**HADOOP FILE SYSTEM**

HDFS is a distributed file system that handles large data sets running on commodity hardware. It is used to scale a single Apache Hadoop cluster to hundreds (and even thousands) of nodes. HDFS is one of the major components of [Apache Hadoop](https://www.ibm.com/in-en/analytics/hadoop), the others being [MapReduce](https://www.ibm.com/in-en/topics/mapreduce" \o "mapreduce)and YARN. HDFS should not be confused with or replaced by [Apache HBase](https://www.ibm.com/in-en/topics/hbase), which is a column-oriented non-relational database management system that sits on top of HDFS and can better support real-time data needs with its in-memory processing engine.

**The goals of HDFS**

* Fast recovery from hardware failures

Because one HDFS instance may consist of thousands of servers, failure of at least one server is inevitable. HDFS has been built to detect faults and automatically recover quickly.

* Access to streaming data

HDFS is intended more for batch processing versus interactive use, so the emphasis in the design is for high data throughput rates, which accommodate streaming access to data sets.

* Accommodation of large data sets

HDFS accommodates applications that have data sets typically gigabytes to terabytes in size. HDFS provides high aggregate data bandwidth and can scale to hundreds of nodes in a single cluster.

* Portability

To facilitate adoption, HDFS is designed to be portable across multiple hardware platforms and to be compatible with a variety of underlying operating systems.

**An example of HDFS**

Consider a file that includes the phone numbers for everyone in the United States; the numbers for people with a last name starting with A might be stored on server 1, B on server 2, and so on.

With Hadoop, pieces of this phonebook would be stored across the cluster, and to reconstruct the entire phonebook, your program would need the blocks from every server in the cluster.

To ensure availability if and when a server fails, HDFS replicates these smaller pieces onto two additional servers by default. (The redundancy can be increased or decreased on a per-file basis or for a whole environment; for example, a development Hadoop cluster typically doesn’t need any data redundancy.) This redundancy offers multiple benefits, the most obvious being higher availability.

The redundancy also allows the Hadoop cluster to break up work into smaller chunks and run those jobs on all the servers in the cluster for better scalability. Finally, you gain the benefit of data locality, which is critical when working wi

**HDFS, Read & Write Operation**

Hadoop obeys Master-Slave Architecture for distributed data processing and data storage. A Hadoop cluster is made up of an individual master and various slave nodes. A NameNode is master and DataNodes are slave nodes.

## NameNode:

An HDFS contains only one NameNode, a master server that maintains the file system namespace and controls entrance to files by clients. Besides, there are several DataNodes, typically, one per node in the cluster, that control storage connected to the nodes that they run on. Inside, a file is split into more than one blocks, and these blocks are kept in a set of DataNodes. The NameNode performs file system namespace services like renaming, closing, opening, and files and folders. It also manages the mapping of blocks to DataNodes. The DataNodes are accountable for assisting read and write calls from the file system’s clients. The DataNodes too do block replication, deletion, and creation upon guidance from the NameNode.

The NameNode, as well as DataNode, are portions of software intended to work on commodity machines.

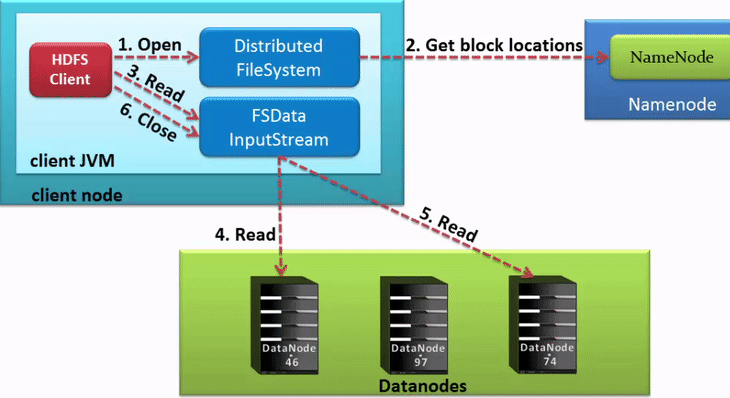
## DataNode:

The commodity machines usually work on a GNU/Linux OS. Devices that support Java can run NameNode and as well as DataNode. Due to JAVA, [HDFS](https://www.h2kinfosys.com/blog/what-is-hadoop-an-introduction/) can be deployed on any machine that supports JAVA.

The NameNode is the judge and folder for all HDFS metadata. The system is created in such a way that clients’ data never moves through the NameNode.

## Read operation in HDFS

While reading data from HDFS, we need to get the information about the location where our respective data is stored from NameNode. The NameNode provides a handle to read data from data nodes. NameNode also provides an authentication token to the client so the only client with an authentication token can read data from the data node.



### Step1:

Client Node needs to interacts with the NameNode to get the information of the data nodes. For that client will send an open request to the [Distributed file system](https://www.webopedia.com/TERM/D/distributed_file_system.html).

### Step 2:

The distributed file system will talk with the NameNode and get’s the authorization token and the location of the nodes.

### Step 3:

The client will send a read request to the common JAVA  input stream to read data from the data nodes.

### Step 4:

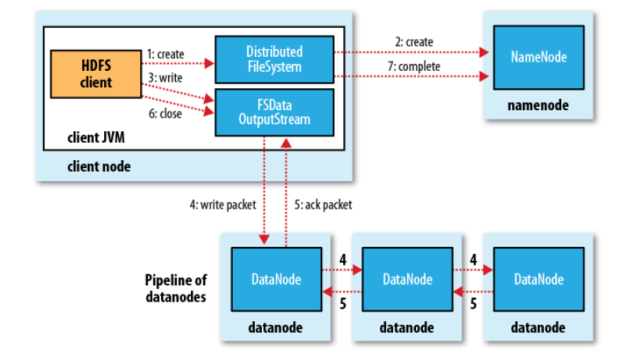
The input stream will read the data from the data nodes.

### Step 5:

The client after getting data will send the Close command to the input stream.

## Write operation in HDFS

Writing data is a little bit complex than reading data.



### Step1:

Client Node needs to interacts with the NameNode to get the information of the data nodes where data need to be stored. For that client will send a create request to the Distributed file system.

### Step 2:

The distributed file system will talk with the NameNode using a remote procedure call and get’s the authorization token and asks to create a new file. NameNode checks whether the file is requested already exists or not. Once everything is fine the NameNode will send back the information of the location.

### Step 3:

The client will send a read request to the common JAVA  output stream to write data in the data nodes.

### Step 4:

The input stream will write the data in the data nodes. A pipeline is created for the process of replication. We are creating a replication of 3 levels.

### Step 5:

The client after writing the data will send the Close command to the output stream.

To communicate with Hadoop’s filesystem programmatically, Hadoop gives many JAVA classes. Package named org.apache.hadoop.fs includes classes helpful in the administration of a file in Hadoop’s filesystem. These services include open, read, write, and close.

The code given below can be used to read data from HDFS.

FileSystem fileSystem = FileSystem.get(confguration);

Path path\_of\_file = new Path("/path/to/file.ext");

if (!fileSystem.exists(path\_of\_file)) {

    System.out.println("File not exists”);

    return;

}

FSDataInputStream in = fileSystem.open(path\_of\_file);

int numBytes = 0;

while ((numBytes = in.read(b))> 0) {

    System.out.prinln((char)numBytes));

}

in.close();

out.close();

fileSystem.close();

# Unix command-line tools for data analysis in Hadoop

## head and tail

Enter **head** and **tail**. Both are utilities for showing a specified number of lines from the top or bottom of the file. If you don't specify the number of lines you want to see, you'll get 10

$ tail -n 3 jan2017articles.csv

02 Jan 2017,Article,Scott Nesbitt,3 t

………………..

$ head -n 1 jan2017articles.csv

Post date,Content type,Author

……………

## wc

Short for "word count," **wc** can count the number of bytes, characters, words, or lines in the file.

$ wc -l jan2017articles.csv

93 jan2017articles.csv

## grep

With **grep**, you can search a file or other input for a particular pattern of characters

$ grep -i "security" jan2017articles.csv

30 Jan 2017,Article,Tiberius Hefflin,4 ways to improve your security online right now,3,/article/17/1/4-ways-improve-your-online-security,Security and encryption,1242

…………..

## tr

format data for analysis

$ grep "20 Jan 2017" jan2017articles.csv | tr ',' '\t' > jan20only.tsv

First, we searched for the date in question, using **grep**. We piped this output to the **tr** command, which we used to replace the commas with tabs (denoted with **'\t'**) the **>** character redirected the output to our new file instead of the screen.

$ cat jan20only.tsv

20 Jan 2017 Article Kushal Das 5 ways to expand your project's contributor base 2 /article/17/1/expand-project-contributor-base Getting started 690

20 Jan 2017 Article D Ruth Bavousett How to write web apps in R with Shiny 2 /article/17/1/writing-new-web-apps-shiny Web development 218

20 Jan 2017 Article Jason Baker "Top 5: Shell scripting the Cinnamon Linux desktop environment and more" 0 /article/17/1/top-5-january-20 Top 5 214

20 Jan 2017 Article Tracy Miranda How is your community promoting diversity? 1

## sort

**sort** command to sort by the word count column

$ sort -nr -t$'\t' -k8 jan20only.tsv | head -n 1

FIrst, we're using the **sort**command to sort by the number of words. The **-nr** option tells **sort** to do a numeric sort, and to return the results in reverse order (largest to smallest). The next **-t$'\t'**tells **sort**that the delimiter is the tab ('\t'). The **-k8**portion of the command tells sort to use the eighth column, which is the column for word count in our example. Finally, the whole output is piped to **head** with instructions just to show the top line, which is our result, the article from this file with the highest word count.

## Sed

**sed**, short for stream editor,

create a new file with no headers from our list of articles

$ sed '1 d' jan2017articles.csv > jan17no\_headers.csv

The **'1 d'** option tells **sed** to delete the first line.

………………………………………………………………………….

<https://data-flair.training/blogs/key-value-pair-in-hadoop-mapreduce/>

<https://data-flair.training/blogs/data-locality-in-hadoop-mapreduce/>

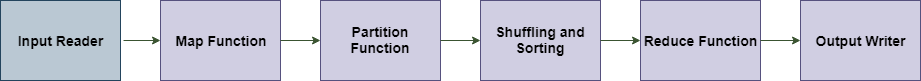
<https://www.javatpoint.com/data-flow-in-mapreduce>

MapReduce is used to compute the huge amount of data . To handle the upcoming data in a parallel and distributed form, the data has to flow from various phases.

**Data Flow In MapReduce**

Phases of Map

Reduce data flow



Input reader

The input reader reads the upcoming data and splits it into the data blocks of the appropriate size (64 MB to 128 MB). Each data block is associated with a Map function.

Once input reads the data, it generates the corresponding key-value pairs. The input files reside in HDFS.

Note - The input data can be in any form.

Map function

The map function process the upcoming key-value pairs and generates the corresponding output key-value pairs. The map input and output type may be different from each other.

Partition function

The partition function assigns the output of each Map function to the appropriate reducer. The available key and value provide this function. It returns the index of reducers.

Shuffling and Sorting

The data are shuffled between/within nodes so that it moves out from the map and get ready to process for reduce function. Sometimes, the shuffling of data can take much computation time.

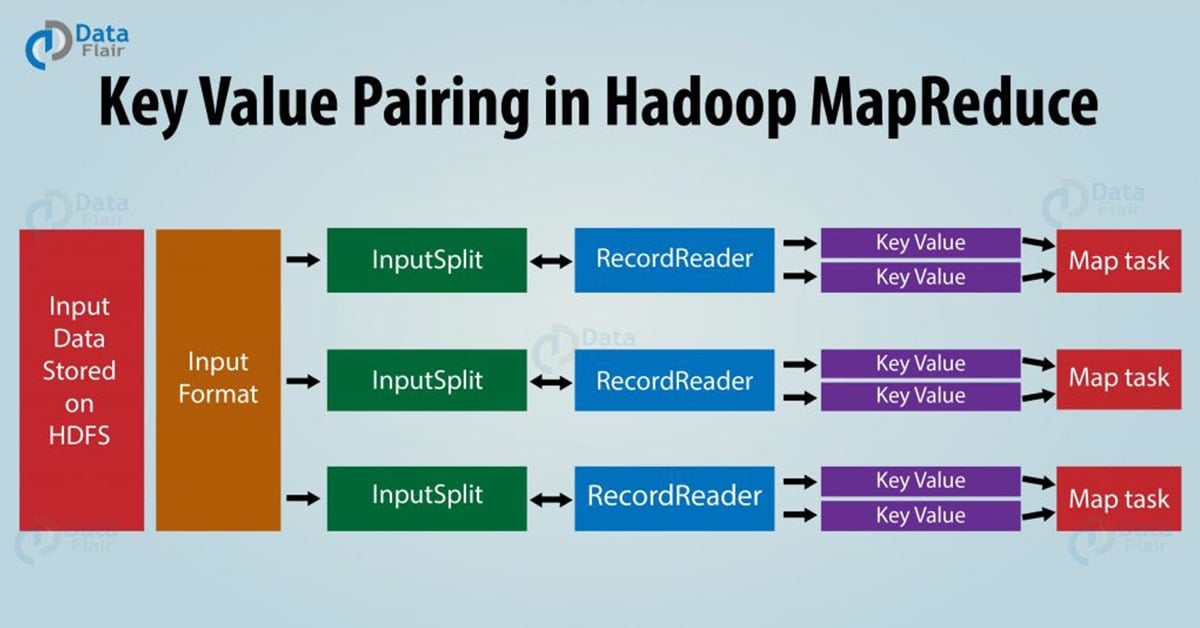
The sorting operation is performed on input data for Reduce function. Here, the data is compared using comparison function and arranged in a sorted form.

Reduce function

The Reduce function is assigned to each unique key. These keys are already arranged in sorted order. The values associated with the keys can iterate the Reduce and generates the corresponding output.

Output writer

Once the data flow from all the above phases, Output writer executes. The role of Output writer is to write the Reduce output to the stable storage.



In MapReduce process, before passing the data to the[**mapper**](https://data-flair.training/blogs/mapper-in-hadoop-mapreduce/), data should be first converted into key-value pairs as mapper only understands key-value pairs of data.  
key-value pairs in Hadoop MapReduce is generated as follows:

* **InputSplit –**It is the logical representation of data. The data to be processed by an individual Mapper is presented by the InputSplit. [Learn MapReduce InputSplit in detail](https://data-flair.training/blogs/inputsplit-in-hadoop-mapreduce/).
* **RecordReader –**It communicates with the InputSplit and it converts the Split into records which are in form of key-value pairs that are suitable for reading by the mapper. By default, RecordReader uses TextInputFormat for converting data into a key-value pa
* ir.

In MapReduce, map function processes a certain key-value pair and emits a certain number of key-value pairs and the Reduce function processes values grouped by the same key and emits another set of key-value pairs as output.

# Hadoop Combiner

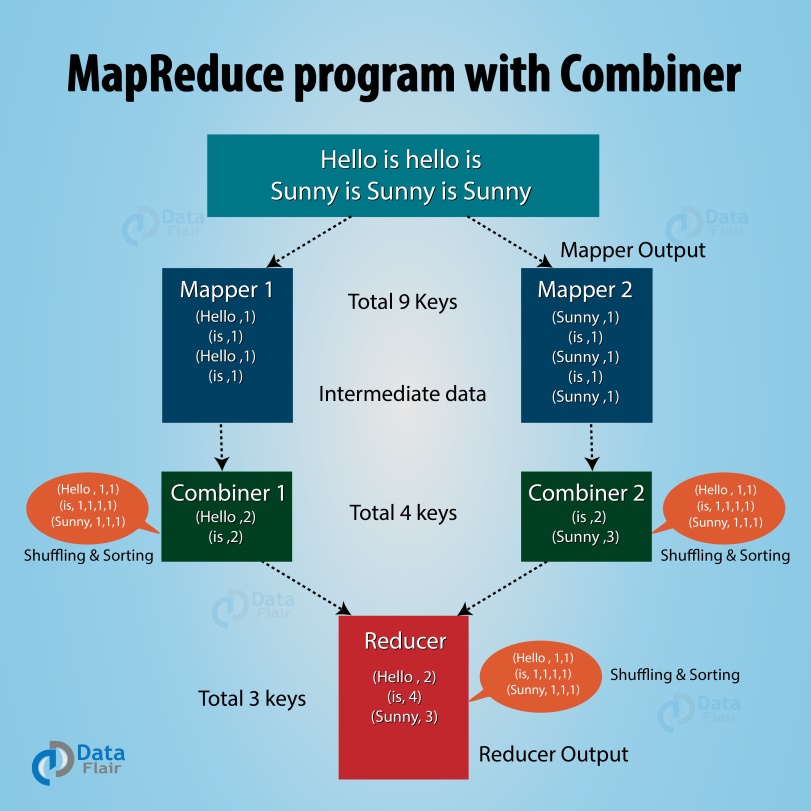
# [Hadoop](https://data-flair.training/blogs/hadoop-tutorial-for-beginners/)**Combiner** is also known as “**Mini-Reducer**” that summarizes the Mapper output record with the same Key before passing to the Reducer.

# On a large dataset when we run[**MapReduce job**](https://data-flair.training/blogs/how-hadoop-mapreduce-works/), large chunks of intermediate data is generated by the Mapper and this intermediate data is passed on the Reducer for further processing, which leads to enormous network congestion. MapReduce framework provides a function known as **Hadoop Combiner**that plays a key role in reducing network congestion.

# The primary job of Combiner is to process the output data from the Mapper, before passing it to Reducer. It runs after the mapper and before the Reducer and its use is optional.

# MapReduce Combiner : MapReduce program without combiner

In the above diagram, no combiner is used. Input is split into two mappers and 9 keys are generated from the mappers. Now we have (9 [**key/value**](https://data-flair.training/blogs/key-value-pairs-hadoop-mapreduce/)) intermediate data, the further mapper will send directly this data to reducer and while sending data to the reducer, it consumes some network bandwidth (bandwidth means time taken to transfer data between 2 machines). It will take more time to transfer data to reducer if the size of data is big.



Now in between mapper and reducer if we use a  hadoop combiner, then combiner shuffles intermediate data (9 key/value) before sending it to the reducer and generates 4 key/value pair as an output.

# Reducer now needs to process only 4 key/value pair data which is generated from 2 combiners. Thus reducer gets executed only 4 times to produce final output, which increases the overall performance.

## Advantages of MapReduce Combiner

* Hadoop Combiner reduces the time taken for data transfer between mapper and reducer.
* It decreases the amount of data that needed to be processed by the reducer.
* The Combiner improves the overall performance of the reducer.

## Disadvantages of Hadoop combiner in MapReduce

* MapReduce jobs cannot depend on the Hadoop combiner execution because there is no guarantee in its execution.
* In the local filesystem, the key-value pairs are stored in the Hadoop and run the combiner later which will cause expensive disk IO.

**SCHEDULING IN HADOOP**

Prior to Hadoop 2, **Hadoop MapReduce** is a software framework for writing applications that process huge amounts of data (terabytes to petabytes) in-parallel on the large Hadoop cluster. This framework is responsible for scheduling tasks, monitoring them, and re-executes the failed task.

In Hadoop 2, a **YARN** called Yet Another Resource Negotiator was introduced. The basic idea behind the YARN introduction is to split the functionalities of resource management and job scheduling or monitoring into separate daemons that are ResorceManager, ApplicationMaster, and NodeManager.

Resource Manager is the master daemon that arbitrates resources among all the applications in the system. NodeManager is the slave daemon responsible for containers, monitoring their resource usage, and reporting the same to ResourceManager or Schedulers. ApplicationMaster negotiates resources from the ResourceManager and works with NodeManager in order to execute and monitor the task.

The ResourceManager has two main components that are Schedulers and ApplicationsManager.

**Schedulers** in YARN ResourceManager is a pure scheduler which is responsible for allocating resources to the various running applications.

It is not responsible for monitoring or tracking the status of an application. Also, the scheduler does not guarantee about restarting the tasks that are failed either due to hardware failure or application failure.

The MapReduce system supports a minimum of one queue, named default. Hence, this parameter's value should always contain the string default. Some job schedulers, like the Capacity Scheduler, support multiple queues.

The default job scheduler is the Fair Scheduler, which is designed for a production environment with multiple users or groups that compete for cluster resources.

The MapR Converged Data Platform supports these job schedulers:

* **FIFO queue-based scheduler**: The FIFO queue scheduler runs jobs based on the order in which the jobs were submitted. You can prioritize a job by changing the value of the mapred.job.priority property or by calling the setJobPriority() method.
* **Fair Scheduler**: This is the default scheduler. The Fair Scheduler allocates a share of cluster capacity to each user over time. The design goal of the Fair Scheduler is to assign resources to jobs so that each job receives an equal share of resources over time. The Fair Scheduler enforces fair sharing within each queue. Running jobs share the queue's resources.
* **Capacity Scheduler**: The Capacity Scheduler enables users or organizations to simulate an individual hadoop cluster with FIFO scheduling for each user or organization. You can define organizations using *queues*.

The scheduler performs scheduling based on the resource requirements of the applications.

It has some pluggable policies that are responsible for partitioning the cluster resources among the various queues, applications, etc.

1. The FIFO Scheduler
2. Capacity Scheduler, and
3. Fair Scheduler

. FIFO Scheduler

**First In First Out** is the default scheduling policy used in Hadoop. FIFO Scheduler gives more preferences to the application coming first than those coming later. It places the applications in a queue and executes them in the order of their submission (first in, first out).

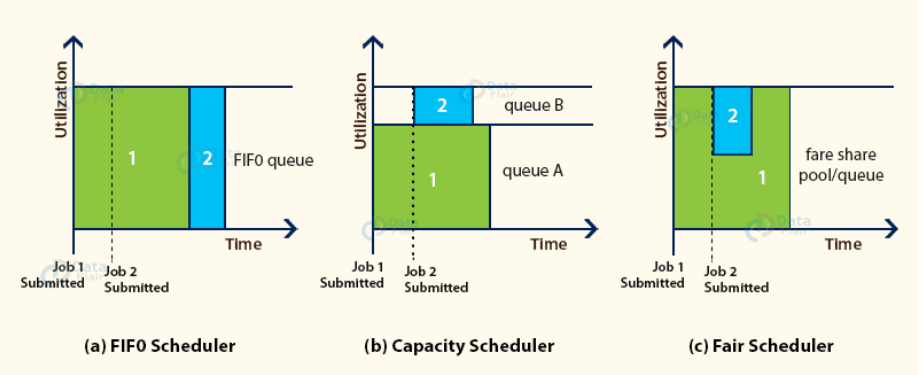
Here, irrespective of the size and priority, the request for the first application in the queue are allocated first. Once the first application request is satisfied, then only the next application in the queue is served.

**Advantage:**

* It is simple to understand and doesn’t need any configuration.
* Jobs are executed in the order of their submission.

**Disadvantage:**

* It is not suitable for shared clusters. If the large application comes before the shorter one, then the large application will use all the resources in the cluster, and the shorter application has to wait for its turn. This leads to starvation.
* It does not take into account the balance of resource allocation between the long applications and short applications.



#### 2. Capacity Scheduler

The CapacityScheduler allows multiple-tenants to securely share a large Hadoop cluster. It is designed to run [Hadoop](https://hadoop.apache.org/" \t "_blank) applications in a shared, multi-tenant cluster while maximizing the throughput and the utilization of the cluster.

It supports hierarchical queues to reflect the structure of organizations or groups that utilizes the cluster resources. A queue hierarchy contains three types of queues that are root, parent, and leaf.

The root queue represents the cluster itself, parent queue represents organization/group or sub-organization/sub-group, and the leaf accepts application submission.

The Capacity Scheduler allows the sharing of the large cluster while giving capacity guarantees to each organization by allocating a fraction of cluster resources to each queue.

Also, when there is a demand for the free resources that are available on the queue who has completed its task, by the queues running below capacity, then these resources will be assigned to the applications on queues running below capacity. This provides elasticity for the organization in a cost-effective manner.

# 3. Hadoop 2.x Fair Scheduler

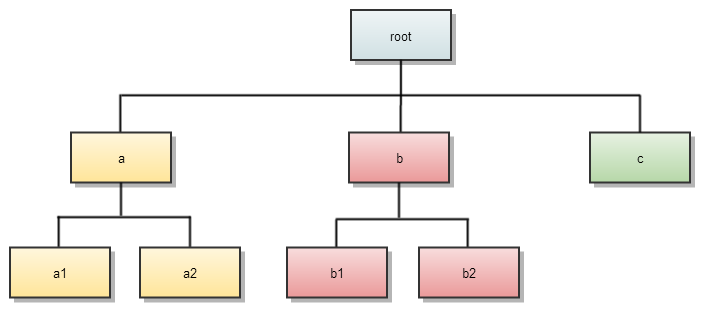
The FairScheduler is a pluggable scheduler for Hadoop that allows YARN applications to share resources in a large cluster fairly. Fair scheduling is a method of assigning resources to applications such that all applications get, on average, an equal share of resources over time. Hadoop 2.x is capable of scheduling multiple resource types.

By default, the Fair Scheduler bases scheduling fairness decisions only on memory. It can be configured to schedule resources based on memory, CPU, and disk usage. When only one application is running, that application uses the entire cluster. When other applications are submitted, resources that free up are assigned to the new applications, so that each application eventually gets approximately the same amount of resources. Unlike the default Hadoop scheduler, which forms a queue of applications, this lets short applications finish in reasonable time while not starving long-lived applications. It is also a reasonable way to share a cluster between a number of users. Finally, fair sharing also uses priorities applied as weights to determine the fraction of total resources that each application should get.

# Scheduling Queues

The scheduler organizes applications further into queues, and shares resources fairly between these queues. By default, all users share a single queue, named default. If an application specifically lists a queue in a container resource request, the request is submitted to that queue. You can also assign queues based on the user name included with the request through configuration. Within each queue, a scheduling policy is used to share resources between the running applications. The default is memory-based fair sharing, but FIFO and multi-resource with Dominant Resource Fairness can also be configured.

Queues can be arranged in a hierarchy to divide resources, and they can be configured with weights to share the cluster in specific proportions. The Fair Scheduler uses a concept called a queue path to configure a hierarchy of queues. The queue path is the full path of the queue's hierarchy, starting at root. The following example has three top-level child-queues a, b, and c and some sub-queues for a and b:



In addition to providing fair sharing, the Fair Scheduler allows assigning guaranteed minimum shares to queues, which is useful for ensuring that certain users, groups or production applications always get sufficient resources. When a queue contains apps, it gets at least its minimum share, but when the queue does not need its full guaranteed share, the excess is split between other running apps. This lets the scheduler guarantee capacity for queues while utilizing resources efficiently when these queues do not contain applications.

Understanding the Hadoop Input Output System

Unlike any I/O subsystem, Hadoop also comes with a set of primitives. These primitive considerations, although generic in nature, go with the Hadoop IO system as well with some special connotation to it, of course. Hadoop deals with multi-terabytes of datasets; a special consideration on these primitives will give an idea how Hadoop handles data input and output.

Data Integrity

Data integrity means that data should remain accurate and consistent all across its storing, processing, and retrieval operations.

To ensure that no data is lost or corrupted during persistence and processing, Hadoop maintains stringent data integrity constraints.

Every read/write operation occurs in disks, more so through the network is prone to errors.The usual way to detect corrupt data is through checksums. A checksum is computed when data first enters into the system and is sent across the channel during the retrieval process. The retrieving end computes the checksum again and matches with the received ones. If it matches exactly then the data deemed to be error free else it contains error.

But the problem is – what if the checksum sent itself is corrupt? This is highly unlikely because it is a small data, but not an undeniable possibility. Using the right kind of hardware such as ECC memory can be used to alleviate the situation.

This is mere detection. Therefore, to correct the error, another technique, called CRC (Cyclic Redundancy Check), is used.

Hadoop takes it further and creates a distinct checksum for every 512 (default) bytes of data. Because CRC-32 is 4 bytes only, the storage overhead is not an issue. All data that enters into the system is verified by the datanodes before being forwarded for storage or further processing.

Data sent to the datanode pipeline is verified through checksums and any corruption found is immediately notified to the client with ChecksumException. The client read from the datanode also goes through the same drill.

The datanodes maintain a log of checksum verification to keep track of the verified block. The log is updated by the datanode upon receiving a block verification success signal from the client. This type of statistics helps in keeping the bad disks at bay.

Apart from this, a periodic verification on the block store is made with the help of DataBlockScanner running along with the datanode thread in the background. This protects data from corruption in the physical storage media.

Hadoop maintains a copy or replicas of data. This is specifically used to recover data from massive corruption. Once the client detects an error while reading a block, it immediately reports to the datanode about the bad block from the namenode before throwing ChecksumException. The namenode then marks it as a bad block and schedules any further reference to the block to its replicas. In this w

ay, the replica is maintained with other replicas and the marked bad block is removed from the system.

For every file created in the Hadoop LocalFileSystem, a hidden file with the same name in the same directory with the extension .<filename>.crc is created. This file maintains the checksum of each chunk of data (512 bytes) in the file. The maintenance of metadata helps in detecting read error before throwing ChecksumException by the LocalFileSystem.

Compression

Keeping in mind the volume of data Hadoop deals with, compression is a requirement.

There are many obvious benefits of file compression rightly used by Hadoop. It economizes storage requirements and is a must-have capability to speed up data transmission over the network and disks. There are many tools, techniques, and algorithms commonly used by Hadoop. Many of them are quite popular and have been used in file compression over the ages. For example, gzip, bzip2, LZO, zip, and so forth are often used.

Serialization

The process that turns structured objects to stream of bytes is called serialization. This is specifically required for data transmission over the network or persisting raw data in disks. Deserialization is just the reverse process, where a stream of bytes is transformed into a structured object. This is particularly required for object implementation of the raw bytes. Therefore, it is not surprising that distributed computing uses this in a couple of distinct areas: inter-process communication and data persistence.

Hadoop uses RPC (Remote Procedure Call) to enact inter-process communication between nodes. Therefore, the RPC protocol uses the process of serialization and deserialization to render a message to the stream of bytes and vice versa and sends it across the network. However, the process must be compact enough to best use the network bandwidth, as well as fast, interoperable, and flexible to accommodate protocol updates over time.

Hadoop has its own compact and fast serialization format, Writables, that MapReduce programs use to generate keys and value types.

Data Structure of Files

There are a couple of high-level containers that elaborate the specialized data structure in Hadoop to hold special types of data.

For example, to maintain a binary log, the SequenceFile container provides the data structure to persist binary key-value pairs. We then can use the key, such as a timestamp represented by LongWritable and value by Writable, which refers to logged quantity.

There is another container, a sorted derivation of SequenceFile, called MapFile. It provides an index for convenient lookups by key.

These two containers are interoperable and can be converted to and from each other.

The Java Interface

https://timepasstechies.com/java-interface-hadoop-hdfs-filesystems-examples-concept/

In this section, we dig into the Hadoop FileSystem class: the API for interacting with one of Hadoop’s filesystems.

Reading Data Using the FileSystem API

A file in a Hadoop filesystem is represented by a Hadoop Path object. FileSystem is a general filesystem API, so the first step is to retrieve an instance for the filesystem we want to use—HDFS, in this case. There are several static factory methods for getting a FileSystem instance

public static FileSystem get(Configuration conf) throws IOException

public static FileSystem get(URI uri, Configuration conf) throws IOException

public static FileSystem get(URI uri, Configuration conf, String user) throws IOException

public static LocalFileSystem getLocal(Configuration conf) throws IOException

A Configuration object encapsulates a client or server’s configuration, which is set using configuration files read from the classpath, such as core-site.xml. The first method returns the default filesystem (as specified in core-site.xml, or the default local filesystem if not specified there). The second uses the given URI’s scheme and authority to determine the filesystem to use, falling back to the default filesystem if no scheme is specified in the given URI. The third retrieves the filesystem as the given user, which is important in the context of security.The fourth one retrieves a local filesystem instance.

With a FileSystem instance in hand, we invoke an open() method to get the input stream for a file.The first method uses a default buffer size of 4 KB.The second one gives an option to user to specify the buffer size.

public FSDataInputStream open(Path f) throws IOException

public abstract FSDataInputStream open(Path f, int bufferSize) throws IOException

FSDataInputStream

The open() method on FileSystem actually returns an FSDataInputStream rather than a standard java.io class. This class is a specialization of java.io.DataInputStream with support for random access, so you can read from any part of the stream:

**1) GZIP**

* Provides High compression ratio.
* Uses high CPU resources to compress and decompress data.
* Good choice for Cold data which is infrequently accessed.
* Compressed data is not splittable and hence not suitable for [MapReduce](https://data-flair.training/blogs/hadoop-mapreduce-introduction-tutorial-comprehensive-guide/) jobs.

**2) BZIP2**

* Provides High compression ratio (even higher than GZIP).
* Takes long time to compress and decompress data.
* Good choice for Cold data which is infrequently accessed.
* Compressed data is splittable.
* Even though the compressed data is splittable, it is generally not suited for MR jobs because of high compression/decompression time.

**3) LZO**

* Provides Low compression ratio.
* Very fast in compressing and decompressing data.
* Compressed data is splittable if an appropriate indexing algorithm is used.
* Best suited for MR jobs because of property (ii) and (iii).

**4) SNAPPY**

* Provides average compression ratio.
* Aimed at very fast compression and decompression time.
* Compressed data is not splittable if used with normal file like .txt
* Generally used to compress Container file formats like Avro and SequenceFile because the files inside a Compressed Container file can be split.

https://www.analyticsvidhya.com/blog/2020/12/15-basic-and-highly-used-hive-queries-that-all-data-engineers-must-know/