**Hadoop**

* Hadoop is an Apache open source framework written in java that allows distributed processing of large datasets across clusters of computers using simple programming models.
* The Hadoop framework application works in an environment that provides distributed **storage** and **computation** across clusters of computers.
* Hadoop is designed to scale up from single server to thousands of machines, each offering local computation and storage.

**Hadoop Architecture**

At its core, Hadoop has two major layers namely:

* Processing/Computation layer (MapReduce), and
* Storage layer (Hadoop Distributed File System).



**Advantagesof Hadoop**

* Hadoop framework allows the user to quickly write and test distributed systems. It is efficient, and it automatic distributes the data and work across the machines and in turn, utilizes the underlying parallelism of the CPU cores.
* Hadoop does not rely on hardware to provide fault-tolerance and high availability (FTHA), rather Hadoop library itself has been designed to detect and handle failures at the application layer.
* Servers can be added or removed from the cluster dynamically and Hadoop continues to operate without interruption.
* Another big advantage of Hadoop is that apart from being open source, it is compatible on all the platforms since it is Java based.

**Analyzing data with Hadoop**

With rapid innovations, frequent evolutions of technologies and a rapidly growing internet population, systems and enterprises are generating huge amounts of data to the tune of terabytes and even petabytes of information. Since data is being generated in very huge volumes with great velocity in all multi-structured formats like images, videos, weblogs, sensor data, etc. from all different sources, there is a huge demand to efficiently store, process and analyze this large amount of data to make it usable.

Hadoop is undoubtedly the preferred choice for such a requirement due to its key characteristics of being reliable, flexible, economical, and a scalable solution. While Hadoop provides the ability to store this large scale data on HDFS (Hadoop Distributed File System), there are multiple solutions available in the market for analyzing this huge data like MapReduce, Pig and Hive.

**Scale Out a Hadoop Cluster**

You specify the number of nodes in the cluster when you create Hadoop clusters. You can later scale out the cluster by increasing the number of worker nodes and client nodes.

You can scale the cluster using the vSphere Web Client or the Serengeti Command-Line Interface Client. The command-line interface provides more configuration options than the vSphere Web Client.

You cannot decrease the number of worker and client nodes from the Data Director for Hadoop Web console.

**Prerequisites**

* Start the cluster if it is not running.

**Procedure**

|  |  |
| --- | --- |
| 1 | Log in to the vSphere Web Client. |
| 2 | Select **Big Data Extensions**. |
| 3 | From the Inventory Lists, click **Big Data Clusters**. |
| 4 | Select the cluster that you want to scale out from the Hadoop Cluster Name column. |
| 5 | Click the **All Actions** icon, and select **Scale Out**. |
| 6 | Select the worker or client node group to scale out from the **Node group** drop-down menu. |
| 7 | Specify the target number of node instances to add in the Instance number text box and click **OK**.  Because you cannot decrease the number of nodes, a Scale Out Failed error occurs if you specify an instance number that is less than or equal to the current number of instances. |

The cluster is updated to include the specified number of nodes.

**Hadoop Streaming**

Hadoop Streaming is a utility that comes with the Hadoop distribution. The utility allows you to create and run Map/Reduce jobs with any executable or script as the mapper and/or the reducer.

For example:

**$HADOOP\_HOME/bin/hadoop jar $HADOOP\_HOME/hadoop-streaming.jar \**

**-input myInputDirs \**

**-output myOutputDir \**

**-mapper /bin/cat \**

**-reducer /bin/wc**

**How Streaming Works**

In the above example, both the mapper and the reducer are executables that read the input from stdin (line by line) and emit the output to stdout. The utility will create a Map/Reduce job, submit the job to an appropriate cluster, and monitor the progress of the job until it completes.

When an executable is specified for mappers, each mapper task will launch the executable as a separate process when the mapper is initialized. As the mapper task runs, it converts its inputs into lines and feed the lines to the stdin of the process. In the meantime, the mapper collects the line oriented outputs from the stdout of the process and converts each line into a key/value pair, which is collected as the output of the mapper. By default, the prefix of a line up to the first tab character is the key and the rest of the line (excluding the tab character) will be the value. If there is no tab character in the line, then entire line is considered as key and the value is null. However, this can be customized, as discussed later.

When an executable is specified for reducers, each reducer task will launch the executable as a separate process then the reducer is initialized. As the reducer task runs, it converts its input key/values pairs into lines and feeds the lines to the stdin of the process. In the meantime, the reducer collects the line oriented outputs from the stdout of the process, converts each line into a key/value pair, which is collected as the output of the reducer. By default, the prefix of a line up to the first tab character is the key and the rest of the line (excluding the tab character) is the value. However, this can be customized, as discussed later.

This is the basis for the communication protocol between the Map/Reduce framework and the streaming mapper/reducer.

You can supply a Java class as the mapper and/or the reducer. The above example is equivalent to:

**$HADOOP\_HOME/bin/hadoop jar $HADOOP\_HOME/hadoop-streaming.jar \**

**-input myInputDirs \**

**-output myOutputDir \**

-**mapper org.apache.hadoop.mapred.lib.IdentityMapper \**

**-reducer /bin/wc**

User can specify stream.non.zero.exit.is.failure as true or false to make a streaming task that exits with a non-zero status to be Failure or Success respectively. By default, streaming tasks exiting with non-zero status are considered to be failed tasks.

**Hadoop File System (HDFS)**

Hadoop File System was developed using distributed file system design. It is run on commodity hardware. Unlike other distributed systems, HDFS is highly fault-tolerant and designed using low-cost hardware.

HDFS holds very large amount of data and provides easier access. To store such huge data, the files are stored across multiple machines. These files are stored in redundant fashion to rescue the system from possible data losses in case of failure. HDFS also makes applications available to parallel processing.

**Features of HDFS**

* It is suitable for the distributed storage and processing.
* Hadoop provides a command interface to interact with HDFS.
* The built-in servers of namenode and datanode help users to easily check the status of cluster.
* Streaming access to file system data.
* HDFS provides file permissions and authentication

**HDFS Architecture**

Given below is the architecture of a Hadoop File System.



HDFS follows the master-slave architecture and it has the following elements.

**Namenode**

The namenode is the commodity hardware that contains the GNU/Linux operating system and the namenode software. It is a software that can be run on commodity hardware. The system having the namenode acts as the master server and it does the following tasks:

Manages the file system namespace.

Regulates client’s access to files.

It also executes file system operations such as renaming, closing, and opening files and directories.

**Datanode**

The datanode is a commodity hardware having the GNU/Linux operating system and datanode software. For every node (Commodity hardware/System) in a cluster, there will be a datanode. These nodes manage the data storage of their system.

* Datanodes perform read-write operations on the file systems, as per client request.
* They also perform operations such as block creation, deletion, and replication according to the instructions of the namenode.

**Block**

Generally the user data is stored in the files of HDFS. The file in a file system will be divided into one or more segments and/or stored in individual data nodes. These file segments are called as blocks. In other words, the minimum amount of data that HDFS can read or write is called a *Block.* The default block size is 64MB, but it can be increased as per the need to change in HDFS configuration.

**Goals of HDFS**

**Fault detection and recovery:** Since HDFS includes a large number of commodity hardware, failure of components is frequent. Therefore HDFS should have mechanisms for quick and automatic fault detection and recovery.

**Huge datasets:** HDFS should have hundreds of nodes per cluster to manage the applications having huge datasets.

**Hardware at data:** A requested task can be done efficiently, when the computation takes place near the data. Especially where huge datasets are involved, it reduces the network traffic and increases the throughput.

**HDFS OPERATIONS**

* **Starting HDFS**

Initially you have to format the configured HDFS file system, open namenode (HDFS server), and execute the following command.

$ hadoopnamenode -format

After formatting the HDFS, start the distributed file system. The following command will start the namenode as well as the data nodes as cluster.

**$ start-dfs.sh**

* **Listing Files in HDFS**

After loading the information in the server, we can find the list of files in a directory, status of a file, using ‘**ls**’. Given below is the syntax of **ls** that you can pass to a directory or a filename as an argument.

**$ $HADOOP\_HOME/bin/hadoop fs -ls <args>**

* **Inserting Data into HDFS**

Assume we have data in the file called file.txt in the local system which is ought to be saved in the hdfs file system. Follow the steps given below to insert the required file in the Hadoop file system.

**Step 1**

You have to create an input directory.

**$ $HADOOP\_HOME/bin/hadoop fs -mkdir /user/input**

**Step 2**

Transfer and store a data file from local systems to the Hadoop file system using the put command.

**$ $HADOOP\_HOME/bin/hadoop fs -put /home/file.txt /user/input**

**Step 3**

You can verify the file using ls command.

**$ $HADOOP\_HOME/bin/hadoop fs -ls /user/input**

* **Retrieving Data from HDFS**

Assume we have a file in HDFS called **outfile**. Given below is a simple demonstration for retrieving the required file from the Hadoop file system.

**Step 1**

Initially, view the data from HDFS using **cat** command.

**$ $HADOOP\_HOME/bin/hadoop fs -cat /user/output/outfile**

**Step 2**

Get the file from HDFS to the local file system using **get** command.

**$ $HADOOP\_HOME/bin/hadoop fs -get /user/output/ /home/hadoop\_tp/**

* **Shutting Down the HDFS**

You can shut down the HDFS by using the following command.

**$ stop-dfs.sh**

**Hadoop I/O**

* Hadoop Comes with a set of primitives for data I/O.
* Some of these are techniques that are more general than Hadoop, such as data integrity and compression, but deserve special consideration when dealing with multiterabyte datasets.
* Others are Hadoop tools or APIs that form the building blocks for developing distributed system, such as serialization frameworks and on-disk data structures.

**Data Integrity**

* Since every I/O operation on the disk or network carries with it a small chance of introducing errors into the data that it is reading or writing.
* When the volumes of data flowing through the system are as large as the ones Hadoop is capable of handling, the chance of data corruption occurring is high
* The usual way of detecting corrupted data is by computing a checksum for the data.
* This technique doesn’t offer any way to fix the data, just only error detection
* Note that it is possible that it’s the checksum that is corrupt, not the data, but this is very unlikely, since the checksum is much smaller than the data.
* A commonly used error-detecting code is CRC-32, which computes a 32-bit integer checksum for input of any size.

**Data Integrity in HDFS**

* Since HDFS stores replica of blocks, it can “heal” corrupted blocks by copying one of the good replicas to produce a new, uncorrupt replica.
* If a client detects an error when reading a block
  1. It reports the bad block and datanode it was trying to read from to the namenode before throwing a ChecksumException.
  2. Thenamenode marks the block replica as corrupt, so it doesn’t direct clients to it, or try to copy this replica to another datanode.
  3. It then schedules a copy of the block to be replicated on another datanode, so its replication factor is back at the expected level.
  4. Once this has happened, the corrupt replica is deleted.
* It is possible to disable verification of checksums by passing false to the setVerifyChecksum() method on FileSystem, before using the open() method to read a file.
* The same effect is possible from the shell by using the –ignoreCrc option with the –get or the equivalent –copyToLocal command

[**Data Compression in Hadoop**](http://comphadoop.weebly.com/)

File compression brings two major benefits: it reduces the space needed to store files, and it speeds up data transfer across the network or to or from disk. When dealing with large volumes of data, both of these savings can be significant, so it pays to carefully consider how to use compression in Hadoop.

1. **What to compress?**
2. **Compressing input files**

If the input file is compressed, then the bytes read in from HDFS is reduced, which means less time to read data. This time conservation is beneficial to the performance of job execution.  
  
If the input files are compressed, they will be decompressed automatically as they are read by MapReduce, using the filename extension to determine which codec to use. For example, a file ending in .gz can be identified as gzip-compressed file and thus read with GzipCodec.  
  
**2) Compressing output files**

Often we need to store the output as history files. If the amount of output per day is extensive, and we often need to store history results for future use, then these accumulated results will take extensive amount of HDFS space. However, these history files may not be used very frequently, resulting in a waste of HDFS space. Therefore, it is necessary to compress the output before storing on HDFS.   
  
**3) Compressing map output**

Even if your MapReduce application reads and writes uncompressed data, it may benefit from compressing the intermediate output of the map phase. Since the map output is written to disk and transferred across the network to the reducer nodes, by using a fast compressor such as LZO or Snappy, you can get performance gains simply because the volume of data to transfer is reduced.

**Serialization**

**What is Serialization?**

Serialization is the process of translating data structures or objects state into binary or textual form to transport the data over network or to store on some persistent storage. Once the data is transported over network or retrieved from the persistent storage, it needs to be deserialized again. Serialization is termed as **marshalling** and deserialization is termed as **Unmarshalling**

**Serialization in Hadoop**

Generally in distributed systems like Hadoop, the concept of serialization is used for **Interprocess Communication** and **Persistent Storage**.

**Interprocess Communication**

To establish the interprocess communication between the nodes connected in a network, RPC technique was used.

* RPC used internal serialization to convert the message into binary format before sending it to the remote node via network. At the other end the remote system deserializes the binary stream into the original message.

The RPC serialization format is required to be as follows −

* **Compact** − To make the best use of network bandwidth, which is the most scarce resource in a data center.
* **Fast** − Since the communication between the nodes is crucial in distributed systems, the serialization and deserialization process should be quick, producing less overhead.
* **Extensible** − Protocols change over time to meet new requirements, so it should be straightforward to evolve the protocol in a controlled manner for clients and servers.
* **Interoperable** − The message format should support the nodes that are written in different languages.

**Advantage of Hadoop over Java Serialization**

Hadoop’s Writable-based serialization is capable of reducing the object-creation overhead by reusing the Writable objects, which is not possible with the Java’s native serialization framework.

**Disadvantages of Hadoop Serialization**

To serialize Hadoop data, there are two ways

* You can use the **Writable** classes, provided by Hadoop’s native library.
* You can also use **Sequence Files** which store the data in binary format.

The main drawback of these two mechanisms is that Writables and Sequence Files have only a Java API and they cannot be written or read in any other language.

Therefore any of the files created in Hadoop with above two mechanisms cannot be read by any other third language, which makes Hadoop as a limited box. To address this drawback, Doug Cutting created **Avro,** which is a **language independent data structure**.

**Avro**

**What is Avro?**

Apache Avro is a language-neutral data serialization system. It was developed by Doug Cutting, the father of Hadoop. Since Hadoop writable classes lack language portability, Avro has become quite helpful, as it deals with data formats that can be processed by multiple languages. Avro is a preferred tool to serialize data in Hadoop.

Avro has a schema-based system. A language-independent schema is associated with its read and write operations. Avro serializes the data which has a built-in schema. Avro serializes the data into a compact binary format, which can be de-serialized by any application.

Avro uses JSON format to declare the data structures. Presently, it supports languages such as Java, C, C++, C#, Python, and Ruby.

**Features of Avro**

Listed below are some of the prominent features of Avro

* Avro is a **language-neutral** data serialization system.
* It can be processed by many languages (currently C, C++, C#, Java, Python, and Ruby).
* Avro creates binary structured format that is both **compressible** and **splittable**. Hence it can be efficiently used as the input to Hadoop MapReduce jobs.
* Avro provides **rich data structures**. For example, you can create a record that contains an array, an enumerated type, and a sub record. These datatypes can be created in any language, can be processed in Hadoop, and the results can be fed to a third language.
* Avro **schemas** defined in **JSON**, facilitate implementation in the languages that already have JSON libraries.
* Avro creates a self-describing file named *Avro Data File,* in which it stores data along with its schema in the metadata section.
* Avro is also used in Remote Procedure Calls (RPCs). During RPC, client and server exchange schemas in the connection handshake.

**How to use Avro?**

To use Avro, you need to follow the given workflow

* **Step 1** − Create schemas. Here you need to design Avro schema according to your data.
* **Step 2** − Read the schemas into your program. It is done in two ways –
* **By Generating a Class Corresponding to Schema** − Compile the schema using Avro. This generates a class file corresponding to the schema
* **By Using Parsers Library** − You can directly read the schema using parsers library.
* **Step 3** − Serialize the data using the serialization API provided for Avro, which is found in the **package org.apache.avro.specific**.
* **Step 4** – De-serialize the data using deserialization API provided for Avro, which is found in the **package org.apache.avro.specific**