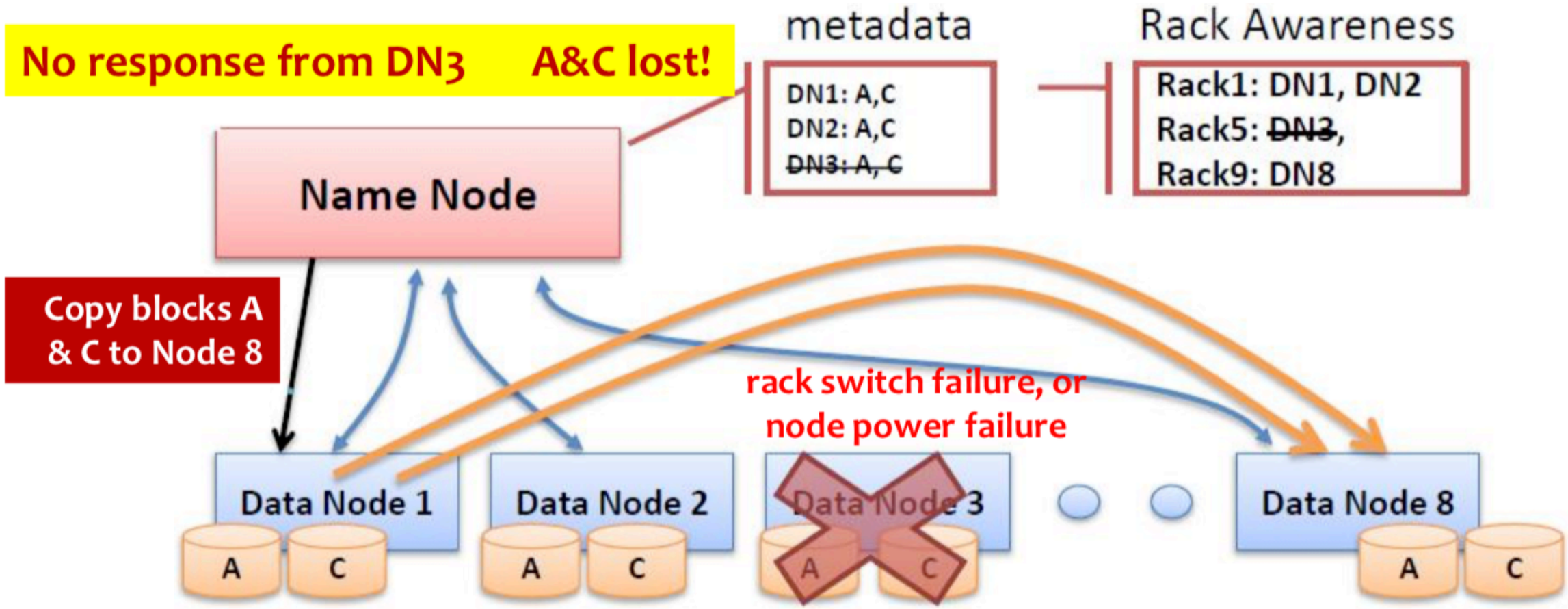
Replica management (1)

- The Name Node ensures that each block always has the intended number of replicas, also makes sure not all replicas of a block are located on one single rack.
- Under- or over-replicated detection upon receiving a block report from a Data Node.

Re-replicating missing replicas

- Missing heartbeats signify lost nodes.
- NameNode consults metadata, finds affected data.
- NameNode consults Rack Awareness script.
- NameNode tells a DataNode to re-replicate.



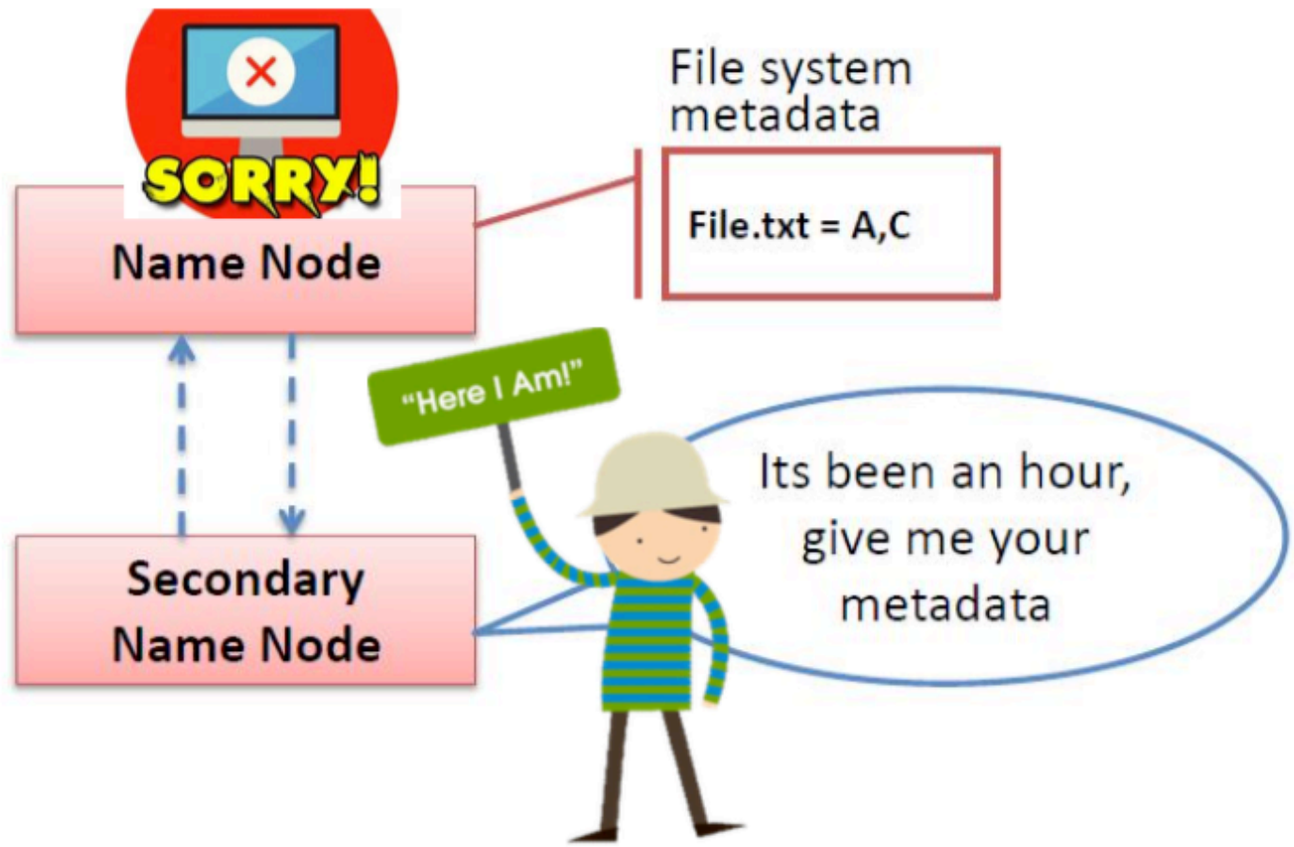
Replica management (2)

- Over replicated: chooses a replica to remove.
  - (1) Prefer not to reduce the number of racks that host replicas
  - (2) Prefer to remove a replica from the Data Node with the least amount of available disk space.
- balance storage utilization across Data Nodes without reducing the block's availability.

Hadoop 2.x: secondary namenode

- If Name Node is down, HDFS is down.
  Solution: add a 2nd Name Node
  Connects to 1st Name Node every hour.
  Housekeeping, backup of Name Node metadata.
  Saved metadata can rebuild a failed Name Node.

Hadoop 2.x used a single active Name Node and a single Standby Name Node.



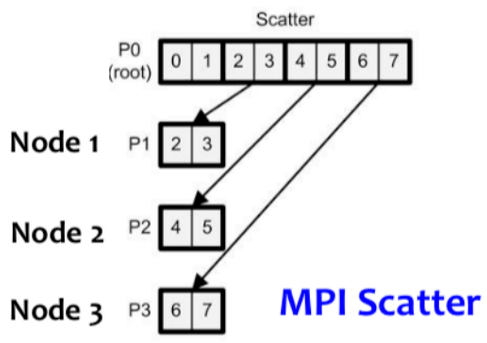
Real-world use case of HDFS

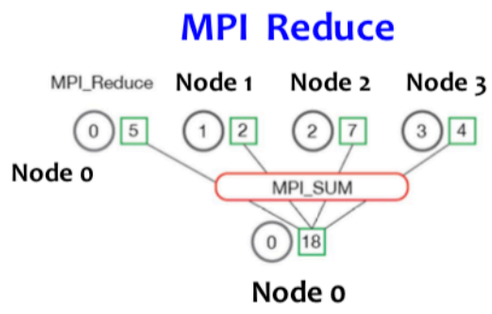
- NetApp provides storage solution to businesses/companies
- Large financial firm: 60 PB of raw data
- Requires 1200 HDFS storage nodes organized as a data lake
- Includes 3x replicas + over-provisioned space for failures etc.
- 288 TB disk capacity per HDFS node (storage dense configuration)

MapReduce
(as a programming model)

MapReduce

- Software framework introduced by Google to support distributed computing on large data sets on clusters
  - Google's paper "MapReduce: Simplified Data Processing on Large Clusters" in OSDI2004.
- Conceptually similar to the scatter and reduce operations in the very well known Message Passing Interface (MPI) standard since 1995
- Hadoop is its open-source implementation.





Motivation

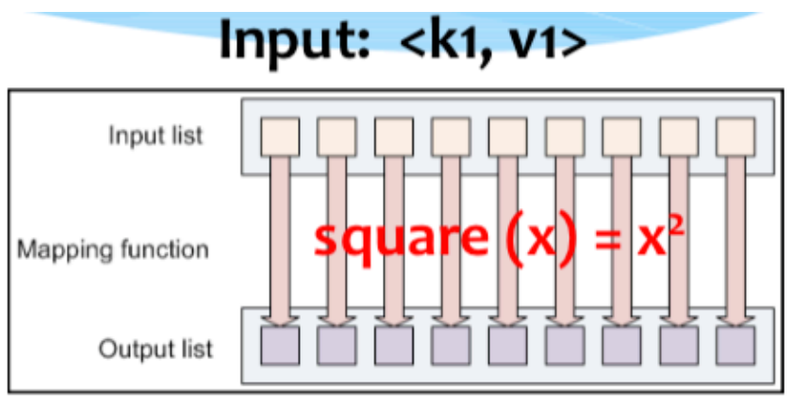
- The problem (in Google)
  - Large input data (the whole Web, billions of web pages), but simple operations (scan keywords, construct index, ..).
  - Many computations over VERY large datasets. Performed on lots of machines
  - Question: how do you use large # of machines efficiently?

Motivation

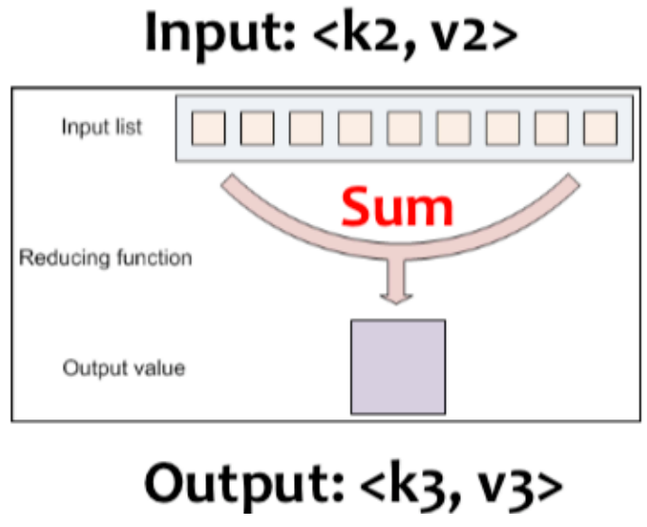
- Solution: reduce computational model down to two steps:
  - Map: take one operation, apply to many data tuples.
  - Reduce: take result, aggregate them.
  - Automatic distribution of work.

Programming concepts

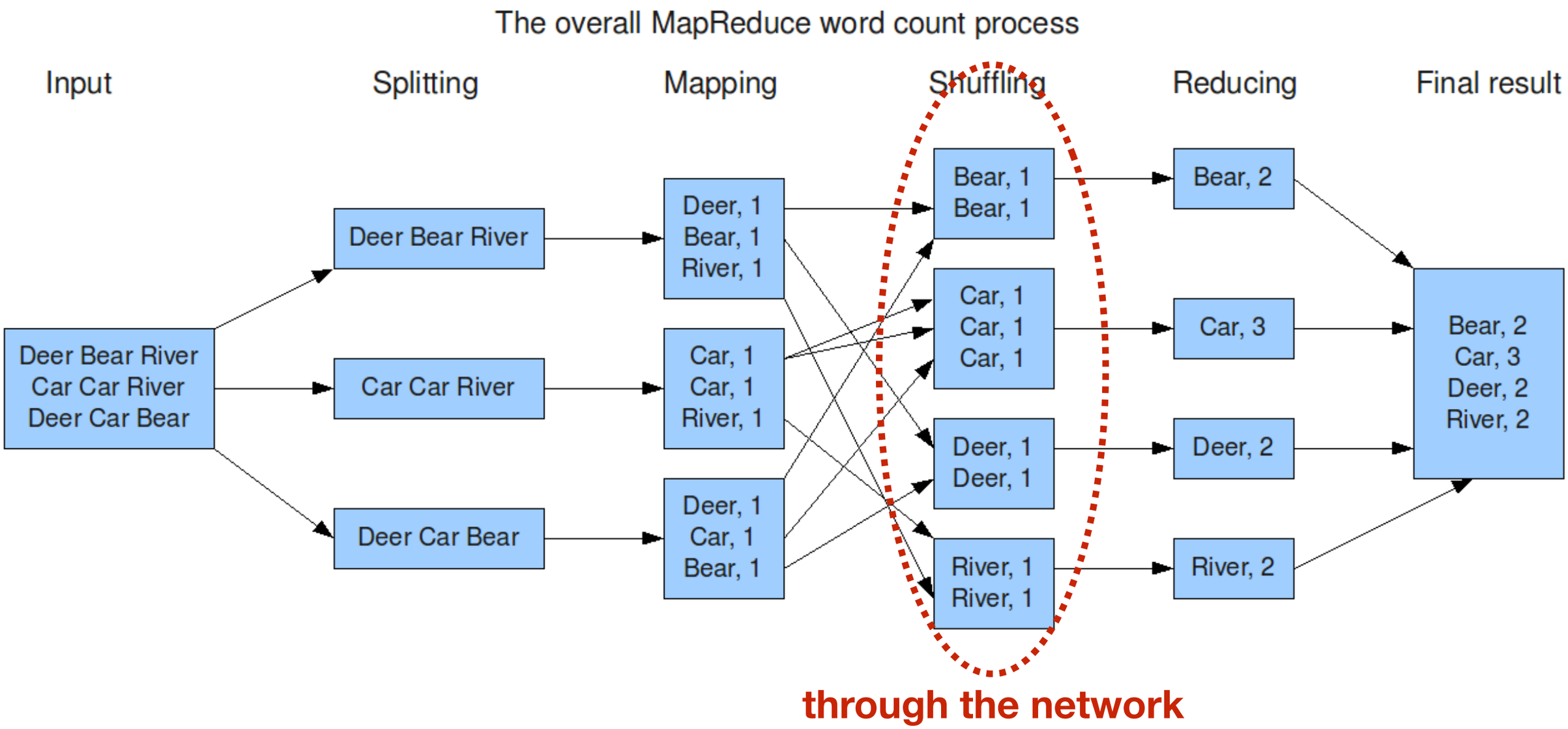
- Map
  - Perform a function on individual values in a data set to create a new list of values
  - Example: square (x) = x * x. Map: square ([1,2,3,4,5]) returns [1,4,9,16,25]



- Reduce
  - Combine values in a data set to create a new value
  - Example: sum = (each element in an array, total +=)
  - Reduce: sum ([1,2,3,4,5]) returns 15 (the sum of the 5 elements)



MapReduce framework



The MapReduce framework breaks down data processing into map, shuffle, and reduce phases. Processing is mainly in parallel on multiple compute nodes.

MapReduce design principle (1)

- Key concept: Divide and Conquer
  - Map() run in parallel, creating different intermediate values from different input data sets.
  - Reduce() run in parallel, each working on a different output key
  - Highest parallelism: All values are processed independently.

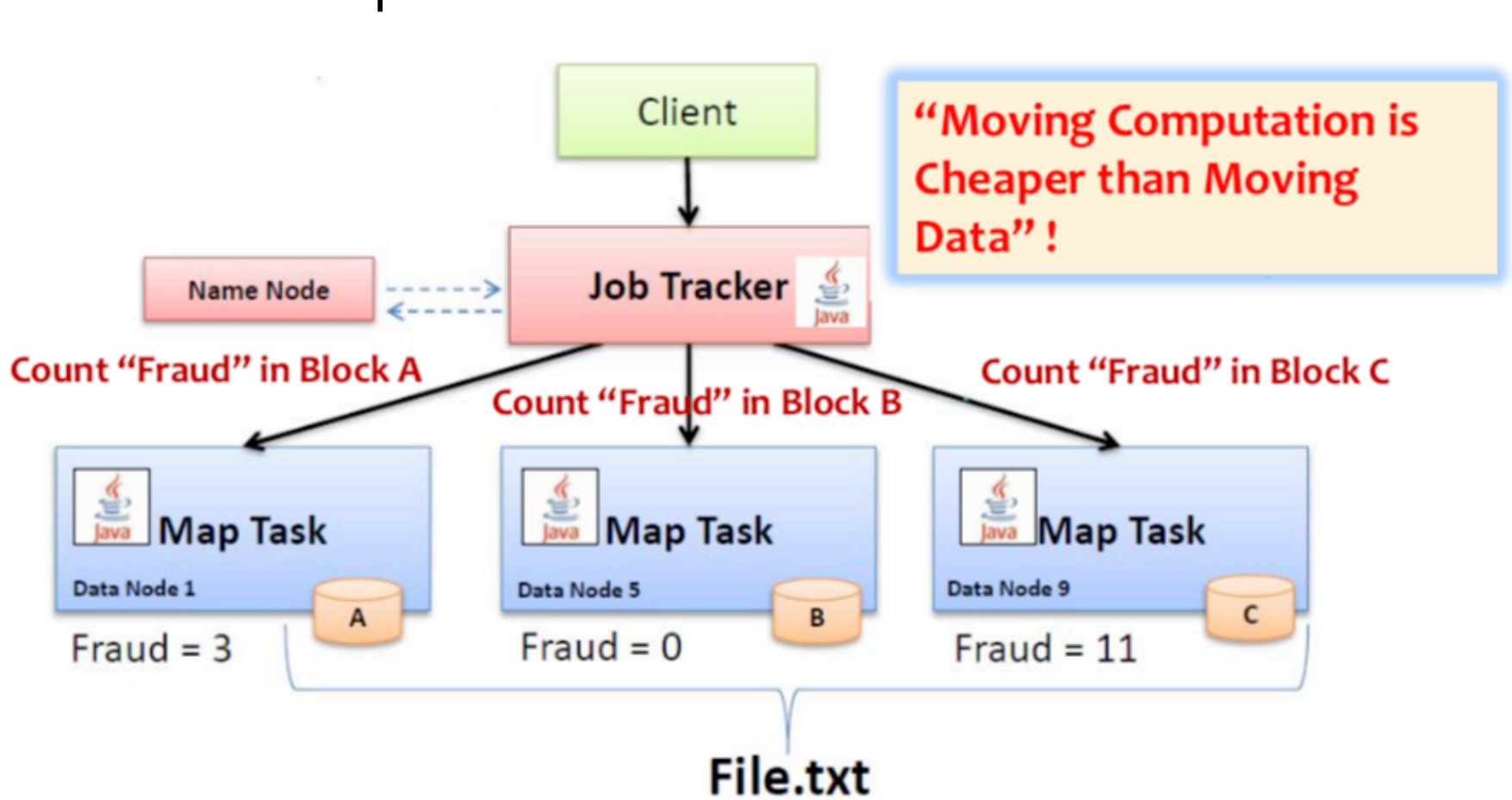
MapReduce design principle (2)

- Users specify the computation in terms of a map and a reduce function (You just write “Map” & “Reduce” functions). MapReduce runtime system does the rest for you, including
  - (1) Fault-tolerance, (2) I/O scheduling, (3) Job status monitoring
  - Automatically parallelizes the computation across machines.
  - Hadoop handles machine failures, efficient communications, and performance issues for you.

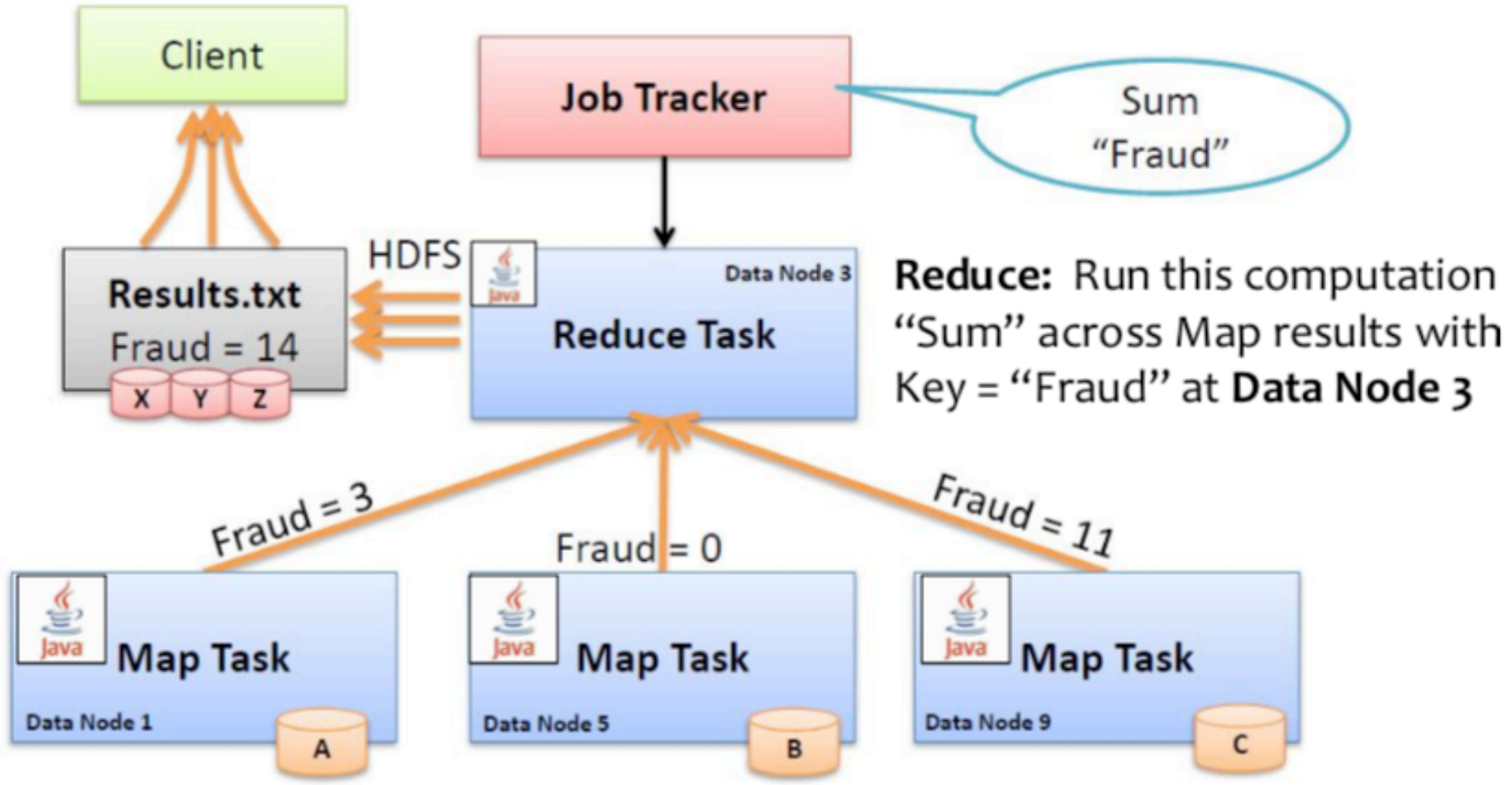
Map

- Hadoop 1.0: JobTracker first consults the NameNode to learn which Data Nodes have blocks of File.txt.

- JobTracker delivers Map code to Nodes with the datablock.



Map + Reduce

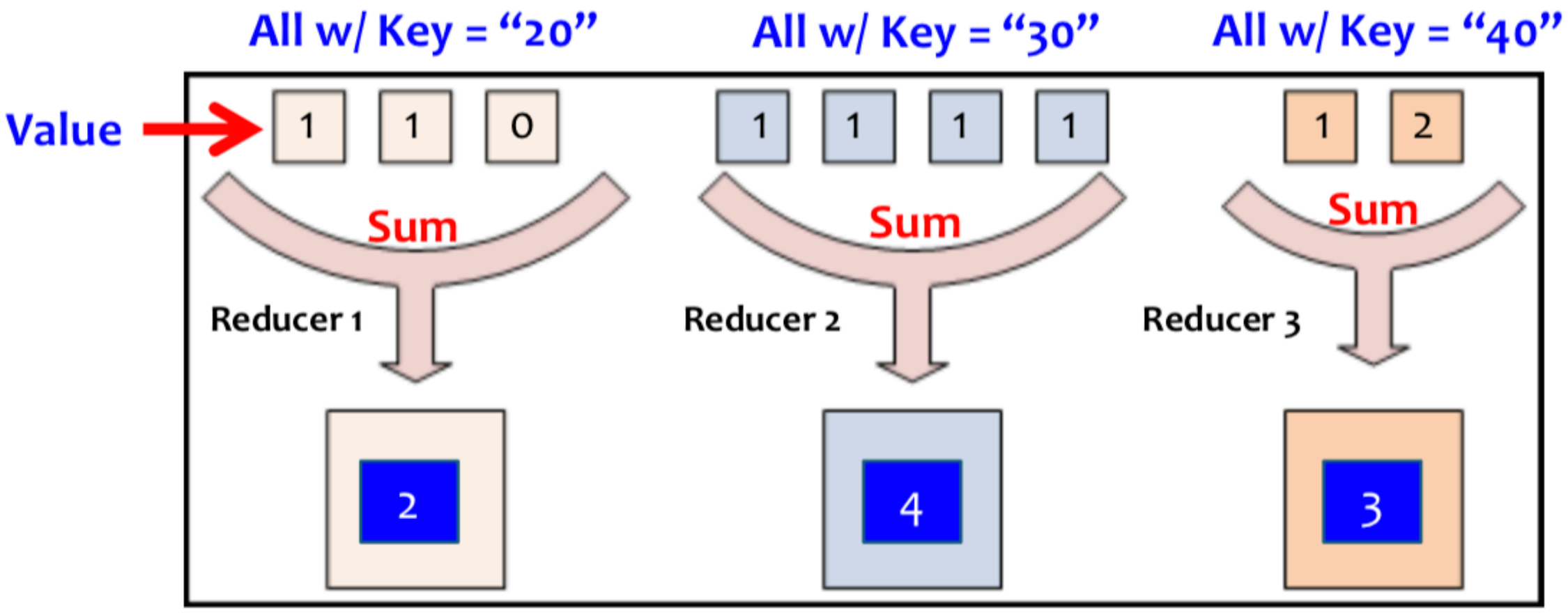


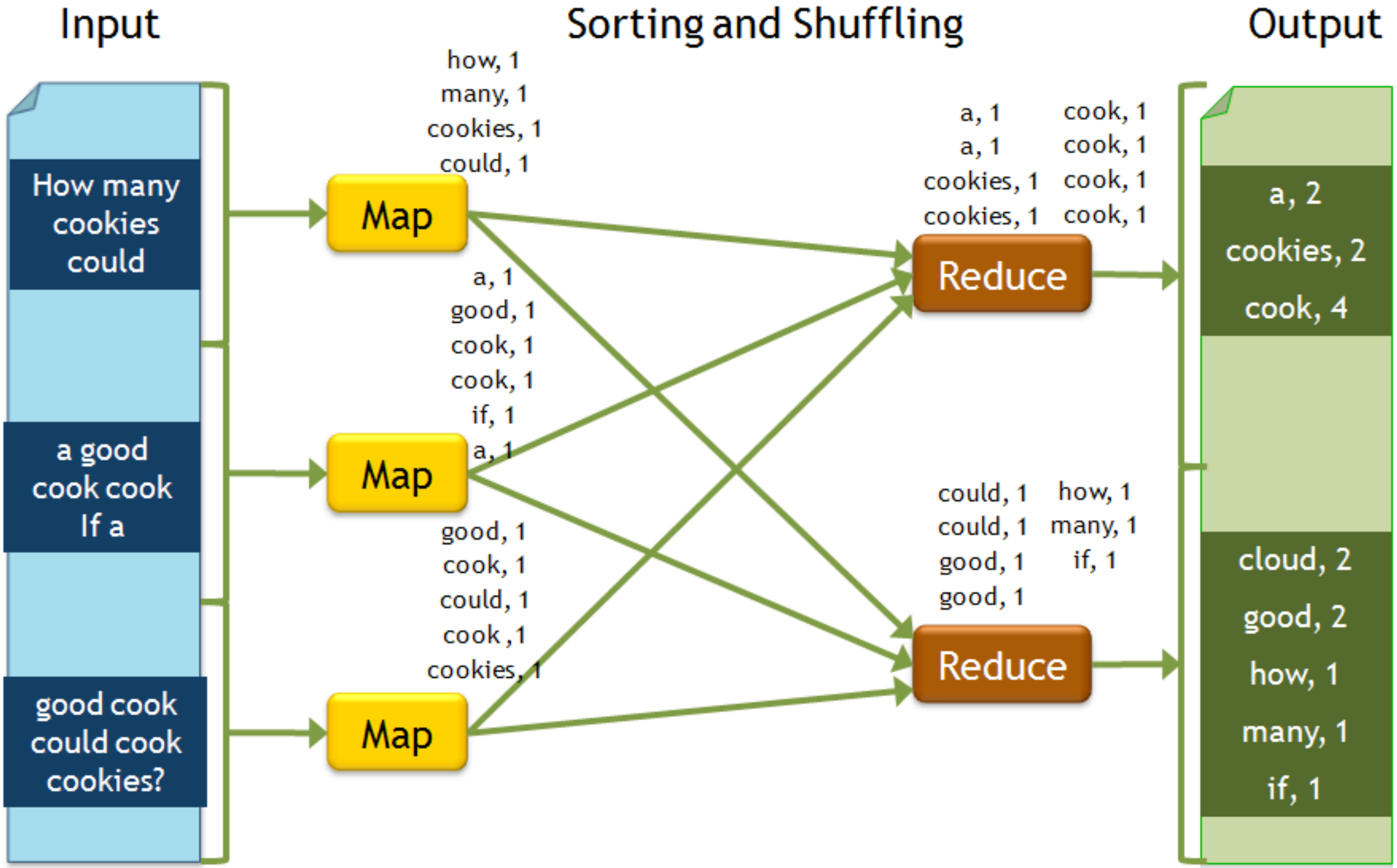
- Map Tasks deliver output data over the network.

- Reduce Task: data output is written to HDFS (disk I/O)

“Keys Divide the Reduce Space”

- All of the values with the **same key** (e.g., people of the same age) are presented to a single reducer together. The reducing function turns a large list of values into **one (or a few)** output values.




Input
Sorting and Shuffling
Output
How many cookies could
a good cook cook If a
good cook could cook cookies?
Map
Map
Map
how, 1
many, 1
cookies, 1
could, 1
a, 1
good, 1
cook, 1
cook, 1
if, 1
a, 1
good, 1
cook, 1
could, 1
cook ,1
cookies, 1
a, 1 cook, 1
a, 1 cook, 1
cookies, 1 cook, 1
cookies, 1 cook, 1
could, 1 how, 1
could, 1 many, 1
good, 1 if, 1
good, 1
Reduce
Reduce
a, 2
cookies, 2
cook, 4
cloud, 2
good, 2
how, 1
many, 1
if, 1

Scheduling & load balancing

- One master, many workers
  Input data split (typically 64/128MB in size) into M map tasks

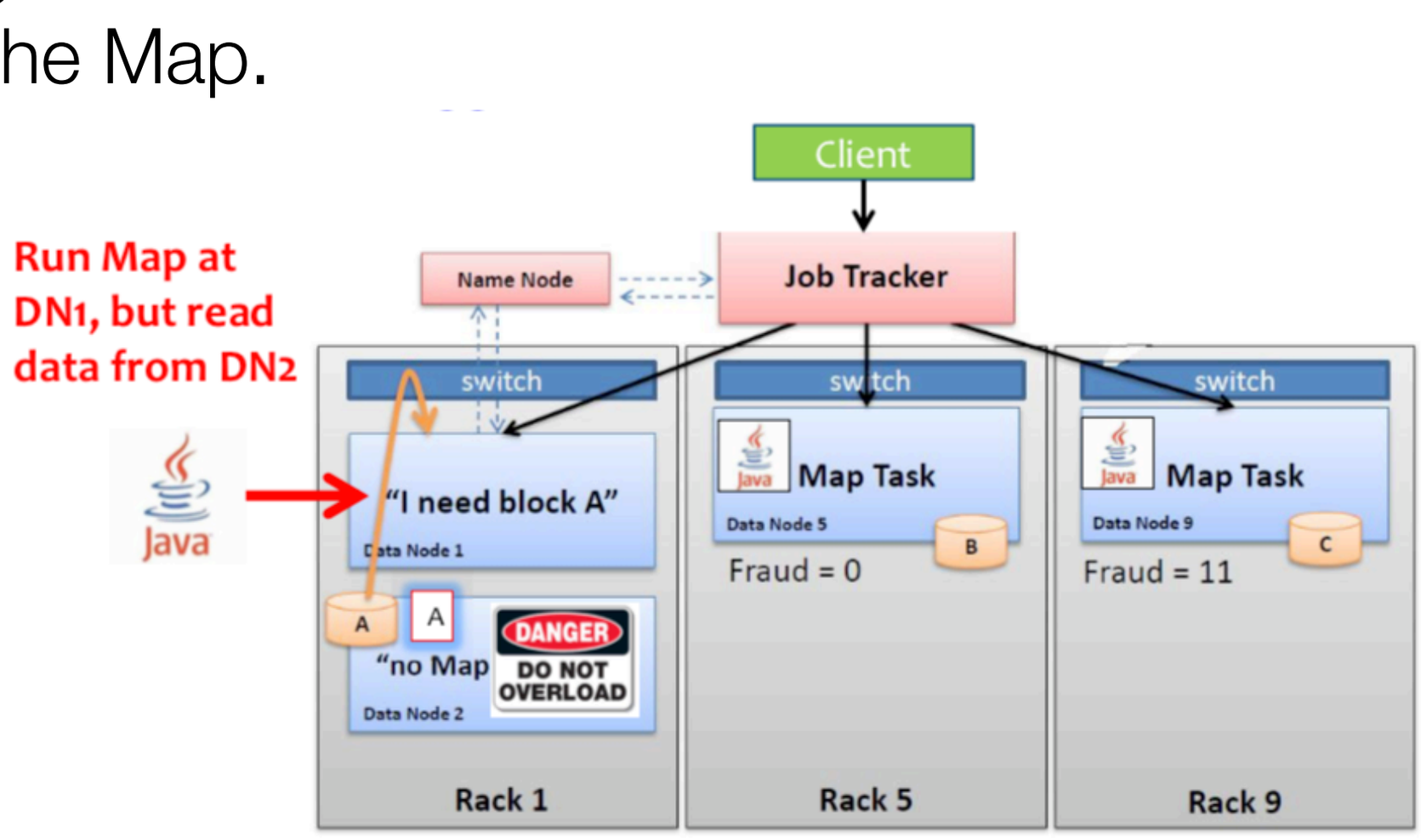
- Reduce phase partitioned into R reduce tasks.
  Tasks are assigned to workers dynamically.

- Master node assigns each map task to a free worker.
  Considers locality of data to worker when assigning task
  Worker reads task input (often from local disk!)
  Worker produces R local files containing intermediate k/v pairs

- Master node assigns each reduce task to a free worker.
  Worker reads intermediate k/v pairs from map workers.
  Worker sorts & applies user's Reduce op to produce the output.

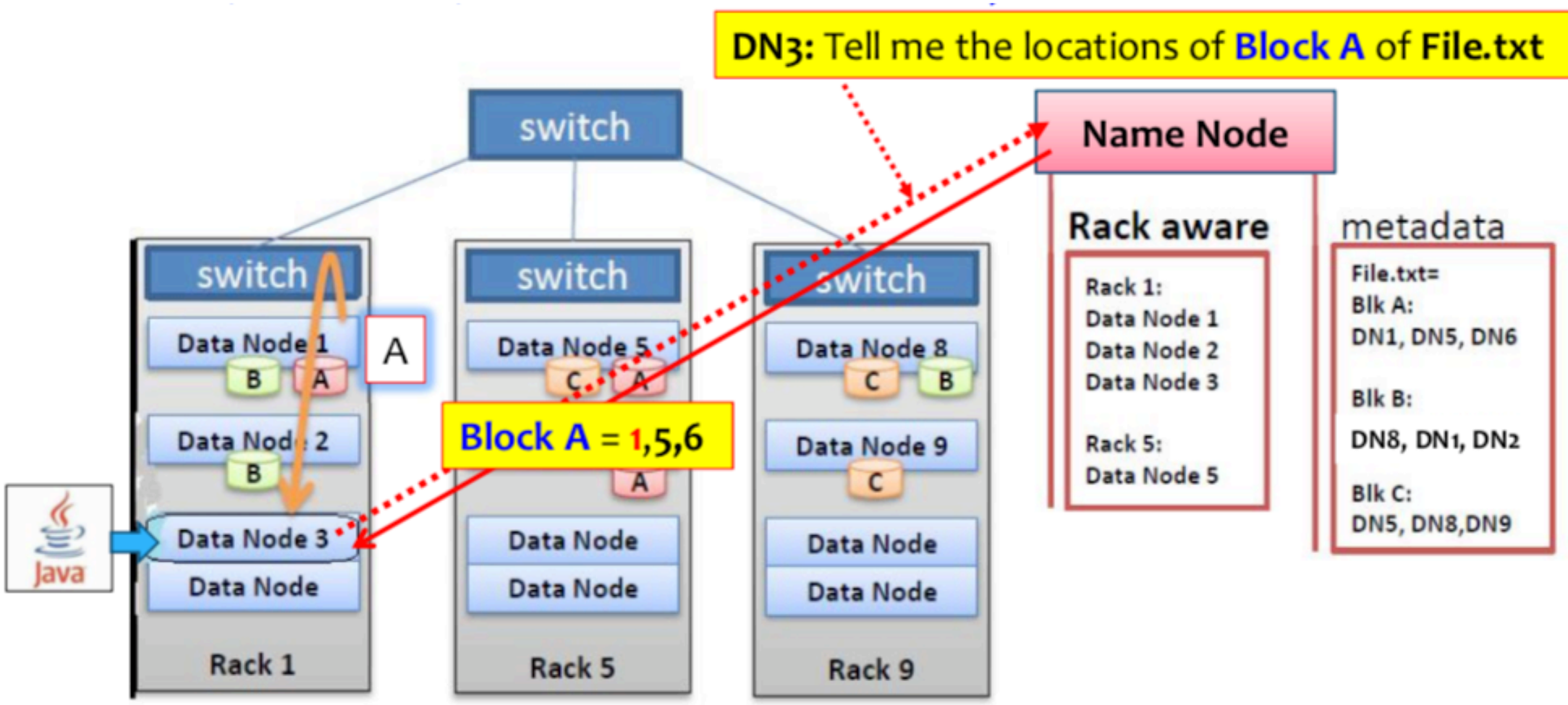
What if data isn’t local?

- Why it happens?
  All the nodes with local data already have too many other tasks running and cannot accept anymore. NameNode suggests other nodes without data in the same rack to run the Map.



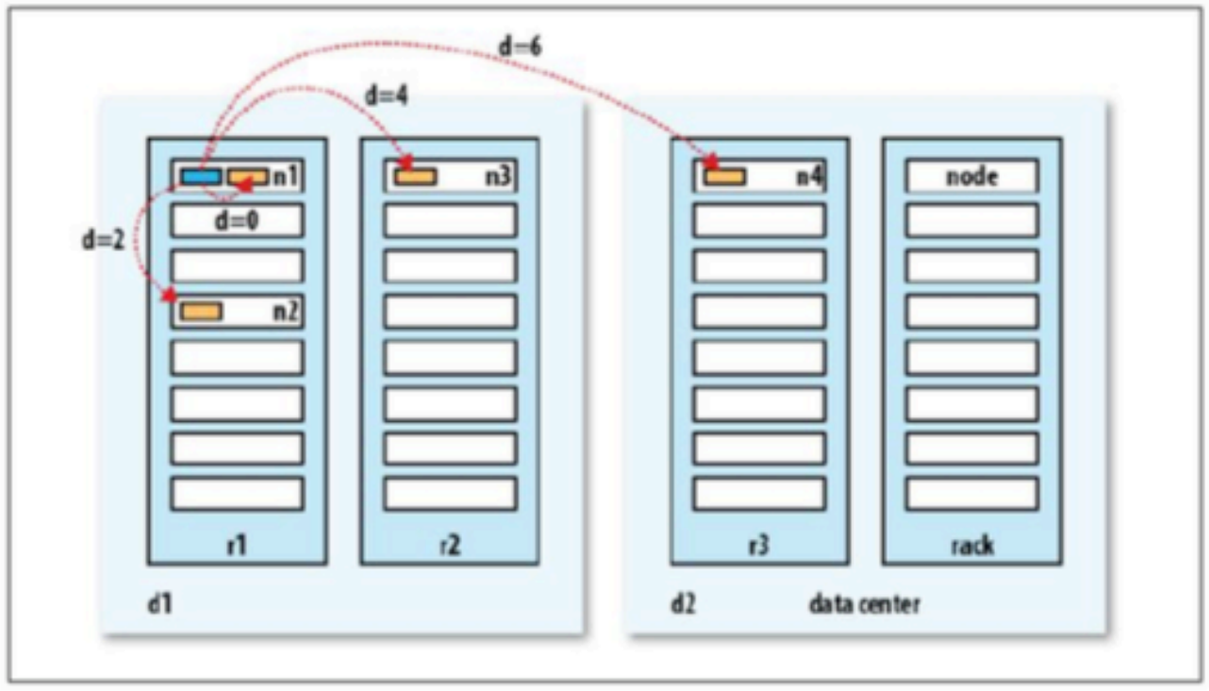
But where to read Block “A”?
Just ask NameNode

- HDFS tries to satisfy a read request from a replica that is closest to the reader. (e.g., Name Node suggests: DN1,DN5, DN6; ordered, DN1 is the closest one)



Locality

- Hadoop's scheduling policy:
  —Schedule a map task on a machine that contains a replica of corresponding input data.
  — Or schedule a map task near the replica of that task's input data (e.g, at the same rack)

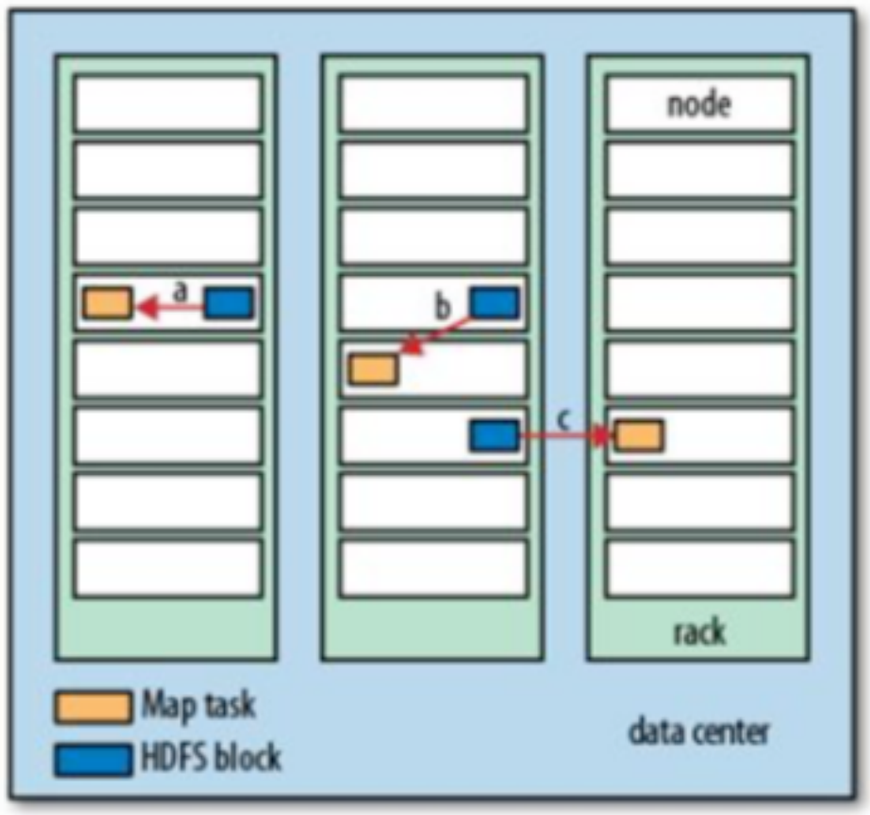


Priority:

(Best) Processes on the same node
(2nd) Different nodes on the same rack
(3rd) Nodes on different racks in the same data center
(Worst) Nodes in different data centers (no worry, we use http)

Rack-local vs. Data-local

- Data local map task: the map task is running local to the machine that contains the actual data.
- Rack local map task: while the data isn't local to the node running the map task, it is still on the same rack.
- If data local map tasks are more the performance will be improved much.



(a) data-local, (b) rack-local, (c) off-rack

Reduce

- (1) Shuffle/Copy: moving map outputs to the reducers
  - Each reducer fetches the relevant partition of the output of all the mappers via HTTP. Note: This causes network traffic !
- (2) Sort
  - The framework merge-sorts Reducer inputs by keys (since different Mappers may have output the same key).
  - Automatically sorted before they are presented to the Reducer.
- (3) Reduce
  - Perform the actual Reduce function that you wrote.
  - The reduce output is normally stored in HDFS for reliability.
  - The output of all Reducers is not re-sorted/merged.