Hadoop Distributed File System (HDFS)

Mahesh G Huddar

Asst. Professor

Dept. of CSE, HIT, Nidasoshi

Design Features of Hadoop Distributed File System (HDFS)

Mahesh G Huddar

Asst. Professor

Dept. of CSE, HIT, Nidasoshi

Introduction

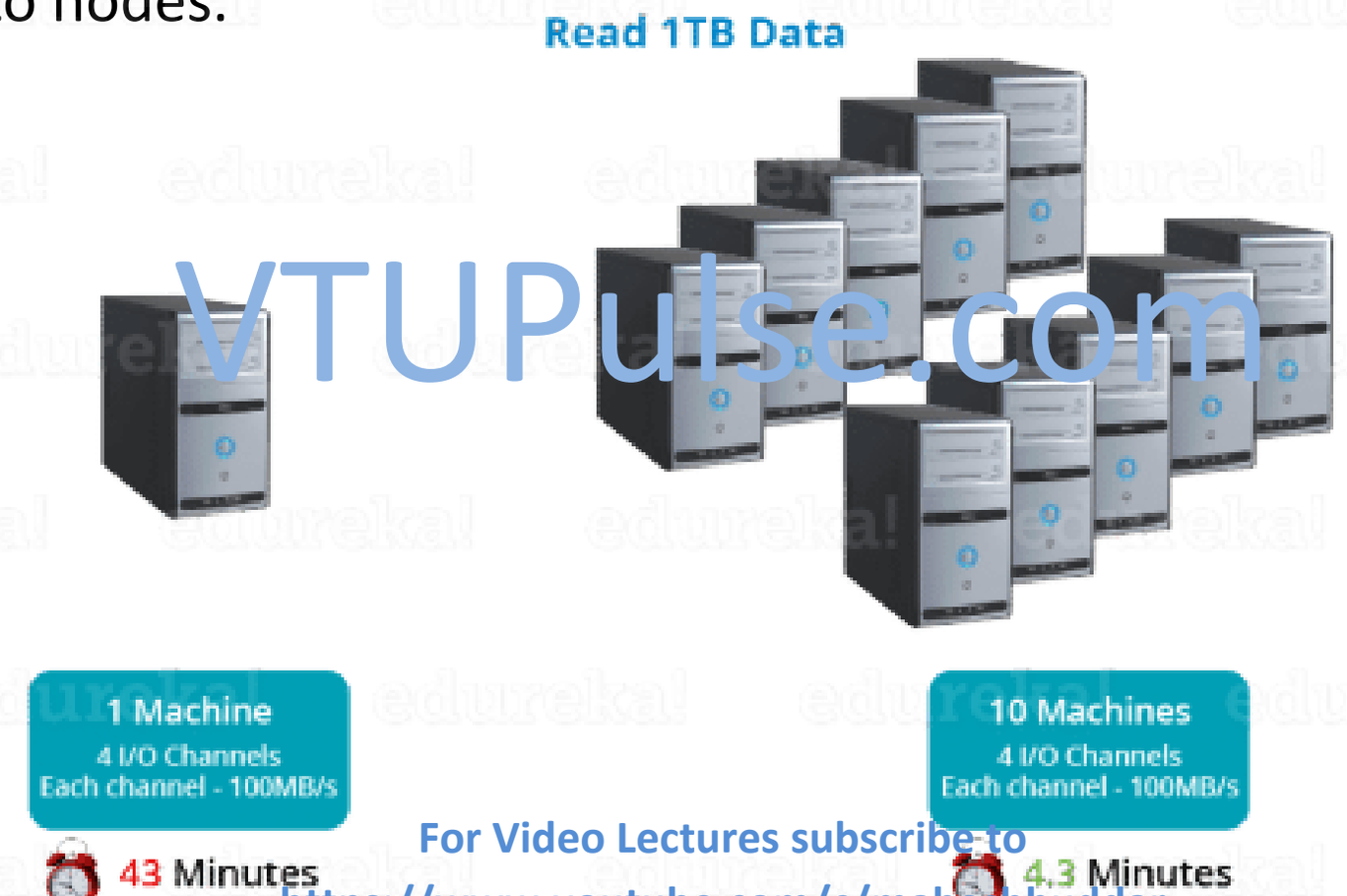
- The Hadoop Distributed File System (HDFS) was designed for Big Data processing.
- It is a core part of Hadoop which is used for data storage.
- It is designed to run on commodity hardware.

Features of HDFS

- Distributed
- Parallel Computation
- Replication
- Fault tolerance
- Streaming Data Access
- Portable

Features of HDFS

- **Distributed and Parallel Computation -** This is one of the most important features of HDFS that makes Hadoop very powerful. Here, data is divided into multiple blocks and stored into nodes.



Features of HDFS

- **Distributed and Parallel Computation**.
- Let's understand this concept with an example. Suppose, it takes 43 minutes to process 1 TB file on a single machine.
- So, how much time will it take to process the same 1 TB file when you have 10 machines in a Hadoop cluster with similar configuration – 43 minutes or 4.3 minutes? 4.3 minutes, Right! What happened here? Each of the nodes is working with a part of the 1 TB file in parallel.
- Therefore, the work which was taking 43 minutes before, gets finished in just 4.3 minutes now as the work got divided over ten machines.

Features of HDFS

- **Highly Scalable -** HDFS is highly scalable as it can scale hundreds of nodes in a single cluster.

Features of HDFS

- 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.

Features of HDFS

- 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.

Features of HDFS

- **Replication -** Due to some unfavorable conditions, the node containing the data may be failed to work. So, to overcome such problems, HDFS always maintains the copy of data on a different machine.
- **Fault tolerance -** In HDFS, the fault tolerance signifies the robustness of the system in the event of failure. The HDFS is highly fault-tolerant that if any machine fails, the other machine containing the copy of that data automatically become active.
- **Portable -** HDFS is designed in such a way that it can easily portable from platform to another.
- **Streaming Data Access:** The write-once/read-many design is intended to facilitate streaming reads.

Components and Architecture Hadoop Distributed File System (HDFS)

Mahesh G Huddar

Asst. Professor

Dept. of CSE, HIT, Nidasoshi

HDFS Components

- The design of HDFS is based on two types of nodes: a **NameNode** and multiple **DataNodes**.
- No data is actually stored on the NameNode.
- A single NameNode manages all the metadata needed to store and retrieve the actual data from the DataNodes.
- For a minimal Hadoop installation, there needs to be a single NameNode daemon and a single DataNode daemon running on at least one machine

HDFS Components

- The design of HDFS follows a master/slave architecture
- Master Node is NameNode and Slave Node is DataNode.
- The master node (NameNode) manages the file system namespace and regulates access to files by clients.
- File system namespace operations such as opening, closing, and renaming files and directories are all managed by the NameNode.
- The NameNode also determines the mapping of blocks to DataNodes and handles DataNode failures.

HDFS Components

- The slaves (DataNodes) are responsible for serving read and write requests from the file system to the clients. The NameNode manages block creation, deletion, and replication.
- When a client wants to writes data, it first communicates with the NameNode and requests to create a file.
- The NameNode determines how many blocks are needed and provides the client with the DataNodes that will store the data.
- As part of the storage process, the data blocks are replicated after they are written to the assigned node.

HDFS Components

- Reading data happens in a similar fashion.
- The client requests a file from the NameNode, which returns the best DataNodes from which to read the data. The client then accesses the data directly from the DataNodes.
- Thus, once the metadata has been delivered to the client, the NameNode steps back and lets the conversation between the client and the DataNodes proceed.
- While data transfer is progressing, the NameNode also monitors the DataNodes by listening for heartbeats sent from DataNodes.
- The lack of a heartbeat signal indicates a potential node failure. In such a case, the NameNode will route around the failed DataNode and begin re-replicating the now-missing blocks.

HDFS Components

- The mappings between data blocks and the physical DataNodes are not kept in persistent storage on the NameNode.
- For performance reasons, the NameNode stores all metadata in memory.
- Upon startup, each DataNode provides a block report to the NameNode.
- The block reports are sent every 10 heartbeats. (The interval between reports is a configurable property.)
- The reports enable the NameNode to keep an up-to-date account of all data blocks in the cluster.

HDFS Components

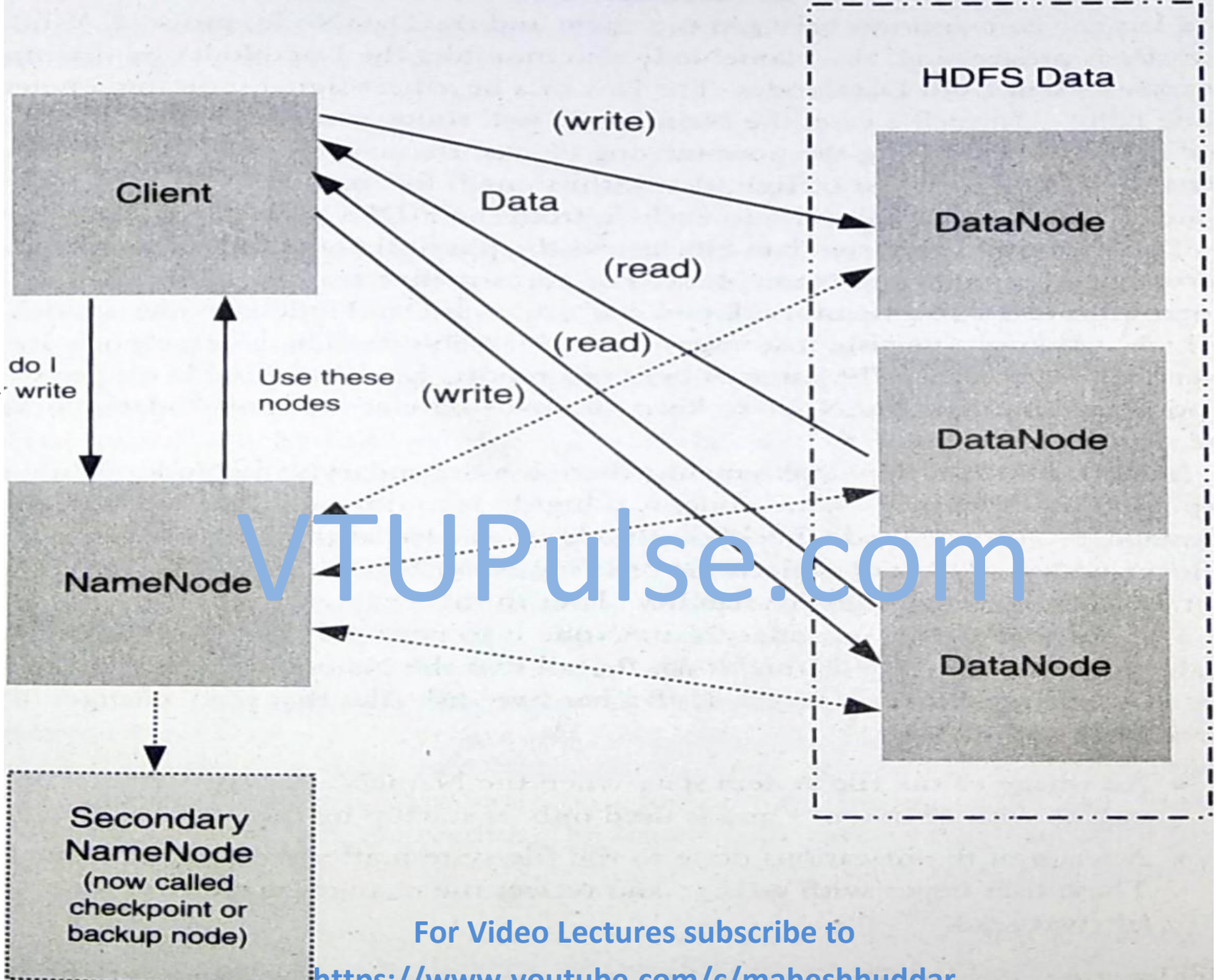
- In almost all Hadoop deployments, there is a SecondaryNameNode.
- While not explicitly required by a NameNode, it is highly recommended.
- The term "SecondaryNameNode" (now called CheckPointNode) is somewhat misleading.
- It is not an active failover node and cannot replace the primary NameNode in case of its failure.

HDFS Components

- The purpose of the SecondaryNameNode is to perform periodic checkpoints that evaluate the status of the NameNode.
- As the NameNode keeps all system metadata memory for fast access. It also has two disk files that track changes to the metadata:
  - An image of the file system state when the NameNode was started. This file begins with fsimage_* and is used only at startup by the NameNode.
  - A series of modifications done to the file system after starting the NameNode. These files begin with edit_* and reflect the changes made after the file was read.
- The location of these files is set by the dfs.namenode.name.dir property in the hdfs-site.xml file.

HDFS Components

- The SecondaryNameNode periodically downloads fsimage and edits files, joins them into a new fsimage, and uploads the new fsimage file to the NameNode.
- Thus, when the NameNode restarts, the fsimage file is reasonably up-to-date and requires only the edit logs to be applied since the last checkpoint.
- If the SecondaryNameNode were not running, a restart of the NameNode could take a long time due to the number of changes to the file system.



HDFS Components

Thus, the various roles in HDFS can be summarized as follows:

1. HDFS uses a master/slave model designed for large file reading/streaming.
2. The NameNode is a metadata server or "data traffic cop."
3. HDFS provides a single namespace that is managed by the NameNode.
4. Data is redundantly stored on DataNodes; there is no data on the NameNode.
5. The SecondaryNameNode performs checkpoints of NameNode file system's state but is not a failover node.

Replication
Hadoop Distributed File System (HDFS)

Mahesh G Huddar

Asst. Professor

Dept. of CSE, HIT, Nidasoshi

HDFS Block Replication

- The design of HDFS is based on two types of nodes: a **NameNode** and multiple **DataNodes**.
- For Example:
  - Assume there are 4 DataNodes
  - Client wants to write 150MB of Data
  - NameNode divides the data into 3 blocks (maximum block size is 64MB)
  - First Block is 64 MB, second is 64MB and third is 22MB
  - NameNode Assigns DataNode1 and DataNode2 to client to store the data.

HDFS Block Replication

DataNode	Data Block
1	1 and 2
2	3
3	
4	

HDFS Block Replication

DataNode	Data Block
1	1 and 2
2	3
3	1
4	2 and 3

HDFS Block Replication

- When a client wants to writes data, it first communicates with the NameNode and requests to create a file.
- The NameNode determines how many blocks are needed and provides the client with the DataNodes that will store the data.
- As part of the storage process, the data blocks are replicated after they are written to the assigned node.

HDFS Block Replication

Rack	DataNode	Data Block
1	1	1 and 2
	2	
	3	3
	4	

HDFS Block Replication

Rack	DataNode	Data Block
1	1	1 and 2
	2	1
	3	3 and 2
	4	3

HDFS Block Replication

Rack	DataNode	Data Block
1	1	1 and 2
	2	
2	3	3
	4	

HDFS Block Replication

Rack	DataNode	Data Block
1	1	1 and 2
	2	3
2	3	3 and 1
	4	2

HDFS Block Replication

- NameNode will attempt to write replicas of the data blocks on nodes that are in other separate racks (if possible).
- If there is only one rack, then the replicated blocks are written to other servers in the same rack.
- After the DataNode acknowledges that the file block replication is complete, the client closes the file and informs the NameNode that the operation is complete.
- Note that the NameNode does not write any data directly to the DataNodes.
- However, it gives the client a limited amount of time to complete the operation. If it does not complete in the time period, the operation is canceled.

HDFS Block Replication

- When HDFS writes a file, it is replicated across the cluster.
- For Hadoop clusters containing more than eight DataNodes, the replication value is usually set to 3.
- In a Hadoop cluster of eight or fewer DataNodes but more than one DataNode, a replication factor of 2 is adequate.
- The amount of replication is based on the value of dfs.replication in the hdfs-site.xml file.
- This default value can be overruled with the hdfs dfs-setrep command.

HDFS Block Replication

- In a typical operating system, the block size is 4KB or 8KB.
- The HDFS default block size is often 64MB (Maximum).
- The HDFS default block size is not the minimum block size.
- If a 20KB file is written to HDFS, it will create a block that is approximately 20KB in size.
- If a file of size 80MB is written to HDFS, a 64MB block and a 16MB block will be created.

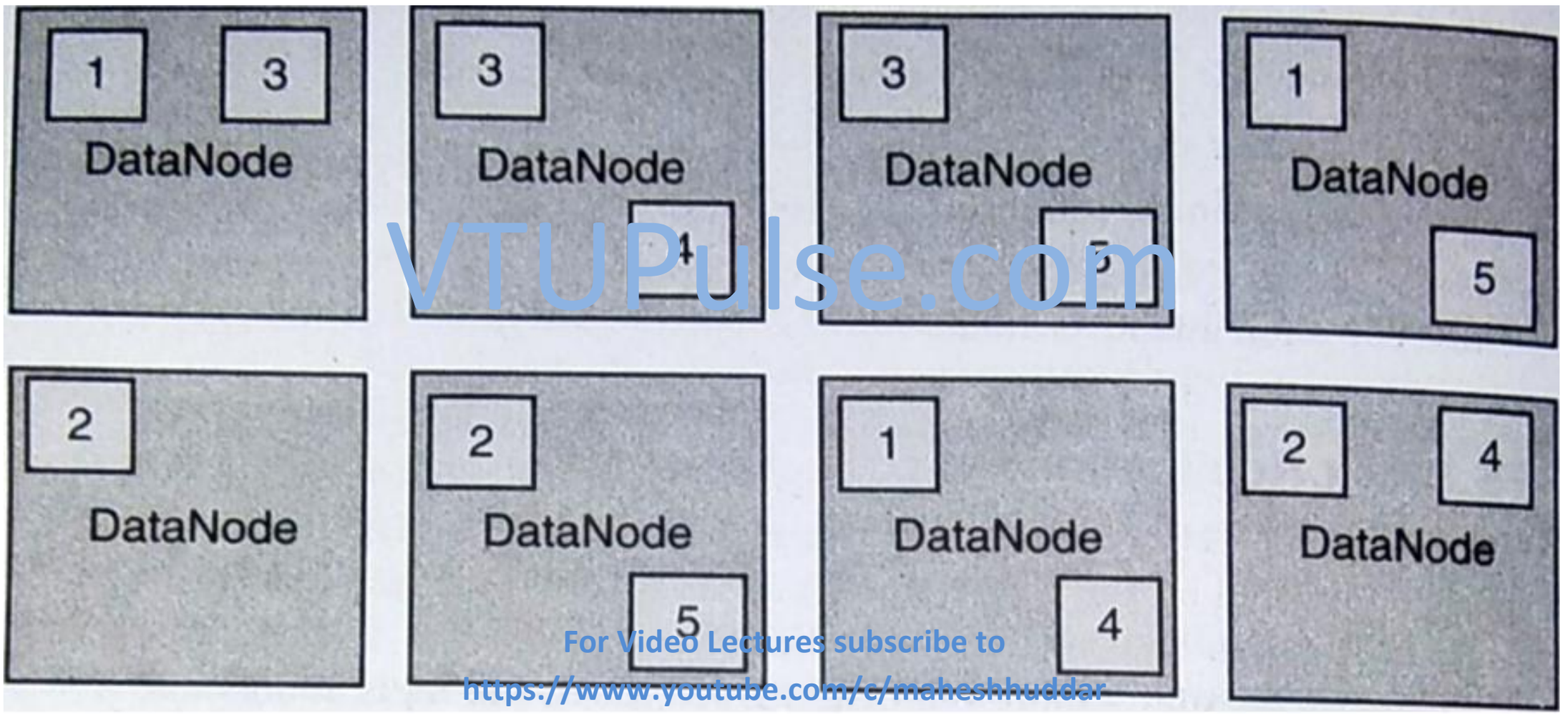
HDFS Block Replication

- Figure provides an example of how a file is broken into blocks and replicated across the cluster.
- A replication factor of 3 ensures that any one DataNode can fail and the replicated blocks will be available on other nodes—and then subsequently re-replicated on other DataNodes.

HDFS Block Replication

- Data File (64MB Blocks)

1	2	3	4	5



Hadoop Distributed File System (HDFS)

Mahesh G Huddar

Asst. Professor

Dept. of CSE, HIT, Nidasoshi

Hadoop Distributed File System (HDFS)

- Safe Mode
- Rack Awareness
- High NameNode Availability
- NameNode Federation
- Checkpoint and Backup
- Snapshots

HDFS Safe Mode

- What is Safe Mode…?
- When the NameNode starts, it enters a read-only safe mode where blocks cannot be added, replicated or deleted.
- Safe Mode enables the NameNode to perform two important processes:
  - The previous file system state is reconstructed by loading the fsimage file into memory and replaying the edit log.
  - The mapping between blocks and data nodes is created by waiting for enough of the DataNodes to register so that at least one copy of the data is available. Not all DataNodes are required to register before HDFS exits from Safe Mode. The registration process may continue for some time.

HDFS Safe Mode

- When there is a file system issue that must be addressed by the administrator by entering into safe mode manually by using command:

hdfs dfsadmin-safemode

Rack Awareness

- Rack awareness deals with data locality.
- Assume that there are 6 DataNode's in the haddop cluster

Rack	DataNode
1	1
	2
	3
	4
	5
	6

Rack	DataNode
1	1
	2
	3
2	4
	5
	6

Rack	DataNode
1	1
	2
2	3
	4
3	5
	6

Rack Awareness

- A typical Hadoop cluster will exhibit three levels of data locality:
  - Data resides on the local machine (best).
    - Best performance but suffers from single point of failure
  - Data resides in the same rack (better).
    - Better performance but suffers from single point of failure
  - Data resides in a different rack (good).
    - Good performance

Rack Awareness

- The NameNode tries to place replicated data blocks on multiple racks for improved fault tolerance.
- In such a case, an entire rack failure will not cause data loss or stop HDFS from working.
- However performance may be degraded.
- A default Hadoop installation assumes all the nodes belong to the same (large) rack.

NameNode High Availability

- With early Hadoop installations, the NameNode was a single point of failure that could bring down the entire Hadoop cluster.
- NameNode hardware often employed redundant power supplies and storage to guard against such problems, but it was still susceptible to other failures.
- The solution was to implement NameNode High Availability (HA) as a means to provide true failover service.
- An NameNode High Availability Hadoop cluster has two (or more) separate NameNode machines.
- Each machine is configured with exactly the same software.

NameNode High Availability

- One of the NameNode machines is in the Active state, and the other is in the Standby state.
- Like a single NameNode cluster, the Active NameNode is responsible for all client HDFS operations in the cluster.
- The Standby NameNode maintains enough state to provide a fast failover (if required).
- To guarantee the file system state is preserved, both the Active and Standby NameNodes receive block reports from the DataNodes.

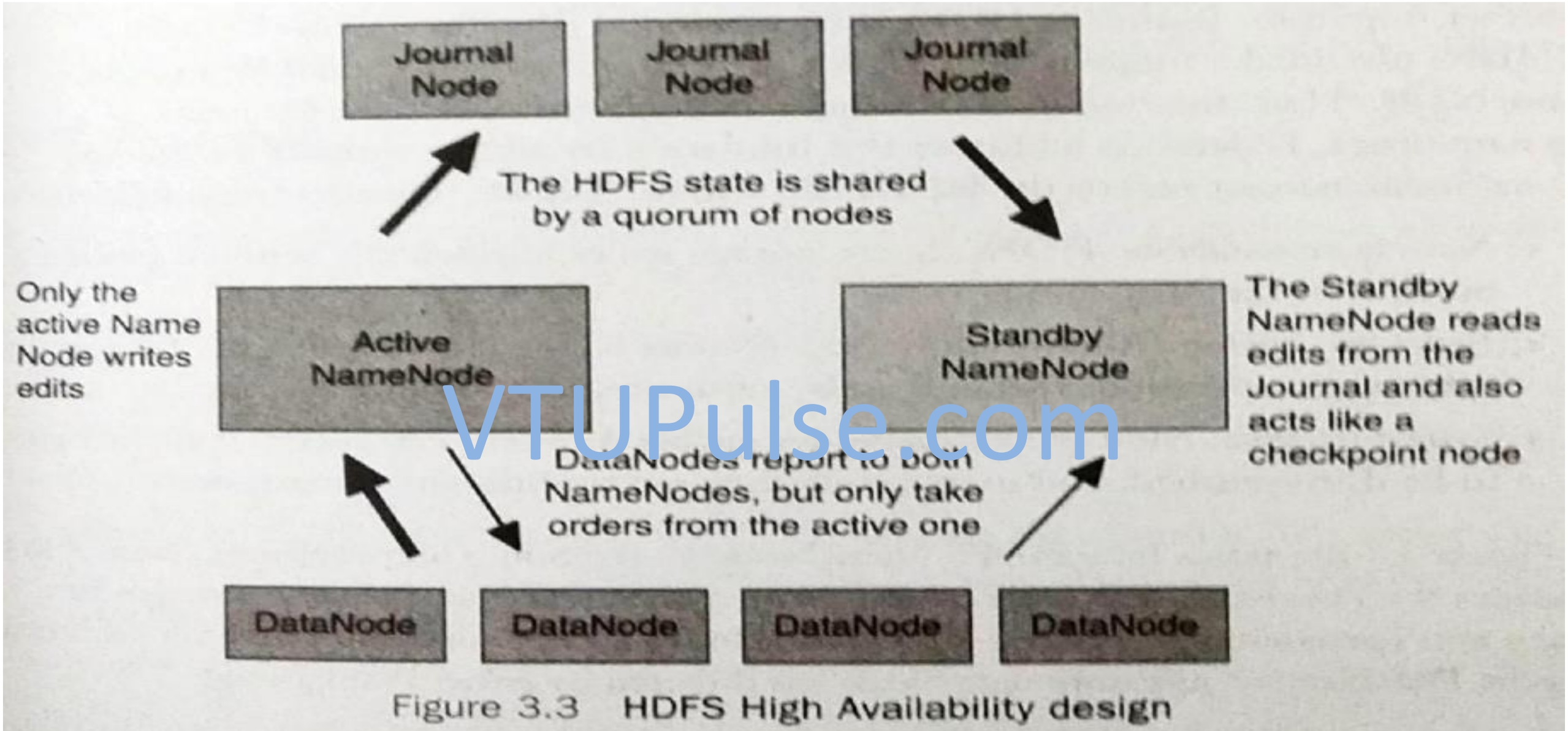


Figure 3.3 HDFS High Availability design

NameNode High Availability

- The Active node also sends all file system edits to a quorum of Journal nodes.
- At least three physically separate Journal Node daemons are required, because edit log modifications must be written to a majority of the Journal Nodes.
- This design will enable the system to tolerate the failure of a single Journal Node machine.
- The Standby node continuously reads the edits from the Journal Nodes to ensure its namespace is synchronized with that of the Active node.
- In the event of an Active NameNode failure, the Standby node reads all remaining edits from the JournalNodes before promoting itself to the Active state.

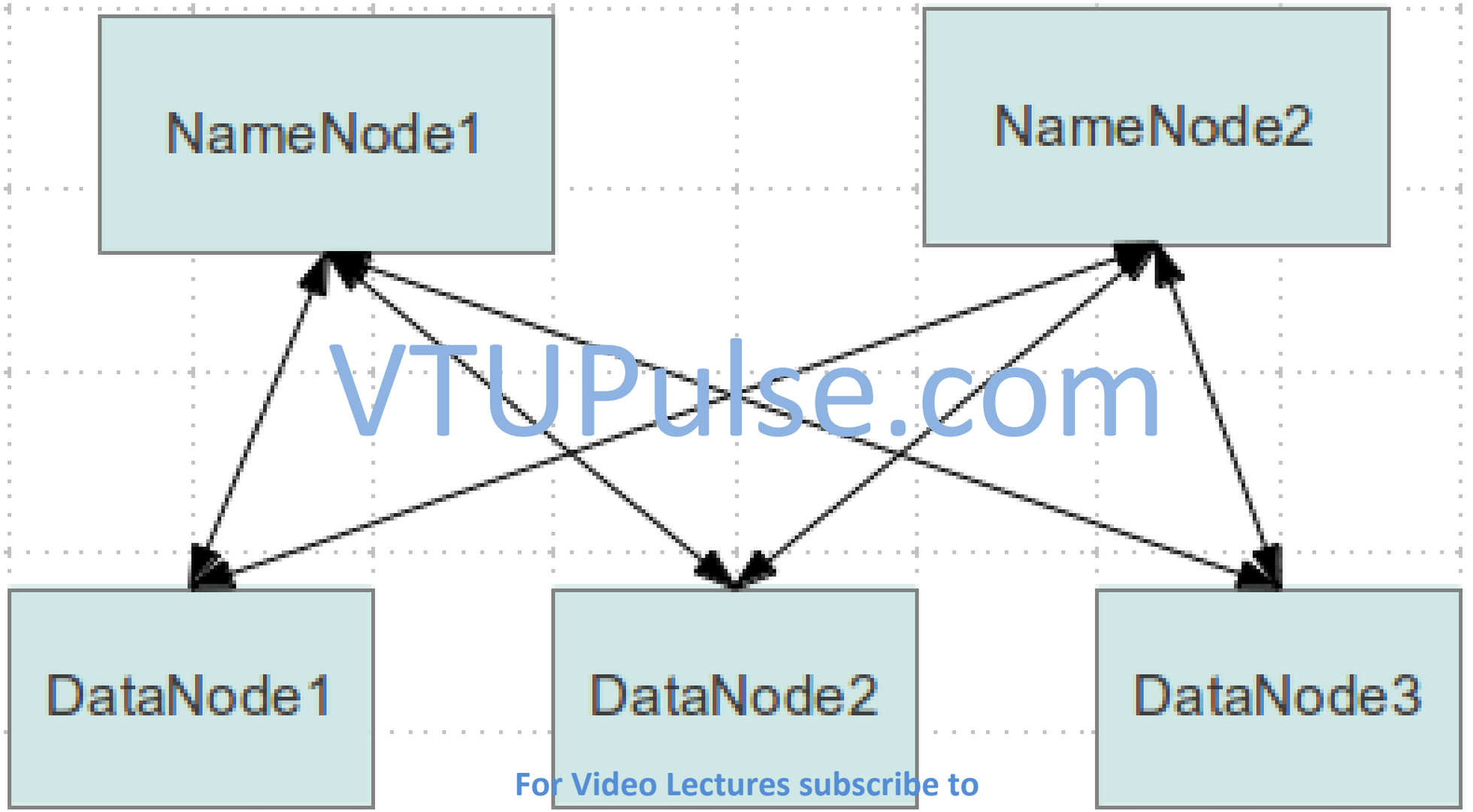
NameNode High Availability

- To prevent confusion between NameNodes, the JournalNodes allow only one NameNode to be a writer at a time. During failover, the NameNode that is chosen to become active takes over the role of writing to the JournalNodes.
- A SecondaryNameNode is not required in the HA configuration because the Standby node also performs the tasks of the Secondary NameNode.
- Apache Zookeeper is used to monitor the NameNode health.
- HDFS failover relies on ZooKeeper for failure detection and for Standby to Active NameNode election

HDFS NameNode Federation

- Older versions of HDFS provided a single namespace for the entire cluster managed by a single NameNode.
- Thus, the resources of a single NameNode determined the size of the namespace.
- Federation addresses this limitation by adding support for multiple NameNodes / namespaces to the HDFS file system.
- The key benefits are as follows:
  - **Namespace scalability.** HDFS cluster storage scales horizontally without placing a burden on the NameNode.
  - **Better performance.** Adding more NameNodes to the cluster scales the file system read/write operations throughput by separating the total namespace.
  - **System isolation.** Multiple NameNodes enable different categories of applications to be distinguished, and users can be isolated to different namespaces.

HDFS NameNode Federation



HDFS Checkpoints and Backups

- The NameNode stores the metadata informaion of the HDFS file system in a file called fsimage.
- File systems modifications are written to an edits log file, and at startup the NameNode merges the edits into a new fsimage.
- The SecondaryNameNode or CheckpointNode periodioally fetches edits from the NameNode, merges them, and returns an updated fsimage to the NameNode.

HDFS Checkpoints and Backups

- An HDFS BackupNode is similar, but also maintains an up-to-date copy of the file system namespace both in memory and on disk.
- Unlike a CheckpointNode, the BackupNode does not need to download the fsimage and edits files from the active NameNode because it already has an up-to-date namespace state in memory.
- A NameNode supports one BackupNode at a time.
- No CheckpointNodes may be registered if a Backup node is in use.

HDFS Snapshots

- HDFS snapshots are similar to backups, but are created by administrators using the **hdfs dfs -snapshot** command.
- HDFS snapshots are read-only point-in-time copies of the file system.
- They offer the following features:
  - Snapshots can be taken of a sub-tree of the file system or the entire file system.
  - Snapshots can be used for data backup, protection against user errors, and disaster recovery.
  - Snapshot creation is instantaneous.
  - Blocks on the DataNodes are not copied, because the snapshot files record the block list and the file size. There is no data copying, although it appears to the user that there are duplicate files.

HDFS User Commands
Hadoop Distributed File System (HDFS)

Mahesh G Huddar

Asst. Professor

Dept. of CSE, HIT, Nidasoshi

HDFS User Commands

General HDFS commands

- hdfs [--config confdir] COMMAND

- hdfs version
  - Hadoop 2.6.0.2.2.4.3-2

- hdfs –dfs
  - List all comands in HDFS

HDFS User Commands

Lists files in HDFS

- hdfs dfs –ls /
  - –Lists files in the root HDFS directory

- hdfs dfs –ls OR
- hdfs dfs –ls /user/hdfs
  - –Lists the files in user home directory

HDFS User Commands

Make Directory in HDFS

- hdfs dfs –mkdir stuff
  - –Create directory

HDFS User Commands

Copy Files to HDFS

- hdfs dfs –put test stuff

  –Copy files to HDFS

- hdfs dfs –ls stuff

  -rw-r--r-- 2 hdfs hdfs 128572020-04-18 12:12 stuff/test

HDFS User Commands

Copy Files from HDFS

- hdfs dfs –get stuff/test test-local
  - –Copy files from HDFS

Copy Files Within HDFS

- hdfs dfs –cp stuff/test test.hdfs
  - –Copy files within HDFS

HDFS User Commands

Delete File

hdfs dfs –rm test.hdfs

–Delete files within HDFS and place into .Trash Folder

hdfs dfs –rm –skipTrash test.hdfs

–Permanent delets the File

HDFS User Commands

Delete Directory

hdfs dfs –rm –r stuff

–Delete files within HDFS and places into .Trash folder

- hdfs dfs –rm –r –skipTrash stuff

–Delete directory in HDFS permanently

HDFS User Commands

Get HDFS Status report

hdfs dfsadmin -report

- Configures Capacity: 1.37TB
- Present Capacity: 1.28TB
- DFS remaining: 1.14TB
- DFS used: 141.87GB
- DFS Used%: 10.83
- Under Replicated Blocks: 54
- Blocks with Crrupt Replicas: 0
- Missing Blocks: 0

Hadoop MapReduce Model

Mahesh G Huddar

Asst. Professor

Dept. of CSE, HIT, Nidasoshi

Hadoop MapReduce Model

- The MapReduce computation model provides a very powerful tool for big data applications.
- Its underlying idea is very simple.
- There are two stages: a **mapping stage** and a **reducing stage**.
- In the mapping stage, a mapping procedure is applied to input data.
  - For instance, assume you need to count how many times each word appears in the novel. One solution is to gather 20 people and give them each a section of the book to search. This step is the map stage.

MapReduce Model

- **Let the content of text file is:**
  - Hello World Hadoop World
  - Hello Hadoop Goodbye Hadoop

  - First Person:
    <Hello, 1>
    <World, 2>
    < Hadoop, 1>
  - Second Person:
    <Hello, 1>
    <Hadoop, 2>
    <Goodbye, 1>

MapReduce Model

- The reduce phase happens when everyone is done counting and reducer sum the total of each word as each one of them tell their counts.

<Hello, 2>

<World, 2>

< Hadoop, 3>

<Goodbye, 1>

MapReduce Parallel Data Flow

1. Input Splits
   - HDFS distributes and replicates data over multiple servers called DatNodes.
   - The default data chunk or block size is 64MB.
   - Thus, a 150MB file would be broken into 3 blocks and written to different machines in the cluster.
   - The data are also replicated on multiple machines (typically three machines).

MapReduce Parallel Data Flow

2. Map Step
   - The mapping process is where the parallel nature of Hadoop comes into play.
   - For large amounts of data, many mappers can be operating at the same time.
   - The user provides the specific mapping process.
   - MapReduce will try to execute the mapper on the machines where the block resides.
   - Because the file is replicated in HDFS, the least busy node with the data will be chosen.
   - If all nodes holding the data are too busy, MapReduce will try to pick a node that is closest to the node that hosts the data block (a characteristic called rack awareness).
   - The last choice is any node in the cluster that has access to HDFS.

MapReduce Parallel Data Flow

3. Combiner step.
   - It is possible to provide an optimization or pre-reduction as part of the map stage where key—value pairs are combined prior to the next stage. The combiner stage is optional.

   - Let the text at mapper is:
     - Hello World Hadoop World
   - Output of Mapper

     <Hello, 1>

     <World, 1>

     < Hadoop, 1>

     <World, 1>

- Output of Combiner

<Hello, 1>

<World, 2>

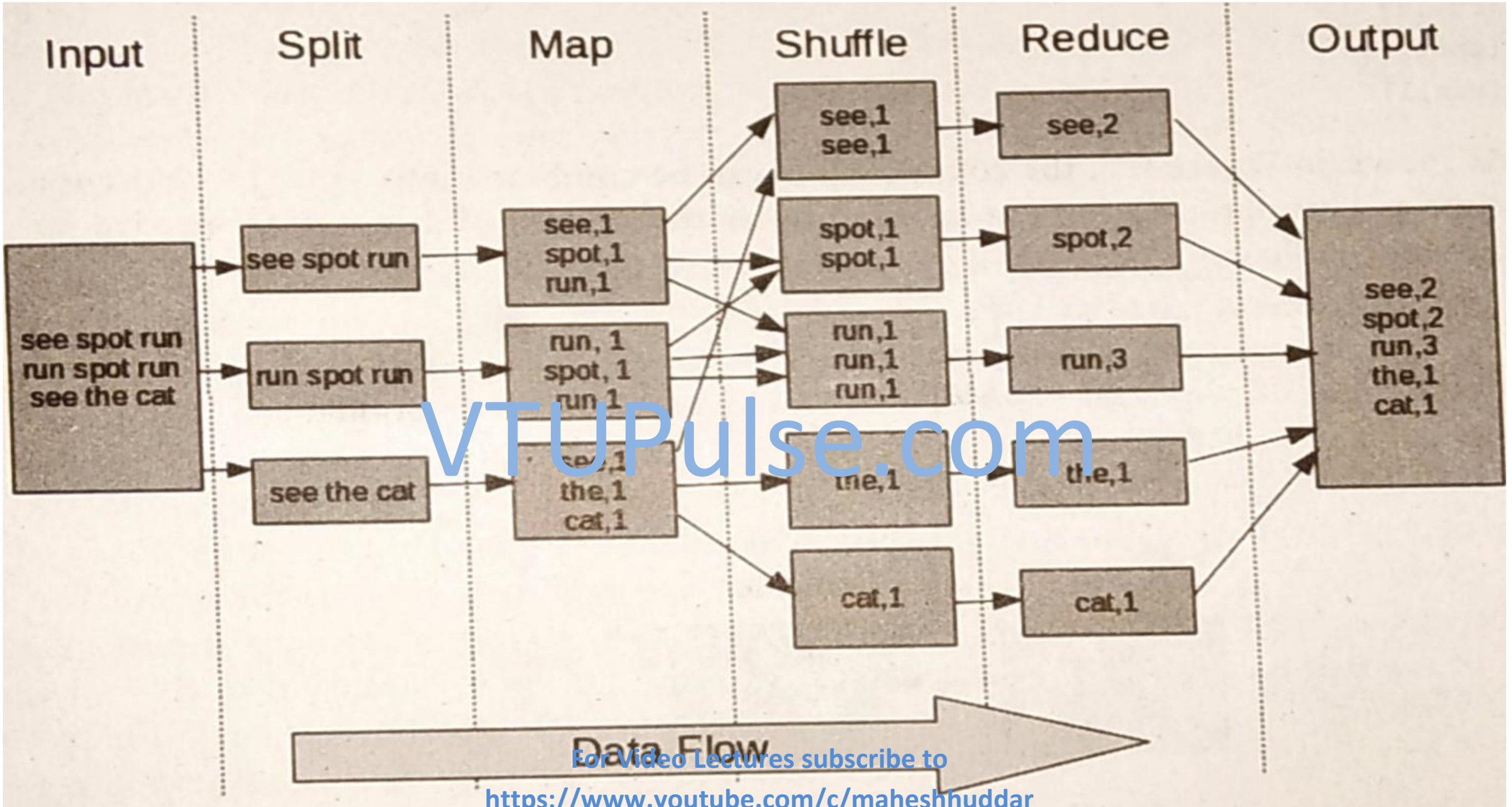
< Hadoop, 1>

MapReduce Parallel Data Flow

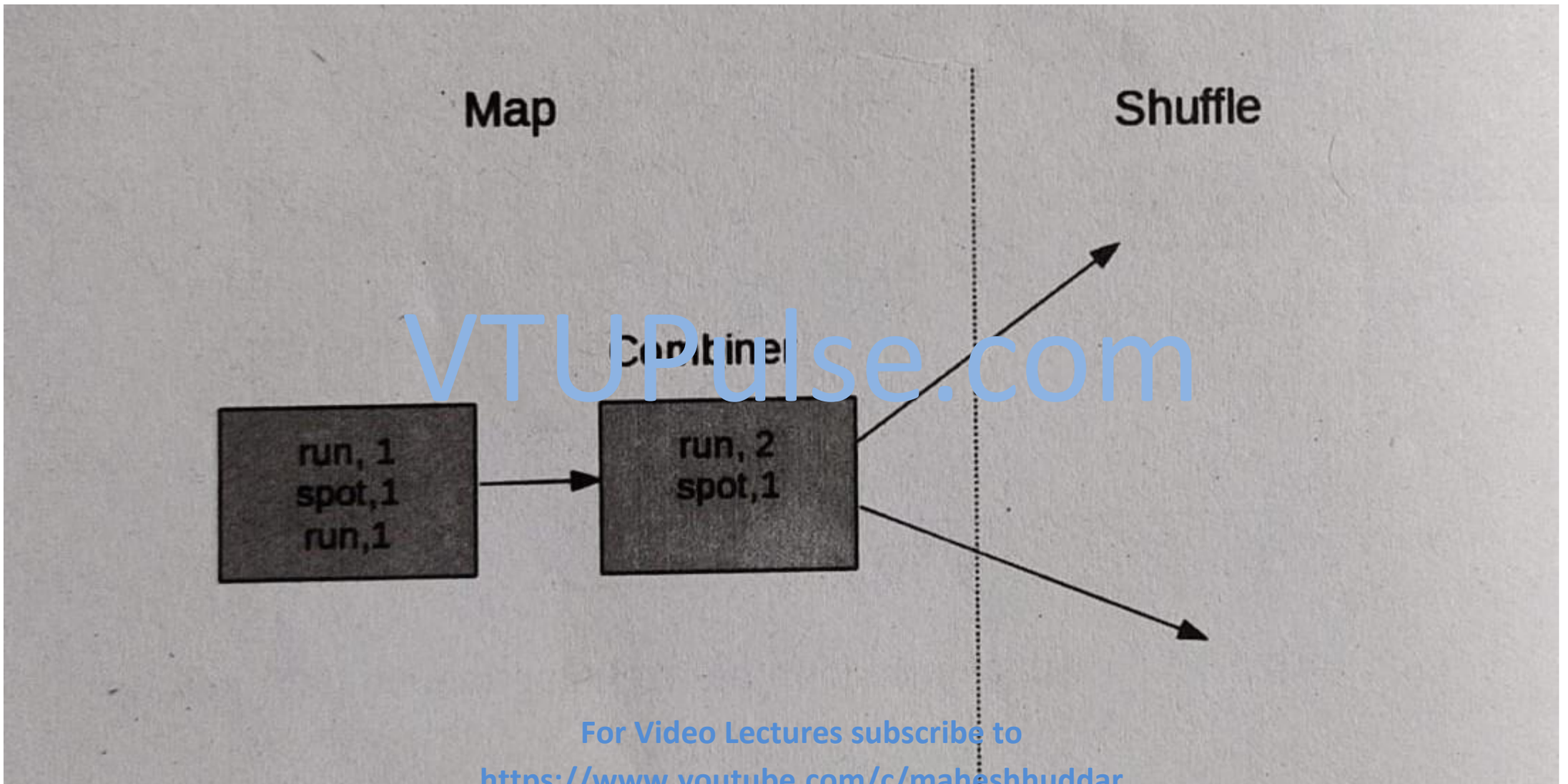
4. Shuffle step.
   - Before the parallel reduction stage can complete, all similar keys must be combined and counted by the same reducer process.
   - Therefore, results of the map stage must be collected by key—value pairs and shuffled to the same reducer process. If only a single reducer process is used, the Shuffle stage is not needed.

MapReduce Parallel Data Flow

5. Reduce Step.
   - The final step is the actual reduction.
   - In this stage, the data reduction is performed as per the programmer's design.
   - The reduce step is also optional
   - The results are written to HDFS. Each reducer will write an output file.
   - For example, a MapReduce job running two reducers will create files called part-0000 and part-0001



MapReduce Parallel Data Flow



Mapper Script (Shell Script)

```
#!/bin/bash
While read line ;
do
  for token in $line;
  do
    if [“$token” = “Ram”];
    then
      echo “Ram, 1”
    elif [“$token” = “Sita”];
    then
      echo “Sita, 1”
    fi
   done
done
```

```
#!/bin/bash
Rcount=0
Scount=0
While read line ;
do
   if [ $line =“Ram, 1”];
     then
          Rcount = Rcount+1
     elif [ $line =“Sita, 1”];
     then
          Scount = Scount+1
     fi
Done
echo “Ram, $Rcount”
echo “Sita, $Scount”
```

To compile and run the program from the command line, perform the following steps:

1. Make a local wordcount_classes directory.

```
$ mkdir wordcount_classes
```

2. Compile the WordCount program using the 'hadoop classpath' command to include all the available Hadoop class paths.

```
$ javac -cp `hadoop classpath' -d wordcount_classes WordCount.java
```

3. The jar file can be created using the following command:

```
$ jar -cvf wordcount.jar -C wordcount_classes/
```


4. To run the example, create an input directory in HDFS and place a text file in the new directory. For this example, we will use the war-and-peace.txt file (available from the book download page; see Appendix A):

```
$ hdfs dfs -mkdir /Demo
$ hdfs dfs -put input. txt /Demo
```

5. Run the WordCount application using the following command:

```
$ hadoop jar wordcount.jar WordCount /Demo/input /output
```

Debugging MapReduce Applications

Hadoop Distributed File System (HDFS)

Mahesh G Huddar

Asst. Professor

Dept. of CSE, HIT, Nidasoshi

Debugging Parallel MapReduce

- The best advice for debugging parallel MapReduce applications is this: **Don't.**
- The key word here is **parallel**.
- Debugging on a distributed system is hard and should be avoided.
- The best approach is to make sure applications run on a simpler system (i.e., the pseudo-distributed single-machine install) with smaller data sets.
- When investigating program behavior at scale, the best approach is to use the application logs to inspect the actual MapReduce progress .
- The time-tested debug print statements are also visible in the logs.

Listing, Killing, and Job Status

- The jobs can be managed using the mapred job command.
- The most import options are -list, -kill, and -status.
  - mapred –list
  - mapred –kill AppID
  - mapred –status AppID
- In addition, the yarn application command can be used to control all applications running on the cluster.

Hadoop Log Management

- The MapReduce logs provide a comprehensive listing of both mappers and reducers.
- The actual log output consists of three files - **stdout**, **stderr**, and **syslog** for the application.
- There are two modes for log storage.
- The first (and best) method is to use **log aggregation**.
- In this mode, logs are aggregated in HDFS and can be displayed in the YARN ResourceManager user interface or examined with the yarn logs command.
- Second, If log aggregation is not enabled, the logs will be placed locally on the cluster nodes on which the mapper or reducer ran. The location of the unaggregated local logs is given by the **yarn.nodemanager.log-dirs** property in the **yarn-site.xml** file

Enabling YARN Log Aggregation

- To manually enable log aggregation, follows these steps:
- As the HDFS superuser administrator (usually user hdfs), create the following directory in HDFS:

```
$ hdfs dfs -mkdir -p /yarn/logs
$ hdfs dfs -chown -R yarn:hadoop /yarn/logs
$ hdfs dfs -chmod -R g+rw /yarn/logs
```

Enabling YARN Log Aggregation

- Add the following properties in the **yarn-site.xml** and restart all YARN services on all nodes.

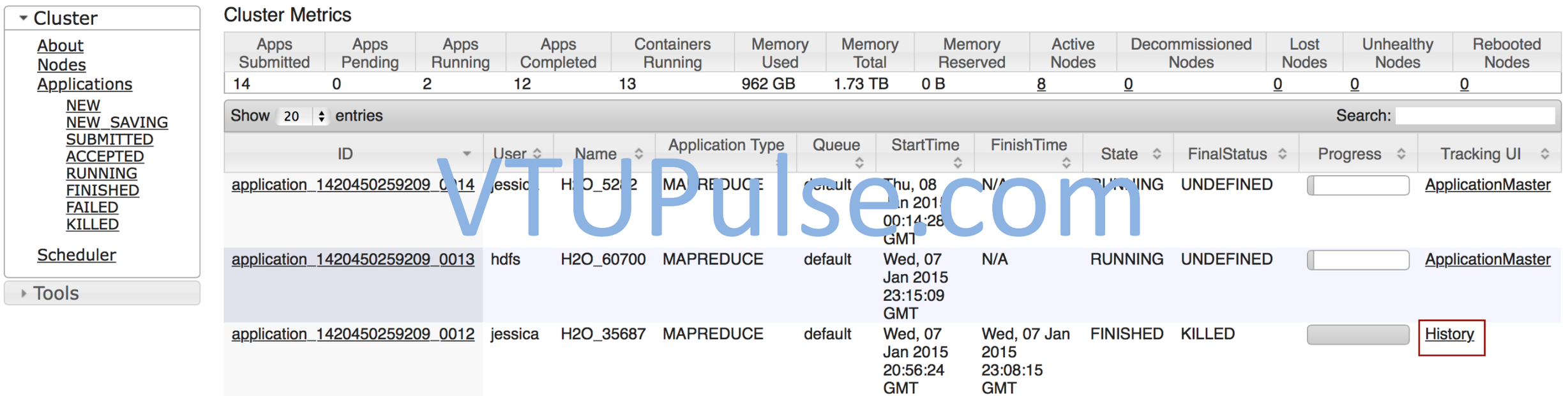
```
<property>
    <name>yarn.nodemanager.remote-app-log-dir</name>
    <value>/yarn/logs</value>
</property>

<property>
    <name>yarn.log-aggregation-enable</name>
    <value>true</value>
</property>
```

Web Interface Log View

- The most convenient way to view logs is to use the YARN ResourceManager web user interface.
- In the figure, the contents of **stdout**, **stderr**, and **syslog** are displayed on a single page.
- If log aggregation is not enabled, a message stating that the logs are not available will be displayed.
- The follwing URL is used to launch the web Interface

http://localhost:8088/

Hadoop Log Management



Hadoop Log Management

- Application
- Job
  - Overview
  - Counters
  - Configuration
  - Map tasks
  - Reduce tasks
- Tools

Job Overview

Job Name:	H2O_35687
User Name:	jessica
Queue:	default
State:	KILLED
Uberized:	false
Submitted:	Wed Jan 07 12:56:24 PST 2015
Started:	Wed Jan 07 12:56:47 PST 2015
Finished:	Wed Jan 07 15:08:15 PST 2015
Elapsed:	2hrs, 11mins, 28sec
Diagnostics:	K[illegible] job j[illegible] ceived from j[illegible]ssica (auth:SIMPLE) at 172.16.2.187 Jo[illegible] rec[illegible]ved k[illegible]ll whi[illegible] in R[illegible]NN[illegible]NG s[illegible]at[illegible]

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Wed Jan 07 12:56:29 PST 2015	mr-0xd1.0xdata.loc:8042	logs

Task Type	Total	Complete
Map	1	0
Reduce	0	0

Attempt Type	Failed	Killed	Successful
Maps	0	1	0
Reduces	0	0	0

Hadoop Log Management

Log Type: stderr
Log Length: 222

```
log4j:WARN No appenders could be found for logger (org.apache.hadoop.ipc.Server).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
```

Log Type: stdout
Log Length: 0

Log Type: syslog
Log Length: 168068
Showing 4096 bytes of 168068 total. Click here for the full log.

```
p, writing event TASK_FAILED
2015-01-07 15:08:15,676 INFO [Thread-324] org.apache.hadoop.mapreduce.jobhistory.JobHistoryEventHandler: In stop, writing event JOB_KILLED
2015-01-07 15:08:15,744 INFO [Thread-324] org.apache.hadoop.mapreduce.jobhistory.JobHistoryEventHandler: Copying hdfs://mr-0xd6.0xdata.loc/user/jessica/.staging/jo
2015-01-07 15:08:15,795 INFO [Thread-324] org.apache.hadoop.mapreduce.jobhistory.JobHistoryEventHandler: Copied to done location: hdfs://mr-0xd6.0xdata.loc/mr-hist
2015-01-07 15:08:15,805 INFO [Thread-324] org.apache.hadoop.mapreduce.jobhistory.JobHistoryEventHandler: Copying hdfs://mr-0xd6.0xdata.loc/user/jessica/.staging/jo
2015-01-07 15:08:15,870 INFO [Thread-324] org.apache.hadoop.mapreduce.jobhistory.JobHistoryEventHandler: Copied to done location: hdfs://mr-0xd6.0xdata.loc/mr-hist
2015-01-07 15:08:15,890 INFO [Thread-324] org.apache.hadoop.mapreduce.jobhistory.JobHistoryEventHandler: Moved tmp to done: hdfs://mr-0xd6.0xdata.loc/mr-history/tm
2015-01-07 15:08:15,895 INFO [Thread-324] org.apache.hadoop.mapreduce.jobhistory.JobHistoryEventHandler: Moved tmp to done: hdfs://mr-0xd6.0xdata.loc/mr-history/tm
2015-01-07 15:08:15,903 INFO [Thread-324] org.apache.hadoop.mapreduce.jobhistory.JobHistoryEventHandler: Moved tmp to done: hdfs://mr-0xd6.0xdata.loc/mr-history/tm
2015-01-07 15:08:15,903 INFO [Thread-324] org.apache.hadoop.mapreduce.jobhistory.JobHistoryEventHandler: Stopped JobHistoryEventHandler. super.stop()
2015-01-07 15:08:15,908 INFO [Thread-324] org.apache.hadoop.mapreduce.v2.app.rm.RMContainerAllocator: Setting job diagnostics to Kill job job_1420450259209_0012 re
Job received Kill while in RUNNING state.

2015-01-07 15:08:15,909 INFO [Thread-324] org.apache.hadoop.mapreduce.v2.app.rm.RMContainerAllocator: History url is http://mr-0xd7.0xdata.loc:19888/jobhistory/job
2015-01-07 15:08:15,926 INFO [Thread-324] org.apache.hadoop.mapreduce.v2.app.rm.RMContainerAllocator: Waiting for application to be successfully unregistered.
2015-01-07 15:08:16,928 INFO [Thread-324] org.apache.hadoop.mapreduce.v2.app.rm.RMContainerAllocator: Final Stats: PendingReds:0 ScheduledMaps:0 ScheduledReds:0 As
2015-01-07 15:08:16,931 INFO [Thread-324] org.apache.hadoop.mapreduce.v2.app.MRAppMaster: Deleting staging directory hdfs://mr-0xd6.0xdata.loc /user/jessica/.stagi
2015-01-07 15:08:16,939 INFO [Thread-324] org.apache.hadoop.ipc.Server: Stopping server on 39529
2015-01-07 15:08:16,941 INFO [IPC Server listener on 39529] org.apache.hadoop.ipc.Server: Stopping IPC Server listener on 39529
2015-01-07 15:08:16,942 INFO [IPC Server Responder] org.apache.hadoop.ipc.Server: Stopping IPC Server Responder
2015-01-07 15:08:16,943 INFO [TaskHeartbeatHandler PingChecker] org.apache.hadoop.mapreduce.v2.app.TaskHeartbeatHandler: TaskHeartbeatHandler thread interrupted
```

Command-Line Log Viewing

- MapReduce logs can also be viewed from the command line.
- The yarn logs command enables the logs to be easily viewed together without having to hunt for individual log files on the cluster nodes.
- As before, log aggregation is required for use.
- The options to yarn logs are as follows:

  $ yarn logs

  Retrieve logs for completed YARN applications .

  usage: **yarn logs -applicationId <application ID> (OPTIONS)**
- general options are:
- -appOwner <Application Owner>
- -containerId <Container ID>
- -nodeAddress <Node Address>

Command-Line Log Viewing

- For example, after running the WordCount example program

$ yarn application -list -appStates FINISHED

- Next, run the following command to produce a dump of all the logs for that application. Note that the output can be long and is best saved to a file.

$ yarn logs -applicationId application_1432667013445_0001 > AppOut

- The actual container can be found using following command:

$ grep –B 1 ===AppOut

Hadoop MapReduce WordCount Program using Streaming Interface in Python

Mahesh G Huddar

Asst. Professor

Dept. of CSE, HIT, Nidasoshi

WordCount Program using Streaming Interface in Python

- Content of Input text file:
- foo foo linux labs foo bar linux

- Output of Mapper:

<foo, 1>

<foo, 1>

<linux, 1>

<labs, 1>

<foo, 1>

<bar, 1>

<linux, 1>

Mapper Sorts the output based on Key Values:

<bar, 1>

<foo, 1>

<foo, 1>

<foo, 1>

<labs, 1>

<linux, 1>

<linux, 1>

Output of Reducer:

<bar, 1>

<foo, 3>

<labs, 1>

<linux, 2>

WordCount Program using Streaming Interface in Python

Mapper Program

```
#!/usr/bin/env python
import sys
for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
        print '%s\t%s' % (word, 1)
```

WordCount Program using Streaming Interface in Python

Reducer Program

```
#!/usr/bin/env python
from operator import itemgetter
import syscurrent_word = None
current_count = 0
word = None
for line in sys.stdin:
        line = line.strip()
        word, count = line.split '\t', 1)
        count = int(coun )
        if current_word == word:
                current_count += count
        else:
                if current_word:
                        print '%s\t%s' % (current_word, current_count)
                current_count = count
                current_word = word
if current_word == word:
        print '%s\t%s' % (current_word, current_count)
```

Steps to execute WordCount Program using Streaming Interface in Python

1. Create a directory and move the file into HDFS

   hdfs dfs -mkdir war-and-peace-input

   hdfs dfs -put war-and-peace.txt war-and-peace-input

2. make sure output directory is removed from any previous runs

   hdfs dfs -rm -r -skipTrash war-and-peace-output

Steps to execute WordCount Program using Streaming Interface in Python

3. run the following (using the run.sh script may be easier)

```
hadoop jar /usr/hdp/current/hadoop-mapreduce-client/hadoop-streaming.jar  \
-file ./mapper.py \
-mapper ./mapper.py \
-file ./reducer.py \
-reducer ./reducer.py \
-input war-and-peace.txt \
-output war-and-peace-output
```

Hadoop MapReduce WordCount Program in Java

Mahesh G Huddar

Asst. Professor

Dept. of CSE, HIT, Nidasoshi

WordCount Program in Java

Content of Input text file:

- Hello World Bye World
- Hello Hadoop Goodbye Hadoop

- Output of Mapper - 1
- <Hello, 1>
- <World, 1>
- <Bye, 1>
- <World, 1>

Output of Mapper - 2

<Hello, 1>

<Hadoop, 1>

<Goodbye, 1>

<Hadoop, 1>

Output of Combiner - 1

<Bye, 1>

<Hello, 1>

<World, 2>

Output of Combiner - 2

<Goodbye, 1>

<Hadoop, 2>

<Hello, 1>

WordCount Program in Java

- Output Shuffle Step:

  <Bye, 1>

  <Goodbye, 1>

  <Hadoop, 2>

  <Hello, 1>

  <Hello, 1>

  <World, 2>

Output of Reducer's

<Bye, 1>

<Goodbye, 1>

<Hadoop, 2>

<Hello, 2>

<World, 2>

WordCount Program in Java

```
import java.io.IOException;import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
```

WordCount Program in Java

```
public class WordCount
{
        public static class TokenizerMapper      extends Mapper<Object, Text, Text, IntWritable>
        {
                private Text word = new Text();
                private final static IntWritable one = new IntWritable(1);
                public void map(Object key, Text value, Context context  ) throws IOException,
InterruptedException
                {
                        StringTokenizer itr = new StringTokenizer(value.toString());
                        while (itr.hasMoreTokens())
                        {
                                word.set(itr.nextToken());
                                context.write(word, one);
                        }
                }
        }
```

WordCount Program in Java

```
public static class IntSumReducer    extends Reducer<Text,IntWritable,Text,IntWritable>
{
        private IntWritable result = new IntWritable();
        public void reduce(Text key, Iterable<IntWritable> values,  Context context )
throws IOException, InterruptedException
        {
                int sum = 0;
                for (IntWritable val : values)
                {
                        sum += val.get();
                 }
                result.set(sum);
                context.write(key, result);
        }
}
```

WordCount Program in Java

```
public static void main(String[] args) throws Exception
{
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}
```

Steps to execute Basic Hadoop Java word count example Hadoop V2

1. Make Locak WordCount_classes directory

   mkdir wordcount_classes

2. Compile the WordCount.java program, use the `hadoop classpath`

   javac -cp `hadoop classpath` -d wordcount_classes WordCount.java

3. Create a java archive for distribution

   jar -cvf wordcount.jar -C wordcount_classes/ .

Steps to execute Basic Hadoop Java word count example Hadoop V2

4. Create a directory and move the file into HDFS

```
hdfs dfs -mkdir war-and-peace-input
hdfs dfs -put war-and-peace.txt war-and-peace-input
```


5. run work count, but first

```
hadoop jar wordcount.jar  WordCount war-and-peace-output
```

6. check for output form Hadoop job

```
hdfs dfs -ls war-and-peace-output
```

Steps to execute Basic Hadoop Java word count example Hadoop V2

7. move it back to working directory (example of "hadoop dfs -get")

hdfs dfs -get war-and-peace-output/part-00000

hdfs dfs -get war-and-peace-output/part-00001

Note:

- If you run program again it wont work because /war-and-peace-output exists.
- Hadoop will not overwrite files!

Hadoop MapReduce WordCount Program in C++ using Pipes interface

Mahesh G Huddar

Asst. Professor

Dept. of CSE, HIT, Nidasoshi

WordCount Program in C++ using Pipes Interface

Content of Input text file:

- foo foo linux labs foo bar linux

- Output of Mapper:

<foo, 1>

<foo, 1>

<linux, 1>

<labs, 1>

<foo, 1>

<bar, 1>

<linux, 1>

Output of Reducer:

<bar, 1>

<foo, 3>

<labs, 1>

<linux, 2>

WordCount Program in C++ using Pipes interface

```
#include <string>
#include  "stdint.h"  // <--- to prevent uint64_t errors!
#include "Pipes.hh“
#include "StringUtils.hh“

using namespace std;
```

WordCount Program in C++ using Pipes interface

```
class WordCountMapper : public HadoopPipes::Mapper
{
        public:    WordCountMapper( HadoopPipes::TaskContext& context ) {  }
        void map( HadoopPipes::MapContext& context )
        {
                string line = context.getInputValue();
                vector< string > words =  HadoopUtils::splitString( line, " " );
                for ( unsigned int i=0; i < words.size(); i++ )
                {
                        context.emit( words[i], HadoopUtils::toString( 1 ) );
                }
        }
};
```

WordCount Program in C++ using Pipes interface

```
class WordCountReducer : public HadoopPipes::Reducer
{
        public:  WordCountReducer(HadoopPipes::TaskContext& context) {  }
        void reduce( HadoopPipes::ReduceContext& context )
        {
                int count = 0;
                while ( context.nextValue() )
                {
                        count += HadoopUtils::toInt( context.getInputValue() );
                }
        context.emit(context.getInputKey(), HadoopUtils::toString( count ));
        }
};
```

WordCount Program in C++ using Pipes interface

```
vint main(int argc, char *argv[])
{
        return HadoopPipes::runTask(HadoopPipes::TemplateFactory<
                                    WordCountMapper,
                                    WordCountReducer >()
                                    );
}
```

Steps to execute Basic Hadoop C++ word count example (using Pipes interface) Hadoop V2

1. Compile C++ program

   g++ wordcount.cpp -o wordcount -L$HADOOP_HOME//lib/native/ -I$HADOOP_HOME/../usr/include -lhadooppipes -lhadooputils -lpthread –lcrypto

2. Create a directory and move the file into HDFS

   hdfs dfs -mkdir war-and-peace-input

   hdfs dfs -put war-and-peace.txt war-and-peace-input

3. put executable into HDFS so tasktrackers can find the program

   hdfs dfs -put wordcount bin

   hdfs dfs -rm -r -skipTrash war-and-peace-output

Steps to execute Basic Hadoop C++ word count example (using Pipes interface) Hadoop V2

4. Run program

```
mapred pipes \
-D hadoop.pipes.java.recordreader=true  \
-D hadoop.pipes.java.recordwriter=true \
-input war-and-peace.txt   \
-output war-and-peace-output  \
-program bin/wordcount
```