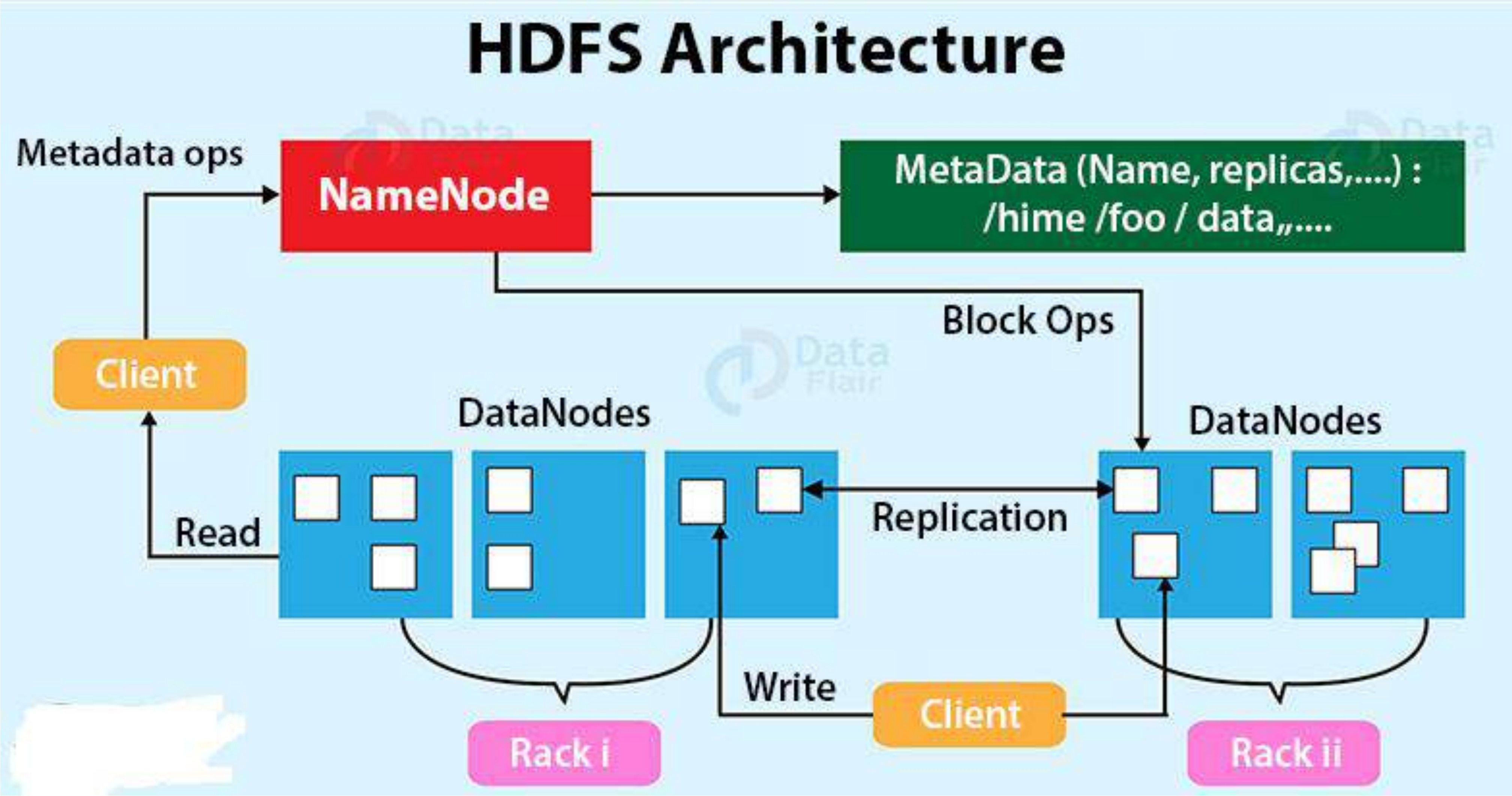
SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited with Grade "A" by NAAC | Approved by AICTE

UNIT - IV

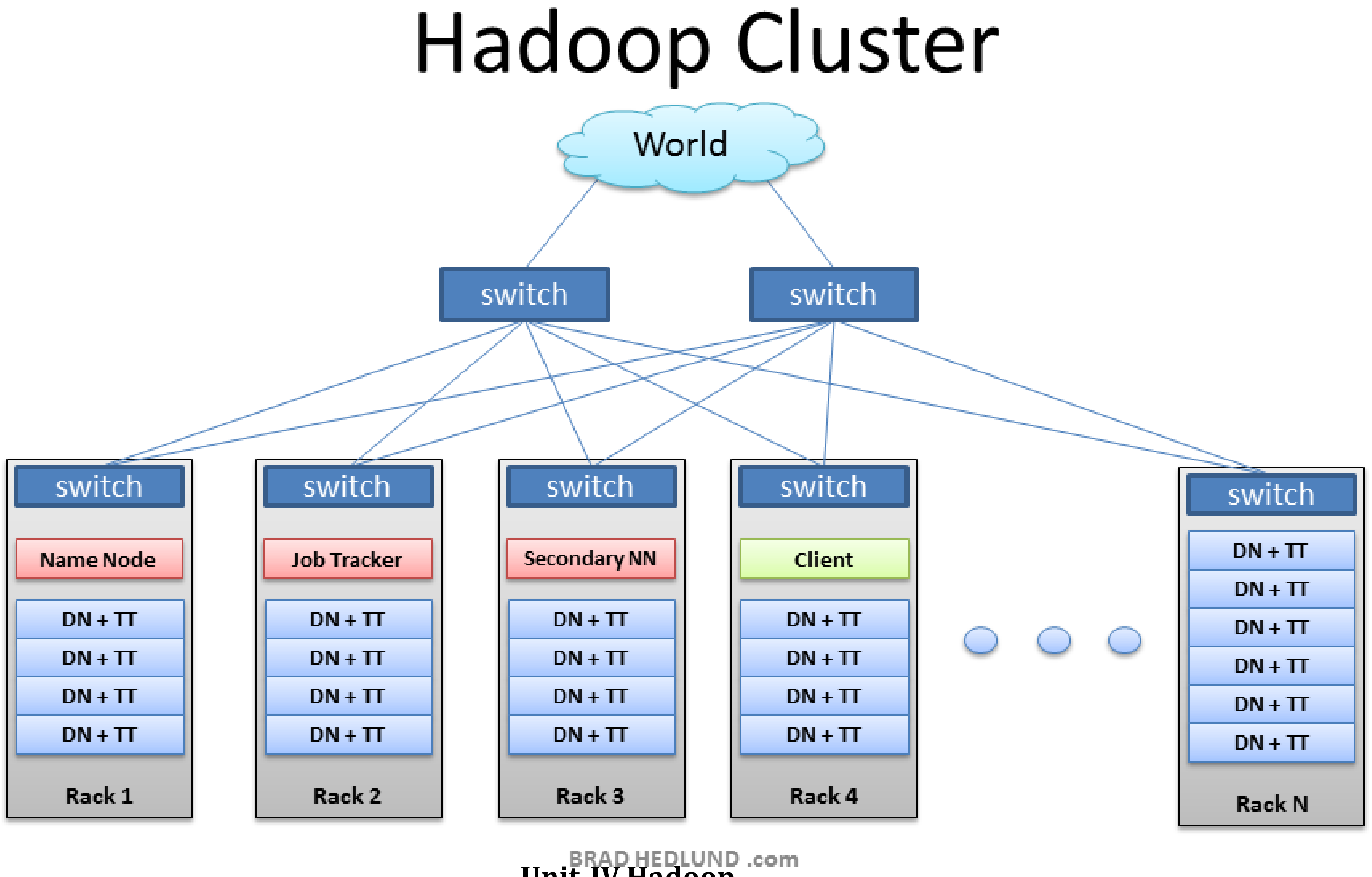
Analysing data with Hadoop – Scaling – Streaming

HDFS Architecture

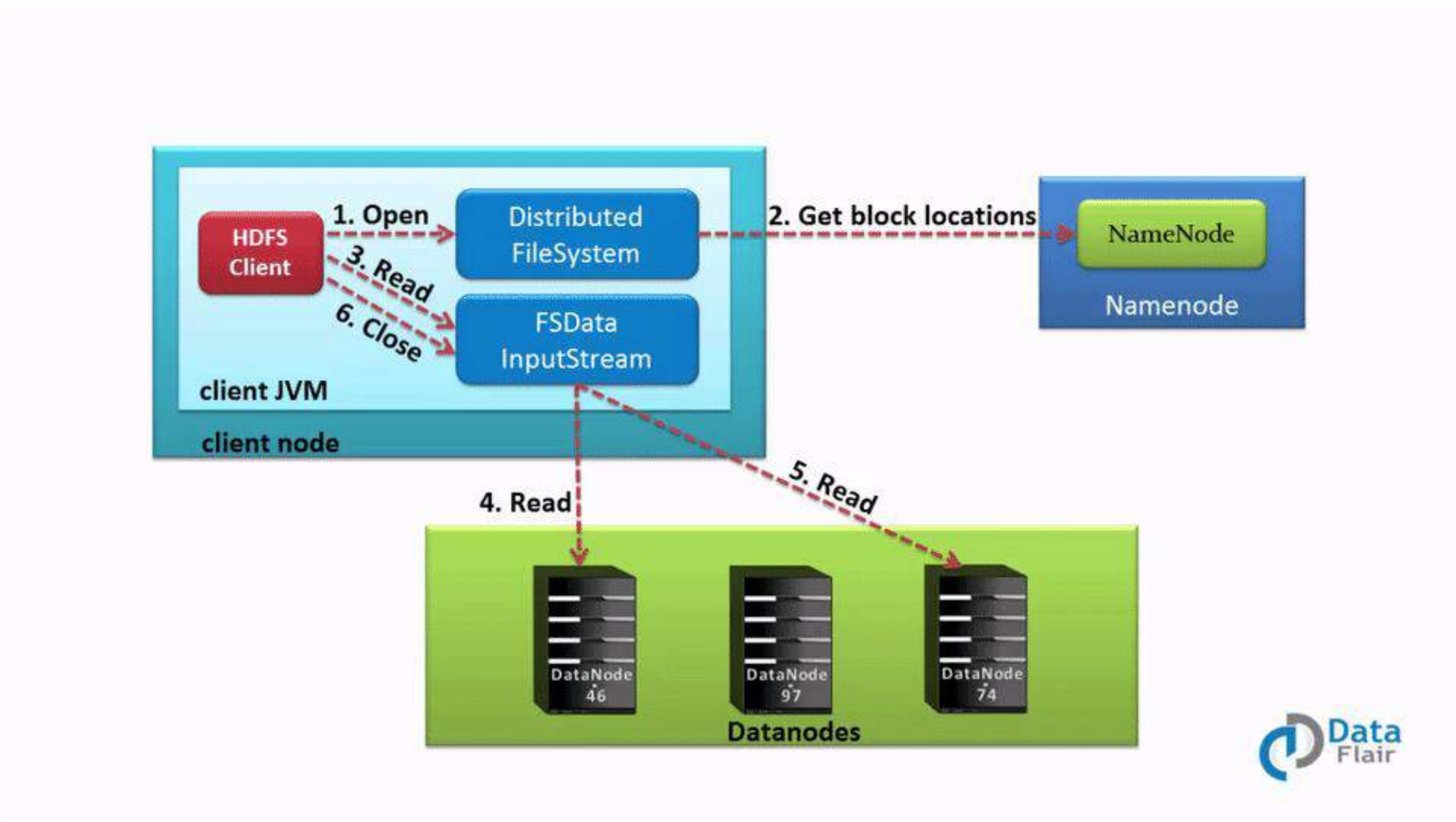


Hadoop Cluster

- Hadoop Distributed File System follows the **master-slave architecture**.
- Each cluster comprises a **single master node** and **multiple slave nodes**.



Hadoop HDFS Data Read Operation



Hadoop HDFS Data Read Operation

- Client opens the file it wishes to read by calling open() on the FileSystem object.
- *DistributedFileSystem* calls the namenode using RPC to determine the locations of the blocks for the first few blocks in the file.
- *DistributedFileSystem* returns a *FSDataInputStream* to the client for it to read data from.
- *FSDataInputStream*, thus, wraps the *DFSInputStream* which manages the datanode and namenode I/O.
- Client calls **read()** on the on the **FSDataInputStream** object.
- DFSInputStream which has stored the datanode addresses then connects to the closest datanode for the first block in the file.

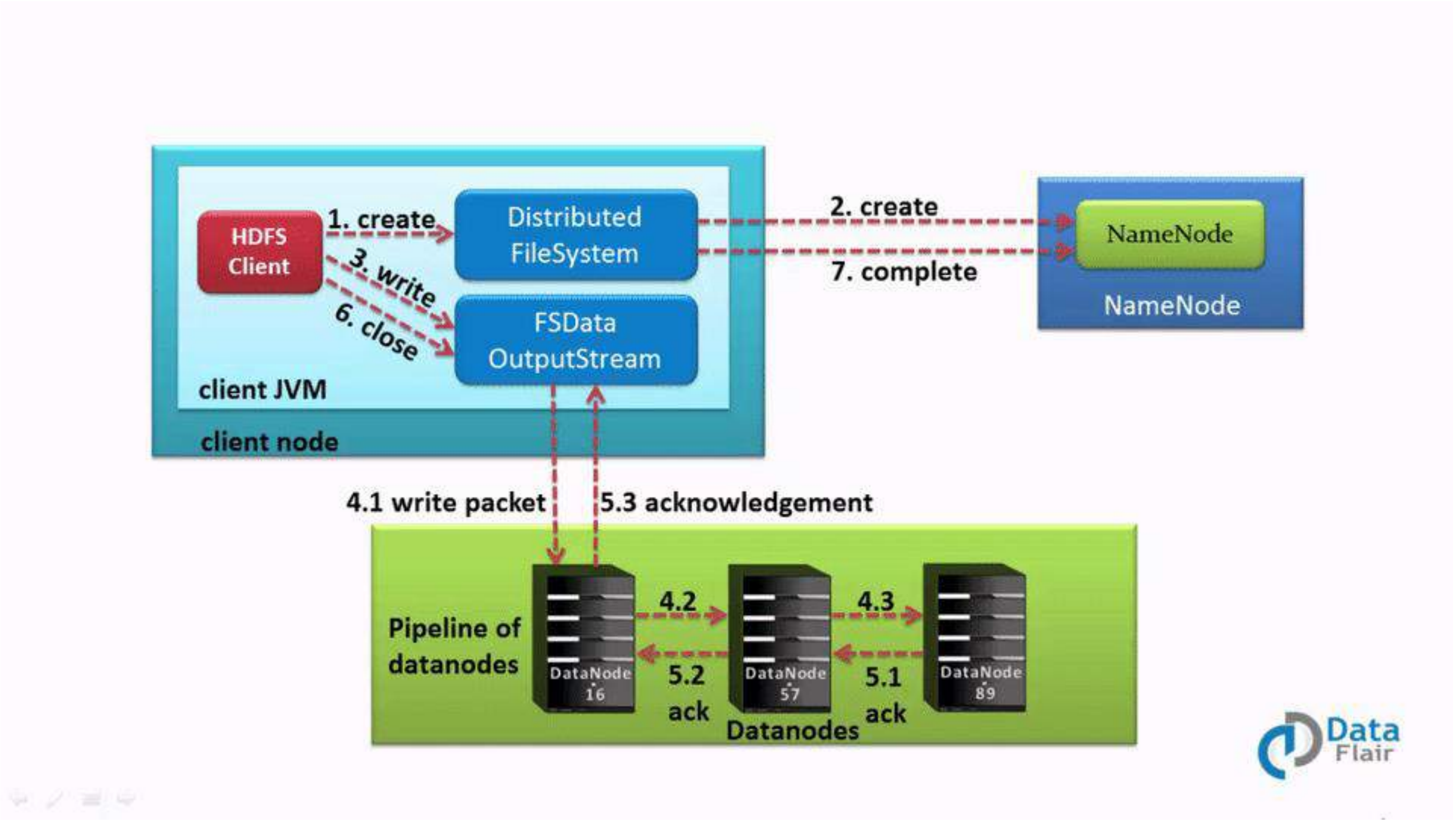
Hadoop HDFS Data Read Operation

- Data is streamed from the datanode back to the client, as a result client can call **read()** repeatedly on the stream.
- When the block ends, DFSInputStream will close the connection to the datanode and then finds the best datanode for the next block.
- If the *DFSInputStream* encounters an error while communicating with a datanode, it will try the next closest one for that block.
- When the client has finished reading the data, it calls **close()** on the stream.

Hadoop HDFS Data Read Operation

- The DFSInputStream, which contains the addresses for the first few blocks in the file, connects to the closest DataNode to read the first block in the file.
- When the client has finished reading the data, it calls **close()** on the **FSDataInputStream**.

Hadoop HDFS Data Write Operation



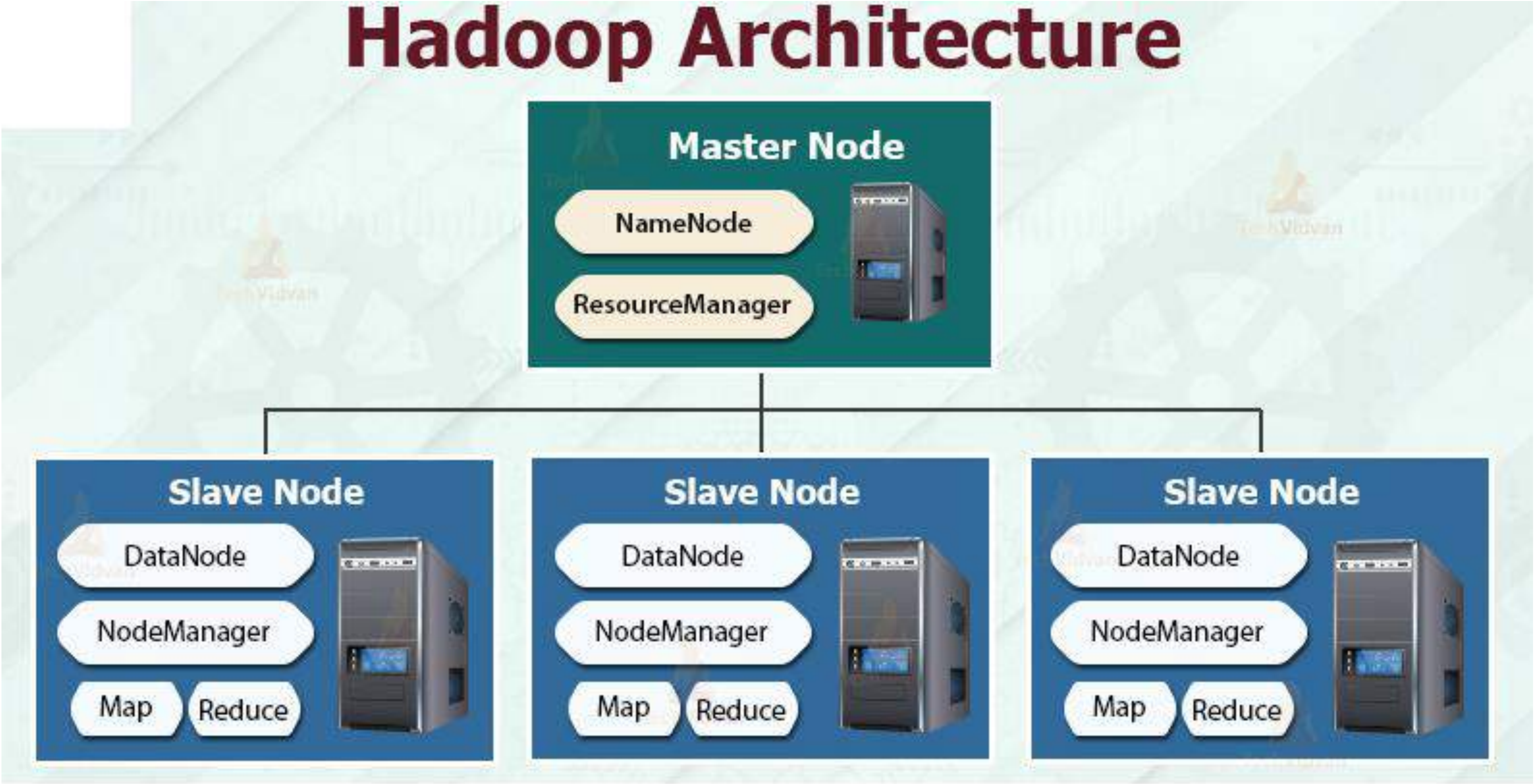
Hadoop HDFS Data Write Operation

- The HDFS client sends a **create** request on *DistributedFileSystem* APIs.
- *DistributedFileSystem* makes an RPC call to the namenode to create a new file in the file system's namespace.
- The namenode performs various checks to make sure that the file doesn't already exist
- When these checks pass, then only the namenode makes a record of the new file;
- otherwise, file creation fails and the client is thrown an *IOException.*
- The *DistributedFileSystem* returns a *FSDataOutputStream* for the client to start writing data.
- The list of datanodes form a pipeline, and here we'll assume the replication level is three, so there are three nodes in the pipeline.

Hadoop HDFS Data Write Operation

- The *DataStreamer* streams the packets to the first datanode in the pipeline.
- which stores the packet and forwards it to the second datanode in the pipeline.
- Similarly, the second datanode stores the packet and forwards it to the third (and last) datanode in the pipeline.
- *DFSOutputStream* also maintains an internal queue *ack queue*.
- The ack data packets are removed from the queue.
- When the client has finished writing data, it calls **close()** on the stream.
- This action flushes all the remaining packets to the datanode pipeline and waits for acknowledgments

Hadoop framework



Hadoop data types

Hadoop Data Types

Class	Size in bytes	Description	Sort Policy
BooleanWritable	1	Wrapper for a standard Boolean variable	False before and true after
ByteWritable	1	Wrapper for a single byte	Ascending order
DoubleWritable	8	Wrapper for a Double	Ascending order
FloatWritable	4	Wrapper for a Float	Ascending order
IntWritable	4	Wrapper for a Integer	Ascending order
LongWritable	8	Wrapper for a Long	Ascending order
Text	2GB	Wrapper to store text using the unicode UTF8 format	Alphabetic order
NullWritable		Placeholder when the key or value is not needed	Undefined
Your Writable		Implement the Writable Interface for a value or WritableComparable<T> for a key	Your sort policy

Hadoop data types : Serialization

- Serialization is the process of converting object data into byte stream data.
- **Deserialization:**
- Deserialization is the reverse process of serialization and converts byte stream data into object data for reading data from HDFS.
- Hadoop provides *Writables* for serialization and deserialization purpose.
- Hadoop provided two important interfaces Writable and WritableComparable.

org.apache.hadoop.io package

Hadoop HDFS Data Read Operation

- Writable and WritableComparable Interfaces To provide mechanisms for serialization and deserialization of data.

Writable interface specification is as follows:

```
package org.apache.hadoop.io;

import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

public interface Writable
{
 void write(DataOutput out) throws IOException;
 void readFields(DataInput in) throws IOException;
}
```

Hadoop HDFS Data Read Operation

- **WritableComparable** interface is subinterface of Hadoop's Writable and Java's Comparable interfaces. and its specification is shown below:

```
public interface WritableComparable extends Writable, Comparable
{
}
```

- The standard java.lang.*Comparable* Interface contains single method compareTo() method for comparing the operators passed to it.

```
public interface Comparable
{
        public int compareTo(Object obj);
}
```

Hadoop HDFS Data Read Operation

- The compareTo() method returns -1, 0 , or 1 depending on whether the compared object is less than, equal to, or greater than the current object.

Constraints on Key values in Mapreduce

- Hadoop data types used in Mapreduce for key or value fields must satisfy two constraints

1. Any data type used for a Value field in mapper or reducer input/output must implement Writable Interface
2. Any data type used for a Key field in mapper or reducer input/output must implement WritableComparable interface along with Writable interface to compare the keys of this type with each other for sorting purposes.

Writable Classes – Hadoop Data Types

- Hadoop provides classes that wrap the Java primitive types and implement the **WritableComparable** and **Writable** Interfaces.
- They are provided in the org.apache.hadoop.io package
- All the Writable wrapper classes have a get() and a set() method for retrieving and storing the wrapped value.

Primitive Writable Classes

- They hold a single primitive value that can be set either at construction or via a setter method.
- All these primitive writable wrappers have get() and set() methods to read or write the wrapped value.
- Below is the list of primitive writable data types available in Hadoop.
  - BooleanWritable
  - ByteWritable
  - IntWritable - size of IntWritable is 4 bytes
  - VIntWritable - used for variable length Integer types
  - FloatWritable
  - LongWritable - size of LongWritable is 4 bytes
  - VLongWritable - variable length long types
  - DoubleWritable

Primitive Writable Classes

- In the above list VIntWritable and VLongWritable are used for variable length Integer types and variable length long types respectively.

Array Writable Classes

- Hadoop provided two types of array writable classes, one for *singledimensional* and another for *twodimensional* arrays.
- Supported in IntWritable or LongWritable only but not the java native data types like int or float.
  - ArrayWritable
  - TwoDArrayWritable

Map Writable Classes

- Hadoop provided below MapWritable data types which implement java.util.Map interface
- AbstractMapWritable – This is abstract or base class for other MapWritable classes.
- MapWritable – This is a general purpose map mapping Writable keys to Writable values.
- SortedMapWritable – This is a specialization of the MapWritable class that also implements the SortedMap interface.

Other Writable Classes

<u>NullWritable</u>

- NullWritable is a special type of Writable representing a null value.
- No bytes are read or written when a data type is specified as NullWritable.
- In Mapreduce, a key or a value can be declared as a NullWritable when we don’t need to use that field

<u>ObjectWritable</u>

- This is a general purpose generic object wrapper.
- It support to store any objects like Java primitives, String, Enum, Writable, null, or arrays.

Other Writable Classes

Text

- Text can be used as the Writable equivalent of java.lang.String and It’s max size is 2 GB.
- Unlike java’s String data type, Text is mutable in Hadoop.

BytesWritable

- BytesWritable is a wrapper for an array of binary data.

Other Writable Classes

GenericWritable

- It is similar to ObjectWritable but supports only a few types. User need to subclass this
- GenericWritable class and need to specify the types to support. Example Program to Test Writables

Hadoop – MapReduce Paradigms

What is MapReduce?

- Data-parallel programming model for clusters of commodity machines

- Pioneered by Google
  - Processes 20 PB of data per day
- Popularized by open-source Hadoop project
  - Used by Yahoo!, Facebook, Amazon, …

What is MapReduce used for?

- At Google:
  - Index building for Google Search
  - Article clustering for Google News
  - Statistical machine translation
- At Yahoo!:
  - Index building for Yahoo! Search
  - Spam detection for Yahoo! Mail
- At Facebook:
  - Data mining
  - Ad optimization
  - Spam detection

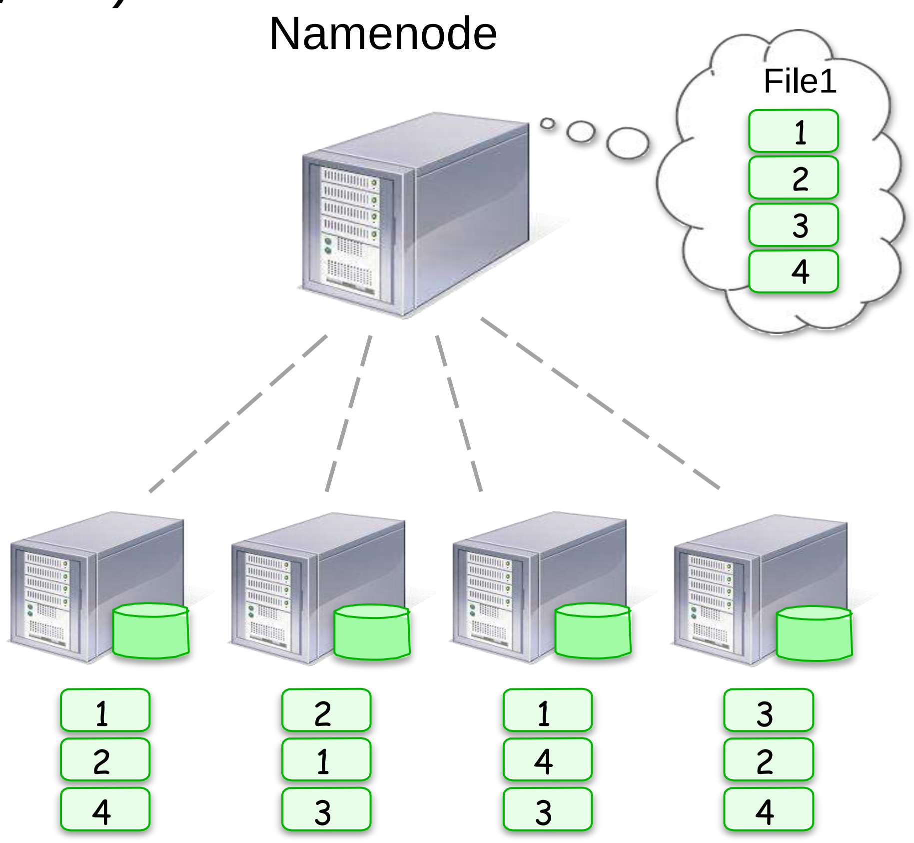
MapReduce Goals

1. **Scalability** to large data volumes:
   – Scan 100 TB on 1 node @ 50 MB/s = 24 days
   – Scan on 1000-node cluster = 35 minutes

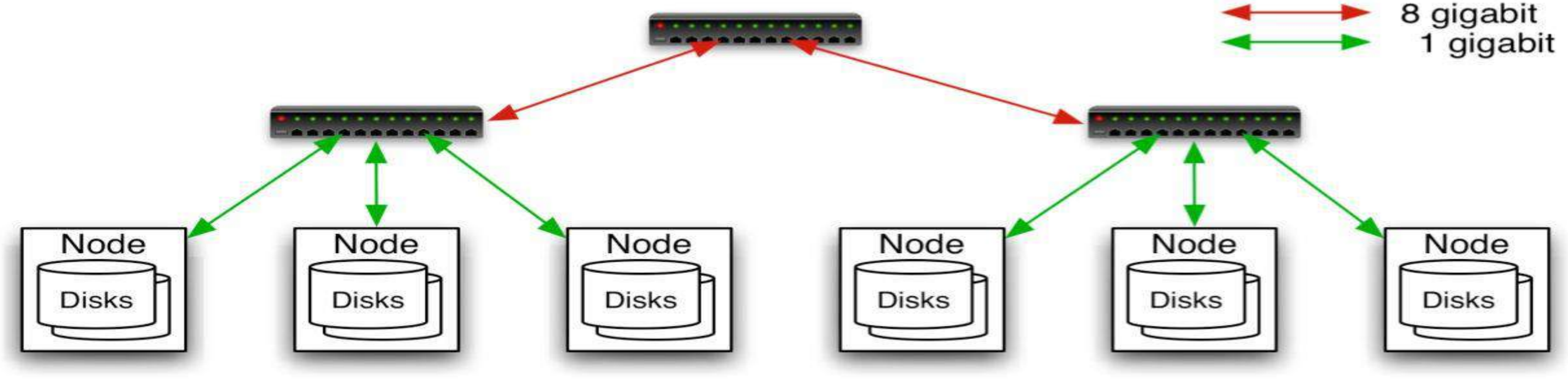
2. **Cost-efficiency:**
   – Commodity nodes (cheap, but unreliable)
   – Commodity network
   – Automatic fault-tolerance (fewer admins)
   – Easy to use (fewer programmers)

Hadoop Distributed File System

- Files split into 128MB *blocks*
- Blocks replicated across several *datanodes* (usually 3)
- *Namenode* stores metadata (file names, locations, etc)
- Optimized for large files, sequential reads
- Files are append-only



Typical Hadoop Cluster



- 40 nodes/rack, 1000-4000 nodes in cluster
- 1 GBps bandwidth in rack, 8 GBps out of rack
- Node specs (Yahoo! terasort):
  - 8 x 2.0 GHz cores, 8 GB RAM, 4 disks (= 4 TB?)

Typical Problems Solved by MR

- Read a lot of data
- Map: extract something you care about from each record
- Shuffle and Sort
- Reduce: aggregate, summarize, filter, transform
- Write the results

Hadoop – MapReduce Paradigms

- MapReduce is a framework used to process huge amounts of data in parallel.
- MapReduce is a processing technique and a program model for distributed computing based on java.
- The MapReduce algorithm has two tasks
  - Map
  - Reduce.
- Map takes a set of data and converts it into another set of data.
- Individual elements are broken down into tuples (key/value pairs)

MapReduce Paradigm

- Programming model developed at Google
- Sort/merge based distributed computing
- Initially, it was intended for their internal search/indexing application, but now used extensively by more organizations (e.g., Yahoo, Amazon.com, IBM, etc.)
- It is functional style programming (e.g., LISP) that is naturally parallelizable across a large cluster of workstations or PCS.
- **The underlying system takes care of the partitioning of the input data, scheduling the program's execution across several machines, handling machine failures, and managing required inter-machine communication. (This is the key for Hadoop's success)**

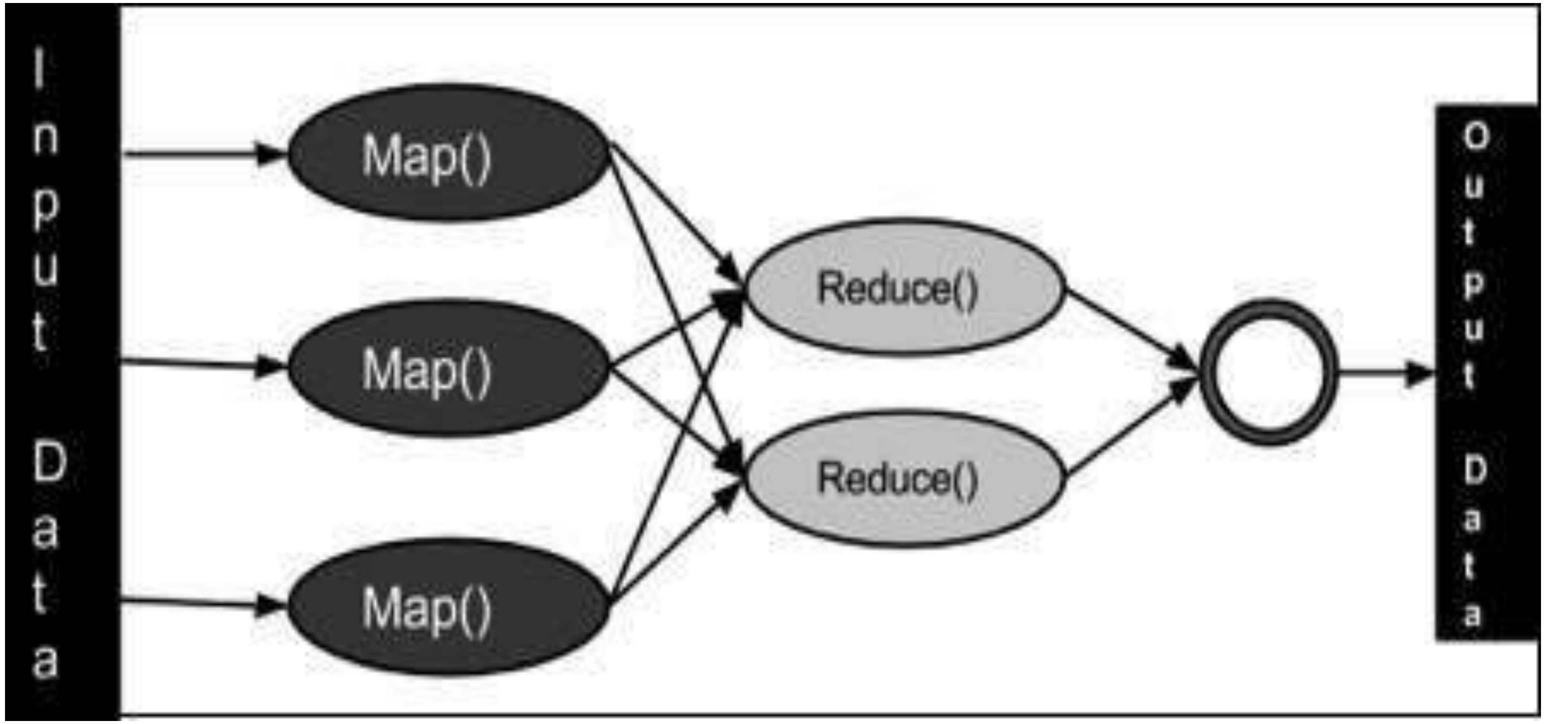
Hadoop – MapReduce Paradigms

- MapReduce program executes in three stages, namely map stage, shuffle stage, and reduce stage.
- **Map stage**
  - The map or mapper’s job is to process the input data
  - The input file is passed to the mapper function line by line.
  - The mapper processes the data and creates several small chunks of data.
- **Reduce stage**
  - This stage is the combination of the **Shuffle** stage and the **Reduce** stage.
  - The Reducer’s job is to process the data that comes from the mapper.
  - After processing, it produces a new set of output, which will be stored in the HDFS.

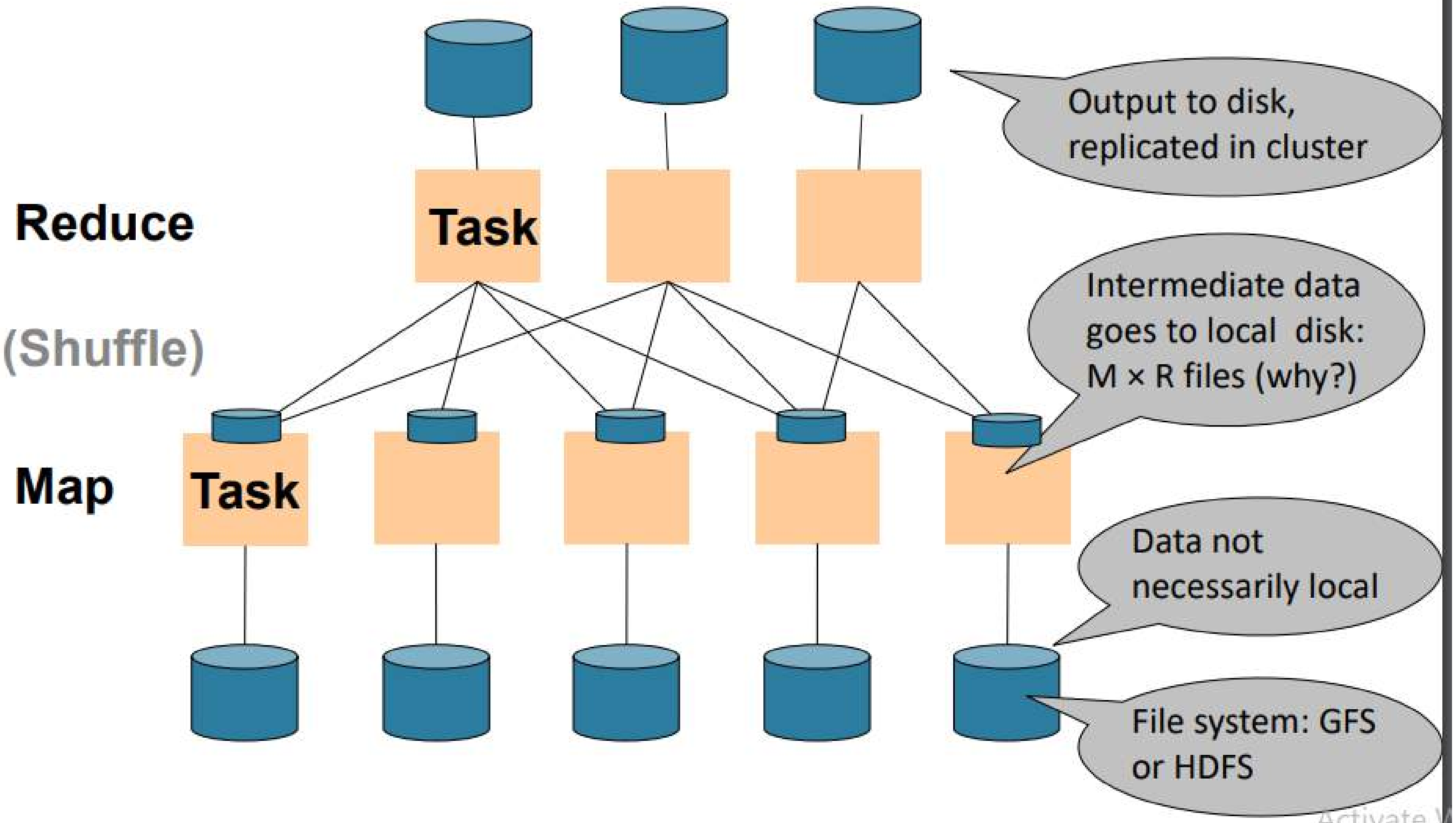
Hadoop – MapReduce Paradigms

- During a MapReduce job, Hadoop sends the Map and Reduce tasks to the appropriate servers in the cluster.
- The framework manages all the details of data-passing such as issuing tasks, verifying task completion, and copying data around the cluster between the nodes.
- Most of the computing takes place on nodes with data on local disks that reduces the network traffic.
- After completion of the given tasks, the cluster collects and reduces the data to form an appropriate result, and sends it back to the Hadoop server.

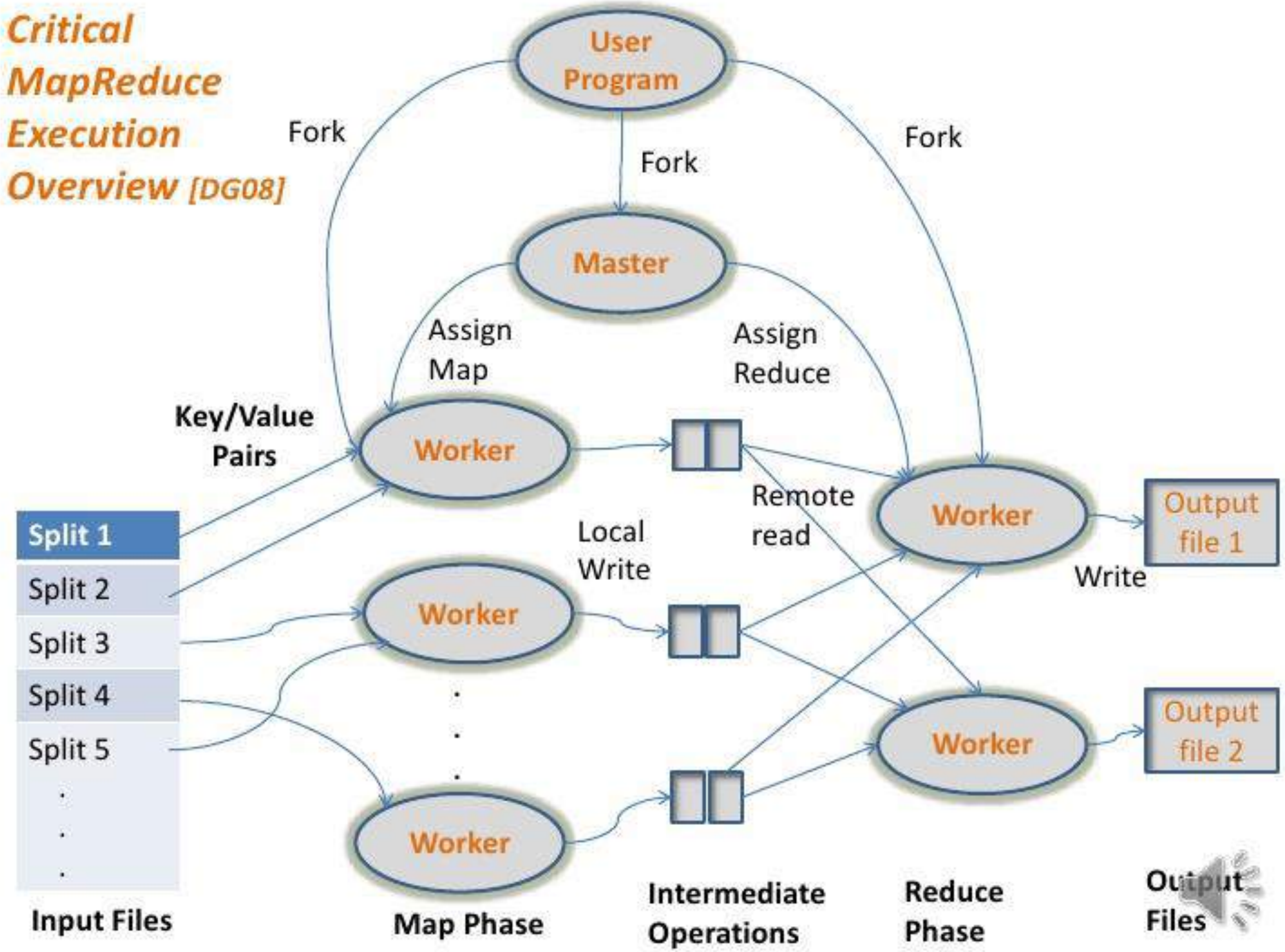
Hadoop – MapReduce Paradigms



MapReduce Execution Details



MapReduce Execution



Data Model

- Files!

- A file = a bag of (key, value) pairs

A MapReduce program:

- Input: a bag of (inputkey, value) pairs
- Output: a bag of (outputkey, value) pairs

Inputs and Outputs (Java Perspective)

- The MapReduce framework operates on <key, value> pairs.
- input to the job as a set of <key, value> pairs and produces a set of <key, value> pairs as the output of the job.
- key and the value classes should be in serialized manner , so implement Writable interface.
- The key classes have to implement the Writable-Comparable interface to facilitate sorting by the framework.

	Input	Output
Map	<k1, v1>	list (<k2, v2>)
Reduce	<k2, list(v2)>	list (<k3, v3>)

Example MapReduce: To count the occurrences of words in the given set of documents

map(String key, String value):

// key: document name; value: document contents; map (k1,v1) → list(k2,v2)

for each word w in value: EmitIntermediate(w, "1");

(Example: If input string is (“Saibaba is God. I am I”), Map produces

{<“Saibaba”,1”>, <“is”, 1>, <“God”, 1>, <“I”,1>, <“am”,1>,<“I”,1>}

reduce(String key, Iterator values):

// key: a word; values: a list of counts; reduce (k2,list(v2)) → list(v2)

int result = 0;

for each v in values:

result += ParseInt(v);

Emit(AsString(result));

(Example: reduce(“I”, <1,1>) → 2)

How does MapReduce work?

- The run time partitions the input and provides it to different Map instances;
- Map (key, value) → (key’, value’)
- The run time collects the (key’, value’) pairs and distributes them to several Reduce functions so that each Reduce function gets the pairs with the same key’.
- Each Reduce produces a single (or zero) file output.
- Map and Reduce are user written functions

Terminology

- **PayLoad** − Applications implement the Map and the Reduce functions, and form the core of the job.
- **Mapper** − Mapper maps the input key/value pairs to a set of intermediate key/value pair.
- **NamedNode** − Node that manages the Hadoop Distributed File System (HDFS).
- **DataNode** − Node where data is presented in advance before any processing takes place.
- **MasterNode** − Node where JobTracker runs and which accepts job requests from clients.
- **SlaveNode** − Node where Map and Reduce program runs.
- **JobTracker** − Schedules jobs and tracks the assign jobs to Task tracker.
- **Task Tracker** − Tracks the task and reports status to JobTracker.

Hadoop – MapReduce Paradigms

- **Job** − A program is an execution of a Mapper and Reducer across a dataset.
- **Task** − An execution of a Mapper or a Reducer on a slice of data.
- **Task Attempt** − A particular instance of an attempt to execute a task on a SlaveNode.

Fault Tolerance

- MapReduce handles fault tolerance by writing intermediate files to disk:
  - Mappers write file to local disk
  - Reducers read the files (=reshuffling); if the server fails, the reduce task is restarted on another server

Q&A

1. Why Map Reduce
   - Scale data processing over multiple computing nodes.
2. What is map reduce
   - Data-parallel programming model for clusters of commodity machines