



# Outline

1. Main Features of Hadoop.
2. Data Storage: Hadoop Distributed File System.
3. Data Processing: Map – Sort – Reduce.

# Outline

1. Main Features of Hadoop.
2. Data Storage: Hadoop Distributed File System.
3. Data Processing: Map – Sort – Reduce.

# Main Features of Hadoop

- ❑ Apache Hadoop is a data framework for storing and processing *big data* distributed on clusters of “commodity” machines.



# Main Features of Hadoop

- ❑ Commodity server:
  - A piece of fairly standard hardware that can be purchased at retail, and have any particular software installed on it (e.g Linux, Hadoop).

	Commodity machine	Super computer
Performance	Low	High
Cost	Low	High
Availability	Readily available	Hard to obtain



# Main Features of Hadoop

## MongoDB vs Hadoop: Main Differences

### Hadoop:

#### ❑ Offline Data Processing:

- Massively Scalable Data Processing (hundreds of GBs or TBs).
- You need to store and process all this data.
- Most use cases: e.g. Post-processing of data generated by sensors.
- Client interaction: Order of minutes / hours.



# Main Features of Hadoop

## MongoDB vs Hadoop: Main Differences

### MongoDB:

#### ❑ Online Data Processing:

- You usually don't work with a lot of data at the same time (even if MongoDB is capable of storing GBs or TBs of info), but you only have to deal with a small subset of the data.
- Most use cases: Central part of an online platform → Storing a set of clients, their characterisation, interaction, etc.
- Client interaction: Order of milliseconds / seconds.



# Main Features of Hadoop

- ❑ Hadoop is based on the work done by Google in early 2000:
  - Specifically on the Google papers describing the Google File System: GFS (2003) and MapReduce (2004).
- ❑ It improves past distributed data frameworks by taking a *radical* new approach to the problem of distributed computing.

*The entire system was designed specifically for dealing with the storage and processing challenges of big data.*



# Main Features of Hadoop

## Hadoop Main Features:

- ❑ Data is distributed among the nodes of the cluster for both its storage and processing.
  - The goal is that each individual node works as much as possible on the data it hosts, minimising the need to talk to other nodes of the cluster (and thus minimising the data transferred over the network).
  - Data is replicated among different nodes of the cluster for redundancy, availability and fault tolerant-*free* computations.

# Main Features of Hadoop

## Hadoop Main Features:

- ❑ The system is fault tolerant on the cluster nodes:
  - If a node is unexpectedly shut down, the system is still ready to store and process the entire dataset without suffering a noticeable penalisation in the performance to achieve it.
  - At failure, the workload of the node is assumed by any other alive node of the cluster, without any loss of data. Thus, the outcome of the computation does not become affected.
  - When the node recovers, it just joins again the set of operative nodes, becoming available again for storing and processing data.

# Main Features of Hadoop

## Hadoop Main Features:

- ❑ The system is scalable:
  - The addition of new nodes is easy, and provide extra storage and processing capacity.
  - As the nodes cooperate on the tasks, the system is capable of dealing with peak workloads, which result only in a slight penalty in overall performance.

# Main Features of Hadoop

*For all these reasons Hadoop is said to be a highly-used efficient, resilient and scalable data framework.*

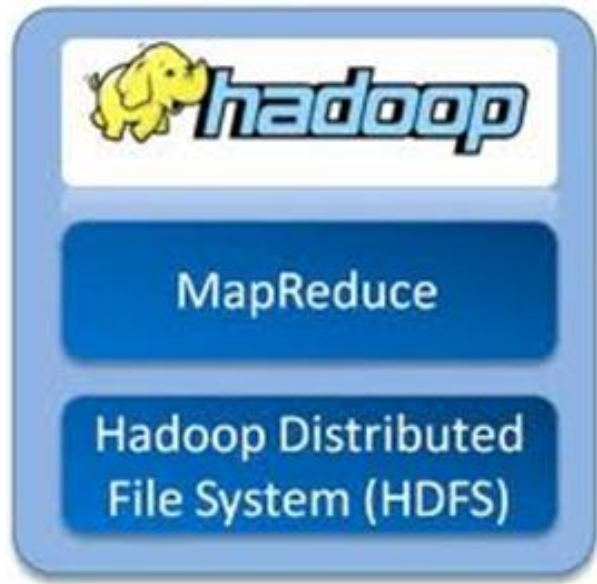


# Outline

1. Main Features of Hadoop.
2. Data Storage: Hadoop Distributed File System.
3. Data Processing: Map – Sort – Reduce.

# Data Storage: HDFS

- ❑ Hadoop achieves big data storage via one of its two main components: The Hadoop Distributed File System (HDFS).
- ❑ It is a *distributed* file system designed to efficiently allocate data across the multiple commodity machines (nodes) of the cluster, also providing self-healing functions when some of these nodes go down.
- ❑ The HDFS is completely independent of the other main component, MapReduce, which is used for data processing (and we will study in the next section).



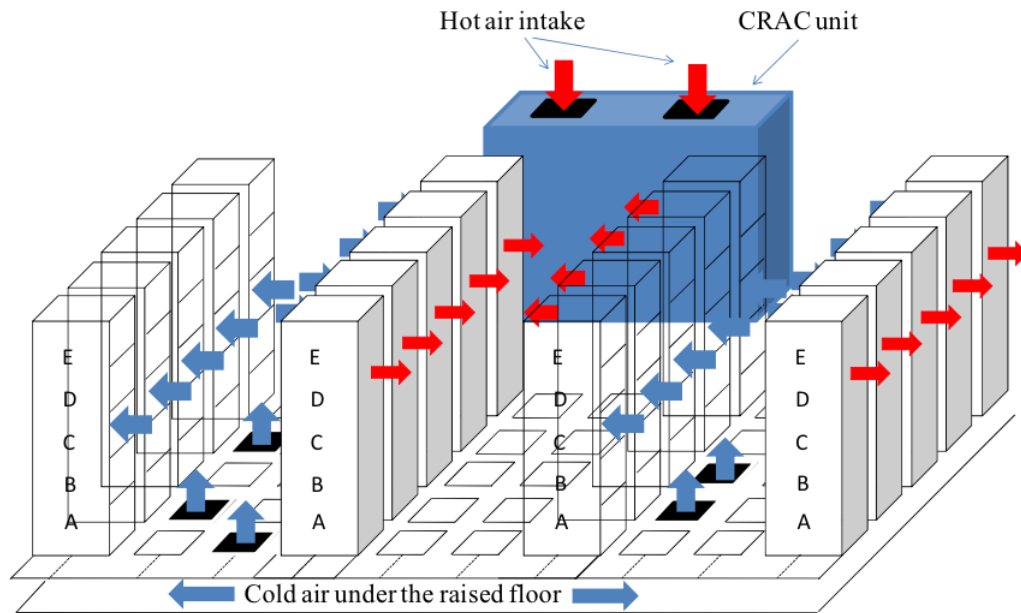
# Data Storage: HDFS

- ❑ To describe the Hadoop cluster and how HDFS operates on top of it we are going to compare it with some of the features of a MongoDB cluster studied weeks ago.

# Data Storage: HDFS

## Cluster Size:

- ❑ A Hadoop cluster consists of  $n \geq 1$  nodes, placed either in a single data centre or in geographically distributed data centres.





# Data Storage: HDFS

## Cluster Size:

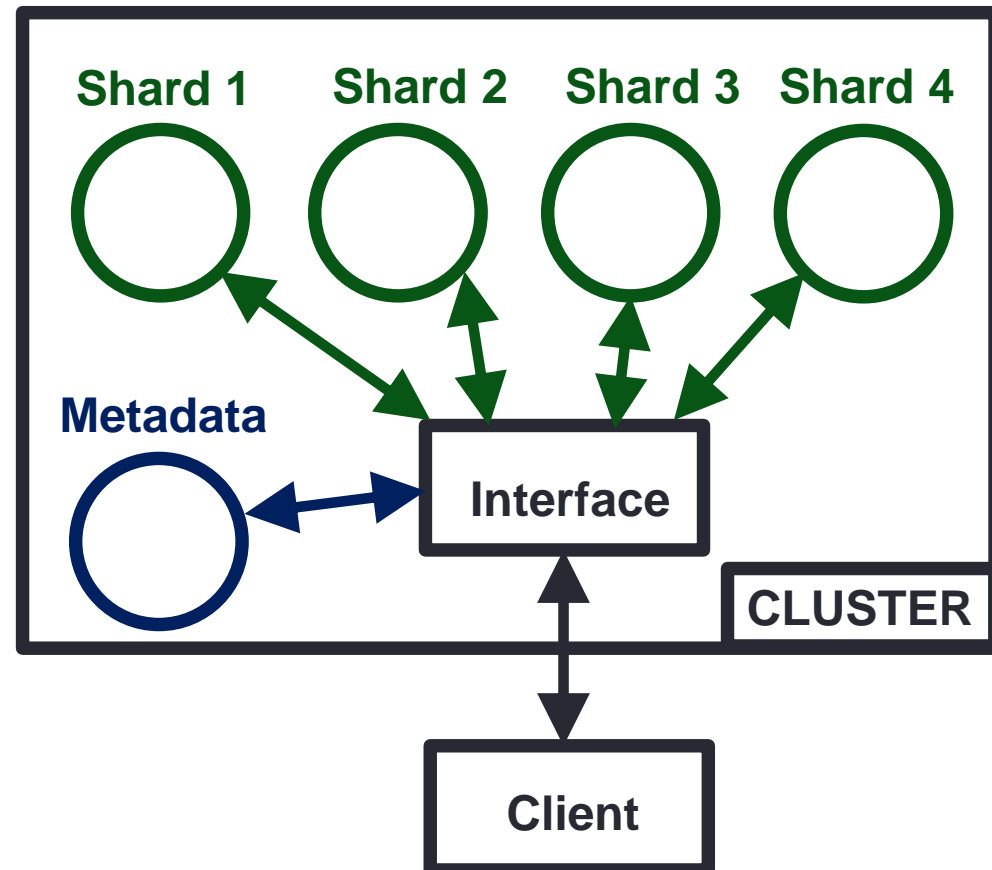
- ❑ If a cluster contains just 1 node then the cluster is said to work in pseudo-distributed mode (as actually all the daemons operating in Hadoop still take place, but all of them operate in a single machine).
  - This is similar to our “simulated-cluster” in MongoDB, where all the nodes were actually placed on one single machine.
  
- ❑ On the other hand, a cluster can operate with  $n > 1$  nodes. Indeed, the more the nodes the better the performance of Hadoop, so a real-world production example of a Hadoop cluster can contain thousands of nodes.

# Data Storage: HDFS

## Node Classification:

❑ In MongoDB we had two kind of nodes:

1. Shard nodes, responsible of hosting chunks of data.
2. Metadata or config nodes, responsible for giving an entire view of the cluster by providing the set of available databases, collections, chunks of data per collection and how the chunks were distributed among the shards.

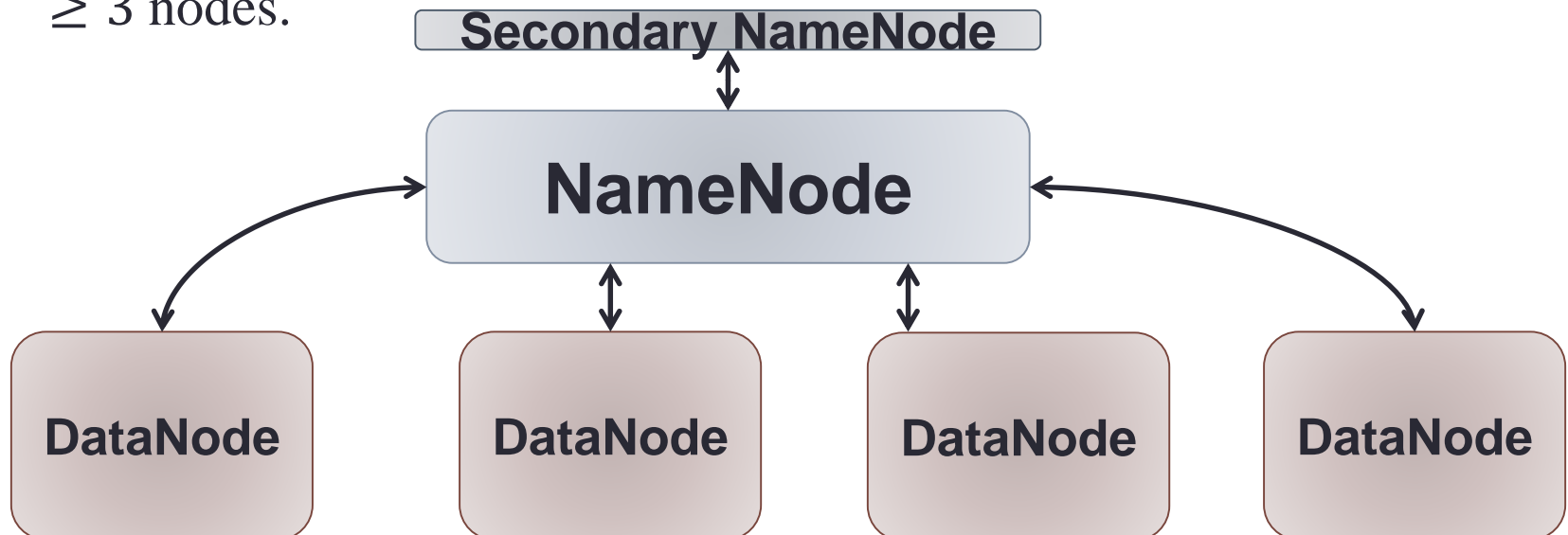


# Data Storage: HDFS

## Node Classification:

In Hadoop the classification is pretty much the same:

1. The shard nodes are called now called the DataNodes, but they are still the ones actually containing the data.
  - Perhaps the biggest difference is that each DataNode is a truly single node, whereas a Shard was typically formed as the replica set of  $n \geq 3$  nodes.

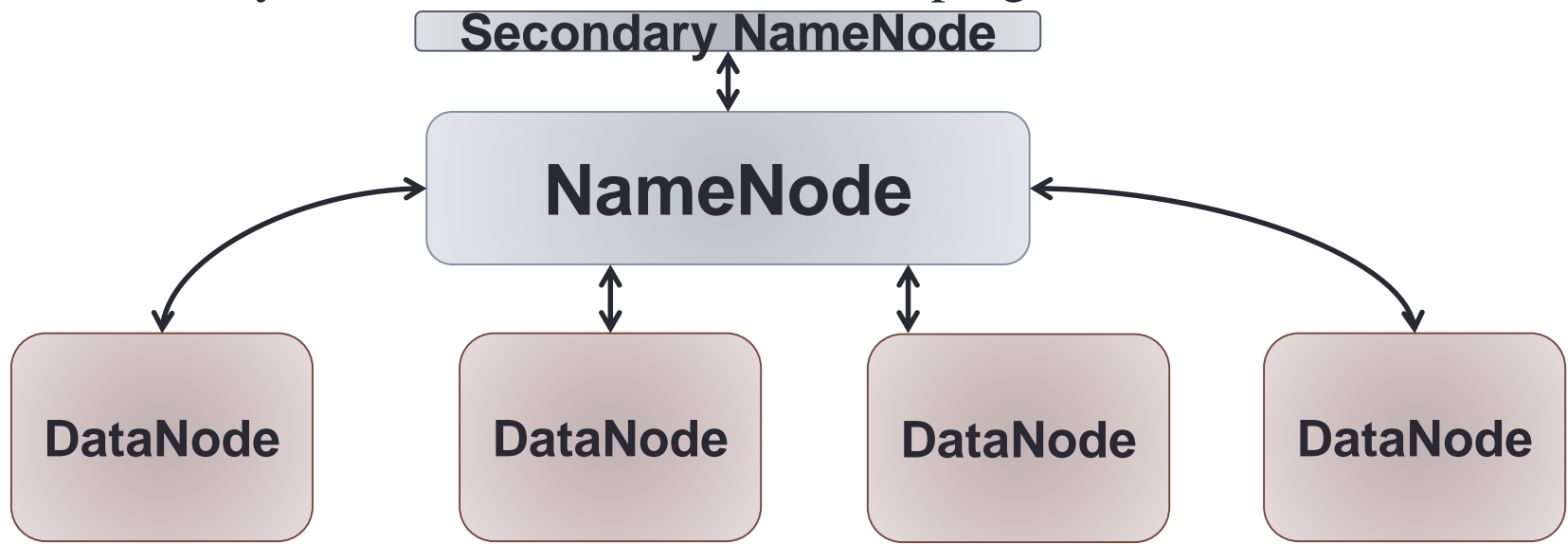


# Data Storage: HDFS

## Node Classification:

Now in Hadoop the classification is pretty much the same:

2. The config nodes are now called the NameNode, but it is still the one containing the metadata (i.e., table of contents giving the view of the entire cluster by knowing who hosts what).
  - There is one single NameNode operating at any moment, with a Secondary NameNode for the housekeeping of some tasks (not backup!)

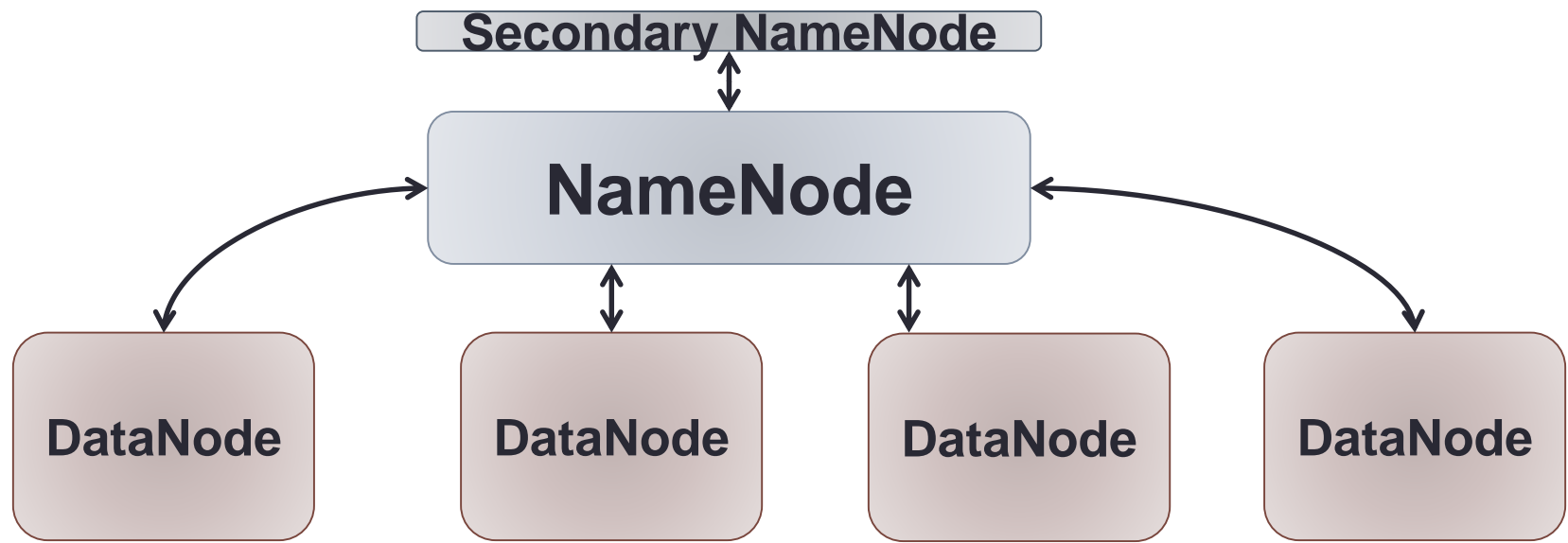


# Data Storage: HDFS

## Node Classification:

Overall, the way of dealing with the management of data is pretty much the same:

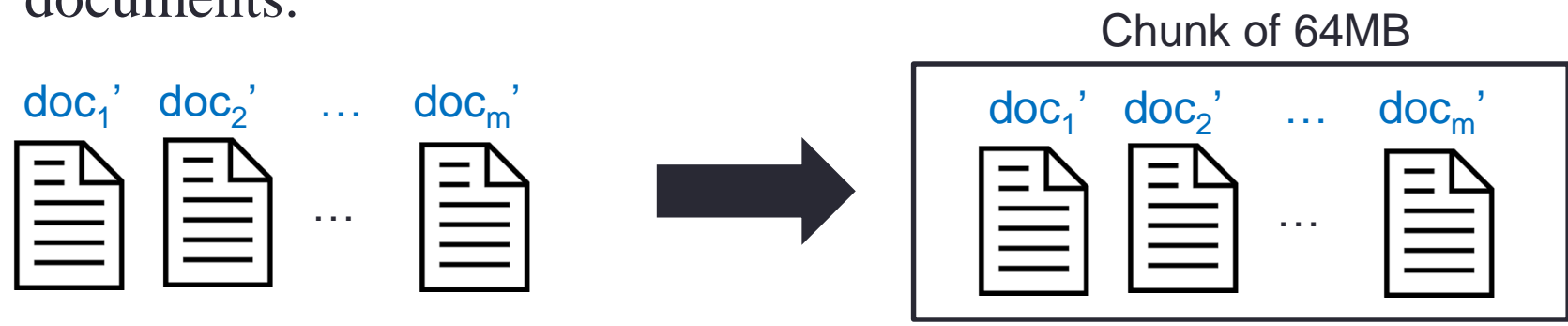
- ❑ The NameNode knows which files are to be processed, and which concrete DataNodes contain each of these files.



# Data Storage: HDFS

## Data Entity:

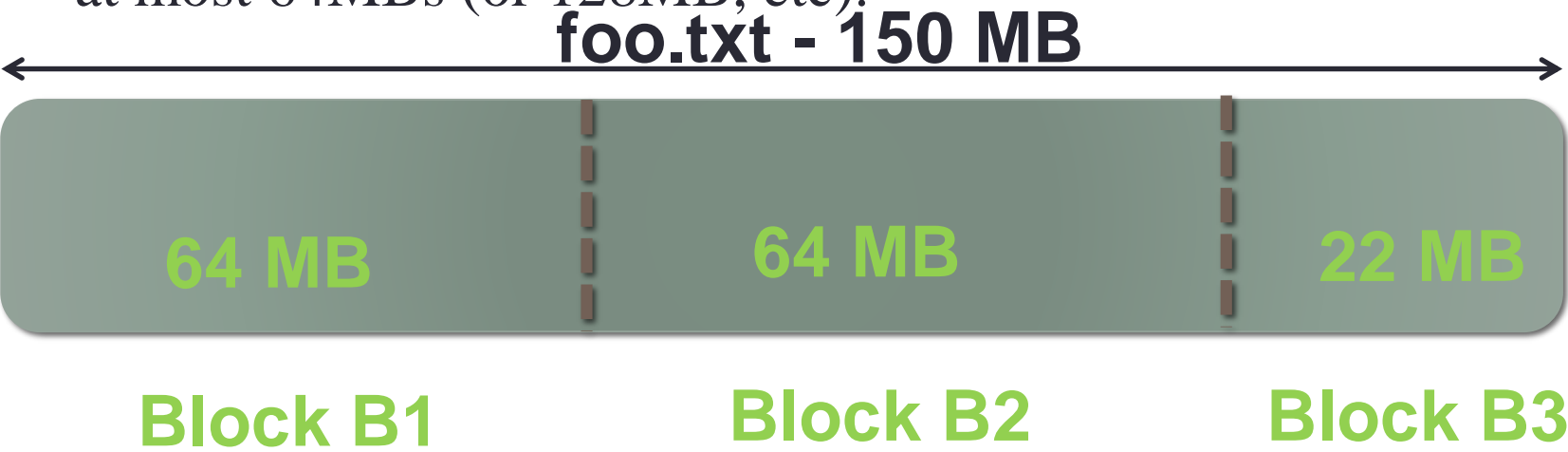
- ❑ As MongoDB does, Hadoop has the notion of a data entity or chunk of data, which consists of a block of information of about 64MB (128MB in Hadoop V2) by default – this compares with a typical block size in Linux of 4K.
- ❑ However, here the approach is different:
  - To come up with a chunk, in MongoDB one has to aggregate documents.



# Data Storage: HDFS

## Data Entity:

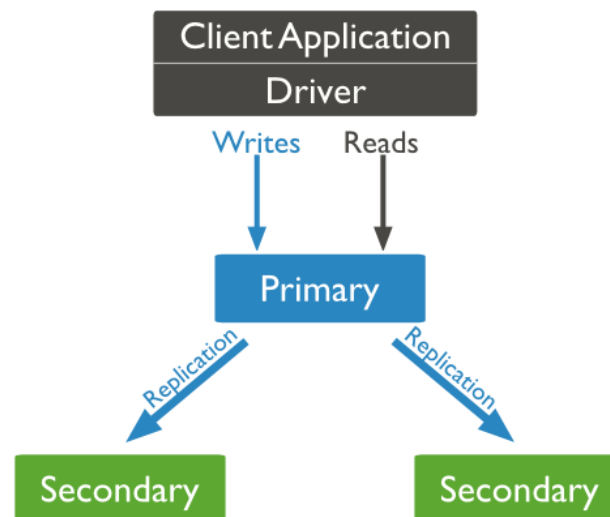
- On the other hand, Hadoop is typically for working with extremely large files: Let's say files of hundreds of MBs or even GBs.
  - Thus, Hadoop is happy with a file occupying less than 64MB (it leaves it as it is, does not aggregate it to anybody else).
  - But it does split any file over 64MBs into **blocks** (or chunks) of at most 64MBs (or 128MB, etc).



# Data Storage: HDFS

## Data Replication:

- ❑ As in MongoDB, Hadoop replicates each chunk or block of information  $k$  times (where  $k \geq 3$ ).
  - Perhaps the biggest difference is that in MongoDB replication was done at the level of a shard (or replica set). That is, each shard contains  $k$  nodes, all of them with the very same information.

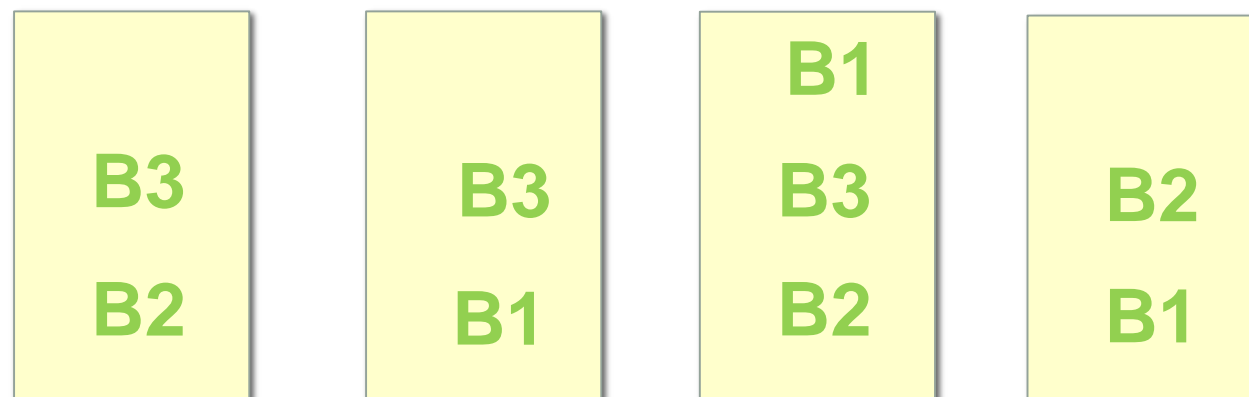




# Data Storage: HDFS

## Data Replication:

- ❑ As in MongoDB, Hadoop replicates each chunk or block of information  $k$  times (where  $k \geq 3$ ).
  - In Hadoop replication is done at the level of **blocks**. That is, each block (either a file or *a part of a file* if the file was split into several blocks) is to be replicated 3 times. The 3 replicas of the block are placed into 3 different nodes (e.g., place block B2 at nodes n1, n3 and n4).



# Data Storage: HDFS

## Data Replication:

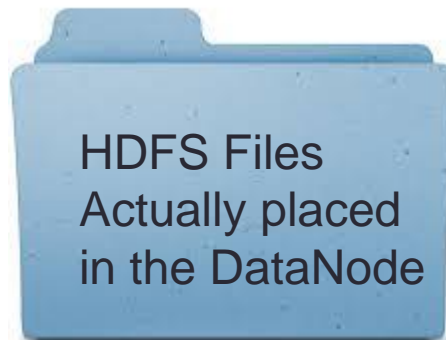
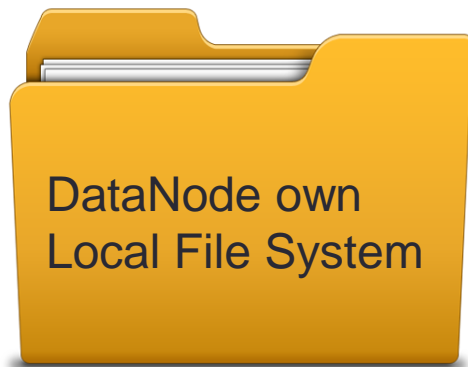
- ❑ As in MongoDB, Hadoop replicates each chunk or block of information  $k$  times (where  $k \geq 3$ ).
  - However, this does not mean that nodes  $n1$ ,  $n3$  and  $n4$  contain exactly the same information!  
(as you can see, they share block  $b2$ , but not the rest of blocks).



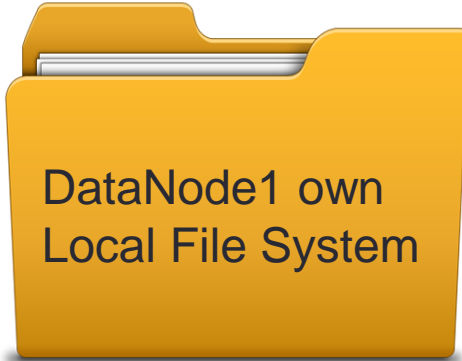
# Data Storage: HDFS

## Local File System vs. Hadoop File System:

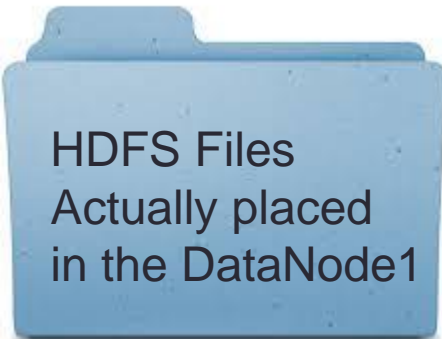
- ❑ Each DataNode contains isolated name spaces for:
  - I. Its own local file system.
  - II. The Hadoop Distributed file system (actually, the sub-part of the HDFS placed into it).



# Data Storage: HDFS



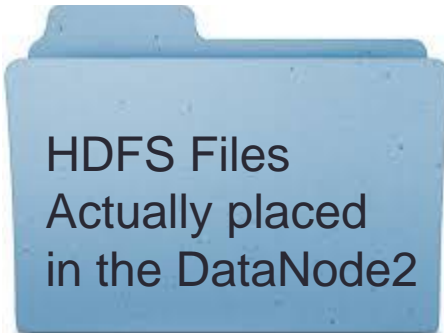
DataNode1 own  
Local File System



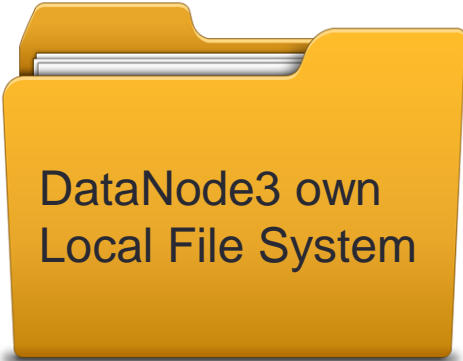
HDFS Files  
Actually placed  
in the DataNode1



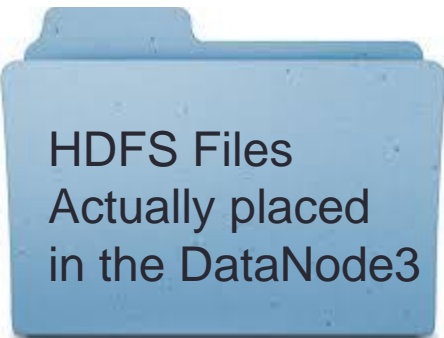
DataNode2 own  
Local File System



HDFS Files  
Actually placed  
in the DataNode2



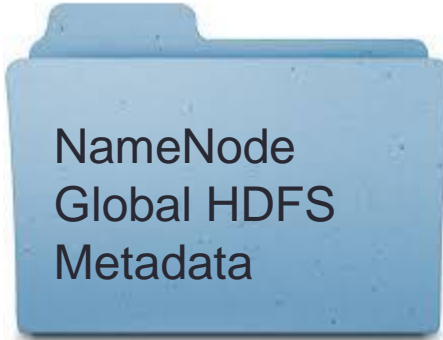
DataNode3 own  
Local File System



HDFS Files  
Actually placed  
in the DataNode3



NameNode own  
Local File System



NameNode  
Global HDFS  
Metadata

Without metadata of  
NameNode it will be  
impossible to know  
where each file/block is

# Data Storage: HDFS

## Local File System vs. Hadoop File System:

*Transfer of information:*

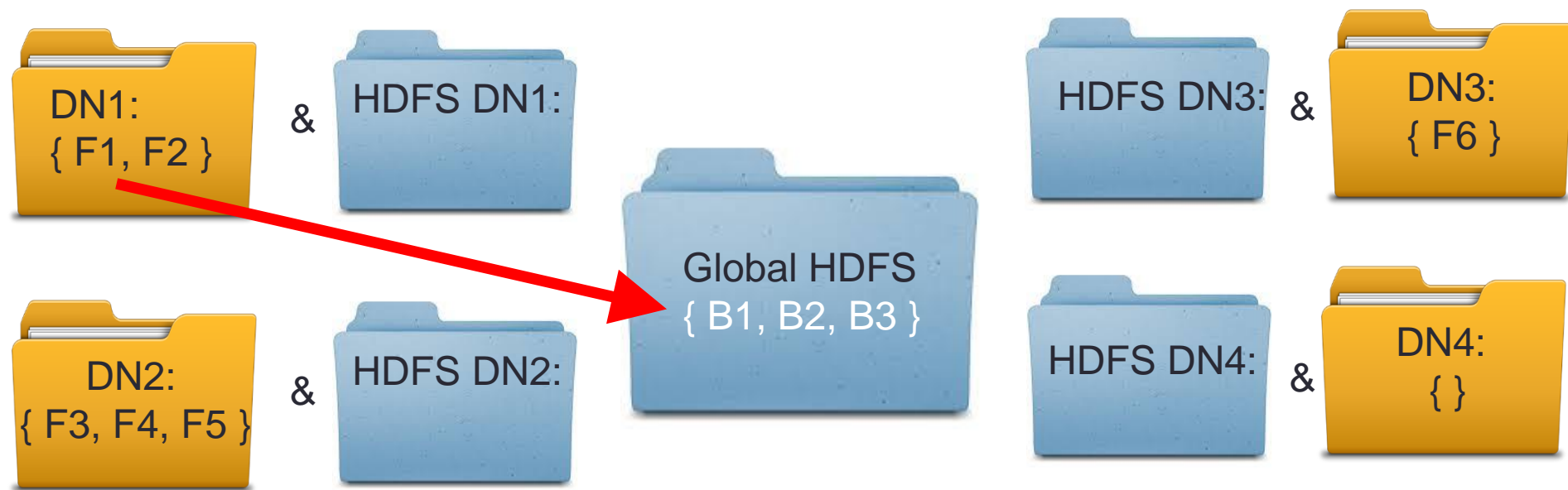
- ❑ So, a **DataNode** can add (put) some files  $\{F1, F2, \dots, Fk\}$  from its own Local File System files to the HDFS.
- ❑ To do so, it contacts the **NameNode**, which:
  1. Splits the files  $\{F1, F2, \dots, Fk\}$  to the actual blocks  $\{B1, B2, \dots, Bn\}$  these files lead to.
  2. Decides which concrete DataNodes (DNa, DNb, DNC) are going to be hosting the 3 replications of each block  $B_i$  (different nodes for each block).

# Data Storage: HDFS

## Local File System vs. Hadoop File System:

*Transfer of information:*

1. Imagine DataNode1 transferring {F1, F2} to Global HDFS.
2. Imagine F1 is small (it leads to a single block B1) and F2 is big (it leads to the two blocks B2 and B3).

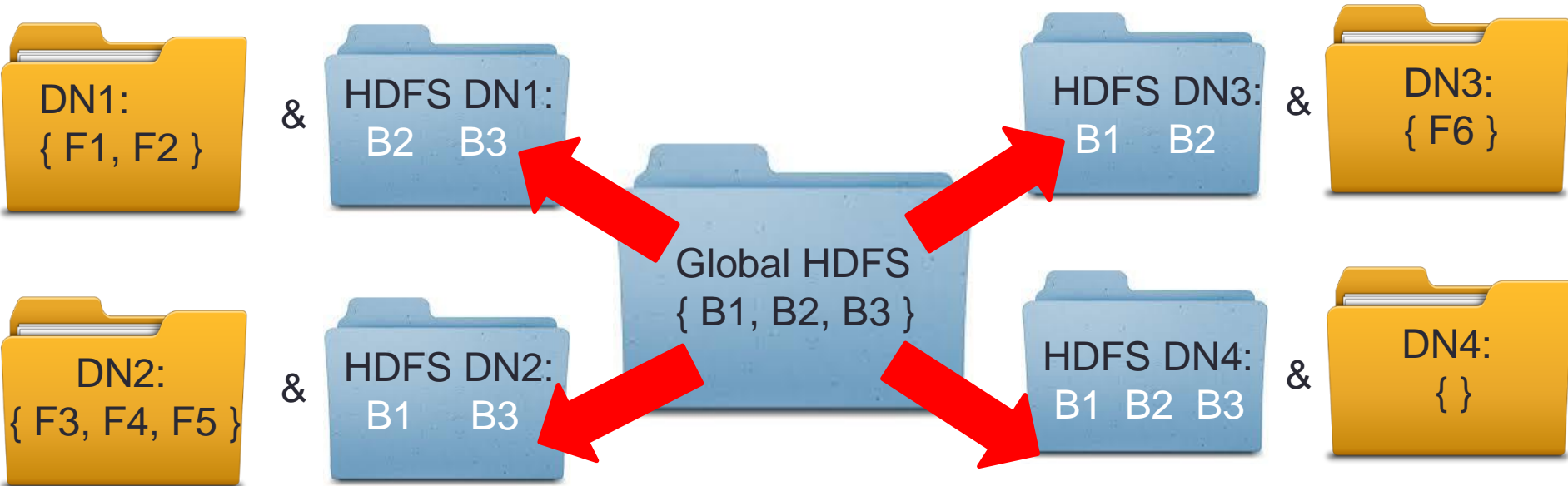


# Data Storage: HDFS

## Local File System vs. Hadoop File System:

*Transfer of information:*

3. NameNode decides the 3 nodes (DNa, DNb, DNC) hosting each block.



# Data Storage: HDFS

## Local File System vs. Hadoop File System:

*Transfer of information:*

- 4. NameNode stores metadata so as to know which nodes contain each block





# Data Storage: HDFS

## Local File System vs. Hadoop File System:

*Transfer of information:*

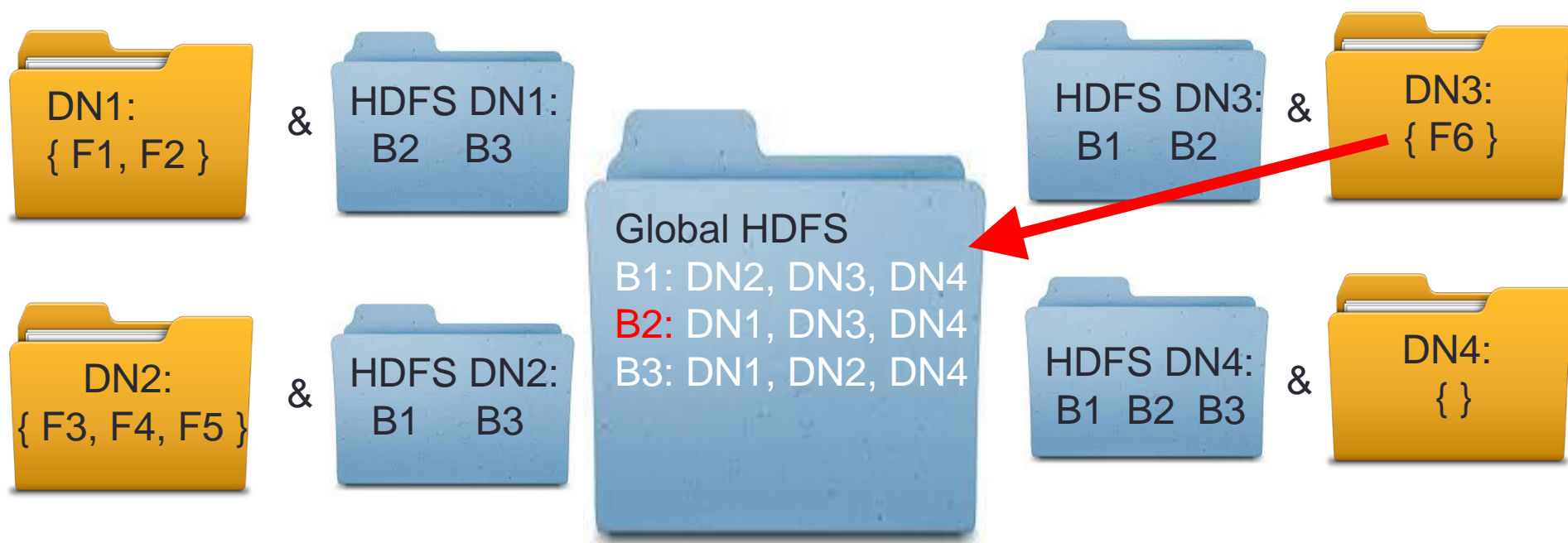
- ❑ Likewise, a **DataNode** can bring back (get) some blocks from HDFS to its own Local File System, so as to be able to access/edit them.
  - To do so, it contacts the NameNode, which uses its metadata to know the concrete DataNodes hosting the block.
  - NameNode asks these DataNodes to transfer the block.

# Data Storage: HDFS

## Local File System vs. Hadoop File System:

*Transfer of information:*

- 1. Imagine DataNode 3 wants to bring block B2 back to its own file system.
- 2. It asks NameNode for it.

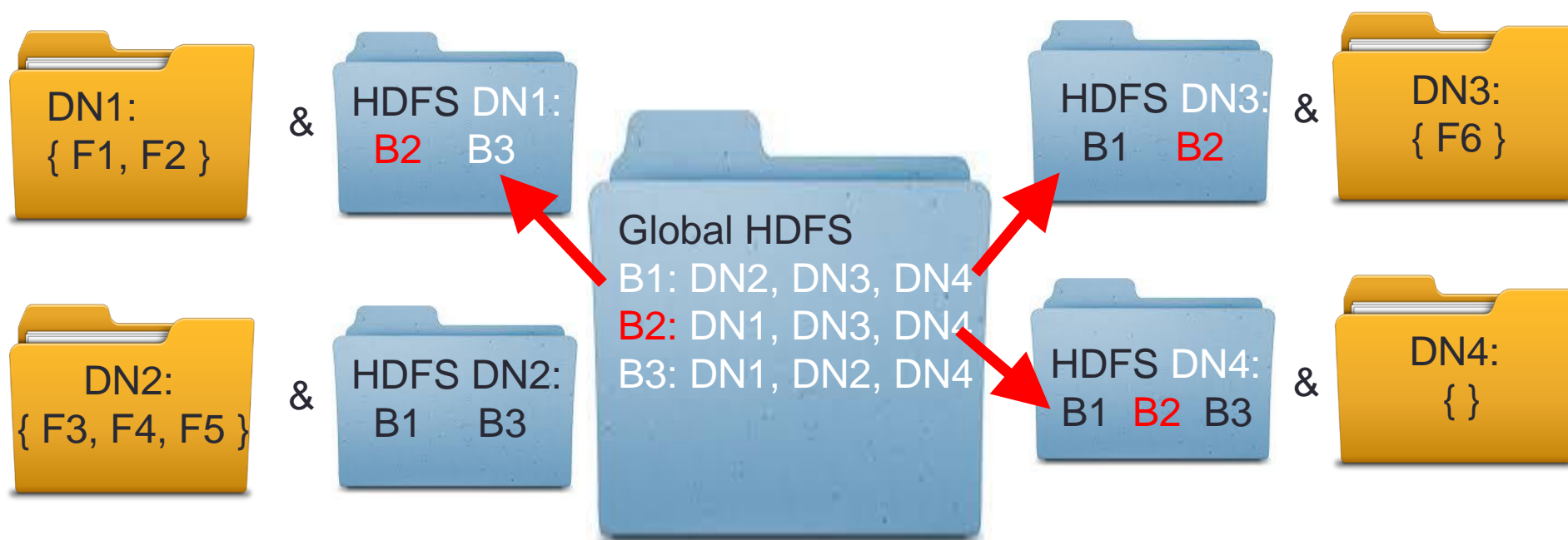


# Data Storage: HDFS

## Local File System vs. Hadoop File System:

*Transfer of information:*

4. NameNode checks metadata to know that B2 is at { DN1, DN3, DN4 }.

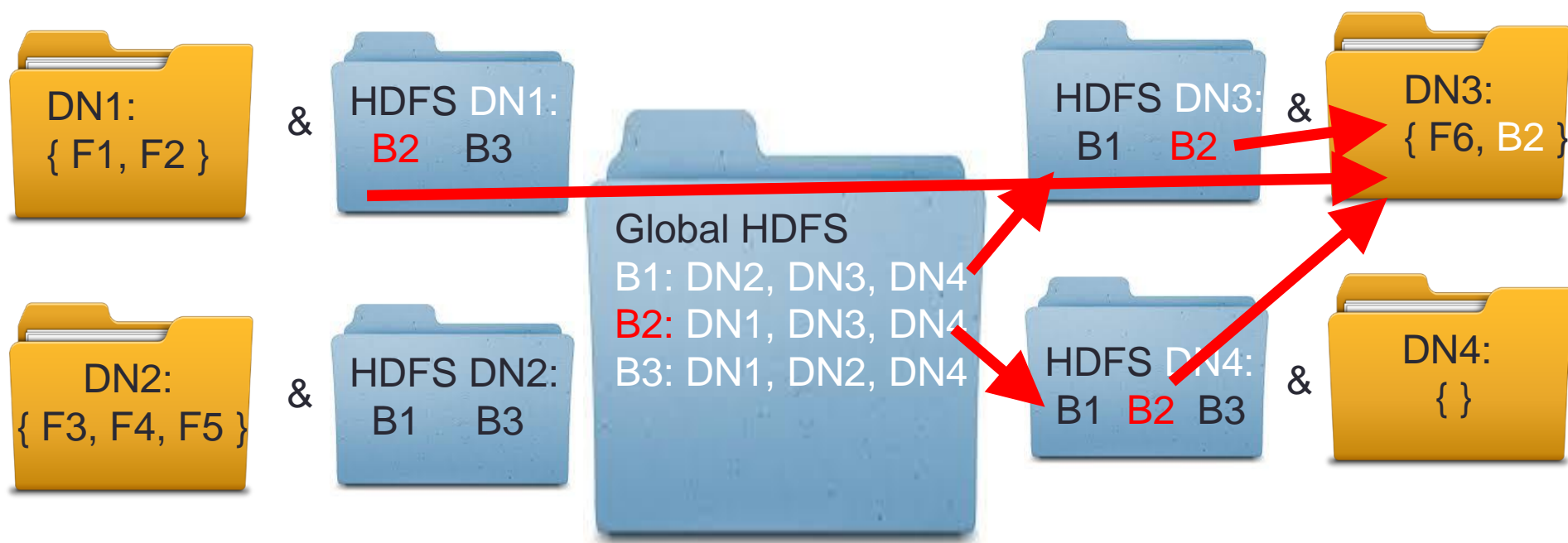


# Data Storage: HDFS

## Local File System vs. Hadoop File System:

*Transfer of information:*

- 5. The DataNodes transfer the block B2 back, so that it can be processed by the DataNode 3 on its own file system.

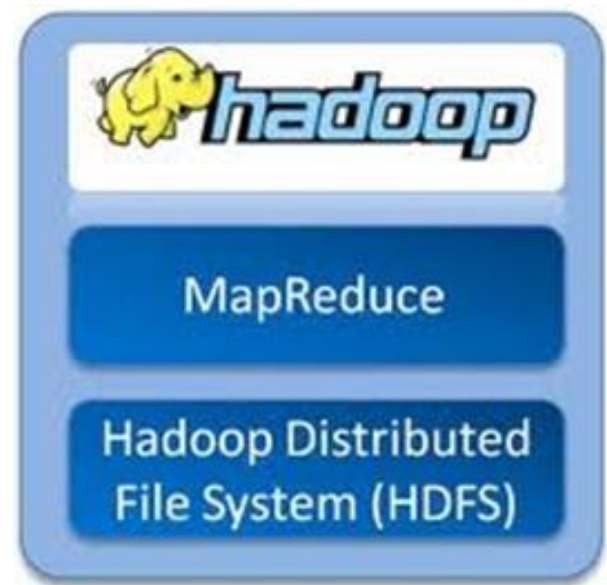


# Outline

1. Main Features of Hadoop.
2. Data Storage: Hadoop Distributed File System.
3. Data Processing: Map – Sort – Reduce.

# Data Processing: Map – Sort – Reduce

- ❑ Hadoop achieves big data processing via one of its two main components: The MapReduce Framework.
- ❑ It is:
  - An efficient programming framework
  - for processing parallelizable problems
  - across huge datasets
  - using a large number of machines.
- ❑ The MapReduce framework works on top of the HDFS.



# Data Processing: Map – Sort – Reduce

*Reusing our previous knowledge on MongoDB,  
we can define the MapReduce framework as:*

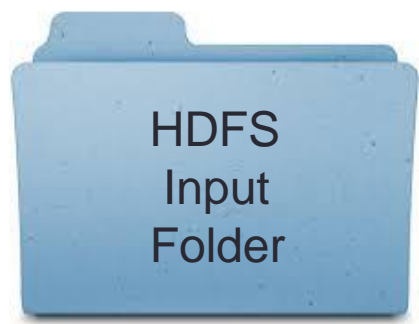
- ❑ **A pipeline process**  
**operating on files**  
that uses **streaming for communication**  
and **high-level programming features** to  
**isolate the data processing**  
**from both the cluster and the pipeline**  
**complexities.**

# Data Processing: Map – Sort – Reduce

## MapReduce Framework Pipeline Process

Input to the process:

- ❑ A HDFS folder *input\_folder*, containing {B1, B2, ..., Bn} blocks distributed among the DataNodes of the Cluster.
  - NameNode knows the 3 concrete DataNodes that contain each block Bi



- B1 → DataNodes D1, D3, D4.
- B2 → DataNodes D2, D4, D5.
- ...
- Bn → DataNodes D1, D4, D5.



# Data Processing: Map – Sort – Reduce

## MapReduce Framework Pipeline Process

Output from the process:

- ❑ A newly-generated HDFS folder *output\_folder*, containing new  $\{\mathbf{B1'}, \mathbf{B2'}, ..., \mathbf{Bk'}\}$  blocks (possibly distributed among the DataNodes of the cluster).
  - For each block  $\mathbf{B_i}$ , a single DataNode contains it.



$\mathbf{B1} \rightarrow \text{DataNode D1.}$   
 $\mathbf{B2} \rightarrow \text{DataNode D3.}$   
 $\dots$   
 $\mathbf{Bk} \rightarrow \text{DataNode D1.}$

# Data Processing: Map – Sort – Reduce

## MapReduce Framework Pipeline Process

Output from the process:

- ❑ A new generated HDFS folder *output\_folder*, containing new {B1', B2', ..., Bk'} blocks (possibly distributed among the DataNodes of the cluster).
- **Most of the times we just want a single output block B!**

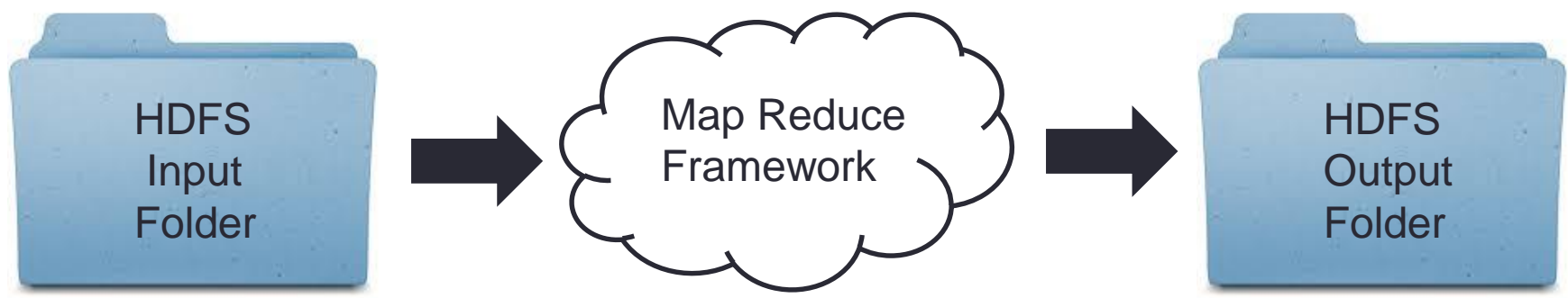


B → DataNode D1.

# Data Processing: Map – Sort – Reduce

## MapReduce Framework Pipeline Process

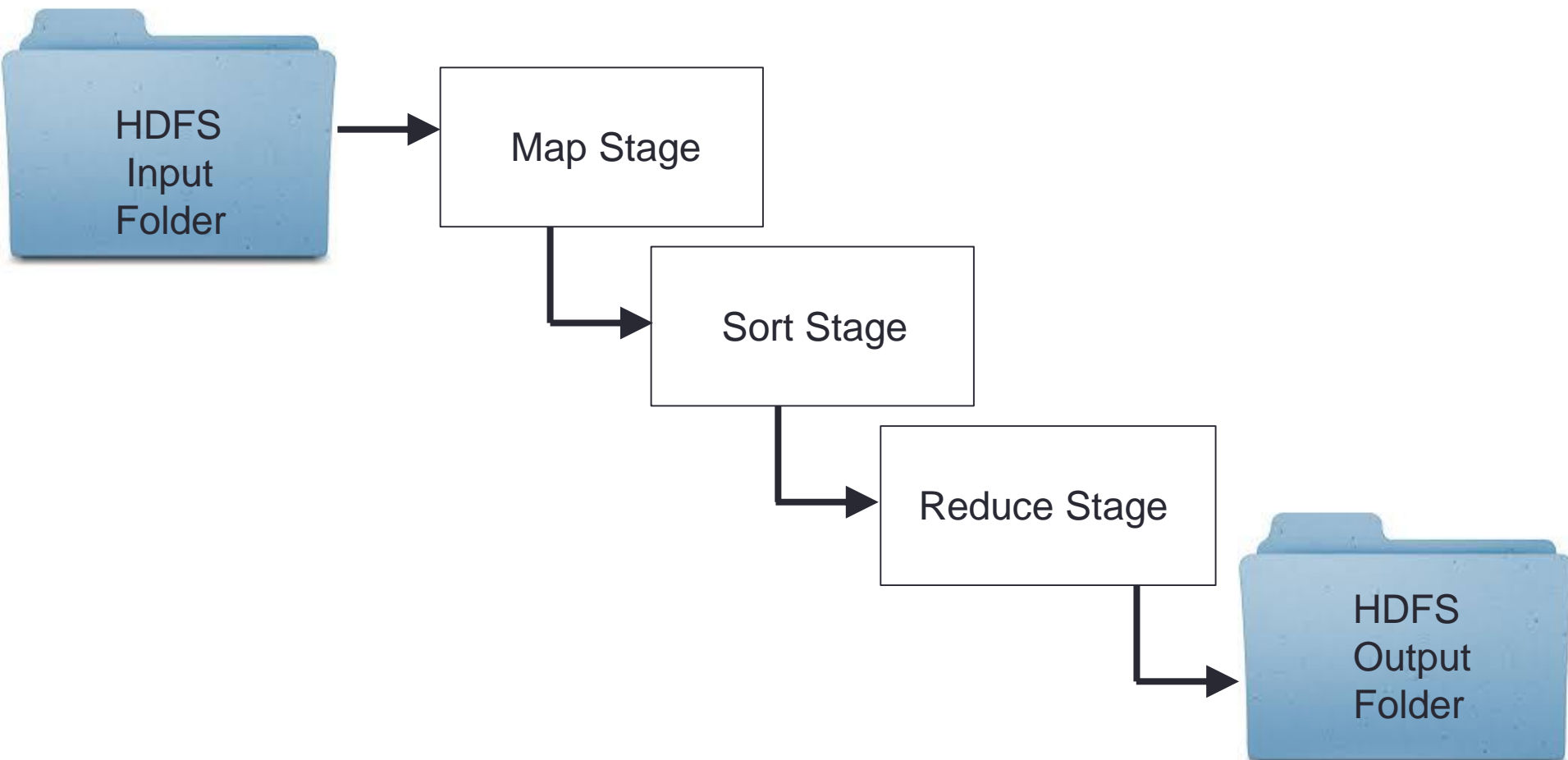
MapReduce Process as a black-box:



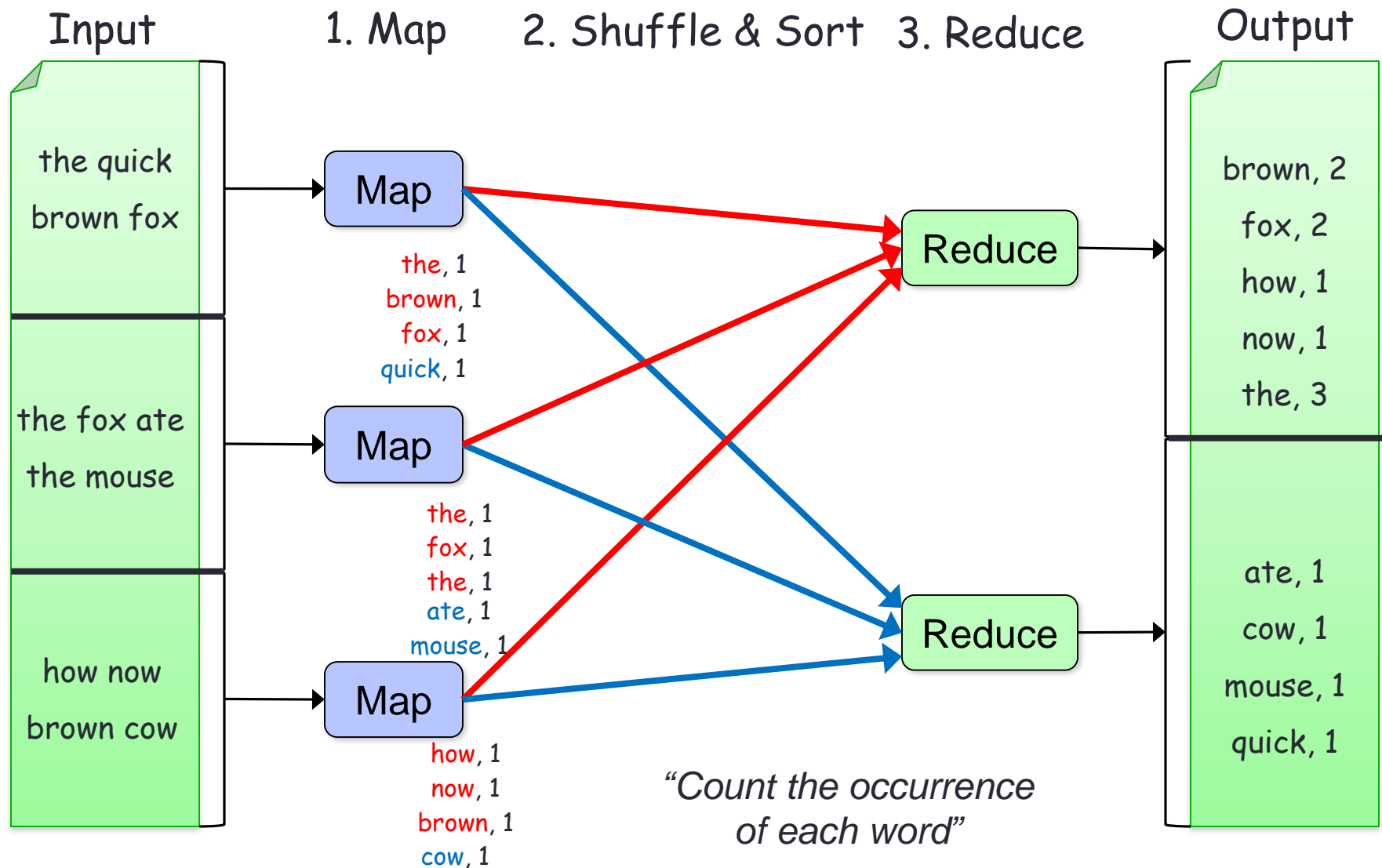
So, what's inside the black-box?

# Data Processing: Map – Sort – Reduce

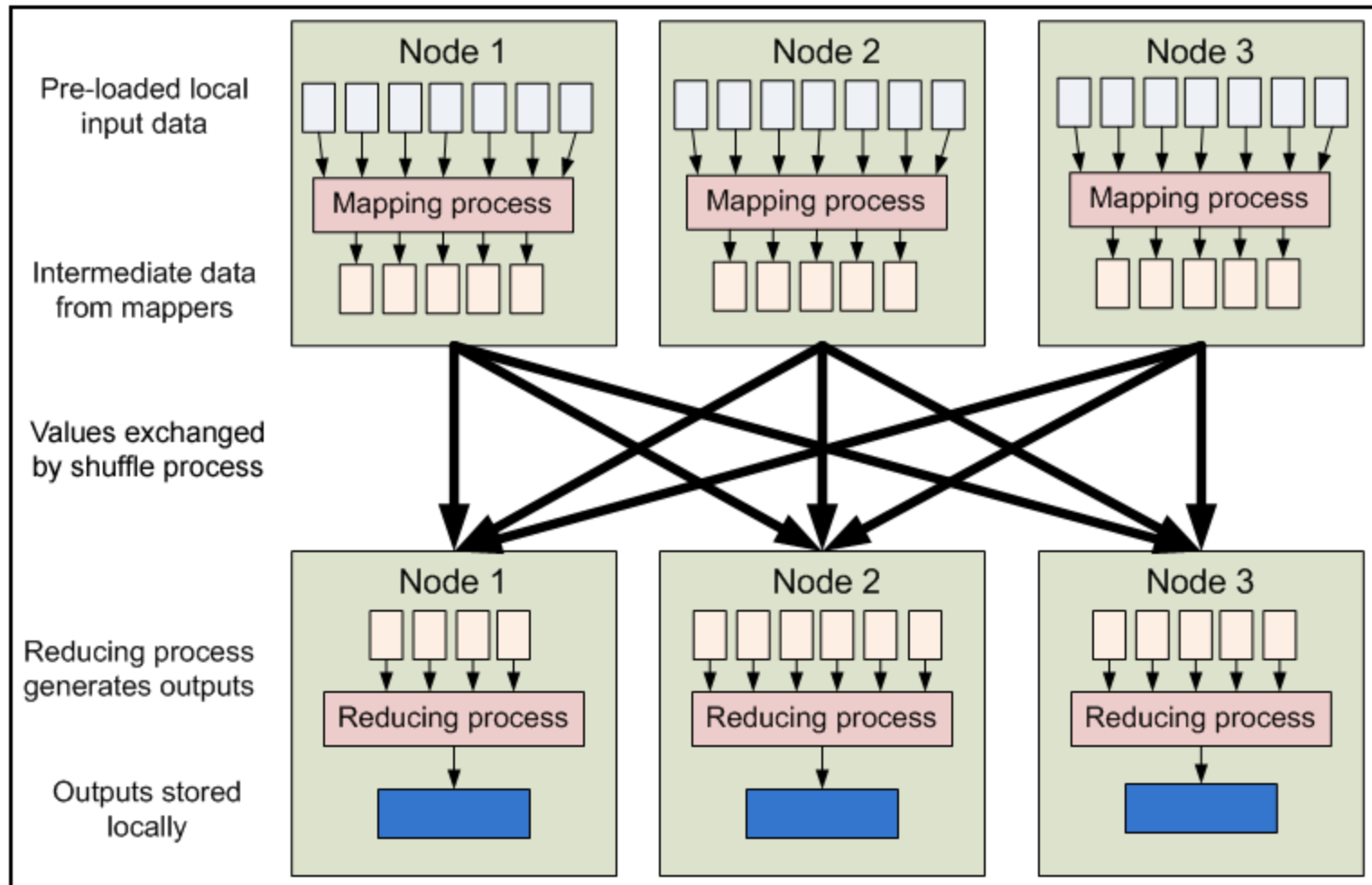
## MapReduce Framework Pipeline Process



# Data Processing: Map – Sort – Reduce

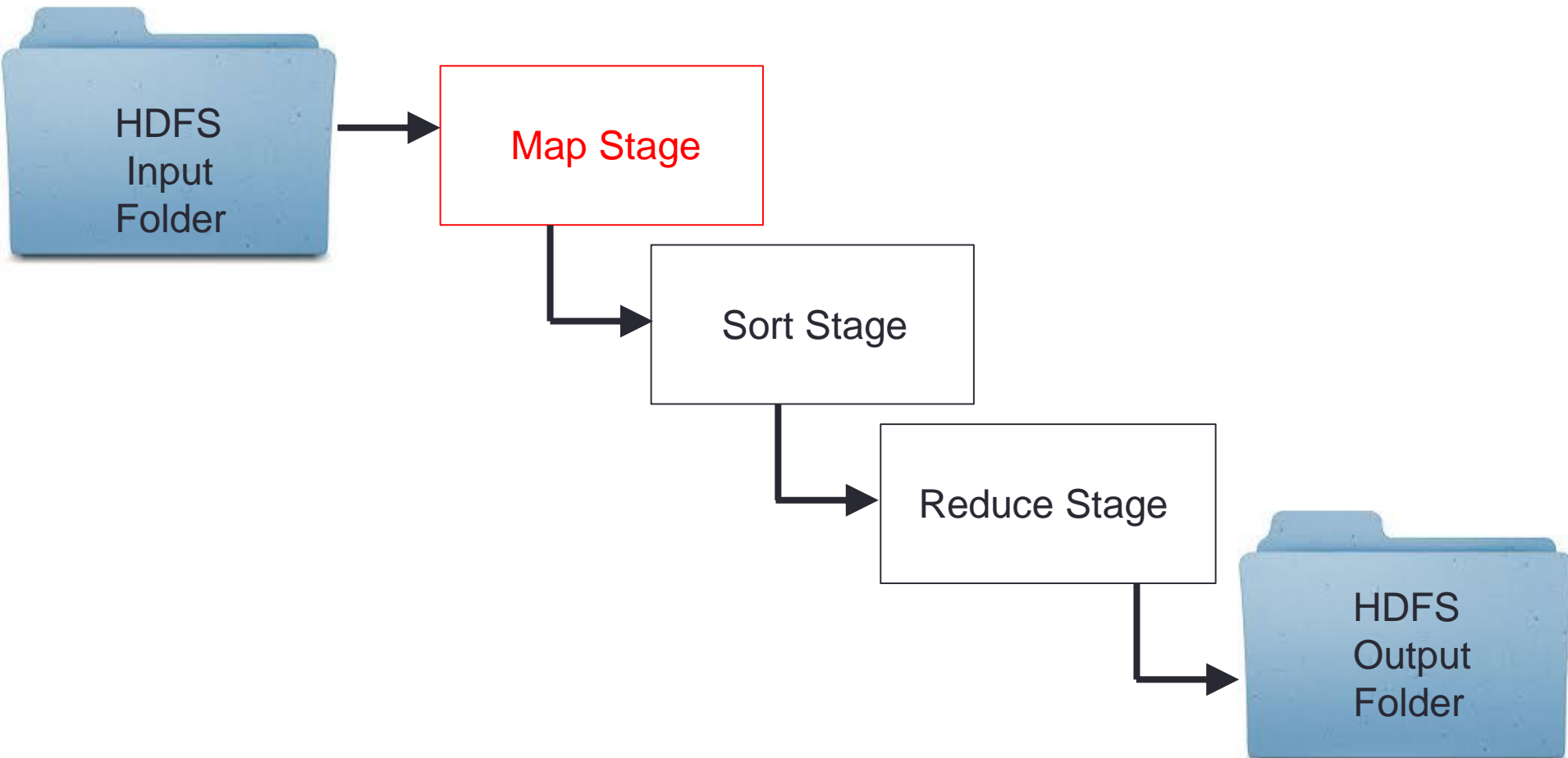


# MapReduce Framework



# Data Processing: Map – Sort – Reduce

## MapReduce Framework Pipeline Process



# Data Processing: Map – Sort – Reduce

## Map Stage

- ❑ The Map stage gives a first round of data processing, to compute intermediate results.
- ❑ It takes care of the entire set of blocks  $\{B1, B2, \dots, Bn\}$  contained in the HDFS folder.
  - So, if these blocks of data represent, altogether, 20GBs of information, the Map Phase processes 20GBs of information!



# Data Processing: Map – Sort – Reduce

## Map Stage

- ❑ The secret of the pyramid is the Divide and conquer approach:

*The 20GBs of data contained in  $\{B1, B2, \dots, Bn\}$  have to be processed, there is no escape from this. Wow, what a huge amount of work!*

*But... if these blocks are distributed among the DataNodes, and each block occupies at most 64MBs, then I can ask all DataNodes to do in parallel the processing of their local blocks!*

*This way, I'm reducing the work of processing 20GBs to the work of parallel processing multiple blocks of 64MBs →*

***That should take orders of magnitude less time!***

# Data Processing: Map – Sort – Reduce

## Map Stage

- ❑ But, to set up Map stage, a few questions have to be addressed:
  1. How many Map processes to create?
    - One per block?
    - One gathering multiple blocks (if the files are small enough)?
  2. Which DataNode is assigned to perform each Map?
    - Ideally, one wants to set each DataNode to work on the local data it's already storing, so as minimise the network transference.
    - But, on the other hand, one wants a fairly balanced distribution of the computation among the nodes, so as to be able to process the entire 20GBs as soon as possible.

# Data Processing: Map – Sort – Reduce

## Map Stage

### Key Concept:

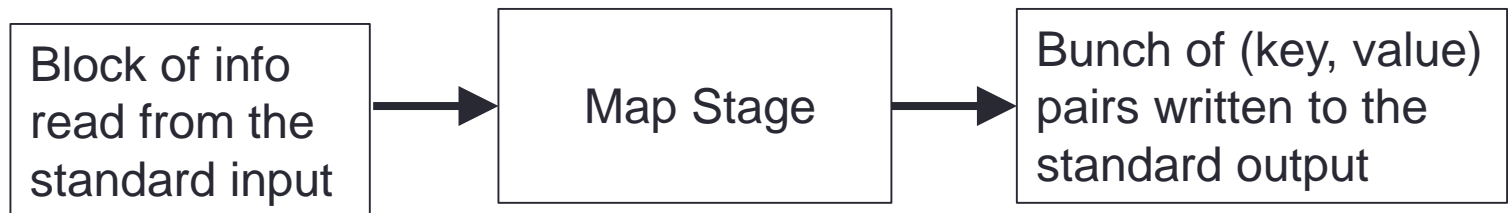
- ❑ As a user, you do not have to worry about these questions!  
Hadoop sets up **the amount of mappers**, **the assignment of blocks and DataNodes to mappers**, and the **proper network transference of blocks when needed**.

# Data Processing: Map – Sort – Reduce

## Map Stage

### Programming the mapper

- ❑ So, what do we do with each block of 64 MBs?
- ❑ How do we process it?

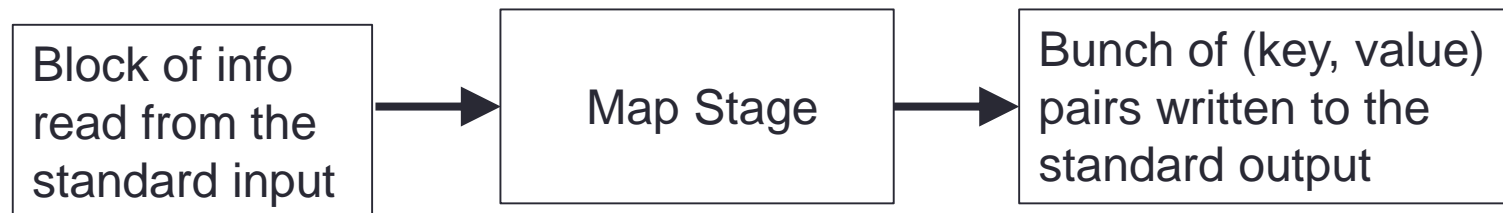


# Data Processing: Map – Sort – Reduce

## Map Stage

### Programming the mapper

- ❑ Unfortunately, how to program the mapper is not very open.
  - You know you are receiving the content of an input file from the streaming standard input. Not even receiving the file, just its content!
  - In this case, all you can do is to process this content!

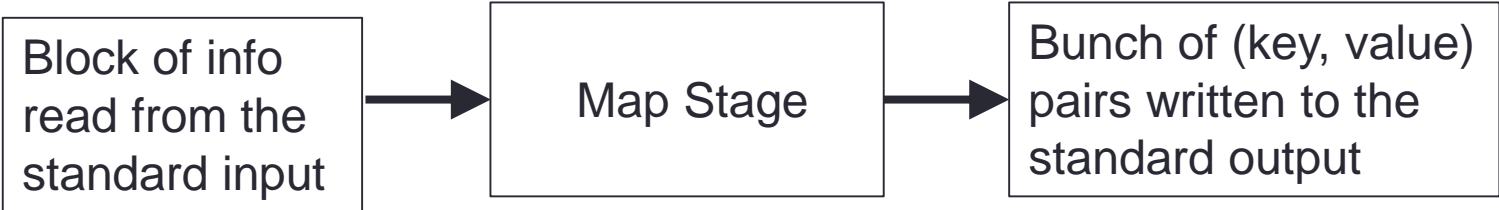


# Data Processing: Map – Sort – Reduce

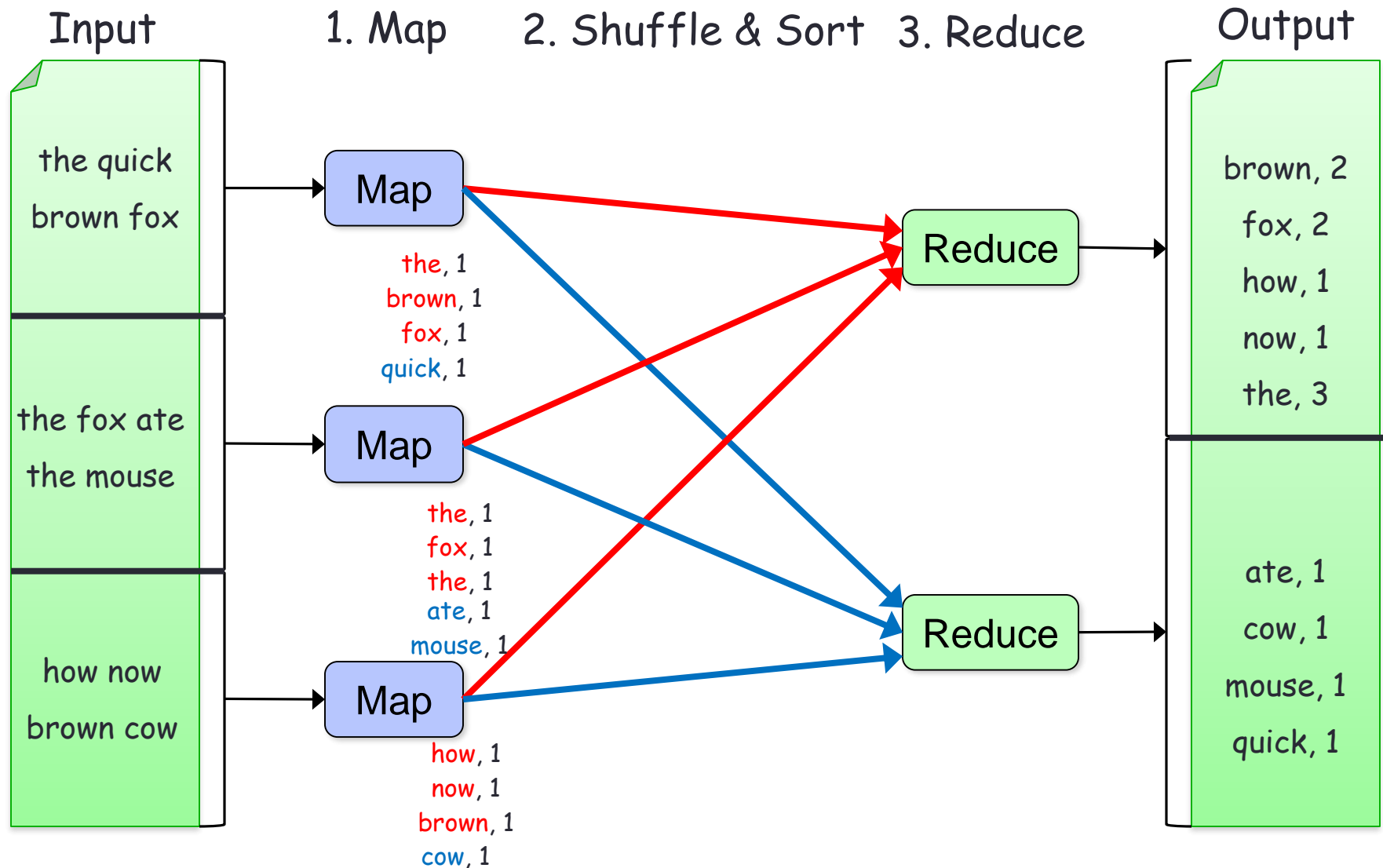
## Map Stage

### Programming the mapper

- ❑ If that was not enough, you're also limited with the output.
  - Do whatever you want for processing the file content, **but...**
  - Your output must be a set of key-value pairs.
    - That you must write as a bunch of lines to the standard output! 🤨
    - This is kind of counterintuitive, but is how it actually works.

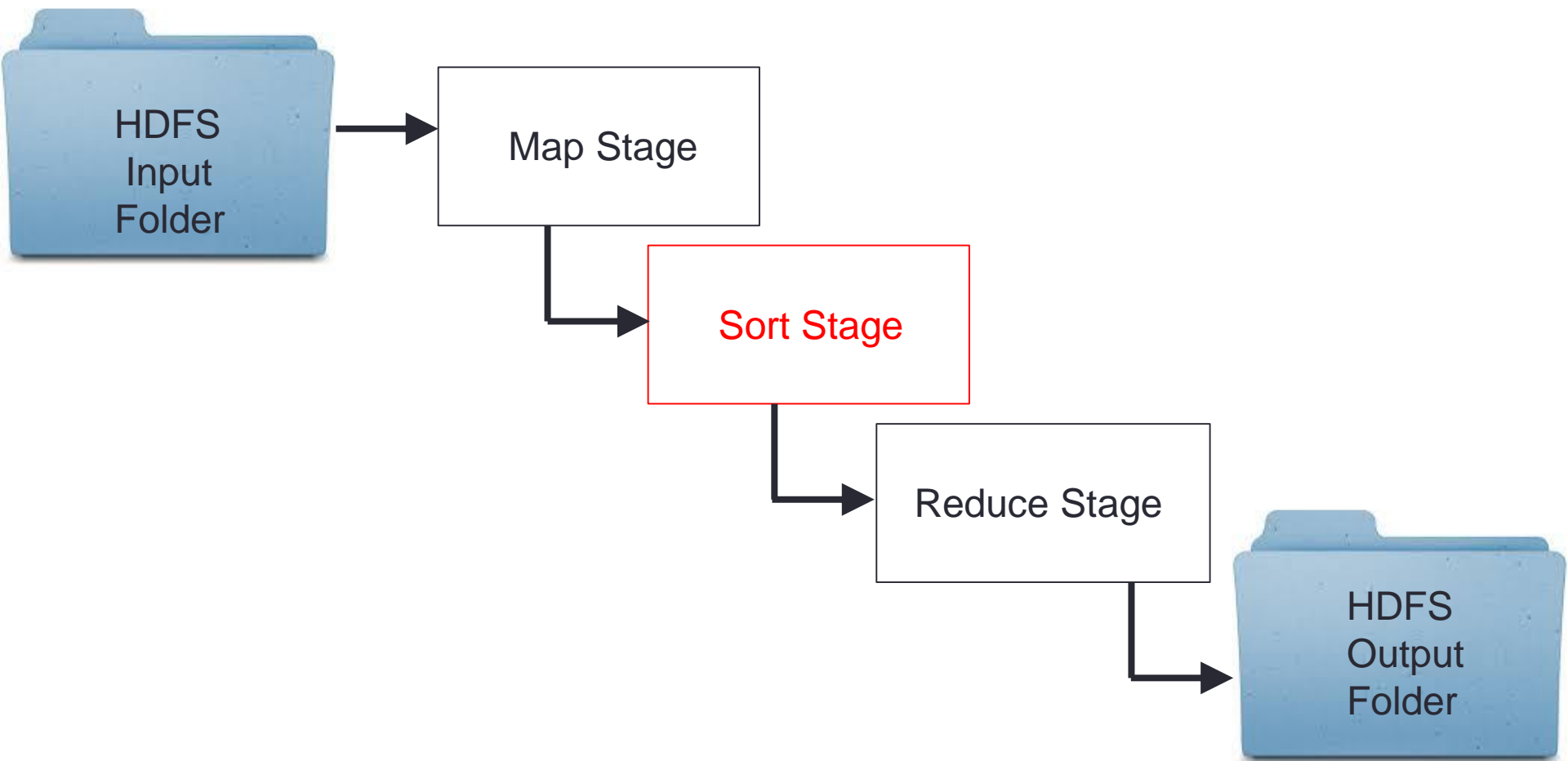


# Data Processing: Map – Sort – Reduce



# Data Processing: Map – Sort – Reduce

## MapReduce Framework Pipeline Process





# Data Processing: Map – Sort – Reduce

## Sort Stage

- ❑ At this point we must be a bit confused.
- ❑ We started this story with 20GBs of info.
  - So far all this info has been processed via multiple mappers.
  - Each mapper has processed a block of 64MBs by:
    - Reading its content from streaming standard input.
    - Compute something with this content.
    - Output the result of this computation via a set of (key, value) lines of text and send them by streaming to the standard output.



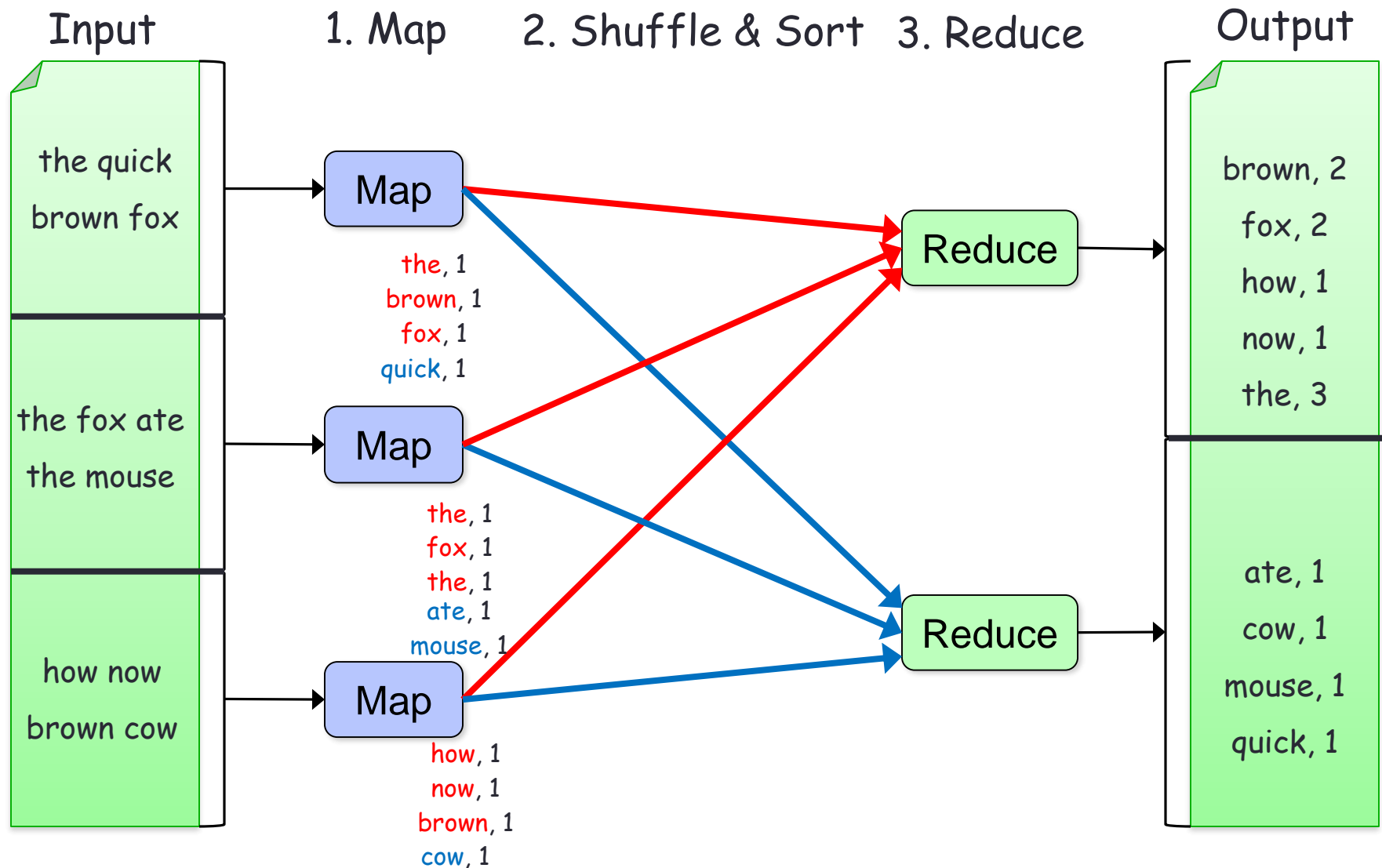
# Data Processing: Map – Sort – Reduce

## Sort Stage

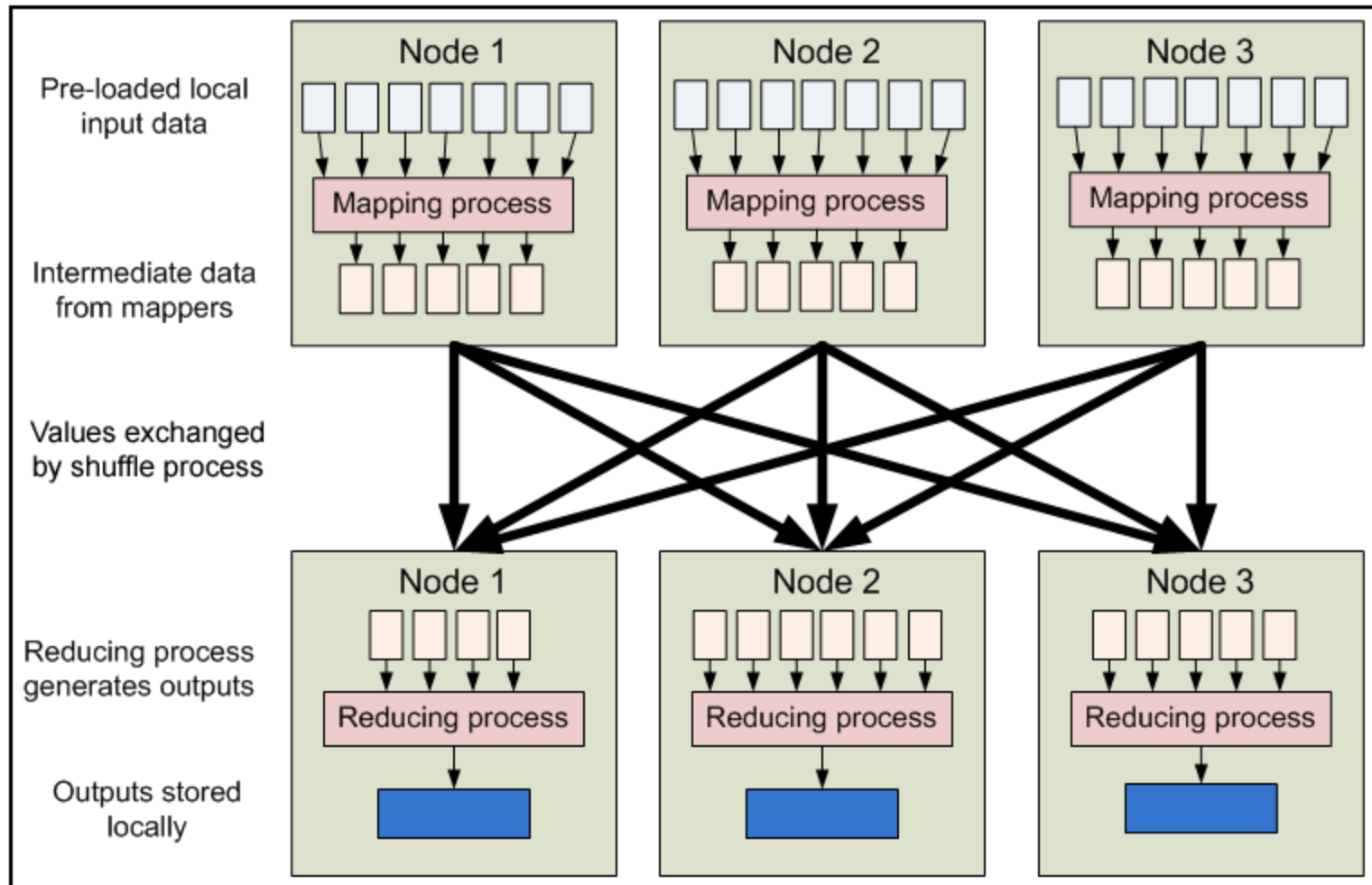
□ So now is when the Sort stage comes to the rescue:

1. It collects all the (key, value) pairs outputted by the map phase.
2. It sorts them.
3. It assigns the subset of (key, value) pairs that are sent to each reducer.
  - If  $k = 1$  reduce process, then all pair (key, value) go to it in their new sorted order.
  - If  $k > 1$  reduce processes, then **sort** distributes in a fairly balanced way the keys to the reducers, ensuring all pairs (key, value) with same key go to the same reducer.

# Data Processing: Map – Sort – Reduce



# MapReduce Framework



# Data Processing: Map – Sort – Reduce

## Sort Stage

❑ Again, the entire process of...

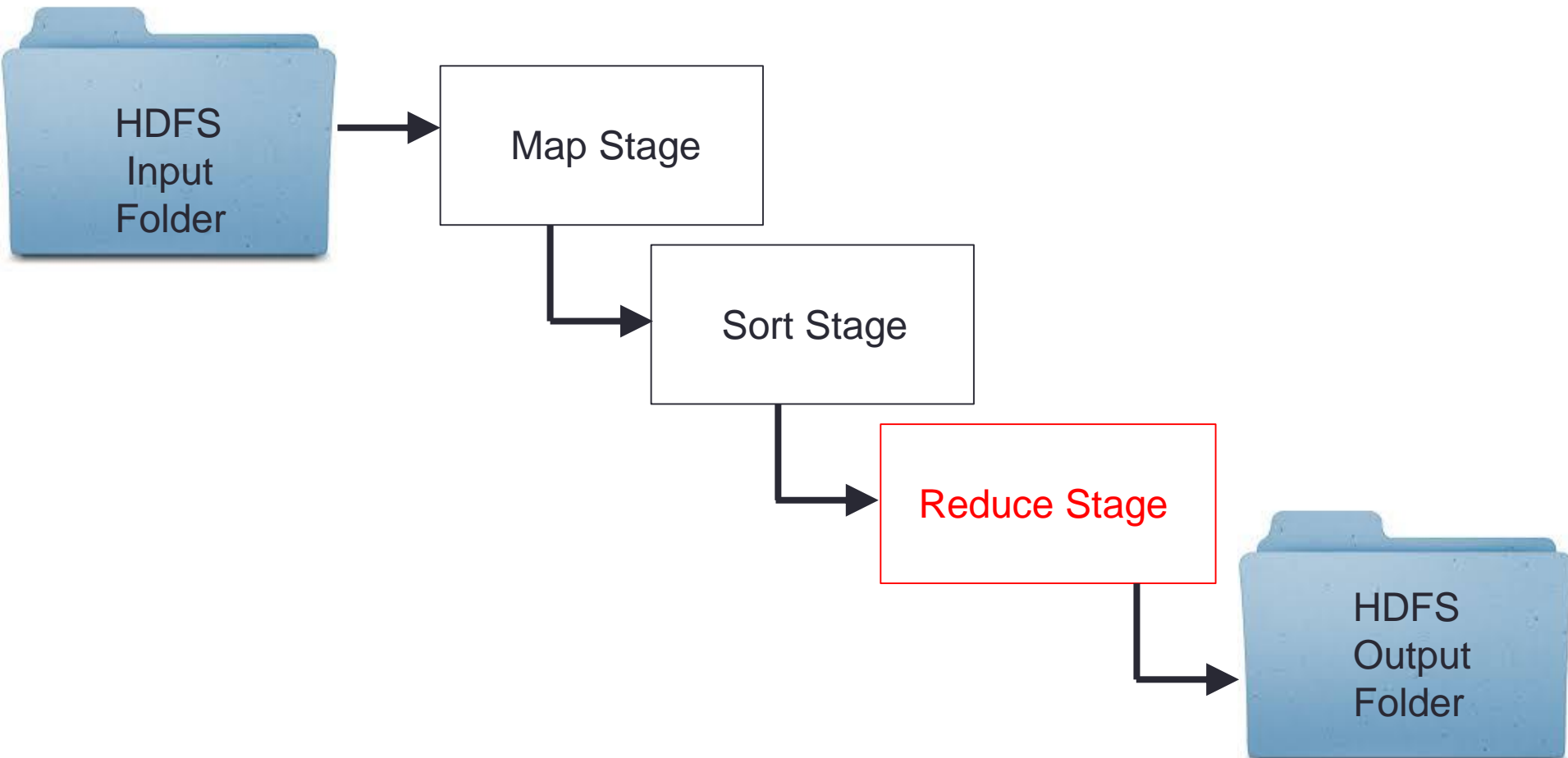
- Collecting the (key, value) pairs outputted by the mappers (and the inherent network transference to achieve it).
- Sorting the entire set of (key, value) pairs by the key.
- Deciding how many reducers to create.
- Deciding which subset of sorted pairs (key, value) are assigned to each reducer (and the inherent network transference to achieve it).

...is all set up for you!

- Now, your single task is to program what a reducer does.
- All reducers have to do the same, so that you just program it once.

# Data Processing: Map – Sort – Reduce

## MapReduce Framework Pipeline Process

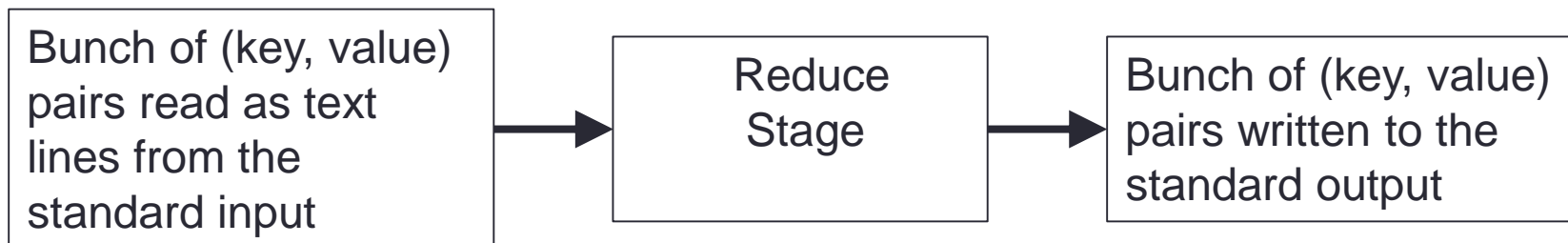


# Data Processing: Map – Sort – Reduce

## Reduce Stage

### Programming the reducer

- ❑ The Reduce stage combines the intermediate results provided by the mappers to compute the final result of the process.
- ❑ Again, how to program the reducer is not very open.
  - You know you are receiving pairs (key, value) in the form of a text line via streaming from the standard input. One pair after another.
  - All you can do is to process the content of these lines.

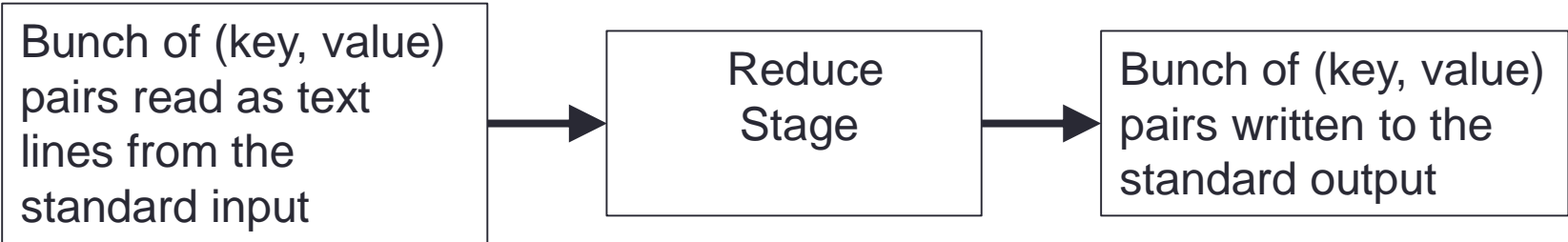


# Data Processing: Map – Sort – Reduce

## Reduce Stage

### Programming the mapper

- ❑ If that was not enough, you're also limited with the output.
  - Do whatever you want for processing the (key, value) lines, **but...**
  - Your output must be a set of key-value pairs.
    - That you must write as a bunch of lines to the standard output.
    - This is kind of counterintuitive, but is how it actually works.



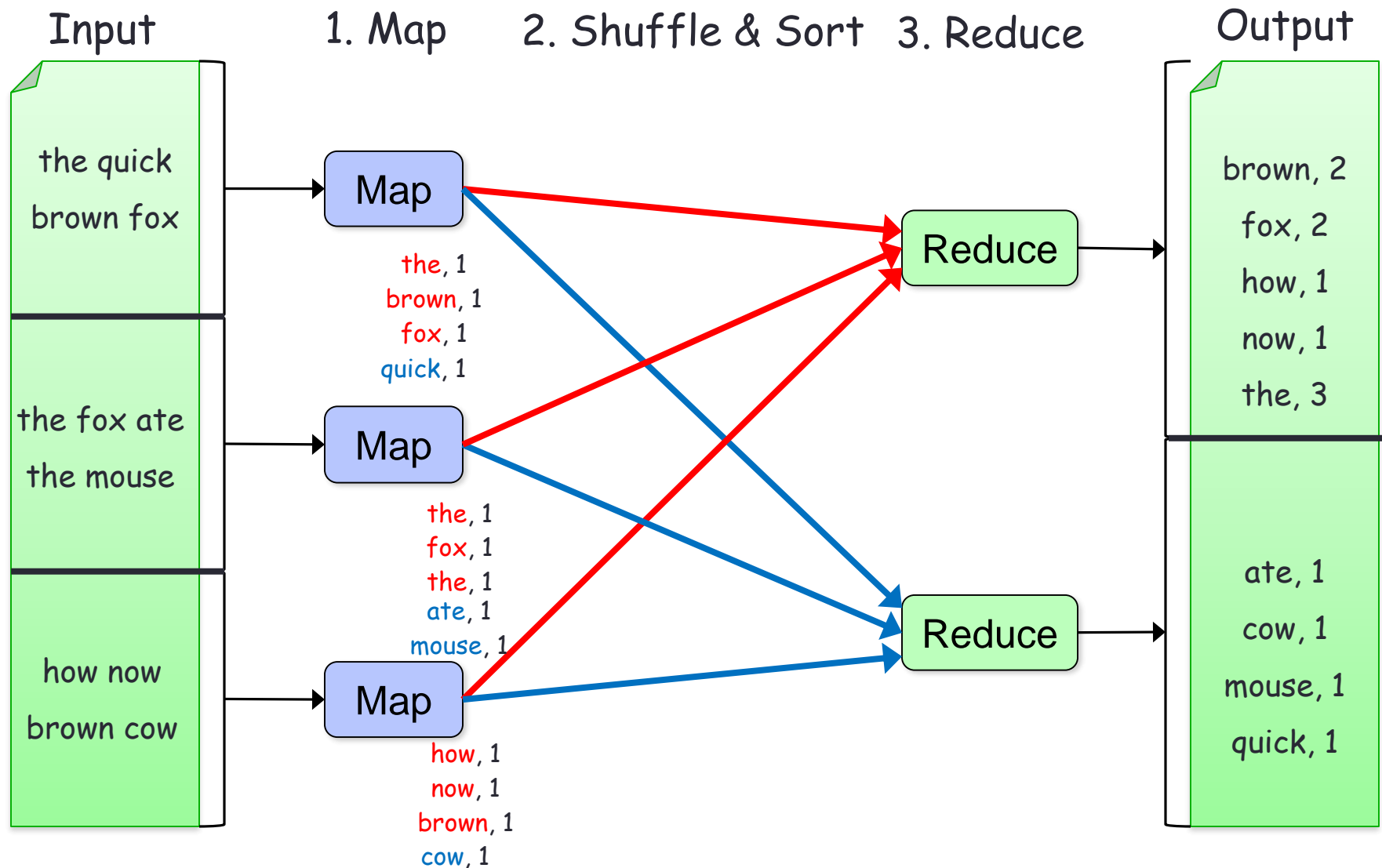


# Data Processing: Map – Sort – Reduce

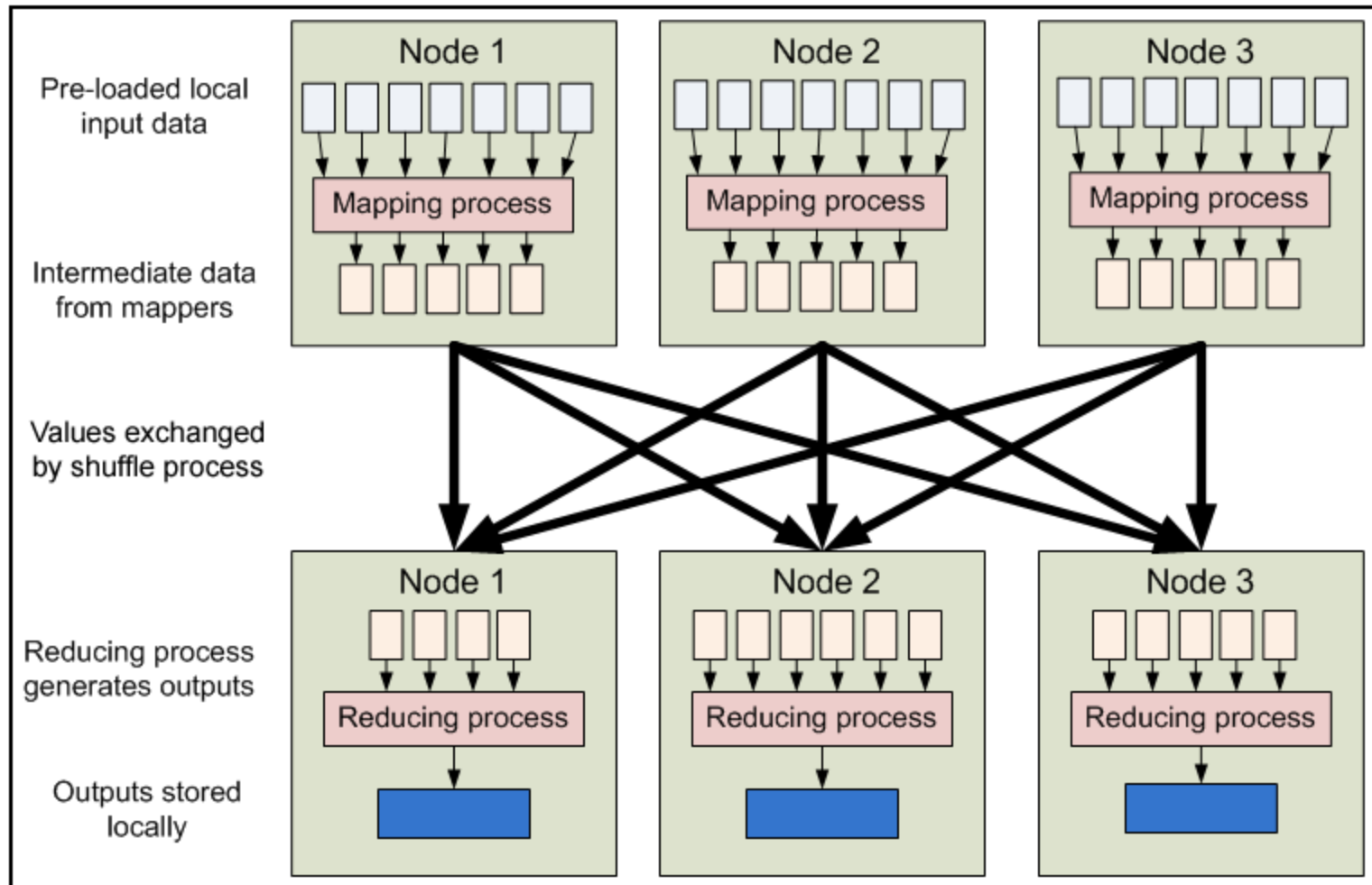
## Reduce Stage

- ❑ So, at this point we must be a bit confused once again.
- ❑ If the output of the single reducer is to output some (key, values) pairs in the form of text lines to the standard output...
- ❑ How is it that the output of the entire MapReduce process is a bunch of files?
  - Once again, this is done automatically by the framework, which creates an output file per reducer, and writes all the text lines outputted to it.
  - If  $k > 1$  reducers, the  $k$  output files being generated can be automatically merged to a single file.

# Data Processing: Map – Sort – Reduce

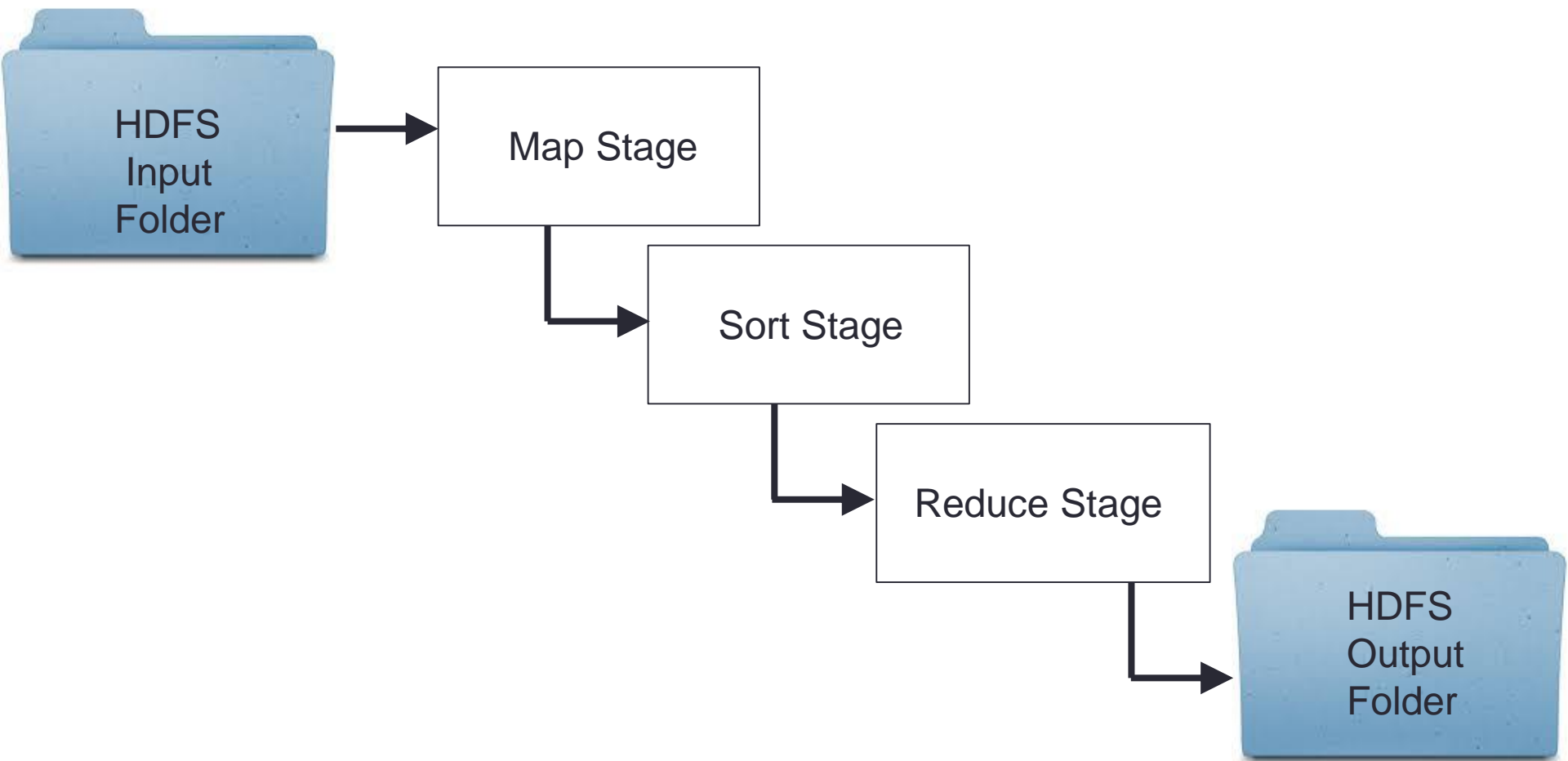


# MapReduce Framework



# Data Processing: Map – Sort – Reduce

## MapReduce Framework Pipeline Process

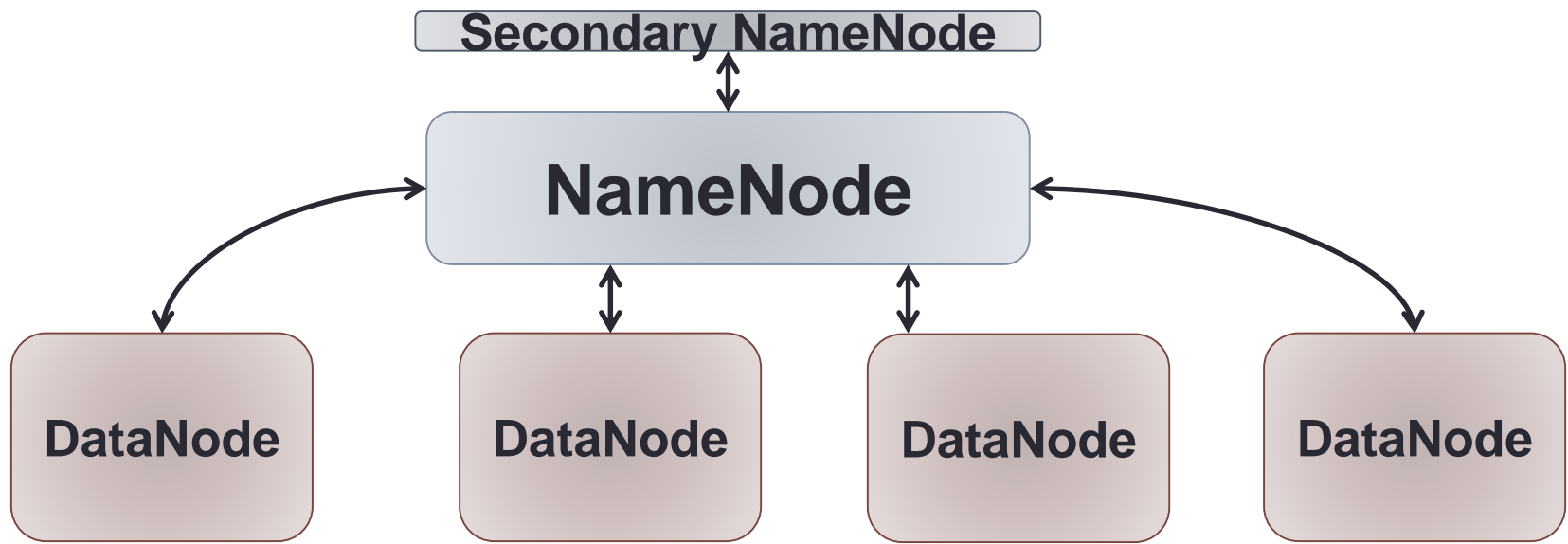


# Data Processing: Map – Sort – Reduce

## Cluster Roles

❑ So far, in the HDFS we had two different roles:

1. NameNode → Being the master, controlling the metadata.
2. DataNode → Being the slave, or the one actually storing the data.



# Data Processing: Map – Sort – Reduce

## Cluster Roles

- ❑ Now for MapReduce we are going to have two roles as well:
  1. JobTracker → This is the software daemon (background service) controlling the execution of a MapReduce job.
- The JobTracker has to reside on a master node (NameNode):
  - Clients submit the MapReduce job to the JobTracker.
  - The JobTracker assigns Map and Reduce tasks to concrete DataNodes of the cluster.

# Data Processing: Map – Sort – Reduce

## Cluster Roles

- ❑ Now for MapReduce we are going to have two roles as well:
  2. TaskTracker → A DataNode being assigned a Map or Reduce job is said to be a TaskTracker.
    - These nodes are responsible for reporting progress back to the JobTracker.

# Data Processing: Map – Sort – Reduce

## Cluster Roles

- ❑ In general, a **job** is a full program:
  - That is, the complete execution of all mappers and reducers.
  - All together, they cover the data of the entire dataset.
- ❑ A **task** is the execution of a single Map/Reduce:
  - It works over a slice of the dataset.



# Data Processing: Map – Sort – Reduce

## Cluster Roles

- ❑ A **task attempt** is a particular node executing a task.
  - This means that several nodes can be assigned the same task (this is called **speculative execution**).
  - In this case, as soon as one of the nodes finishes the execution of the task, the result is collected and the remaining nodes attempting the same task are stopped.
  - This approach is used when one node is particularly slow in performing the task being assigned to it.
  - By allowing multiple task attempts, if a node fails, nothing happens: The JobTracker just assigns the task to another node.

# Outline

1. Main Features of Hadoop.
2. Data Storage: Hadoop Distributed File System.
3. Data Processing: Map – Sort – Reduce.

Thank you for your attention!