* Five ‘V’s associated with Big Data.

Volume – Amount of Data

Variety – Different types of data, structured, semi-structured (like logs, xml, JSON files etc), unstructured

Velocity – Speed in which data is generated, stored, analyzed and moved around.

Veracity – Quality of the data, such as trustworthiness, biases, noise and abnormality in the data.

Variability – Changing of the data, it means that the same data could have different meanings in different contexts.

Value – knowledge gained from data analysis on data.

* Difference between RDBMS and HDFS

|  |  |  |
| --- | --- | --- |
|  | RDBMS | Hadoop |
| **Data Types** | RDBMS relies on the structured data and the schema of the data is always known. | Any kind of data can be stored into Hadoop i.e. Be it structured, unstructured or semi-structured. |
| **Processing** | RDBMS provides limited or no processing capabilities. | Hadoop allows us to process the data which is distributed across the cluster in a parallel fashion. |
| **Schema on Read Vs. Write** | RDBMS is based on ‘schema on write’ where schema validation is done before loading the data. | On the contrary, Hadoop follows the schema on read policy. |
| **Read/Write Speed** | In RDBMS, reads are fast because the schema of the data is already known. | The writes are fast in HDFS because no schema validation happens during HDFS write. |
| **Cost** | Licensed software, therefore, I have to pay for the software. | Hadoop is an open source framework. So, I don’t need to pay for the software. |
| **Best Fit Use Case** | RDBMS is used for OLTP (Online Transactional Processing) system. | Hadoop is used for Data discovery, data analytics or OLAP system. |
| **Operations** | Data can be updated, deleted | It is not possible to update and delete the data. |

* Hadoop is framework that allows us to store and process large datasets in parallel and distributed manner.
* HDFS follows Write Once – Read Many Philosophy. So, you can’t edit files already stored in HDFS. But, you can append new data by re-opening the file.
* Advantage of HDFS:
* **Distributed Storage** – it means you can store single large file across multiple machines in cluster
* **Distributed & Parallel processing** – Data is distributed across multiple machines, all machines process the data same time to speed up turnaround time.
* Horizontal Scaling or Scale out - **horizontal scaling** or **scaling out**in Hadoop. There are two types of scaling: **vertical** and**horizontal**. In vertical scaling (scale up), you increase the hardware capacity of your system. In other words, you procure more RAM or CPU and add it to your existing system to make it more robust and powerful. But there are challenges associated with vertical scaling or scaling up:
  + There is always a limit to which you can increase your hardware capacity. So, you can’t keep on increasing the RAM or CPU of the machine.
  + In vertical scaling, you stop your machine first. Then you increase the RAM or CPU to make it a more robust hardware stack. After you have increased your hardware capacity, you restart the machine. This down time when you are stopping your system becomes a challenge.

In case of **horizontal scaling (scale out)**, you add more nodes to existing cluster instead of increasing the hardware capacity of individual machines. And most importantly, you can add more machines on the go i.e. Without stopping the system. Therefore, while scaling out we don’t have any down time or green zone, nothing of such sort. At the end of the day, you will have more machines working in parallel to meet your requirements.

* **Reliability and Fault Tolerance:** When you store data on HDFS, it internally divides the given data into data blocks and stores it in a distributed fashion across your Hadoop cluster. The information regarding which data block is located on which of the data nodes is recorded in the NameNode metadata. NameNode manages the metadata and the DataNodes are responsible for storing the data.  
   Name node also replicates the data i.e. maintains multiple copies of the data. This replication of the data makes HDFS very reliable and fault tolerant. So, even if any of the nodes fails, we can retrieve the data from the replicas residing on other data nodes. By default, the replication factor is 3. Therefore, if you store 1 GB of file in HDFS, it will finally occupy 3 GB of space. The name node periodically updates the metadata and maintains the replication factor consistent.
* **Data Integrity:** Data Integrity talks about whether the data stored in my HDFS are correct or not. HDFS constantly checks the integrity of data stored against its checksum. If it finds any fault, it reports to the name node about it. Then, the name node creates additional new replicas and therefore deletes the corrupted copies.
* **Data Locality:**Data locality talks about moving processing unit to data rather than the data to processing unit. In our traditional system, we used to bring the data to the application layer and then process it. But now, because of the architecture and huge volume of the data, bringing the data to the application layer will reduce the network performance to a noticeable extent. So, in HDFS, we bring the computation part to the data nodes where the data is residing. Hence, you are not moving the data, you are bringing the program or processing part to the data.

**Open Source:** One can modify it’s code to business requirements

Hadoop runs on a cluster of commodity hardware which is **not very expensive**.

Hadoop is very **easy to use**, as there is no need of client to deal with distributed computing; the framework takes care of all the things.

**Limitations of Hadoop:**

**Issue with small files:** HDFS was designed to work properly with a small number of large files for storing large data sets rather than a large number of small files. If there are too many small files, then the **NameNode** will be overloaded since it stores the namespace of HDFS.

Solution:

* Solution to this Drawback of Hadoop to deal with small file issue is simple. Just merge the small files to create bigger files and then copy bigger files to HDFS.
* **HAR files** (Hadoop Archives) were introduced to reduce the problem of lots files putting pressure on the namenode’s memory. By building a layered filesystem on the top of HDFS, HAR files works. Using Hadoop archive command, HAR files are created, which runs a [MapReduce](http://data-flair.training/blogs/hadoop-mapreduce-introduction-tutorial-comprehensive-guide/) job to pack the files being archived into a small number of HDFS files. Reading through files in a HAR is not more efficient than reading through files in HDFS. Since each HAR file access requires two index files read as well the data file to read, this makes it slower.
* **Sequence files**work very well in practice to overcome the ‘small file problem’, in which we use the filename as the key and the file contents as the value. By writing a program for files (100 KB), we can put them into a single Sequence file and then we can process them in a streaming fashion operating on the Sequence file. MapReduce can break Sequence file into chunks and operate on each chunk independently because Sequence file is splitable.
* Storing files in **[HBase](http://data-flair.training/blogs/hbase-tutorial-beginners-guide/)**is a very common design pattern to overcome small file problem with HDFS. We are not actually storing millions of small files into HBase, rather adding the binary content of the file to a cell

**Slow Processing:** In Hadoop, with a parallel and distributed algorithm, MapReduce process large data sets. There are tasks that need to be performed: [**Map**](http://data-flair.training/blogs/mapper-in-hadoop-mapreduce/) and [**Reduce**](http://data-flair.training/blogs/reducer-in-hadoop-mapreduce/)and, MapReduce requires a lot of time to perform these tasks thereby increasing latency. Data is distributed and processed over the cluster in MapReduce which increases the time and reduces processing speed.

Solution:

As a Solution to this Limitation of Hadoop spark has overcome this issue, by in-memory processing of data. [**In-memory processing**](http://data-flair.training/blogs/apache-spark-in-memory-computing/) is faster as no time is spent in moving the data/processes in and out of the disk. Spark is 100 times faster than MapReduce as it processes everything in memory. Flink is also used, as it processes faster than spark because of its streaming architecture and Flink may be instructed to process only the parts of the data that have actually changed, thus significantly increases the performance of the job

**Batch Processing only:** Hadoop supports batch processing only, it does not process streamed data. Apache Hadoop is designed for batch processing, that means it take a huge amount of data in input, process it and produce the result. Although batch processing is very efficient for processing a high volume of data, but depending on the size of the data being processed and computational power of the system, an output can be delayed significantly. Hadoop is not suitable for Real-time data processing.

Solution:

* **Apache Spark** supports stream processing. Stream processing involves continuous input and output of data. It emphasizes on the velocity of the data, and data is processed within a small period of time.
* **Apache Flink** provides single run-time for the streaming as well as batch processing, so one common run-time is utilized for data streaming application and batch processing application. Flink is a stream processing system that is able to process row after row in real time
* **HDFS** – Hadoop Distributed File System is core component of Hadoop ecosystem, HDFS is java based distributed file system that allows you to store large datasets across multiple nodes in hadoop cluster. HDFS follows master-slave architecture.
* HDFS has two components – NameNode and DataNode.

**Safemode:**

On startup, the NameNode enters a special state called Safemode. Replication of data blocks does not occur when the NameNode is in the Safemode state. The NameNode receives Heartbeat and Blockreport messages from the DataNodes. A Blockreport contains the list of data blocks that a DataNode is hosting. Each block has a specified minimum number of replicas. A block is considered safely replicated when the minimum number of replicas of that data block has checked in with the NameNode. After a configurable percentage of safely replicated data blocks checks in with the NameNode (plus an additional 30 seconds), the NameNode exits the Safemode state. It then determines the list of data blocks (if any) that still have fewer than the specified number of replicas. The NameNode then replicates these blocks to other DataNodes.

**Communication Protocol:**

All HDFS communication protocols are layered on top of the TCP/IP protocol. A client establishes a connection to a configurable TCP port on the NameNode machine. It talks the ClientProtocol with the NameNode. The DataNodes talk to the NameNode using the DataNode Protocol. A Remote Procedure Call (RPC) abstraction wraps both the Client Protocol and the DataNode Protocol. By design, the NameNode never initiates any RPCs. Instead, it only responds to RPC requests issued by DataNodes or clients.

**File Delete and Undeletes:**

If trash configuration is enabled, files removed by [FS Shell](https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/FileSystemShell.html#rm) is not immediately removed from HDFS. Instead, HDFS moves it to a trash directory (each user has its own trash directory under /user/<username>/.Trash). The file can be restored quickly as long as it remains in trash.

Most recent deleted files are moved to the current trash directory (/user/<username>/.Trash/Current), and in a configurable interval, HDFS creates checkpoints (under /user/<username>/.Trash/<date>) for files in current trash directory and deletes old checkpoints when they are expired. See [expunge command of FS shell](https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/FileSystemShell.html#expunge) about checkpointing of trash.

After the expiry of its life in trash, the NameNode deletes the file from the HDFS namespace. The deletion of a file causes the blocks associated with the file to be freed. Note that there could be an appreciable time delay between the time a file is deleted by a user and the time of the corresponding increase in free space in HDFS.

Following is an example which will show how the files are deleted from HDFS by FS Shell. We created 2 files (test1 & test2) under the directory delete

|  |
| --- |
| $ hadoop fs -mkdir -p delete/test1  $ hadoop fs -mkdir -p delete/test2  $ hadoop fs -ls delete/  Found 2 items  drwxr-xr-x - hadoop hadoop 0 2015-05-08 12:39 delete/test1  drwxr-xr-x - hadoop hadoop 0 2015-05-08 12:40 delete/test2 |

We are going to remove the file test1. The comment below shows that the file has been moved to Trash directory.

|  |
| --- |
| $ hadoop fs -rm -r delete/test1  Moved: hdfs://localhost:8020/user/hadoop/delete/test1 to trash at: hdfs://localhost:8020/user/hadoop/.Trash/Current |

now we are going to remove the file with skipTrash option, which will not send the file to Trash.It will be completely removed from HDFS.

|  |
| --- |
| $ hadoop fs -rm -r -skipTrash delete/test2  Deleted delete/test2 |

We can see now that the Trash directory contains only file test1.

|  |
| --- |
| $ hadoop fs -ls .Trash/Current/user/hadoop/delete/  Found 1 items\  drwxr-xr-x - hadoop hadoop 0 2015-05-08 12:39 .Trash/Current/user/hadoop/delete/test1 |

So file test1 goes to Trash and file test2 is deleted permanently.

* The NameNode and Datanodes have built in web servers that makes it easy to check current status of the cluster
* **NameNode:** (MasterNode)
* It is the master daemon that maintains and manages the DataNodes (slave nodes)
* It records the metadata of all the files stored in the cluster, e.g. The location of blocks stored, the size of the files, permissions, hierarchy, etc. There are two files associated with the metadata:
  + **FsImage:** It contains the complete state of the file system namespace since the start of the NameNode.
  + **EditLogs:** It contains all the recent modifications made to the file system with respect to the most recent FsImage.
* It records each change that takes place to the file system metadata. For example, if a file is deleted in HDFS, the NameNode will immediately record this in the EditLog.
* It regularly receives a Heartbeat and a block report from all the DataNodes in the cluster to ensure that the DataNodes are live.
* It keeps a record of all the blocks in HDFS and in which nodes these blocks are located.
* The NameNode is also responsible to take care of the **replication factor**of all the blocks which we will discuss in detail later in this HDFS tutorial blog.
* In **case of the DataNode failure**, the NameNode chooses new DataNodes for new replicas, balance disk usage and manages the communication traffic to the DataNodes.
* NameNode is highly available server that manages the File System Namespace and controls access to files by clients.
* **DataNode:** (SlaveNode)
* These are slave daemons or process which runs on each slave machine.
* The actual data is stored on DataNodes.
* The DataNodes perform the low-level read and write requests from the file system’s clients.
* They send heartbeats to the NameNode periodically to report the overall health of HDFS, by default, this frequency is set to 3 seconds.
* **Secondary NameNode:**

The NameNode stores modifications to the file system as a log appended to a native file system file, edits. When a NameNode starts up, it reads HDFS state from an image file, fsimage, and then applies edits from the edits log file. It then writes new HDFS state to the fsimage and starts normal operation with an empty edits file. Since NameNode merges fsimage and edits files only during start up, the edits log file could get very large over time on a busy cluster. Another side effect of a larger edits file is that next restart of NameNode takes longer.

The secondary NameNode merges the fsimage and the edits log files periodically and keeps edits log size within a limit. It is usually run on a different machine than the primary NameNode since its memory requirements are on the same order as the primary NameNode.

The start of the checkpoint process on the secondary NameNode is controlled by two configuration parameters.

* *dfs.namenode.checkpoint.period*, set to 1 hour by default, specifies the maximum delay between two consecutive checkpoints, and
* *dfs.namenode.checkpoint.txns*, set to 1 million by default, defines the number of uncheckpointed transactions on the NameNode which will force an urgent checkpoint, even if the checkpoint period has not been reached.

The secondary NameNode stores the latest checkpoint in a directory which is structured the same way as the primary NameNode’s directory. So that the check pointed image is always ready to be read by the primary NameNode if necessary.

* Default block size is 128MB in apache Hadoop 2.x
* By default, the heartbeat interval is 3 seconds, which can be configured using dfs.heartbeat.interval in hdfs-site.xml
* SNN will do image copy every 60 min.
* By default, heartbeat interval b/w Resource Manager and Name Manager is 1 sec.

**Balancer**

HDFS data might not always be be placed uniformly across the DataNode. One common reason is addition of new DataNodes to an existing cluster. While placing new blocks (data for a file is stored as a series of blocks), NameNode considers various parameters before choosing the DataNodes to receive these blocks. Some of the considerations are:

* Policy to keep one of the replicas of a block on the same node as the node that is writing the block.
* Need to spread different replicas of a block across the racks so that cluster can survive loss of whole rack.
* One of the replicas is usually placed on the same rack as the node writing to the file so that cross-rack network I/O is reduced.
* Spread HDFS data uniformly across the DataNodes in the cluster.

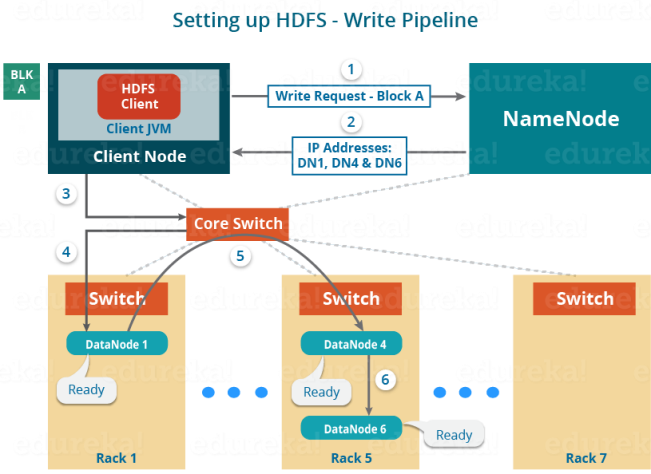
Due to multiple competing considerations, data might not be uniformly placed across the DataNodes. HDFS provides a tool for administrators that analyzes block placement and rebalanaces data across the DataNode

**Safemode**

During start up the NameNode loads the file system state from the fsimage and the edits log file. It then waits for DataNodes to report their blocks so that it does not prematurely start replicating the blocks though enough replicas already exist in the cluster. During this time NameNode stays in Safemode. Safemode for the NameNode is essentially a read-only mode for the HDFS cluster, where it does not allow any modifications to file system or blocks. Normally the NameNode leaves Safemode automatically after the DataNodes have reported that most file system blocks are available. If required, HDFS could be placed in Safemode explicitly using bin/hdfs dfsadmin -safemode command. NameNode front page shows whether Safemode is on or off

* **Rack Awareness:**
* NameNode also ensures that all the replicas are not stored on the same rack or a single rack. It follows an in-built Rack Awareness Algorithm to reduce latency as well as provide fault tolerance. Considering the replication factor is 3, the Rack Awareness Algorithm says that the first replica of a block will be stored on a local rack and the next two replicas will be stored on a different (remote) rack but, on a different DataNode within that (remote) rack. If you have more replicas, the rest of the replicas will be placed on random DataNodes provided not more than two replicas reside on the same rack, if possible.
* **Advantages of Rack Awareness:**
* **It improves network performance**, In general, you will find greater network bandwidth between machines in the same rack than the machines residing in different rack.
* **To prevent loss of data**, we don’t have to worry about the data even if an entire rack fails because of the switch failure or power failure.
* **HDFS Write Architecture:**
* HDFS client wants to write a file size of 248 MB, block size is 128MB, file will be split into two blocks – 128MB (Block - A) + 120MB (Block - B)
* Following protocol will be followed whenever the data is written into HDFS:
* At first, the HDFS client will reach out to the NameNode for a Write Request against the two blocks, say, Block A & Block B.
* The NameNode will then grant the client the write permission and will provide the IP addresses of the DataNodes where the file blocks will be copied eventually.
* The selection of IP addresses of DataNodes is purely randomized based on availability, replication factor and rack awareness that we have discussed earlier.
* Let’s say the replication factor is set to default i.e. 3. Therefore, for each block the NameNode will be providing the client a list of (3) IP addresses of DataNodes. The list will be unique for each block.
* Suppose, the NameNode provided following lists of IP addresses to the client:
  + For Block A, list A = {IP of DataNode 1, IP of DataNode 4, IP of DataNode 6}
  + For Block B, set B = {IP of DataNode 3, IP of DataNode 7, IP of DataNode 9}
* Each block will be copied in three different DataNodes to maintain the replication factor consistent throughout the cluster.
* Now the whole data copy process will happen in three stages:
  + 1. Set up of Pipeline
  + 2. Data streaming and replication
  + 3. Shutdown of Pipeline (Acknowledgement stage)
* **Set up of Pipeline:**
* Before writing the blocks, the client confirms whether the DataNodes, present in each of the list of IPs, are ready to receive the data or not. In doing so, the client creates a pipeline for each of the blocks by connecting the individual DataNodes in the respective list for that block. Let us consider Block A. The list of DataNodes provided by the NameNode is: {IP of DataNode 1, IP of DataNode 4, IP of DataNode 6}

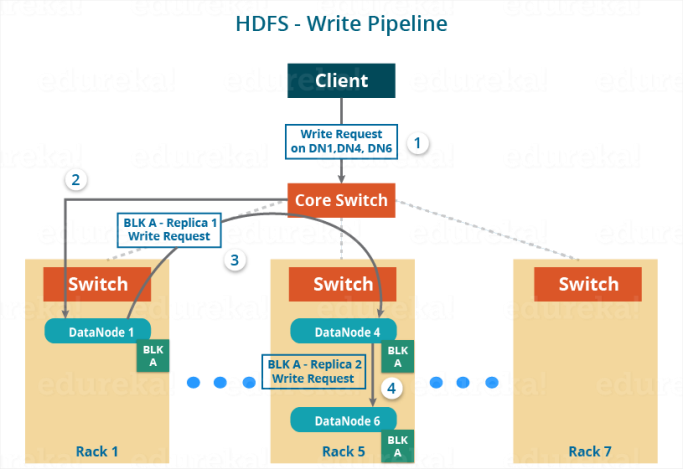
So, for block A, the client will be performing the following steps to create a pipeline:



* The client will choose the first DataNode in the list (DataNode IPs for Block A) which is DataNode 1 and will establish a TCP/IP connection.
* The client will inform DataNode 1 to be ready to receive the block. It will also provide the IPs of next two DataNodes (4 and 6) to the DataNode 1 where the block is supposed to be replicated.
* The DataNode 1 will connect to DataNode 4. The DataNode 1 will inform DataNode 4 to be ready to receive the block and will give it the IP of DataNode 6. Then, DataNode 4 will tell DataNode 6 to be ready for receiving the data.
* Next, the acknowledgement of readiness will follow the reverse sequence, i.e. From the DataNode 6 to 4 and then to 1.
* At last DataNode 1 will inform the client that all the DataNodes are ready and a pipeline will be formed between the client, DataNode 1, 4 and 6.
* Now pipeline set up is complete and the client will finally begin the data copy or streaming process.

**Data Streaming:**

As the pipeline has been created, the client will push the data into the pipeline. Now, don’t forget that in HDFS, data is replicated based on replication factor. So, here Block A will be stored to three DataNodes as the assumed replication factor is 3. Moving ahead, the client will copy the block (A) to DataNode 1 only. The replication is always done by DataNodes sequentially



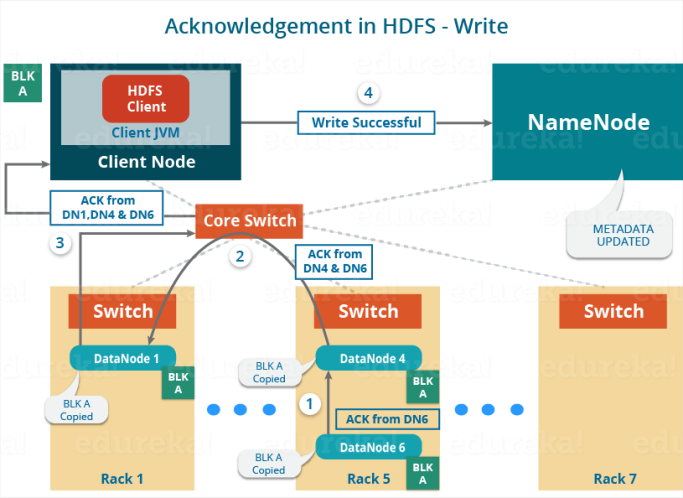
So, the following steps will take place during replication:

* Once the block has been written to DataNode 1 by the client, DataNode 1 will connect to DataNode 4.
* Then, DataNode 1 will push the block in the pipeline and data will be copied to DataNode 4.
* Again, DataNode 4 will connect to DataNode 6 and will copy the last replica of the block.

**Shutdown of Pipeline or Acknowledgement stage:**

Once the block has been copied into all the three DataNodes, a series of acknowledgements will take place to ensure the client and NameNode that the data has been written successfully. Then, the client will finally close the pipeline to end the TCP session.

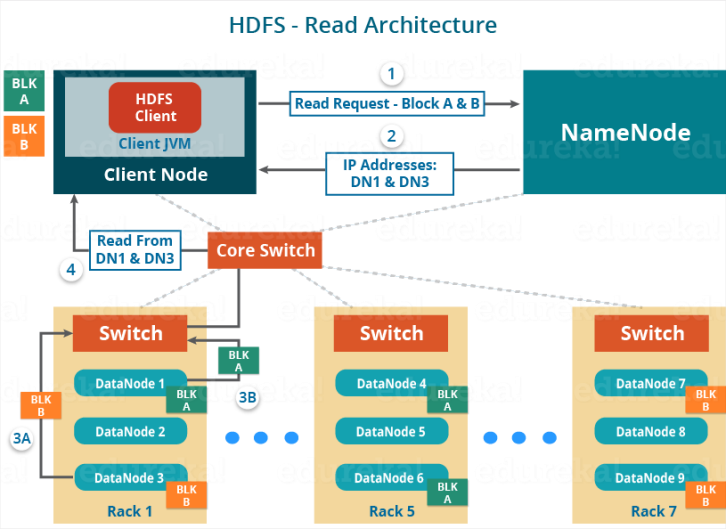
As shown in the figure below, the acknowledgement happens in the reverse sequence i.e. from DataNode 6 to 4 and then to 1. Finally, the DataNode 1 will push three acknowledgements (including its own) into the pipeline and send it to the client. The client will inform NameNode that data has been written successfully. The NameNode will update its metadata and the client will shut down the pipeline.



Similarly, Block B will also be copied into the DataNodes in parallel with Block A. So, the following things are to be noticed here:

* The client will copy Block A and Block B to the first DataNode **simultaneously**.
* Therefore, in our case, two pipelines will be formed for each of the block and all the process discussed above will happen in parallel in these two pipelines.
* The client writes the block into the first DataNode and then the DataNodes will be replicating the block sequentially.

**HDFS Read Architecture:**



Now, following steps will be taking place while reading the file:

* The client will reach out to NameNode asking for the block metadata for the file “example.txt”.
* The NameNode will return the list of DataNodes where each block (Block A and B) are stored.
* After that client, will connect to the DataNodes where the blocks are stored.
* The client starts reading data parallel from the DataNodes (Block A from DataNode 1 and Block B from DataNode 3).
* Once the client gets all the required file blocks, it will combine these blocks to form a file.

While serving read request of the client, HDFS selects the replica which is closest to the client. This reduces the read latency and the bandwidth consumption. Therefore, that replica is selected which resides on the same rack as the reader node, if possible.

**High Availability:**

The concept of High Availability cluster was introduced in Hadoop 2.x to solve the single point of failure problem in Hadoop 1.x. If you consider the standard configuration of HDFS cluster, the NameNode becomes a **single point of failure**. It happens because the moment the NameNode becomes unavailable, the whole cluster becomes unavailable until someone restarts the NameNode or brings a new one.

The reasons for unavailability of NameNode can be:

* A planned event like maintenance work such has upgradation of software or hardware.
* It may also be due to an unplanned event where the NameNode crashes because of some reasons.

In either of the above cases, we have a downtime where we are not able to use the HDFS cluster which becomes a challenge.

Let us understand that how HDFS HA Architecture solved this critical problem of NameNode availability:

The HA architecture solved this problem of NameNode availability by allowing us to have two NameNodes in an active/passive configuration. So, we have two running NameNodes at the same time in a High Availability cluster:

* Active NameNode
* Standby/Passive NameNode.

If one NameNode goes down, the other NameNode can take over the responsibility and therefore, reduce the cluster down time. The standby NameNode serves the purpose of a backup NameNode (unlike the Secondary NameNode) which incorporate failover capabilities to the Hadoop cluster. Therefore, with the StandbyNode, we can have automatic failover whenever a NameNode crashes (unplanned event) or we can have a graceful (manually initiated) failover during the maintenance period.

There are two issues in maintaining consistency in the HDFS High Availability cluster:

* Active and Standby NameNode should always be in sync with each other, i.e. They should have the same metadata. This will allow us to restore the Hadoop cluster to the same namespace state where it got crashed and therefore, will provide us to have fast failover.
* There should be only one active NameNode at a time because two active NameNode will lead to corruption of the data. *This kind of scenario is termed as a split-brain scenario where a cluster gets divided into smaller cluster, each one believing that it is the only active cluster.* To avoid such scenarios fencing is done. Fencing is a process of ensuring that only one NameNode remains active at a particular time.

**Hadoop 1.x** MapReduce execution framework:

Job Tracker is a Master daemon, responsible to assign and track task execution progress

Task Trackers are slave daemons, they run on systems where data nodes reside and responsible to spawn a child jvm to execute Map, Reduce, and intermediate tasks.

**YARN – Yet Another Resource Negotiator**

*YARN* comprises of two major component: **ResourceManager** and **NodeManager**.

YARN also works for non MapReduce frameworks

#### ****ResourceManager****

* It is a cluster level (one for each cluster) component and runs on the master machine
* It manages resources and schedule applications running on top of YARN
* It has two components: Scheduler & ApplicationManager
* The Scheduler is responsible for allocating resources to the various running applications, and does not offers guarantee about restarting failed task. Scheduler has pluggable policy like CapacityScheduler and FairScheduler.
* The ApplicationManager is responsible for accepting job submissions and negotiating the first container (Yarn Child) for executing the application and also provides the service for restarting the ApplicationMaster container on failure.
* It keeps a track of the heartbeats from the Node Manager
* Container is nothing but jvm, a piece of main memory and CPU

#### ****NodeManager****

* It is a node level component (one on each node) and runs on each slave machine
* It is responsible for managing containers and monitoring resource utilization in each container
* It also keeps track of node health and log management
* It continuously communicates with ResourceManager to remain up-to-date

**Application Manager:**

* Acts as helper process for ResourceManager
* Initializes the job and track of the job’s progress
* Retrieves the input splits computed by client
* Negotiates the resources needed for running a job with the ResourceManager
* Create a map task object for each split.

Limitations of current HDFS architecture:

* *Namespace layer* and*storage layer* are tightly coupled. It makes alternate implementation of namenode difficult. And it restricts other services to use block storage directly.
* The namespace is not scalable like datanode. Scaling in HDFS cluster is horizontally by adding datanodes. But we can’t add more namespace to an existing cluster. We can scale namespace vertically on a single namenode.
* Hadoop entire performance depends on the throughput of the namenode. An operation of current file system depends on the throughput of a single namenode. NameNode at present supports 60,000 concurrent tasks. Upcoming **[MapReduce](http://data-flair.training/blogs/hadoop-mapreduce-introduction-tutorial-comprehensive-guide/)** will have support for more than 1,00,000 concurrent tasks. And this will need more namenode.
* There is no separation of the namespace. So there is no isolation among tenant organization that is using the cluster.

HDFS Federation architecture:

Federation in Hadoop uses multiple independent Namenode/namespaces to scale the name service horizontally. In HDFS Federation Architecture, at the bottom, datanodes are present. And datanodes are used as common storage for blocks by all the namenodes. Each datanodes registers with all the namenodes in the cluster. These datanodes send periodic heartbeats, block report and handle command from the namenodes.

Benefits of HDFS Federation architecture: Federation Hadoop overcomes the limitations of prior HDFS architecture.

Isolation: In HDFS federation different categories of application and users can be isolated to different namespaces by using many namenodes.

NameSpace Scalability: In federation many namenodes horizontally scales up in the filesystem namespace.

Performance: We can improve Read/write operation throughput by adding more namenodes.

**MapReduce:**

**MapReduce** is the processing layer of **Hadoop**. MapReduce programming model is designed for processing large volumes of data in parallel by dividing the work into a set of independent tasks. MapReduce is a programming framework that allows us to perform distributed and parallel processing on large data sets in a distributed environment.

* MapReduce consists of two distinct tasks – Map and Reduce.
* As the name MapReduce suggests, reducer phase takes place after mapper phase has been completed.
* So, the first is the map job, where a block of data is read and processed to produce key-value pairs as intermediate outputs.
* The output of a Mapper or map job (key-value pairs) is input to the Reducer.
* The reducer receives the key-value pair from multiple map jobs.
* Then, the reducer aggregates those intermediate data tuples (intermediate key-value pair) into a smaller set of tuples or key-value pairs which is the final output.

The two biggest advantages of MapReduce are:

**Parallel Processing:** In MapReduce, we are dividing the job among multiple nodes and each node works with a part of the job simultaneously. So, MapReduce is based on Divide and Conquer paradigm which helps us to process the data using different machines. As the data is processed by multiple machine instead of a single machine in parallel, the time taken to process the data gets reduced by a tremendous amount

**Data Locality:**Instead of moving data to the processing unit, we are moving processing unit to the data in the MapReduce Framework.  In the traditional system, we used to bring data to the processing unit and process it. But, as the data grew and became very huge, bringing this huge amount of data to the processing unit posed following issues:

* Moving huge data to processing is costly and deteriorates the network performance.
* Processing takes time as the data is processed by a single unit which becomes the bottleneck.
* Master node can get over-burdened and may fail.

Now, MapReduce allows us to overcome above issues by bringing the processing unit to the data. So, as you can see in the above image that the data is distributed among multiple nodes where each node processes the part of the data residing on it. This allows us to have the following advantages:

* It is very cost effective to move processing unit to the data.
* The processing time is reduced as all the nodes are working with their part of the data in parallel.
* Every node gets a part of the data to process and therefore, there is no chance of a node getting overburdened.

**Process of Spilling:**

* The output of map task is written into a circular memory buffer (RAM)
* Default buffer size is set to 100 MB as specified in the ***mapreduce.task.io.sort.mb***
* Spilling is the process of copying the data from memory buffer to disc after a certain threshold is reached
* Default spilling threshold is 0.8 as specified in ***mapreduce.map.sort.spill.percent***

Main hadoop configuration files:

Hadoop-env.sh – Environment details like java home, hadoop home

Core-site.xml – NameNode should run,

Hdfs-site.xml – Configuration details like replication factor, NameNode physically present

Yarn-site.xml

Mapred-site.xml

Masters – SNN resides

Slaves – DataNodes

* How to copy a file into HDFS with a different block size to that of existing block size configuration

Configuration size = 128MB, want to copy with block size = 32MB. First convert into bytes.

32MB = 33554432 bytes

Command: ***hadoop fs –Ddfs.blocksize=33554432 –copyFromLocal /local/test.txt /sample\_hdfs/text.txt***

Check the block size of file:

***Hadoop fs –stat %o /sample\_hdfs/text.txt***

FSDataInputStream:

Hadoop Load balancer: ensures data blocks are replicated as per configuration factor.

Input Split: Logical chunks of data to be processed by an individual mapper

Input Split computational formula???

**RecordReader**: It works as interface between Inputsplit and Mapper. It communicates with InputSplit and it converts the data present in a file into (Key, Value) pairs suitable for reading by the Mapper task, RecordReader instance is defined by InputFormat. Default it uses TextInputFormat for converting the data into the key-value pair. RecordReader’s responsibility is to keep reading/converting data into key-value pairs until the end of the file. Byte offset (unique number) is assigned to each line present in the file by RecordReader. Further, this key-value pair is sent to the mapper. The output of the mapper program is called as intermediate data (key-value pairs which are understandable to reduce). RecordReader reads the records in the InputSplits and sends key-value pairs to mapper, if InputSplit has 5 records, it will send 5 key-value pairs to Mapper and mapper will execute 5 times.

For the first Inputsplit, start of the split is 0, so RecordReader starts reading the record from offset 0, for the subsequent InputSplits start of the split is non-zero, so corresponding RecordReader ignores the record until end of the line character, then starts reading the records until length of the split, this ensures no records is read twice and no record is missed.

**InputSplit**: InputSplits converts the physical representation of the block into logical for the Hadoop mapper. By default number of InputSplits equals to number of mappers. InputSplit does not contain any data, it’s just the file path + start of the split + length of the split + address of the node where block is present. InputSpits do not always depend on the number of blocks, we can customize the number of splits for a particular file by setting ***mapred.max.split.size*** property during job execution. Need to very careful if split size is set manually, assume split size is set as 256MB, HDFS block size is 128MB, means split size is equal to 2 blocks, while reading the records, 2nd block should be brought to the node where first block is located as RecordReaderis launched in the node where first block is located, data transfer is costly in Hadoop, that’s why default split size equal to block size.

**No. of Mapper= {(total data size)/ (input split size)}**

For example, if data size is 1 TB and InputSplit size is 100 MB then,

No. of Mapper= (1000\*1000)/100= 10,000

With the help of *Job.setNumreduceTasks(int)* the user set the number of reducers for the job. The right number of reducers are 0.95 or 1.75 multiplied by (<no. of nodes> \* <no. of the maximum container per node>).

With 0.95, all reducers immediately launch and start transferring map outputs as the maps finish. With 1.75, the first round of reducers is finished by the faster nodes and second wave of reducers is launched doing a much better job of load balancing.

Increasing the number of MapReduce Reducers:

* Increases the Framework overhead.
* Increases load balancing.
* Lowers the cost of failures.

InputFormat class calls the **getSplits()** function and computes splits for each file and then sends them to the **JobTracker**, which uses their storage locations to schedule map tasks to process them on the **TaskTrackers**. Map task then passes the split to the **createRecordReader()** method on InputFormat in task tracker to obtain a RecordReader for that split. The RecordReader load’s data from its source and converts into key-value pairs suitable for reading by the**[mapper.](http://data-flair.training/blogs/mapper-in-hadoop-mapreduce/)**

Hadoop RecordReader uses the data within the boundaries that are being created by the inputsplit and creates Key-value pairs for the mapper.

**Significance of counters in MapReduce:** Used to gathering about the job for quality control, for application level statistics

* Easier to retrieve counters as compared to log messages for large distributed job

Ex: counting the number of invalid records

**Speculative Execution:**

* If a task is detected to be running slow, an equivalent task is launched so as to maintain the critical path of jobs.
* Scheduler tracks the progress of all the tasks (map and reduce) and launches speculative duplicates for slower tasks
* After completion of a task, all running duplicate tasks are killed.

How will you prevent a file from splitting in case you want the whole file to be processed by the same mapper?

* Increase the minimum split size to be larger than the largest file inside the driver section 🡪 conf.set(“mapred.min.split.size”,”size-larger-than-file-size”)
* Modify the InputFormat class that you want to use, override isSplitable () method to return false.

Reducer task will be set to zero if aggregation is not required

**Uber Mode:**

If a job is small, Application Master chooses to run the tasks in it’s own JVM and are called uber mode

It reduces the overhead of allocating new containers for running the tasks

A MapReduce job is decided as uber task if:

* It requires less than 10 mappers
* It requires only one reducer
* The input size is less than HDFS block size

Parameters to be set for deciding uber task:

* ***Mapreduce.job.ubertask.maxmaps***
* ***Mapreduce.job.ubertask.maxreduces***
* ***Mapreduce.job.ubertask.maxbytes***

To enable uber task: ***Mapreduce.job.ubertask.enable to true.***

How will you enhance the performance of MapReduce job when dealing with too many small files?

* CombineFileInputFormat can be used to solve this problem
* CombineFileInputFormat packs all the small files into input splits where each split is processed by a single mapper
* Takes node and rack locality into account when deciding which blocks to place in the same split
* Can process the input files efficiently in a typical MapReduce job

**Task Attempt** is a particular instance of an attempt to execute a task on a node. There is a possibility that anytime any machine can go down. For example, while processing data if any node goes down, framework reschedules the task to some other node. This rescheduling of the task cannot be infinite. There is an upper limit for that as well. The default value of task attempt is 4. If a task (Mapper or reducer) fails 4 times, then the job is considered as a failed job. For high priority job or huge job, the value of this task attempt can also be increased.

**InputFormat:** Using InputFormat we define how these input files are split and read. The InputFormat class is one of the fundamental classes in the Hadoop MapReduce framework which provides the following functionality:

* The files or other objects that should be used for input is selected by the InputFormat.
* InputFormat defines the Data splits, which defines both the size of individual [Map tasks](http://data-flair.training/blogs/mapper-in-hadoop-mapreduce/)and its potential execution server.
* InputFormat defines the [RecordReader](http://data-flair.training/blogs/recordreader-in-hadoop-mapreduce/), which is responsible for reading actual records from the input files.

**TextInputFormat:**

It is the default InputFormat of MapReduce. TextInputFormat treats each line of each input file as a separate record and performs no parsing. This is useful for unformatted data or line-based records like log files.

* **Key –**It is the byte offset of the beginning of the line within the file (not whole file just one split), so it will be unique if combined with the file name.
* **Value –**It is the contents of the line, excluding line terminators.

**KeyValueTextInputFormat:**

It is similar to TextInputFormat as it also treats each line of input as a separate record. While TextInputFormat treats entire line as the value, but the KeyValueTextInputFormat breaks the line itself into key and value by a tab character (‘/t’). Here Key is everything up to the tab character while the value is the remaining part of the line after tab character

**SequenceFileInputFormat:**

Hadoop SequenceFileInputFormat is an InputFormat which reads sequence files. Sequence files are binary files that stores sequences of binary[key-value pairs](http://data-flair.training/blogs/key-value-pairs-hadoop-mapreduce/). Sequence files are block-compressed and provide direct serialization and deserialization of several arbitrary data types (not just text). Here Key & Value both are user-defined.

**SequenceFileAsTextInputFormat:**

Hadoop **SequenceFileAsTextInputFormat** is another form of SequenceFileInputFormat which converts the sequence file key values to Text objects. By calling **‘tostring()’** conversion is performed on the keys and values. This InputFormat makes sequence files suitable input for streaming.

**Partitioner:**

The Default Hadoop partitioner in Hadoop MapReduce is Hash Partitioner which computes a hash value for the key and assigns the partition based on this result. The total number of Partitioners that run in Hadoop is equal to the number of reducers i.e. Partitioner will divide the data according to the number of reducers which is set by *JobConf.setNumReduceTasks()* method. Thus, the data from single partitioner is processed by a single reducer. And partitioner is created only when there are multiple reducers.

**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.

On what circumstances we need to change the block size?