Understanding Hadoop Ecosystem

Problems with big data

- Storage:storing huge and exponentially growing dataset
- Processing data having complex structure

(structured unstructured and semi structure)

- Bringing huge amount of data to computation becomes bottleneck
- Hadoop is the solution to these problems

What is Hadoop?

Hadoop is a **Framework** that allows for distributed processing of large data sets across clusters using a simple programming model

It is a open source Data management with scale out storage and distributed processing

History of Hadoop

"Map-reduce"
2004

Reads paper

Doug Cutting

"It is an important technique!"

Joins Yahoo! at 2006

Apache Nutch

Extended

The great journey begins…

2005: Doug Cutting and Michael J. Cafarella developed Hadoop to support distribution for the Nutch search engine project.

The project was funded by Yahoo.

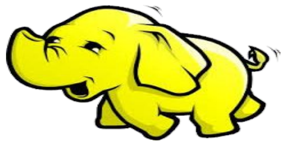
2006: Yahoo gave the project to Apache Software Foundation.

HADOOP

- Apache top level project, open-source implementation of frameworks for reliable, scalable, distributed computing and data storage.
- It is a flexible and highly-available architecture for large scale computation and data processing on a network of commodity hardware.

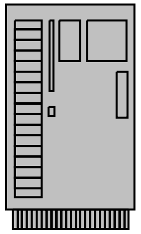
Goals

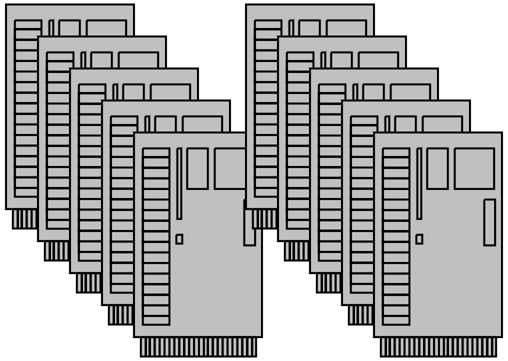
- High scalability and availability
- Use commodity (cheap!) hardware with little redundancy
- Fault-tolerance
- Move computation rather than data



Why Distributed processing?

Suppose there is a situation of Reading 1 TB of data



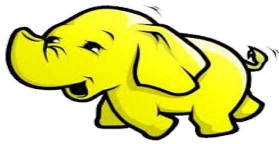


1 Machine
4 I/O channels
Each Channel - 100 mbps

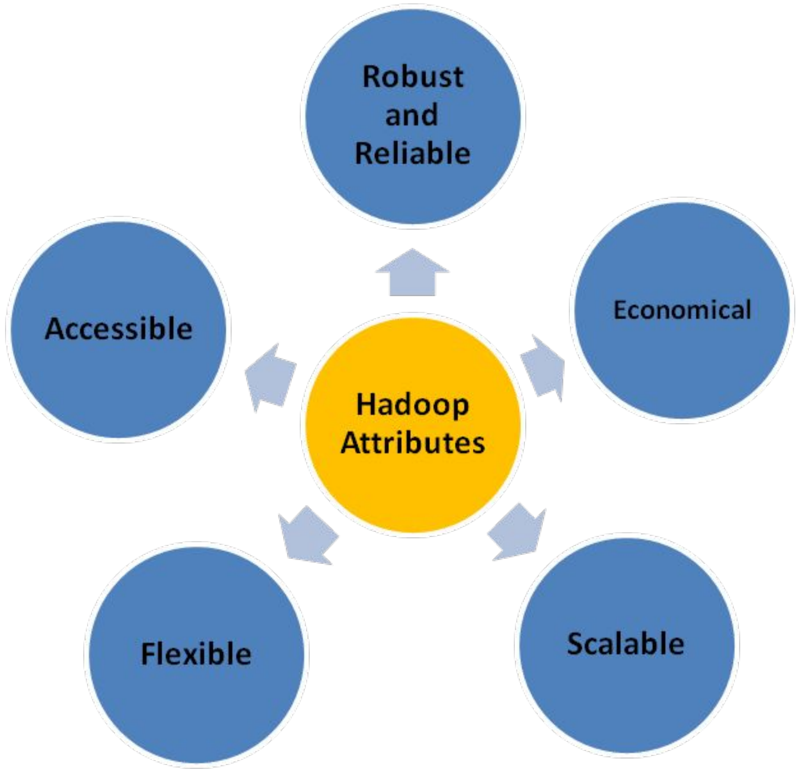
10 Machines
4 I/O channels
Each Channel - 100 mbps

45 MINUTES

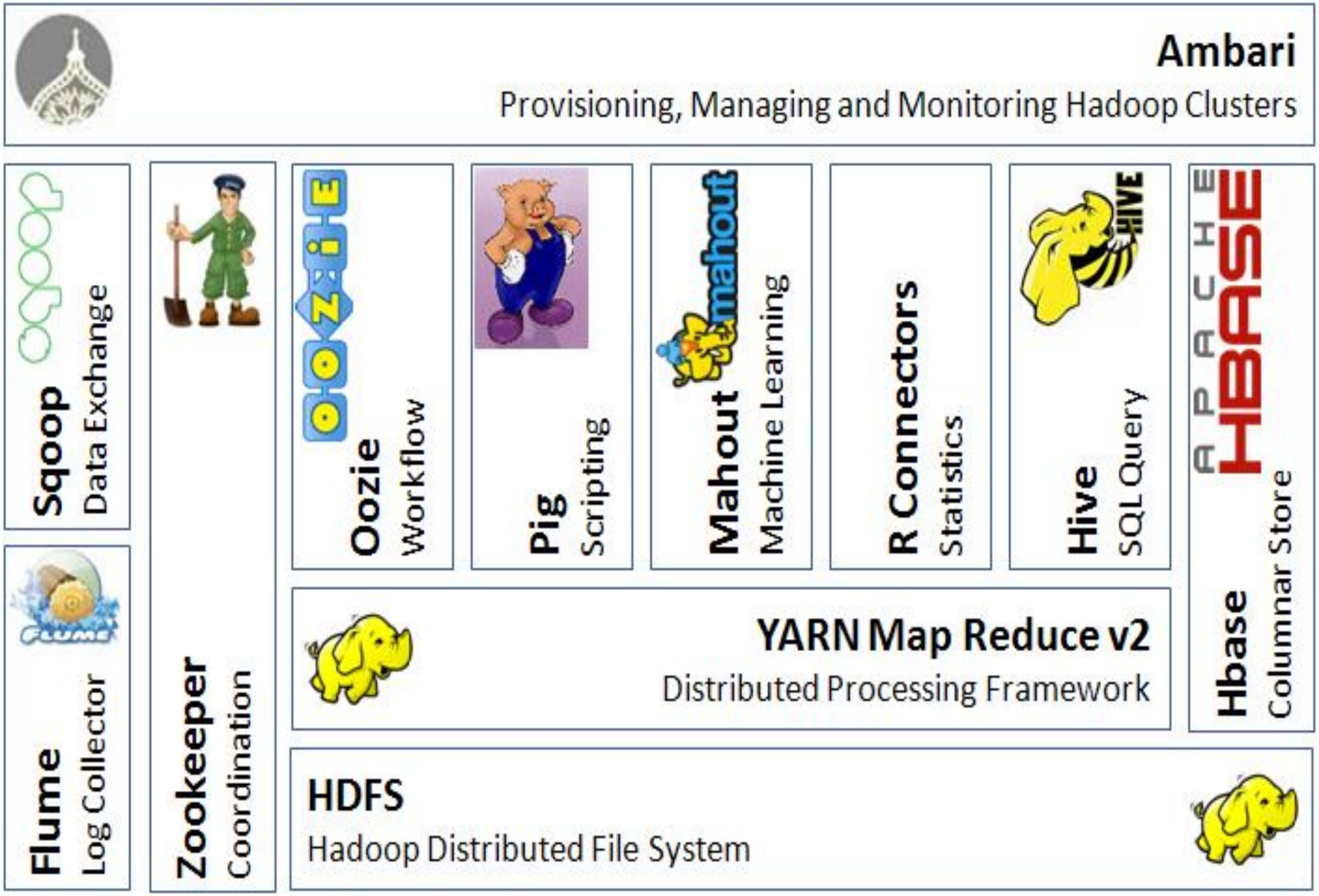
4.5 MINUTES



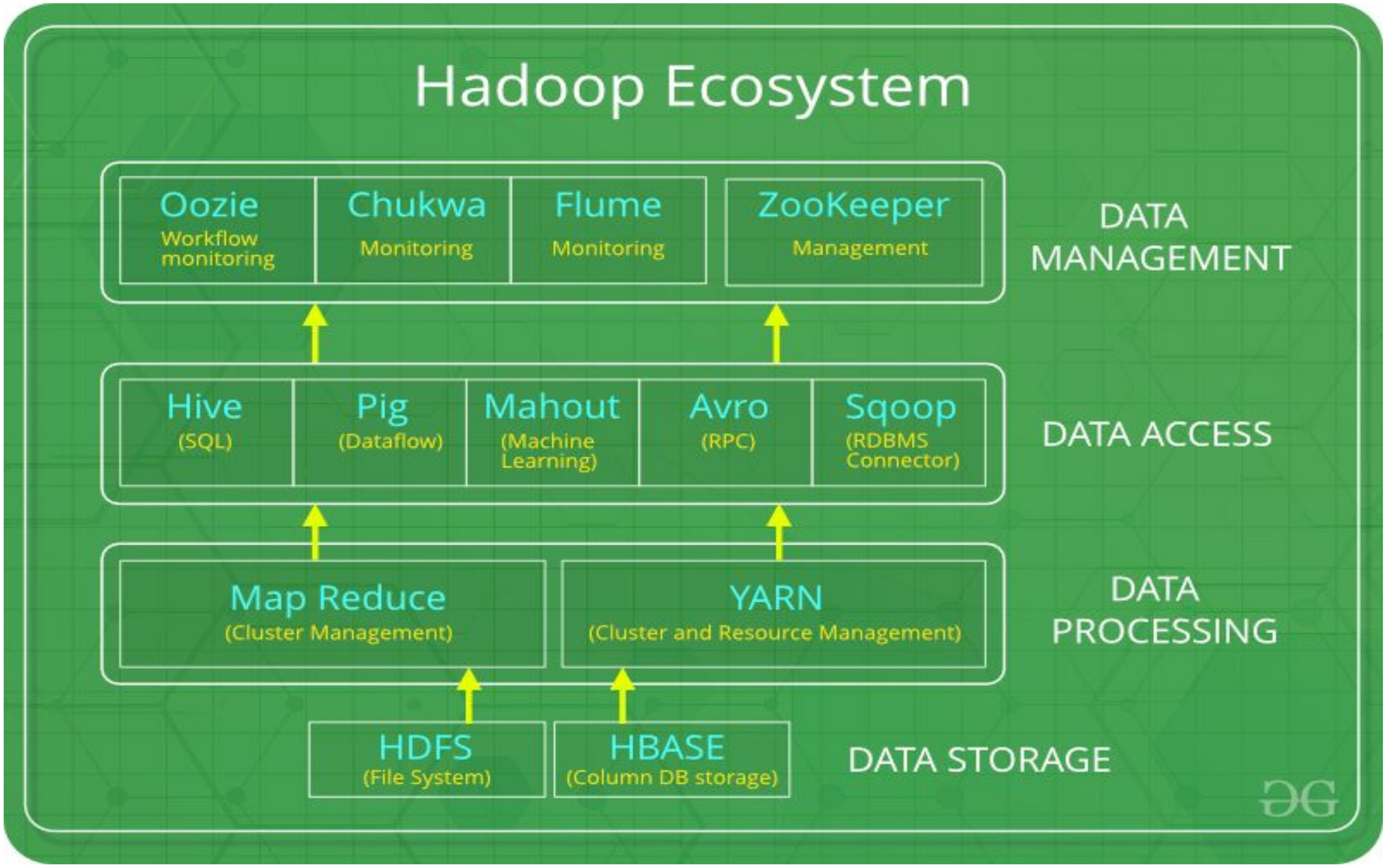
Hadoop Key Characteristics



Apache Hadoop Ecosystem



Apache Hadoop Ecosystem



Core components of Hadoop

- **Mapreduce and HDFS are core components**

- Following are the components that collectively form a Hadoop ecosystem:
- **HDFS:** Hadoop Distributed File System
- **YARN:** Yet Another Resource Negotiator
- **MapReduce:** Programming based Data Processing
- **Spark:** In-Memory data processing
- **PIG, HIVE:** Query based processing of data services
- **HBase:** NoSQL Database
- **Mahout, Spark MLLib:** Machine Learning algorithm libraries
- **Solar, Lucene:** Searching and Indexing
- **Zookeeper:** Managing cluster
- **Oozie:** Job Scheduling d File System(HDFS)

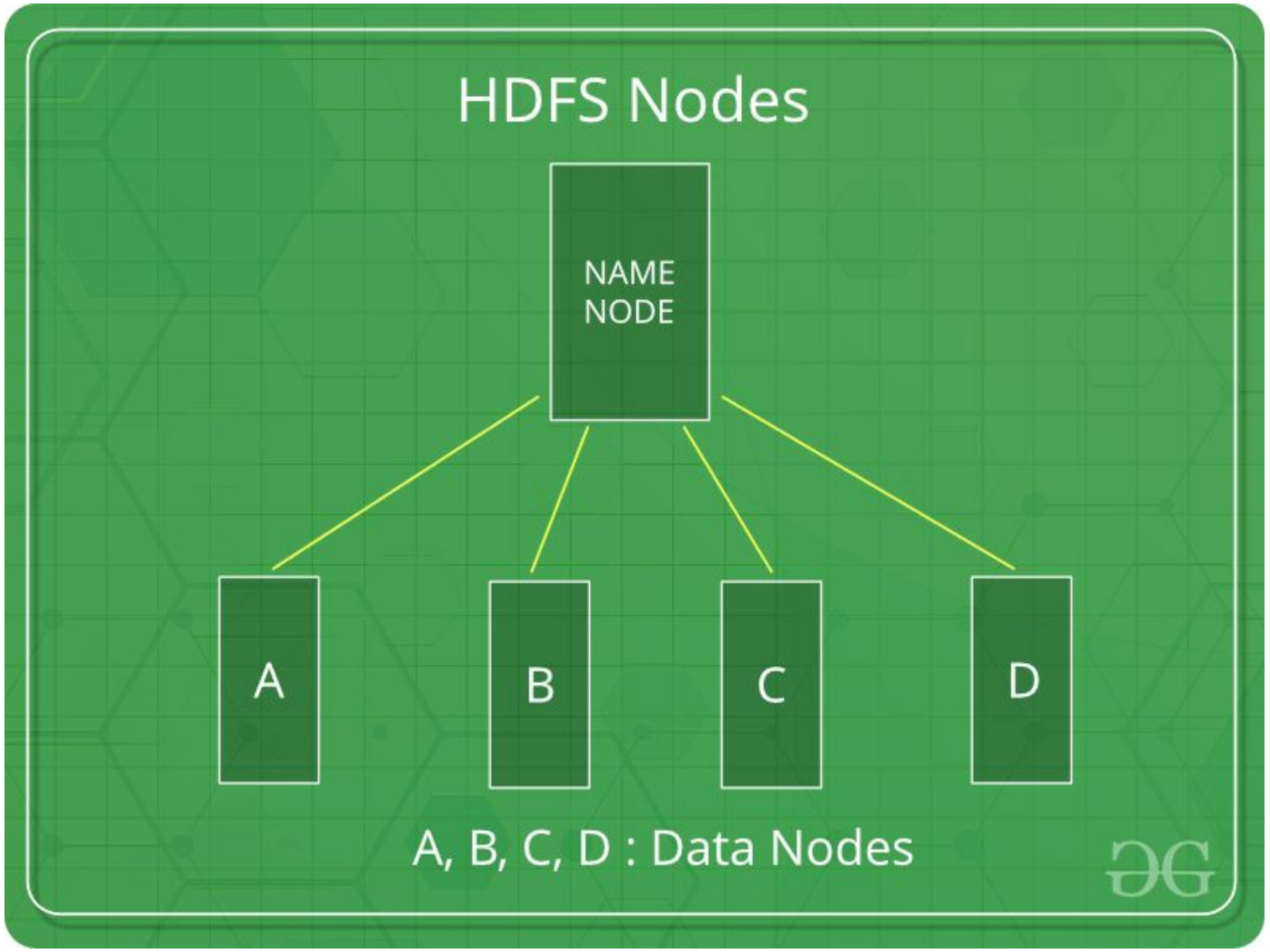
HDFS

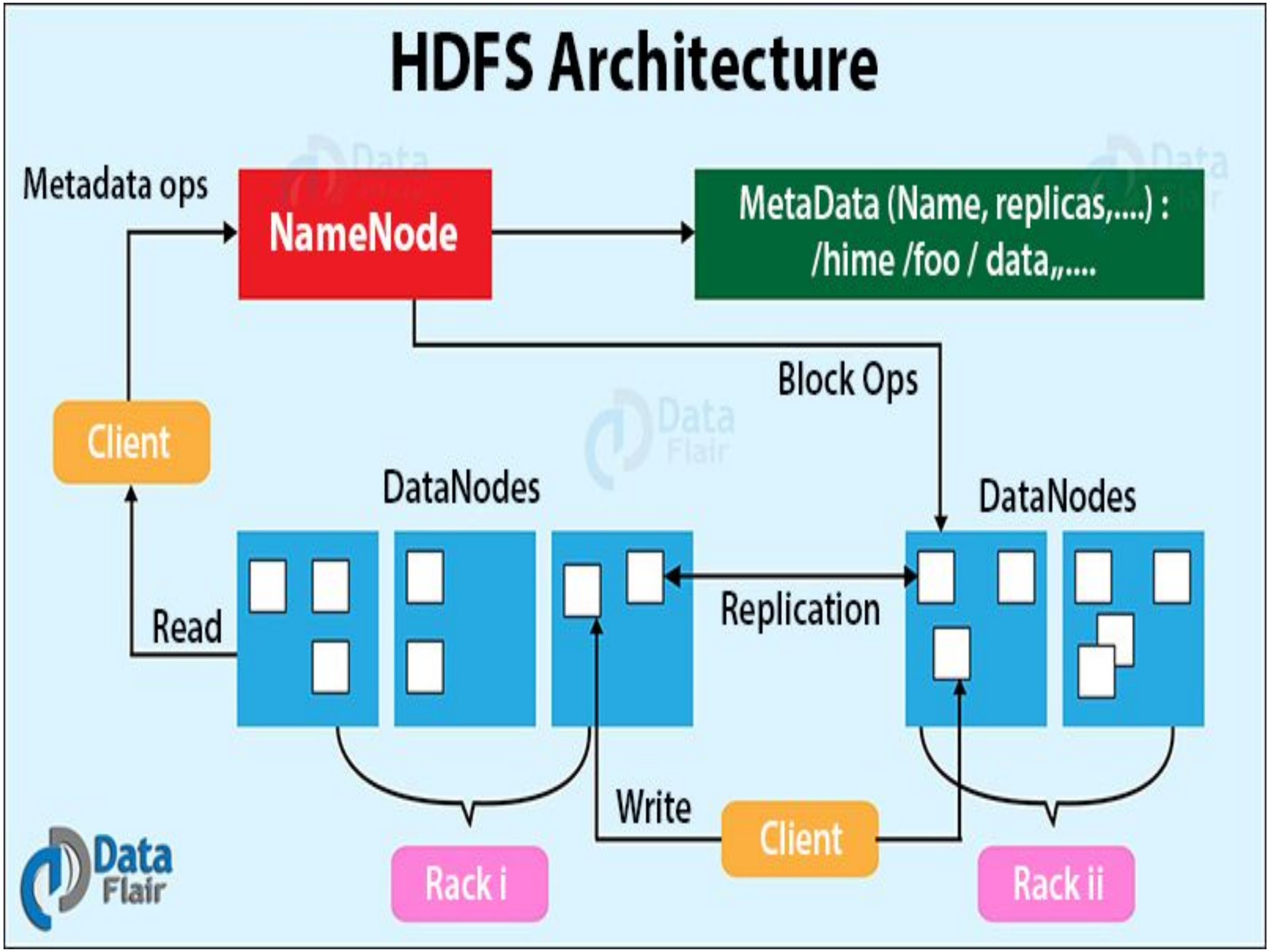
- HDFS is the primary or major component of Hadoop ecosystem and is responsible for storing large data sets of structured or unstructured data across various nodes and thereby maintaining the metadata in the form of log files.
- HDFS consists of two core components i.e.
  - Name node
  - Data Node

HDFS

- Name Node is the prime node which contains metadata (data about data) requiring comparatively fewer resources than the data nodes that stores the actual data.
- These data nodes are commodity hardware in the distributed environment. Undoubtedly, making Hadoop cost effective.
- HDFS maintains all the coordination between the clusters and hardware, thus working at the heart of the system.

HDFS




HDFS Architecture
Metadata ops
NameNode
MetaData (Name, replicas,....) : /hime /foo / data,,....
Block Ops
Client
DataNodes
DataNodes
Read
Replication
Write
Client
Rack i
Rack ii

What is HDFS NameNode?

- NameNode is the centerpiece of the Hadoop Distributed File System. It maintains and manages the **file system namespace** and provides the right access permission to the clients.
- The NameNode stores information about blocks locations, permissions, etc. on the local disk in the form of two files:
- **Fsimage:** Fsimage stands for File System image. It contains the complete namespace of the Hadoop file system since the NameNode creation.
- **Edit log:** It contains all the recent changes performed to the file system namespace to the most recent Fsimage.

Functions of HDFS NameNode

- It executes the file system namespace operations like opening, renaming, and closing files and directories.
- NameNode manages and maintains the DataNodes.
- It determines the mapping of blocks of a file to DataNodes.
- NameNode records each change made to the file system namespace.
- It keeps the locations of each block of a file.
- NameNode takes care of the replication factor of all the blocks.
- NameNode receives heartbeat and block reports from all DataNodes that ensure DataNode is alive.
- If the DataNode fails, the NameNode chooses new DataNodes for new replicas.

What is HDFS DataNode?

- DataNodes are the slave nodes in Hadoop HDFS.
- DataNodes are **inexpensive commodity hardware**. They store blocks of a file.

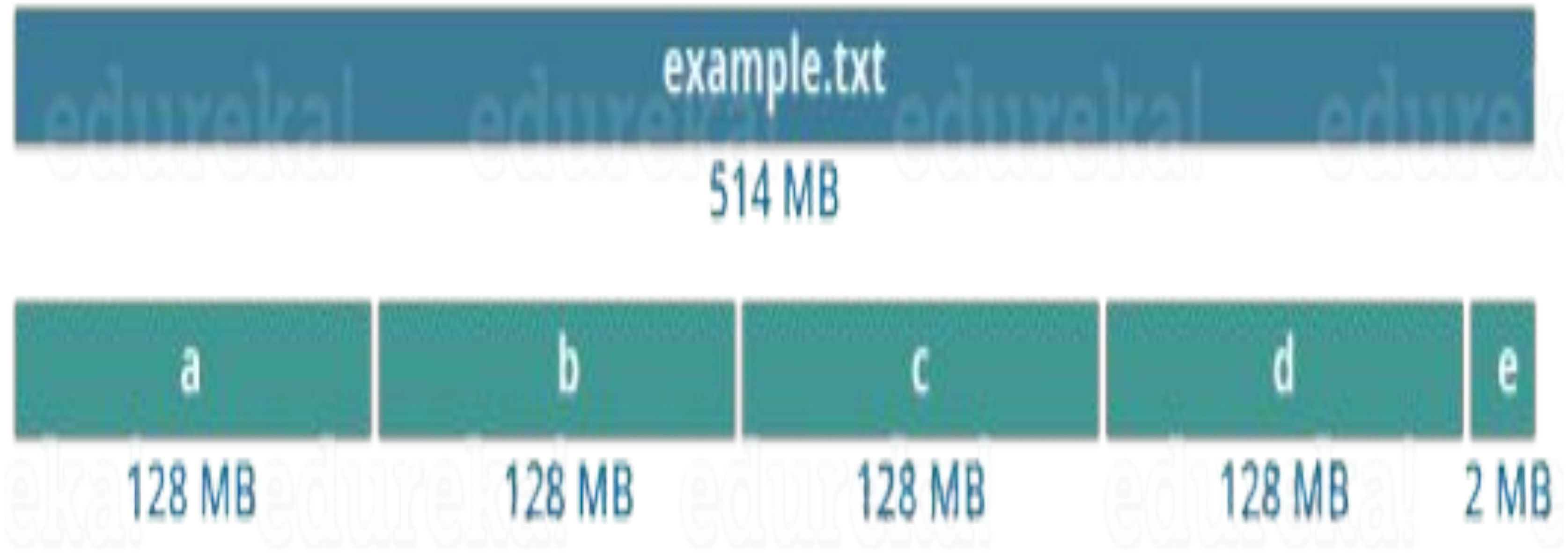
Functions of DataNode

- DataNode is responsible for serving the client read/write requests.
- Based on the instruction from the NameNode, DataNodes performs block creation, replication, and deletion.
- DataNodes send a heartbeat to NameNode to report the health of HDFS.
- DataNodes also sends block reports to NameNode to report the list of blocks it contains.

Blocks:

- Now, as we know that the data in HDFS is scattered across the DataNodes as blocks. **Let’s have a look at what is a block and how is it formed?**
- Blocks are the nothing but the smallest continuous location on your hard drive where data is stored. In general, in any of the File System, you store the data as a collection of blocks. Similarly, HDFS stores each file as blocks which are scattered throughout the Apache Hadoop cluster.
- The default size of each block is 128 MB in Apache Hadoop 2.x (64 MB in Apache Hadoop 1.x) which you can configure as per your requirement.

example

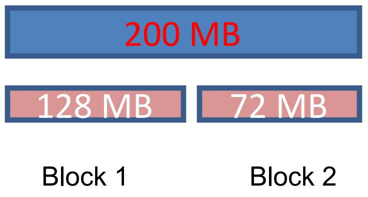

example.txt
514 MB
a
128 MB
b
128 MB
c
128 MB
d
128 MB
e
2 MB
edureka!

FILE BLOCKS

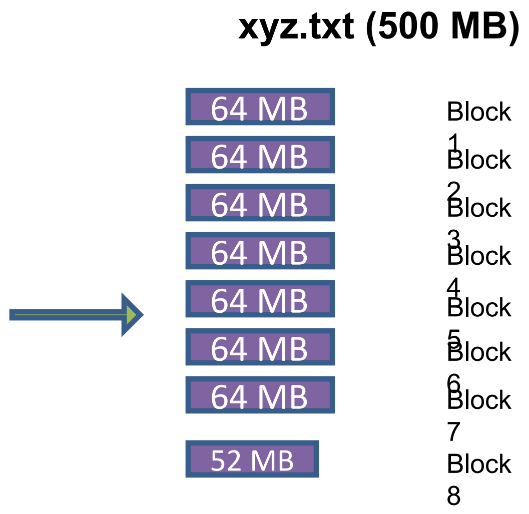
By Default in hadoop 1 data is split as 64 MB block

By Default in hadoop 2 data is split as 128 MB block

Suppose consider a file abc.txt of size 200 MB, considering hadoop 2 the file would be divided as 128 MB as one block and 72MB as another block



Suppose consider a file xyz.txt of size 500 MB, considering hadoop1 the file would be divided as 7 * 64 MB blocks and 52 MB as one block as can be seen from the figure



Advantages of HDFS Block

- The benefits with HDFS block are:

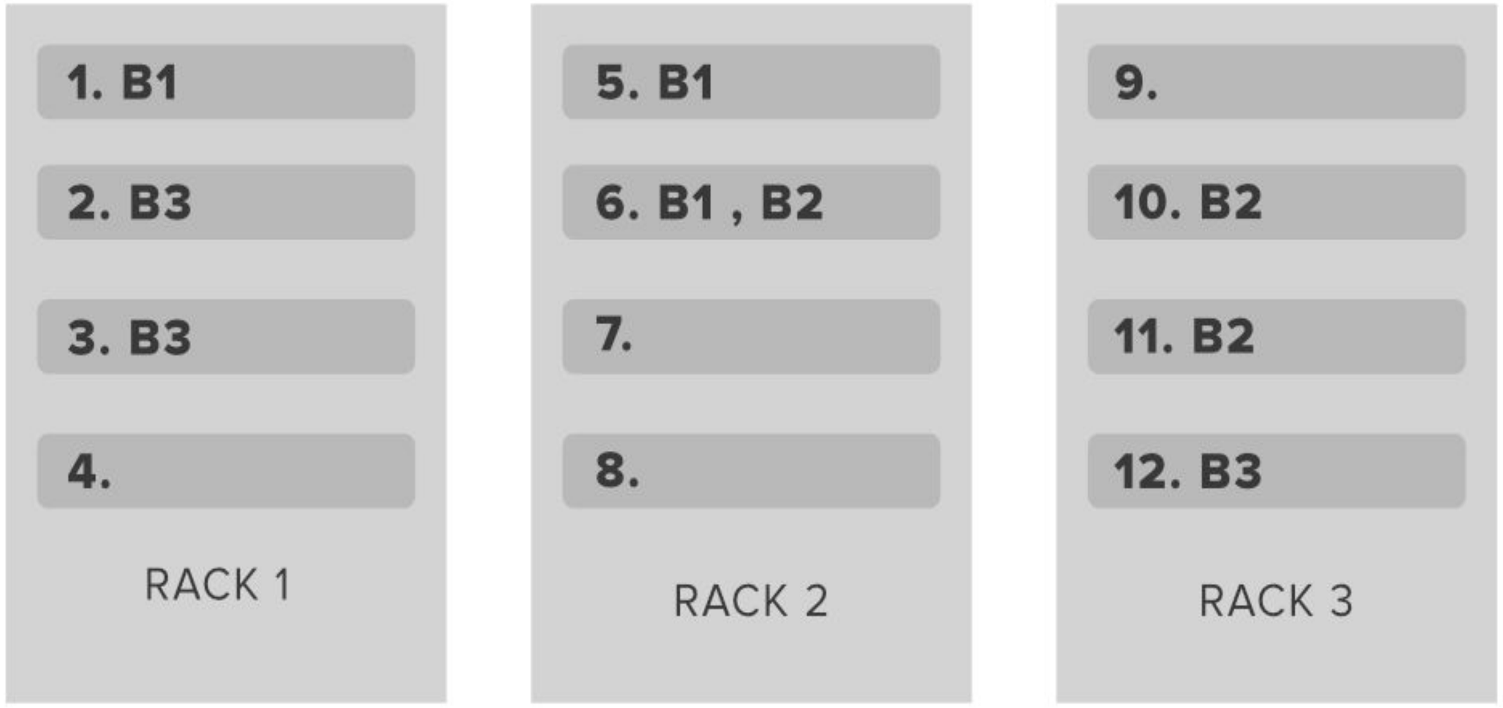
- The blocks are of fixed size, so it is very easy to calculate the number of blocks that can be stored on a disk.
- HDFS block concept simplifies the storage of the datanodes.
- The datanodes doesn’t need to concern about the blocks metadata data like file permissions etc.
- The namenode maintains the metadata of all the blocks.
- If the size of the file is less than the HDFS block size, then the file does not occupy the complete block storage.
- As the file is chunked into blocks, it is easy to store a file that is larger than the disk size as the data blocks are distributed and stored on multiple nodes in a hadoop cluster.
- Blocks are easy to replicate between the datanodes and thus provide fault tolerance and high availability. Hadoop framework replicates each block across multiple nodes (default replication factor is 3). I
- n case of any node failure or block corruption, the same block can be read from another node

Rack

- **Rack** is the collection of around 40-50 machines (DataNodes) connected using the same network switch.
- If the network goes down, the whole rack will be unavailable.
- **Rack Awareness** is the concept of choosing the closest node based on the rack information.
- To ensure that all the replicas of a block are not stored on the same rack or a single rack, NameNode follows a rack awareness algorithm to store replicas and provide latency and fault tolerance.
- Suppose if the replication factor is 3, then according to the rack awareness algorithm:
- The first replica will get stored on the local rack.
- The second replica will get stored on the other DataNode in the same rack.
- The third replica will get stored on a different rack.

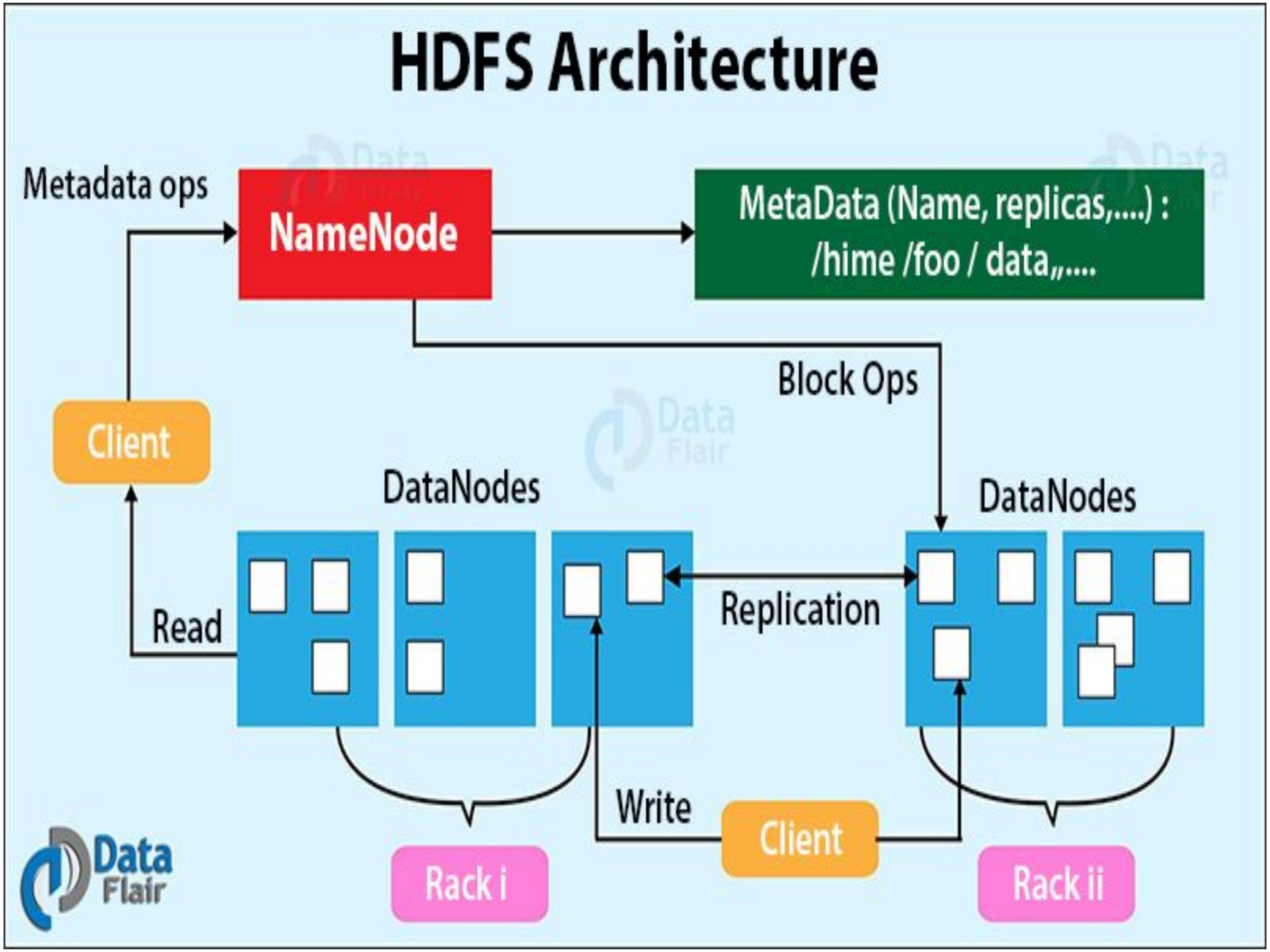
Replica Placement via Rack Awareness:

==> **Block 1 , Block 2 , Block 3**



- In the above image, we have 3 different Racks in our Hadoop cluster each Rack contains 4 Data node.
- Now suppose you have 3 file blocks(Block 1, Block 2, Block 3) that you want to put in this data node.
- As we all know Hadoop has a Feature to make Replica’s of the file blocks to provide the high availability and fault tolerance. By default, the Replication Factor is 3 so Hadoop is so smart that it will place the replica’s of Blocks in Racks in such a way that we can achieve a good network bandwidth.
- For that Hadoop has some *Rack awareness policies*.
- There should not be more than 1 replica on the same Data node.
- More then 2 replica’s of a single block is not allowed on the same Rack.
- The number of racks used inside a Hadoop cluster must be smaller than the number of replicas.

- Now let’s continue with our above example.
- In the diagram, we can easily found that we have block 1 in the first Datanode of Rack 1 and 2 replica’s of Block 1 in 5 and 6 number Data node of Rack which sum up to 3.
- Similarly, we also have a Replica distribution of 2 other blocks in different Racks which are following the above policies.
- **Benefits of Implementing Rack Awareness in our Hadoop Cluster:**
- With the rack awareness policy’s we store the data in different Racks so no way to lose our data.
- Rack awareness helps to maximize the network bandwidth because the data blocks transfer within the Racks.
- It also improves the cluster performance and provides high data availability.


HDFS Architecture
Metadata ops
NameNode
MetaData (Name, replicas,....) : /hime /foo / data,,....
Block Ops
Client
DataNodes
DataNodes
Read
Replication
Write
Client
Rack i
Rack ii
Data Flair

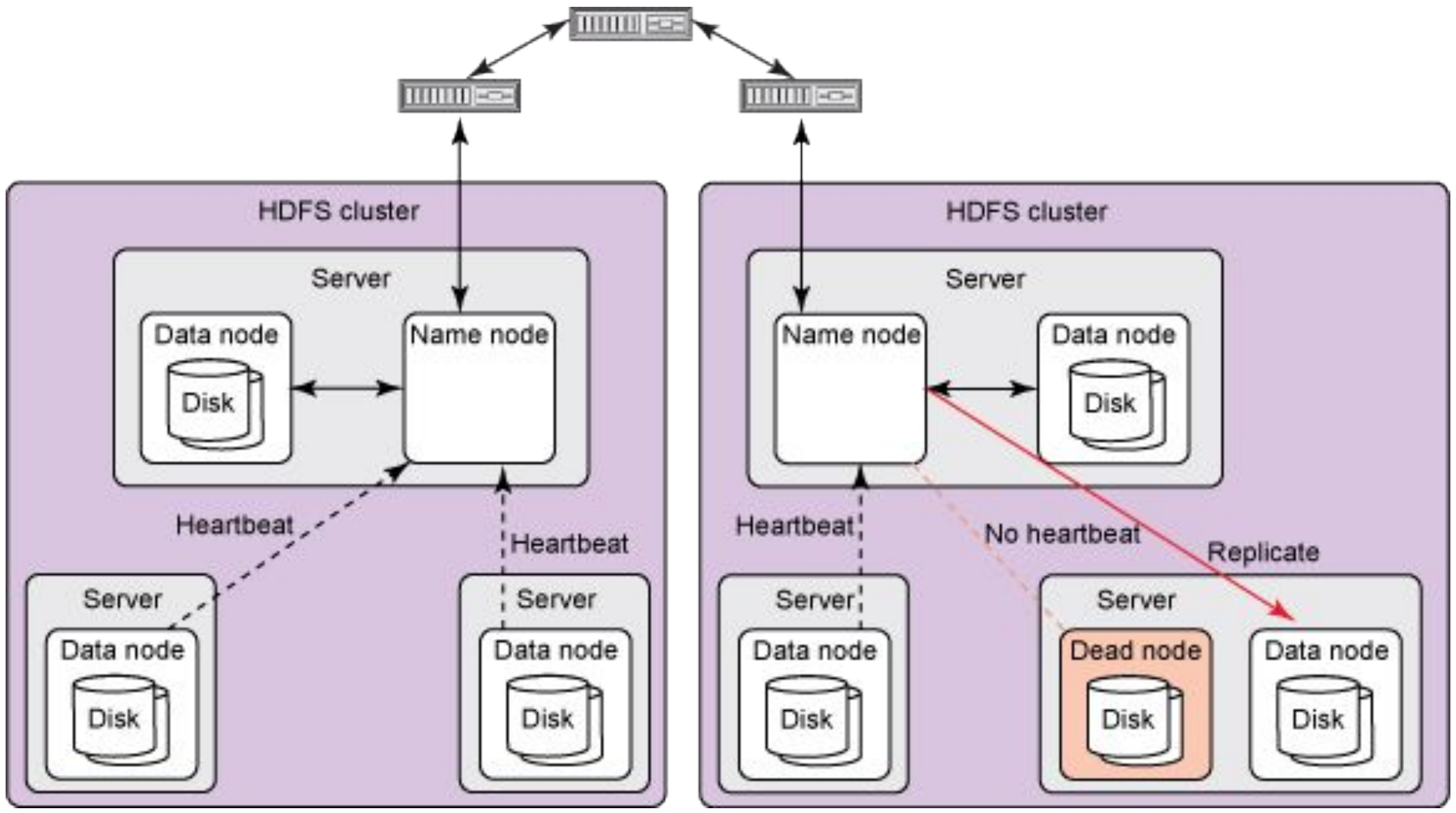
Write Operation

- When a client wants to write a file to HDFS, it communicates to the NameNode for metadata.
- The Namenode responds with a number of blocks, their location, replicas, and other details.
- Based on information from NameNode, the client directly interacts with the DataNode.
- The client first sends block A to DataNode 1 along with the IP of the other two DataNodes where replicas will be stored.
- When Datanode 1 receives block A from the client, DataNode 1 copies the same block to DataNode 2 of the same rack.
- As both the DataNodes are in the same rack, so block transfer via rack switch.
- Now DataNode 2 copies the same block to DataNode 4 on a different rack. As both the DataNoNes are in different racks, so block transfer via an out-of-rack switch.
- When DataNode receives the blocks from the client, it sends write confirmation to Namenode.
- The same process is repeated for each block of the file.

Read Operation

- To **read from HDFS**, the client first communicates with the NameNode for metadata.
- The Namenode responds with the locations of DataNodes containing blocks.
- After receiving the DataNodes locations, the client then directly interacts with the DataNodes.
- The client starts reading data parallelly from the DataNodes based on the information received from the NameNode.
- The data will flow directly from the DataNode to the client.
- When a client or application receives all the blocks of the file, it combines these blocks into the form of an original file.

Concept of Blocks in HDFS Architecture



- When a heartbeat message reappears or a new heartbeat message is received, the respective data node sending the message is added to the cluster
- To enable task of reliability following tasks for failure management
- Monitoring
- Rebalancing
- Metadata replication

Command line interface

- Hdfs can be viewed by interfacing with it from the command line.
- There are various numerous interfaces command line is one of the easiest
- There are two properties that are set in the distributed mode
- The principal is fs.default.name set to hdfs://localhost/ which is used to set a default Hadoop file system

Using HDFS Files

- import java.io.File;
- import java.io.IOException;
- import org.apache.hadoop.conf.Configuration;
- import org.apache.hadoop.fs.FileSystem;
- import org.apache.hadoop.fs.FSDataInputStream;
- import org.apache.hadoop.fs.FSDataOutputStream;
- import org.apache.hadoop.fs.Path;
- public class HDFSExample {
- public static void main (String [] args) throws IOException {
- String exampleF = "example.txt";
- //Creating a Filesystem Object
- Configuration config = new Configuration();//creates a configuration object config
- FileSystem fsys = FileSystem.get(config);//Creates FileSystem object fs

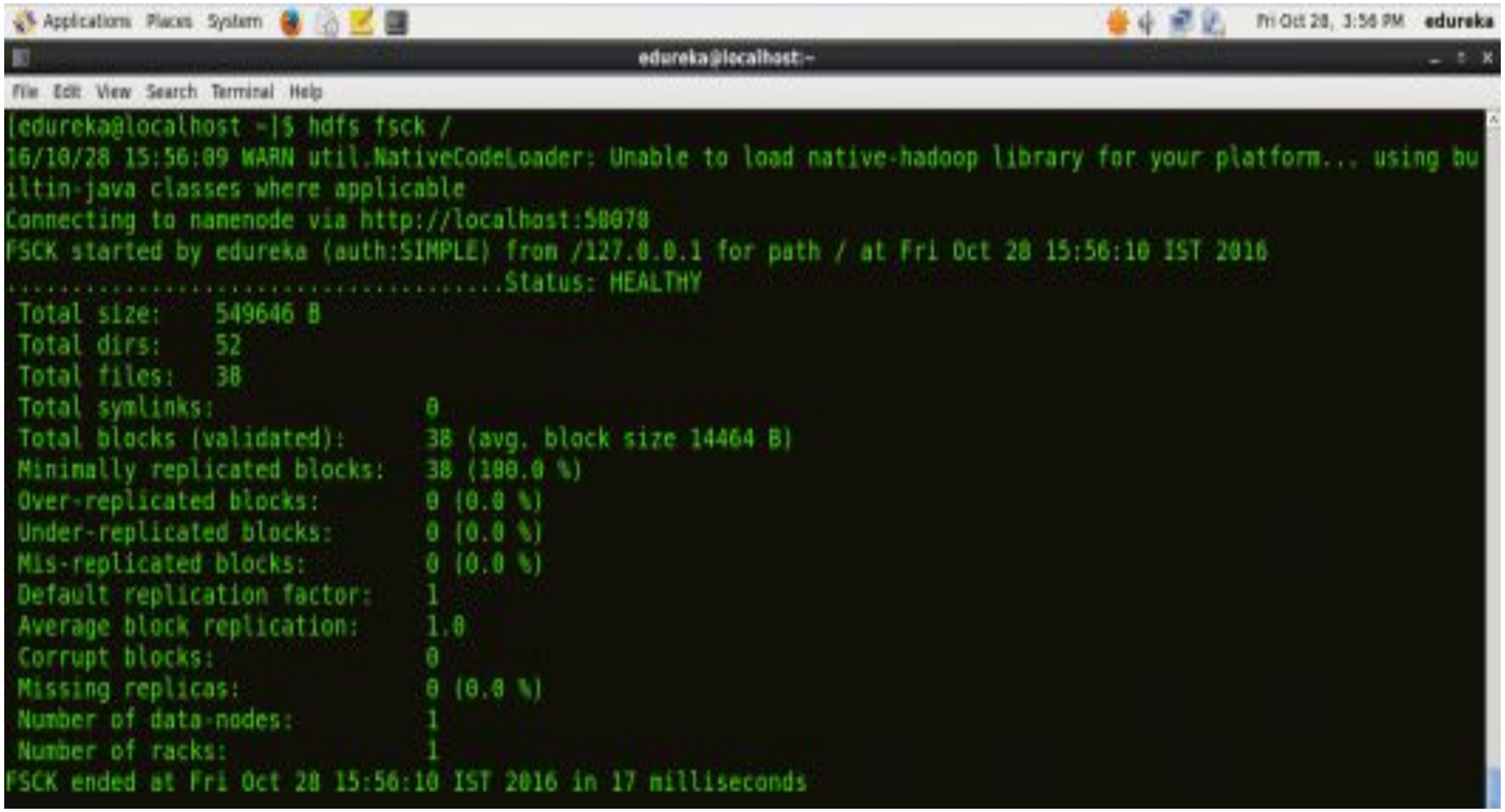
```
Path fp = new Path(exampleF);//creating an object for the path
class//
If(fsys.exists(fp)//checking file path
If (fsys.isFile(fp))
Boolean result=fsys.CreateNewFile(fp);
Boolean result=fsys.delete(fp);
FSDataInputStream fin=fsys.open(fp);//reading from the file
FSDataOutputStream fout=fsys.create(fp);//writing to the file
```

HDFS Commands

- Hadoop FS Command Line
- The Hadoop FS command line is a simple way to access and interface with HDFS.
- Below are some basic HDFS commands in Linux, including operations like creating directories, moving files, deleting files, reading files, and listing directories.

Fsck command

HDFS Command to check the health of the Hadoop file system.
Command: **hdfs fsck /**

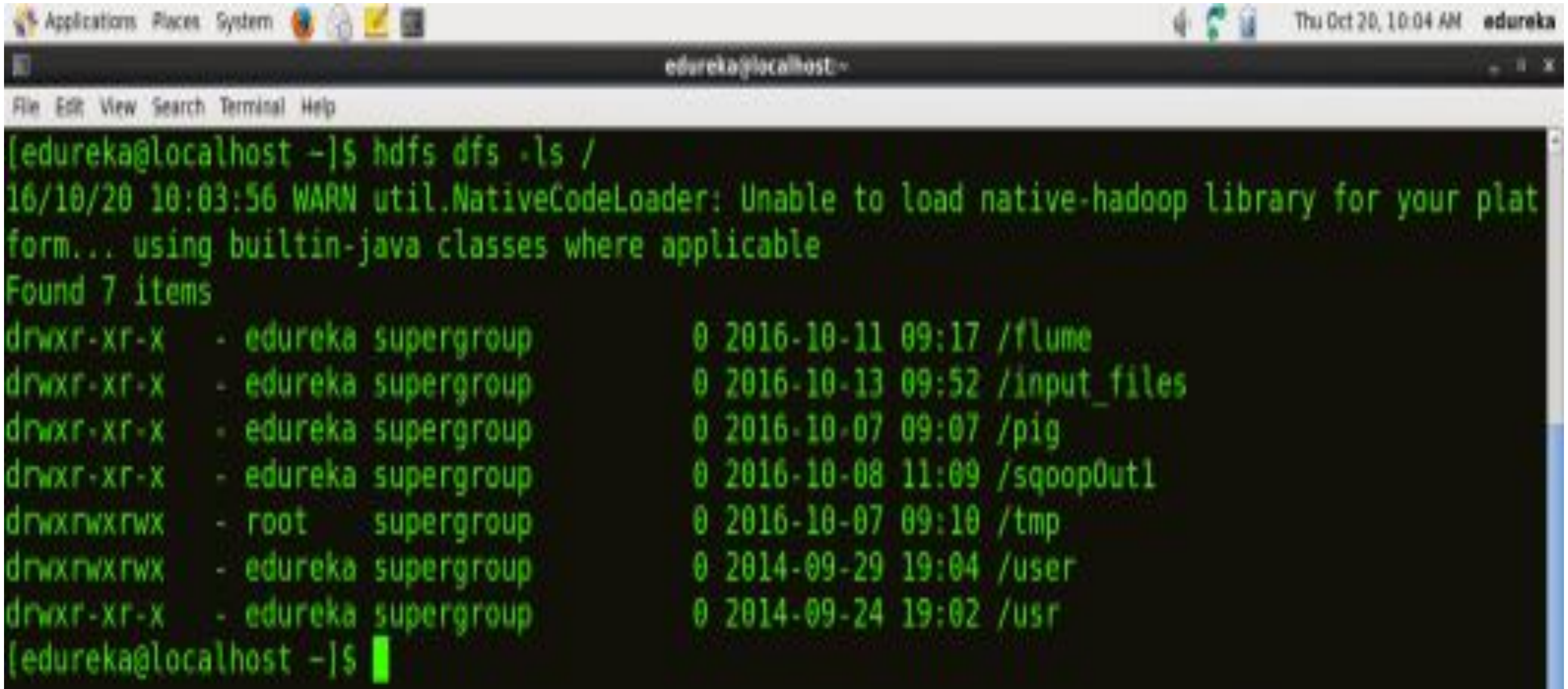


Applications Places System
Fri Oct 28, 3:56 PM edureka
edureka@localhost:~
File Edit View Search Terminal Help

```
[edureka@localhost ~]$ hdfs fsck /
16/10/28 15:56:09 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using bu
iltin-java classes where applicable
Connecting to namenode via http://localhost:50070
FSCK started by edureka (auth:SIMPLE) from /127.0.0.1 for path / at Fri Oct 28 15:56:10 IST 2016
......................................Status: HEALTHY
 Total size:    549646 B
 Total dirs:    52
 Total files:   38
 Total symlinks:                0
 Total blocks (validated):      38 (avg. block size 14464 B)
 Minimally replicated blocks:   38 (100.0 %)
 Over-replicated blocks:        0 (0.0 %)
 Under-replicated blocks:       0 (0.0 %)
 Mis-replicated blocks:         0 (0.0 %)
 Default replication factor:    1
 Average block replication:     1.0
 Corrupt blocks:                0
 Missing replicas:              0 (0.0 %)
 Number of data-nodes:          1
 Number of racks:               1
FSCK ended at Fri Oct 28 15:56:10 IST 2016 in 17 milliseconds
```

ls

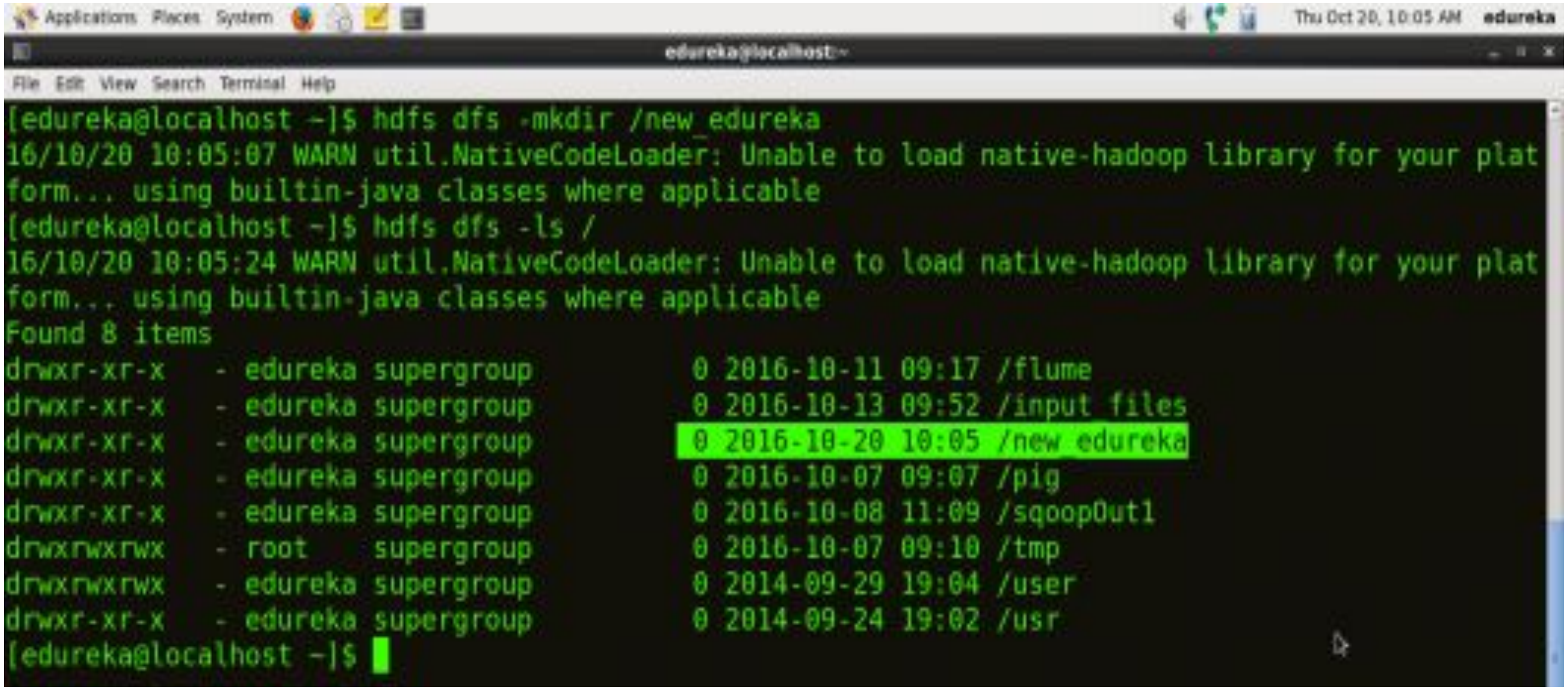
- HDFS Command to display the list of Files and Directories in HDFS.
- ***Command:*** **hdfs dfs –ls /**


Applications Places System
Thu Oct 20, 10:04 AM edureka
edureka@localhost:~
File Edit View Search Terminal Help

```
[edureka@localhost ~]$ hdfs dfs -ls /
16/10/20 10:03:56 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your plat
form... using builtin-java classes where applicable
Found 7 items
drwxr-xr-x   - edureka supergroup          0 2016-10-11 09:17 /flume
drwxr-xr-x   - edureka supergroup          0 2016-10-13 09:52 /input_files
drwxr-xr-x   - edureka supergroup          0 2016-10-07 09:07 /pig
drwxr-xr-x   - edureka supergroup          0 2016-10-08 11:09 /sqoopOut1
drwxrwxrwx   - root    supergroup          0 2016-10-07 09:10 /tmp
drwxrwxrwx   - edureka supergroup          0 2014-09-29 19:04 /user
drwxr-xr-x   - edureka supergroup          0 2014-09-24 19:02 /usr
[edureka@localhost ~]$ 
```

mkdir

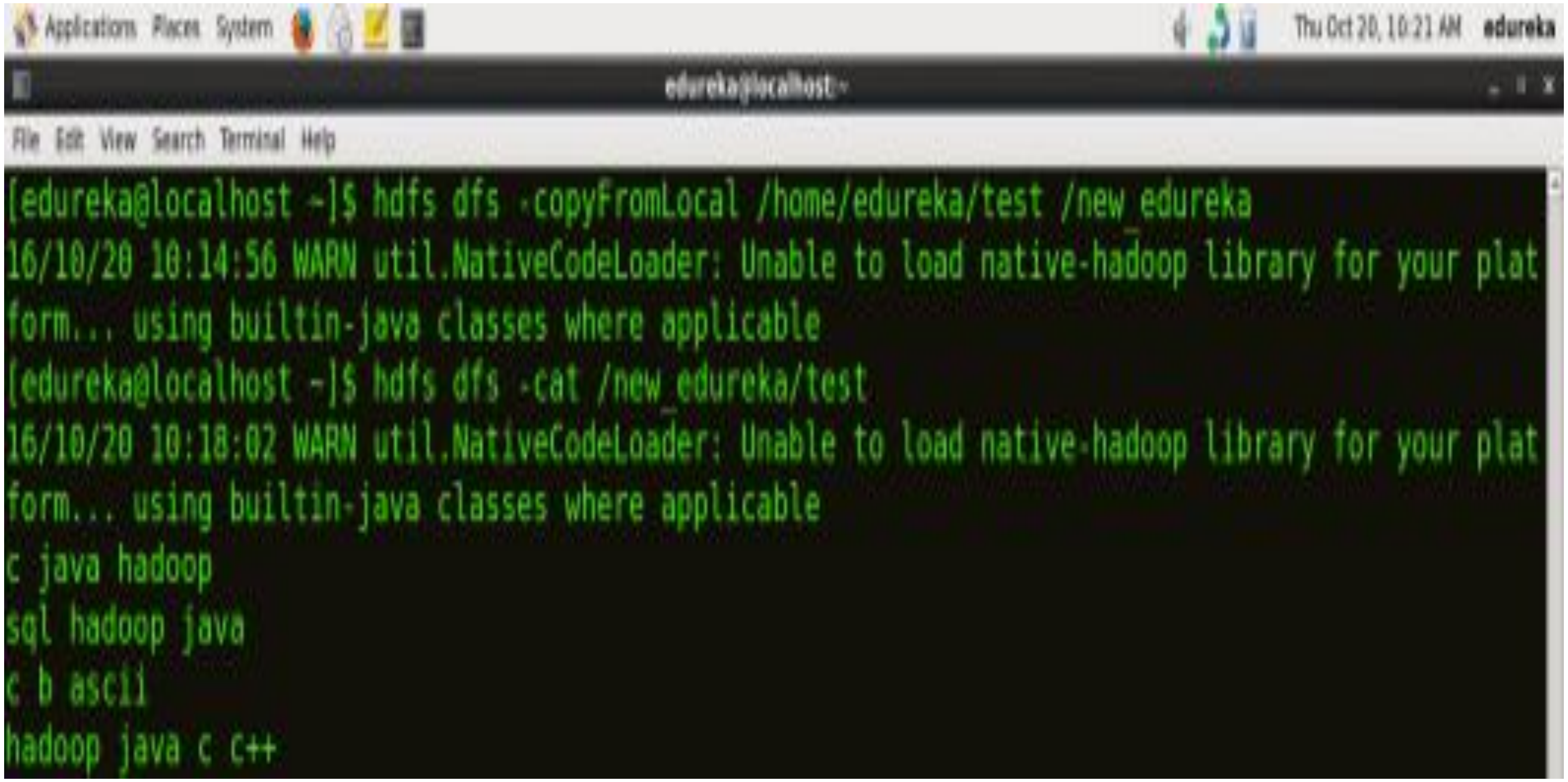
- HDFS Command to create the directory in HDFS.
- ***Usage:* hdfs dfs –mkdir /directory_name**
- ***Command:* hdfs dfs –mkdir /new_edureka**


Applications Places System
Thu Oct 20, 10:05 AM edureka
edureka@localhost:~
File Edit View Search Terminal Help

```
[edureka@localhost ~]$ hdfs dfs -mkdir /new_edureka
16/10/20 10:05:07 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your plat
form... using builtin-java classes where applicable
[edureka@localhost ~]$ hdfs dfs -ls /
16/10/20 10:05:24 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your plat
form... using builtin-java classes where applicable
Found 8 items
drwxr-xr-x   - edureka supergroup          0 2016-10-11 09:17 /flume
drwxr-xr-x   - edureka supergroup          0 2016-10-13 09:52 /input_files
drwxr-xr-x   - edureka supergroup          0 2016-10-20 10:05 /new_edureka
drwxr-xr-x   - edureka supergroup          0 2016-10-07 09:07 /pig
drwxr-xr-x   - edureka supergroup          0 2016-10-08 11:09 /sqoopOut1
drwxrwxrwx   - root    supergroup          0 2016-10-07 09:10 /tmp
drwxrwxrwx   - edureka supergroup          0 2014-09-29 19:04 /user
drwxr-xr-x   - edureka supergroup          0 2014-09-24 19:02 /usr
[edureka@localhost ~]$
```

cat

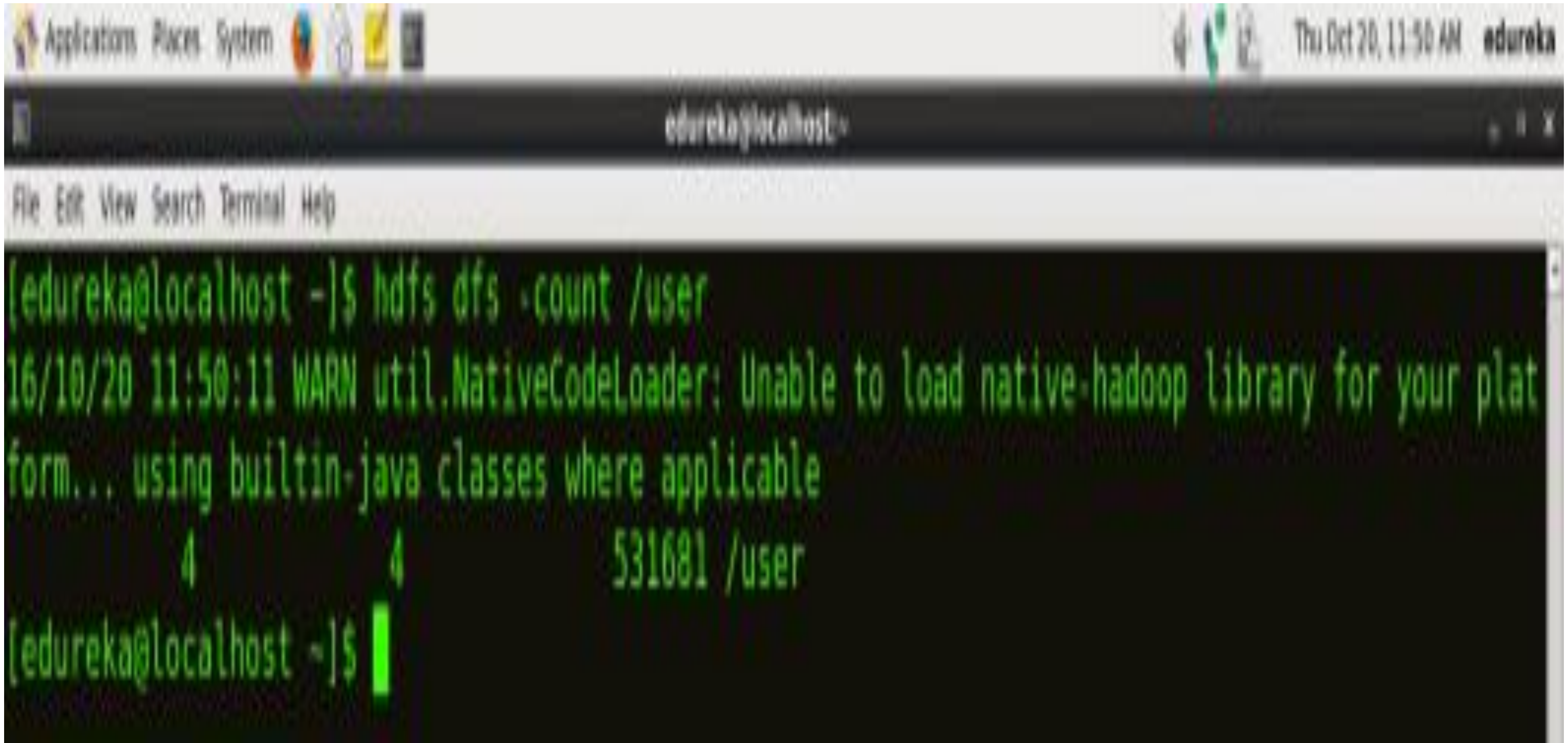
- HDFS Command that reads a file on HDFS and prints the content of that file to the standard output.
- ***Usage:*** **hdfs dfs –cat /path/to/file_in_hdfs**
- ***Command:*** **hdfs dfs –cat /new_edureka/test**


Applications Places System
Thu Oct 20, 10:21 AM edureka
edureka@localhost:~
File Edit View Search Terminal Help

```
[edureka@localhost ~]$ hdfs dfs -copyFromLocal /home/edureka/test /new_edureka
16/10/20 10:14:56 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your plat
form... using builtin-java classes where applicable
[edureka@localhost ~]$ hdfs dfs -cat /new_edureka/test
16/10/20 10:18:02 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your plat
form... using builtin-java classes where applicable
c java hadoop
sql hadoop java
c b ascii
hadoop java c C++
```

count

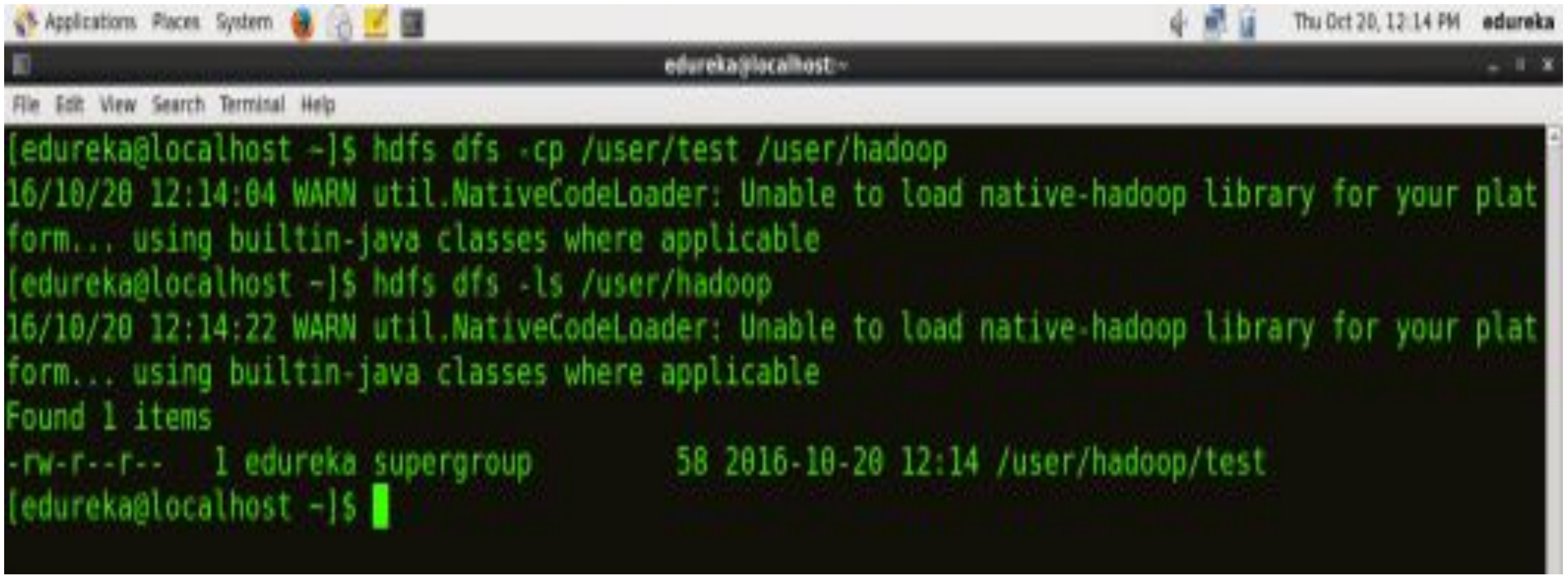
- HDFS Command to count the number of directories, files, and bytes under the paths that match the specified file pattern.
- ***Usage:*** **hdfs dfs -count <path>**
- ***Command:*** **hdfs dfs –count /user**


Applications Places System
Thu Oct 20, 11:50 AM edureka
edureka@localhost:~
File Edit View Search Terminal Help

```
[edureka@localhost ~]$ hdfs dfs -count /user
16/10/20 11:50:11 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your plat
form... using builtin-java classes where applicable
           4            4             531681 /user
[edureka@localhost ~]$
```

cp

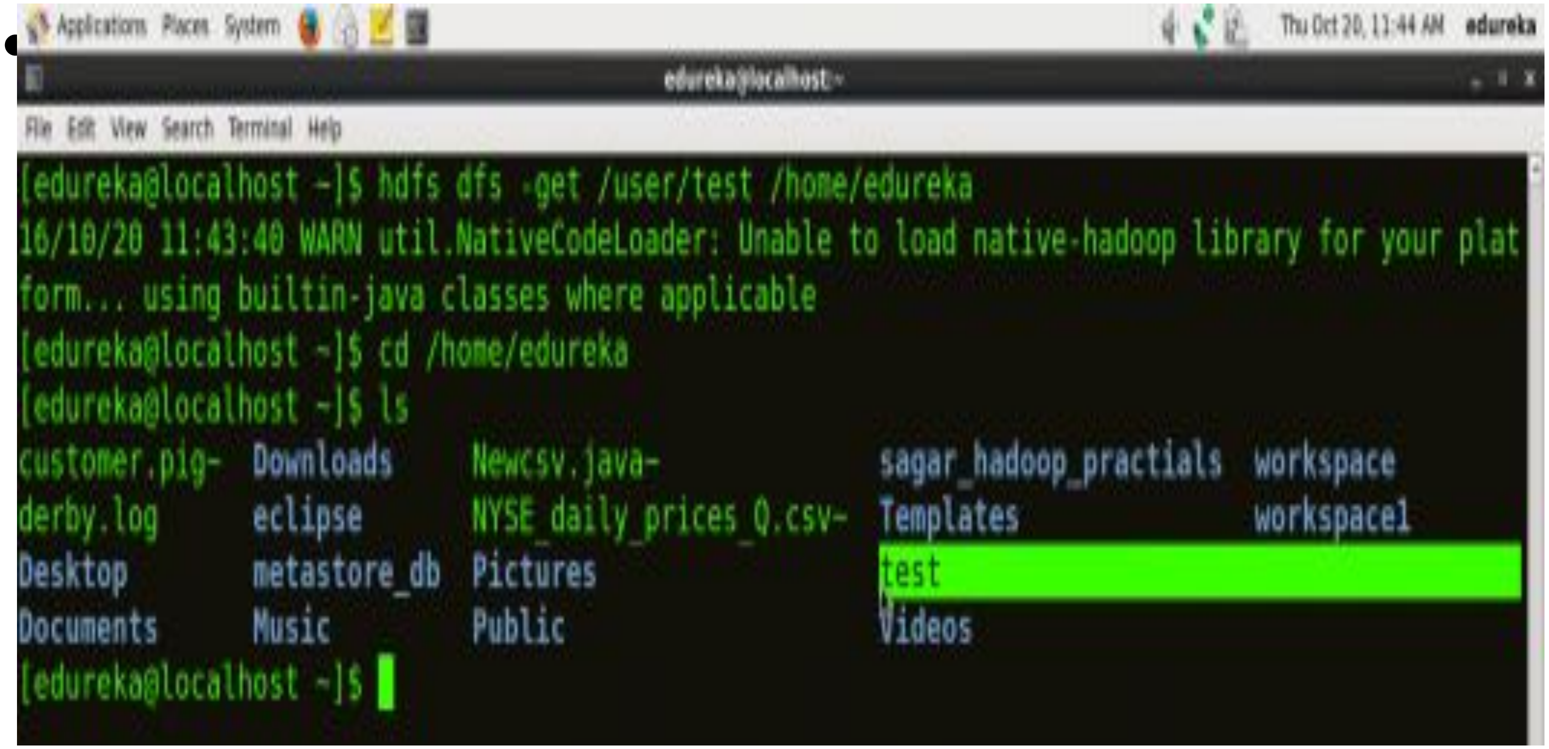
- HDFS Command to copy files from source to destination. This command allows multiple sources as well, in which case the destination must be a directory.
- ***Usage:*** **hdfs dfs -cp <src> <dest>**
- ***Command:*** **hdfs dfs -cp /user/hadoop/file1 /user/hadoop/file2**
- ***Command:*** **hdfs dfs -cp /user/hadoop/file1 /user/hadoop/file2 /user/hadoop/dir**


Applications Places System
Thu Oct 20, 12:14 PM edureka
edureka@localhost:~
File Edit View Search Terminal Help

```
[edureka@localhost ~]$ hdfs dfs -cp /user/test /user/hadoop
16/10/20 12:14:04 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your plat
form... using builtin-java classes where applicable
[edureka@localhost ~]$ hdfs dfs -ls /user/hadoop
16/10/20 12:14:22 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your plat
form... using builtin-java classes where applicable
Found 1 items
-rw-r--r--   1 edureka supergroup         58 2016-10-20 12:14 /user/hadoop/test
[edureka@localhost ~]$ 
```

get

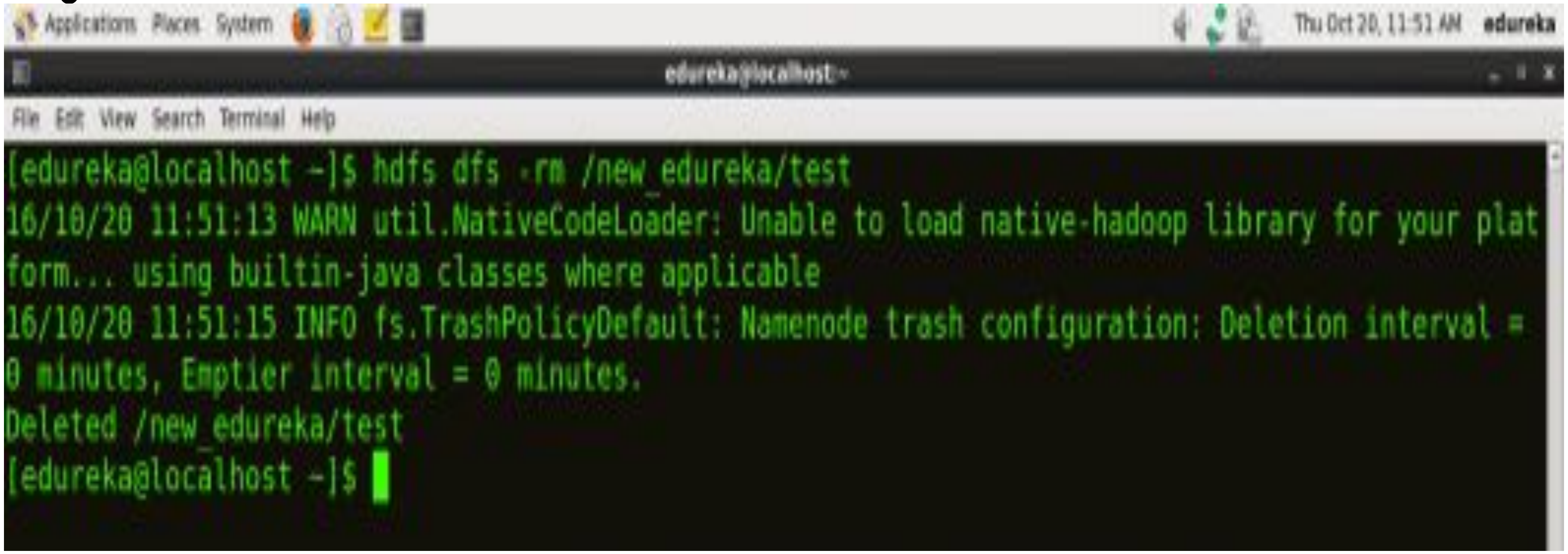
- HDFS Command to copy files from hdfs to the local file system.
- ***Usage:*** **hdfs dfs -get <src> <localdst>**
- ***Command:*** **hdfs dfs –get /user/test /home/edureka**
-


Applications Places System
Thu Oct 20, 11:44 AM edureka
edureka@localhost:~
File Edit View Search Terminal Help

```
[edureka@localhost ~]$ hdfs dfs -get /user/test /home/edureka
16/10/20 11:43:40 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your plat
form... using builtin-java classes where applicable
[edureka@localhost ~]$ cd /home/edureka
[edureka@localhost ~]$ ls
customer.pig~  Downloads     Newcsv.java~              sagar_hadoop_practials  workspace
derby.log      eclipse       NYSE_daily_prices_Q.csv~  Templates               workspace1
Desktop        metastore_db  Pictures                  test
Documents      Music         Public                    Videos
[edureka@localhost ~]$ 
```

rm

- HDFS Command to remove the file from HDFS.
- *Usage:* **hdfs dfs –rm <path>**
- ***Command:*** **hdfs dfs –rm /new_edureka/test**
-


Applications Places System
Thu Oct 20, 11:51 AM edureka
edureka@localhost:~
File Edit View Search Terminal Help

```
[edureka@localhost ~]$ hdfs dfs -rm /new_edureka/test
16/10/20 11:51:13 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your plat
form... using builtin-java classes where applicable
16/10/20 11:51:15 INFO fs.TrashPolicyDefault: Namenode trash configuration: Deletion interval =
0 minutes, Emptier interval = 0 minutes.
Deleted /new_edureka/test
[edureka@localhost ~]$
```

To use HDFS commands, start the Hadoop services using the following command:

```
sbin/start-all.sh
```

To check if Hadoop is up and running:

```
jps
```

mkdir:

To create a directory, similar to Unix ls command.

Options:

-p : Do not fail if the directory already exists

```
$ hadoop fs -mkdir  [-p]
```

ls:

List directories present under a specific directory in HDFS, similar to Unix ls command. The -lsr command can be used for recursive listing of directories and files.

Options:

-d : List the directories as plain files

-h : Format the sizes of files to a human-readable manner instead of number of bytes

-R : Recursively list the contents of directories

```
$ hadoop fs -ls [-d] [-h] [-R]
```

cat:

Display contents of a file, similar to Unix cat command.

```
$ hadoop fs -cat /user/data/sampletext.txt
```

chmod:

Change the permission of a file, similar to Linux shell's command but with a few exceptions.

<MODE> Same as mode used for the shell's command with the only letters recognized are **'rwxXt'**

<OCTALMODE> Mode specified in 3 or 4 digits. It is not possible to specify only part of the mode, unlike the shell command.

Options:

-R : Modify the files recursively

```
$ hadoop fs -chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH
```

chown:

Change owner and group of a file, similar to Linux shell's command but with a few exceptions.

Options:

-R : Modify the files recursively

```
$ hadoop fs -chown [-R] [OWNER][:[GROUP]] PATH
```

mv:

Move files from one directory to another within HDFS, similar to Unix mv command.

```
$ hadoop fs -mv /user/hadoop/sample1.txt /user/text/
```

rm:

Remove a file from HDFS, similar to Unix rm command.
This command does not delete directories.
For recursive delete, use command **-rm -r**.

Options:

-r : Recursively remove directories and files
-skipTrash : To bypass trash and immediately delete the source
-f : Mention if there is no file existing
-rR : Recursively delete directories

```
$ hadoop fs -mv /user/hadoop/sample1.txt /user/text/
```

Hdfs package org.apache.Hadoop.io

Generic i/o code for use when reading and writing data to the network, to databases, and to files.

Interface Summary

RawComparator<T>	A **Comparator** that operates directly on byte representations of objects.
Stringifier<T>	Stringifier interface offers two methods to convert an object to a string representation and restore the object given its string representation.
Writable	A serializable object which implements a simple, efficient, serialization protocol, based on **DataInput** and **DataOutput**.
WritableComparable<T>	A **Writable** which is also **Comparable**.
WritableFactory	A factory for a class of Writable.

Class summary

Class	Description
AbstractMapWritable	Abstract base class for MapWritable and SortedMapWritable Unlike org.apache.nutch.crawl.MapWritable, this class allows creation of MapWritable<Writable, MapWritable> so the CLASS_TO_ID and ID_TO_CLASS maps travel with the class instead of being static.
ArrayFile	A dense file-based mapping from integers to values.
ArrayPrimitiveWritable	This is a wrapper class.
ArrayWritable	A Writable for arrays containing instances of a class.
BinaryComparable	Interface supported by **WritableComparable** types supporting ordering/permutation by a representative set of bytes.
BloomMapFile	This class extends **MapFile** and provides very much the same functionality.
BooleanWritable	A WritableComparable for booleans.
BytesWritable	A byte sequence that is usable as a key or value.
ByteWritable	A WritableComparable for a single byte.
CompressedWritable	A base-class for Writables which store themselves compressed and lazily inflate on field access.
DataOutputOutputStream	OutputStream implementation that wraps a DataOutput.
DefaultStringifier<T>	DefaultStringifier is the default implementation of the **Stringifier** interface which stringifies the objects using base64 encoding of the serialized version of the objects.
DoubleWritable	Writable for Double values.
ElasticByteBufferPool	This is a simple ByteBufferPool which just creates ByteBuffers as needed.
EnumSetWritable<E extends **Enum**<E>>	A Writable wrapper for EnumSet.
FloatWritable	A WritableComparable for floats.
GenericWritable	A wrapper for Writable instances.
IntWritable	A WritableComparable for ints.
IOUtils	An utility class for I/O related functionality.
LongWritable	A WritableComparable for longs.
MapFile	A file-based map from keys to values.
MapWritable	A Writable Map.
MD5Hash	A Writable for MD5 hash values.
NullWritable	Singleton Writable with no data.
ObjectWritable	A polymorphic Writable that writes an instance with it's class name.
SequenceFile	SequenceFiles are flat files consisting of binary key/value pairs.
SetFile	A file-based set of keys.
ShortWritable	A WritableComparable for shorts.
SortedMapWritable	A Writable SortedMap.
Text	This class stores text using standard UTF8 encoding.
TwoDArrayWritable	A Writable for 2D arrays containing a matrix of instances of a class.
VersionedWritable	A base class for Writables that provides version checking.
VIntWritable	A WritableComparable for integer values stored in variable-length format.
VLongWritable	A WritableComparable for longs in a variable-length format.
WritableComparator	A Comparator for **WritableComparable**s.
WritableFactories	Factories for non-public writables.

- **AbstractMapWritable**
- Abstract base class for MapWritable and SortedMapWritable Unlike org.apache.nutch.crawl.MapWritable, this class allows creation of MapWritable<Writable, MapWritable> so the CLASS_TO_ID and ID_TO_CLASS maps travel with the class instead of being static.
- **ArrayFile**
- A dense file-based mapping from integers to values.
- **ArrayPrimitiveWritable**
- This is a wrapper class.
- **ArrayWritable**
- A Writable for arrays containing instances of a class.

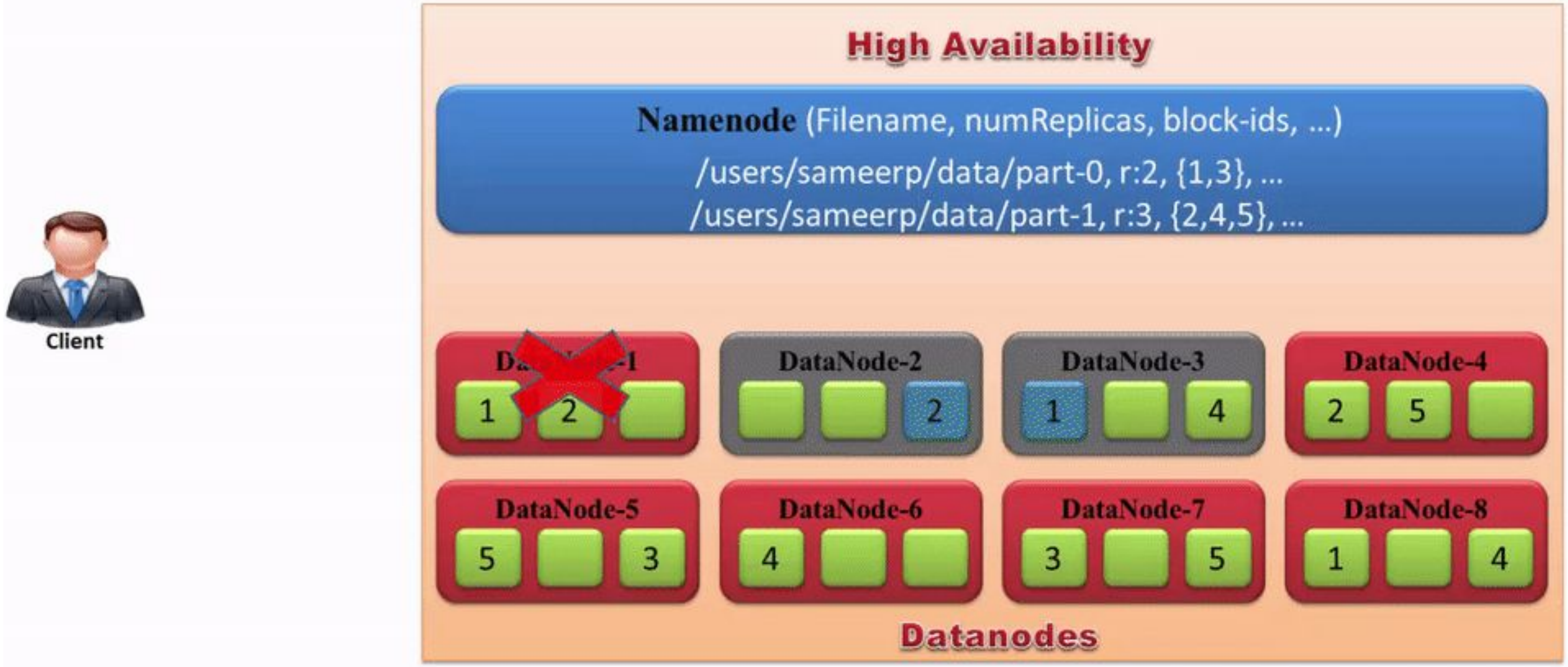
- **BinaryComparable**
- Interface supported by **WritableComparable** types supporting ordering/permutation by a representative set of bytes.
- **BloomMapFile**
- This class extends **MapFile** and provides very much the same functionality.
- **BooleanWritable**
- A WritableComparable for booleans.
- **BytesWritable**
- A WritableComparable for a single byte.

Exception

Exception	Description
MultipleIOException	Encapsulate a list of **IOException** into an **IOException**
VersionMismatchException	Thrown by **VersionedWritable.readFields(DataInput)** when the version of an object being read does not match the current implementation version as returned by **VersionedWritable.getVersion()**.

HDFS High Availability

- High availability refers to the availability of system or data in the wake of component failure in the system.
- High Availability was a new feature added to Hadoop 2.x to solve the Single point of failure problem in the older versions of Hadoop.
- As the Hadoop HDFS follows the master-slave architecture where the NameNode is the master node and maintains the filesystem tree.
- So HDFS cannot be used without NameNode.
- This NameNode becomes a bottleneck. HDFS high availability feature addresses this issue
- The high availability feature in Hadoop ensures the availability of the Hadoop cluster without any downtime, even in unfavourable conditions like NameNode failure, DataNode failure, machine crash, etc.



Client
High Availability
Namenode (Filename, numReplicas, block-ids, ...)
/users/sameerp/data/part-0, r:2, {1,3}, ...
/users/sameerp/data/part-1, r:3, {2,4,5}, ...
DataNode-1
1
2
DataNode-2
2
DataNode-3
1
4
DataNode-4
2
5
DataNode-5
5
3
DataNode-6
4
DataNode-7
3
5
DataNode-8
1
4
Datanodes

Features of HDFS

- Data replication
- Data integrity
- Data resilience

are the three key features of hdfs

Hdfs ensures data integrity throughout the cluster with help of following features

- Maintains transaction logs
- Validating checksum
- Creating data blocks