22MCA2052 -Big Data Analytics

Module 3 – Hadoop, Map-Reduce

When a dataset outgrows the storage capacity of a single physical machine, it becomes necessary to partition it across a number of separate machines. File systems that manage the storage across a network of machines are called *distributed file systems*. Since they are network-based, all the complications of network programming kick in, thus making distributed file systems more complex than regular disk file systems. For example, one of the biggest challenges is making the file system tolerate node failure without suffering data loss.

Hadoop comes with a distributed file system called HDFS, which stands for ***Hadoop Distributed File system.***

The Design of HDFS

HDFS is a file system designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware.

Very large files

"Very large" in this context means files that are hundreds of megabytes, gigabytes, or terabytes in size. There are Hadoop clusters running today that store petabytes of data.

Streaming data access

HDFS is built around the idea that the most efficient data processing pattern is a write-once, read-many-times pattern. A dataset is typically generated or copied from source, and then various analyses are performed on that dataset over time. Each analysis will involve a large proportion of the dataset, so the time to read the whole dataset is more important than the latency in reading the first record.

Commodity hardware

Hadoop doesn't require expensive, highly reliable hardware. It's designed to run on clusters of commodity hardware (commonly available hardware that can be obtained from multiple vendors) for which the chance of node failure across the cluster is high, at least for large clusters. HDFS is designed to carry on working without a noticeable interruption to the user in the face of such failure.

It is also worth examining the applications for which using HDFS does not work so well. These are areas where HDFS is not a good fit today:

Low-latency data access

Applications that require low-latency access to data, in the tens of milliseconds range, will not work well with HDFS. Remember, HDFS is optimized for delivering a high throughput of data, and this may be at the expense of latency. HBase is currently a better choice for low-latency access.

Lots of small files

Because the namenode holds file system metadata in memory, the limit to the number of files in a file system is governed by the amount of memory on the namenode. As a

rule of thumb, each file, directory, and block takes about 150 bytes. So, for example, if you had one million files, each taking one block, you would need at least 300 MB of memory.

Multiple writers, arbitrary file modifications

Files in HDFS may be written to by a single writer. Writes are always made at the end of the file. There is no support for multiple writers or for modifications at arbitrary offsets in the file.

HDFS Concepts

Blocks

A disk has a block size, which is the minimum amount of data that it can read or write. File systems for a single disk build on this by dealing with data in blocks, which are an integral multiple of the disk block size. File system blocks are typically a few kilobytes in size, whereas disk blocks are normally 512 bytes. This is generally transparent to the file system user who is simply reading or writing a file of whatever length. However, there are tools to perform file system maintenance, such as ***df*** and ***fsck*****,** that operate on the file system block level.

HDFS, too, has the concept of a block, but it is a much larger unit—64 MB by default. Like in a file system for a single disk, files in HDFS are broken into block-sized chunks, which are stored as independent units. Unlike a file system for a single disk, a file in HDFS that is smaller than a single block does not occupy a full block's worth of underlying storage.

Having a block abstraction for a distributed file system brings several benefits. The first benefit is the most obvious: a file can be larger than any single disk in the network. There's nothing that requires the blocks from a file to be stored on the same disk, so they can take advantage of any of the disks in the cluster. In fact, it would be possible to store a single file on an HDFS cluster whose blocks filled all the disks in the cluster.

Second, making the unit of abstraction a block rather than a file simplifies the storage subsystem. The storage subsystem deals with blocks, simplifying storage management (because blocks are a fixed size, it is easy to calculate how many can be stored on a given disk) and eliminating metadata concerns.

Furthermore, blocks fit well with replication for providing fault tolerance and availability. To insure against corrupted blocks and disk and machine failure, each block is replicated to a small number of physically separate machines (typically three). If a block becomes unavailable, a copy can be read from another location in a way that is transparent to the client. A block that is no longer available due to corruption or machine failure can be replicated from its alternative locations to other live machines to bring the replication factor back to the normal level. Similarly, some applications may choose to set a high replication factor for the blocks in a popular file to spread the read load on the cluster.

Like its disk file system cousin, HDFS's *fsck* command understands blocks. For example, running:

% **hadoop fsck / -files -blocks**

will list the blocks that make up each file in the file system.

Namenodes and Datanodes

An HDFS cluster has two types of nodes operating in a master-worker pattern: a *name-node* (the master) and a number of *datanodes* (workers). The namenode manages the file system namespace. It maintains the file system tree and the metadata for all the files and directories in the tree. This information is stored persistently on the local disk in the form of two files: **the namespace image and the edit log**. The namenode also knows the datanodes on which all the blocks for a given file are located; however, it does not store block locations persistently, because this information is reconstructed from datanodes when the system starts.

A *client* accesses the file system on behalf of the user by communicating with the namenode and datanodes. The client presents a filesystem interface similar to a Portable Operating System Interface (POSIX), so the user code does not need to know about the namenode and datanode to function.

Datanodes are the workhorses of the filesystem. They store and retrieve blocks when they are told to (by clients or the namenode), and they report back to the namenode periodically with lists of blocks that they are storing.

Without the namenode, the file system cannot be used. In fact, if the machine running the namenode were obliterated, all the files on the file system would be lost since there would be no way of knowing how to reconstruct the files from the blocks on the datanodes. For this reason, it is important to make the namenode resilient to failure, and Hadoop provides two mechanisms for this.

The first way is **to back up the files** that make up the persistent state of the file system metadata. Hadoop can be configured so that the namenode writes its persistent state to multiple file systems. These writes are synchronous and atomic. The usual configuration choice is to write to local disk as well as a remote NFSmount.

It is also possible **to run a *secondary namenode***, which despite its name does not act as a namenode. Its main role is to periodically merge the namespace image with the edit log to prevent the edit log from becoming too large. The secondary namenode usually runs on a separate physical machine because it requires plenty of CPU and as much memory as the namenode to perform the merge. It keeps a copy of the merged namespace image, which can be used in the event of the namenode failing. However, the state of the secondary namenode lags that of the primary, so in the event of total failure of the primary, data loss is almost certain. The usual course of action in this case is to copy the namenode's metadata files that are on NFS to the secondary and run it as the new primary.

HDFS Federation

The namenode keeps a reference to every file and block in the file system 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 namenode 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 combination of replicating namenode metadata on multiple file systems and using the secondary namenode to create checkpoints protects against data loss, but it does not provide high-availability of the file system. The namenode is still 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 file system metadata replicas and configures datanodes and clients to use this new namenode. The new namenode is not able to serve requests until it has

i) loaded its namespace image into memory,

ii) replayed its edit log, and

iii) 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.

The 2.x release series of Hadoop remedies this situation by adding support for HDFS high- availability (HA). In this implementation there is 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.

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***. Failover controllers are pluggable, but the first 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 system employs a range of fencing mechanisms, including killing the namenode's process, revoking its access to the shared storage directory, and disabling its network port via a remote management command. As a last resort, the previously active namenode can be fenced with a technique 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.

Data Flow

Anatomy of a File Read

To get an idea of how data flows between the client interacting with HDFS, the name-node, and the datanodes, consider Figure 3-2, which shows the main sequence of events when reading a file. 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 in Figure below). **DistributedFileSystem** calls the namenode, using RPC, to determine the locations of the blocks for the first few blocks in the file (step 2). 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. 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.

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.

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. Data is streamed from the datanode back to the client, which calls **read()** repeatedly on the stream (step 4). 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). This happens transparently to the client, which from its point of view is just reading a continuous stream.

Blocks are read in order, with the **DFSInputStream** opening new connections to datanodes as the client reads through the stream. It will also call the namenode to retrieve the datanode locations for the next batch of blocks as needed. When the client has finished reading, it calls **close()** on the **FSDataInputStream** (step6).

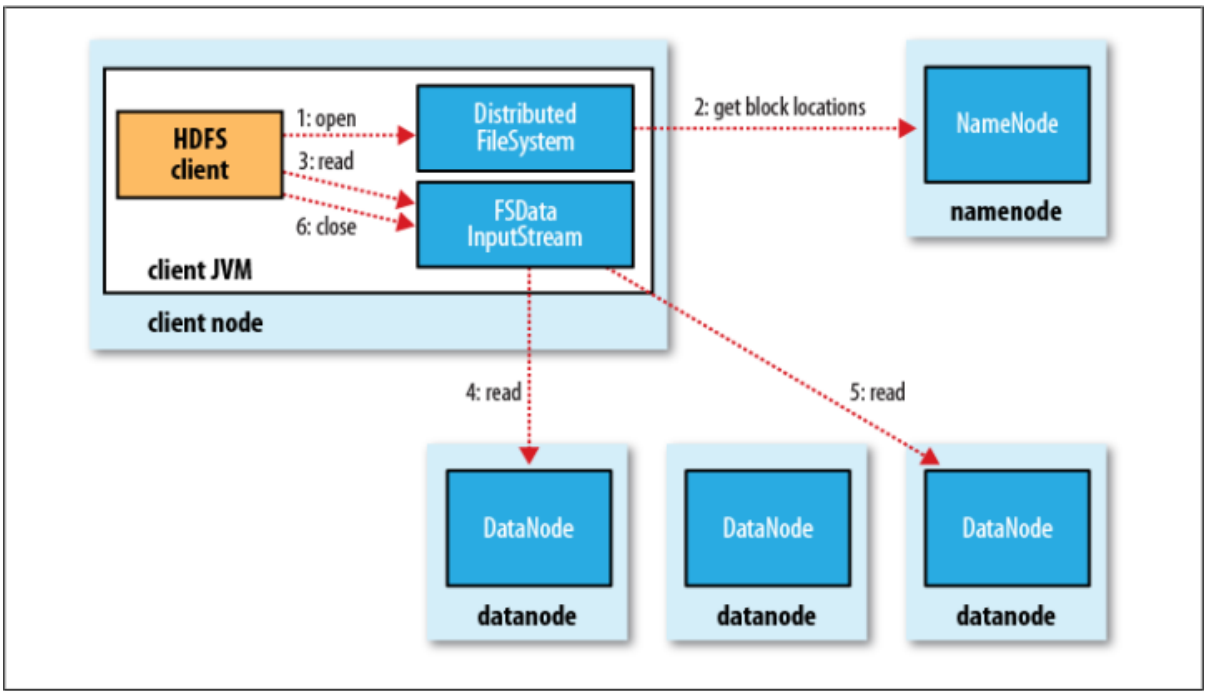


Fig: A client reading data from HDFS

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. The **DFSInputStream** also verifies checksums for the data transferred to it from the datanode. If a corrupted block is found, it is reported to the namenode before the **DFSInputStream** attempts to read a replica of the block from another datanode.

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

Next we'll look at how files are written to HDFS.

We're going to consider the case of creating a new file, writing data to it, then closing the file. See Figure below.

The client creates the file by calling **create()** on **DistributedFileSystem** (step 1 in Figure 3-4). **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). 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. If these checks pass, the namenode makes a record of the new file; otherwise, file creation fails and the client is thrown an **IOException**. 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.

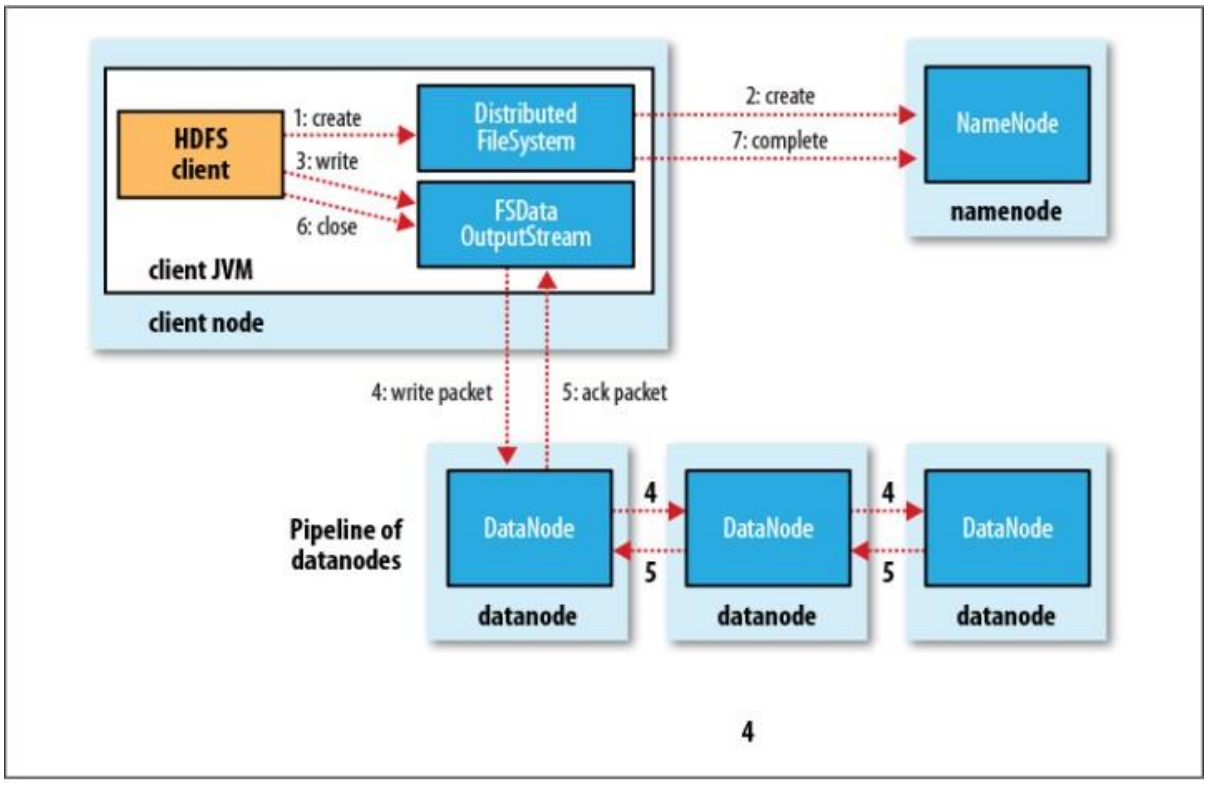


Fig: A client writing data to HDFS

As the client writes data (step 3), **DFSOutputStream** splits it into packets, which it writes to an internal queue, called the ***data queue***. The data queue is consumed by the **Data Streamer**, 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 the packet and forwards it to the second datanode in the pipeline. Similarly, the second datanode stores thepacket and forwards it to the third (and last) datanode in the pipeline (step 4).

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).

If a datanode fails while data is being written to it, then the following actions are taken, which are transparent to the client writing the data. 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. 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. The failed datanode is removed from the pipeline, and the remainder of the block's data is written to the two good datanodes in the pipeline. The namenode notices that the block is under-replicated, and it arranges for a further replica to be created on another node. Subsequent blocks are then treated as normal.

It's possible, but unlikely, that multiple datanodes fail while a block is being written. As long as **dfs.replication.min** replicas (which default to one) are written, the write will succeed, and the block will be asynchronously replicated across the cluster until its target replication factor is reached (**dfs.replication,** which defaults tothree).

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 acknowledgements 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 (via **Data Streamer** asking for block allocations), so it only has to wait for blocks to be minimally replicated before returning successfully.

Hadoop Architecture:

Hadoop is a framework written in Java that utilizes a large cluster of commodity hardware to maintain and store big size data. Hadoop works on MapReduce Programming Algorithm that was introduced by Google. Today lots of Big Brand Companies are using Hadoop in their Organization to deal with big data, eg. Facebook, Yahoo, Netflix, eBay, etc.
The Hadoop Architecture Mainly consists of 4 components.

- MapReduce
- HDFS(Hadoop Distributed File System)
- YARN(Yet Another Resource Negotiator)
- Common Utilities or Hadoop Common

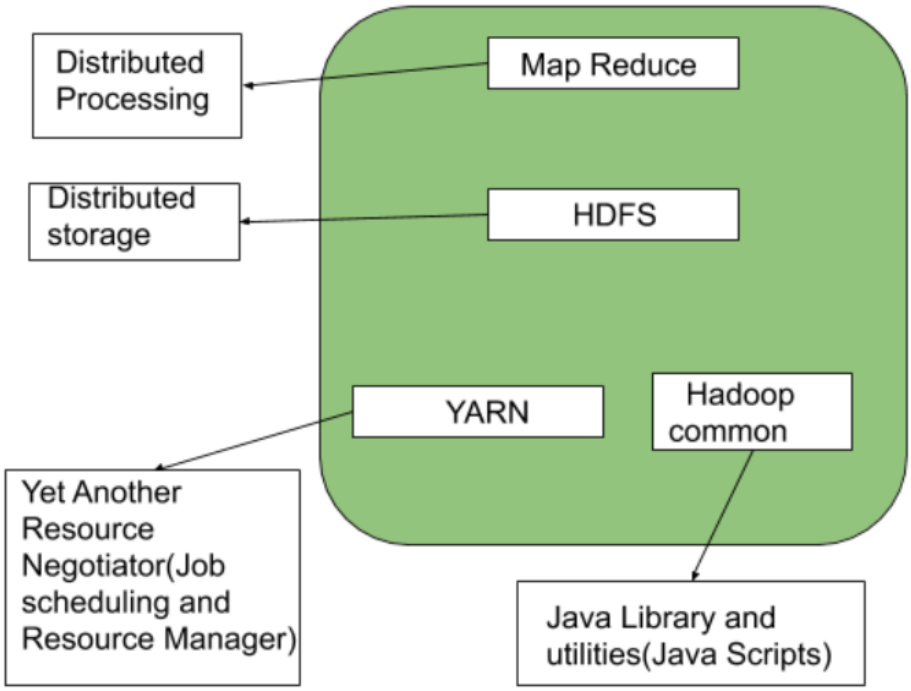
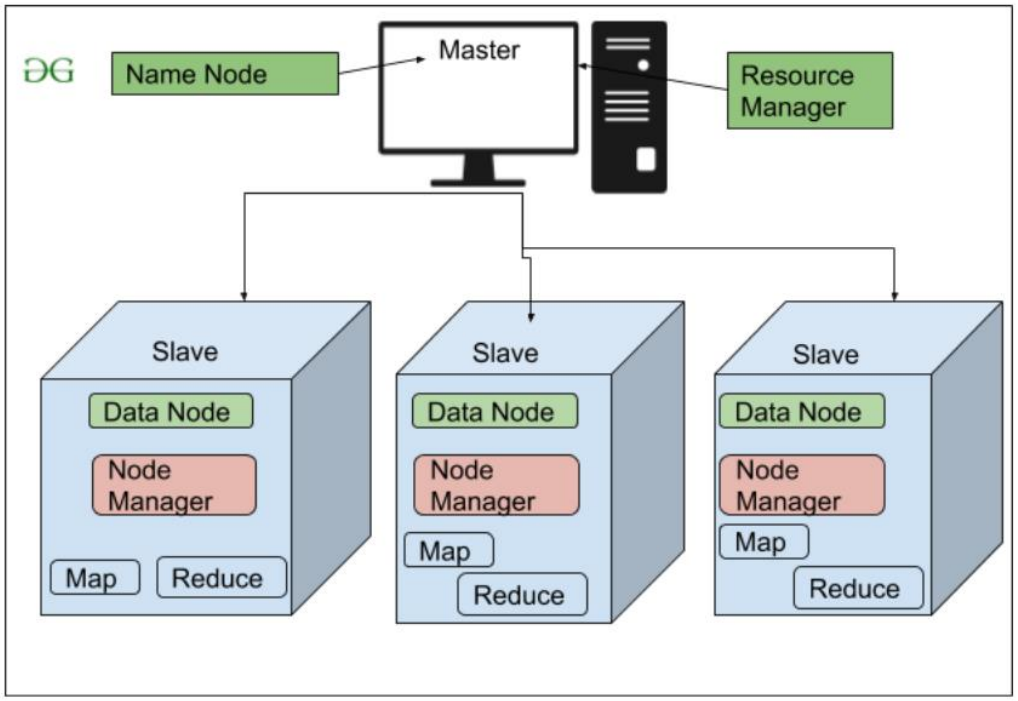


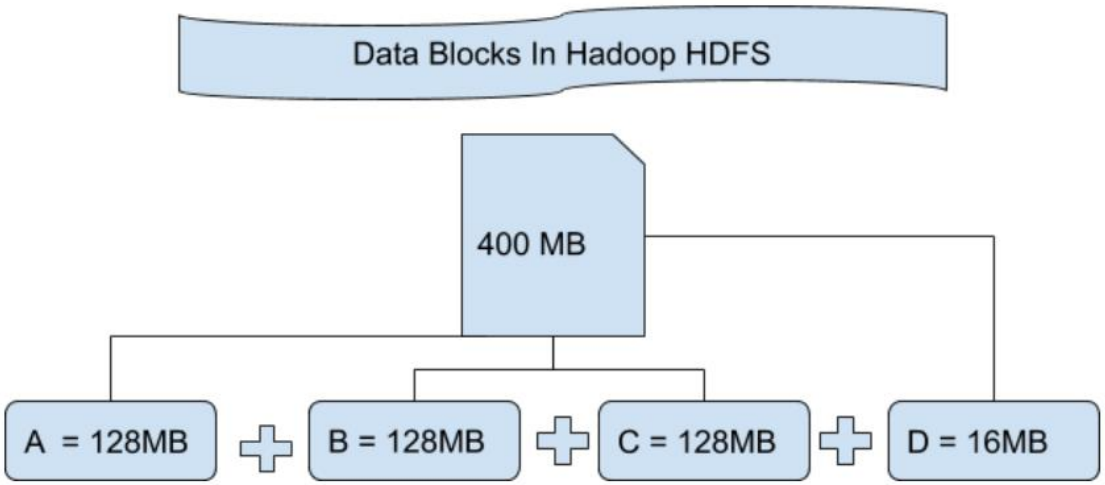
Fig: Hadoop Architecture


Name Node
Master
Resource Manager
Slave
Data Node
Node Manager
Map
Reduce
Slave
Data Node
Node Manager
Map
Reduce
Slave
Data Node
Node Manager
Map
Reduce

1. **MapReduce:** MapReduce nothing but just like an Algorithm or a data structure that is based on the YARN framework. The major feature of MapReduce is to perform the distributed processing in parallel in a Hadoop cluster which Makes Hadoop working so fast. When you are dealing with Big Data, serial processing is no more of any use. MapReduce has mainly 2 tasks which are divided phase-wise: In first phase, Map is utilized and in next phase Reduce is utilized.

2. **HDFS:** HDFS(Hadoop Distributed File System) is utilized for storage permission. It is mainly designed for working on commodity Hardware devices(inexpensive devices), working on a distributed file system design. HDFS is designed in such a way that it believes more in storing the data in a large chunk of blocks rather than storing small data blocks. HDFS in Hadoop provides Fault-tolerance and High availability to the storage layer and the other devices present in that Hadoop cluster. Data storage Nodes in HDFS are NameNode(Master) and DataNode(Slave).

   File Block In HDFS: Data in HDFS is always stored in terms of blocks. So the single block of data is divided into multiple blocks of size 128MB which is default and you can also change it manually.



3. **YARN(Yet Another Resource Negotiator):** YARN is a Framework on which MapReduce works. YARN performs 2 operations that are Job scheduling and Resource Management. The Purpose of Job schedular is to divide a big task into small jobs so that each job can be assigned to various slaves in a Hadoop cluster and Processing can be Maximized. Job Scheduler also keeps track of which job is important, which job has more priority, dependencies between the jobs and all the other information like job timing, etc. And the use of Resource Manager is to manage all the resources that are made available for running a Hadoop cluster.

4. **Hadoop common or Common Utilities:** Hadoop common or Common utilities are nothing but our java library and java files or we can say the java scripts that we need for all the other components present in a Hadoop cluster. these utilities are used by HDFS, YARN, and MapReduce for running the cluster. Hadoop Common verify that Hardware failure in a Hadoop cluster is common so it needs to be solved automatically in software by Hadoop Framework.