this **LB**
- The **LB** keeps track of which of the backend servers are available and then forwards the request to the server:
    - The server talks to the **database** (**DB**) and finishes building the response
    - The response is sent back to the **LB**
- The **LB** sends the response to the **Browser**

If we look at this design carefully, we will see that these are the advantages over the previous one:

- The **LB** hides our infrastructure, so outsiders cannot easily know how many servers are there in our infrastructure

- The **LB** protects our web servers from several attacks
- The **LB** can do SSL offloading where all the encryption/decryption of traffic happens at the **LB** level and our servers can be free from the SSL overhead

Depending on the security policy of the organization, you might need to enable SSL on the web servers as well.

Business continuity planning

Business continuity planning (**BCP**) is a very important thing to consider when the organization is in its growth phase. Any downtime of the network, servers or databases, or any other Cloud infrastructure components can bring down the whole business.

There are several key things to keep in mind when planning for BCP:

- Infrastructure unavailability
- Natural disasters
- Business data

Infrastructure unavailability

If there is an unplanned outage of the services provided by the Cloud provider, it will take down all our services with it. In order to maximize the availability of our business, we need to build a backup setup in another geographical region. This might be expensive for some organizations as the entire setup is going to be duplicated, but in the interest of business continuity, this is an important feature to consider when planning the Cloud infrastructure.

Natural disasters

Events such as earthquakes, floods, fire accidents, and so on are hard to predict. We therefore need to make the necessary plans to keep our business running, depending on where our servers are located on the Cloud, and on what technology standards are followed by the vendors for the data center build-out.

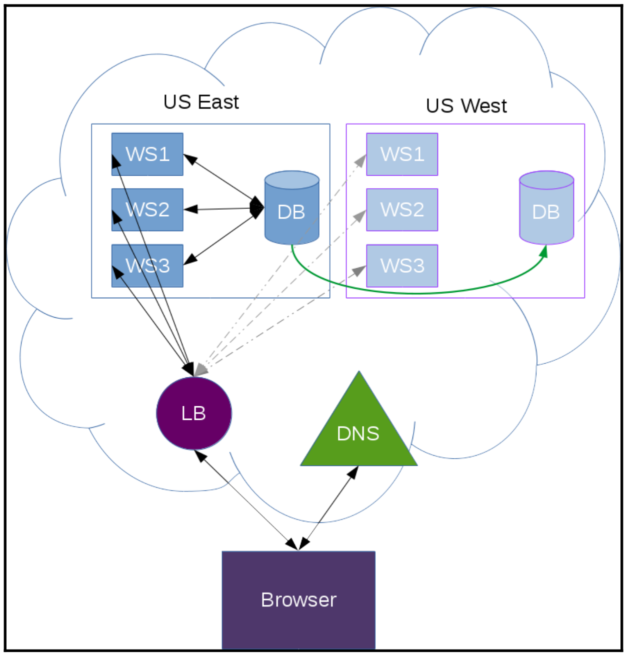
Business data

Business data exists in several forms and is stored in the form of files, database servers, and big data systems. For BCP, we need to carefully analyze in what other remote locations we can plan to keep the copies of our data, and carry out test runs to see if our applications can be seamlessly run from either of the geographical locations with a single click of a button.

As we are dealing with multiple geographies here, we need to understand that, when the volume of data is huge, it takes time for things to get replicated from one data center to another. Our applications must also be redesigned in case we did not consider BCP in the original design.

BCP design example

This diagram tries to explain how we can achieve BCP by setting up the same applications in multiple data centers:



The system can be either:

- Hot–Hot
- Hot–Cold

The Hot–Hot system

In the Hot-Hot system, both the data centers are active at the same time and serve the user's traffic. Here, we employ several CDN and geolocation techniques to route the user to a given data center.

The challenge we face in doing so is that if one region goes completely blank, the other region should have enough headroom to ensure that traffic for the other region is absorbed

The advantage of employing this system is that the user experience is a good one, as in this design the users are routed to the nearest system

The Hot–Cold system

In this system/design, only one of the regions is active at any time and only in the event of BCP (Business Continuity Planning) do we fall back to the other region.

The challenges we face in using this system are as follows:

- It's easy to forget the other region until the problem comes into play; it's very important to keep using both the regions in sync w.r.t both Data & Software on a continuous basis.
- As only one region is active, the correct failover of users to another data center has to be thought through well

The advantage of employing this system is that all the writes happen in one region which keeps database designs simple.

Security

Security is very important when you consider moving to the Cloud. The following are the things to keep in mind:

- Server security
- Application security

- Network security
- Single Sign On
- AAA requirements

Server security

As we are talking about the Cloud, we will never be able to access the servers physically (unless we get permission from the Cloud vendors). In such a scenario, we have to understand what level of policies and practices are followed by the Cloud providers to ensure the physical security of the servers on which our applications are going to be run.

For example, governments might need a whole set of different physical security restrictions when considering a move to the Cloud. On the same lines, there are several standards such as PCI and HIPAA which enforce even stronger rules on this model.

If our business needs to adhere to these standards, we need to choose the Cloud variant which supports all these.

Application security

On the Cloud, we can either choose to host the applications on our own or use the applications provided as a service **Software As A Service** (**SaaS**). If we are hosting the applications on our own provisioned servers (either dedicated or shared), we need to enforce the correct firewall rules at server level, and also the correct user access rules to make sure that our software allows only authorized and properly authenticated users.

If the applications are internal, we should ensure that our employees are given 2FA or 3FA methods to log in to these services.

Network security

In order to safeguard our servers on the Cloud, we need to enforce proper firewall rules, DNS zones, or even have our own virtual private networks to make sure all our assets are not compromised and exposed to the internet.

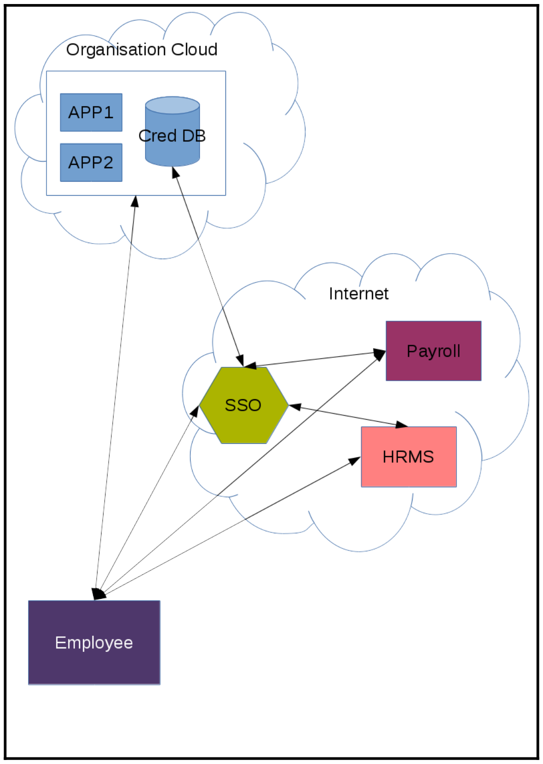
The Cloud is synonymous with the internet and there is a continuous threat to our data and infrastructure. Unless we enforce proper security measures, everything is wide open for everyone to grab whatever they like from our systems.

Single Sign On

Single Sign On (**SSO**) has become popular with organizations that use several applications on the Cloud for various departments. Lately, organizations have stopped building their own applications for running businesses and instead have started adopting the use of other services. When the number of such applications increases, users of these applications are constantly faced with the challenge of entering their usernames and passwords in all these websites.

In order to provide a seamless browsing experience, and at the same time adhere to enterprise security standards, many providers implement OAuth and SAML, as they are industry recognized.

These SSO/identity providers integrate with the corporate employee database to further assimilate the Cloud applications for the enterprise, as depicted here:



This design tries to explain how organizations are leveraging SSO using identity providers:

- Organizations share the employee and organization details with the identity provider:
    - Passwords may or may not be shared as it can compromise the entire organization if there is a breach
    - SSO systems can enforce their own passwords on the employees
- When the user tries to open any of the applications in the organization, it redirects the user to the SSO provider
- The SSO provider completes the authentication and shares necessary credentials with the application
- The application authorizes the user based on the feedback from the SSO
- The application opens the user specific details and then the user can interact with the application

Now, the biggest advantage of these SSOs is that once the user has established a session with the system, they can log in to other corporate-approved applications without further logins.

Confidentiality is the biggest challenge when interacting with SSO providers, so organizations should carefully evaluate and pick the right solution that meets their security requirements.

The AAA requirement

When it comes to security, it is important to understand that applications following the AAA standard will take care of many challenges that enterprises face.

The AAA standard deals with:

- Authentication
- Authorization
- Auditing

Authentication makes sure that the identity of the user is properly validated.

Authorization further controls whether a given user is allowed to access certain resources or not, as per company policies.

Auditing makes sure that all attempts to access and use the resources are tracked—this can also be used in case of any investigation, and provide proper accounting and billing (if needed).

By following these best practices, we can be sure that things run smoothly on a large scale.

Building a Hadoop cluster in the Cloud

We saw earlier that the Cloud offers a flexible and easy way to rent resources such as servers, storage, networking, and so on. The Cloud has made it very easy for consumers with the pay-as-you-go model, but much of the complexity of the Cloud is hidden from us by the providers.

In order to better understand whether Hadoop is well suited to being on the Cloud, let's try to dig further and see how the Cloud is organized internally.

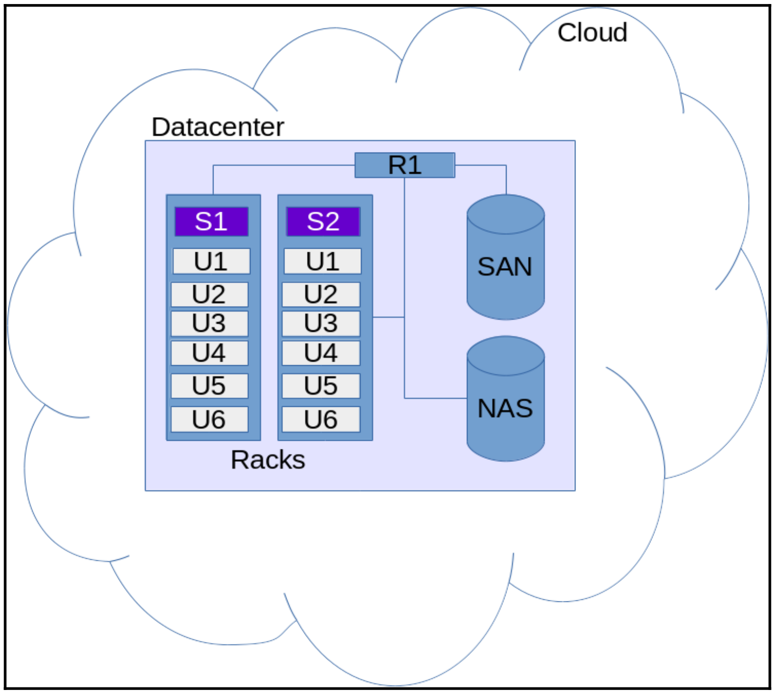
At the core of the Cloud are the following mechanisms:

- A very large number of servers with a variety of hardware configurations
- Servers connected and made available over IP networks
- Large data centers to host these devices
- Data centers spanning geographies with evolved network and data center designs

If we pay close attention, we are talking about the following:

- A very large number of different CPU architectures
- A large number of storage devices with a variety of speeds and performance
- Networks with varying speed and interconnectivity

Let's look at a simple design of such a data center on the Cloud:



We have the following devices in the preceding diagram:

- **S1**, **S2**: Rack switches
- **U1-U6**: Rack servers
- **R1**: Router
- Storage area network
- Network attached storage

As we can see, Cloud providers have a very large number of such architectures to make them scalable and flexible.

You would have rightly guessed that when the number of such servers increases and when we request a new server, the provider can allocate the server anywhere in the region.

This makes it a bit challenging for compute and storage to be together but also provides elasticity.

In order to address this co-location problem, some Cloud providers give the option of creating a virtual network and taking dedicated servers, and then allocating all their virtual nodes on these servers. This is somewhat closer to a data center design, but flexible enough to return resources when not needed.

Let's get back to Hadoop and remind ourselves that in order to get the best from the Hadoop system, we should have the CPU power closer to the storage. This means that the physical distance between the CPU and the storage should be much less, as the BUS speeds match the processing requirements.

The slower the I/O speed between the CPU and the storage (for example, iSCSI, storage area network, network attached storage, and so on) the poorer the performance we get from the Hadoop system, as the data is being fetched over the network, kept in memory, and then fed to the CPU for further processing.

This is one of the important things to keep in mind when designing Hadoop systems on the Cloud.

Apart from performance reasons, there are other things to consider:

- Scaling Hadoop
- Managing Hadoop
- Securing Hadoop

Now, let's try to understand how we can take care of these in the Cloud environment.

In the previous chapters, we saw that Hadoop can be installed by the following methods:

- Standalone
- Semi-distributed
- Fully-distributed

When we want to deploy Hadoop on the Cloud, we can deploy it using the following ways:

- Custom shell scripts
- Cloud automation tools (Chef, Ansible, and so on)
- Apache Ambari
- Cloud vendor provided methods
    - Google Cloud Dataproc
    - Amazon EMR
    - Microsoft HDInsight
- Third-party managed Hadoop
    - Cloudera
- Cloud agnostic deployment
    - Apache Whirr

Google Cloud Dataproc

In this section, we will learn how to use Google Cloud Dataproc to set up a single node Hadoop cluster.

The steps can be broken down into the following:

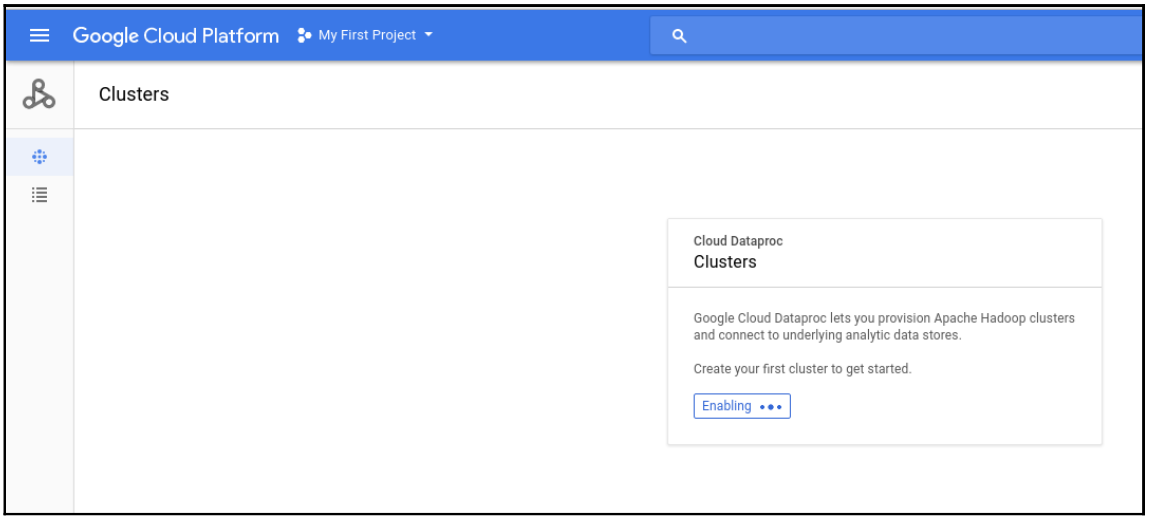
1. Getting a Google Cloud account.
2. Activating Google Cloud Dataproc service.
3. Creating a new Hadoop cluster.
4. Logging in to the Hadoop cluster.
5. Deleting the Hadoop cluster.

Getting a Google Cloud account

This section assumes that you already have a Google Cloud account.

Activating the Google Cloud Dataproc service

Once you log in to the Google Cloud console, you need to visit the Cloud Dataproc service. The activation screen looks something like this:



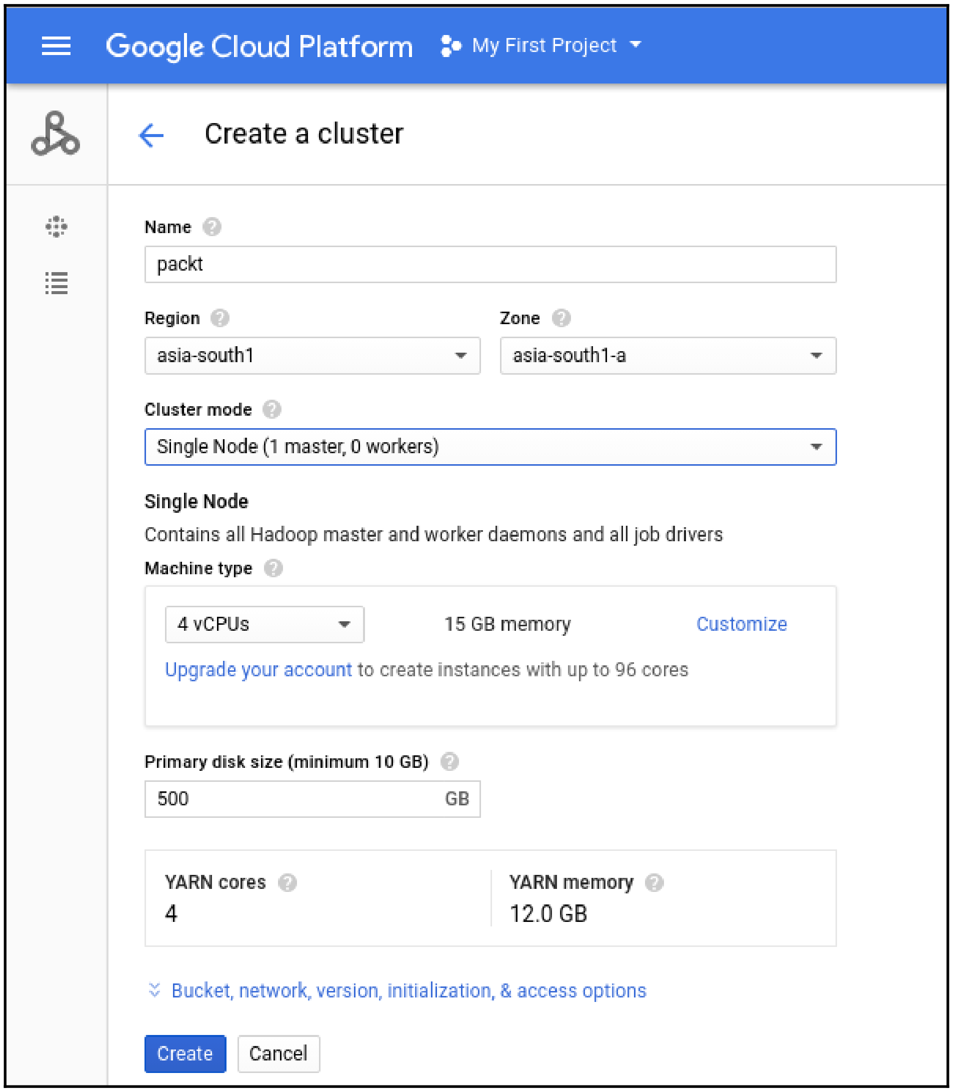
Google Cloud Dataproc lets you provision Apache Hadoop clusters and connect to underlying analytic data stores.

Create your first cluster to get started.

Creating a new Hadoop cluster

Once the Dataproc is enabled in the project, we can click on **Create** to create a new Hadoop cluster.

After this, we see another screen where we need to configure the cluster parameters:



Name: packt

Region: asia-south1

Zone: asia-south1-a

Cluster mode: Single Node (1 master, 0 workers)

Single Node

Contains all Hadoop master and worker daemons and all job drivers

Machine type: 4 vCPUs 15 GB memory Customize

Upgrade your account to create instances with up to 96 cores

Primary disk size (minimum 10 GB): 500 GB

YARN cores: 4

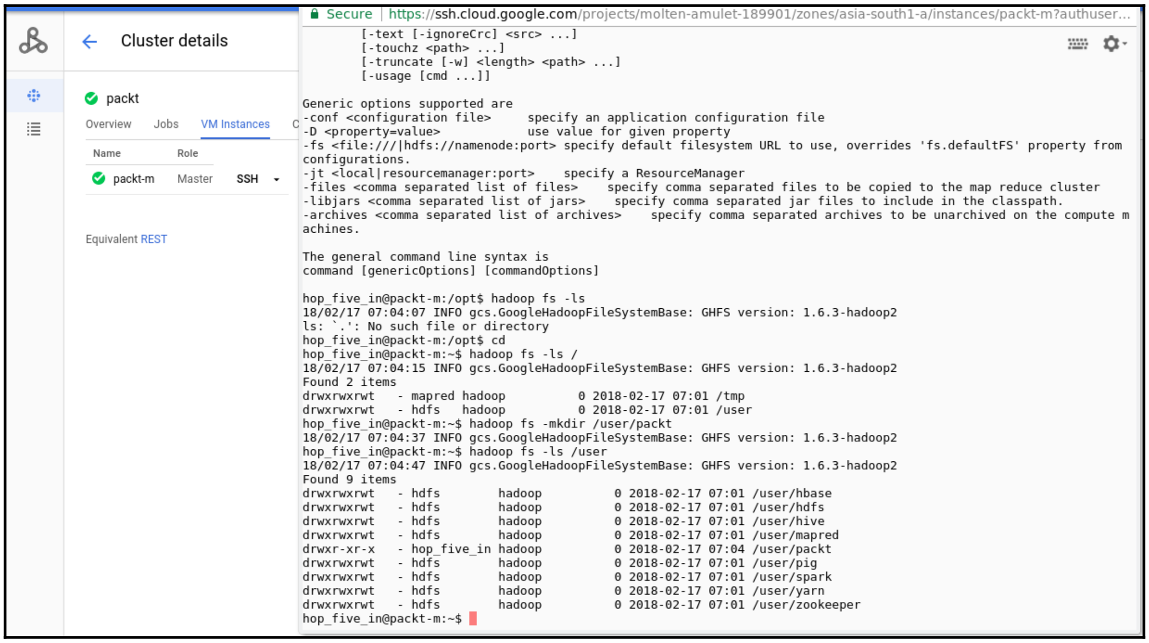
YARN memory: 12.0 GB

I have left most of the things to their default values. Later, we can click on the **Create** button which creates a new cluster for us.

Logging in to the cluster

After the cluster has successfully been created, we will automatically be taken to the cluster lists page. From there, we can launch an SSH window to log in to the single node cluster we have created.

The SSH window looks something like this:

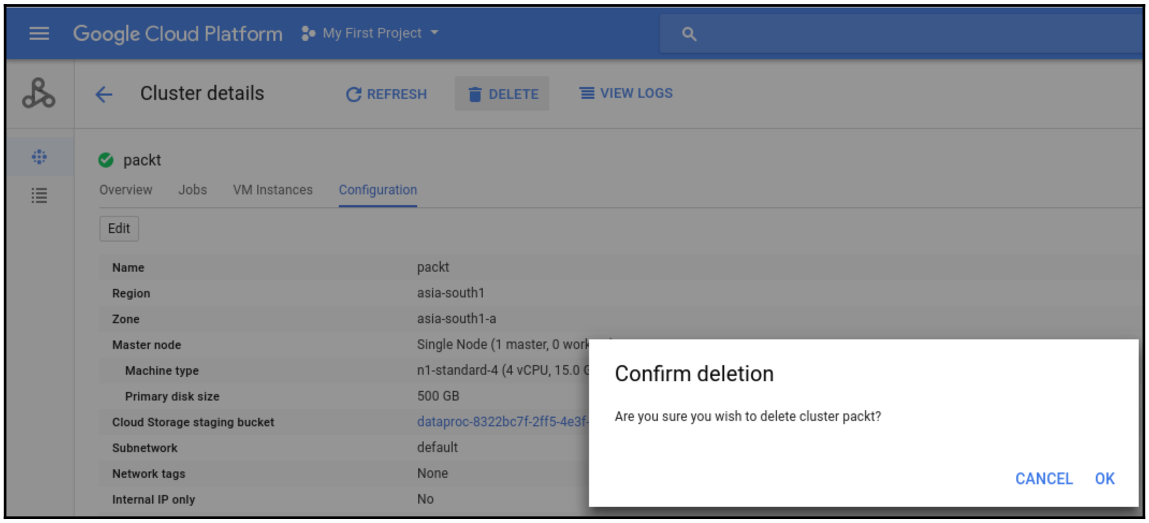

Cluster details
packt
Overview
Jobs
VM Instances
Name
Role
packt-m
Master
SSH
Equivalent REST
Secure
https://ssh.cloud.google.com/projects/molten-amulet-189901/zones/asia-south1-a/instances/packt-m?authuser...

```
        [-text [-ignoreCrc] <src> ...]
        [-touchz <path> ...]
        [-truncate [-w] <length> <path> ...]
        [-usage [cmd ...]]

Generic options supported are
-conf <configuration file>     specify an application configuration file
-D <property=value>            use value for given property
-fs <file:///|hdfs://namenode:port> specify default filesystem URL to use, overrides 'fs.defaultFS' property from
configurations.
-jt <local|resourcemanager:port>    specify a ResourceManager
-files <comma separated list of files>    specify comma separated files to be copied to the map reduce cluster
-libjars <comma separated list of jars>    specify comma separated jar files to include in the classpath.
-archives <comma separated list of archives>    specify comma separated archives to be unarchived on the compute m
achines.

The general command line syntax is
command [genericOptions] [commandOptions]

hop_five_in@packt-m:/opt$ hadoop fs -ls
18/02/17 07:04:07 INFO gcs.GoogleHadoopFileSystemBase: GHFS version: 1.6.3-hadoop2
ls: `.': No such file or directory
hop_five_in@packt-m:/opt$ cd
hop_five_in@packt-m:~$ hadoop fs -ls /
18/02/17 07:04:15 INFO gcs.GoogleHadoopFileSystemBase: GHFS version: 1.6.3-hadoop2
Found 2 items
drwxrwxrwt   - mapred hadoop          0 2018-02-17 07:01 /tmp
drwxrwxrwt   - hdfs   hadoop          0 2018-02-17 07:01 /user
hop_five_in@packt-m:~$ hadoop fs -mkdir /user/packt
18/02/17 07:04:37 INFO gcs.GoogleHadoopFileSystemBase: GHFS version: 1.6.3-hadoop2
hop_five_in@packt-m:~$ hadoop fs -ls /user
18/02/17 07:04:47 INFO gcs.GoogleHadoopFileSystemBase: GHFS version: 1.6.3-hadoop2
Found 9 items
drwxrwxrwt   - hdfs        hadoop          0 2018-02-17 07:01 /user/hbase
drwxrwxrwt   - hdfs        hadoop          0 2018-02-17 07:01 /user/hdfs
drwxrwxrwt   - hdfs        hadoop          0 2018-02-17 07:01 /user/hive
drwxrwxrwt   - hdfs        hadoop          0 2018-02-17 07:01 /user/mapred
drwxr-xr-x   - hop_five_in hadoop          0 2018-02-17 07:04 /user/packt
drwxrwxrwt   - hdfs        hadoop          0 2018-02-17 07:01 /user/pig
drwxrwxrwt   - hdfs        hadoop          0 2018-02-17 07:01 /user/spark
drwxrwxrwt   - hdfs        hadoop          0 2018-02-17 07:01 /user/yarn
drwxrwxrwt   - hdfs        hadoop          0 2018-02-17 07:01 /user/zookeeper
hop_five_in@packt-m:~$
```

As you can see, the Hadoop command is readily available for us and we can run any of the standard Hadoop commands to interact with the system.

Deleting the cluster

In order to delete the cluster, click on the **DELETE** button and it will display a confirmation window, as shown in the following screenshot. After this, the cluster will be deleted:



packt

Name	packt
Region	asia-south1
Zone	asia-south1-a
Master node	Single Node (1 master, 0 work
Machine type	n1-standard-4 (4 vCPU, 15.0 G
Primary disk size	500 GB
Cloud Storage staging bucket	dataproc-8322bc7f-2ff5-4e3f-
Subnetwork	default
Network tags	None
Internal IP only	No

Confirm deletion

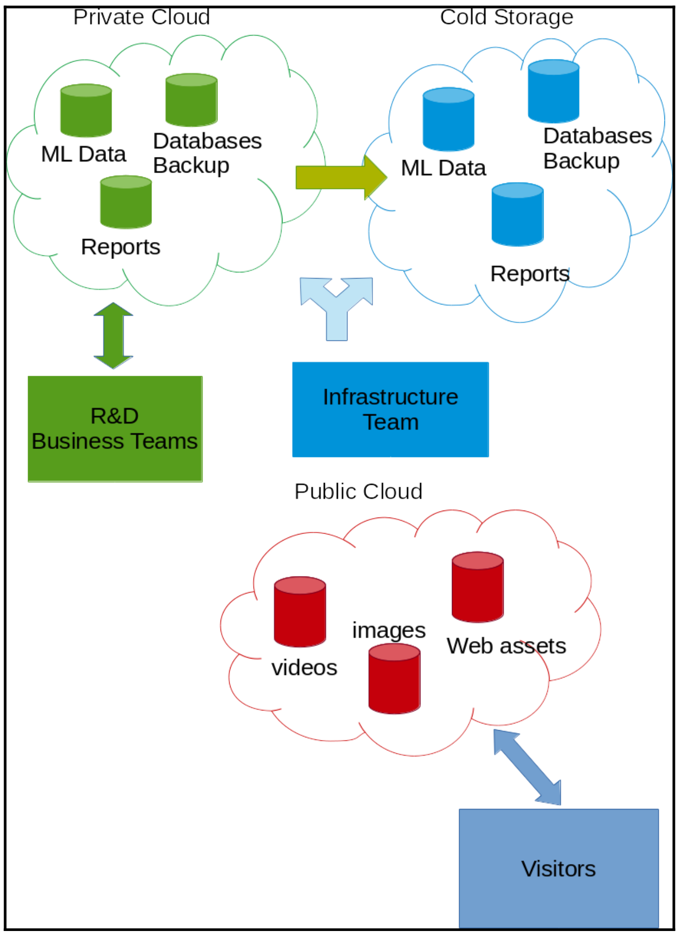
Are you sure you wish to delete cluster packt?

Looks so simple, right? Yes. Cloud providers have made it very simple for users to use the Cloud and pay only for the usage.

Data access in the Cloud

The Cloud has become an important destination for storing both personal data and business data. Depending upon the importance and the secrecy requirements of the data, organizations have started using the Cloud to store their vital datasets.

The following diagram tries to summarize the various access patterns of typical enterprises and how they leverage the Cloud to store their data:



Cloud providers offer different varieties of storage. Let's take a look at what these types are:

- Block storage
- File-based storage
- Encrypted storage
- Offline storage

Block storage

This type of storage is primarily useful when we want to use this along with our compute servers, and want to manage the storage via the host operating system.

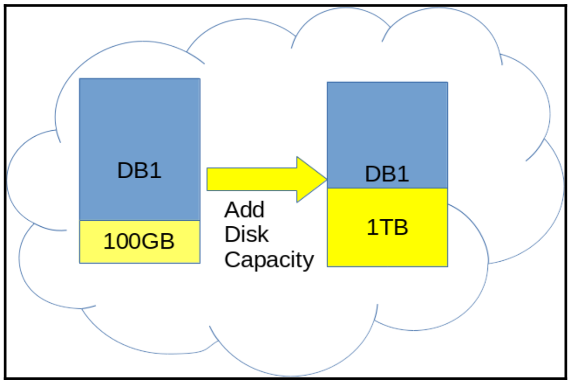
To understand this better, this type of storage is equivalent to the hard disk/SSD that comes with our laptops/MacBook when we purchase them. In case of laptop storage, if we decide to increase the capacity, we need to replace the existing disk with another one.

When it comes to the Cloud, if we want to add more capacity, we can just purchase another larger capacity storage and attach it to our server. This is one of the reasons why the Cloud has become popular as it has made it very easy to add or shrink the storage that we need.

It's good to remember that, since there are many different types of access patterns for our applications, Cloud vendors also offer block storage with varying storage/speed requirements measured with their own capacity/IOPS, and so on.

Let's take an example of this capacity upgrade requirement and see what we do to utilize this block storage on the Cloud.

In order to understand this, let's look at the example in this diagram:



Imagine a server created by the administrator called **DB1** with an original capacity of **100 GB**. Later, due to unexpected demand from customers, an application started consuming all the **100 GB** of storage, so the administrator has decided to increase the capacity to **1 TB** (1,024 GB).

This is what the workflow looks like in this scenario:

1. Create a new 1 TB disk on the Cloud
2. Attach the disk to the server and mount it
3. Take a backup of the database
4. Copy the data from the existing disk to the new disk
5. Start the database
6. Verify the database
7. Destroy the data on the old disk and return the disk

This process is simplified but in production this might take some time, depending upon the type of maintenance that is being performed by the administrator. But, from the Cloud perspective, acquiring new block storage is very quick.

File storage

Files are the basics of computing. If you are familiar with UNIX/Linux environments, you already know that, *everything is a file* in the Unix world. But don't get confused with that as every operating system has its own way of dealing with hardware resources. In this case we are not worried about how the operating system deals with hardware resources, but we are talking about the important documents that the users store as part of their day-to-day business.

These files can be:

- Movie/conference recordings
- Pictures
- Excel sheets
- Word documents

Even though they are simple-looking files in our computer, they can have significant business importance and should be dealt with in a careful fashion, when we think of storing these on the Cloud.

Most Cloud providers offer an easy way to store these simple files on the Cloud and also offer flexibility in terms of security as well.

A typical workflow for acquiring the storage of this form is like this:

1. Create a new storage bucket that's uniquely identified
2. Add private/public visibility to this bucket
3. Add multi-geography replication requirement to the data that is stored in this bucket

Some Cloud providers bill their customers based on the number of features they select as part of their bucket creation.

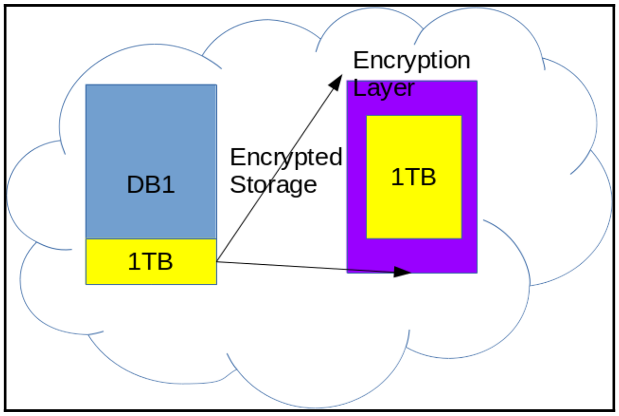
TIP

Please choose a hard-to-discover name for buckets that contain confidential data, and also make them private.

Encrypted storage

This is a very important requirement for business critical data as we do not want the information to be leaked outside the scope of the organization. Cloud providers offer an encryption at rest facility for us. Some vendors choose to do this automatically and some vendors also provide flexibility in letting us choose the encryption keys and methodology for the encrypting/decrypting data that we own. Depending upon the organization policy, we should follow best practices in dealing with this on the Cloud.

With the increase in the performance of storage devices, encryption does not add significant overhead while decrypting/encrypting files. This is depicted in the following image:



Continuing the same example as before, when we choose to encrypt the underlying block storage of **1 TB,** we can leverage the Cloud-offered encryption where they automatically encrypt and decrypt the data for us. So, we do not have to employ special software on the host operating system to do the encryption and decryption.

Remember that encryption can be a feature that's available in both the block storage and file-based storage offer from the vendor.

Cold storage

This storage is very useful for storing important backups in the Cloud that are rarely accessed. Since we are dealing with a special type of data here, we should also be aware that the Cloud vendor might charge significantly high amounts for data access from this storage, as it's meant to be written once and forgetten (until it's needed). The advantage with this mechanism is that we have to pay lesser amounts to store even petabytes of data.

Summary

In this chapter, we looked at what Cloud computing means and saw how the Cloud has completely revolutionized how we can access and manage our servers and applications on the internet. We then walked through a list of different technologies offered on the Cloud by different providers.

We also learned how to plan our Cloud infrastructure and looked at the different steps involved in building our own Hadoop cluster on the Cloud. Finally, we saw different ways of storing and accessing our data on the Cloud.

In the next chapter we will take a look at some strategies and best practices to deploy your Hadoop cluster.

11
Production Hadoop Cluster Deployment

Hadoop itself started with a strong core and File System designed to handle the big data challenges. Later, many applications were developed on top of this, creating a big ecosystem of applications that play nicely with each other. As the number of applications started increasing, the challenges to create and manage the Hadoop environment increased as well.

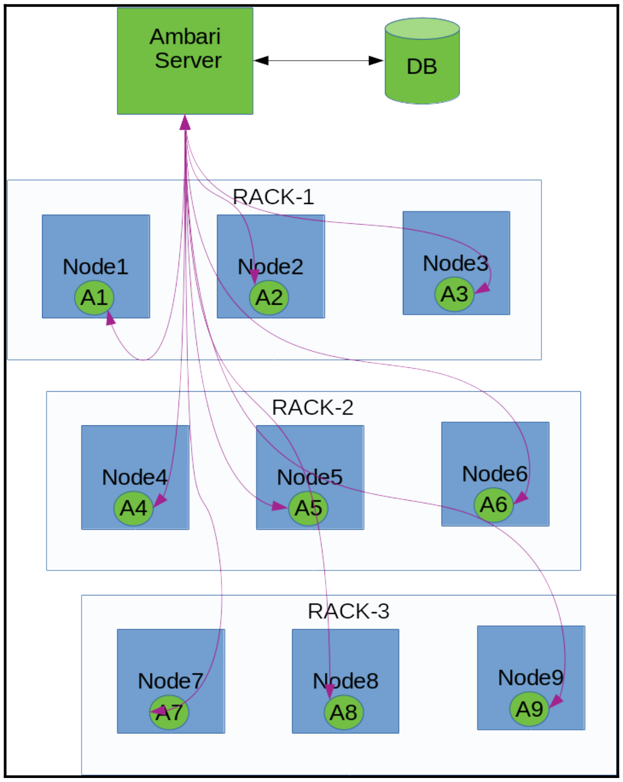
In this chapter, we will look at the following:

- Apache Ambari
- A Hadoop cluster with Ambari

Apache Ambari architecture

Apache Ambari follows a master/slave architecture where the master node instructs the slave nodes to perform certain actions and report back the state of every action. The master node is responsible for keeping track of the state of the infrastructure. In order to do this, the master node uses a database server, which can be configured during setup time.

In order to have a better understanding of how Ambari works, let's take a look at the high level architecture of Ambari, in the following diagram:



At the core, we have the following applications:

- Ambari server
- Ambari agent
- Ambari web UI
- Database

The Ambari server

The Ambari server (`ambari-server`) is a shell script which is the entry point for all administrative activities on the master server. This script internally uses Python code, `ambari-server.py`, and routes all the requests to it.

The Ambari server has the following entry points which are available when passed different parameters to the `ambari-server` program:

- Daemon management
- Software upgrade
- Software setup
- LDAP/PAM/Kerberos management
- Ambari backup and restore
- Miscellaneous options

Daemon management

The daemon management mode is activated when the script is invoked with `start`, `stop`, `reset`, `restart` arguments from the command line.

For example, if we want to start the Ambari background server, we can run the following command:

```
Example: ambari-server start
```

Software upgrade

Once Ambari is installed, we can use this mode to upgrade the Ambari server itself. This is triggered when we call the `ambari-server` program with the `upgrade` flag. In case we want to upgrade the entire stack of Ambari, we can pass the `upgradestack` flag:

```
Example: ambari-server upgrade
```

Software setup

Once Ambari is downloaded from the internet (or installed via YUM and APT), we need to do a preliminary setup of the software. This mode can be triggered when we pass the `setup` flag to the program. This mode will ask us several questions that we need to answer. Unless we finish this step, Ambari cannot be used for any kind of management of our servers:

```
Example: ambari-server setup
```

LDAP/PAM/Kerberos management

The **Lightweight Directory Access Protocol** (**LDAP**) is used for identity management in enterprises. In order to use LDAP-based authentication, we need to use the following flags: `setup-ldap` (for setting up `ldap` properties with `ambari`) and `sync-ldap` (to perform a synchronization of the data from the `ldap` server):

```
Example: ambari-server setup-ldap
Example: ambari-server sync-ldap
```

Pluggable Authentication Module (**PAM**) is at the core of the authentication and authorization in any UNIX or Linux operating systems. If we want to leverage the PAM-based access for Ambari then we need to run it with the `setup-pam` option. If we then want to move from LDAP to PAM-based authentication, we need to run it with `migrate-ldap-pam`:

```
Example: ambari-server setup-pam
Example: ambari-server migrate-ldap-pam
```

Kerberos is another advanced authentication and authorization mechanism which is very helpful in networked environments. This simplifies **Authenticity, Authorisation and Auditing** (**AAA**) on large-scale servers. If we want to use Kerberos for Ambari, we can use the `setup-kerberos` flag:

```
Example: ambari-server setup-kerberos
```

Ambari backup and restore

If we want to take a snapshot of the current installation of Ambari (excluding the database), we can enter this mode. This supports both backup and restore methods invoked via the `backup` and `restore` flags:

```
Example: ambari-server backup
Example: ambari-server restore
```

Miscellaneous options

In addition to these options, there are other options that are available with the Ambari server program which you can invoke with the `-h` (help) flag.

Ambari Agent

Ambari Agent is a program which runs on all the nodes that we want to be managed with Ambari. This program periodically heartbeats to the master node. Using this agent, `ambari-server` executes many of the tasks on the servers.

Ambari web interface

This is one of the powerful features of the Ambari application. This web application is exposed by the Ambari server program that is running on the master host; we can access this application on port `8080` and it is protected by authentication.

Once we log in to this web portal, we can control and view all aspects of our Hadoop clusters.

Database

Ambari supports multiple RDBMS to keep track of the state of the entire Hadoop infrastructure. During the setup of the Ambari server for the first time, we can choose the database we want to use.

At the time of writing, Ambari supports the following databases:

- PostgreSQL
- Oracle
- MySQL or MariaDB
- Embedded PostgreSQL
- Microsoft SQL Server
- SQL Anywhere
- Berkeley DB

Setting up a Hadoop cluster with Ambari

In this section, we will learn how to set up a brand new Hadoop cluster from scratch using Ambari. In order to do this, we are going to need four servers – one server for running the Ambari server and three other nodes for running the Hadoop components.

Server configurations

The following table displays the configurations of the servers we are using as part of this exercise:

Server Type	**Name**	**CPU**	**RAM**	**DISK**
Ambari Server node	master	1	3.7 GB	100 GB
Hadoop node 1	node-1	2	13 GB	250 GB
Hadoop node 2	node-2	2	13 GB	250 GB
Hadoop node 3	node-3	2	13 GB	250 GB

Since this is a sample setup, we are good with this configuration. For real-world scenarios, please choose the configuration according to your requirements.

Preparing the server

This section and all further sections assume that you have a working internet connection on all the servers and are safely firewalled to prevent any intrusions.

All the servers are running the CentOS 7 operating system, as it's a system that uses RPM/YUM for package management. Don't get confused when you see `yum` in the following sections.

Before we go ahead and start using the servers, we need to run basic utility programs which help us troubleshoot various issues with the servers. They are installed as part of the next command. Don't worry if you are not sure what they are. Except for `mysql-connector-java` and `wget`, all other utilities are not mandatory:

```
sudo yum install mysql-connector-java wget iftop iotop smartctl -y
```

Installing the Ambari server

The first step in creating the Hadoop cluster is to get our Ambari server application up and running. So, log in to the master node with SSH and perform the following steps in order:

1. Download the Ambari YUM repository for CentOS 7 with this command:

```
[user@master ~]$ wget
http://public-repo-1.hortonworks.com/ambari/centos7/2.x/updates/2.6
.1.5/ambari.repo
```

2. After this step, we need to move the `ambari.repo` file to the `/etc/yum.repos.d` directory using this command:

```
[user@master ~]$ sudo mv ambari.repo /etc/yum.repos.d/
```

3. The next step is to install the `ambari-server` package with the help of this command:

```
[user@master ~]$ sudo yum install ambari-server -y
```

4. We are going to use a MySQL server for our Ambari server. So, let's install the required packages as well:

```
[user@master ~]$ sudo yum install mariadb-server -y
```

5. Let's configure the MySQL server (or MariaDB) before we touch the Ambari setup process. This is done with the following commands:

```
[user@master ~]$ sudo service mariadb start
Redirecting to /bin/systemctl start mariadb.service
```

6. Then, create a database called `ambari` and a user called `ambari` with the password, `ambari`, so that the Ambari server configuration is easy to set up in the following steps. This can be done with these SQL queries:

```
CREATE DATABASE ambari;
GRANT ALL PRIVILEGES ON ambari.* to ambari@localhost identified by
'ambari';
GRANT ALL PRIVILEGES ON ambari.* to ambari@'%' identified by
'ambari';
FLUSH PRIVILEGES;
```

7. Store these four lines into a text file called `ambari.sql` and execute with the following command:

```
[user@master ~] mysql -uroot < ambari.sql
```

8. This will create a database, users and give necessary privileges.

Please use a strong password for production setup, otherwise your system will be vulnerable to any attacks.

Now that we have done the groundwork, let's run the Ambari server setup. Note that we are required to answer a few questions that are highlighted as follows:

```
[user@master ~]$ sudo ambari-server setup
Using python /usr/bin/python
Setup ambari-server
Checking SELinux...
SELinux status is 'enabled'
SELinux mode is 'enforcing'
Temporarily disabling SELinux
WARNING: SELinux is set to 'permissive' mode and temporarily
disabled.
OK to continue [y/n] (y)? <ENTER>
Customize user account for ambari-server daemon [y/n] (n)? <ENTER>
Adjusting ambari-server permissions and ownership...
Checking firewall status...
WARNING: iptables is running. Confirm the necessary Ambari ports
are accessible. Refer to the Ambari documentation for more details
on ports.
OK to continue [y/n] (y)? <ENTER>
Checking JDK...
[1] Oracle JDK 1.8 + Java Cryptography Extension (JCE) Policy Files
8
[2] Oracle JDK 1.7 + Java Cryptography Extension (JCE) Policy Files
7
[3] Custom JDK
====================================================================
===========
Enter choice (1): <ENTER>
To download the Oracle JDK and the Java Cryptography Extension
(JCE) Policy Files you must accept the license terms found at
http://www.oracle.com/technetwork/java/javase/terms/license/index.h
tml and not accepting will cancel the Ambari Server setup and you
must install the JDK and JCE files manually.
Do you accept the Oracle Binary Code License Agreement [y/n] (y)?
```

```
<ENTER>
Downloading JDK from
http://public-repo-1.hortonworks.com/ARTIFACTS/jdk-8u112-linux-x64.
tar.gz to /var/lib/ambari-server/resources/jdk-8u112-linux-
x64.tar.gz
jdk-8u112-linux-x64.tar.gz... 100% (174.7 MB of 174.7 MB)
Successfully downloaded JDK distribution to /var/lib/ambari-
server/resources/jdk-8u112-linux-x64.tar.gz
Installing JDK to /usr/jdk64/
Successfully installed JDK to /usr/jdk64/
Downloading JCE Policy archive from
http://public-repo-1.hortonworks.com/ARTIFACTS/jce_policy-8.zip to
/var/lib/ambari-server/resources/jce_policy-8.zip

Successfully downloaded JCE Policy archive to /var/lib/ambari-
server/resources/jce_policy-8.zip
Installing JCE policy...
Checking GPL software agreement...
GPL License for LZO:
https://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html
Enable Ambari Server to download and install GPL Licensed LZO
packages [y/n] (n)? y <ENTER>
Completing setup...
Configuring database...
Enter advanced database configuration [y/n] (n)? y <ENTER>
Configuring database...
===================================================================
===========
Choose one of the following options:
[1] - PostgreSQL (Embedded)
[2] - Oracle
[3] - MySQL / MariaDB
[4] - PostgreSQL
[5] - Microsoft SQL Server (Tech Preview)
[6] - SQL Anywhere
[7] - BDB
===================================================================
===========
Enter choice (1): 3 <ENTER>
Hostname (localhost):
Port (3306):
Database name (ambari):
Username (ambari):
Enter Database Password (bigdata): ambari <ENTER>
Re-enter password: ambari <ENTER>
Configuring ambari database...
Configuring remote database connection properties...
WARNING: Before starting Ambari Server, you must run the following
```

```
DDL against the database to create the schema: /var/lib/ambari-
server/resources/Ambari-DDL-MySQL-CREATE.sql
Proceed with configuring remote database connection properties
[y/n] (y)? <ENTER>
Extracting system views...
ambari-admin-2.6.1.5.3.jar
...........
Adjusting ambari-server permissions and ownership...
Ambari Server 'setup' completed successfully.
```

9. Once the setup is complete, we need to create the tables in the Ambari database by using the previous file that is generated during the setup process. This can be done with this command:

```
[user@master ~] mysql -u ambari -pambari ambari < /var/lib/ambari-
server/resources/Ambari-DDL-MySQL-CREATE.sql
```

10. The next step is for us to start the `ambari-server` daemon. This will start the web interface that we will use in the following steps to create the Hadoop cluster:

```
[user@master ~]$ sudo ambari-server start
Using python /usr/bin/python
Starting ambari-server
Ambari Server running with administrator privileges.
Organizing resource files at /var/lib/ambari-server/resources...
Ambari database consistency check started...
Server PID at: /var/run/ambari-server/ambari-server.pid
Server out at: /var/log/ambari-server/ambari-server.out
Server log at: /var/log/ambari-server/ambari-server.log
Waiting for server start..............................
Server started listening on 8080
DB configs consistency check: no errors and warnings were found.
Ambari Server 'start' completed successfully.
```

11. Once the server setup is complete, configure the JDBC driver (which is helpful for all the other nodes as well):

```
[user@master ~] sudo ambari-server setup --jdbc-db=mysql --jdbc-
driver=/usr/share/java/mysql-connector-java.jar
```

Preparing the Hadoop cluster

There are a few more steps that we need to do before we go ahead and create the Hadoop cluster.

Since we have the Ambari server up and running, let's generate an RSA key pair that we can use for communication between the Ambari server and the Ambari agent nodes.

This key pair lets the Ambari server node log in to all the Hadoop nodes and perform the installation in an automated way.

This step is optional if you have already done this as part of procuring the servers and infrastructure:

```
[user@master ~]$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/user/.ssh/id_rsa):
Enter passphrase (empty for no passphrase): <ENTER>
Enter same passphrase again: <ENTER>
Your identification has been saved in /home/user/.ssh/id_rsa.
Your public key has been saved in /home/user/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:JWBbGdAnRHM0JFj35iSAcQk+rC0MhyHlrFawr+d2cZ0 user@master
The key's randomart image is:
+---[RSA 2048]----+
|.oo   *@@**    |
| +oo +o==*.o     |
| .=.. = .oo +    |
| .o+ o . o =     |
|.. .+ . S . .    |
|. .  o . E      |
| . .  o    |
|  o. .           |
|  ...            |
+----[SHA256]-----+
```

This will generate two files inside the `/home/user/.ssh` directory:

- `~/.ssh/id_rsa`: This is the private key file which has to be kept in a secret place
- `~/.ssh/id_rsa.pub`: This is the public key file which allows any SSH login using the private key file

The contents of this `id_rsa.pub` file should be put in `~/.ssh/authorized_keys` on all the Hadoop nodes. In this case, they are node servers (1–3).

TIP

This step of propagating all the public SSH keys can be done during the server provisioning itself, so a manual step is avoided every time we acquire new servers.

Now, we will do all the work with only the Ambari web interface.

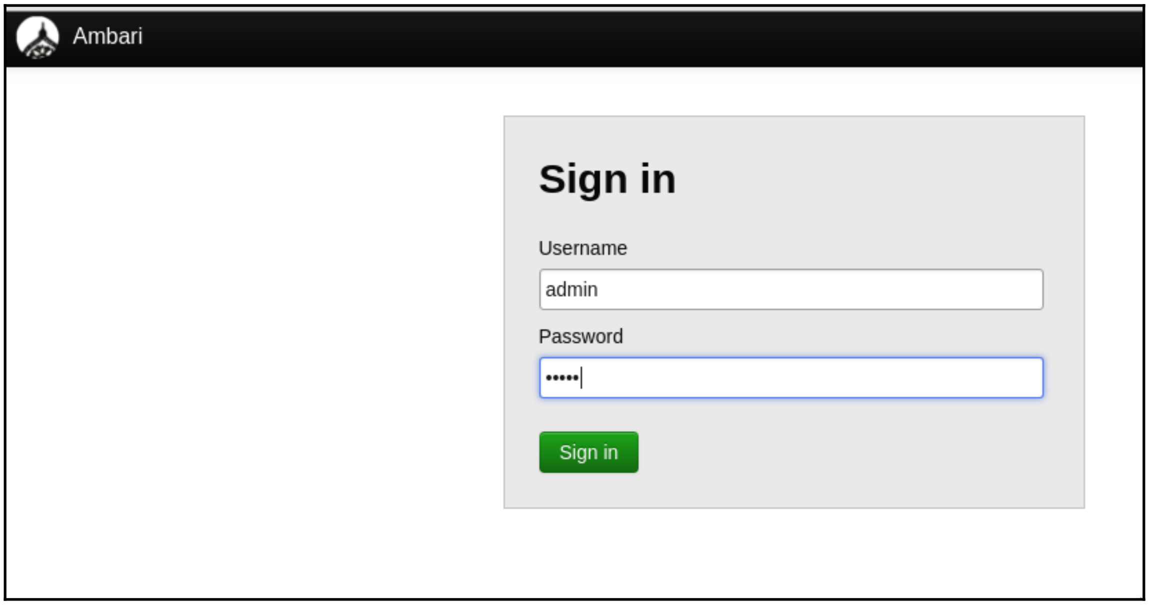
Creating the Hadoop cluster

In this section, we will build a Hadoop cluster using the Ambari web interface. This section assumes the following things:

- The nodes (1–3) are reachable over SSH from the master server
- Admin can log in to the nodes (1–3) using the `id-rsa` private key from the master server
- A UNIX user can run `sudo` and perform all administrative actions on the node (1–3) servers
- The Ambari server setup is complete
- The Ambari web interface is accessible to the browser without any firewall restrictions

Ambari web interface

Let's open a web browser and connect to the Ambari server web interface using `http://<server-ip>:8080`. We are presented with a login screen like this. Please enter `admin` as the username and `admin` as the password to continue:

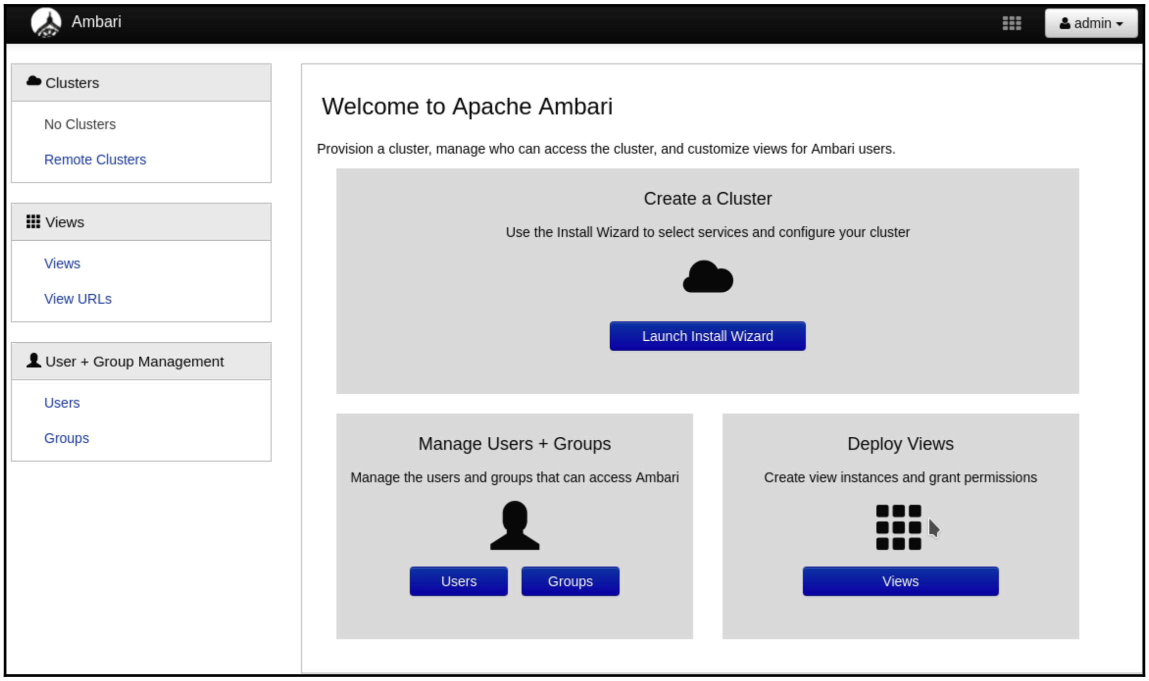


Once the login is successful, we are taken to the home page.

The Ambari home page

This is the main page where there are multiple options on the UI. Since this is a brand new installation, there is no cluster data available yet.

Let's take a look at the home page with this screenshot:



Welcome to Apache Ambari

Provision a cluster, manage who can access the cluster, and customize views for Ambari users.

Create a Cluster

Use the Install Wizard to select services and configure your cluster

Manage Users + Groups

Manage the users and groups that can access Ambari

Deploy Views

Create view instances and grant permissions

From this place, we can carry out the following activities:

Creating a cluster

As you may have guessed, this section is used to launch a wizard that will help us create a Hadoop cluster from the browser.

Managing users and groups

This section is helpful to manage users and groups that can use and manage the Ambari web application.

Deploying views

This interface is helpful in creating views for different types of users and what actions they can perform via the Ambari web interface.

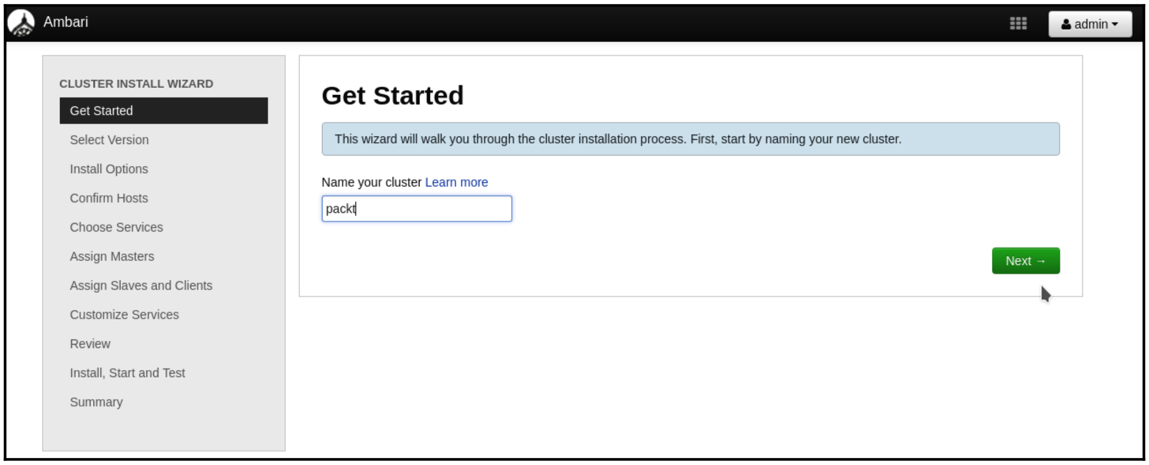
Since our objective is to create a new Hadoop cluster, we will click on the **Launch Install Wizard** button and start the process of creating a Hadoop cluster.

The cluster install wizard

Hadoop cluster creation is broken down into multiple steps. We will go through all these steps in the following sections. First, we are presented with a screen where we need to name our Hadoop cluster.

Naming your cluster

I have chosen `packt` as the Hadoop cluster name. Click **Next** when the Hadoop name is entered in the screen. The screen looks like this:



Get Started

This wizard will walk you through the cluster installation process. First, start by naming your new cluster.

Name your cluster Learn more

packt

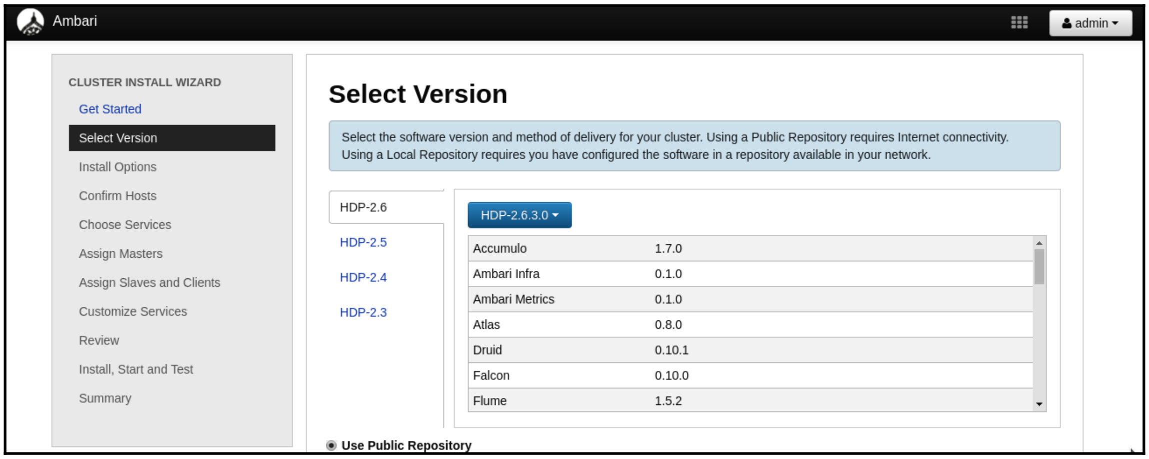
Selecting the Hadoop version

Once we name the Hadoop cluster, we are presented with a screen to select the version of Hadoop we want to run.

At the time of writing, Ambari supports the following Hadoop versions:

- Hadoop 2.3
- Hadoop 2.4
- Hadoop 2.5
- Hadoop 2.6 (upto 2.6.3.0)

You can choose any version for the installation. I have selected the default option which is version 2.6.3.0, which can be seen in this screenshot:



Select Version

Select the software version and method of delivery for your cluster. Using a Public Repository requires Internet connectivity. Using a Local Repository requires you have configured the software in a repository available in your network.

HDP-2.6
HDP-2.5
HDP-2.4
HDP-2.3

HDP-2.6.3.0

Accumulo	1.7.0
Ambari Infra	0.1.0
Ambari Metrics	0.1.0
Atlas	0.8.0
Druid	0.10.1
Falcon	0.10.0
Flume	1.5.2

● Use Public Repository

Click **Next** at the bottom of the screen to continue to the next step.

Selecting a server

The next logical step is to select the list of servers on which we are going to install the Hadoop-2.6.3.0 version. If you remember the original table, we named our node servers (1–3). We will enter those in the UI.

Since the installation is going to be completely automated, we also need to provide the RSA private key that we generated in the previous section in the UI. This will make sure that the master node can log in to the servers without any password over SSH.

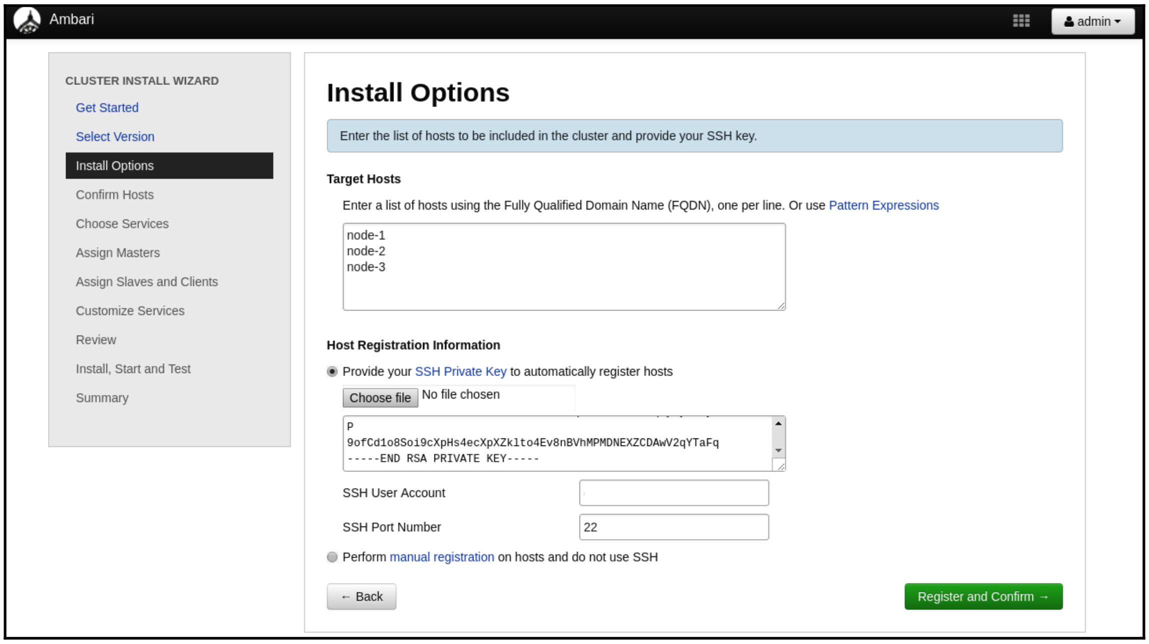
Also, we need to provide a UNIX username that's already been created on all the node (1–3) servers that can also accept RSA key for authentication.

TIP

Add `id_rsa.pub` to `~/.ssh/authorized_keys` on the node (1–3) servers.

Please keep in mind that these hostnames should have proper entries in the **DNS** (**Domain Name System**) Servers otherwise the installation won't be able to proceed from this step.

The names that I have given can be seen in this following screenshot:



Install Options

Enter the list of hosts to be included in the cluster and provide your SSH key.

Target Hosts

Enter a list of hosts using the Fully Qualified Domain Name (FQDN), one per line. Or use Pattern Expressions

```
node-1
node-2
node-3
```

Host Registration Information

● Provide your SSH Private Key to automatically register hosts

Choose file No file chosen

```
P
9ofCd1o8Soi9cXpHs4ecXpXZklto4Ev8nBVhMPMDNEXZCDAwV2qYTaFq
-----END RSA PRIVATE KEY-----
```

SSH User Account

SSH Port Number 22

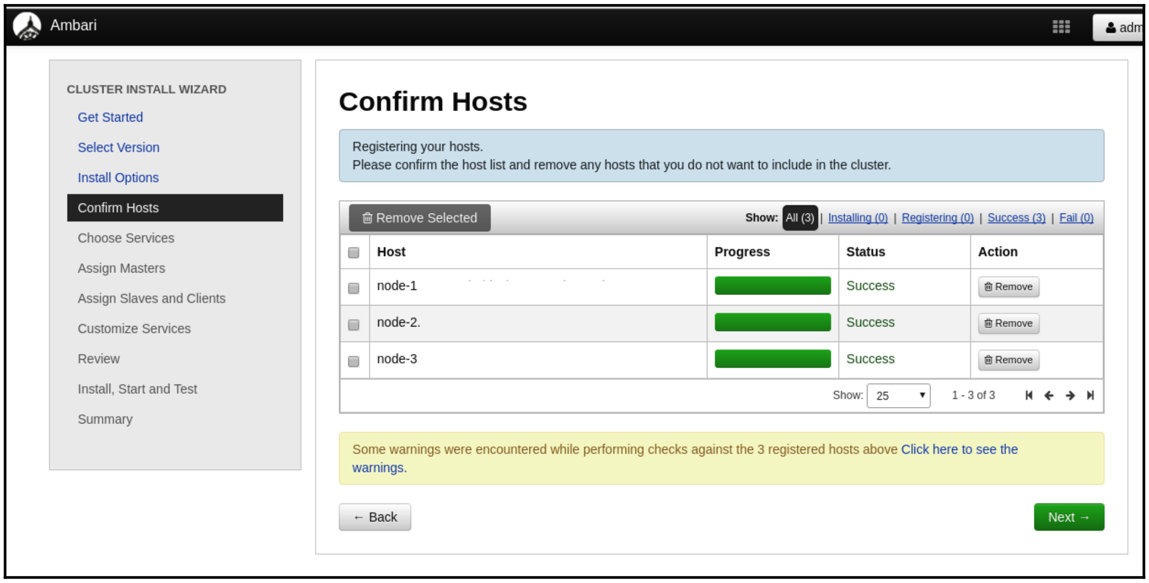
○ Perform manual registration on hosts and do not use SSH

← Back   Register and Confirm →

After the data is entered, click on **Register and Confirm**.

Setting up the node

In this step, the Ambari agent is automatically installed on the given nodes, provided the details are accurate. Success confirmation looks like this:



Confirm Hosts

Registering your hosts.
Please confirm the host list and remove any hosts that you do not want to include in the cluster.

Host	Progress	Status	Action
node-1		Success	Remove
node-2.		Success	Remove
node-3		Success	Remove

Some warnings were encountered while performing checks against the 3 registered hosts above Click here to see the warnings.

If we want to remove any nodes, this is the screen in which we can do it. Click **Next** when we are ready to go to the next step.

Selecting services

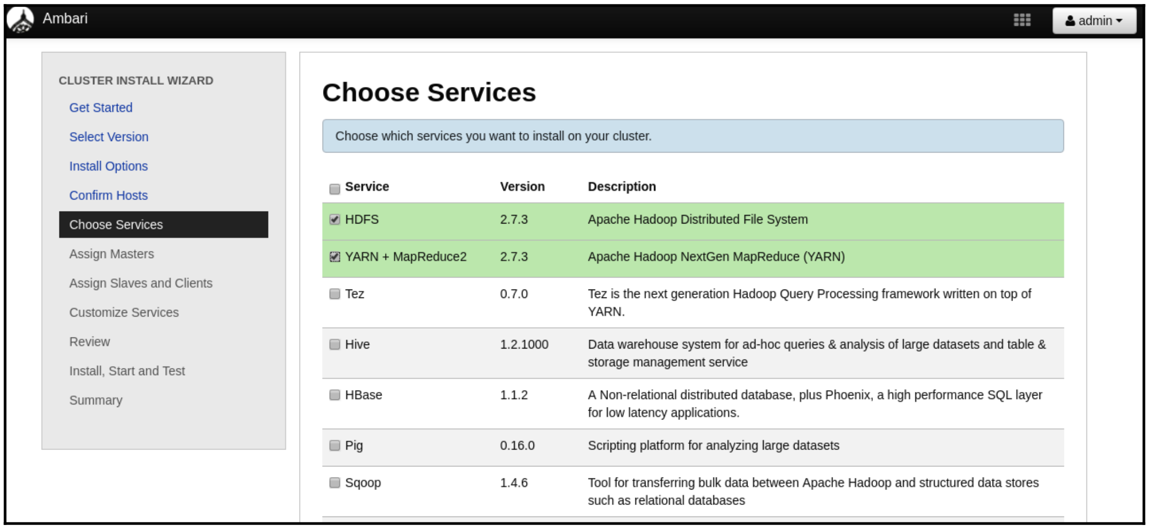
Now, we need to select the list of applications/services that we want to install on the three servers we have selected.

At the time of writing, Ambari supports the following services:

Application/Service	**Application Description**
HDFS	Hadoop Distributed File System
YARN + MapReduce2	Next generation Map Reduce framework
Tez	Hadoop query processing framework built on top of YARN
Hive	Data warehouse system for ad hoc queries

HBase	Non-relational distributed database
Pig	Scripting platform to analyze datasets in HDFS
Sqoop	Tool to transfer data between Hadoop and RDBMS
Oozie	Workflow co-ordination for Hadoop jobs with a web UI
ZooKeeper	Distributed system coordination providing service
Falcon	Data processing and management platform
Storm	Stream processing framework
Flume	Distributed system to collect, aggregate, and move streaming data to HDFS
Accumulo	Distributed key/value store
Ambari Infra	Shared service used by Amari components
Ambari Metrics	Grafana-based system for metric collection and storage
Atlas	Metadata and governance platform
Kafka	Distributed streaming platform
Knox	Single-point authentication provider for all Hadoop components
Log Search	Ambari-managed services log aggregator and viewer
Ranger	Hadoop data security application
Ranger KMS	Key management server
SmartSense	Hortonworks Smart Sense tool to diagnose applications
Spark	Large-scale data processing framework
Zeppelin Notebook	Web-based notebook for data analytics
Druid	Column-oriented data store
Mahout	Machine learning algorithms
Slider	Framework to monitor applications on YARN
Superset	Browser-based data exploration platform for RDBMS and Druid

As part of the current step, we have selected only HDFS and its dependencies. The screen is shown as follows:



Choose Services

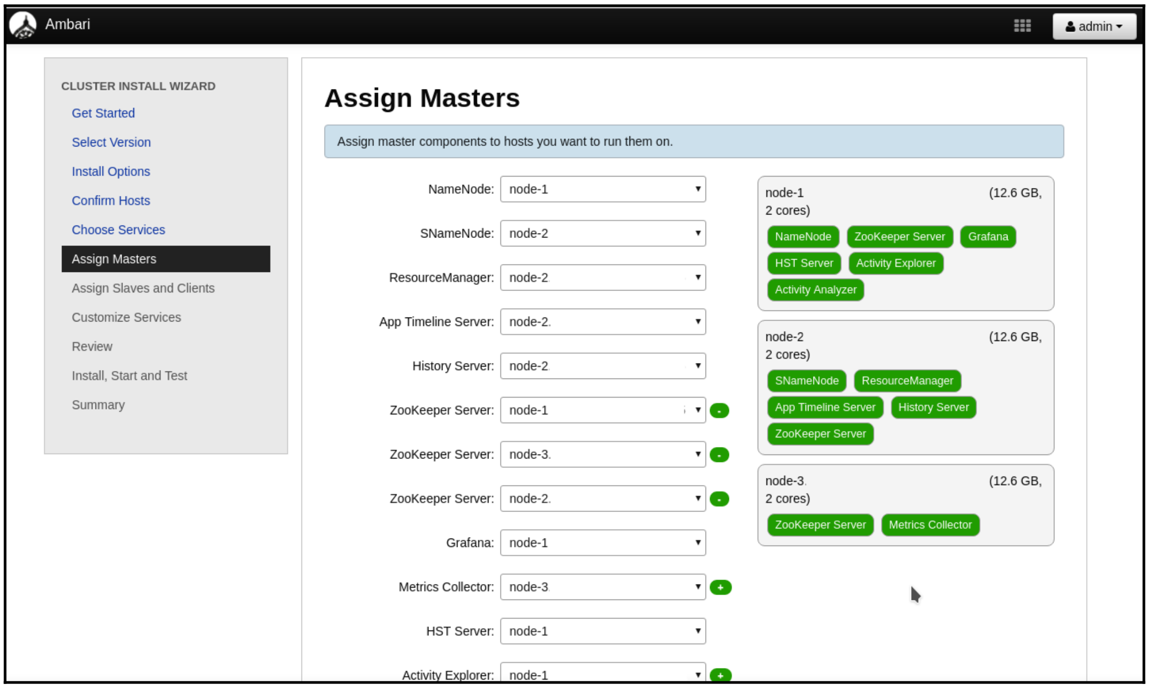
Choose which services you want to install on your cluster.

Service	Version	Description
✓ HDFS	2.7.3	Apache Hadoop Distributed File System
✓ YARN + MapReduce2	2.7.3	Apache Hadoop NextGen MapReduce (YARN)
Tez	0.7.0	Tez is the next generation Hadoop Query Processing framework written on top of YARN.
Hive	1.2.1000	Data warehouse system for ad-hoc queries & analysis of large datasets and table & storage management service
HBase	1.1.2	A Non-relational distributed database, plus Phoenix, a high performance SQL layer for low latency applications.
Pig	0.16.0	Scripting platform for analyzing large datasets
Sqoop	1.4.6	Tool for transferring bulk data between Apache Hadoop and structured data stores such as relational databases

Once you have made your choices, click the **Next** button at the bottom of the UI.

Service placement on nodes

In this step, we are shown the automatic selection of services on the three nodes we have selected for installation. If we want to customize the placement of the services on the nodes, we can do so. The placement looks like this:



Ambari

admin

CLUSTER INSTALL WIZARD

- Get Started
- Select Version
- Install Options
- Confirm Hosts
- Choose Services
- Assign Masters
- Assign Slaves and Clients
- Customize Services
- Review
- Install, Start and Test
- Summary

Assign Masters

Assign master components to hosts you want to run them on.

NameNode:	node-1
SNameNode:	node-2
ResourceManager:	node-2
App Timeline Server:	node-2
History Server:	node-2
ZooKeeper Server:	node-1
ZooKeeper Server:	node-3
ZooKeeper Server:	node-2
Grafana:	node-1
Metrics Collector:	node-3
HST Server:	node-1
Activity Explorer:	node-1

node-1 (12.6 GB, 2 cores)
NameNode, ZooKeeper Server, Grafana, HST Server, Activity Explorer, Activity Analyzer

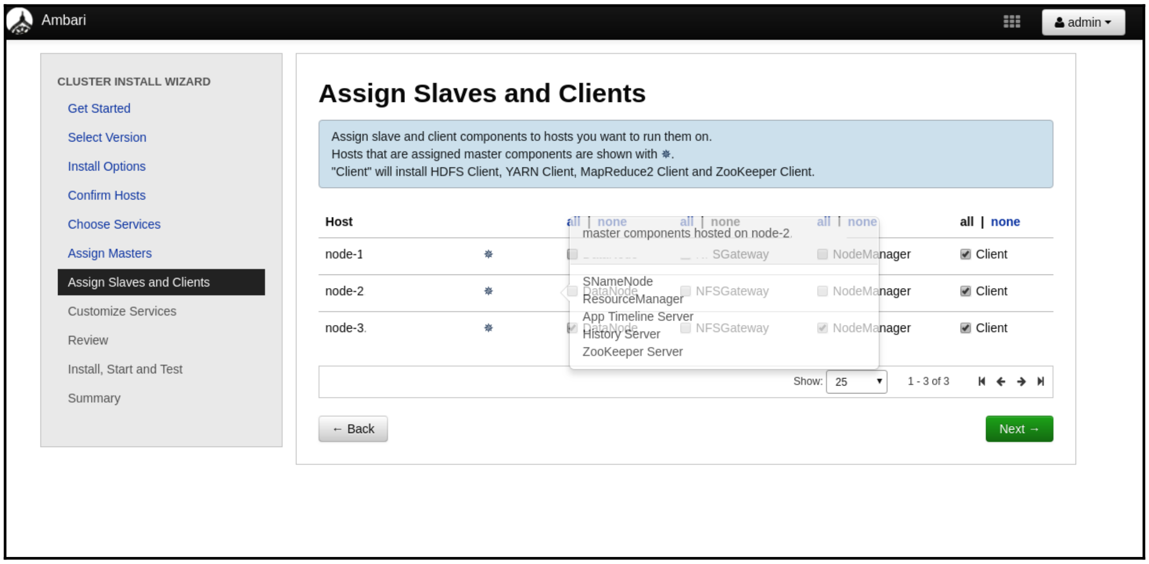
node-2 (12.6 GB, 2 cores)
SNameNode, ResourceManager, App Timeline Server, History Server, ZooKeeper Server

node-3 (12.6 GB, 2 cores)
ZooKeeper Server, Metrics Collector

Click **Next** when the changes look good.

Selecting slave and client nodes

Some applications support slaves and client utilities. In this screen, we need to select the nodes on which we want these applications to be installed. If you are unsure, click **Next**. The screen looks like this:



Assign Slaves and Clients

Assign slave and client components to hosts you want to run them on.
Hosts that are assigned master components are shown with ✵.
"Client" will install HDFS Client, YARN Client, MapReduce2 Client and ZooKeeper Client.

Host		all | none	all | none	all | none	all | none
node-1	✵	☐ DataNode	☐ NFSGateway	☐ NodeManager	☑ Client
node-2	✵	☐ DataNode	☐ NFSGateway	☐ NodeManager	☑ Client
node-3	✵	☑ DataNode	☐ NFSGateway	☑ NodeManager	☑ Client

master components hosted on node-2.
SNameNode
ResourceManager
App Timeline Server
History Server
ZooKeeper Server

Show: 25 1 - 3 of 3

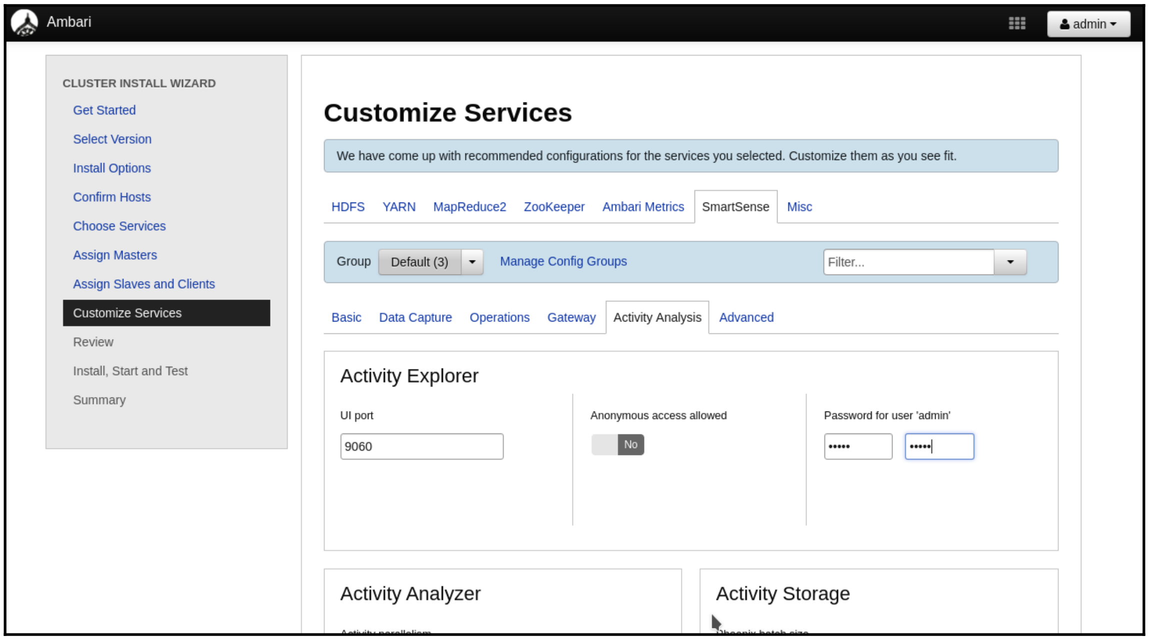
Customizing services

Even though Ambari automatically selects most of the properties and linkage between the applications, it provides us with some flexibility to choose values for some of the features, such as:

- Databases
- Usernames
- Passwords

And other properties that help the applications run smoothly. These are highlighted in the current screen in red.

In order to customize these, we need to go to the tab with the highlighted properties and choose the values according to our need. The screen looks like this:



Customize Services

We have come up with recommended configurations for the services you selected. Customize them as you see fit.

Activity Explorer

UI port: 9060

Anonymous access allowed: No

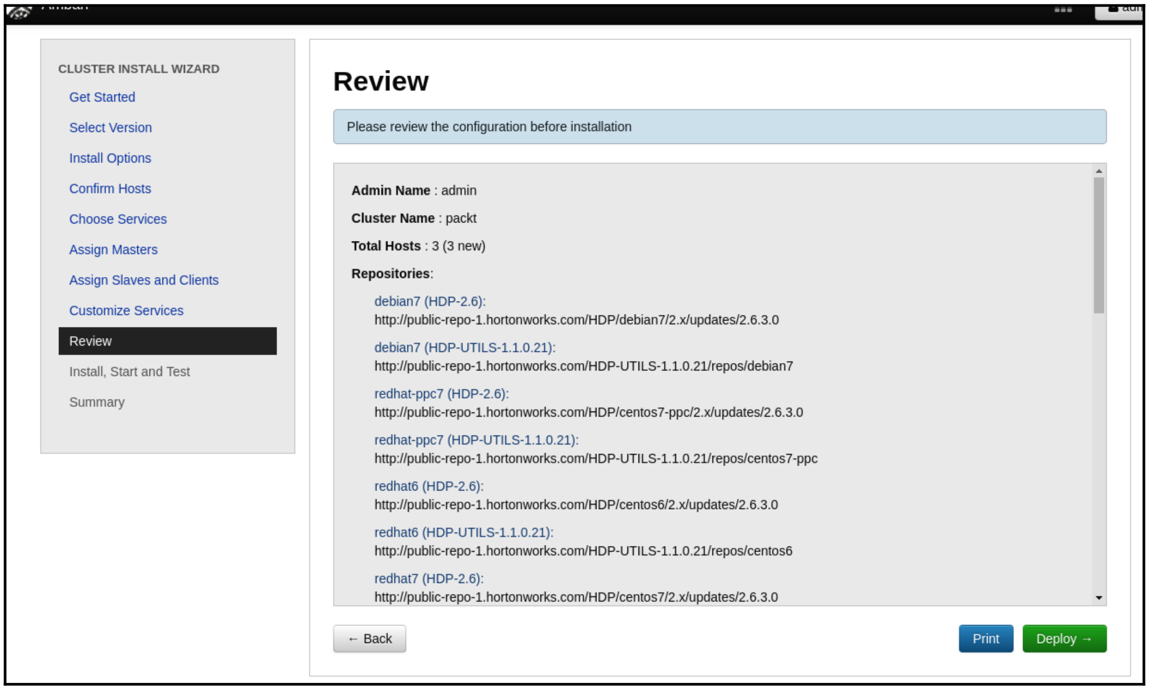
Password for user 'admin': ••••• •••••

After all the service properties are configured correctly, we will not see anything in red in the UI and can click the **Next** button at the bottom of the page.

Reviewing the services

In this step, we are shown a summary of the changes we have made so far. We are given an option to print the changes so that we will not forget them (don't worry, all these are available on the UI later). For now we can click **Deploy**. This is when the actual changes will be made to the nodes.

No changes will be made to the servers if we cancel this process. The current state of the wizard looks like this now:



Review

Please review the configuration before installation

Admin Name : admin
Cluster Name : packt
Total Hosts : 3 (3 new)
Repositories:

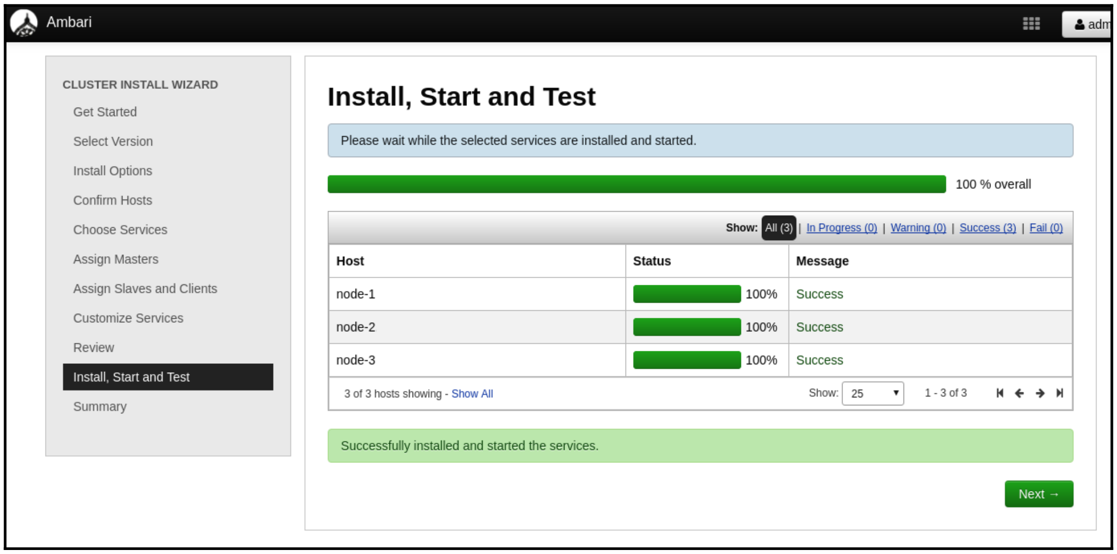
debian7 (HDP-2.6):
http://public-repo-1.hortonworks.com/HDP/debian7/2.x/updates/2.6.3.0
debian7 (HDP-UTILS-1.1.0.21):
http://public-repo-1.hortonworks.com/HDP-UTILS-1.1.0.21/repos/debian7
redhat-ppc7 (HDP-2.6):
http://public-repo-1.hortonworks.com/HDP/centos7-ppc/2.x/updates/2.6.3.0
redhat-ppc7 (HDP-UTILS-1.1.0.21):
http://public-repo-1.hortonworks.com/HDP-UTILS-1.1.0.21/repos/centos7-ppc
redhat6 (HDP-2.6):
http://public-repo-1.hortonworks.com/HDP/centos6/2.x/updates/2.6.3.0
redhat6 (HDP-UTILS-1.1.0.21):
http://public-repo-1.hortonworks.com/HDP-UTILS-1.1.0.21/repos/centos6
redhat7 (HDP-2.6):
http://public-repo-1.hortonworks.com/HDP/centos7/2.x/updates/2.6.3.0

Installing the services on the nodes

After we've clicked **Deploy** in the previous step, a deployment plan is generated by the Ambari server and applications will be deployed on all the nodes in parallel, using the Ambari agents running on all the nodes.

We are shown the progress of what is being deployed in real time in this step.

Once all the components have been installed, they will be automatically started and we can see the successful completion in this screen:



Install, Start and Test

Please wait while the selected services are installed and started.

100 % overall

Host	Status	Message
node-1	100%	Success
node-2	100%	Success
node-3	100%	Success

3 of 3 hosts showing - Show All   Show: 25   1 - 3 of 3

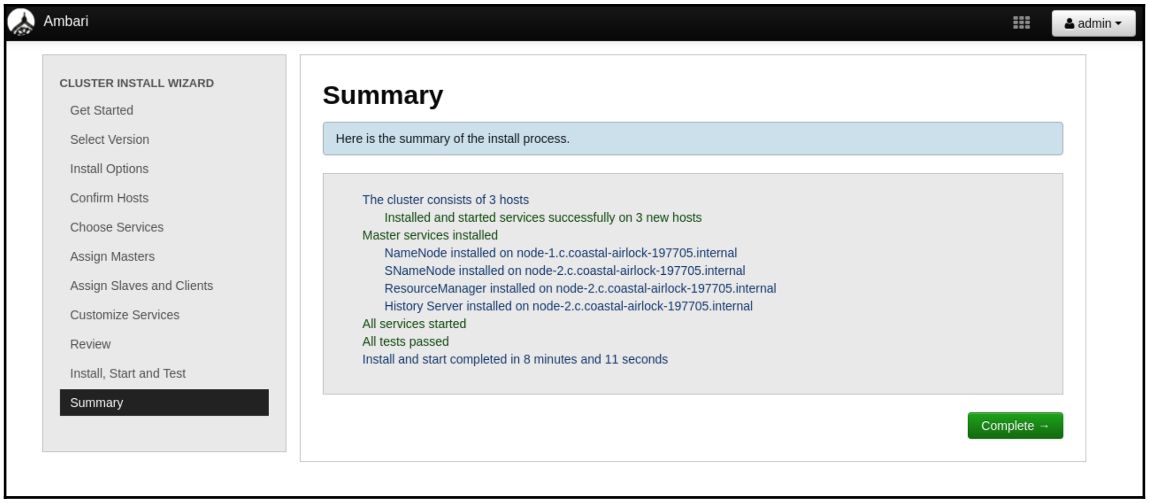
Successfully installed and started the services.

Click **Next** when everything is done successfully. In the case of any failures, we are shown what has failed and will be given an option to retry the installation. If there are any failures, we need to dig into the errors and fix the underlying problems.

If you have followed the instructions given at the beginning of the section you should have everything running smoothly.

Installation summary

In this step, we are shown the summary of what has been installed. The screen looks like this:



Summary

Here is the summary of the install process.

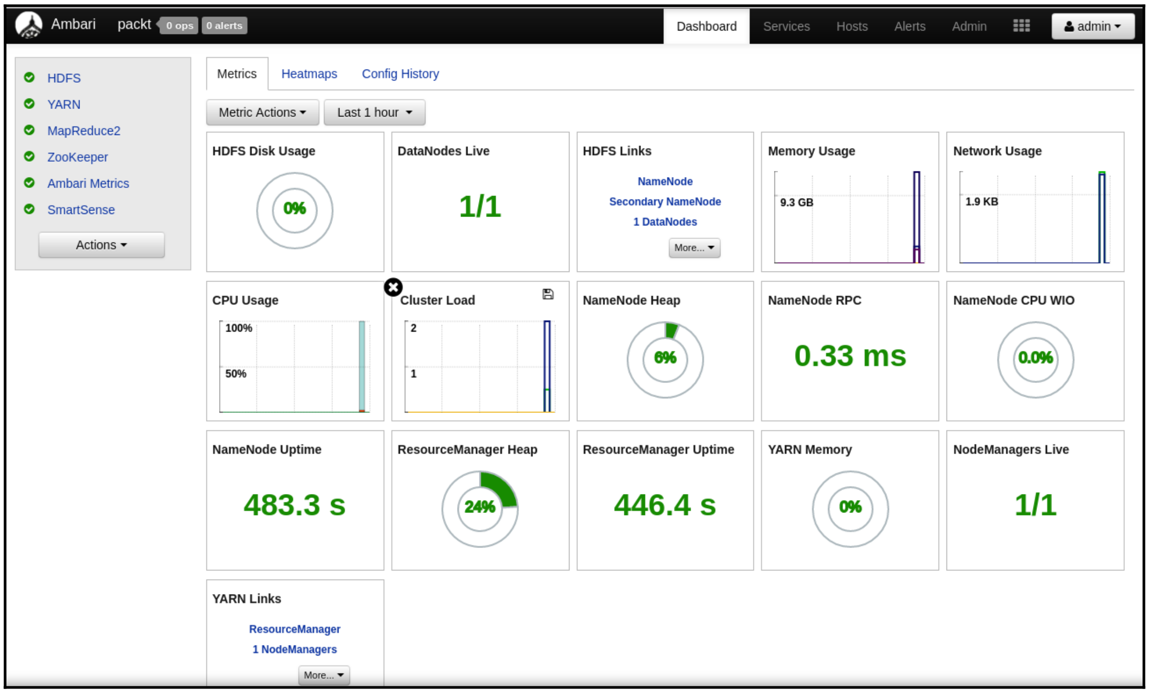
The cluster consists of 3 hosts
    Installed and started services successfully on 3 new hosts
Master services installed
    NameNode installed on node-1.c.coastal-airlock-197705.internal
    SNameNode installed on node-2.c.coastal-airlock-197705.internal
    ResourceManager installed on node-2.c.coastal-airlock-197705.internal
    History Server installed on node-2.c.coastal-airlock-197705.internal
All services started
All tests passed
Install and start completed in 8 minutes and 11 seconds

Click on the **Complete** button which marks the end of the Hadoop cluster setup. Next, we will be taken to the cluster dashboard.

The cluster dashboard

This is the home page of the Hadoop cluster we have just created where we can see the list of all the services that have been installed and the health sensors.

We can manage all aspects of the Hadoop cluster in this interface. Feel free to explore the interface and play with it to understand more:



This marks the end of the Hadoop cluster creation with Ambari.

Hadoop clusters

So far, we have seen how to create a single Hadoop cluster using Ambari. But, is there ever a requirement for multiple Hadoop clusters?

The answer depends on the business requirements. There are trade-offs for both single versus multiple Hadoop clusters.

Before we jump into the advantages and disadvantages of both of these, let's see in what scenarios we might use either.

A single cluster for the entire business

This is the most straightforward approach and every business starts with one cluster, at least. As the diversity of the business increases, organizations tend to choose one cluster per department, or business unit.

The following are some of the advantages:

- **Ease of operability**: Since there is only one Hadoop cluster, managing it is very easy and the team sizes will also be optimal when administering it.
- **One-stop shop**: Since all the company data is in a single place, it's very easy to come up with innovative ways to use and generate analytics on top of the data.
- **Integration cost**: Teams and departments within the enterprise can integrate with this single system very easily. They have less complex configurations to deal with when managing their applications.
- **Cost to serve**: Enterprises can have a better understanding of their entire big data usage and can also plan, in a less stringent way, on scaling their system.

Some disadvantages of employing this approach are as follows:

- **Scale becomes a challenge**: Even though Hadoop can be run on hundreds and thousands of servers, it becomes a challenge to manage such big clusters, particularly during upgrades and other changes.
- **Single point of failure**: Hadoop internally has replication built-in to it in the HDFS File System. When more nodes fail, the chances are that there is loss of data and it's hard to recover from that.
- **Governance is a challenge**: As the scale of data, applications, and users increase, it is a challenge to keep track of the data without proper planning and implementation in place.
    - **Security and confidential data management**: Enterprises deal with a variety of data that varies from highly sensitive to transient data. When all sorts of data is put in a big-data solution, we have to employ very strong authentication and authorization rules so that the data is visible only to the right audience.

With these thoughts, let's take a look at the other possibility of having Hadoop clusters in an enterprise.

Multiple Hadoop clusters

Even though having a single Hadoop cluster is easier to maintain within an organization, sometimes its important to have multiple Hadoop clusters to keep the business running smoothly and reduce dependency on a single point of failure system.

These multiple Hadoop clusters can be used for several reasons:

- Redundancy
- Cold backup
- High availability
- Business continuity
- Application environments

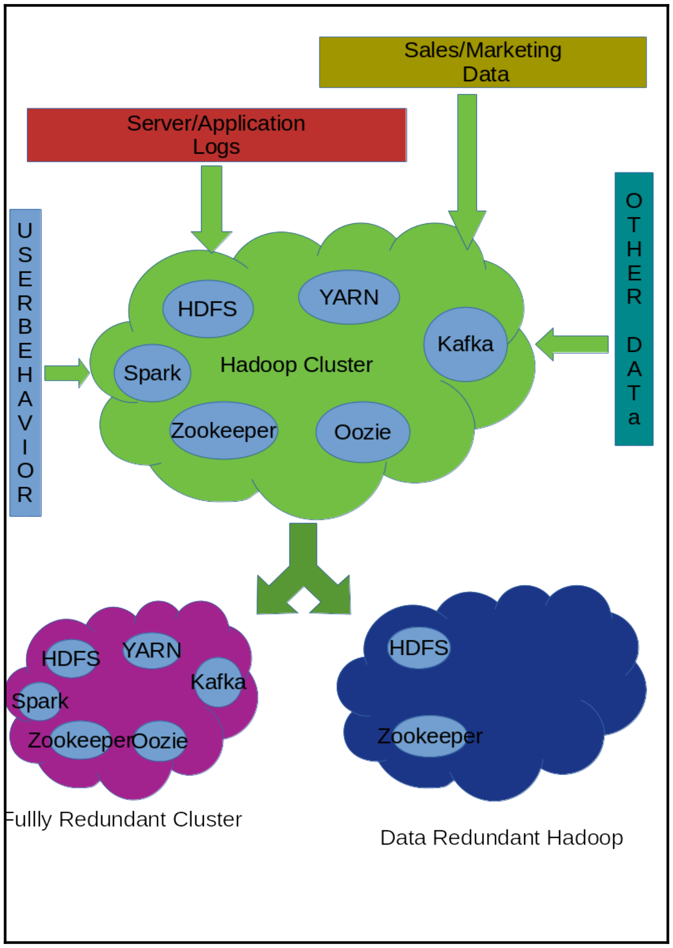
Redundancy

When we think of redundant Hadoop clusters, we should think about how much redundancy we can keep. As we already know, the **Hadoop Distributed File System** (**HDFS**) has internal data redundancy built in to it.

Given that a Hadoop cluster has lot of ecosystem built around it (services such as YARN, Kafka, and so on), we should think and plan carefully about whether to have the entire ecosystem made redundant or make only the data redundant by keeping it in a different cluster.

It's easier to make the HDFS portion of the Hadoop redundant as there are tools to copy the data from one HDFS to another HDFS.

Let's take a look at possible ways to achieve this via this diagram:



As we can see here, the main Hadoop cluster runs a full stack of all its applications, and data is supplied to it via multiple sources.

We have defined two types of redundant clusters:

A fully redundant Hadoop cluster

This cluster runs the exact set of applications as the primary cluster and the data is copied periodically from the main Hadoop cluster. Since this is a one-way copy from the main cluster to the second cluster, we can be 100% sure that the main cluster isn't impacted when we make any changes to this fully redundant cluster.

One important thing to understand is that we are running all other instances of applications in this cluster. Since every application maintains its state in its own predefined location, the application states are not replicated from the main Hadoop cluster to this cluster, which means that the jobs that were created in the main Hadoop cluster are not visible in this cluster. The same applies to the Kafka topics, zookeeper nodes, and many more.

This type of cluster is helpful for running different environments such as QA, Staging, and so on.

A data redundant Hadoop cluster

In this type of cluster setup, we create a new Hadoop cluster and copy the data from the main cluster, like in the previous case; but here we are not worried about the other applications that are run in this cluster.

This type of setup is good for:

- Having data backup for Hadoop in a different geography
- Sharing big data with other enterprises/organizations

Cold backup

Cold backup is important for enterprises as the data gets older. Even though Hadoop is designed to store unlimited amounts of data, it's not always necessary to keep all the data available for processing.

It is sometimes necessary to preserve the data for auditing purposes and also for historical reasons. In such cases, we can create a dedicated Hadoop cluster with only the HDFS (File System) component and periodically sync all the data into this cluster.

The design for this system is similar to the data redundant Hadoop cluster.

High availability

Even though Hadoop has multiple components within the architecture, not all the components are highly available due to the internal design.

The core component of Hadoop is its distributed, fault-tolerant, filesystem HDFS. HDS has multiple components one of them is the NameNode which is the registry of where the files are located in the HDFS. In the earlier versions of HDS NameNode was Single point of Failure, In the recent versions Secondary NameNode has been added to assist with high availability requirements for Hadoop Cluster.

In order to make every component of the Hadoop ecosystem a highly available system, we need to add multiple redundant nodes (they come with their own cost) which work together as a cluster.

One more thing to note is that high availability with Hadoop is possible within a single geographical region, as the locality of the data with applications is one of the key things with Hadoop. The moment we have multiple data centers in play we need to think alternatively to achieve high availability across the data centers.

Business continuity

This is part of **Business Continuity Planning** (**BCP**) where natural disasters can bring an end to the Hadoop system, if not planned correctly.

Here, the strategy would be to use multiple geographical regions as providers to run the big data systems. When we talk about multiple data centers, the obvious challenge is the network and the cost associated with managing both systems. One of the biggest challenges is how to keep multiple regions in sync.

One possible solution is to build a fully redundant Hadoop cluster in other geographical regions and keep the data in sync, periodically. In the case of any disaster/breakdown of one region, our businesses won't come to halt as we can smoothly run our operations.

Application environments

Many businesses internally follow different ways of releasing their software to production. As part of this, they follow several continuous integration methodologies, in order to have better control over the stability of the Hadoop environments. It's good to build multiple smaller Hadoop clusters with X% of the data from the main production environment and run all the applications here.

Applications can build their integration tests on these dedicated environments (QA, Staging, and so on) and can release their software to production once everything is good.

One practice that I have come across is that organizations tend to directly ship the code to production and end up facing outage of their applications because of an untested workflow or bug. It's good practice to have dedicated Hadoop application environments to test the software thoroughly and achieve higher uptime and happier customers.

Hadoop data copy

We have seen in the previous sections that, having highly available data is very important for a business to succeed and stay up to date with its competition.

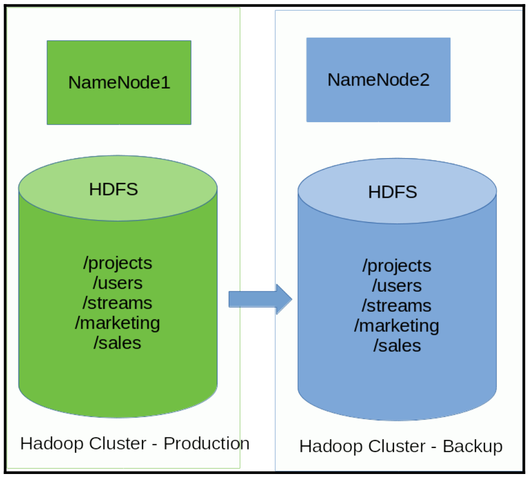
In this section, we will explore the possible ways to achieve highly available data setup.

HDFS data copy

Hadoop uses HDFS as its core to store the files. HDFS is rack aware and is intelligent enough to reduce the network data transfer when applications are run on the data nodes.

One of the preferred ways of data copying in an HDFS environment is to use the DistCp. The official documentation for this is available at the following URL `http://hadoop.apache.org/docs/r1.2.1/distcp.html`.

We will see a few examples of copying data from one Hadoop cluster to another Hadoop cluster. But before that, let's look at how the data is laid out:



In order to copy the data from the production Hadoop cluster to the backup Hadoop cluster, we can use `distcp`. Let's see how to do it:

```
hadoop distcp hdfs://NameNode1:8020/projects hdfs://NameNode2:8020/projects
hadoop distcp hdfs://NameNode1:8020/users hdfs://NameNode2:8020/users
hadoop distcp hdfs://NameNode1:8020/streams hdfs://NameNode2:8020/streams
hadoop distcp hdfs://NameNode1:8020/marketing
hdfs://NameNode2:8020/marketing
hadoop distcp hdfs://NameNode1:8020/sales hdfs://NameNode2:8020/sales
```

When we run the `distcp` command, a MapReduce job is created to automatically find out the list of files and then copy them to the destination.

The full command syntax looks like this:

```
Distcp [OPTIONS] <source path …> <destination path>
```

- `OPTIONS`: These are the multiple options the command takes which control the behavior of the execution.
- `source path`: A source path can be any valid File System URI that's supported by Hadoop. DistCp supports taking multiple source paths in one go.
- `destination path`: This is a single path where all the source paths need to be copied.

Let's take a closer look at a few of the important options:

Flag/Option	Description
`append`	Incrementally writes the data to the destination files if they already exist (only `append` is performed, no block level check is performed to do incremental copy).
`async`	Performs the copy in a non-blocking way.
`atomic`	Perform all the file copy or aborts even if one fails.
`Tmp <path>`	Path to be used for atomic commit.
`delete`	Deletes the files from the destination if they are not present in the source tree.
`Bandwidth <arg>`	Limits how much network bandwidth to be used during the copy process.
`f <file-path>`	Filename consisting of a list of all paths which need to be copied.
`i`	Ignores any errors during file copy.
`Log <file-path>`	Location where the execution log is saved.
`M <number>`	Maximum number of concurrent maps to use for copying.
`overwrite`	Overwrites the files even if they exist on destination.
`update`	Copies only the missing files and directories.
`skipcrccheck`	If passed, CRC checks are skipped during transfer.

Summary

In this chapter, we learned about Apache Ambari and studied its architecture in detail. We then understood how to prepare and create our own Hadoop cluster with Ambari. In order to do this, we also looked into configuring the Ambari server as per the requirement before preparing our cluster. We also learned about single and multiple Hadoop clusters and how they can be used, based on the business requirement.

Index