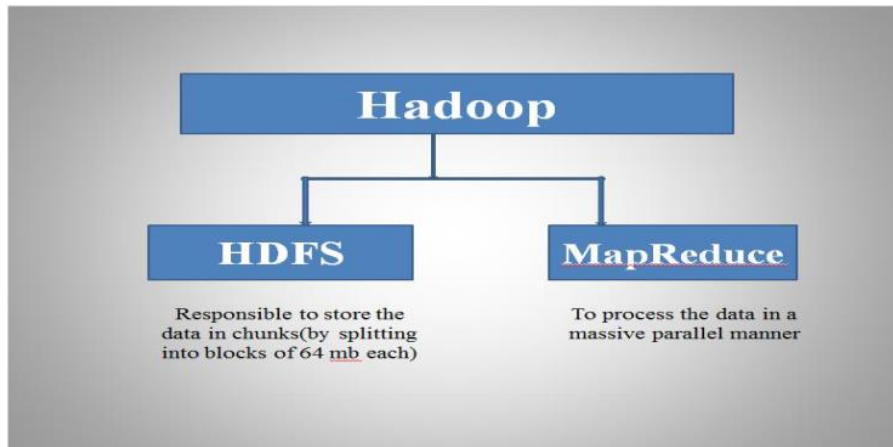


Hadoop Concepts

Topics to be covered

- Hadoop Architecture
- Playing with HDFS
- Working of MapReduce



Hadoop Architecture

Hadoop – An Introduction

- Hadoop is a **Distributed Data Storage & Distributed Data Processing Framework**
 - **Distributed Data Storage** is achieved by **HDFS**
 - **Distributed Data Processing** is achieved by **MAPREDUCE**
- A set of machines running HDFS and MAPREDUCE is known as a Hadoop Cluster
- An Individual machine in the cluster is known as a Node
- A cluster can have as few as one node or as many as thousands
 - **More Nodes = Better Performance**
- Hadoop can process any structured ,unstructured or semi structured data

The Five Daemons of Hadoop

- **NameNode**
 - Holds the metadata for HDFS
- **Secondary NameNode**
 - Performs housekeeping functions for the NameNode
 - Is not a backup or hot standby for the NameNode !
- **DataNode**
 - Stores actual HDFS data blocks
- **JobTracker**
 - Manages MapReduce jobs, distributes individual tasks to the machines on the cluster
- **TaskTracker**
 - Instantiates and monitors individual Map and Reduce tasks

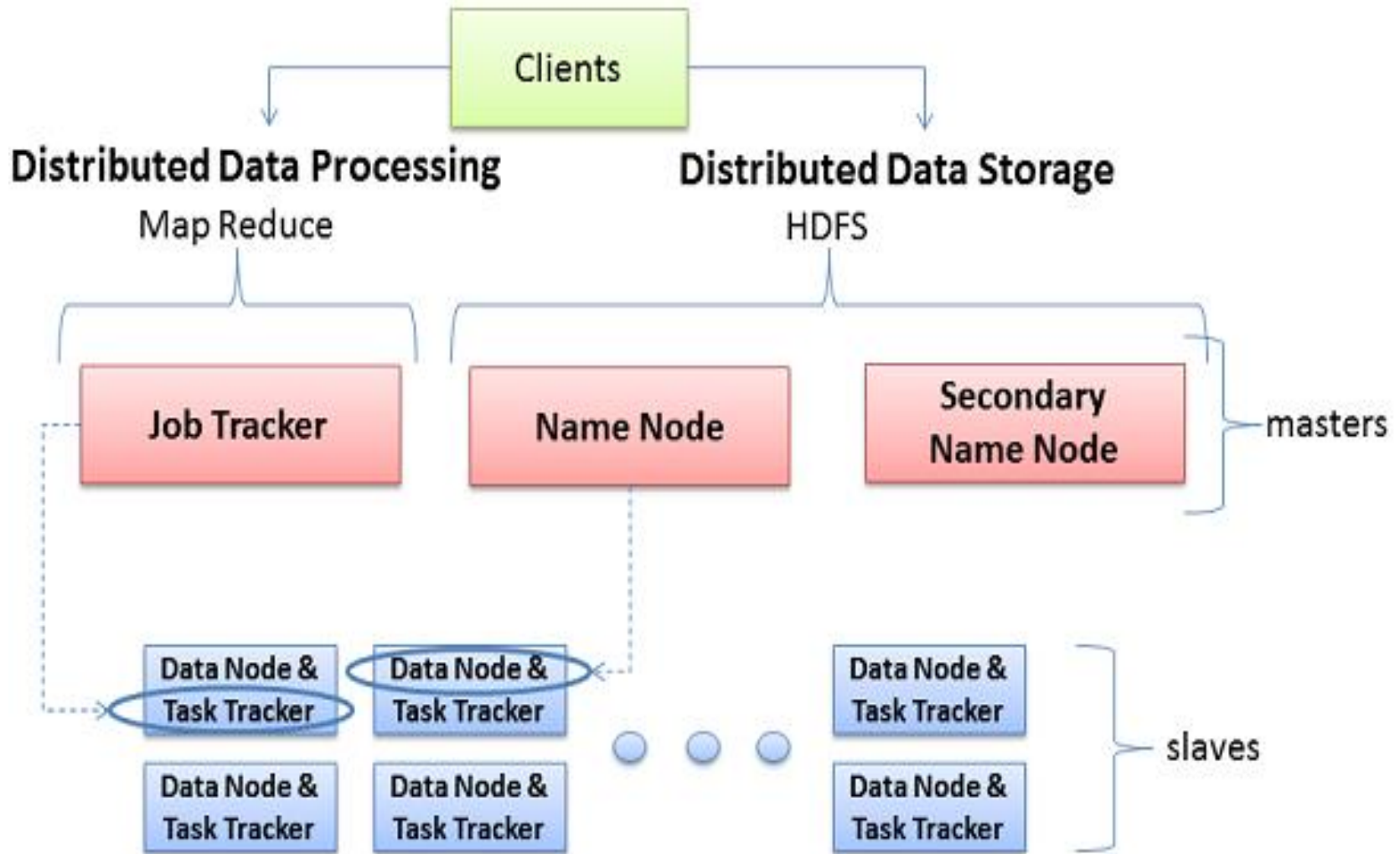
The Five Daemons of Hadoop (Cont'd)

- Each daemon runs in its own Java Virtual Machine (JVM)

We can consider nodes to be in two different categories:

- Master Nodes
 - Run the NameNode, Secondary NameNode, JobTracker daemons
- Slave Nodes
 - Run the DataNode and TaskTracker daemons
 - A slave node will run both of these daemons

Hadoop and its Components



Namenode

- The NameNode daemon is a master daemon that must be running at all times in any Hadoop Cluster
- **NameNode** holds all the **metadata** (file information, data information, which file is in which node, nodes which are down, etc.) and coordinates access to data
- Hardware of **Namenode** has to be of reliable quality hardware as compared to hardware of other nodes in the cluster
- The NameNode is a **Single Point of failure** for the Hadoop Cluster
- The NameNode holds all of its **metadata in RAM** for fast access
 - It keeps a record of changes on disk for crash recovery

File storage in Hadoop

NameNode

Foo.txt: blk_001, blk_002, blk_003
Bar.txt: blk_004, blk_005

DataNodes

blk_001

blk_005

blk_002

blk_003

blk_005

blk_003

blk_004

blk_001

blk_002

blk_005

blk_001

blk_003

blk_004

blk_002

blk_004

Secondary Namenode

- **Secondary NameNode** Daemon is usually run on a different machine than the primary NameNode since its memory requirements are of the same order as the primary NameNode
- There are 2 files called **edits** and **fsimage** which helps in the name node recovery
- The **fsimage** file contains a snapshot of the HDFS metadata. Whenever there is a change in HDFS, it will be appended to the **edits** file
- To prevent the **edits** file growing infinitely, the secondary Namenode periodically merges (every one hour) these two files and then the Namenode starts writing changes to a new edits file
- The Secondary Namenode merges the changes from the **edits** file with the snapshot from the **fsimage** file and creates an updated **fsimage** file which can be used for recovery of Namenode

JobTracker

- The **JobTracker** is a master daemon service for submitting and tracking MapReduce jobs in Hadoop
- There is only one Job Tracker process run on any Hadoop Cluster
- In a typical production cluster its run on a separate machine. Each slave node is configured with job tracker node location
- The JobTracker is the single point of failure for the Hadoop MapReduce service. If it goes down, all running jobs are halted
- The JobTracker coordinates all the jobs running on any Hadoop Cluster, by scheduling tasks to run on tasktrackers
- Tasktrackers run tasks and send progress reports to the jobtracker, which keeps a record of the overall progress of the job
- If a tasks fails, the jobtracker can reschedule it on a different tasktracker

DataNode

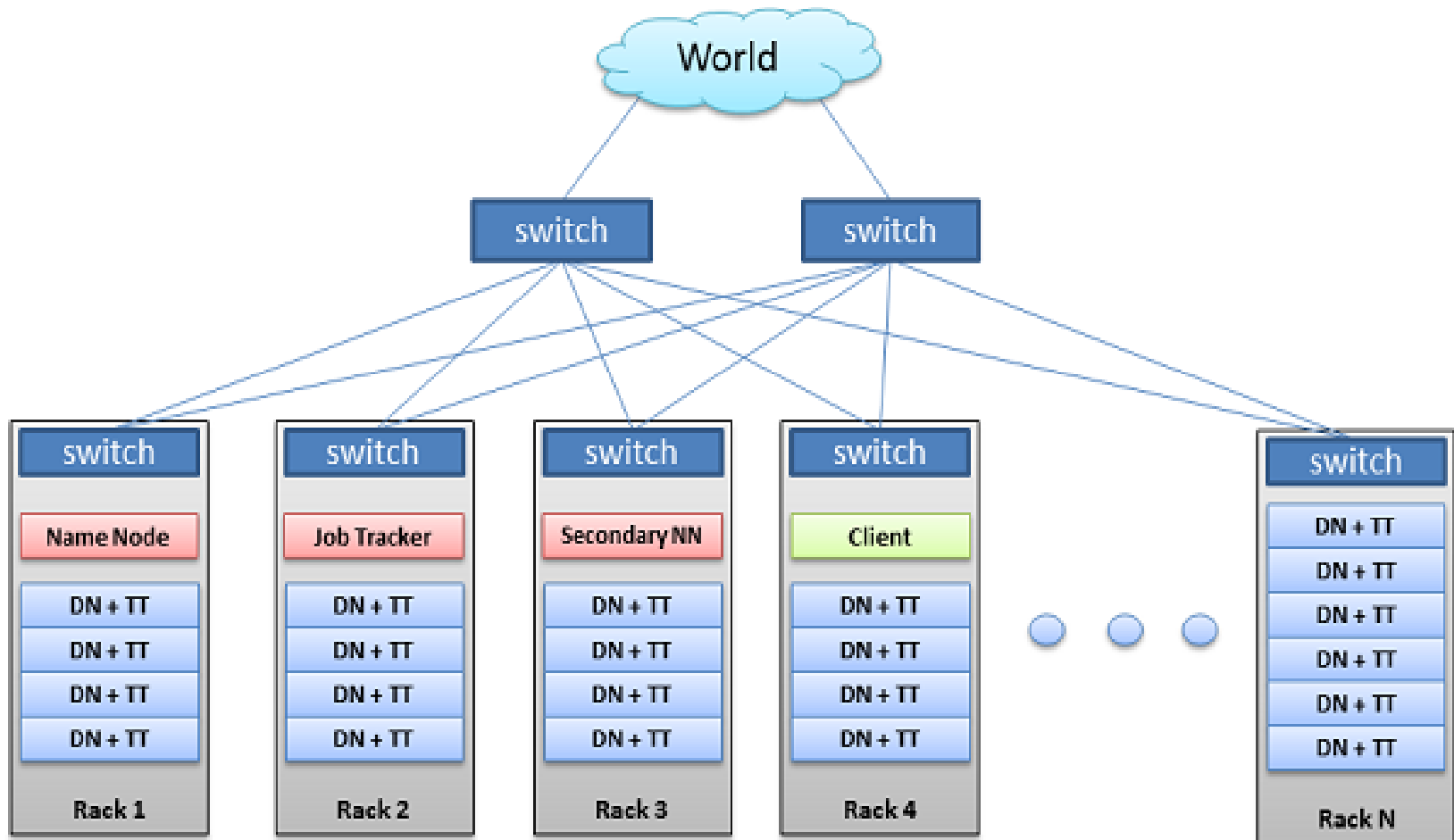
- **DataNode** daemon runs along with tasktracker daemon which runs in every slave node is used to store the actual HDFS data blocks
- DataNode ensures that the data is properly read/written from/to HDFS
- During write, each data node takes responsibility of replication of blocks
- Every Data node in the cluster sends a **heart beat** to the NameNode every 3 seconds which tells the NameNode that the particular data node is alive
- Every 10th heart beat is a **block report** which ensures that the Name Node has the most recent **metadata** and the data blocks are replicated to at least three nodes in the cluster (as per replication property)
- If a data node goes down, the **NameNode consults the metadata**, finds the affected data blocks and ensures that the data gets replicated in accordance with the Rack Awareness principle

TaskTracker

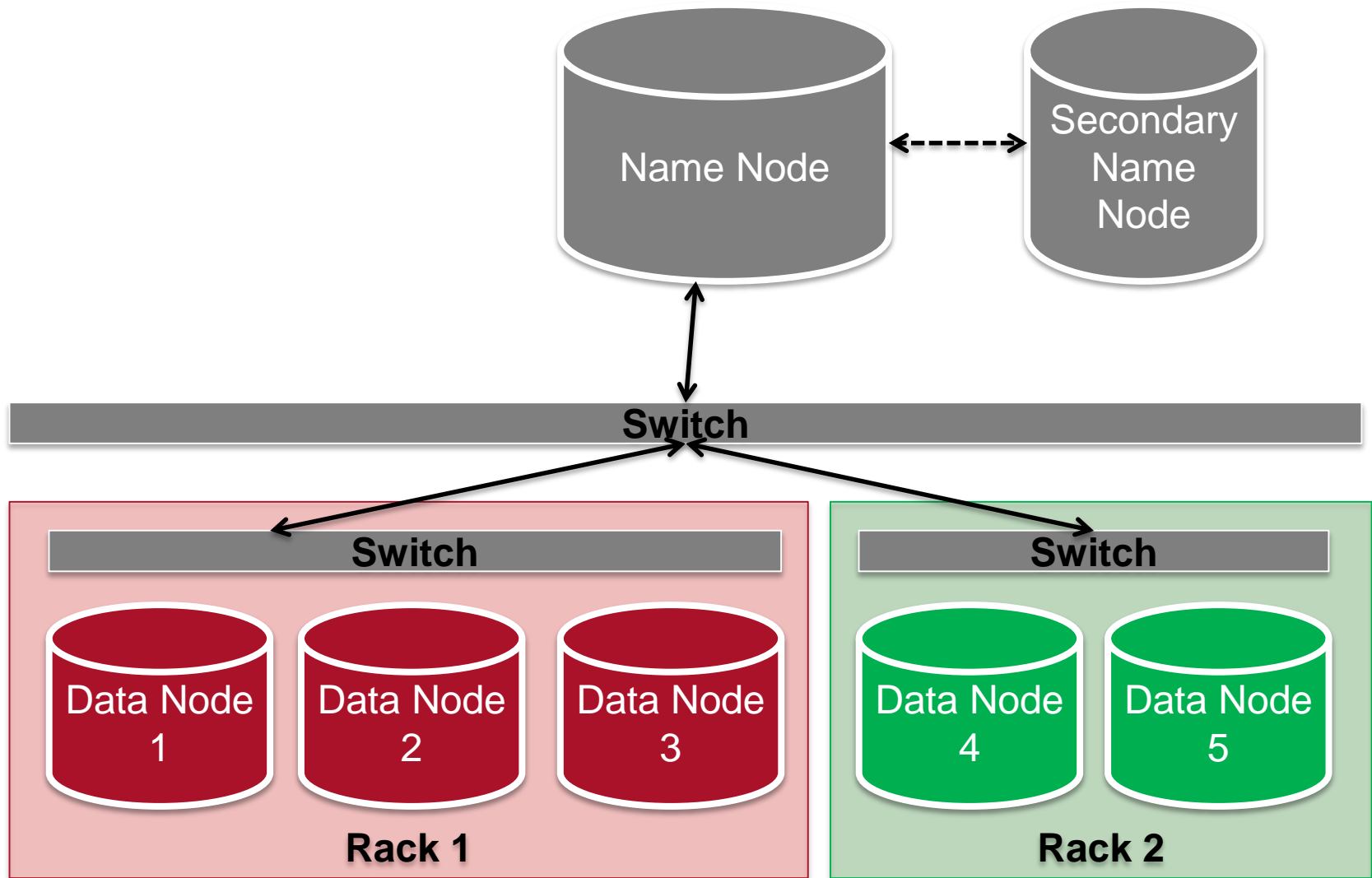
- A **TaskTracker** is also a slave node daemon in the cluster that accepts tasks (Map, Reduce and Shuffle operations) from a JobTracker
- Every TaskTracker is configured with a set of slots, these indicate the number of tasks that it can accept
- The TaskTracker starts a separate JVM processes to process each task(called as Task Instance) to ensure that process failure does not take down the task tracker
- The TaskTracker monitors these task instances, capturing the output and exit codes, whether the Task instances finish, successfully or not, the task tracker notifies the JobTracker
- The TaskTrackers also send out heartbeat messages to the JobTracker, usually every few minutes, to reassure the JobTracker that it is still alive which also informs the JobTracker of the number of available slots, so the JobTracker can stay up-to-date with where in the cluster work can be delegated

A Hadoop Cluster

Hadoop Cluster



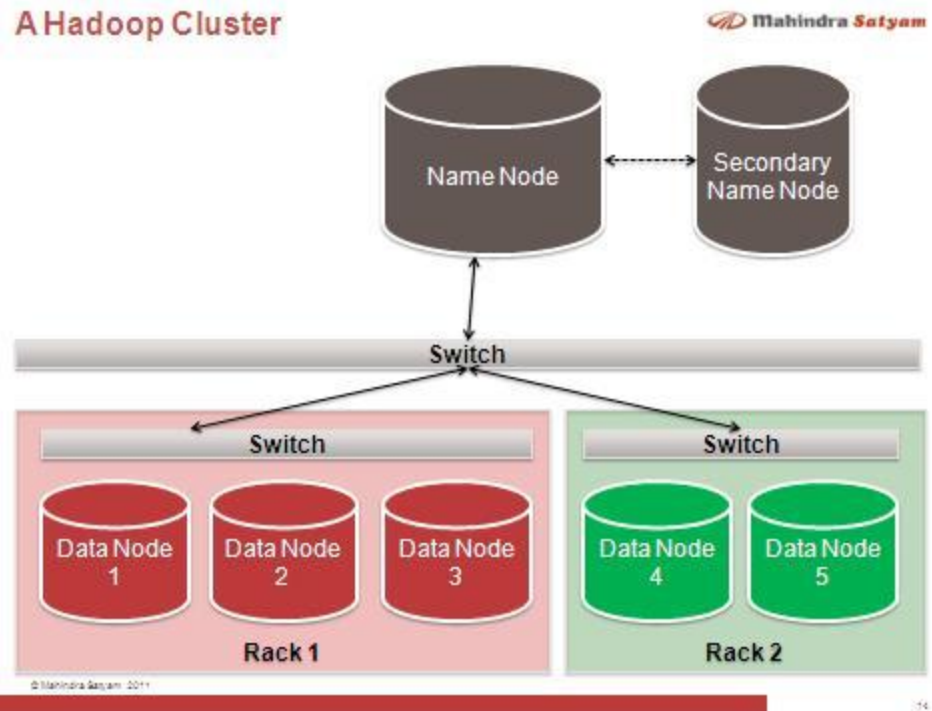
Another Representation of a Hadoop Cluster



Data Loading & Processing in Hadoop

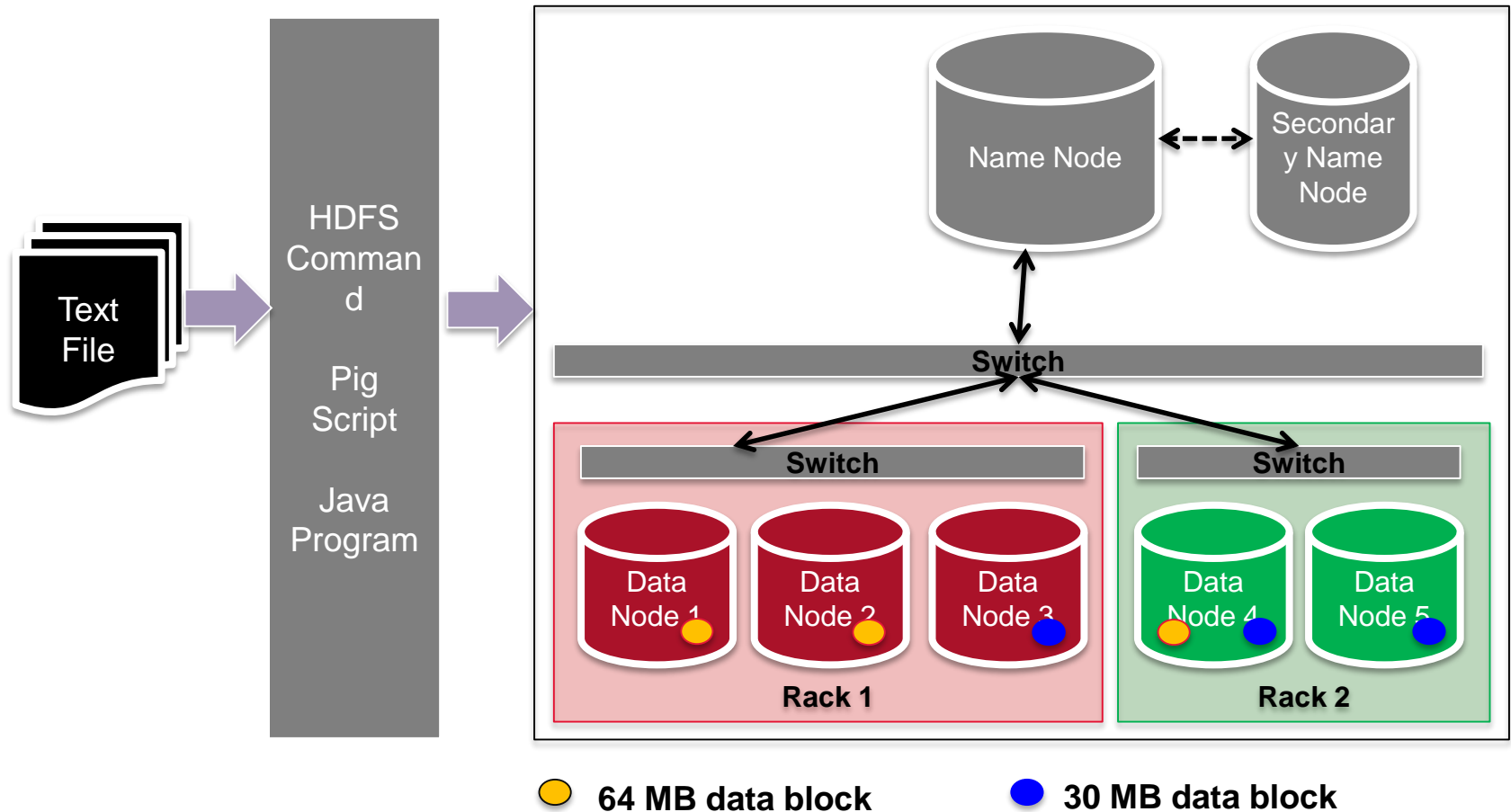
- Assume, we have a file which has a list of geometrical objects (point, line, etc.) in .txt format and want to store it in Hadoop (HDFS). The size of the file is 94 MB. Our Hadoop Cluster has 5 data nodes and 1 name node.
- Once the file is stored in HDFS, we need to know the total number of points for each shape .

A Hadoop Cluster






Storing data in Hadoop



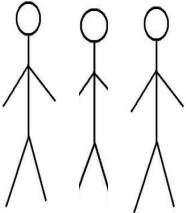
A single file, broken into blocks of data is stored (multiple times) across different nodes in the Hadoop cluster.



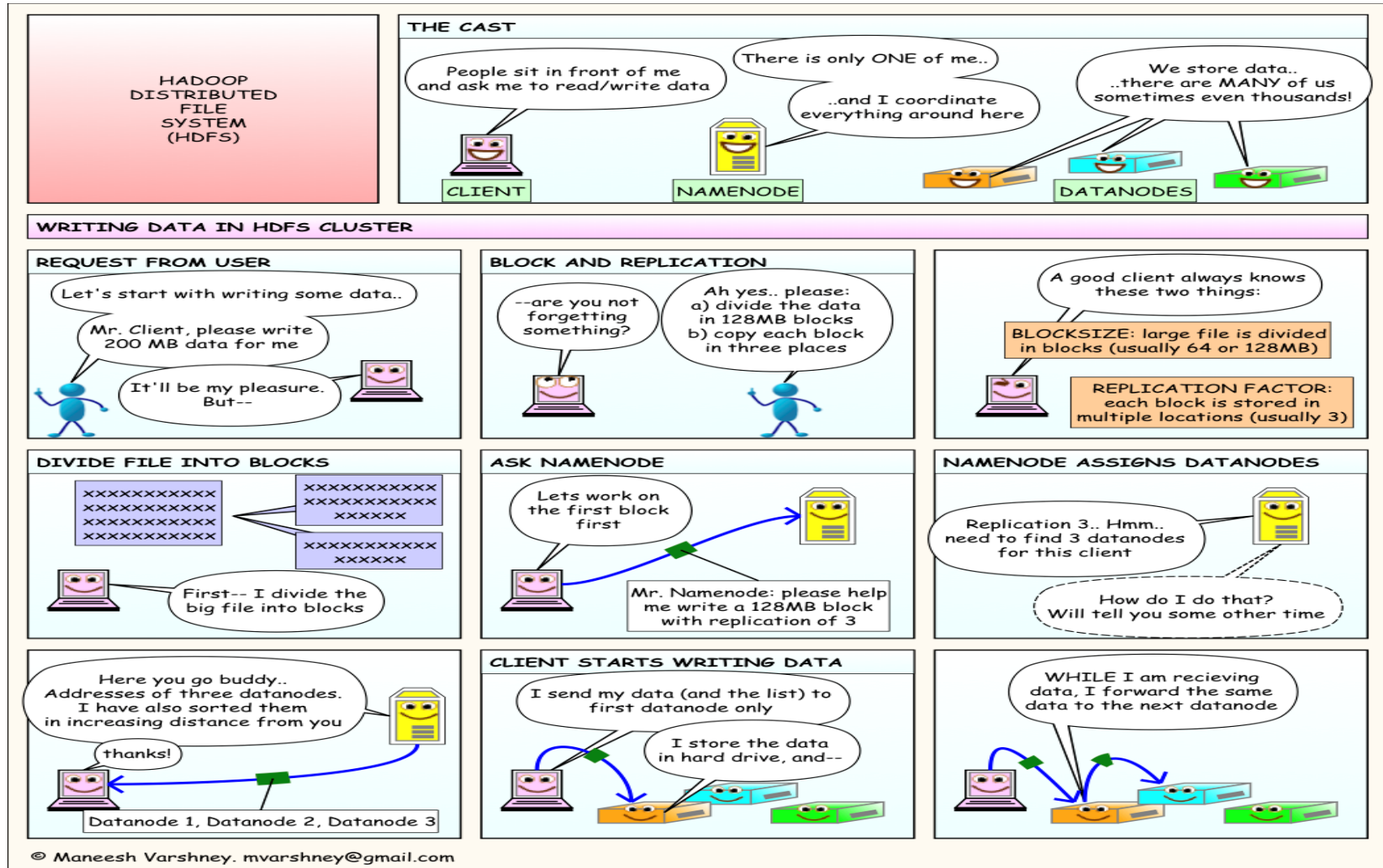
Storage Steps in Hadoop

| Actor | Action | Reaction |
|------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|  | HDFS command to copy a file | Temp file creation A Temp file starts getting created on the client machine itself. When the temp file accumulates data > 1 HDFS block, the client contacts the NameNode. |
|  | Checks with NameNode I want to store a block of data, Tell me where can I store it? | NameNode Confirms You can store it on DataNodes 1, 2 & 4. I have selected these nodes based on Rack Awareness & Replication Property. |
|  | Checks with DataNode1 Hi DataNode1, please confirm if you are available & ready to receive the data. Also check with DataNode2 & DataNode4. | DataNode1 Confirms I & DataNode2 & DataNode4 are available and ready to receive the data. |

Storage Steps in Hadoop

| Actor | Action | Reaction |
|------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------|
|  | Begins Data Transfer Ok, I am sending the data in 4KB packets. Store it and send a copy to DataNode2. Tell DataNode2 to store it and send a copy to DataNode4. Confirm to me and NameNode once the data is copied. | DataNode1, DataNode2 & DataNode4 receive the data The 3 selected nodes receive the data from the client in packets of 4KB. |
|  | Confirm Data Receipt to Client I & DataNode2 and DataNode4 have received the data. | None |
|  | Confirm to NameNode We have received the data. | None |

Writing to HDFS



Hadoop Distributed File System - HDFS

Overview

- The File System (FS) shell includes various shell-like commands that directly interact with the Hadoop Distributed File System (HDFS) as well as other file systems that Hadoop supports, such as Local FS, HFTP FS, S3 FS, and others. The FS shell is invoked by:
 - `bin/hadoop fs <args>`
- All FS shell commands take path URIs as arguments.
- The URI format is `scheme://authority/path`.
 - `hdfs://` For HDFS the scheme is `hdfs`. This is default- Can be configured.
 - `file://` for local file system.
- Most of the commands in FS shell behave like corresponding Unix commands.
- If HDFS is being used, `hdfs dfs` is a synonym.

Hadoop Distributed Filesystem (HDFS)

- HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware
- HDFS is written in java which is based on Google's GFS
- HDFS sits on top of a native file systems (like ext3, ext4 or xfs)
- HDFS provides redundant storage for massive amounts of data
 - Using readily available, industry standard computers
- HDFS performs best with a 'modest' number of large files
 - **Millions**, rather than billions, of files
 - Each file typically **100MB** or more
- Files in HDFS are read-many, write once with no random writes possible

HDFS (Cont'd)

- A file is split into blocks and distributed across multiple nodes in the cluster, each block is typically 64MB or 128MB in size
- Although files are split into 64MB or 128MB blocks, if a file is smaller than this the full 64MB/128MB will not be used
- Each block is replicated multiple times across Nodes called **DataNodes**
 - Default is to replicate each block three times
 - Replicas are stored on different nodes
 - This ensures both reliability and availability
- Access to HDFS from the command line is achieved with the **hadoop fs** command
- Applications can also read and write HDFS files directly via the Java API

HDFS (Cont'd)

- A node called NameNode in the cluster keeps track of which blocks make up a file, and where those blocks are located i.e. metadata
- Without the metadata on the NameNode, there is no way to access the files stored in HDFS
- If a Datanode goes down, the NameNode consults the metadata, finds the affected data blocks and ensures that the data gets replicated in accordance with the Rack Awareness principle

HDFS Commands

```
[root@sandbox ~]# hdfs dfs -ls /user
Found 12 items
drwxr-xr-x   - admin      hdfs          0 2016-01-23 04:55 /user/admin
drwxrwx---   - ambari-qa hdfs          0 2015-10-27 14:39 /user/ambari-qa
drwxr-xr-x   - guest     guest         0 2015-10-27 14:55 /user/guest
drwxr-xr-x   - hcat      hdfs          0 2015-10-27 14:43 /user/hcat
drwx-----   - hdfs     hdfs          0 2015-10-27 15:18 /user/hdfs
drwx-----   - hive     hdfs          0 2016-01-23 04:11 /user/hive
drwxrwxrwx   - hue       hdfs          0 2015-10-27 14:55 /user/hue
drwxrwxr-x   - oozie     hdfs          0 2015-10-27 14:44 /user/oozie
drwxr-xr-x   - solr      hdfs          0 2015-10-27 14:49 /user/solr
drwxrwxr-x   - spark     hdfs          0 2015-10-27 14:40 /user/spark
drwxr-xr-x   - unit      hdfs          0 2015-10-27 14:45 /user/unit
drwxr-xr-x   - zeppelin  zeppelin      0 2015-10-27 15:16 /user/zeppelin
[root@sandbox ~]# _
```

ls

- Usage: `hadoop fs -ls [-d] [-h] [-R] [-t] [-S] [-r] [-u] <args>`
- Options:
 - `-d`: Directories are listed as plain files.
 - `-h`: Format file sizes in a human-readable fashion (eg 64.0m instead of 67108864).
 - `-R`: Recursively list subdirectories encountered.
 - `-t`: Sort output by modification time (most recent first).
 - `-S`: Sort output by file size.
 - `-r`: Reverse the sort order.
 - `-u`: Use access time rather than modification time for display and sorting.
- Example:

\$ `hadoop fs -ls /user/hadoop/file1`

- Exit Code:
 - Returns 0 on success and -1 on error.

lsr

- Usage: `hadoop fs -lsr <args>`
- Recursive version of `ls`.
- **Note:** This command is deprecated. Instead use `hadoop fs -ls -R`

cat

- Usage: `hadoop fs -cat URI [URI ...]`

- Copies source paths to stdout.

- Example:

```
$ hadoop fs -cat hdfs://nn1.example.com/file1 hdfs://nn2.example.com/file2
```

```
$ hadoop fs -cat file:///file3 /user/hadoop/file4
```

- Exit Code:
- Returns 0 on success and -1 on error.

copyFromLocal

- Usage: `hadoop fs -copyFromLocal <localsrc> URI`
- Similar to `put` command, except that the source is restricted to a local file reference.
- Options:
- The `-f` option will overwrite the destination if it already exists.

copyToLocal

- Usage: `hadoop fs -copyToLocal [-ignorecrc] [-crc] URI <localdst>`
- Similar to `get` command, except that the destination is restricted to a local file reference.

cp

- Usage: `hadoop fs -cp [-f] [-p | -p[topax]] URI [URI ...] <dest>`
- Copy files from source to destination. Allows multiple sources as well in which case the destination must be a directory.
- 'raw.*' namespace extended attributes are preserved if (1) the source and destination filesystems support them (HDFS only), and (2) all source and destination pathnames are in the `/.reserved/raw` hierarchy. Determination of whether raw.* namespace xattrs are preserved is independent of the `-p` (preserve) flag.
- Options:
 - The `-f` option will overwrite the destination if it already exists.
 - The `-p` option will preserve file attributes [topx] (timestamps, ownership, permission, ACL, XAttr). If `-p` is specified with no *arg*, then preserves timestamps, ownership, permission. If `-pa` is specified, then preserves permission also because ACL is a super-set of permission. Determination of whether raw namespace extended attributes are preserved is independent of the `-p` flag.

cp

- Example:

\$ `hadoop fs -cp /user/hadoop/file1 /user/hadoop/file2`

\$ `hadoop fs -cp /user/hadoop/file1 /user/hadoop/file2 /user/hadoop/dir`

- Exit Code:

- Returns 0 on success and -1 on error.

moveFromLocal

- Usage: `hadoop fs -moveFromLocal <localsrc> <dst>`
- Similar to `put` command, except that the source `localsrc` is deleted after it's copied.

moveToLocal

- Usage: `hadoop fs -moveToLocal [-crc] <src> <dst>`
- Displays a “Not implemented yet” message.

mv

- Usage: `hadoop fs -mv URI [URI ...] <dest>`
- Moves files from source to destination. This command allows multiple sources as well in which case the destination needs to be a directory. Moving files across file systems is not permitted.
- Example:

\$ `hadoop fs -mv /user/hadoop/file1 /user/hadoop/file2`
\$ `hadoop fs -mv hdfs://nn.example.com/file1 hdfs://nn.example.com/file2`
`hdfs://nn.example.com/file3 hdfs://nn.example.com/dir1`
- Exit Code:
 - Returns 0 on success and -1 on error.

rm

- Usage: `hadoop fs -rm [-f] [-r |-R] [-skipTrash] URI [URI ...]`
- Delete files specified as args.
- Options:
 - The `-f` option will not display a diagnostic message or modify the exit status to reflect an error if the file does not exist.
 - The `-R` option deletes the directory and any content under it recursively.
 - The `-r` option is equivalent to `-R`.
 - The `-skipTrash` option will bypass trash, if enabled, and delete the specified file(s) immediately. This can be useful when it is necessary to delete files from an over-quota directory.
- Example:

\$ `hadoop fs -rm hdfs://nn.example.com/file /user/hadoop/emptydir`
- Exit Code:
 - Returns 0 on success and -1 on error.

count

- Usage: `hadoop fs -count [-q] [-h] [-v] <paths>`
 - Count the number of directories, files and bytes under the paths that match the specified file pattern. The output columns with `-count` are: `DIR_COUNT`, `FILE_COUNT`, `CONTENT_SIZE`, `PATHNAME`
 - The output columns with `-count -q` are: `QUOTA`, `REMAINING_QUOTA`, `SPACE_QUOTA`, `REMAINING_SPACE_QUOTA`, `DIR_COUNT`, `FILE_COUNT`, `CONTENT_SIZE`, `PATHNAME`
 - The `-h` option shows sizes in human readable format.
 - The `-v` option displays a header line.

- Example:

```
$ hadoop fs -count hdfs://nn1.example.com/file1 hdfs://nn2.example.com/file2
$ hadoop fs -count -q hdfs://nn1.example.com/file1
$ hadoop fs -count -q -h hdfs://nn1.example.com/file1
$ hdfs dfs -count -q -h -v hdfs://nn1.example.com/file1
```

- Exit Code:
 - Returns 0 on success and -1 on error

get

- Usage: `hadoop fs -get [-ignorecrc] [-crc] <src> <localdst>`
- Copy files to the local file system. Files that fail the CRC check may be copied with the `-ignorecrc` option. Files and CRCs may be copied using the `-crc` option.
- Example:

\$ `hadoop fs -get /user/hadoop/file localfile`
\$ `hadoop fs -get hdfs://nn.example.com/user/hadoop/file localfile`
- Exit Code:
 - Returns 0 on success and -1 on error.

put

- Usage: `hadoop fs -put <localsrc> ... <dst>`
- Copy single src, or multiple srcs from local file system to the destination file system. Also reads input from stdin and writes to destination file system.

\$ `hadoop fs -put localfile /user/hadoop/hadoopfile`

\$ `hadoop fs -put localfile1 localfile2 /user/hadoop/hadoopdir`

\$ `hadoop fs -put localfile hdfs://nn.example.com/hadoop/hadoopfile`

\$ `hadoop fs -put - hdfs://nn.example.com/hadoop/hadoopfile` Reads the input from stdin.

- Exit Code:
 - Returns 0 on success and -1 on error.

mkdir

- Usage: `hadoop fs -mkdir [-p] <paths>`
- Takes path uri's as argument and creates directories.
- Options:
 - The `-p` option behavior is much like Unix `mkdir -p`, creating parent directories along the path.
- Example:

\$ `hadoop fs -mkdir /user/hadoop/dir1 /user/hadoop/dir2`
\$ `hadoop fs -mkdir hdfs://nn1.example.com/user/hadoop/dir`
`hdfs://nn2.example.com/user/hadoop/dir`
- Exit Code:
 - Returns 0 on success and -1 on error.

rmdir

- Usage: `hadoop fs -rmdir [--ignore-fail-on-non-empty] URI [URI ...]`
 - Delete a directory.
 - Options:
 - `--ignore-fail-on-non-empty`: When using wildcards, do not fail if a directory still contains files.
 - Example:
- \$ `hadoop fs -rmdir /user/hadoop/emptydir`

rmr

- Usage: `hadoop fs -rmr [-skipTrash] URI [URI ...]`
- Recursive version of delete.
- **Note:** This command is deprecated. Instead use `hadoop fs -rm -r`

setrep

- Usage: `hadoop fs -setrep [-R] [-w] <numReplicas> <path>`
- Changes the replication factor of a file. If *path* is a directory then the command recursively changes the replication factor of all files under the directory tree rooted at *path*.
- Options:
 - The `-w` flag requests that the command wait for the replication to complete. This can potentially take a very long time.
 - The `-R` flag is accepted for backwards compatibility. It has no effect.
- Example:

\$ `hadoop fs -setrep -w 3 /user/hadoop/dir1`
- Exit Code:
 - Returns 0 on success and -1 on error.

stat

- Usage: `hadoop fs -stat [format] <path> ...`
 - Print statistics about the file/directory at <path> in the specified format. Format accepts filesize in blocks (%b), type (%F), group name of owner (%g), name (%n), block size (%o), replication (%r), user name of owner(%u), and modification date (%y, %Y). %y shows UTC date as “yyyy-MM-dd HH:mm:ss” and %Y shows milliseconds since January 1, 1970 UTC. If the format is not specified, %y is used by default.
 - Example:
- ```
$ hadoop fs -stat "%F %u:%g %b %y %n" /file
```
- Exit Code: Returns 0 on success and -1 on error.

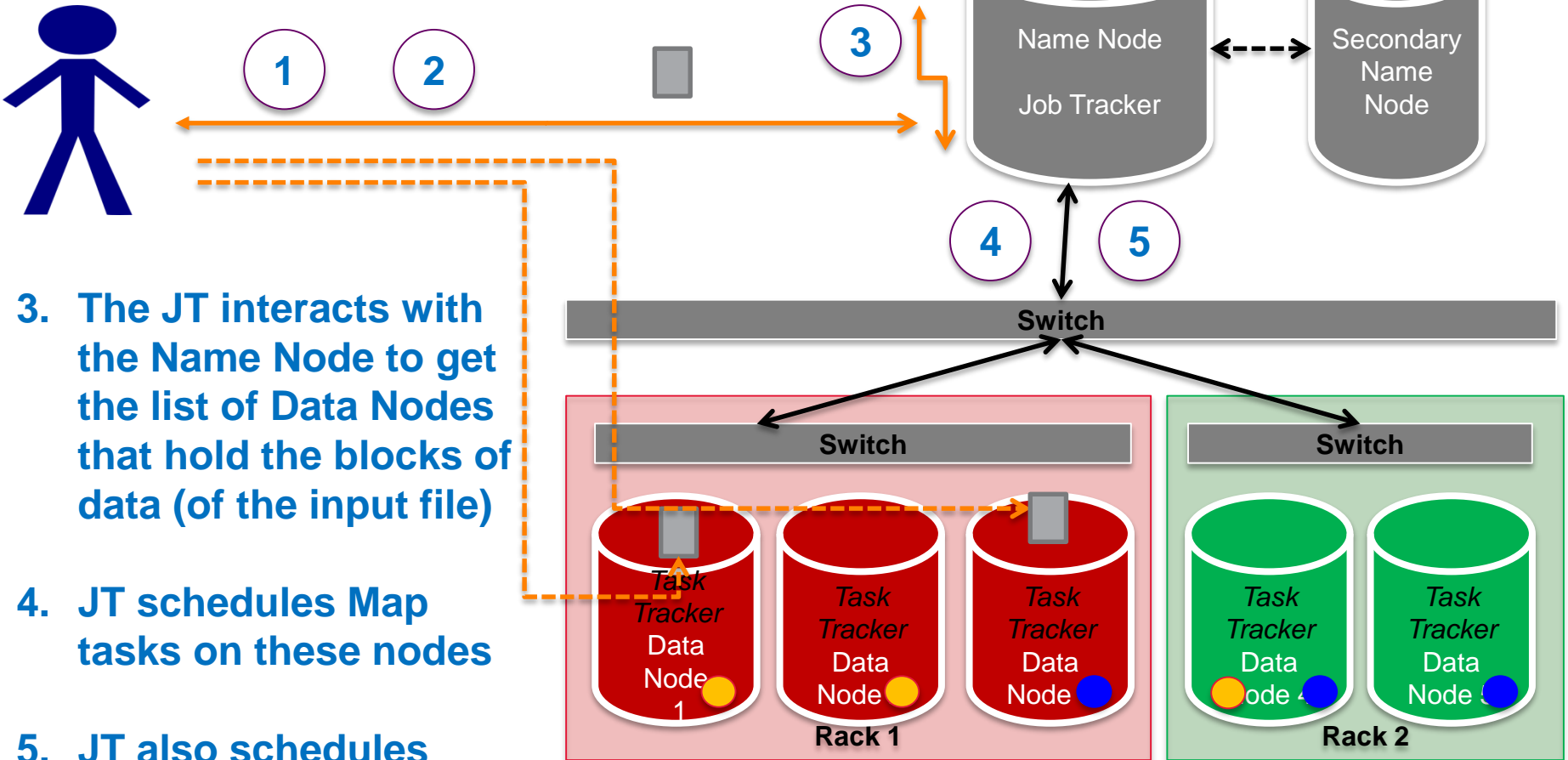
# MapReduce

# MapReduce

- MapReduce is the programming model for data processing on Hadoop
- Hadoop can run MapReduce programs written in various languages such as Java, Ruby, Python and C++ but usually Java is preferred
- Map Reduce works by breaking the processing into 3 phases (sequentially)
  - Map Phase
  - Shuffle and Sort
  - Reducer Phase
- Each Phase has key–value pairs as input and output
- A MapReduce job execution is nothing but complete execution of all Map tasks and reduce tasks
- Each Map task operates on a discrete portion of the overall dataset ,typically one HDFS block of data
- In the reduce phase, the MapReduce system distributes the intermediate data to nodes which performs the Reduce phase

# Data Processing in Hadoop

1. Client submits the program to Job Tracker
2. The Job Tracker provides a Job ID to Client



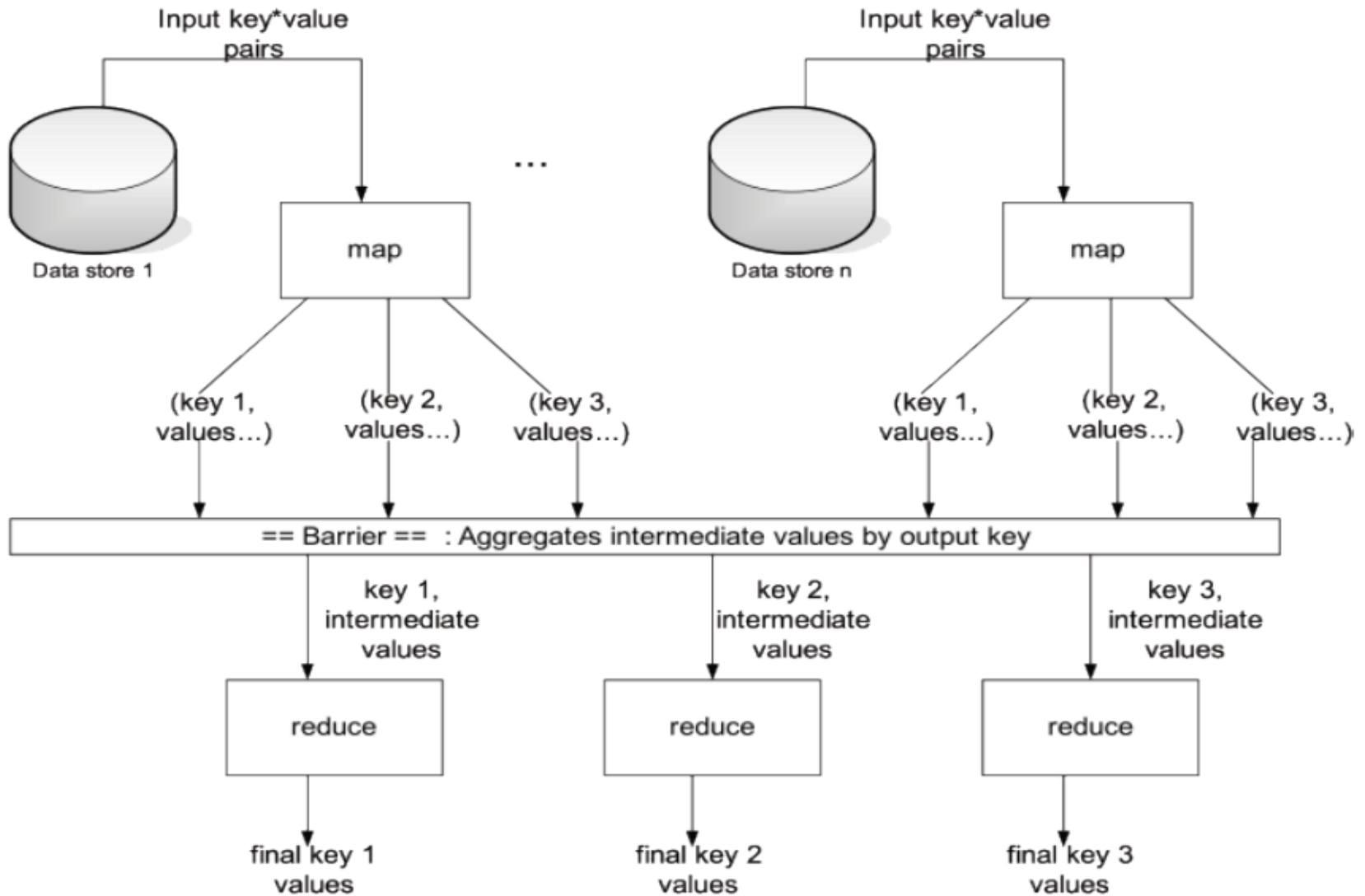
3. The JT interacts with the Name Node to get the list of Data Nodes that hold the blocks of data (of the input file)
4. JT schedules Map tasks on these nodes
5. JT also schedules Reducer tasks on Nodes in the Cluster

● 64 MB data block

● 30 MB data block



# MapReduce Flow



# MapReduce Flow

- Users writes a Map function
  - Mappers each work on each input split
  - The map function process a key/value pair
  - Generates zero or more sets of intermediate key/value pairs
  - The intermediate key/value pairs passes through Shuffle and sort phase
- Shuffle and sort transfers map results to the Reducers
  - Ensures all values with the same key go to the same Reducer
- Users also writes a Reduce function
  - Merges all intermediate values associated with the same intermediate key
  - Reducers consolidate the results, produce the final output
  - It is also possible to write a Map/only job (no Reducers)

# Data Processing in Hadoop – Map Phase

Input data slice on  
respective Node

In Memory with  
Key, Values

Partitioned, Combined &  
spilled on Disk

Point  
Line  
Triangle  
Quadrilateral  
Line  
Triangle  
Point  
Line  
Quadrilateral  
Pentagon  
Pentagon

M  
A  
P

Point,1  
Line,2  
Triangle,3  
Quadrilateral,4  
Line,2  
Triangle,3  
Point,1  
Line,2  
Quadrilateral,4  
Pentagon,5  
Pentagon,5

Partition  
  
Sort  
  
Combiner

S  
P  
I  
L  
L

Line,6  
Pentagon,10  
-----  
Point,2  
Quadrilateral,8  
-----  
Triangle,6

Triangle  
Point  
Pentagon  
Quadrilateral  
Point  
Line  
Triangle  
Pentagon

M  
A  
P

Triangle,3  
Point,1  
Pentagon,5  
Quadrilateral,4  
Point,1  
Line,2  
Triangle,3  
Pentagon,5

Partition  
  
Sort  
  
Combiner

S  
P  
I  
L  
L

Line,2  
Pentagon,10  
-----  
Point,2  
Quadrilateral,4  
-----  
Triangle,6

# Data Processing in Hadoop – Reduce Phase

Copied as  
separate files with  
KeyValues

Merged prior to  
Reducing (with  
KeyValues)

Line,6  
Pentagon,10

Line,2  
Pentagon,10

Point,2  
Quadrilateral,8

Point,2  
Quadrilateral,4

Triangle,6

Triangle,6

Line,6  
Line,2  
Pentagon,10  
Pentagon,10

Point,2  
Point,2  
Quadrilateral,8  
Quadrilateral,4

Triangle,6  
Triangle,6

Reducer 1

Reducer 2

Reducer 3

O/P in HDFS

Line,8  
Pentagon,20  
Point,4  
Quadrilateral,12  
Triangle,12

# Mapper

- Hadoop attempts to ensure that Mappers run on nodes which hold their portion of the data locally, to avoid network traffic
  - Multiple Mappers run in parallel, each processing a portion of the input data
- The Mapper reads data in the form of key/value pairs
- Mapper's output is zero or more key/value pairs
- If the Mapper writes anything out, the output must be in the form of key/value pairs (intermediate output) which is written to the local filesystem

# Example Mapper

Turn input into upper case (pseudocode)

```
let map(k, v) =
emit(k.toUpper(), v.toUpper())
```

```
('foo', 'bar') -> ('FOO', 'BAR')
('foo', 'other') -> ('FOO', 'OTHER')
('baz', 'more data') -> ('BAZ', 'MORE DATA')
```

# Example Mapper: Explode Mapper

Output each input mapper separately (pseudocode)

```
let map(k, v) =
 foreach char c in v:
 emit (k, c)
```

```
('foo', 'bar') -> ('foo', 'b'), ('foo', 'a'),
 ('foo', 'r')
('baz', 'other') -> ('baz', 'o'), ('baz', 't'),
 ('baz', 'h'), ('baz', 'e'),
 ('baz', 'r')
```

# Example Mapper: Changing Keyspaces

The key output by the mapper does not need to be identical to the input key

```
let map(k, v) =
emit(v.length(), v)
```

Output the word length as the key

```
('foo', 'bar') -> (3, 'bar')
('baz', 'other') -> (5, 'other')
('foo', 'abracadabra') -> (11, 'abracadabra')
```



# Partitioner

- The Partitioner divides up the keyspace
  - Controls which Reducer, each intermediate key and its associated values goes to
- Often, the default behavior is fine
  - Default is the HashPartitioner

```
public class HashPartitioner<K, V> extends Partitioner<K, V>
{
 public int getPartition(K key, V value, int numReduceTasks) {
 return (key.hashCode() & Integer.MAX_VALUE) / numReduceTasks;
 }
}
```

- Custom Partitioners can be also defined to avoid potential performance issues i.e for load balancing
  - To avoid one Reducer having to deal with many very large lists of values

# Combiner

- Combiners decreases the amount of network traffic required during the shuffle and sort phase
  - Often also decrease the amount of work needed to be done by the Reducer
- Combiner is a mini Reducer
  - Runs locally on a single Mapper's output after partitioning, sorting and before spilling to disk if one specified
  - Output from the Combiner is sent to the Reducers
  - Input and output data types for the Combiner/Reducer must be identical
- Combiner and Reducer code are often identical
  - Technically, this is possible if the operation performed is commutative and associative

**VERY IMPORTANT: The Combiner may run once, or more than once, on the output from any given mapper**

# Reducer

- After the shuffle and sort phase is over, all the intermediate values for a given intermediate key are combined together into a list. This list is given to a Reducer
  - There may be a single Reducer, or multiple Reducers
  - This can be specified as part of the job configuration (see later)
  - All values associated with a particular intermediate key are guaranteed to go to the same Reducer
  - The intermediate keys, and their value lists, are passed to the Reducer in sorted key order
- The Reducer outputs zero or more final key/value pairs
  - Reducer outputs are written to HDFS
  - In practice, the Reducer usually emits a single key/value pair for each input key

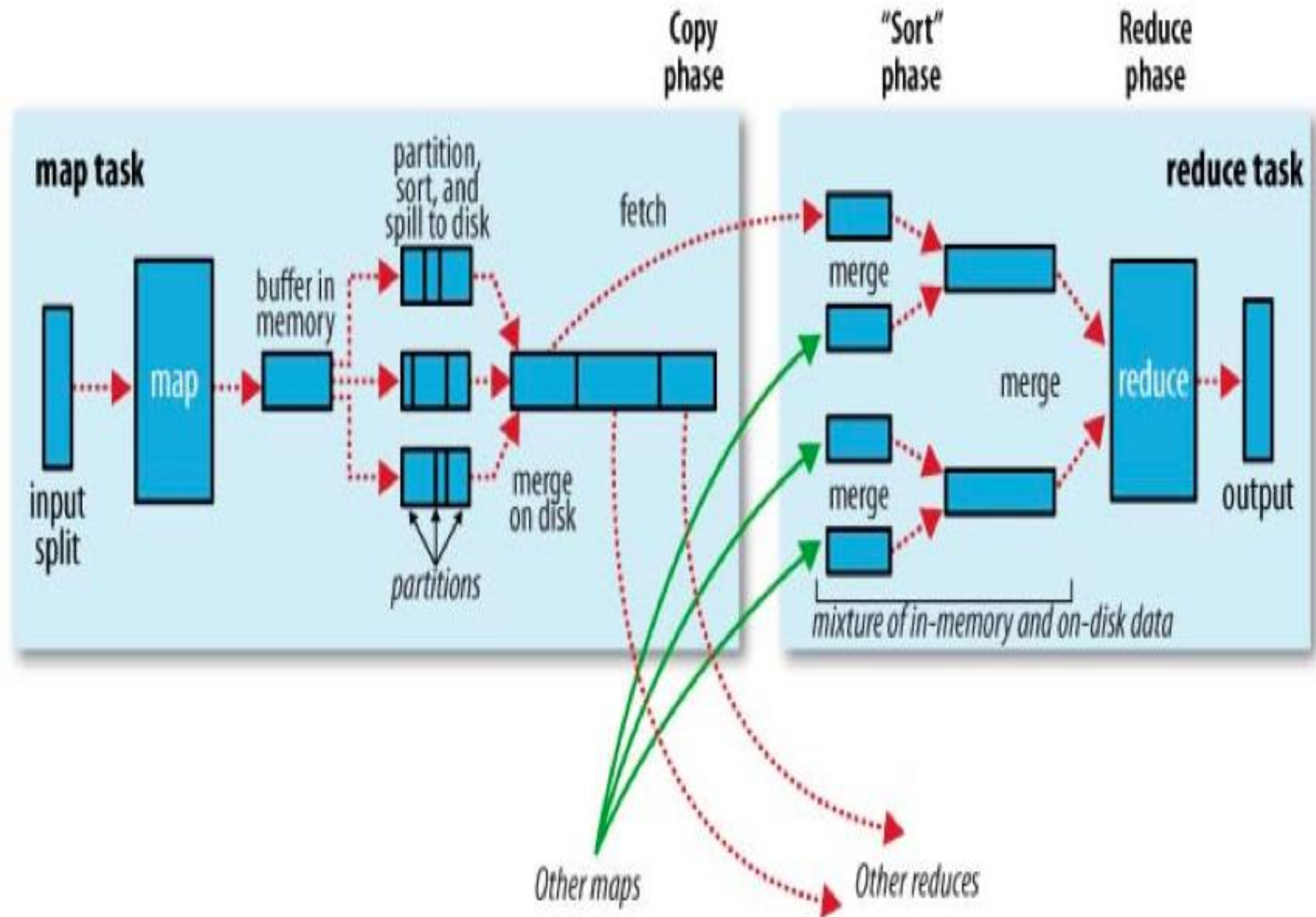
# Example Reducer: Sum Reducer

Add up all the values associated with each intermediate key  
(pseudocode)

```
let reduce(k, vals) =
 sum = 0
 foreach int i in vals:
 sum += i
 emit(k, sum)
```

```
('bar', [9, 3, -17, 44]) -> ('bar', 39)
('foo', [123, 100, 77]) -> ('foo', 300)
```

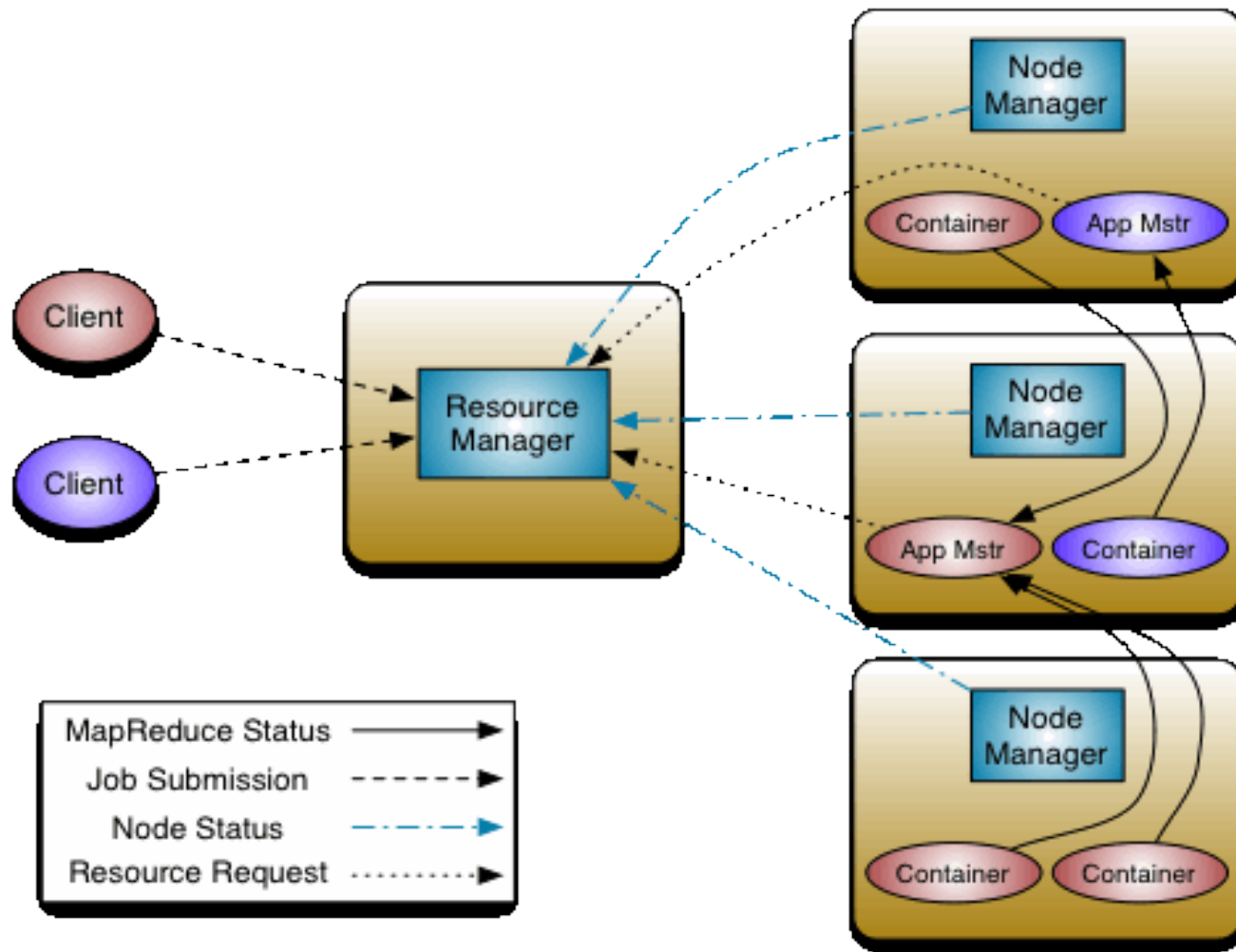
# A Different View of the MapReduce DataFlow



# MapReduce Version 2

- MapReduce version 2 also known as YARN which is a complete rewrite of Version 1 .YARN stands for “**Yet-Another-Resource-Negotiator**”.
- YARN is a generic platform for any form of distributed application to run on, while MR2 is one such distributed application that runs the MapReduce framework on top of YARN
- YARN can run applications that do not follow the MapReduce model, unlike the original Apache Hadoop MapReduce (also called MR1)
- The fundamental idea of MRv2 is to split up the two major functionalities of the JobTracker, resource management and job scheduling/monitoring, into separate daemons.

# MapReduce Version 2



# Thank You

## Disclaimer

Tech Mahindra Limited, herein referred to as TechM provide a wide array of presentations and reports, with the contributions of various professionals. These presentations and reports are for informational purposes and private circulation only and do not constitute an offer to buy or sell any securities mentioned therein. They do not purport to be a complete description of the markets conditions or developments referred to in the material. While utmost care has been taken in preparing the above, we claim no responsibility for their accuracy. We shall not be liable for any direct or indirect losses arising from the use thereof and the viewers are requested to use the information contained herein at their own risk. These presentations and reports should not be reproduced, re-circulated, published in any media, website or otherwise, in any form or manner, in part or as a whole, without the express consent in writing of TechM or its subsidiaries. Any unauthorized use, disclosure or public dissemination of information contained herein is prohibited. Unless specifically noted, TechM is not responsible for the content of these presentations and/or the opinions of the presenters. Individual situations and local practices and standards may vary, so viewers and others utilizing information contained within a presentation are free to adopt differing standards and approaches as they see fit. You may not repackage or sell the presentation. Products and names mentioned in materials or presentations are the property of their respective owners and the mention of them does not constitute an endorsement by TechM. Information contained in a presentation hosted or promoted by TechM is provided “as is” without warranty of any kind, either expressed or implied, including any warranty of merchantability or fitness for a particular purpose. TechM assumes no liability or responsibility for the contents of a presentation or the opinions expressed by the presenters. All expressions of opinion are subject to change without notice.