Unit-I: Bigdata & Distributed Storage with HDFS

Introduction to the World of Big Data:

Understanding Big Data – Evolution of Big Data – Failure of Traditional Database in Handling Big Data - 3Vs of Big Data 4 – Sources of Big Data – Different Types of Data – Big Data Infrastructure – Big Data Life Cycle – Big Data Technology – Big Data Applications – Big Data Use Cases

HDFS:

A Brief History of Apache Hadoop-The Design of HDFS - HDFS Concepts – Blocks – Namenodes and Datanodes – **Hadoop architecture** - Block Caching – HDFS Federation – HDFS High Availability – The Command-Line Interface – Basic Filesystem Operations – Hadoop Filesystems – Interfaces - The Java Interface – Reading Data from a Hadoop URL- Reading Data Using the FileSystem API – Writing Data – Directories - Querying the Filesystem - Deleting Data – Data Flow – Anatomy of a File Read - Anatomy of a File Write - Coherency Model - Parallel Copying with distcp - Keeping an HDFS Cluster Balanced - HDFS File System Shell

---

I.I. Understanding Big Data

What is Data?

The quantities, characters, or symbols on which operations are performed by a computer, which may be stored and transmitted in the form of electrical signals and recorded on magnetic, optical, or mechanical recording media.

What is Big Data?

Big Data is a collection of data that is huge in volume, yet growing exponentially with time. It is a data with so large size and complexity that none of traditional data management tools can store it or process it efficiently. Big data is also a data but with huge size.

Big data usually includes data sets with sizes beyond the ability of commonly used software tools to capture, create, manage, and process the data within a tolerable elapsed time.

Big data is high-volume, high-velocity and high-variety information assets that demand cost-effective, innovative forms of information processing for enhanced insight and decision-making.

How big is the big data?

(NOTE: The statistics shown about big data are taken from the reports of the year 2019.)

The amount of data we produce every day is truly mind-boggling. There are 2.5 quintillion bytes of data created each day at our current pace, but that pace is only accelerating with

the growth of the Internet of Things (IoT). Over the last two years alone 90 percent of the data in the world was generated.

For example: 1,000,000,000,000,000,000

The cardinal number of one quintillion is 1,000,000,000,000,000,000. We can say that a quintillion is a ***million million millions*** *or a* ***billion billions*** *or a* ***million trillions*** *or a* ***thousand quadrillions****.*

I.II. Evolution of Big Data

Figure 1 The big data evolution

Internet

With so much information at our fingertips, we're adding to the data stockpile every time we turn to our search engines for answers.

- We conduct more than half of our web searches from a mobile phone now.
- More than 4.9 billion humans use the internet (that's a growth rate between 7.5 - 33 percent over 2016).
- On average, Google now processes more than 40,000 searches EVERY second (3.5 billion searches per day)!

- While 77% of searches are conducted on Google, it would be remiss not to remember other search engines are also contributing to our daily data generation. Worldwide there are 5 billion searches a day.

Social Media

Our current love affair with social media certainly fuels data creation. According to Domo's Data Never Sleeps 5.0 report, these are numbers generated every minute of the day:

- Snapchat users share 527,760 photos
- More than 120 professionals join LinkedIn
- Users watch 4,146,600 YouTube videos
- 456,000 tweets are sent on Twitter
- Instagram users post 46,740 photos

With 2 billion active users Facebook is still the largest social media platform. More than a quarter of the world's 7 billion humans are active on Facebook! Here are some more intriguing Facebook statistics:

- 1.5 billion people are active on Facebook daily
- Europe has more than 307 million people on Facebook
- There are five new Facebook profiles created every second!
- More than 300 million photos get uploaded per day
- Every minute there are 510,000 comments posted and 293,000 statuses updated

Even though Facebook is the largest social network, Instagram (also owned by Facebook) has shown impressive growth. Here's how this photo-sharing platform is adding to our data deluge:

- There are 600 million Instagrammers; 400 million who are active every day
- Each day 95 million photos and videos are shared on Instagram

- 100 million people use the Instagram “stories” feature daily

Communication

We leave a data trail when we use our favourite communication methods today from sending texts to emails. Here are some incredible stats for the volume of communication we send out every minute:

- We send 16 million text messages
- 156 million emails are sent; worldwide it is estimated that there are 2.9 billion email users by 2019
- More than 15,000 GIFs are sent via Facebook messenger
- Every minute there are 103,447,520 spam emails sent
- There are 154,200 calls on Skype

Digital Photos

Now that our smartphones are exemplary cameras as well, everyone is a photog and the trillions of photos stored is the proof. Since there are no signs of this slowing down, expect these digital photo numbers to continue to grow:

- People will take 1.2 trillion photos by the end of 2017
- There will be 4.7 trillion photos stored

Services

There are some really interesting statistics coming out of businesses and service providers in our new platform-driven economy. Here are just a few numbers that are generated every minute that are interesting to know:

- The Weather Channel receives 18,055,556 forecast requests
- Venmo processes $51,892 peer-to-peer transactions
- Spotify adds 13 new songs
- Uber riders take 45,788 trips!

- There are 600 new page edits to Wikipedia

Internet of Things

The Internet of Things, connected “smart” devices that interact with each other and us while collecting all kinds of data, is exploding (from 2 billion devices in 2006 to a projected 200 billion by 2020) and is one of the primary drivers for our data vaults exploding as well.

Let’s take a look at just some of the stats and predictions for just one type of device, voice search:

- There are 33 million voice-first devices in circulation
- 8 million people use voice control each month
- Voice search queries in Google for 2016 were up 35 times over 2008

Observing all these numbers, one can start to imagine just how much data we collectively generate every single day.

I.III. Why traditional database systems fail to support “big data”?

“Big data” encompass a wide range of the tremendous data generated from various sources such as mobile devices, digital repositories, and enterprise applications. The data can be structured as well as unstructured. It ranges from terabytes—10^12 bytes—to even exabytes—10^18 bytes. Working with “big data” is complex because of the five v’s associated with “big data.” Facebook (FB) gets ~10 million new photos uploaded every hour and Google (GOOG) processes over 24 petabytes of data every day. Twitter (TWTR) tweets ~400 million tweets per day. All this shows the magnificent volume, variety, value, and velocity of “big data.”

RDBMS for data storage

The relational database management system (or RDBMS) had been the one solution for all database needs. Oracle, IBM (IBM), and Microsoft (MSFT) are the leading players of RDBMS. RDBMS uses structured query language (or SQL) to define, query, and update the database. However, the volume and velocity of business data has changed dramatically in the last couple of years.

Limitations of RDBMS to support “big data”

1. The data size has increased tremendously to the range of petabytes—one petabyte = 1,024 terabytes. RDBMS finds it challenging to handle such huge data volumes. To address this, RDBMS added more central processing units (or CPUs) or more memory to the database management system to scale up vertically.

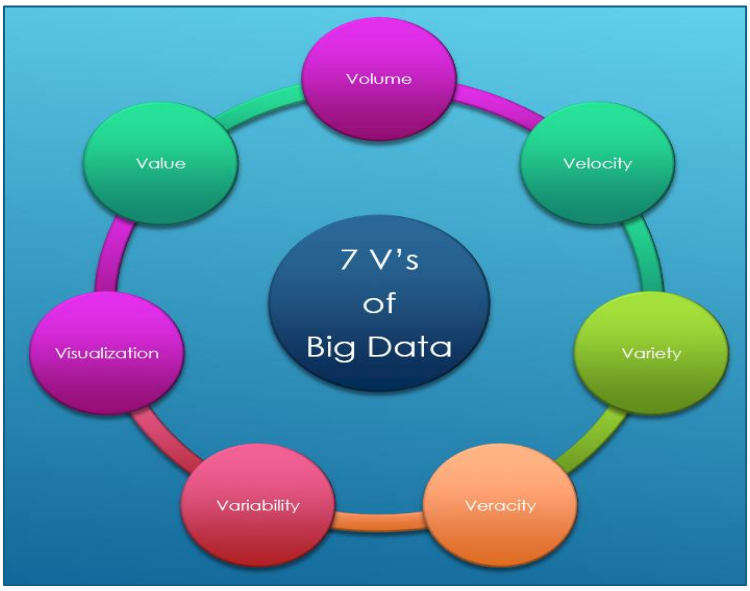
2. The majority of the data comes in a semi-structured or unstructured format from social media, audio, video, texts, and emails. RDBMS can’t categorize unstructured data. They’re designed and structured to accommodate structured data such as weblog sensor and financial data.

3. Also, “big data” is generated at a very high velocity. RDBMS lacks in high velocity because it’s designed for steady data retention rather than rapid growth. Even if RDBMS is used to handle and store “big data,” it will turn out to be very expensive. As a result, the inability of relational databases to handle “big data” led to the emergence of new technologies.

I. IV. The 7 V's of Big Data

The seven V's that sum up big data appropriately are listed below:

- Volume
- Velocity
- Variety
- Veracity and
- Value



1. **Volume**

   The name Big Data itself is related to a size which is enormous. Size of data plays a very crucial role in determining value out of data. Also, whether a particular data can actually be considered as a Big Data or not, is dependent upon the volume of data. Hence, 'Volume' is one characteristic which needs to be considered while dealing with Big Data solutions.

2. **Velocity**

   The term 'velocity' refers to the speed of generation of data. How fast the data is generated and processed to meet the demands, determines real potential in the data. Big Data Velocity deals with the speed at which data flows in from sources like business processes, application logs, networks, and social media sites, sensors, Mobile devices, etc. The flow of data is massive and continuous.

3. **Variety**

   The next aspect of Big Data is its variety. Variety refers to heterogeneous sources and the nature of data, both structured and unstructured. During earlier days, spreadsheets and databases were the only sources of data considered by most of the applications. Nowadays,

data in the form of emails, photos, videos, monitoring devices, PDFs, audio, etc. are also being considered in the analysis applications. This variety of unstructured data poses certain issues for storage, mining and analysing data.

4. **Veracity**

   Veracity is all about making sure the data is accurate, which requires processes to keep the bad data from accumulating in your systems. The simplest example is contacts that enter an online marketing system with false names and inaccurate contact information.

5. **Value**

   The organizations that experience the data growth or the organizations in which the data generation is marching towards big data will make sure that the volumes of data must generate appropriate value.

I.V. SOURCES OF BIG DATA

There are some of many sources of Big Data:

1. **Sensors/meters and activity records from electronic devices:** This kind of information is produced on real-time. The number and periodicity of the observations will be variable. For example, sometimes it will depend of a lap of time, on others of the occurrence of some event (per example a car passing by the vision angle of a camera) and in others will depend of manual manipulation (from a strict point of view it will be the same that the occurrence of an event). Quality of this kind of source depends mostly of the capacity of the sensor to take accurate measurements in the way it is expected.
2. **Social interactions:** It is the data produced by human interactions through a network, like Internet. It is the most common type of data produced in big data. This kind of data implies qualitative and quantitative aspects which are of some interest to be measured. Quantitative aspects are easier to measure than qualitative aspects. First one implies counting number of observations grouped by geographical or temporal characteristics, while the quality of the second one mostly relies on the accuracy of the algorithms applied to extract the meaning of the contents which are commonly found as unstructured text written in natural language, examples of analysis that are made from this data are sentiment analysis, trend topics analysis, etc.;
3. **Business transactions:** Data produced as a result of business activities can be recorded in structured or unstructured databases. When recorded on structured data bases the most common problem to analyze that information and get statistical indicators is the big volume of information and the periodicity of its production because sometimes these data is produced at

a very fast pace, thousands of records can be produced in a second when big companies like supermarket chains are recording their sales. But these kinds of data is not always produced in formats that can be directly stored in relational databases, an electronic invoice is an example of this case of source, it has more or less an structure but if we need to put the data that it contains in a relational database, we will need to apply some process to distribute that data on different tables (in order to normalize the data accordingly with the relational database theory), and maybe is not in plain text (could be a picture, a PDF, Excel record, etc.), one problem that we could have here is that the process needs time and as previously said, data maybe is being produced too fast, so we would need to have different strategies to use the data, processing it as it is without putting it on a relational database, discarding some observations (which criteria?), using parallel processing, etc. Quality of information produced from business transactions is tightly related to the capacity to get representative observations and to process them;

4. **Electronic Files:** These refers to unstructured documents, statically or dynamically produced which are stored or published as electronic files, like Internet pages, videos, audios, PDF files, etc. They can have contents of special interest but are difficult to extract, different techniques could be used, like text mining, pattern recognition, and so on. Quality of our measurements will mostly rely on the capacity to extract and correctly interpret all the representative information from those documents;
5. **Broadcastings:** Mainly referred to video and audio produced on real time, getting statistical data from the contents of this kind of electronic data by now is too complex and implies big computational and communications power, once solved the problems of converting “digital-analogue” contents to “digital-data” contents we will have similar complications to process it like the ones that we can find on social interactions.

I.VI. Different types of Big Data

Following are the types of Big Data:

- Structured
- Unstructured
- Semi-structured

Structured

Any data that can be stored, accessed and processed in the form of fixed format is termed as a ‘structured’ data. Over the period of time, talent in computer science has achieved greater success in developing techniques for working with such kind of data (where the format is well

known in advance) and also deriving value out of it. However, nowadays, we are foreseeing issues when a size of such data grows to a huge extent, typical sizes are being in the range of multiple zettabytes.

Unstructured

Any data with unknown form or the structure is classified as unstructured data. In addition to the size being huge, un-structured data poses multiple challenges in terms of its processing for deriving value out of it. A typical example of unstructured data is a heterogeneous data source containing a combination of simple text files, images, videos etc. Now day organizations have wealth of data available with them but unfortunately, they don't know how to derive value out of it since this data is in its raw form or unstructured format.

Semi-structured

Semi-structured data can contain both the forms of data. We can see semi-structured data as structured in form but it is actually not defined with e.g. a table definition in relational DBMS. Example of semi-structured data is a data represented in an XML file.

I.VII. Big data infrastructure

Big data infrastructure is what it sounds like: The IT infrastructure that hosts your "big data."

More specifically, big data infrastructure entails the tools and agents that collect data, the software systems and physical storage media that store it, the network that transfers it, the application environments that host the analytics tools that analyze it and the backup or archive infrastructure that backs it up after analysis is complete.

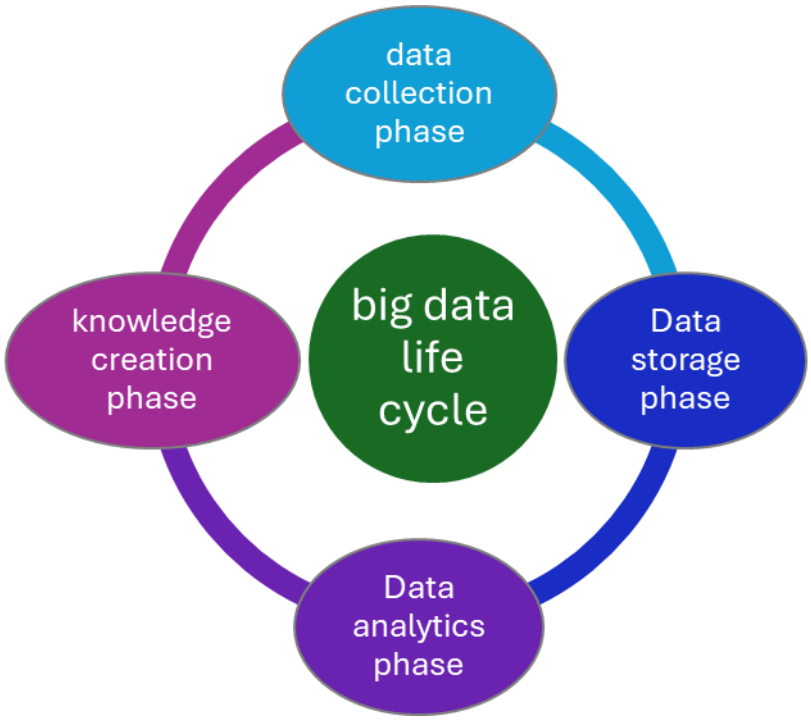
I.VII. Big data infrastructure

Big data infrastructure is what it sounds like: The IT infrastructure that hosts your “big data.”

More specifically, big data infrastructure entails the tools and agents that collect data, the software systems and physical storage media that store it, the network that transfers it, the application environments that host the analytics tools that analyze it and the backup or archive infrastructure that backs it up after analysis is complete.

I.VIII. Big Data Life Cycle

Big data life cycle consists of data collection phase, data storage phase, data processing and analysis, and knowledge creation. The following diagram represents the main elements in big data lifecycle.



i) Data Collection Phase

In data collection phase, data from different sources comes with different formats: structured, semi-structured, and unstructured. From a security perspective, securing big data technology should start from the first phase of the lifecycle. It is important to gather data from trusted sources and make sure that this phase is secured and protected. In fact, we need to take some security measures in order to keep data from being released. Some security measures can be used in this phase like limited access control (for those who

receive data from data provider) and encrypting some data fields (personal information identifier).

ii) Data Storage Phase

In data storage phase, the collected data is stored and prepared for being used in the next phase (data analytics phase). As the collected data may contain of sensitive information, it is essential to take sufficient precautions during data storing. In order to guarantee the safety of the collected data, some security measures can be used like data anonymization approach, permutation, and data partitioning (vertically or horizontally).

iii) Data Analytics Phase

After collecting data and storing it in secured storage solutions, data processing analysis is performed to generate useful knowledge. In this phase, data mining methods such as clustering, classification, and association rule mining are used. It is crucial to provide secure processing environment. In fact, data miners use powerful data mining algorithms that can extract sensitive data. Thus, a security breach may happen. Therefore, data mining process and its output must be protected against data mining-based attacks and make sure that only authorized staff work in this phase.

iv) Knowledge Creation Phase

Finally, the analytics phase comes up with new information and valued knowledge to be used by decision makers. The created knowledge is considered as sensitive information especially in a competition environment. Organizations take care of their sensitive information to be far away from their rivals. Further, they aware of their sensitive data (e.g., client personal data) not to be publicly released.

I.IX. Big data technology

Big data technology is defined as software-utility. This technology is primarily designed to analyze, process and extract information from a large data set and a huge set of extremely complex structures. This is very difficult for traditional data processing software to deal with.

Among the larger concepts of rage in technology, big data technologies are widely associated with many other technologies such as deep learning, machine learning, artificial intelligence (AI), and Internet of Things (IoT) that are massively augmented. In combination with these technologies, big data technologies are focused on analysing and handling large amounts of real-time data and batch-related data.

Types of Big Data Technology

Before we start with the list of big data technologies, let us first discuss this technology's board classification. Big Data technology is primarily classified into the following two types:

Operational Big Data Technologies

This type of big data technology mainly includes the basic day-to-day data that people used to process. Typically, the operational-big data includes daily basis data such as online transactions, social media platforms, and the data from any particular organization or a firm, which is usually needed for analysis using the software based on big data technologies. The data can also be referred to as raw data used as the input for several Analytical Big Data Technologies.

Some specific examples that include the Operational Big Data Technologies can be listed as below:

- Online ticket booking system, e.g., buses, trains, flights, and movies, etc.
- Online trading or shopping from e-commerce websites like Amazon, Flipkart, Walmart, etc.
- Online data on social media sites, such as Facebook, Instagram, Whatsapp, etc.
- The employees' data or executives' particulars in multinational companies.

Analytical Big Data Technologies

Analytical Big Data is commonly referred to as an improved version of Big Data Technologies. This type of big data technology is a bit complicated when compared with operational-big data. Analytical big data is mainly used when performance criteria are in use, and important real-time business decisions are made based on reports created by analysing operational-real data. This means that the actual investigation of big data that is important for business decisions falls under this type of big data technology.

Some common examples that involve the Analytical Big Data Technologies can be listed as below:

- Stock marketing data
- Weather forecasting data and the time series analysis
- Medical health records where doctors can personally monitor the health status of an individual
- Carrying out the space mission databases where every information of a mission is very important

Top Big Data Technologies

We can categorize the leading big data technologies into the following four sections:

- Data Storage
- Data Mining
- Data Analytics
- Data Visualization

Data Storage

Leading Big Data Technologies that come under Data Storage:

- Hadoop
- MongoDB
- Cassandra

Data Mining

Leading Big Data Technologies that come under Data Mining:

- Presto
- Rapidminer

Data Analytics

Leading Big Data Technologies that come under Data Analytics:

- Apache Kafka
- Spark

Data Visualization

Leading Big Data Technologies that come under Data Visualization:

- Tableau
- Plotly

Apart from the above-mentioned big data technologies, there are several other emerging big data technologies. The following are some essential technologies among them:

- Tensorflow
- Docker
- Kubernetes

These are emerging technologies. However, they are not limited because the ecosystem of big data is constantly emerging. That is why new technologies are coming at a very fast pace based on the demand and need for big data processing.

I.X. Big data Applications

Applications of big data

Big data applications can help companies to make better business decisions by analysing large volumes of data and discovering hidden patterns. These data sets might be from social media, data captured by sensors, website logs, customer feedbacks, etc. Organizations are spending huge amounts on big data applications to discover hidden patterns, unknown associations, market style, consumer preferences, and other valuable business information. The following are the prominent domains where big data can be applied:

health care

media and entertainment

IoT

manufacturing and

government.

Health care

There is a significant improvement in the healthcare domain by personalized medicine and prescriptive analytics due to the role of big data systems. Researchers analyze the data to determine the best treatment for a particular disease, side effects of the drugs, forecasting the health risks, etc. Mobile applications on health and wearable devices are causing available data to grow at an exponential rate. It is possible to predict a disease outbreak by mapping healthcare data and geographical data. Once predicted, containment of the outbreak can be handled and plans to eradicate the disease made.

Media and entertainment

The media and entertainment industries are creating, advertising, and distributing their content using new business models. This is due to customer requirements to view digital content from any location and at any time. The introduction of online TV shows, Netflix channels, etc. is proving that new customers are not only interested in watching TV but are interested in accessing data from any location. The media houses are targeting audiences by predicting what they would like to see, how to target the ads, content monetization, etc. Big data systems are thus increasing the revenues of such media houses by analysing viewer patterns.

Internet of Things

IoT devices generate continuous data and send them to a server on a daily basis. These data are mined to provide the interconnectivity of devices. This mapping can be put to

good use by government agencies and also a range of companies to increase their competence. IoT is finding applications in smart irrigation systems, traffic management, crowd management, etc.

Manufacturing

Predictive manufacturing can help to increase efficiency by producing more goods by minimizing the downtime of machines. This involves a massive quantity of data for such industries. Sophisticated forecasting tools follow an organized process to explore valuable information for these data. The following are the some of the major advantages of employing big data applications in manufacturing industries:

high product quality

tracking faults

supply planning

predicting the output

increasing energy efficiency

testing and simulation of new manufacturing processes and

large-scale customization of manufacturing.

Government

By adopting big data systems, the government can attain efficiencies in terms of cost, output, and novelty. Since the same data set is used in many applications, many departments can work in association with each other. Government plays an important role in innovation by acting in all these domains.

Big data applications can be applied in each and every field. Some of the major areas where big data finds applications include:

agriculture

aviation

cyber security and intelligence

crime prediction and prevention

e-commerce

fake news detection

fraud detection

pharmaceutical drug evaluation

scientific research

weather forecasting and

tax compliance.

I.XI. Big data Usecases

1. 360° VIEW OF THE CUSTOMER

Many enterprises use big data to build a dashboard application that provides a 360° view of the customer. These dashboards pull together data from a variety of internal and external sources, analyze it and present it to customer service, sales and/or marketing personnel in a way that helps them do their jobs.

2. FRAUD PREVENTION

For credit card holders, fraud prevention is one of the most familiar use cases for big data. Even before advanced big data analytics became popular, credit card issuers were using rules-based systems to help them flag potentially fraudulent transactions.

3. SECURITY INTELLIGENCE

On the theme of criminal activity, organizations are also using big data analytics to help them thwart hackers and cyber-attackers. Operating an enterprise IT department generates an enormous amount of log data. In addition, cyber threat intelligence data is available from external sources, such as law enforcement or security providers. Many organizations or now using big data solutions to help them aggregate and analyze all of this internal and external information to help them prevent, detect and mitigate attacks.

4. DATA WAREHOUSE OFFLOAD

One of the easiest — and potentially most cost-effective — ways for organizations to begin using big data tools is to remove some of the burden from their data warehouses. Even among the few organizations that haven’t yet started experimenting with big data analytics, it is common to have a data warehouse that facilitates their business intelligence (BI) efforts.

5. PRICE OPTIMIZATION

Both business-to-consumer (B2C) and business-to-business (B2B) enterprises are also using big data analytics to optimize the prices that they charge their customers. For any company, the goal is to set prices so that they maximize their income. If the price is too

high, they will sell fewer products, decreasing their net returns. But if the price is too low, they may leave money on the table.

6. OPERATIONAL EFFICIENCY

In addition to helping organizations optimize their pricing, big data analytics can also help companies identify other potential opportunities to streamline operations or maximize their profits. Often, this particular big data use case is the purview of BI or financial analysts.

7. RECOMMENDATION ENGINES

Speaking of popularity, one of the most familiar use cases for big data is the recommendation engine. When you are watching a movie at Netflix or shopping for products from Amazon, you probably now take it for granted that the website will suggest similar items that you might enjoy. Of course, the ability to offer those recommendations arises from the use of big data analytics to analyze historical data.

8. SOCIAL MEDIA ANALYSIS AND RESPONSE

The flood of posts that flow through social media outlets like Facebook, Twitter, Instagram and others is one of the most obvious examples of big data. Today, companies are expected to monitor what people are saying about them in social media and respond appropriately — and if they do not, they quickly lose customers.

9. PREVENTIVE MAINTENANCE AND SUPPORT

Many of the big data use cases mentioned so far relate to retail or financial companies, but businesses in manufacturing, energy, construction, agriculture, transportation and similar sectors of the economy can also benefit from big data. In these examples, some of the biggest benefit might come from using big data to improve equipment maintenance.

10. INTERNET OF THINGS

And enterprises in every industry are beginning to see the possibilities of the Internet of Things (IoT). As in the preventive maintenance example, they are using sensors to collect data that they can then analyze to achieve actionable insights. They might track customer or product movement, monitor the weather or keep an eye on security camera footage.

HDFS:
A BRIEF HISTORY OF APACHE HADOOP

High **A**vailability **D**istributed **O**bject **O**riented **P**latform. And that's exactly what Hadoop technology offers developers - high availability via parallel distribution of object-oriented tasks.

Apache Hadoop is an open source, Java-based software platform that manages data processing and storage for big data applications. The platform works by distributing Hadoop big data and analytics jobs across nodes in a computing cluster, breaking them down into smaller workloads that can be run in parallel.

Hadoop processes structured and unstructured data and scale up reliably from a single server to thousands of machines.

When was HADOOP invented?

Apache Hadoop was born out of a need to process ever increasingly large volumes of big data and deliver web results faster as search engines like Yahoo and Google were getting off the ground.

Inspired by Google's MapReduce, a programming model that divides an application into small fractions to run on different nodes, Doug Cutting and Mike Cafarella started Hadoop in 2002 while working on the Apache Nutch project. According to a New York Times article, Doug named Hadoop after his son's toy elephant.

A few years later, Hadoop was spun off from Nutch. Nutch focused on the web crawler element, and Hadoop became the distributed computing and processing portion. Two years after Cutting joined Yahoo, Yahoo released Hadoop as an open-source project in 2008. The Apache Software Foundation (ASF) made Hadoop available to the public in November 2012 as Apache Hadoop.

What's the impact of Hadoop?

Hadoop was a major development in the big data space. In fact, it's credited with being the foundation for the modern cloud data lake. Hadoop made it possible for companies to analyze and query big data sets in a scalable manner using free, open-source software and inexpensive, off-the-shelf hardware.

This was a significant development because it offered a viable alternative to the proprietary data warehouse (DW) solutions and closed data formats that had - until then - ruled the day.

With the introduction of Hadoop, organizations quickly had access to the ability to store and process huge amounts of data, increased computing power, fault tolerance, flexibility in data management, lower costs compared to DWs, and greater scalability. Ultimately, Hadoop paved the way for future developments in big data analytics, like the introduction of Apache Spark.

How does Hadoop work?

Hadoop is a framework that allows for the distribution of giant data sets across a cluster of commodity hardware. Hadoop processing is performed in parallel on multiple servers simultaneously.

Clients submit data and programs to Hadoop. In simple terms, HDFS (a core component of Hadoop) handles the Metadata and distributed file system. Next, Hadoop MapReduce processes and converts the input/output data. Lastly, YARN divides the tasks across the cluster.

With Hadoop, clients can expect much more efficient use of commodity resources with high availability and a built-in point of failure detection. Additionally, clients can expect quick response times when performing queries with connected business systems.

In all, Hadoop provides a relatively easy solution for organizations looking to make the most out of big data.

What language is Hadoop written in?

The Hadoop framework itself is mostly built from Java. Other programming languages include some native code in C and shell scripts for command lines. However, Hadoop programs can be written in many other languages including Python or C++. This allows programmers the flexibility to work with the tools they're most familiar with.

What is the Hadoop ecosystem?

The term Hadoop is a general name that may refer to any of the following:

> The overall Hadoop ecosystem, encompasses both the core modules and related sub-modules.
>
> The core Hadoop modules, including Hadoop Distributed File System (HDFS), Yet Another Resource Negotiator (YARN), MapReduce, and Hadoop Common (discussed below). These are the basic building blocks of a typical Hadoop deployment.

Hadoop-related sub-modules, including: Apache Hive, Apache Impala, Apache Pig, and Apache Zookeeper, and Apache Flume among others. These related pieces of software can be used to customize, improve upon, or extend the functionality of core Hadoop.

What are the core Hadoop modules?

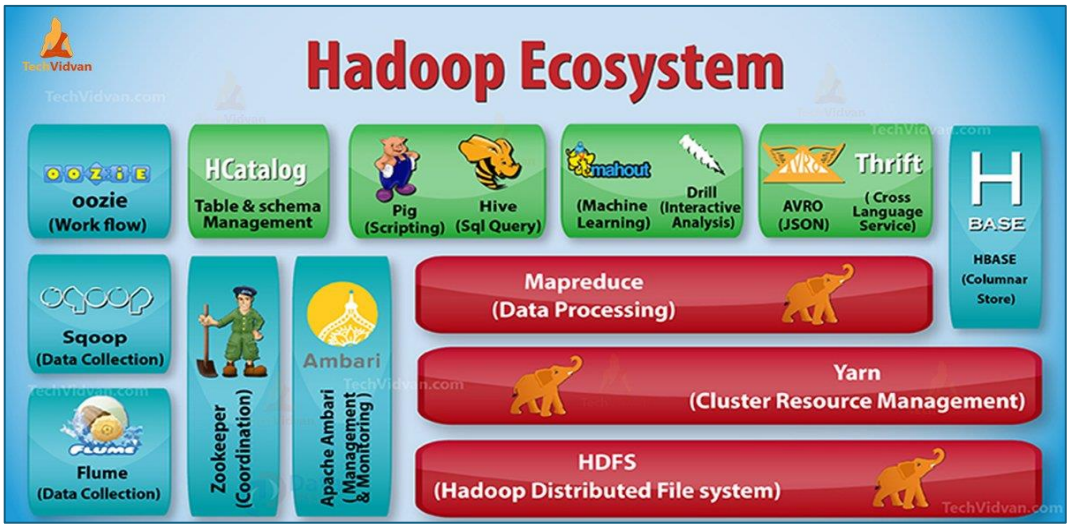
HDFS - Hadoop Distributed File System. HDFS is a Java-based system that allows large data sets to be stored across nodes in a cluster in a fault-tolerant manner. HDFS is the file system component of Hadoop. Yahoo! Has developed and contributed to 80% of HDFS.

YARN - Yet Another Resource Negotiator. YARN is used for cluster resource management, planning tasks, and scheduling jobs that are running on Hadoop.

MapReduce - MapReduce is both a programming model and big data processing engine used for the parallel processing of large data sets. Originally, MapReduce was the only execution engine available in Hadoop. But, later on Hadoop added support for others, including Apache Tez and Apache Spark. Map-Reduce is the Distributed Processing environment of Hadoop. Yahoo has developed 80% of MR.

Hadoop Common - Hadoop Common provides a set of services across libraries and utilities to support the other Hadoop modules.

The Design of HDFS - HDFS Concepts – Blocks – Namenodes and Datanodes



Other Components of Hadoop ecosystem

Other popular packages that are not strictly a part of the core Hadoop modules but that are frequently used in conjunction with them include:

Apache Hive is data warehouse software that runs on Hadoop and enables users to work with data in HDFS using a SQL-like query language called HiveQL.

Apache Impala is the open source, native analytic database for Apache Hadoop.

Apache Pig is a tool that is generally used with Hadoop as an abstraction over MapReduce to analyze large sets of data represented as data flows. Pig enables operations like join, filter, sort, and load.

Apache Zookeeper is a centralized service for enabling highly reliable distributed processing.

Apache Sqoop is a tool designed for efficiently transferring bulk data between Apache Hadoop and structured datastores such as relational databases.

Apache Oozie is a workflow scheduler system to manage Apache Hadoop jobs. Oozie Workflow jobs are Directed Acyclic Graphs (DAGs) of actions.

Why Hadoop?

As we have learned the Introduction, Now, we are going to learn what is the need of Hadoop?

It emerged as a solution to the "**Big Data**" problems-

a. Storage for Big Data – HDFS Solved this problem. It stores Big Data in Distributed Manner. HDFS also stores each file as blocks. Block is the smallest unit of data in a filesystem.

Suppose you have 512MB of data. And you have configured HDFS such that it will create 128Mb of data blocks. So HDFS divide data into 4 **blocks** (512/128=4) and stores it across different DataNodes. It also replicates the data blocks on different datanodes.

Hence, storing big data is not a challenge.

b. Scalability – It also solves the Scaling problem. It mainly focuses on horizontal scaling rather than vertical scaling. You can add extra datanodes to HDFS cluster as and when required. Instead of scaling up the resources of your datanodes.

Hence enhancing performance dramatically.

c. Storing the variety of data – HDFS solved this problem. HDFS can store all kind of data (structured, semi-structured or unstructured). It also follows write once and read many models.

Due to this, you can write any kind of data once and you can read it multiple times for finding insights.

d. Data Processing Speed – This is the major problem of big data. In order to solve this problem, move computation to data instead of data to computation. This principle is **Data locality**.

HADOOP ARCHITECTURE

The goal of designing Hadoop is to develop an inexpensive, reliable, and scalable framework that stores and analyses the rising big data.

Apache Hadoop is a software framework designed by Apache Software Foundation for storing and processing large datasets of varying sizes and formats.

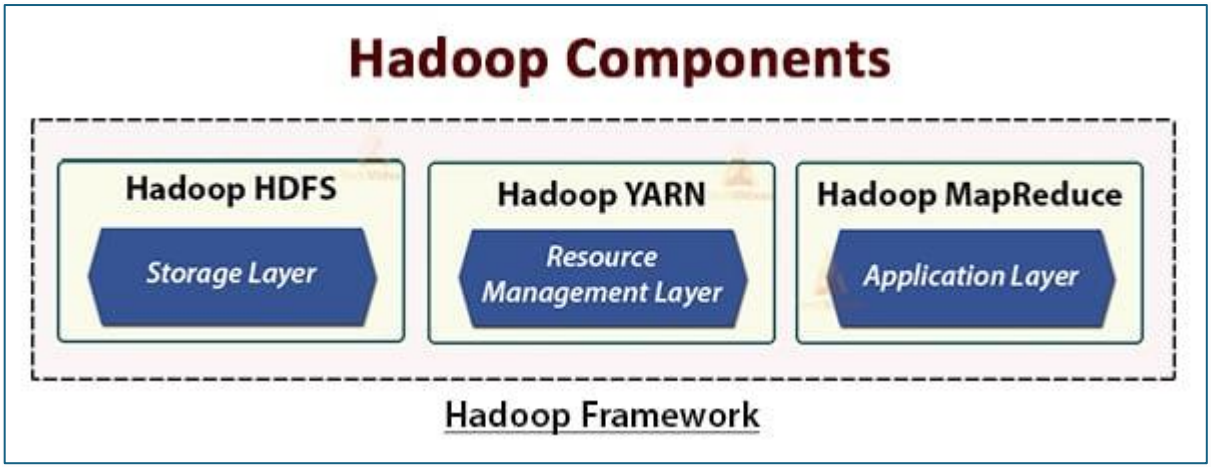
Hadoop follows the master-slave architecture for effectively storing and processing vast amounts of data. The master nodes assign tasks to the slave nodes.

The slave nodes are responsible for storing the actual data and performing the actual computation/processing. The master nodes are responsible for storing the metadata and managing the resources across the cluster.

Slave nodes store the actual business data, whereas the master stores the metadata.

The Hadoop architecture comprises three layers. They are:

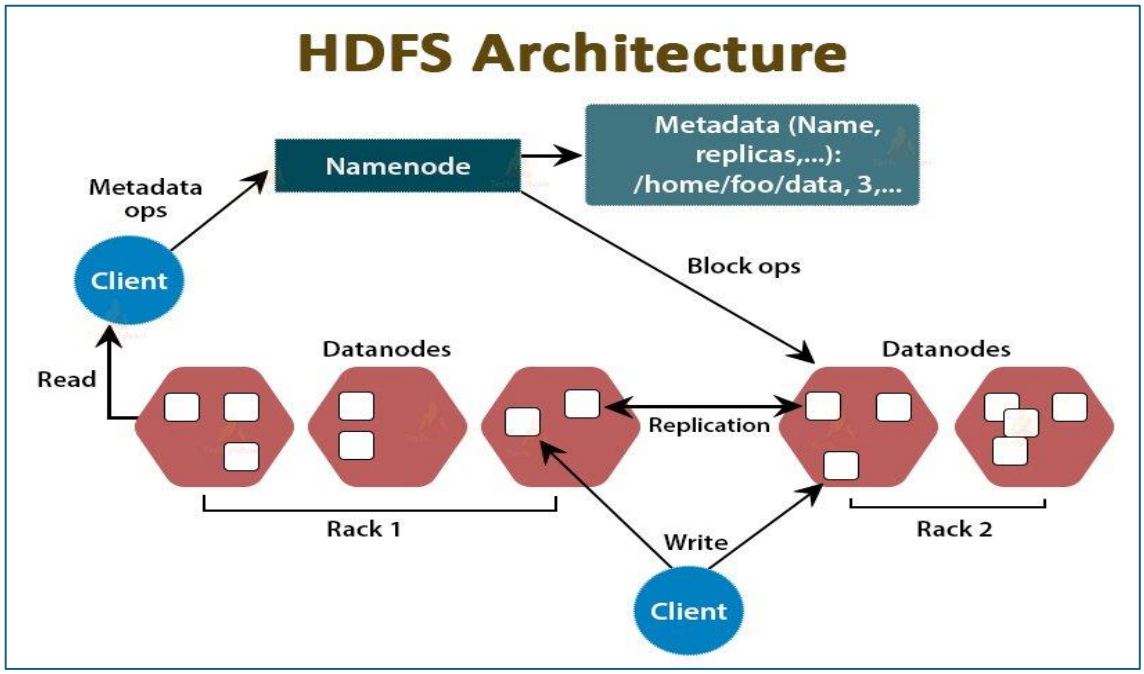
- Storage layer (HDFS)`
- Resource Management layer (YARN)
- Processing layer (MapReduce)



The HDFS, YARN, and MapReduce are the core components of the Hadoop Framework.

Let us now study these three core components in detail.

1. HDFS



HDFS is the Hadoop Distributed File System, which runs on inexpensive commodity hardware. It is the storage layer for Hadoop. The files in HDFS are broken into block-size chunks called data blocks.

These blocks are then stored on the slave nodes in the cluster. The block size is 128 MB by default, which we can configure as per our requirements.

Like Hadoop, HDFS also follows the master-slave architecture. It comprises two daemons- NameNode and DataNode. The NameNode is the master daemon that runs on the master node. The DataNodes are the slave daemon that runs on the slave nodes.

NameNode

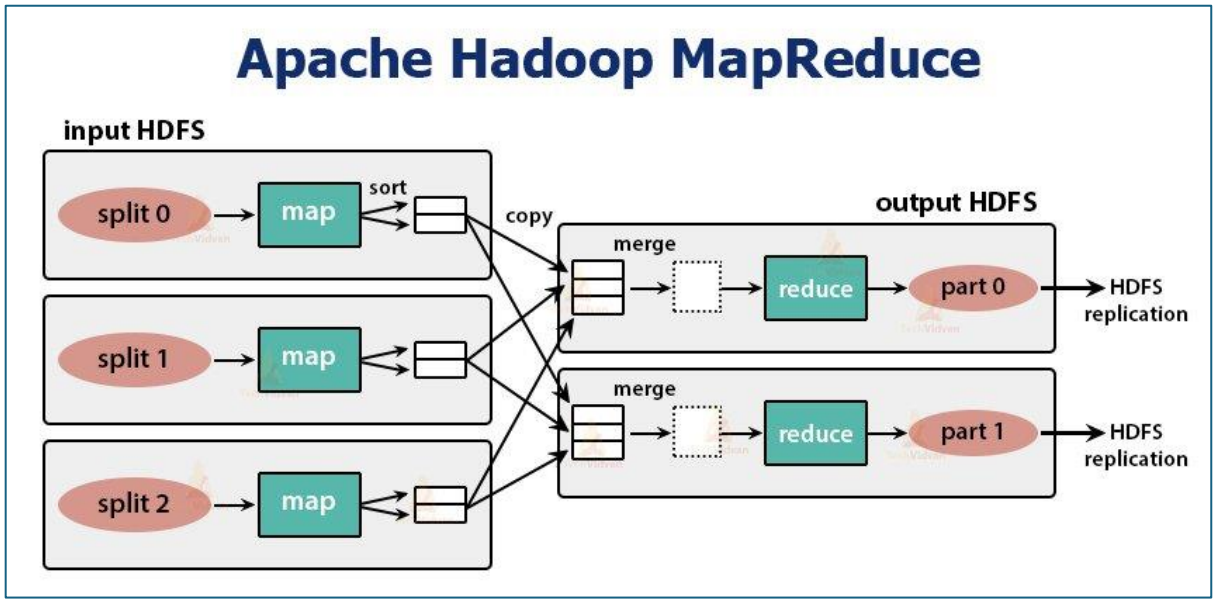
NameNode stores the filesystem metadata, that is, files names, information about blocks of a file, blocks locations, permissions, etc. It manages the Datanodes.

DataNode

DataNodes are the slave nodes that store the actual business data. It serves the client read/write requests based on the NameNode instructions.

DataNodes stores the blocks of the files, and NameNode stores the metadata like block locations, permission, etc.

2. MapReduce



It is the data processing layer of Hadoop. It is a software framework for writing applications that process vast amounts of data (terabytes to petabytes in range) in parallel on the cluster of commodity hardware.

The MapReduce framework works on the <key, value> pairs.

The MapReduce job is the unit of work the client wants to perform. MapReduce job mainly consists of the input data, the MapReduce program, and the configuration information. Hadoop runs the MapReduce jobs by dividing them into two types of tasks that are **map tasks** and **reduce tasks**. The Hadoop YARN schedules these tasks and runs on the nodes in the cluster.
Due to some unfavourable conditions, if the tasks fail, they will automatically get rescheduled on a different node.
The user defines the **map function** and the **reduce function** for performing the MapReduce job. The input to the map function and output from the reduce function is the key, value pair.
The function of the map tasks is to load, parse, filter, and transform the data. The output of the map task is the input to the reduce task. Reduce task then performs grouping and aggregation on the output of the map task.

The MapReduce task is done in two phases-

1. Map phase

a. RecordReader

Hadoop divides the inputs to the MapReduce job into the fixed-size splits called **input splits** or splits. The RecordReader transforms these splits into records and parses the data into records but

it does not parse the records itself. **RecordReader** provides the data to the mapper function in key-value pairs.

b. Map

In the map phase, Hadoop creates one map task which runs a user-defined function called map function for each record in the input split. It generates zero or multiple intermediate key-value pairs as map task output.

The map task writes its output to the local disk. This intermediate output is then processed by the reduce tasks which run a user-defined reduce function to produce the final output. Once the job gets completed, the map output is flushed out.

c. Combiner

Input to the single reduce task is the output from all the Mappers that is output from all map tasks. Hadoop allows the user to define a combiner function that runs on the map output.

Combiner groups the data in the map phase before passing it to Reducer. It combines the output of the map function which is then passed as an input to the reduce function.

d. Partitioner

When there are multiple reducers then the map tasks partition their output, each creating one partition for each reduce task. In each partition, there can be many keys and their associated values but the records for any given key are all in a single partition.

Hadoop allows users to control the partitioning by specifying a user-defined partitioning function. Generally, there is a default Partitioner that buckets the keys using the hash function.

2. Reduce phase:

The various phases in reduce task are as follows:

a. Sort and Shuffle:

The Reducer task starts with a shuffle and sort step. The main purpose of this phase is to collect the equivalent keys together. Sort and Shuffle phase downloads the data which is written by the partitioner to the node where Reducer is running.

It sorts each data piece into a large data list. The MapReduce framework performs this sort and shuffles so that we can iterate over it easily in the reduce task.

The **sort and shuffling** are performed by the framework automatically. The developer through the comparator object can have control over how the keys get sorted and grouped.

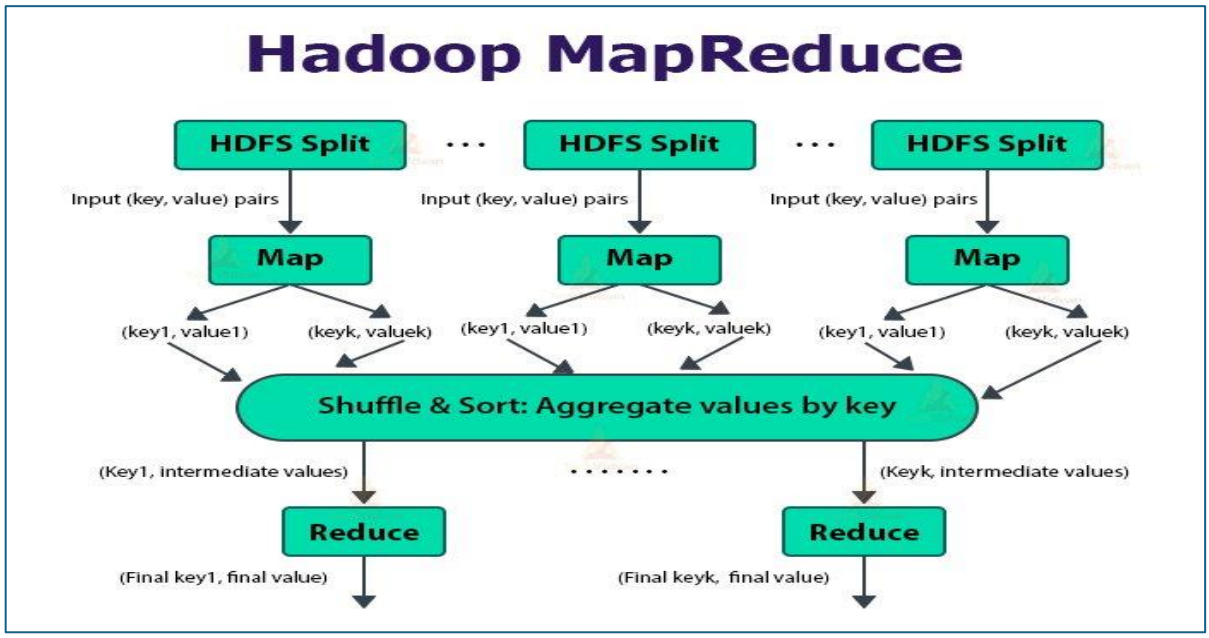
b. Reduce:

The Reducer which is the user-defined reduce function performs once per key grouping. The reducer filters, aggregates, and combines data in several different ways. Once the reduce task is

completed, it gives zero or more key-value pairs to the OutputFormat. The reduce task output is stored in Hadoop HDFS.

c. OutputFormat

It takes the reducer output and writes it to the HDFS file by RecordWriter. By default, it separates key, value by a tab and each record by a newline character.

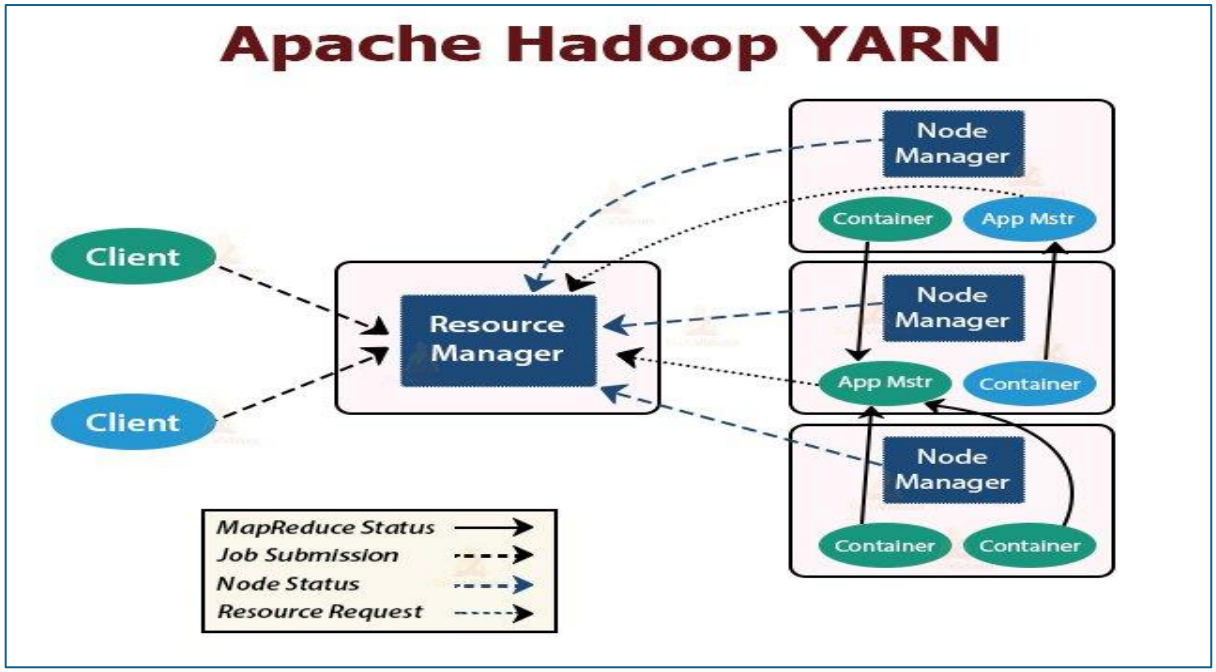


3. YARN

YARN stands for **Yet Another Resource Negotiator**. It is the resource management layer of Hadoop. It was introduced in Hadoop 2.

YARN is designed with the idea of splitting up the functionalities of job scheduling and resource management into separate daemons. The basic idea is to have a global ResourceManager and application Master per application where the application can be a single job or DAG of jobs.

YARN consists of ResourceManager, NodeManager, and per-application ApplicationMaster.



1. ResourceManager

It arbitrates resources amongst all the applications in the cluster. It has two main components that are Scheduler and the ApplicationManager.

a. Scheduler

- The Scheduler allocates resources to the various applications running in the cluster, considering the capacities, queues, etc.
- It is a pure Scheduler. It does not monitor or track the status of the application.
- Scheduler does not guarantee the restart of the failed tasks that are failed either due to application failure or hardware failure.
- It performs scheduling based on the resource requirements of the applications.

b. ApplicationManager

- They are responsible for accepting the job submissions.
- ApplicationManager negotiates the first container for executing application-specific ApplicationMaster.
- They provide service for restarting the ApplicationMaster container on failure.
- The per-application ApplicationMaster is responsible for negotiating containers from the Scheduler. It tracks and monitors their status and progress.

2. NodeManager:

NodeManager runs on the slave nodes. It is responsible for containers, monitoring the machine resource usage that is CPU, memory, disk, network usage, and reporting the same to the ResourceManager or Scheduler.

3. ApplicationMaster:

The per-application ApplicationMaster is a framework-specific library. It is responsible for negotiating resources from the ResourceManager. It works with the NodeManager(s) for executing and monitoring the tasks.

The Application Master is the process that coordinates the execution of an application in the cluster. Each application has its own unique Application Master that is tasked with negotiating resources (Containers) from the Resource Manager and working with the Node Managers to execute and monitor the tasks.

Once the Application Master is started (as a Container), it will periodically send heartbeats to the Resource Manager to affirm its health, and to update the record of its resource demands.

After building a model of its requirements, the Application Master encodes its preferences and constraints in a heartbeat message to the Resource Manager. In response to subsequent heartbeats, the Application Master will receive a lease on Containers bound to an allocation of resources at a particular node in the cluster.

Depending on the Containers it receives from the Resource Manager, the Application Master may update its execution plan to accommodate the excess or lack of resources. Container allocation/de-allocation can take place in a dynamic fashion as the application progresses.

BLOCK CACHING

Centralized cache management in HDFS is an explicit caching mechanism that allows users to specify paths to be cached by HDFS. The NameNode will communicate with DataNodes that have the desired blocks on disk, and instruct them to cache the blocks in off-heap caches.

Centralized cache management in HDFS has many significant advantages.

Explicit pinning prevents frequently used data from being evicted from memory. This is particularly important when the size of the working set exceeds the size of main memory, which is common for many HDFS workloads.

Because DataNode caches are managed by the NameNode, applications can query the set of cached block locations when making task placement decisions. Co-locating a task with a cached block replica improves read performance.

When block has been cached by a DataNode, clients can use a new, more-efficient, zero-copy read API. Since checksum verification of cached data is done once by the DataNode, clients can incur essentially zero overhead when using this new API.

Centralized caching can improve overall cluster memory utilization. When relying on the OS buffer cache at each DataNode, repeated reads of a block will result in all n replicas of the block being pulled into buffer cache. With centralized cache management, a user can explicitly pin only m of the n replicas, saving n-m memory.

Use Cases

Centralized cache management is useful for files that are accessed repeatedly. For example, a small fact table in Hive which is often used for joins is a good candidate for caching. On the other hand, caching the input of a one-year reporting query is probably less useful, since the historical data might only be read once. Centralized cache management is also useful for mixed workloads with performance SLAs. Caching the working set of a high-priority workload ensures that it does not contend for disk I/O with a low-priority workload.

HDFS FEDERATION

The namenode keeps a reference to every file and block in the filesystem in memory, which means that on very large clusters with many files, memory becomes the limiting factor for scaling. HDFS federation, introduced in the 2.x release series, allows a cluster to scale by adding namenodes, each of which manages a portion of the filesystem namespace. For example, one

namenode might manage all the files rooted under /user, say, and a second name- node might handle files under /share.

Under federation, each namenode manages a namespace volume, which is made up of the metadata for the namespace, and a block pool containing all the blocks for the files in the namespace. Namespace volumes are independent of each other, which means namenodes do not communicate with one another, and furthermore the failure of one namenode does not affect the availability of the namespaces managed by other namenodes. Block pool storage is not partitioned, however, so datanodes register with each namenode in the cluster and store blocks from multiple block pools.

To access a federated HDFS cluster, clients use client-side mount tables to map file paths to namenodes. This is managed in configuration using ViewFileSystem and the viewfs:// URIs.

HDFS High Availability

The namenode is a single point of failure (SPOF). If it did fail, all clients—including MapReduce jobs—would be unable to read, write, or list files, because the namenode is the sole repository of the metadata and the file-to-block mapping. In such an event, the whole Hadoop system would effectively be out of service until a new namenode could be brought online.

To recover from a failed namenode in this situation, an administrator starts a new primary namenode with one of the filesystem metadata replicas and configures datanodes and clients to use this new namenode. The new namenode is not able to serve requests until it has

loaded its namespace image into memory
replayed its edit log and
received enough block reports from the datanodes to leave safe mode.

On large clusters with many files and blocks, the time it takes for a namenode to start from cold can be 30 minutes or more.

The long recovery time is a problem for routine maintenance, too. In fact, because unexpected failure of the namenode is so rare, the case for planned downtime is actually more important in practice. Hadoop 2 remedied this situation by adding support for HDFS high availability (HA). In this implementation, there are a pair of namenodes in an active-standby configuration. In the event of the failure of the active namenode, the standby takes over its duties to continue servicing client requests without a significant interruption. A few architectural changes are needed to allow this to happen:

The namenodes must use highly available shared storage to share the edit log. When a standby namenode comes up, it reads up to the end of the shared edit log to synchronize its state with

the active namenode, and then continues to read new entries as they are written by the active namenode.

Datanodes must send block reports to both namenodes because the block mappings are stored in a namenode's memory, and not on disk.

Clients must be configured to handle namenode failover, using a mechanism that is transparent to users.

The secondary namenode's role is subsumed by the standby, which takes periodic checkpoints of the active namenode's namespace.

There are two choices for the highly available shared storage: an NFS filer, or a quorum journal manager (QJM). The QJM is a dedicated HDFS implementation, designed for the sole purpose of providing a highly available edit log, and is the recommended choice for most HDFS installations. The QJM runs as a group of journal nodes, and each edit must be written to a majority of the journal nodes. Typically, there are three journal nodes, so the system can tolerate the loss of one of them. If the active namenode fails, the standby can take over very quickly (in a few tens of seconds) because it has the latest state available in memory: both the latest edit log entries and an up-to-date block mapping. The actual observed failover time will be longer in practice (around a minute or so), because the system needs to be conservative in deciding that the active namenode has failed.

In the unlikely event of the standby being down when the active fails, the administrator can still start the standby from cold. This is no worse than the non-HA case, and from an operational point of view it's an improvement, because the process is a standard operational procedure built into Hadoop.

Failover and fencing

The transition from the active namenode to the standby is managed by a new entity in the system called the failover controller. There are various failover controllers, but the default implementation uses ZooKeeper to ensure that only one namenode is active.

Each namenode runs a lightweight failover controller process whose job it is to monitor its namenode for failures (using a simple heartbeating mechanism) and trigger a failover should a namenode fail.

Failover may also be initiated manually by an administrator, for example, in the case of routine maintenance. This is known as a graceful failover, since the failover controller arranges an orderly transition for both namenodes to switch roles.

In the case of an ungraceful failover, however, it is impossible to be sure that the failed namenode has stopped running. For example, a slow network or a network partition can trigger a

failover transition, even though the previously active namenode is still running and thinks it is still the active namenode.

The HA implementation goes to great lengths to ensure that the previously active namenode is prevented from doing any damage and causing corruption—a method known as fencing.

The QJM only allows one namenode to write to the edit log at one time; however, it is still possible for the previously active namenode to serve stale read requests to clients, so setting up an SSH fencing command that will kill the namenode's process is a good idea.

Stronger fencing methods are required when using an NFS filer for the shared edit log, since it is not possible to only allow one namenode to write at a time (this is why QJM is recommended). The range of fencing mechanisms includes revoking the namenode's access to the shared storage directory (typically by using a vendor-specific NFS command), and disabling its network port via a remote management command.

As a last resort, the previously active namenode can be fenced with a technique rather graphically known as STONITH, or "shoot the other node in the head," which uses a specialized power distribution unit to forcibly power down the host machine.

Client failover is handled transparently by the client library. The simplest implementation uses client-side configuration to control failover. The HDFS URI uses a logical hostname that is mapped to a pair of namenode addresses (in the configuration file), and the client library tries each namenode address until the operation succeeds.

The Command-Line Interface

Hadoop Mainly works on 3 different Modes:

1. Standalone Mode
2. Pseudo-distributed Mode
3. Fully-Distributed Mode

In Standalone Mode none of the Daemon will run i.e., Namenode, Datanode, Secondary Name node, Job Tracker, and Task Tracker. Standalone Mode also means that we are installing Hadoop only in a single system. Hadoop works very much Fastest in this mode among all of these 3 modes.
In Pseudo-distributed Mode we also use only a single node, but the main thing is that the cluster is simulated, which means that all the processes inside the cluster will run independently to each other. All the daemons that are Namenode, Datanode, Secondary Name node, Resource Manager, Node Manager, etc. will be running as a separate process on separate JVM (Java Virtual Machine) or we can say run on different java processes that is why it is called a Pseudo-distributed.
One thing we should remember that as we are using only the single node set up so all the Master and Slave processes are handled by the single system.

In Fully distribute Mode multiple nodes are used few of them run the Master Daemon's that are Namenode and Resource Manager and the rest of them run the Slave Daemon's that are DataNode and Node Manager. Here Hadoop will run on the clusters of Machine or nodes. Here the data that is used is distributed across different nodes.
Filesystems are specified by a URI, and here we use an hdfs URI to configure Hadoop to use HDFS by default, by setting fs.default.name property to hdfs.
Also, in a single system Hadoop environment, we set dfs.replication to 1 so that HDFS doesn't replicate filesystem blocks by the default factor of 3.

Basic Filesystem Operations

We create a directory first just to see how it is displayed in the listing:

```
% hadoop fs -mkdir books

% hadoop fs -ls .

Found 2 items

drwxr-xr-x - tom supergroup 0 2014-10-04 13:22 books

-rw-r--r-- 1 tom supergroup 119 2014-10-04 13:21 quangle.txt
```

The information returned is very similar to that returned by the Unix command ls - l, with a few minor differences.

- The first column shows the file mode.
- The second column is the replication factor of the file (something a traditional Unix filesystem does not have). Remember we set the default replication factor in the site-wide configuration to be 1, which is why we see the same value here. The entry in this column is empty for directories because the concept of replication does not apply to them—directories are treated as metadata and stored by the namenode, not the datanodes.
- The third and fourth columns show the file owner and group.
- The fifth column is the size of the file in bytes, or zero for directories.
- The sixth and seventh columns are the last modified date and time.
- Finally, the eighth column is the name of the file or directory.

HADOOP FILE SYSTEMS

Hadoop has an abstract notion of filesystems, of which HDFS is just one implementation. The Java abstract class org.apache.hadoop.fs.FileSystem represents the client interface to a filesystem in Hadoop, and there are several concrete implementations. The main ones that ship with Hadoop are described in the following table:

Filesystem	URI scheme	Java implementation (all under org.apache.hadoop)	Description
Local	file	fs.LocalFileSystem	A filesystem for a locally connected disk with client-side checksums. Use RawLocal FileSystem for a local filesystem with no checksums.
HDFS	hdfs	hdfs.DistributedFileSystem	Hadoop's distributed filesystem. HDFS is designed to work efficiently in conjunction with MapReduce.
WebHDFS	webhdfs	hdfs.web.WebHdfsFileSystem	A filesystem providing authenticated read/ write access to HDFS over HTTP.
Secure WebHDFS	swebhdfs	hdfs.web.SWebHdfsFileSystem	The HTTPS version of WebHDFS.
HAR	har	fs.HarFileSystem	A filesystem layered on another filesystem for archiving files. Hadoop Archives are used for packing lots of files in HDFS into a

			single archive file to reduce the namenode's memory usage.
View	viewfs	viewfs.ViewFileSystem	A client-side mount table for other Hadoop filesystems. Commonly used to create mount points for federated namenodes
FTP	ftp	fs.ftp.FTPFileSystem	A filesystem backed by an FTP server
S3	s3a	fs.s3a.S3AFileSystem	A filesystem backed by Amazon S3. Replaces the older s3n (S3 native) implementation.
Azure	wasb	fs.azure.NativeAzureFileSystem	A filesystem backed by Microsoft Azure.
Swift System	swift	fs.swift.snative.SwiftNativeFile	A filesystem backed by OpenStack Swift.

Interfaces

Hadoop is written in Java, so most Hadoop filesystem interactions are mediated through the Java API. The filesystem shell, for example, is a Java application that uses the Java FileSystem class to provide filesystem operations. The following provides a description of other interfaces that also work well with HDFS.

HTTP

The HTTP REST API exposed by the WebHDFS protocol makes it easier for other languages to interact with HDFS. Note that the HTTP interface is slower than the native Java client, so should be avoided for very large data transfers if possible.

C

Hadoop provides a C library called libhdfs that mirrors the Java FileSystem interface. It works using the Java Native Interface (JNI) to call a Java filesystem client. There is also a libwebhdfs library that uses the WebHDFS interface.

NFS

It is possible to mount HDFS on a local client's filesystem using Hadoop's NFSv3 gateway. We can then use Unix utilities (such as ls and cat) to interact with the filesystem, upload files, and in general use POSIX libraries to access the filesystem from any programming language.

FUSE

Filesystem in Userspace (FUSE) allows filesystems that are implemented in user space to be integrated as Unix filesystems. Hadoop's Fuse-DFS contrib module allows HDFS (or any Hadoop filesystem) to be mounted as a standard local filesystem. Fuse-DFS is implemented in C using libhdfs as the interface to HDFS.

THE JAVA INTERFACE

The Hadoop FileSystem class: the API for interacting with one of Hadoop's filesystems.

In Hadoop 2 and later, there is a new filesystem interface called FileContext with better handling of multiple filesystems (so a single FileContext can resolve multiple filesystem schemes, for example) and a cleaner, more consistent interface.

A few important main classes which provide I/O operations on Hadoop files.

FileSystem – org.apache.hadoop.fs – An abstract file system API.

IOUtils – org.apache.hadoop.io – Generic i/o code for reading and writing data to HDFS.

IOUtils:

It is a utility class (handy tool) for I/O related functionality on HDFS. It is present in org.apache.hadoop.io package.

Below are some of its important methods which we use very frequently in HDFS File I/O Operations. All these methods are static methods.

copyBytes:

```
IOUtils.copyBytes(InputStream in, OutputStream out, int buffSize, boolean close);
```

This method copies data from one stream to another. The last two arguments are the buffer size used for copying and whether to close the streams when the copy is complete.

readFully:

```
IOUtils.readFully(InputStream in, byte[] buf, int off, int len) ;
```

This methods reads len bytes into byte array. off – offset in the buffer

skipFully:

```
IOUtils.skipFully(InputStream in, long len) ;
```

Similar to readFully. Skips len bytes

writeFully:

```
IOUtils.writeFully(FileChannel fc, ByteBuffer buf, long offset) ;
```

This method writes a ByteBuffer to a FileChannel at a given offset, handling short writes.

```
IOUtils.writeFully(WritableByteChannel bc, ByteBuffer buf);
```

This method writes a ByteBuffer to a WritableByteChannel, handling short writes.

closeStream:

```
IOUtils.closeStream(Closeable stream) ;
```

This method is used to close input or output streams irrespective of any IOException. This method is generally placed finally clause of a try-catch block.

Below are some of the important methods from FileSystem class.

Getting FileSystem Instance:

For any File I/O operation in HDFS through Java API, the first thing we need is FileSystem instance. To get file system instance, we have three static methods from FileSystem class.

static FileSystem get(Configuration conf) — Returns the configured file system implementation.

static FileSystem get(URI uri, Configuration conf) — Returns the FileSystem for this URI.

static FileSystem get(URI uri, Configuration conf, String user) — Get a file system instance based on the uri, the passed configuration and the user.

Opening Existing File:

In order to read a file from HDFS, we need to open an input stream for the same. We can do the same by invoking open() method on FileSystem instance.

```
public FSDataInputStream open(Path f)

public abstract FSDataInputStream open(Path f, int bufferSize)
```

The first method uses a default buffer size of 4 K.

Creating a new File:

There are several ways to create a file in HDFS through FileSystem class. But one of the simplest method is to invoke create() method that takes a Path object for the file to be created and returns an output stream to write to.

public FSDataOutputStream create(Path f)

There are overloaded versions of this method that allow you to specify whether to forcibly overwrite existing files, the replication factor of the file, the buffer size to use when writing the file, the block size for the file, and file permissions.

The create() method creates any parent directories of the file to be written that don’t already exist.

EXAMPLE FOR JAVA IMPLEMENTATIONS OF SOME COMMANDS

i) cat filename COMMAND

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

                //File to read in HDFS
                String uri = args[0];
                Configuration conf = new Configuration();
                //Get the filesystem - HDFS
                FileSystem fs = FileSystem.get(URI.create(uri), conf);
                FSDataInputStream in = null;

                try
                {
                        //Open the path mentioned in HDFS
                        in = fs.open(new Path(uri));
                        IOUtils.copyBytes(in, System.out, 4096,
false);                                 System.out.println("End Of file: HDFS file read
complete");
                }
                finally
                {
                        IOUtils.closeStream(in);
                }
        }
}
```

ii) ls COMMAND

```
import java.io.File;

public class Jls {
    public static void main(String[] args) {
        File dir = new File(System.getProperty("user.dir"));
        String childs[] = dir.list();
        for(String child: childs){
            System.out.println(child);
        }
    }
}
```

iii) Write file into HDFS

```
public static void writeFileToHDFS() throws IOException {
        Configuration configuration = new Configuration();
        configuration.set("fs.defaultFS", "hdfs://localhost:9000");
        FileSystem fileSystem = FileSystem.get(configuration);
        //Create a path
```

```
        String fileName = "read_write_hdfs_example.txt";
        Path hdfsWritePath = new Path("/user/javadeveloperzone/javareadwriteexample/" + fileName);
        FSDataOutputStream fsDataOutputStream = fileSystem.create(hdfsWritePath,true);
        BufferedWriter bufferedWriter = new BufferedWriter(new
OutputStreamWriter(fsDataOutputStream,StandardCharsets.UTF_8));
        bufferedWriter.write("Java API to write data in HDFS");
        bufferedWriter.newLine();
        bufferedWriter.close();
        fileSystem.close();
    }
```

ADDITIONAL REFERENCE FOR HADOOP JAVA INTERFACE CONCEPTS

Addition of Resources:

conf.addResource(String name) — adds a resource called ‘name’

Getting Property Values:

conf.get(String name) — gets value of the property ‘name’.

conf.getBoolean(String name, boolean defaultValue) — gets value of property ‘name’ as boolean

conf.getClass(String name, Class<?> defaultValue) — gets class of property ‘name’

conf.getDouble(String name, double defaultValue)

conf.getFloat(String name, float defaultValue)

conf.getInt(String name, int defaultValue)

conf.getStrings(String name) — Gets ‘,’ delimited values of ‘name’ property as an array of strings.

Setting Property Values:

conf.set(String name, String value) — Set the value of the name property.

conf.setBoolean(String name, boolean value) — set the ‘name’ property to a boolean ‘value’.

conf.setClass(String name, Class<?> theClass, Class<?> intface) — Set the value of the ‘name’ property to the name of a theClass implementing the given interface ‘intface’

conf.setDouble(String name, double value)

conf.setEnum(String name, T value)

conf.setFloat(String name, float value)

conf.setInt(String name, int value)

Reading Data from a Hadoop URL

One of the simplest ways to read a file from a Hadoop filesystem is by using a java.net.URL object to open a stream to read the data from. The general idiom is:

```
InputStream in = null;
try
{
        in = new URL("hdfs://host/path").openStream();
        // process in
}
finally
{
        IOUtils.closeStream(in);
```

Example 3-1 shows a program for displaying files from Hadoop filesystems on standard output, like the Unix cat command.

Example 3-1. Displaying files from a Hadoop filesystem on standard output using a URLStreamHandler

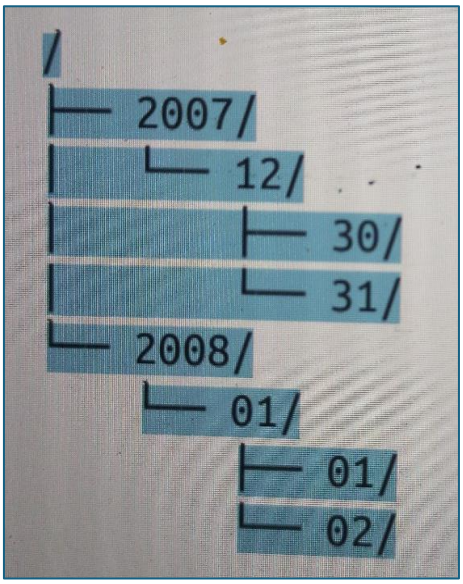
```
public class URLCat
{
        static
        {
                URL.setURLStreamHandlerFactory(new FsUrlStreamHandlerFactory());
        }
        public static void main(String[] args) throws Exception
        {
                InputStream in = null;
                try
                {
                        in = new URL(args[0]).openStream();
                        IOUtils.copyBytes(in, System.out, 4096, false);
                }
                finally
                {
                        IOUtils.closeStream(in);
                }
        }
}
```

The setURLStreamHandlerFactory() method on URL with an instance of FsUrlStreamHandlerFactory is called to make Java recognize Hadoop's hdfs URL scheme.

Hadoop supports the same set of glob characters as the Unix bash shell.

Glob	Name	Matches
*	asterisk	Matches zero or more characters
?	question mark	Matches a single character
[ab]	character class	Matches a single character in the set {a, b}
[^ab]	negated character class	Matches a single character that is not in the set {a, b}
[a-b]	character range	Matches a single character in the (closed) range [a, b], where a is lexicographically less than or equal to b
[^a-b]	negated character range	Matches a single character that is not in the (closed) range [a, b], where a is lexicographically less than or equal to b
{a,b}	alternation	Matches either expression a or b
\c	escaped character	Matches character c when it is a metacharacter

Imagine that logfiles are stored in a directory structure organized hierarchically by date. So, logfiles for the last day of 2007 would go in a directory named /2007/12/31, for example. Suppose that the full file listing is:



Glob	Expansion
/*	/2007 /2008
/*/*	/2007/12 /2008/01
/*/12/*.	/2007/12/30 /2007/12/31
/200?	/2007 /2008
/200[78]	/2007 /2008
/200[7-8]	/2007 /2008
/200[^01234569]	/2007 /2008
/*/*/{31,01}	/2007/12/31 /2008/01/01
/*/*/3{0,1}	/2007/12/30 /2007/12/31
/*/{12/31,01/01}	/2007/12/31 /2008/01/01

DATA FLOW

ANATOMY OF A FILE READ OPERATION

To get an idea of how data flows between the client interacting with HDFS, the name- node, and the datanodes, consider Figure 2, which shows the main sequence of events when reading a file.

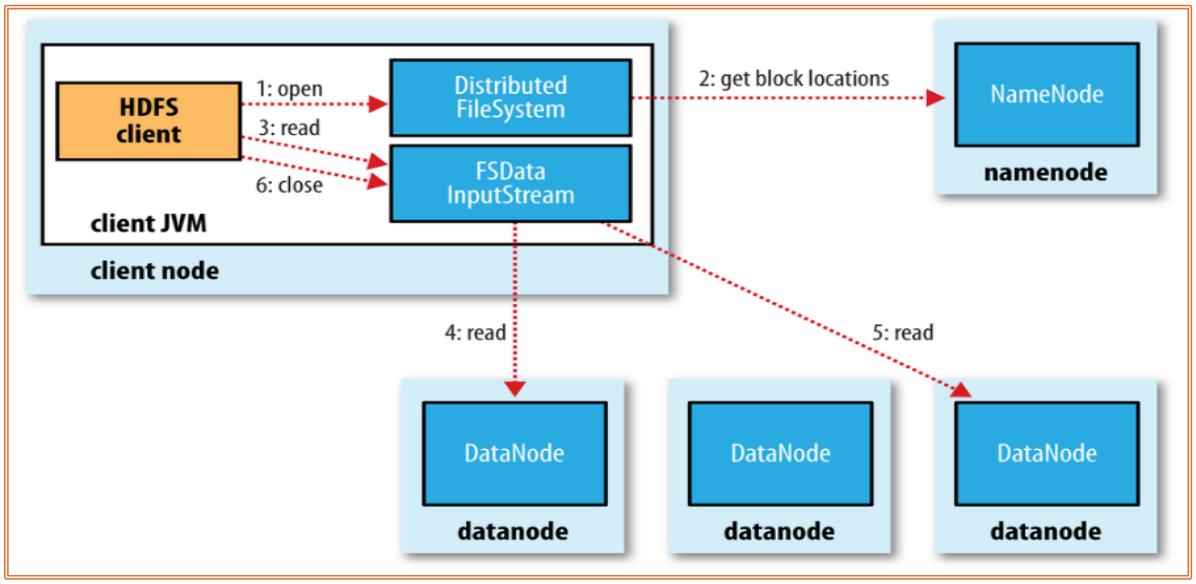


Figure 1 Anatomy of a file read operation in HDFS

The list of operations / activities involved in a File READ are listed below:

i) The client opens the file it wishes to read by calling open() on the FileSystem object, which for HDFS is an instance of DistributedFileSystem (step 1).

ii) DistributedFileSystem calls the namenode, using remote procedure calls (RPCs), to determine the locations of the first few blocks in the file (step 2).

   a) For each block, the namenode returns the addresses of the datanodes that have a copy of that block. Furthermore, the datanodes are sorted according to their proximity to the client (according to the topology of the cluster's network).

   b) If the client is itself a datanode (in the case of a MapReduce task, for instance), the client will read from the local datanode if that datanode hosts a copy of the block. This is called "Short-circuit local reads".

   c) The DistributedFileSystem returns an FSDataInputStream (an input stream that supports file seeks) to the client for it to read data from. FSDataInputStream in turn wraps a DFSInputStream, which manages the datanode and namenode I/O.

iii) The client then calls read() on the stream (step 3). DFSInputStream, which has stored the datanode addresses for the first few blocks in the file, then connects to the first (closest) datanode for the first block in the file.

iv) Data is streamed from the datanode back to the client, which calls read() repeatedly on the stream (step 4).

v) When the end of the block is reached, DFSInputStream will close the connection to the datanode, then find the best datanode for the next block (step 5).
   a) This happens transparently to the client, which from its point of view is just reading a continuous stream.
   b) Blocks are read in order, with the DFSInputStream opening new connections to datanodes as the client reads through the stream.
   c) It will also call the namenode to retrieve the datanode locations for the next batch of blocks as needed.

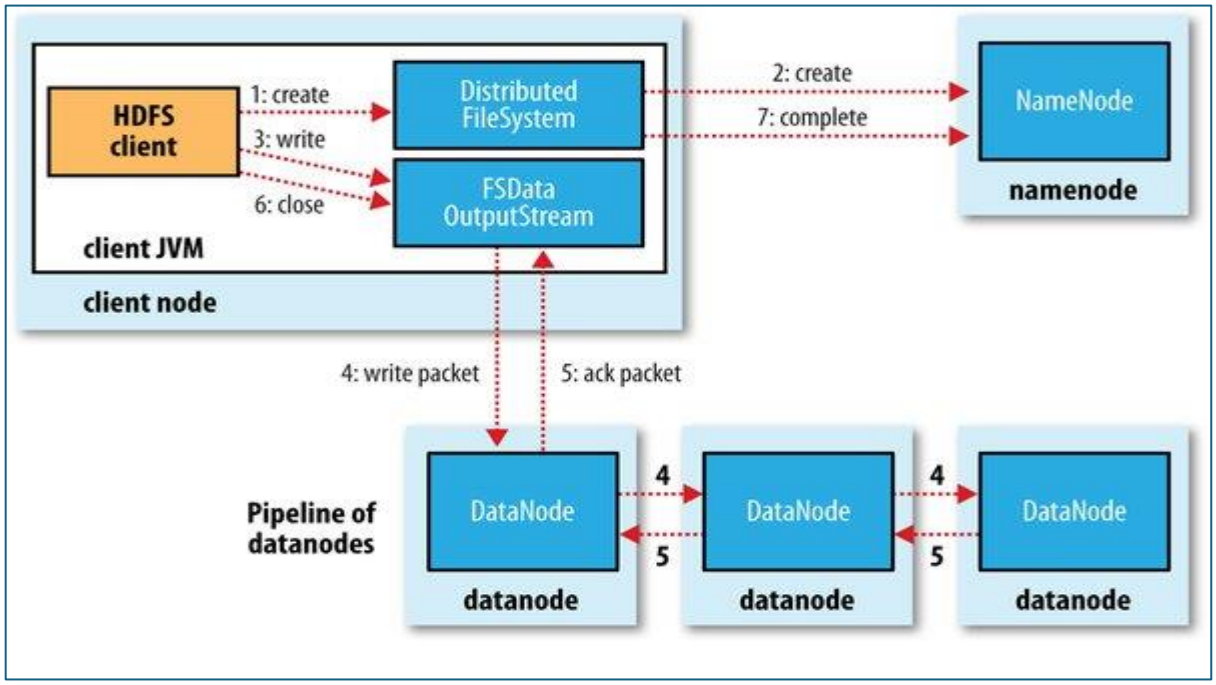
vi) When the client has finished reading, it calls close() on the FSDataInputStream (step 6).

During the READ operation, the following activities also take place:

a) During reading, if the DFSInputStream encounters an error while communicating with a datanode, it will try the next closest one for that block. It will also remember datanodes that have failed so that it doesn't needlessly retry them for later blocks.
b) The DFSInputStream also verifies checksums for the data transferred to it from the datanode. If a corrupted block is found, the DFSInputStream attempts to read a replica of the block from another datanode; it also reports the corrupted block to the namenode.
c) One important aspect of this design is that the client contacts datanodes directly to retrieve data and is guided by the namenode to the best datanode for each block. This design allows HDFS to scale to a large number of concurrent clients because the data traffic is spread across all the datanodes in the cluster. Meanwhile, the namenode merely has to service block location requests (which it stores in memory, making them very efficient) and does not, for example, serve data, which would quickly become a bottleneck as the number of clients grew.

ANATOMY OF A FILE WRITE

We're going to consider the case of creating a new file, writing data to it, then closing the file. This is illustrated in Figure 3.



The list of operations / activities involved in a File WRITE are listed below:

i) The client creates the file by calling create() on DistributedFileSystem (step 1).

ii) DistributedFileSystem makes an RPC call to the namenode to create a new file in the filesystem's namespace, with no blocks associated with it (step 2).

   a) The namenode performs various checks to make sure the file doesn't already exist and that the client has the right permissions to create the file.

   b) If these checks pass, the namenode makes a record of the new file; otherwise, file creation fails and the client is thrown an IOException.

   c) The DistributedFileSystem returns an FSDataOutputStream for the client to start writing data to. Just as in the read case, FSDataOutputStream wraps a DFSOutputStream, which handles communication with the datanodes and namenode.

iii) As the client writes data (step 3), the DFSOutputStream splits it into packets, which it writes to an internal queue called the data queue. The data queue is consumed by the DataStreamer, which is responsible for asking the namenode to allocate new blocks by picking a list of suitable datanodes to store the replicas. The list of datanodes forms a pipeline, and here we'll assume the replication level is three, so there are three nodes in the pipeline. The DataStreamer streams the packets to the first datanode in the pipeline, which stores each packet and forwards it to the second datanode in the pipeline.

Similarly, the second datanode stores the packet and forwards it to the third (and last) datanode in the pipeline (step 4).

iv) The DFSOutputStream also maintains an internal queue of packets that are waiting to be acknowledged by datanodes, called the ack queue. A packet is removed from the ack queue only when it has been acknowledged by all the datanodes in the pipeline (step 5).

v) When the client has finished writing data, it calls close() on the stream (step 6). This action flushes all the remaining packets to the datanode pipeline and waits for acknowledgments before contacting the namenode to signal that the file is complete (step 7).

The namenode already knows which blocks the file is made up of (because Data Streamer asks for block allocations), so it only has to wait for blocks to be minimally replicated before returning successfully.

During the WRITE operation, if a datanode fails, the following activities take place:

If any datanode fails while data is being written to it, then the following actions are taken, which are transparent to the client writing the data.

a) First, the pipeline is closed, and any packets in the ack queue are added to the front of the data queue so that datanodes that are downstream from the failed node will not miss any packets.
b) The current block on the good datanodes is given a new identity, which is communicated to the namenode, so that the partial block on the failed datanode will be deleted if the failed datanode recovers later on.
c) The failed datanode is removed from the pipeline, and a new pipeline is constructed from the two good datanodes.
d) The remainder of the block's data is written to the good datanodes in the pipeline.
e) The namenode notices that the block is under-replicated, and it arranges for a further replica to be created on another node.
f) Subsequent blocks are then treated as normal.

COHERENCY MODEL

A coherency model for a filesystem describes the data visibility of reads and writes for a file. HDFS trades off some POSIX requirements for performance, so some operations may behave differently than you expect them to.

1. After creating a file, it is visible in the filesystem namespace. However, any content written to the file is not guaranteed to be visible, even if the stream is flushed. So, the file appears to have a length of zero.
2. Once more than a block's worth of data has been written, the first block will be visible to new readers. This is true of subsequent blocks, too: it is always the current block being written that is not visible to other readers.
3. HDFS provides a way to force all buffers to be flushed to the datanodes via the hflush() method on FSDataOutputStream. After a successful return from hflush(), HDFS guarantees that the data written up to that point in the file has reached all the datanodes in the write pipeline and is visible to all new readers.
4. The hflush() does not guarantee that the datanodes have written the data to disk, only that it's in the datanodes' memory (so in the event of a data center power outage, for example, data could be lost). For this stronger guarantee, use hsync() instead. The behavior of hsync() is similar to that of the fsync() system call in POSIX that commits buffered data for a file descriptor.
5. But these two, hflush() and hsync(), leave an overhead on the HDFS. So, it is the decision to be taken whether to use them or not depending the application under consideration.

PARALLEL COPYING WITH distcp

The HDFS access patterns that we have seen so far focus on single-threaded access. It's possible to act on a collection of files—by specifying file globs, for example—but for efficient parallel processing of these files, you would have to write a program yourself. Hadoop comes with a useful program called distcp for copying data to and from Hadoop filesystems in parallel.
One use for distcp is as an efficient replacement for

```
hadoop fs -cp.
```

For example, you can copy one file to another with:

```
% hadoop distcp file1 file2
```

You can also copy directories:

```
% hadoop distcp dir1 dir2
```

If dir2 does not exist, it will be created, and the contents of the dir1 directory will be copied there. You can specify multiple source paths, and all will be copied to the destination.
If dir2 already exists, then dir1 will be copied under it, creating the directory structure dir2/dir1. If this isn't what you want, you can supply the -overwrite option to keep the same directory

structure and force files to be overwritten. You can also update only the files that have changed using the -update option. This is best shown with an example. If we changed a file in the dir1 subtree, we could synchronize the change with dir2 by running:

```
% hadoop distcp -update dir1 dir2
```

KEEPING AN HDFS CLUSTER BALANCED

When copying data into HDFS, it's important to consider cluster balance. HDFS works best when the file blocks are evenly spread across the cluster, so you want to ensure that distcp doesn't disrupt this.

For example, if you specified -m 1, a single map would do the copy, which—apart from being slow and not using the cluster resources efficiently— would mean that the first replica of each block would reside on the node running the map (until the disk filled up).

The second and third replicas would be spread across the cluster, but this one node would be unbalanced. By having more maps than nodes in the cluster, this problem is avoided. For this reason, it's best to start by running distcp with the default of 20 maps per node.

However, it's not always possible to prevent a cluster from becoming unbalanced. Perhaps you want to limit the number of maps so that some of the nodes can be used by other jobs. In this case, you can use the balancer tool to subsequently even out the block distribution across the cluster.

HDFS FILE SYSTEM SHELL

The File System (FS) shell includes various shell-like commands that directly interact with the Hadoop Distributed File System (HDFS) as well as other file systems that Hadoop supports, such as Local FS, HFTP FS, S3 FS, and others. The FS shell is invoked by:

```
bin/hadoop fs <args>
```

All FS shell commands take path URIs as arguments. For HDFS the scheme is hdfs, and for the Local FS the scheme is file. The scheme and authority are optional. If not specified, the default scheme specified in the configuration is used.

An HDFS file or directory such as /parent/child can be specified as hdfs://namenodehost/parent/child or simply as /parent/child (given that your configuration is set to point to hdfs://namenodehost).

Most of the commands in FS shell behave like corresponding Unix commands. Error information is sent to stderr and the output is sent to stdout.