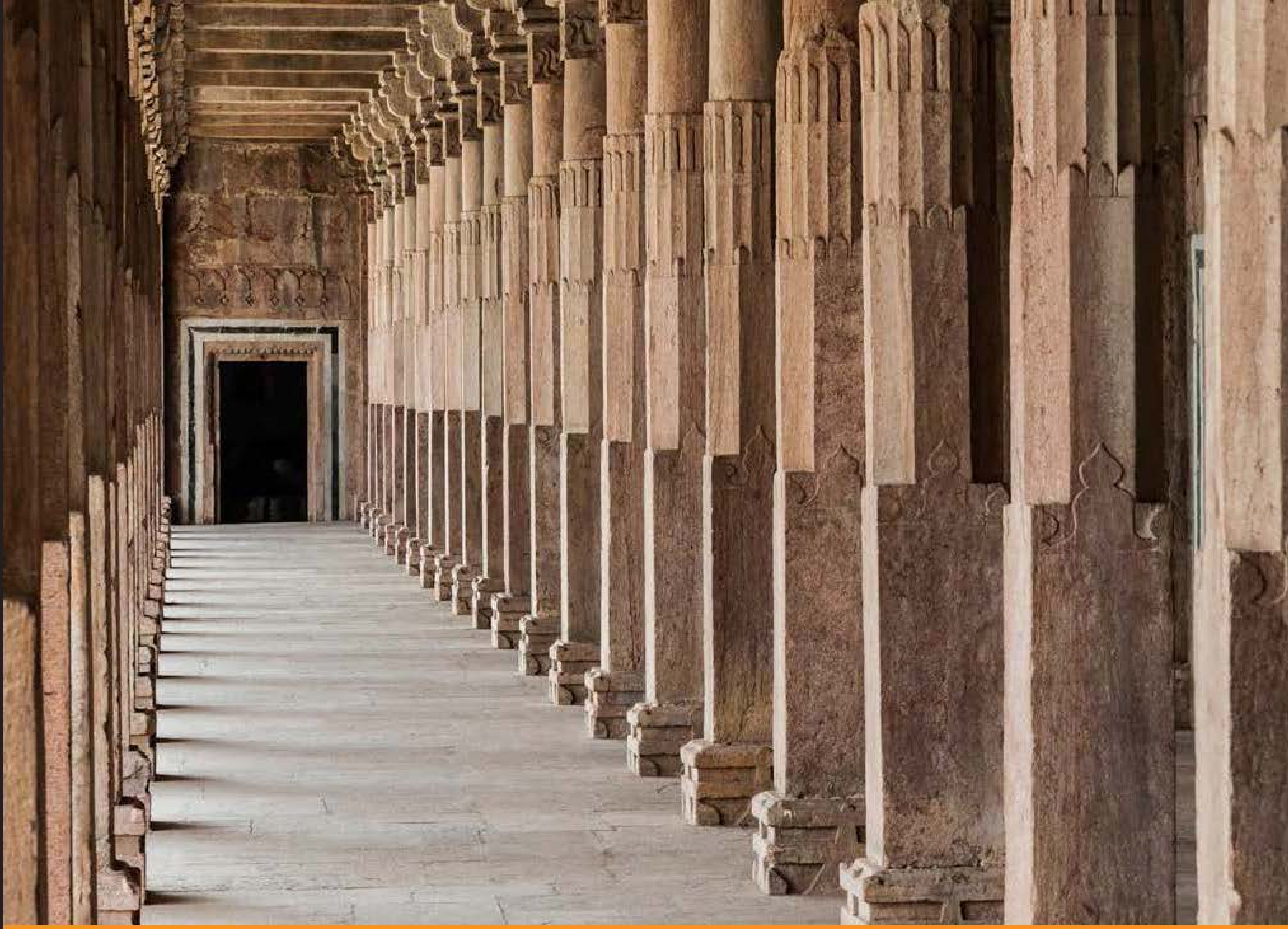


Community Experience Distilled

Hadoop Explained

An introduction to the most popular Big Data platform in the world

Aravind Shenoy

Hadoop Explained

An introduction to the most popular Big Data platform in the world

Aravind Shenoy

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

Hadoop Explained

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

First Published: June 2014

Production reference: 1100614

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

www.packtpub.com

Cover Image by Feroze Babu (mail@feroze.me)

Credits

Author
Aravind Shenoy

Reviewer
Mark Kerzner

Commissioning Editor
Edward Gordon

Content Development Editor
Madhuja Chaudhari

Technical Editors
Pankaj Kadam
Veena Pagare

Proofreader
Paul Hindle

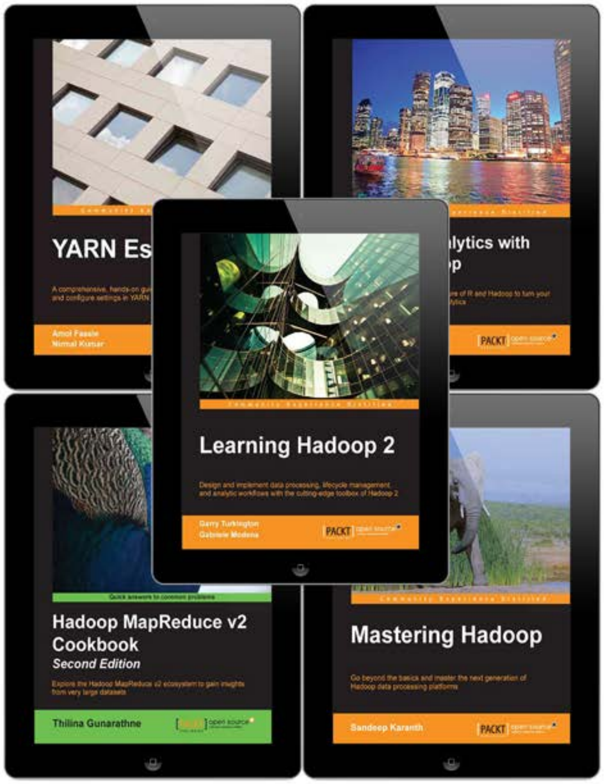
Graphics
Sheetal Aute

Production Coordinator
Adonia Jones

Cover Work
Adonia Jones

About the Author

Aravind Shenoy is an in-house author at Packt Publishing. An Engineering graduate from the Manipal Institute of Technology, his core interests include technical writing, web designing, and software testing. He is a native of Mumbai, India, and currently resides there. He has authored books such as, *Thinking in JavaScript* and *Thinking in CSS*. He has also authored the bestselling book *HTML5 and CSS3 Transition, Transformation, and Animation*, *Packt Publishing* (`http://www.packtpub.com/html5-and-css3-for-transition-transformation-animation/book`). He is a music buff with *The Doors*, *Oasis*, and *R.E.M* ruling his playlists.


YARN Es
Amol Fasale
Nirmal Kumar
lytics with
Learning Hadoop 2
Design and implement data processing, lifecycle management and analytic workflows with the cutting-edge toolbox of Hadoop 2
Garry Turkington
Gabriele Modena
PACKT open source
Quick answers to common problems
Hadoop MapReduce v2 Cookbook
Second Edition
Explore the Hadoop MapReduce v2 ecosystem to gain insights from very large datasets
Thilina Gunarathne
Mastering Hadoop
Go beyond the basics and master the next generation of Hadoop data processing platforms
Sandeep Karanth

Overview

With the almost unfathomable increase in web traffic over recent years, driven by millions of connected users, businesses are gaining access to massive amounts of complex, unstructured data from which to gain insight.

When Hadoop was introduced by Yahoo in 2007, it brought with it a paradigm shift in how this data was stored and analyzed. Hadoop allowed small- and medium-sized companies to store huge amounts of data on cheap commodity servers in racks. This data could thus be processed and used to make business decisions that were supported by ‘Big Data’.

Hadoop is now implemented in major organizations such as Amazon, IBM, Cloudera, and Dell to name a few. This book introduces you to Hadoop and to concepts such as MapReduce, Rack Awareness, YARN, and HDFS Federation, which will help you get acquainted with the technology.

Hadoop Explained

Understanding Big Data

A lot of organizations used to have structured databases on which data processing would be implemented. The data was limited and maintained in a systematic manner using database management systems (think RDBMS). When Google developed its search engine, it was compounded with the task of maintaining a large amount of data, as the web pages used to get updated on a regular basis. A lot of information had to be stored in the database, and most of the data was in an unstructured format. Video, audio, and web logs resulted in a humongous amount of data. The Google search engine is an amazing tool that users can use to search for the information they need with a lot of ease. Research on data can determine the user preferences which can be used to increase the customer base as well as customer satisfaction. An example of that would be the advertisements found on Google.

As we all know, Facebook is widely used today, and the users upload a lot of content in the form of videos and other posts. Hence, a lot of data had to be managed quickly. With the advent of social networking applications, the amount of data to be stored increased day by day and so did the rate of increase. Another example would be financial institutions. Financial institutions target customers by the data at their disposal, which shows the trends in the market. They can also determine user preferences by their transaction history. Online portals also help to determine the purchase history of the customers, and based on the pattern, they gauge the need of the customers, thereby customizing the portals according to the market trend. Hence, data is very crucial, and the increase in data at a rapid pace is of significant importance. This is how the concept of Big Data came into existence.

Let's now understand the concept of the conventional architecture used prior to the Big Data revolution. Conventionally, servers with a lot of processing power along with a huge amount of storage and memory (high-end servers) were used comprehensively. If the amount of data was increasing, the storage capacity would be increased, and eventually, the data would be moved to a larger server with a higher capacity and massive processing power. Suppose the data is in the range of to terabytes and eventually increases to petabytes; there would be a need to scale up. However, there is only so much a single server can do. There is a limit to the size and capacity of a single system. Moreover, these servers were very expensive and not feasible, as having a single server would result in allowing a single point of failure.

The second option would be to use multiple low-end servers in conjunction with each other. Such an approach would be economical; however, the processing has to be done on multiple servers. It is difficult to program the processing to be implemented on multiple servers. There is a drawback to this approach. Even though the capacity of the processors has increased considerably, the speed of storage devices and memory has not increased relatively. The hard disks available nowadays cannot feed the data as fast as the processors can process it. There was one more problem with this methodology. Suppose a part of the data needs to be accessed by all the servers for computing purposes. In that case, there is a high chance of a bottleneck failure. The concept of distributed computing was useful, but it could not resolve the issue of failures that was crucial to data processing at a basic level.

Hence, it was vital that a different approach be used to solve this issue. The concept of moving the data to the processors was an outdated process. If the data is in petabytes or terabytes, data transfer throughout the cluster would be a time-consuming process. Instead, it would be logical to move processing towards the data. That way, it would resolve the latency issue and also ensure high throughput.

Failures are part and parcel of networking. It is very likely that machines would fail at some point or the other. Due to failures at the bottleneck, it was vital that the probability of failure was taken into consideration. Therefore, the concept of introducing commodity hardware came into existence.

> Commodity hardware refers to economical, low-performance systems that are able to function properly without any special devices.

In the concept of Big Data, the processing would be directed towards the huge amount of data, and to store the data, we would use commodity hardware. One more point to be considered is that distributed computing would be applied, wherein replication of data would be possible. If there is a failure, then the replicated data would be considered and the failure on some machine in the cluster would not be considered; as a matter of fact, it would be re-executed on some other node. This will ensure a smooth flow of operation and will not affect the overall process.

Google implemented the concept of **MapReduce** for computing. For storage, it created its own filesystem, also known as the **Google File System**. However, the concept of MapReduce was implemented by a lot of organizations such as Yahoo!, and to counter the issue of storage, the **Hadoop Distributed File System** (**HDFS**) came into existence. Currently, the Apache foundation is handling the Hadoop project. The concept of MapReduce as well as that of the Hadoop Distributed File System will be explained in this book so that we understand the concept of Big Data.

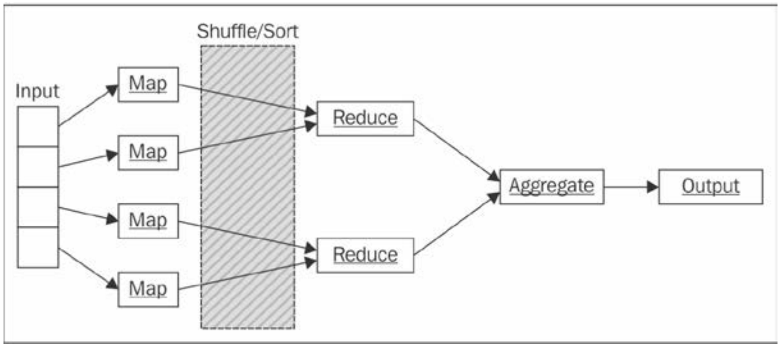
We also have to deal with different types of devices used nowadays, such as tablets and smartphones. The brisk rise in the amount of data can be attributed to these data-creative devices. With the advent of these devices, it is all about data. Users now want to access data everywhere. Therefore, the websites and web applications on the Internet have to deliver and update information at a very high speed. Blogging is an activity that is common nowadays. Comments, tweets, and uploaded content on blogs have to be maintained efficiently. Since data upload and download is taking place so often, it is imperative that we have systems in place that can process the requests in a timely fashion.

MapReduce

Prior to the evolution of Big Data, organizations used powerful servers for computational and storage purposes. Apart from that, a lot of money was spent on specialists handling these servers as well as the licensing of the various systems in place. This was alright until the data storage was in gigabytes. Moreover, the data was structured, and hence we could store it in a systematic manner. The drawback was that only large organizations with sufficient finances at their disposal could afford this kind of setup. However, in this era, with the increase in social media, a lot of data is unstructured. Google came out with the concept of MapReduce to counter this issue.

> At the time of writing, Google had a patent on MapReduce.

When dealing with a huge amount of data (in petabytes or terabytes), MapReduce seems to be the solution. As the name suggests, MapReduce consists of two functions: `map` and `reduce`. Let's look at the following schematic diagram to understand what MapReduce is all about:



In MapReduce, there are no rows or columns as in an RDBMS. Instead, data is in the form of key-value pairs. The key defines the information that the user is looking for. The value is the data associated with that specific key.

Let's understand an example of key-value pairs:

- <Shoes: ABC>
- <Shoes: PQR>
- <Shoes: XYZ>

The key in this case is *Shoes*. The value represents the brand of data that is associated with Shoes. Suppose the user wants to sort out shoes according to their brand name. In this case, the previously mentioned key-value pairs will make sense. Usually, the key-value pairs are defined in the following manner:

```
<k1:v1>
```

Where `k1` is the key and `v1` is the value.

Map

The mapper transforms the input data into intermediate values.

The intermediate values are also in the key-value pair format. However, an intermediate value can also be different from the input. It is not mandatory that the intermediate data is the same as the input. The `map` function is responsible for the transformation.

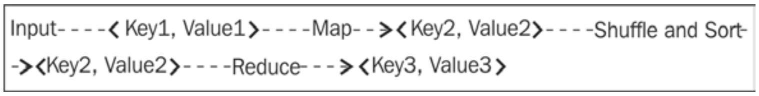
Sort/shuffle

The next is the sort and shuffle phase. Here, the data is sorted and the same data is sent to the reducer for computing purposes. At times, the data may have the same input key. Hence, it is sorted in such a manner that it groups the common keys together and then the shuffling takes place wherein the data is fed to the reducers.

Reduce

The `reduce` function combines the output and gives us an aggregated value in a list format. The final output may be single or multiple lists depending on the computing.

Hence, the entire process can be summed up in the following way:



Input----< Key1, Value1>----Map-->< Key2, Value2>----Shuffle and Sort-->< Key2, Value2>----Reduce---> < Key3, Value3 >

Let's understand this theory with the help of a practical example. In this example, we will take the case of an online shopping website. Suppose the user wants to buy some shoes and a shirt from the online shopping website.

The user will click on the criteria to sort out shirts and shoes according to their brand names. Suppose the user is looking for the **ABC** brand of shoes and the **XYZ** brand of shirts. MapReduce will take the input in the form of key-value pairs and the sorting would be done on various nodes. Let's assume that on one node, we get the following key-value pair:

ABC shoes: 1000

XYZ shirts: 200

Suppose on the other node used for computing, we get the following key-value pair:

ABC shoes: 500

XYZ shirts: 350

Hence, the `map` function has produced the following output, but we need to remember that these are the intermediate values. Similarly, after computation at all the nodes, the `reduce` function combines all these results and the final output will be displayed as follows:

ABC shoes: 5000

XYZ shirts: 1300

As we can see, the output from the `map` function was sorted and the intermediate values were fed to the reducer. The `reduce` function gave a list of the total number of the **ABC** shoes and the **XYZ** shirts on the website. This example is confined to a search of the required items on the online shopping website. However, in real-life scenarios, even complex computing can be performed with ease by using the MapReduce concept.

Hadoop

MapReduce works best when the amount of data is very large. However, we need a filesystem that can store this amount of data. Moreover, data transfer during computing needs an advanced framework. It is expensive to maintain an infrastructure on which MapReduce can be implemented. Hence, we cannot go for a conservative client-server model as it would defeat the purpose of distributed computing. This is where Hadoop comes into the picture.

Hadoop is an open source project using which we can process huge data sets across clusters of commodity servers. In Hadoop, we use the simple MapReduce concept for computing purposes. Hadoop has its own distributed filesystem known as HDFS.

Advantages of using Hadoop

The following sections explain the major advantages of using Hadoop.

Scalability

Since commodity servers are used in Hadoop, it is easy and economical to increase the number of machines in case of extra load or increasing the number of users.

Fault tolerance

The chances of failure cannot be ruled out. HDFS offers redundancy and recovery. If one of the commodity servers stops functioning, the other nodes have the data stored as a result of replication, which is an imperative feature of Hadoop. Re-execution of tasks is an important feature as the computation will be shifted to a different node in case of the failure of a specific node. Hence, there is no data loss, and this also ensures high availability.

Resource sharing

Hadoop follows the distributed computing concept. Hence, the resources and CPUs across the clusters and racks are used in conjunction with each other. Parallel computation can be achieved easily with Hadoop.

Components that make up a Hadoop 1.x cluster

Before we venture into the working of HDFS in conjunction with MapReduce, we need to understand the terms in Hadoop explained in the following sections.

NameNode

A NameNode is a single point of failure for HDFS. In Hadoop, data is broken down into blocks for processing. The information regarding these blocks as well as the entire directory tree of the data is stored in the NameNode. In short, the metadata is stored in the NameNode. In addition to the NameNode, we also have the secondary NameNode. However, this is not a replacement for the NameNode. Checkpoints and snapshots are taken by the secondary NameNode when it contacts the NameNode. In case of a failure, these snapshots and checkpoints are used to restore the filesystem.

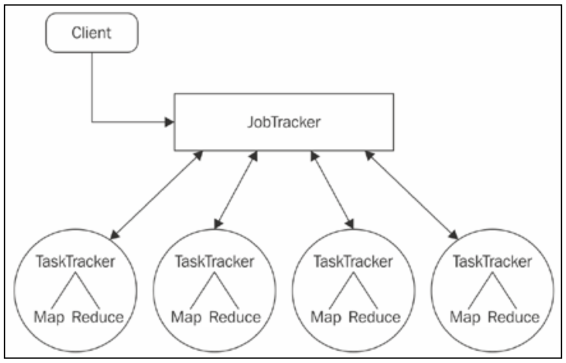
DataNode

DataNodes are used to store blocks of data. On request, they also assist in retrieving data. The NameNode keeps track of all the blocks of data that are stored on the DataNode. The DataNode updates the NameNode periodically to keep it in sync with the smooth flow of operations.

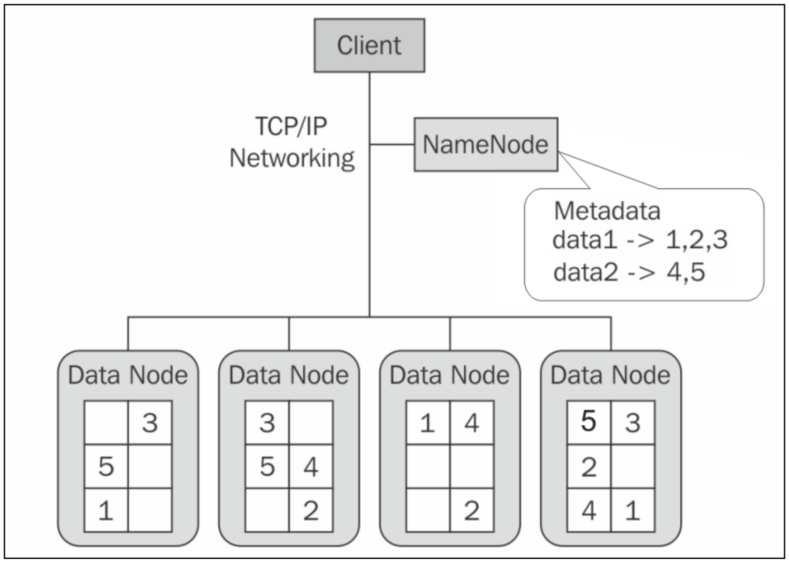
JobTracker and TaskTracker

A JobTracker implements the MapReduce feature on various nodes in the cluster. A JobTracker contacts the NameNode to get information about the location of the data. Once it gets the location, it contacts the TaskTracker that is close to the DataNode that stores the specific data. A TaskTracker will send regular heartbeats (the signal in this case is generally referred to as a heartbeat) to the JobTracker to notify it that it is still functional. If the TaskTracker has failed at some particular node, the lack of a heartbeat will notify the JobTracker that the node is not functional, and the JobTracker reassigns the job to a different TaskTracker. The JobTracker updates the information as soon as the job is completed. Hence, the client contacting the JobTracker will know about the availability of the desired nodes.

Let's have a look at the following schematic diagram to understand the process flow:



Now let's understand the nuances of HDFS to understand how it works. Let's look at the following schematic diagram to understand the functionality of HDFS:

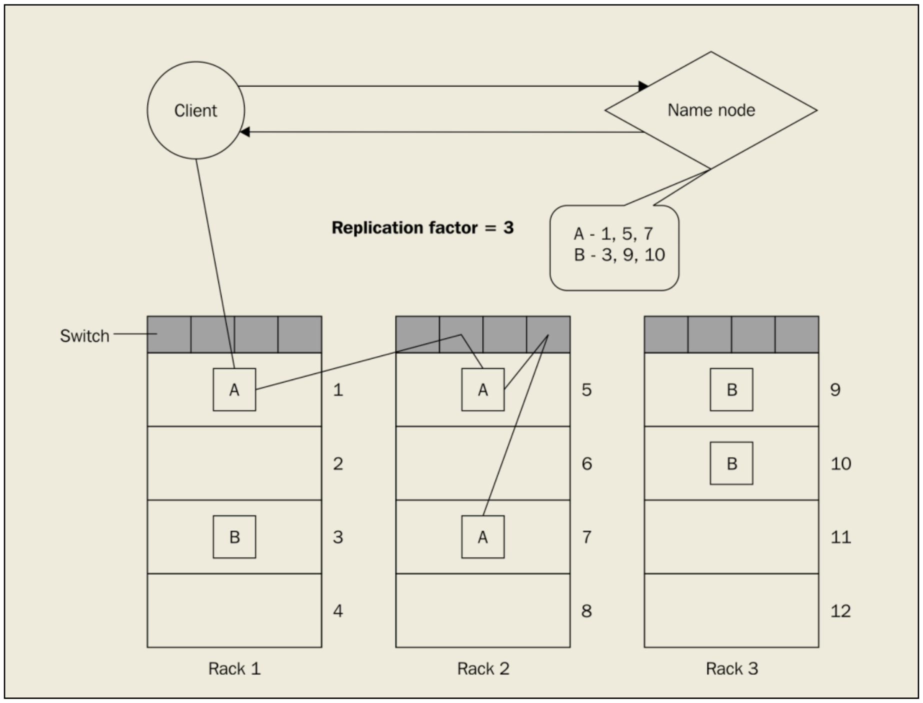


As shown in the previous diagram, **NameNode** is contacted by **Client**. **NameNode** contains the metadata and the location of the data. Once the location is known to the client, it directly interacts with the DataNode. Data is split into blocks in HDFS. **data1** is split into three blocks (**1**, **2**, and **3**) whereas **data2** is split into two blocks (**4** and **5**). The data is stored in the DataNodes. In the preceding diagram, we have four DataNodes. We can see that block **1** is present in the first DataNode as well as in the third and fourth DataNode. This is due to the replication factor. The default replication factor is `3`, and hence, in the diagram, the data is stored in three separate DataNodes. If the replication factor is `4`, it will store the four blocks in four DataNodes. The protocol used is TCP/IP.

In a real-life scenario, there is a huge amount of data. Hence, a lot of commodity servers are used. At times, as the data increases in terabytes and petabytes, the scale-out methodology is used. The number of servers increases, and at times, there may be thousands of servers. With Hadoop, the data can be managed and stored with ease.

Rack awareness

An important concept to understand is rack awareness. Let's look at the following schematic diagram to understand the concept better:



In this scenario, we have three racks: **Rack 1**, **Rack 2**, and **Rack 3**. **Rack 1** has DataNodes **1**, **2**, **3**, and **4**. **Rack 2** has DataNodes **5**, **6**, **7**, and **8**. **Rack 3** has DataNodes **9**, **10**, **11**, and **12**.

Rack awareness is an imperative feature in HDFS. The replication factor used is `3`. Hence, the data will be stored on three DataNodes. Data block **A** is stored on DataNodes **1**, **5**, and **7**. The client will contact the NameNode to get the location of the data. From the NameNode, the client gets the information that the data is stored on DataNode **1** on **Rack 1** and DataNodes **5** and **7** on **Rack 2**. It is quite obvious that the data is stored on different racks. The reason is that if the rack has a downtime and is not functional, access to data is possible as the data is replicated on different rack. The replication assists in fault tolerance and rules out the chaos in terms of data loss, as we have a copy of the data on a different rack.

The client will interact with the NameNode to get the location of the data. Hence, the NameNode is just used to store the location of the data and the directory tree, and not the data itself. The reason the NameNode does not store the data is that the NameNode is a single point of failure. If the data was to be streamed through the NameNode, it would be disastrous as there would be a bottleneck failure due to concurrency.

The client will contact the NameNode for the location. The NameNode gives that information to the client and the client will then interact directly with the DataNodes. In the previous diagram, the NameNode would have notified the client that the data is stored in DataNodes **1**, **5**, and **7**. Subsequently, the client will contact the DataNode **1** on **Rack 1**. Then, the DataNode **1** will send a signal to the other DataNodes that have the same data block **A**. Hence, the DataNode **1** will send a signal to DataNode **5** on **Rack 2**, which in turn will send the signal to DataNode **7** on **Rack 2**. If we observe the previous diagram, we can see that the data block **A** exists in DataNodes **5** and **7** on the same **Rack 2**. This is to ensure low latency. The DataNode **5** does not have to send a signal out of the rack. It will send a signal to a switch on the same rack instead, and it will divert the signal to DataNode **7**. Thus, the DataNodes notify the client about the data block **A** stored at different locations in the cluster.

We must understand that if the replication factor is set to `1`, then the client will interact with that specific DataNode, and will not replicate the data block. Commodity servers can fail at any point of time. Suppose in the previous example, the DataNode **1** fails. In that case, the NameNode does not receive a heartbeat. Hence, there is no communication. The NameNode will then look for another DataNode in the cluster. Usually, it will look out for a nearby DataNode. It will then inform the client and the client will contact the other DataNode as per the notification of the NameNode. Hence, replication ensures fault tolerance in the cluster topology.

Hadoop 2.x (Next Generation)

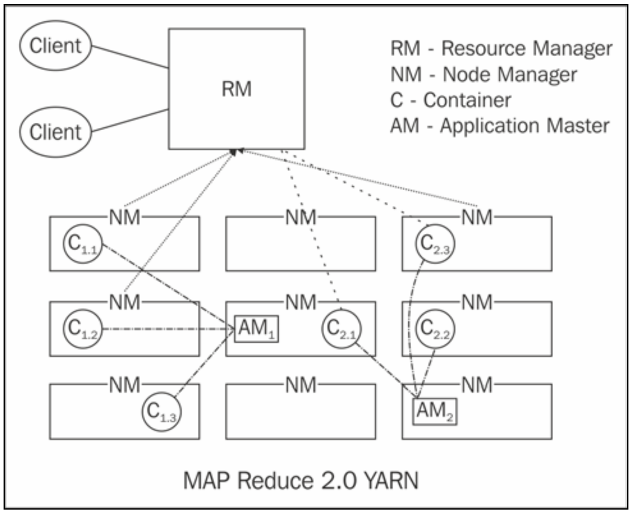
Hadoop 1.x had its limitations due to scalability, low availability, non-optimal resource utilization, as well as lack of support for alternate paradigms and services.

For example, the maximum cluster size was around 5,000 nodes, the maximum number of concurrent tasks was around 40,000, and iterative applications in MapReduce were 10 times slower. To counter these kinds of limitations, Hadoop MapReduce 2.0 (YARN) came into existence. Let's delve into the concept to understand it better.

The Apache Hadoop community has recently declared Apache Hadoop 2.x as GA (General Availability).

Hadoop MapReduce 2.0 (YARN)

Let's have a look at the following schematic representation of YARN to learn more about its components:



The resource manager is at the root of the YARN hierarchy and governs the entire cluster. While the resource manager manages the process of allocating computing resources to the application, the application master manages the life cycle of the application (an application in MapReduce 2.0 is a single job that is executed by the application master). There is also one more entity, which is the per-machine node manager server that takes care of all the user processes on the specific machine. The resource manager coupled with the node manager accounts for all the computation. The resource manager undertakes the division of the resources (memory, CPU, network bandwidth, and so on) to the underlying node managers.

The application master negotiates resources with the resource manager and then works in conjunction with the node manager to monitor the execution and resource consumption of containers. A container in YARN represents a resource on a single node of a given cluster.

The node manager is accountable for managing resources and deployment on a node. The node manager takes care of several tasks such as managing the life cycle of the containers, monitoring the resources, tracking the health of the node, and keeping up to date with the resource manager.

Initially, the client submits an application request to the resource manager. The resource manager negotiates the resources for a container and then launches an application master for that particular application. The application master in turn registers with the resource manager and negotiates the appropriate containers using the resource-request protocol. The application master then takes the specific container and presents it to the node manager of the host which incorporates that container. The application execution takes place under the supervision of the application master. The entire life cycle of the container is monitored until completion. On completion, the application master de-registers itself from the resource manager. This briefly sums up the functionality of how YARN works.

YARN differs from MapReduce in various ways. Prior to YARN, you required a different cluster for other computing models and programming paradigms such as **Bulk Synchronous Parallel** (**BSP**). YARN is more general than MapReduce and other computing models such as BSP that can co-exist along with MapReduce in a single cluster. Also, there are no fixed slots for Map and Reduce tasks. The containers in YARN don't have a fixed purpose and can be used for Map, Reduce, or something else. This leads to optimum cluster utilization and high performance. The separation of resource management and application life cycle management results in a highly scalable architecture. Unlike legacy MapReduce, you can run various versions of Hadoop in a single cluster. YARN is fully compatible with your existing MapReduce applications and users can run their applications on top of YARN without any hindrance.

Changes in HDFS for Hadoop 2.x

Hadoop 2.0 addresses the concept of high availability which was a limitation in the legacy HDFS. In the legacy version of HDFS, there was the concept of the secondary NameNode which used to keep a copy of the NameNode metadata. If the NameNode crashes, you would have to take the metadata copy from the secondary NameNode and use it to resume the work at hand once the NameNode is functional again. The secondary NameNode didn't have any failover capability and the NameNode was a single point of failure.

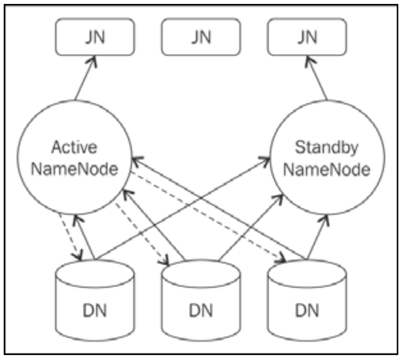
Hadoop 2.0 introduces the concept of more than one NameNode. If the primary NameNode crashes, the redundant NameNode (the standby NameNode) takes over so that the cluster is active at any given point of time. The primary NameNode and the standby NameNode are in an active/passive configuration. Thus, the standby NameNode acts like a slave with the ability to provide a failover in any situation.

There are two ways in which the failover is implemented:

- Quorum-based storage
- Shared storage using **Network File System** (**NFS**)

Quorum-based storage

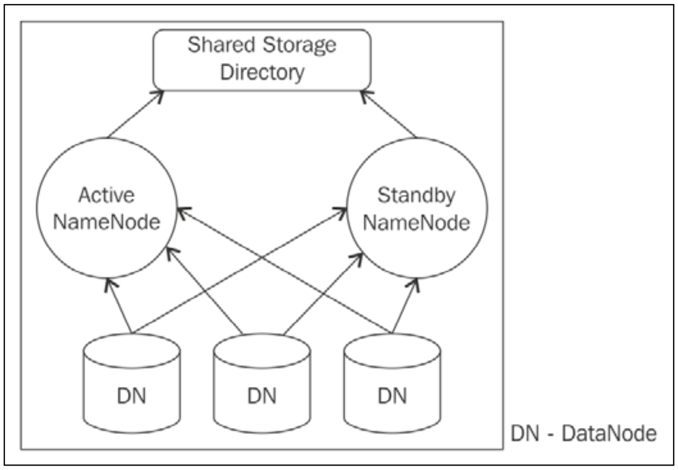
Before we delve in to the Quorum-based storage, let's have a look at the following schematic diagram to understand it better:



In the preceding diagram, DataNode is denoted by **DN** and Journal Node is denoted by **JN**. Both the active NameNode and the standby NameNode communicate with a group of daemons called Journal Nodes. If there are changes or modifications in the namespace, the active NameNode logs the changes to the majority of Journal Nodes. An active NameNode has read-write access whereas the standby NameNode has only read access. The standby NameNode constantly watches the changes to the edit log file, and on coming across such edits, it incorporates them in its namespace. Also, the DataNodes communicate with both the NameNodes and update the block location information and send heartbeats to them, thereby keeping the standby NameNode up to date with the cluster. If there is a failover, the standby NameNode after reading all the edits promotes itself to the active state. This node will perform the task of writing to the Journal Nodes, thereby making it the new active NameNode and preventing the failed NameNode from interfering because only one NameNode has write access at a given point of time.

Shared storage using NFS

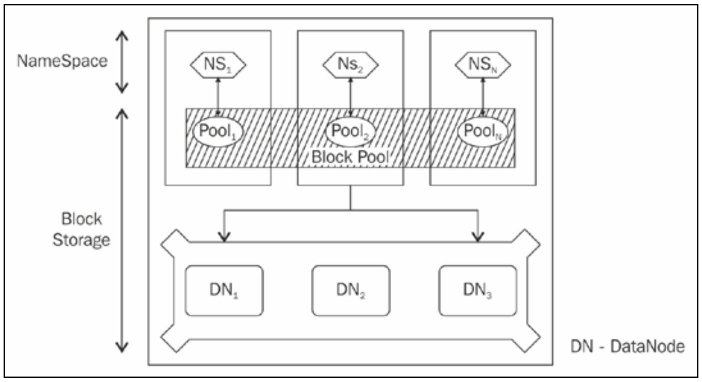
Before we delve into the concept of Shared storage using NFS, let's have a look at the following schematic diagram to understand it better:



In this scenario, instead of Journal Nodes, both the nodes have access to a shared directory or shared storage device. The modifications to the namespace are logged in an edit file in the shared directory. The standby NameNode constantly watches the changes to the edit files, and on coming across edits, it incorporates them in its namespace. Similar to the Quorum method, the standby NameNode, after reading all the edits, promotes itself to the active state. However, the failover procedure is a bit different in this case. Since only one of the NameNodes can be active at a given point of time, there should be no clash between them as it can lead to incorrect results or data loss. Therefore, a fencing mechanism is used wherein the fencing cuts the previously active NameNode from accessing the shared directory or storage. Thus, the standby NameNode in case of a failover is the only one that has access to the shared storage, thereby erasing any type of conflict.

HDFS Federation

The legacy HDFS was restricted to a single namespace and had its limitations, and to counter this, the concept of HDFS Federation came into existence. HDFS Federation enables the use of multiple namespaces/NameNodes in a single cluster opposed to just a single namespace in the legacy HDFS system. As the legacy system is limited to a single namespace/NameNode, scalability is an issue. Moreover, the block storage is tightly coupled with the namespace, thereby restricting other services from accessing the block storage directly. As the latest MapReduce (YARN) and the recent paradigms support more than 1 million tasks concurrently, there was a requirement for a powerful HDFS. HDFS Federation resolves this by enabling you to use multiple namespaces/NameNodes. The NameNodes in Federation are independent of each other and use DataNodes as common storage of blocks. Let's have a look at the following schematic diagram to understand it better:



The Block Pool is a set of blocks that are managed by a single namespace. The Block Pool manages block management tasks, including processing block reports and maintaining the location of blocks. As mentioned earlier, the DataNodes are a common storage for all the blocks in the Block Pool. A namespace allocates a block ID for new blocks without communicating with the other namespaces as they are federated and do not depend on each other. This capability enables the DataNodes to serve numerous NameNodes irrespective of the failure of any specific NameNode. Each Block Pool has a dependency on a particular NameNode, and in case of any mishap or deletion of a namespace, it is deleted from the DataNode.

Thus, you can achieve a high level of scalability by adding multiple NameNodes resulting in multiple namespaces. Multiple NameNodes results in a considerable amount of read/write operations, thereby increasing the throughput and achieving high performance. Isolation of NameNodes can be attained as they are independent of each other, resulting in isolated namespaces for experimental as well as live applications without the hindrance of an overload.

Hadoop is a vast subject, and the Apache Hadoop Community has several projects such as Mahout, ZooKeeper, and Ambari to mention a few. The learning curve is always on the upside. The sea of knowledge is far beyond.