Database Systems
Lecture25 – Chapter 10: Hadoop

SUNG KYUN KWAN UNIVERSITY

Beomseok Nam (남범석)

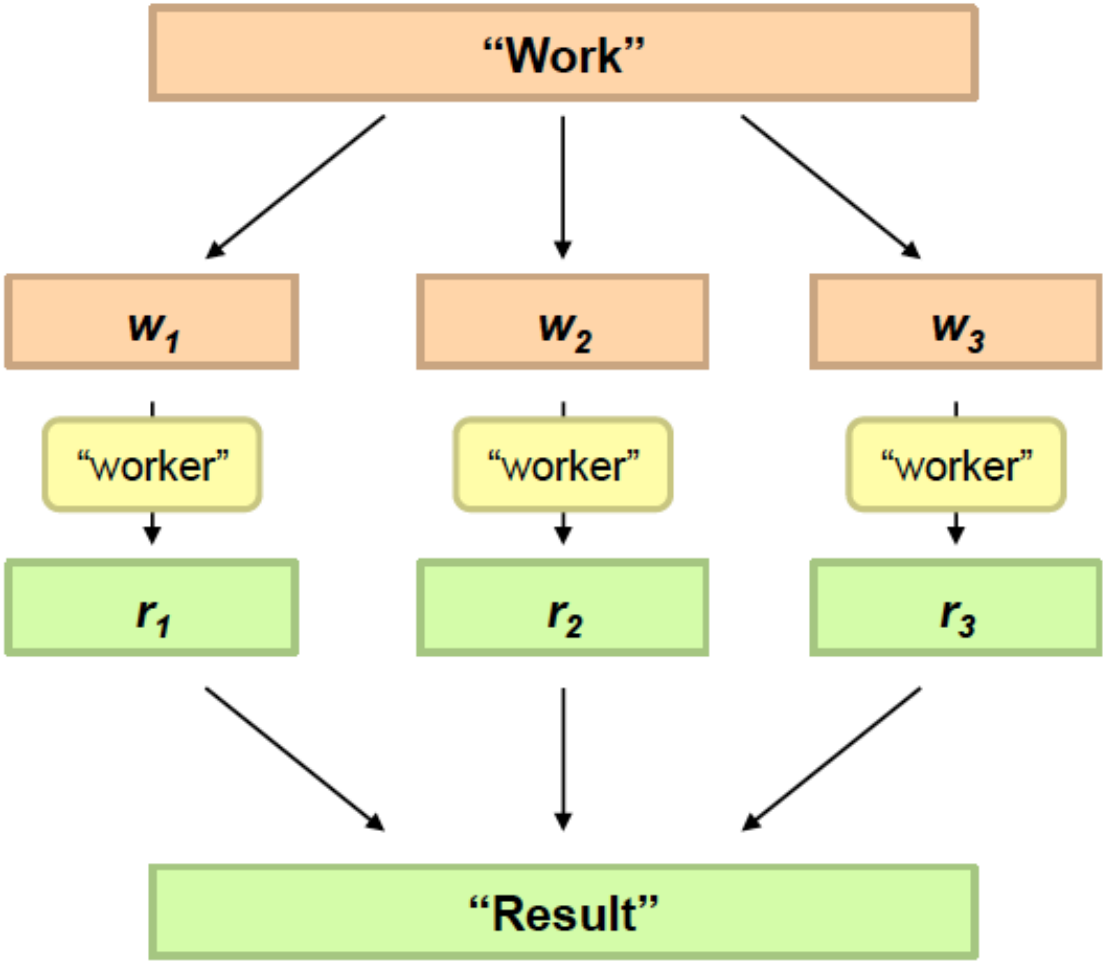
bnam@skku.edu

How to process a large data?

- Q: How do you sort a trillion rows of data of integers (a few terabytes) in a file with only 16 GB of main memory.?

- Hint: Divide and Conquer

Divide and Conquer



Divide Work

Combine Results

Distributed processing is non-trivial

- How to assign tasks to different workers in an efficient way?
- What happens if tasks fail?
- How do workers exchange results?
- How to synchronize distributed tasks allocated to different workers?
- Data Volumes are massive
- Reliability of Storing PBs of data is challenging
- All kinds of failures: Disk/Hardware/Network Failures
- Probability of failures simply increase with the number of machines …
- …

Hadoop offers

- Redundant, Fault-tolerant data storage
- Parallel computation framework
- Job coordination

Hadoop offers

- Redundant, Fault-tolerant data storage
- Parallel computation framework
- Job coordination

Programmers

No longer need to worry about

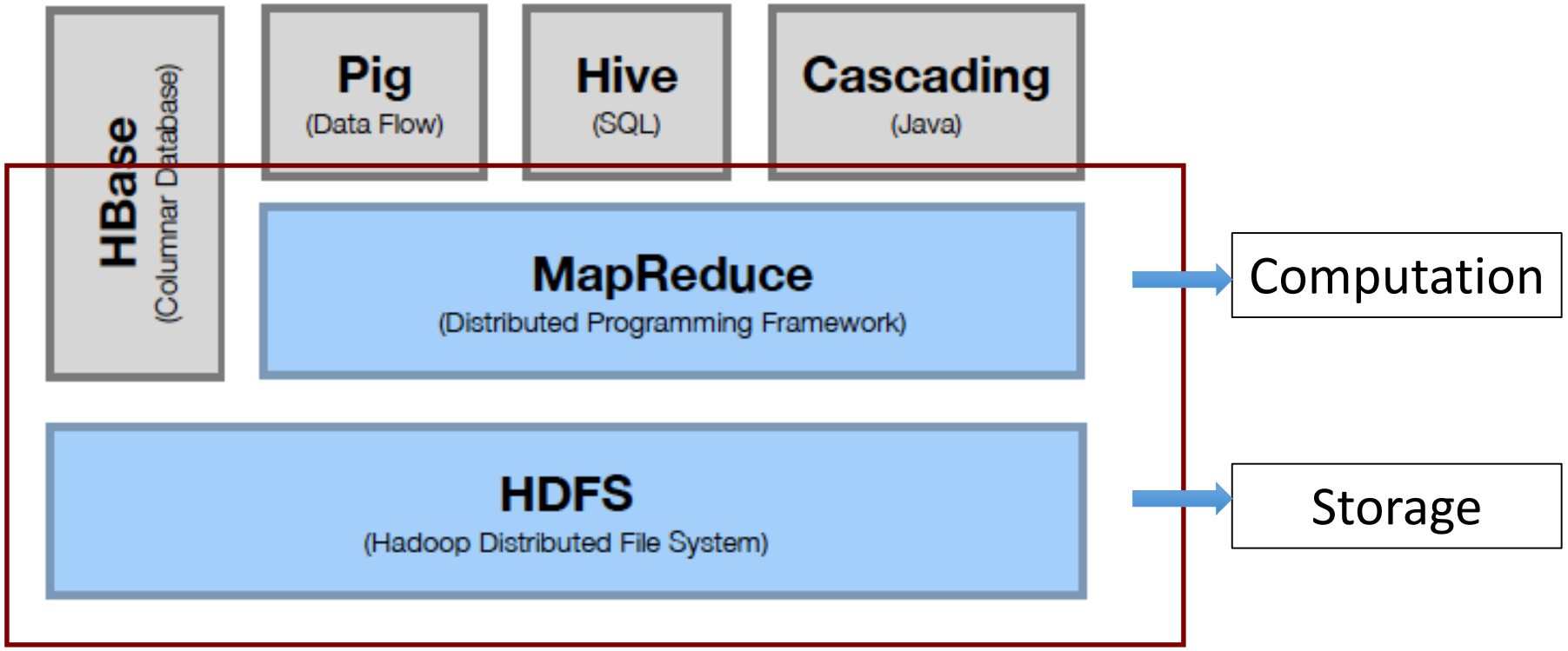
Q: Where file is located?

Q: How to handle failures & data lost?

Q: How to divide computation?

Q: How to program for scaling?

Hadoop Stack

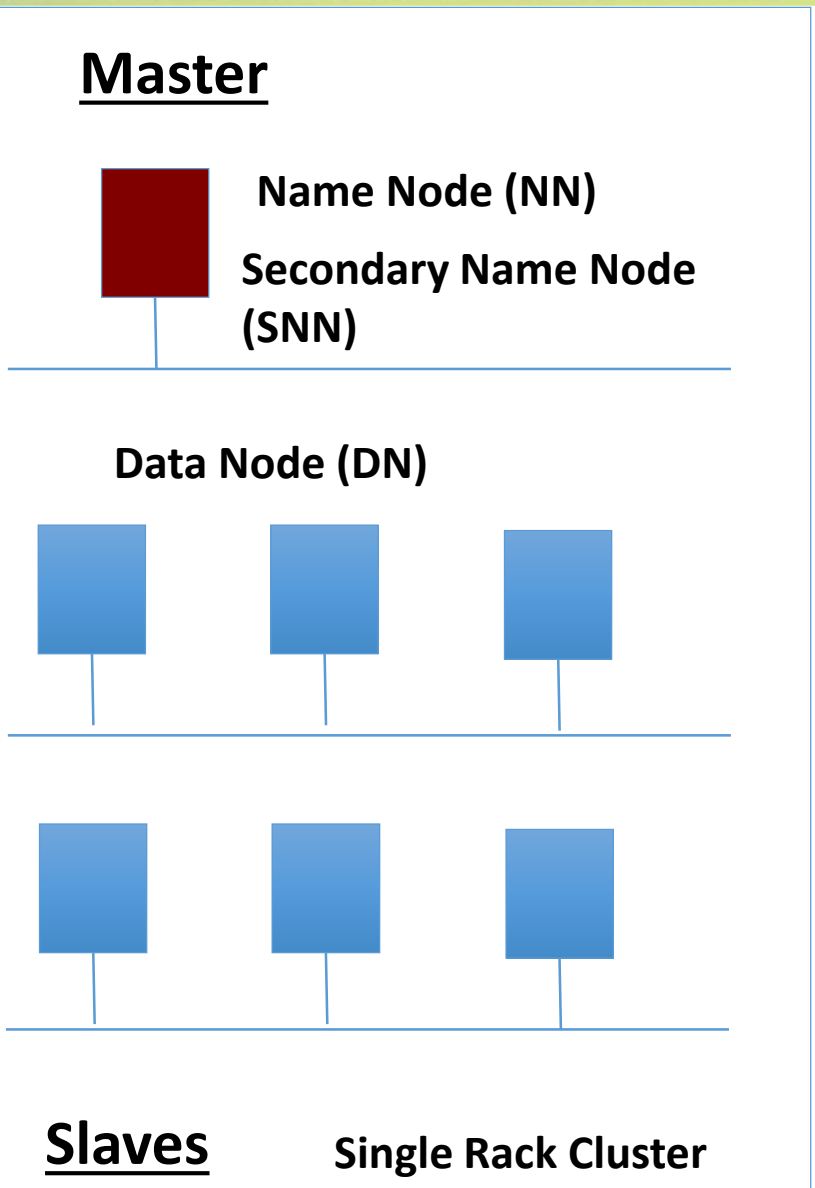


HDFS
Hadoop Distributed File System

Hadoop Distributed File System (HDFS)

- **NameNode**
  - Maps a filename to list of Block IDs
  - Maps each Block ID to DataNodes containing a replica of the block
- **DataNode**: Maps a Block ID to a physical location on disk
- Data Coherency
  - Write-once-read-many access model
  - Client can only append to existing files
- Distributed file systems good for millions of large files
  - But have very high overheads and poor performance with billions of smaller tuples

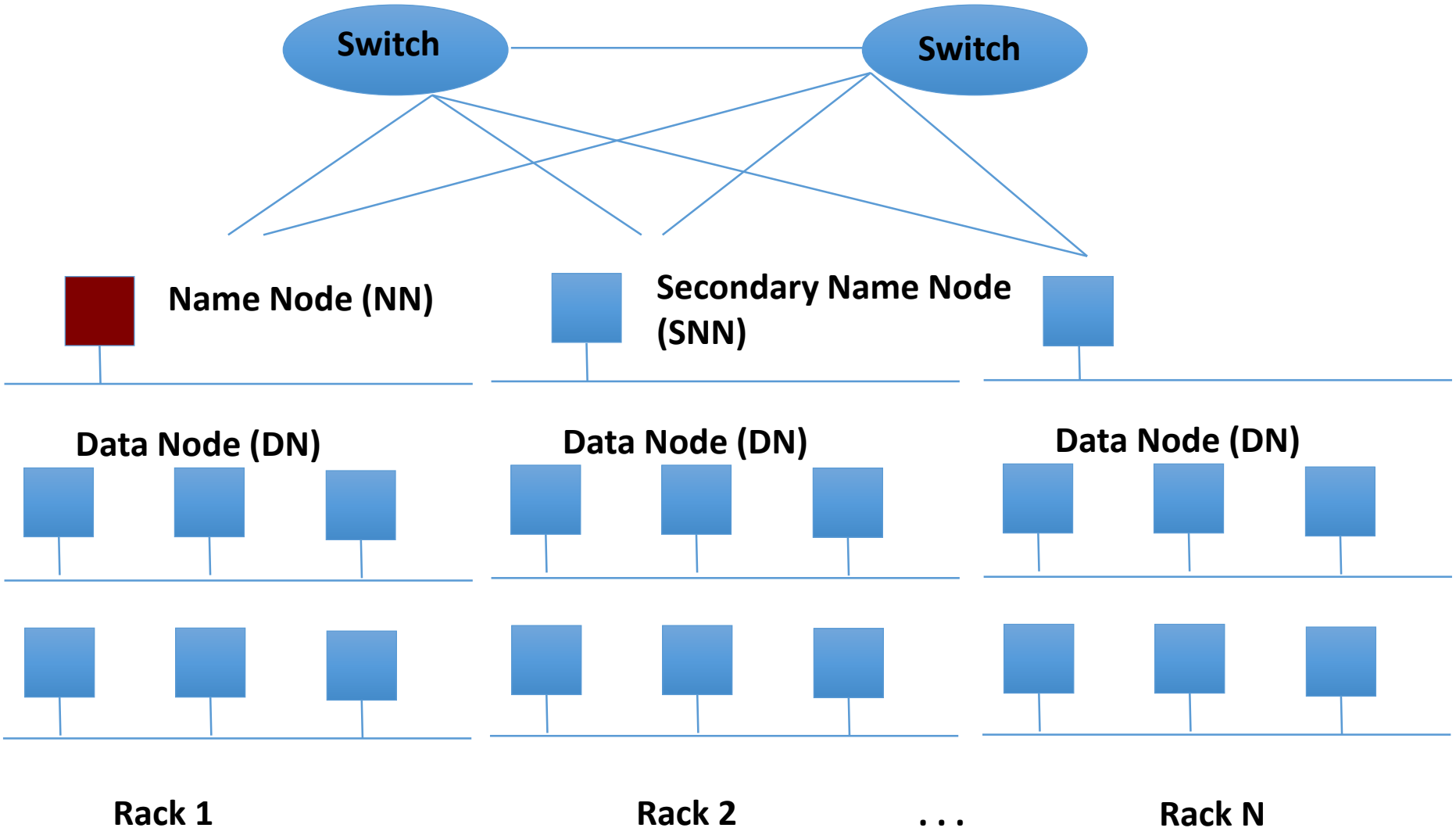
HDFS Architecture: Master-Slave



- Name Node: Controller
  - File System Name Space Management
  - Block Mappings
- Data Node: Work Horses
  - Block Operations
  - Replication
- Secondary Name Node:
  - Checkpoint node

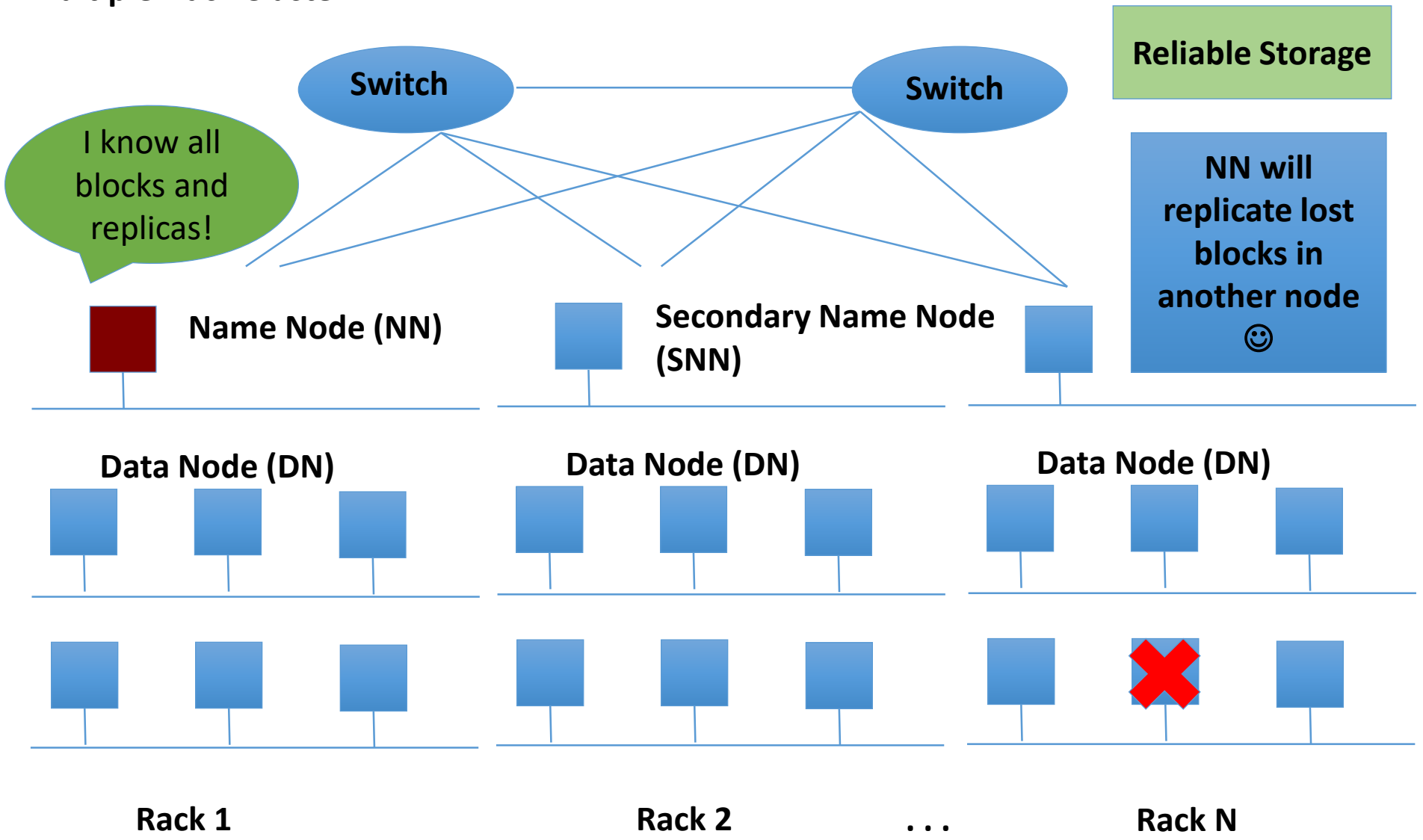
HDFS Architecture: Master-Slave

Multiple-Rack Cluster



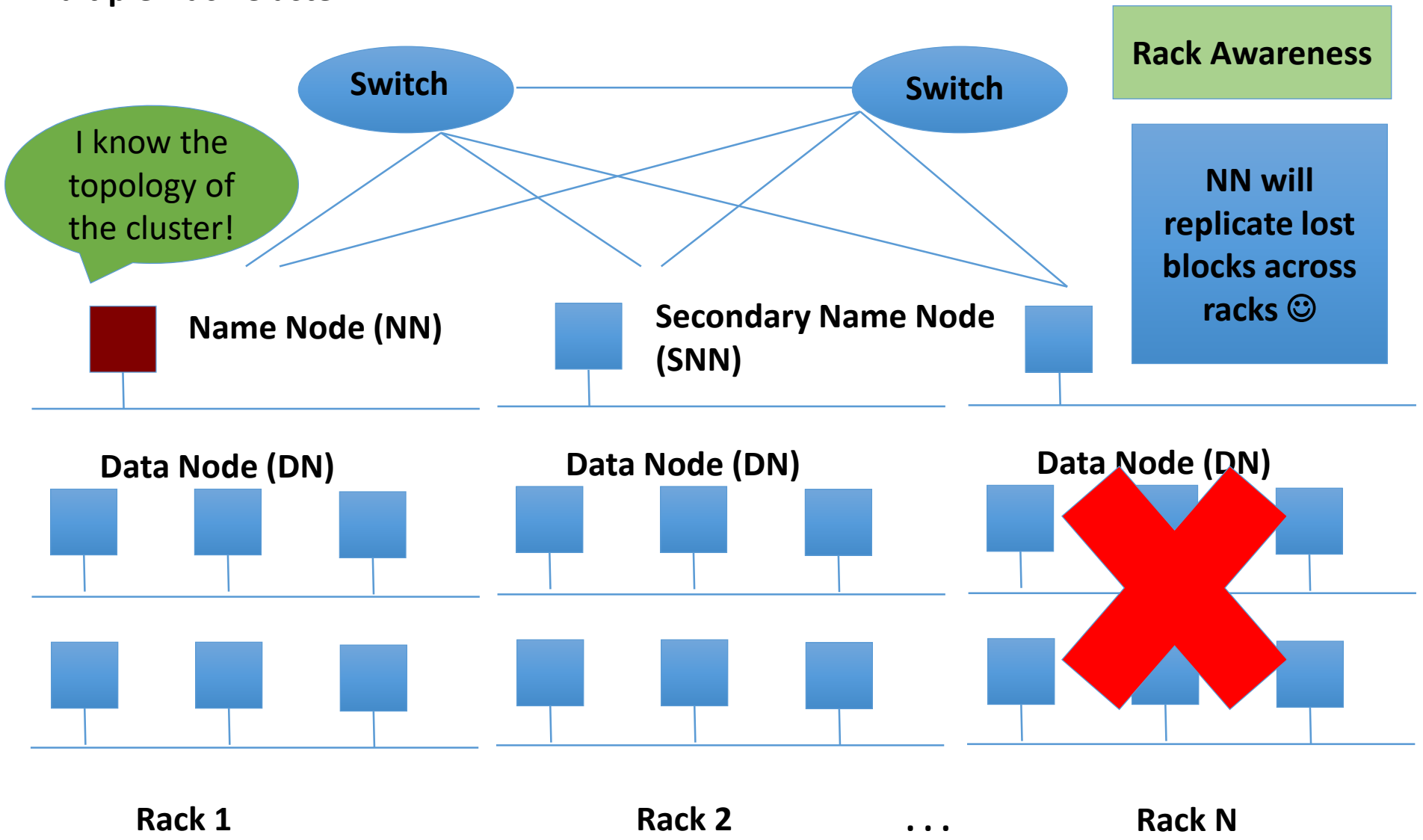
HDFS Architecture: Master-Slave

Multiple-Rack Cluster



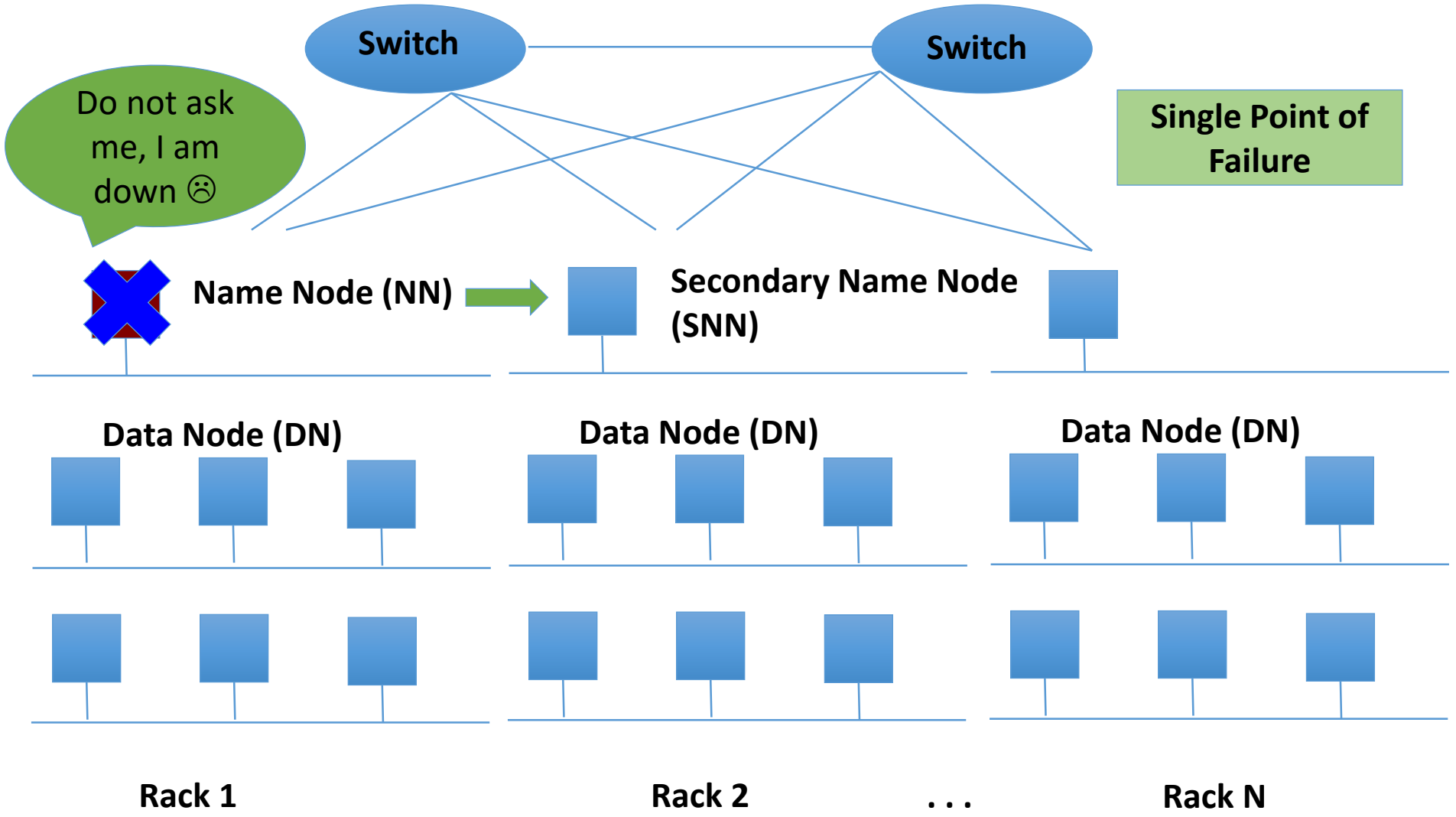
HDFS Architecture: Master-Slave

Multiple-Rack Cluster



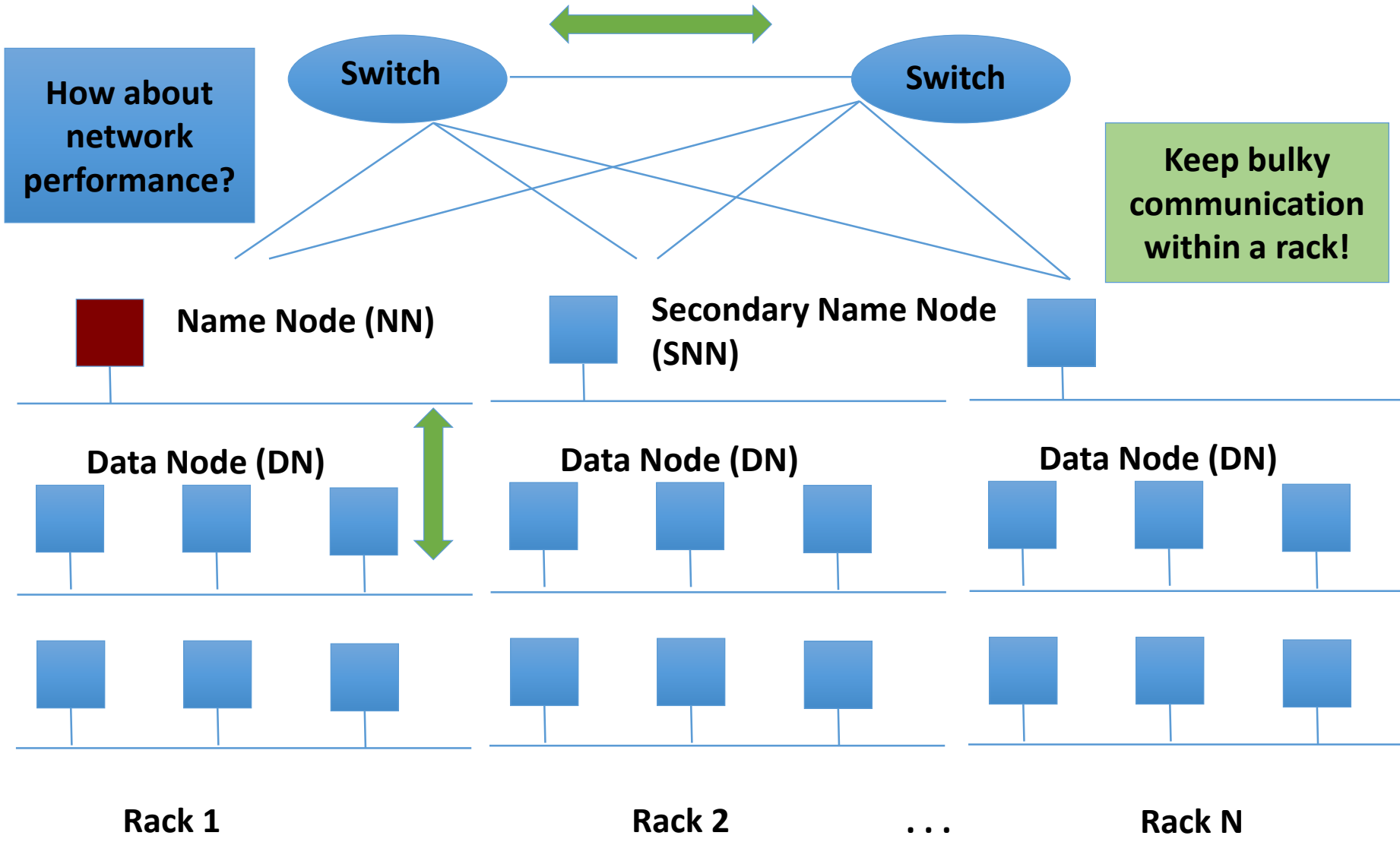
HDFS Architecture: Master-Slave

Multiple-Rack Cluster

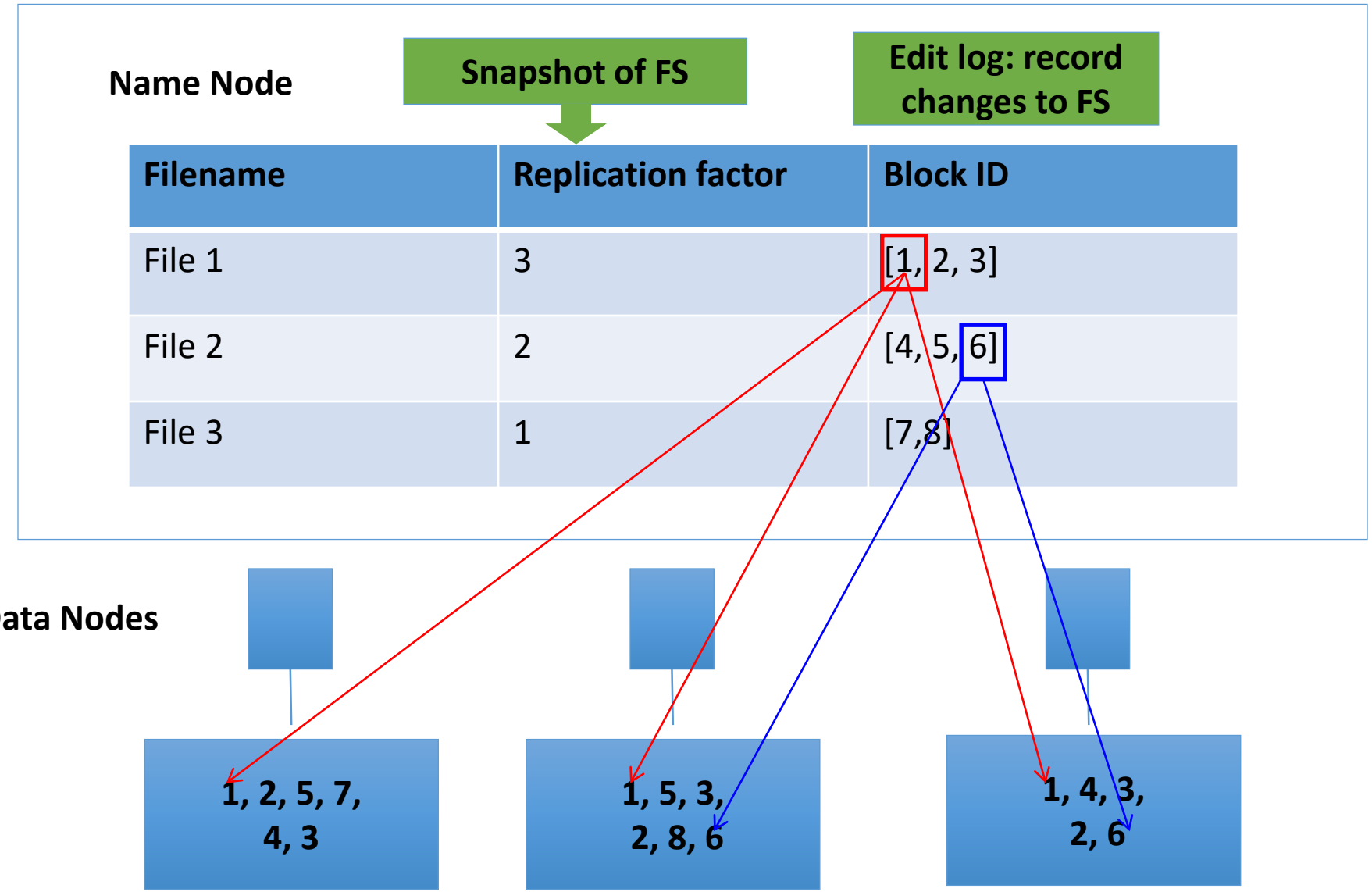


HDFS Architecture: Master-Slave

Multiple-Rack Cluster



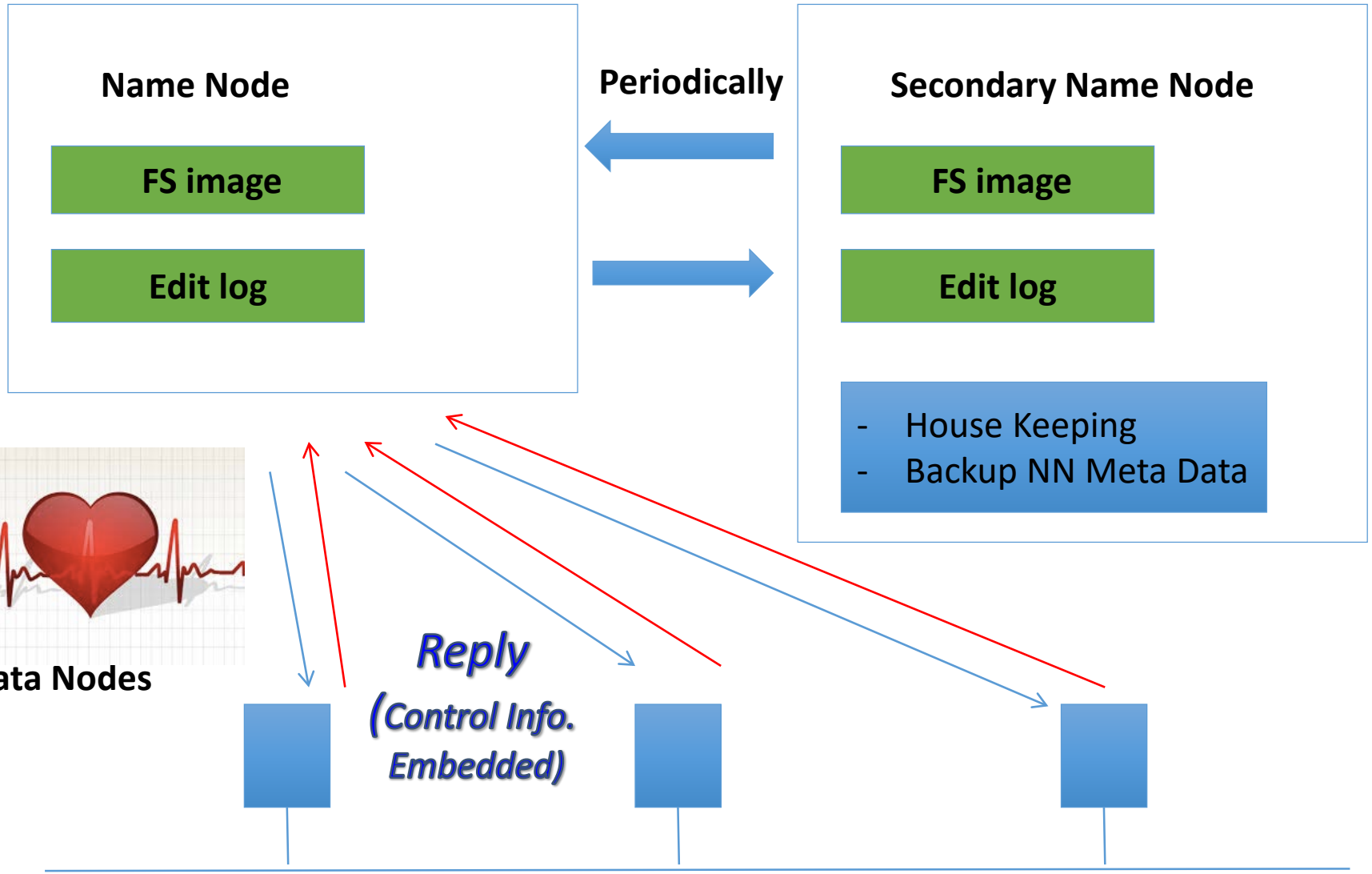
HDFS Inside: Name Node



HDFS Inside: Blocks

- Q: Why do we need the abstraction “Blocks” in addition to “Files”?
- Reasons:
  - File can be larger than a single disk
  - Block is of fixed size, easy to manage and manipulate
  - Easy to replicate and do more fine grained load balancing

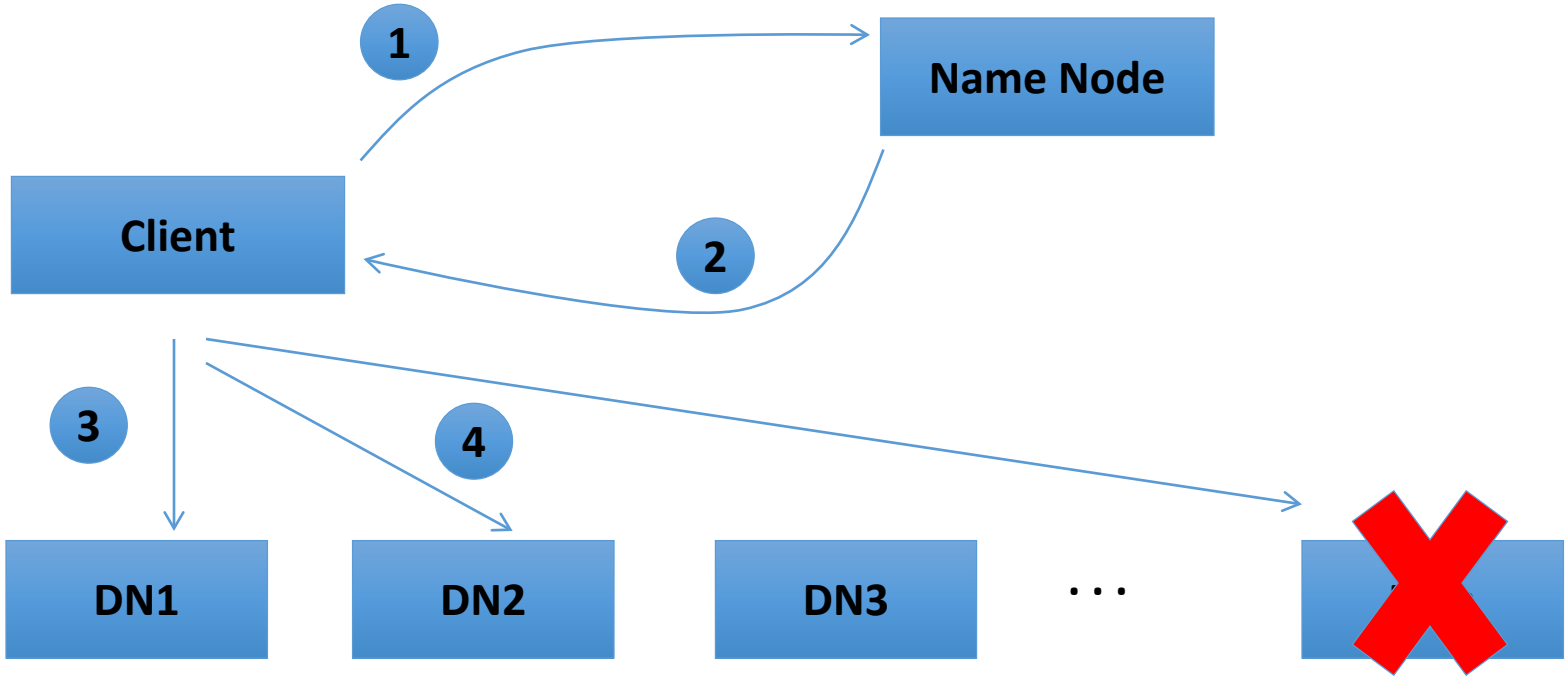
HDFS Inside: Name Node



HDFS Inside: Blocks

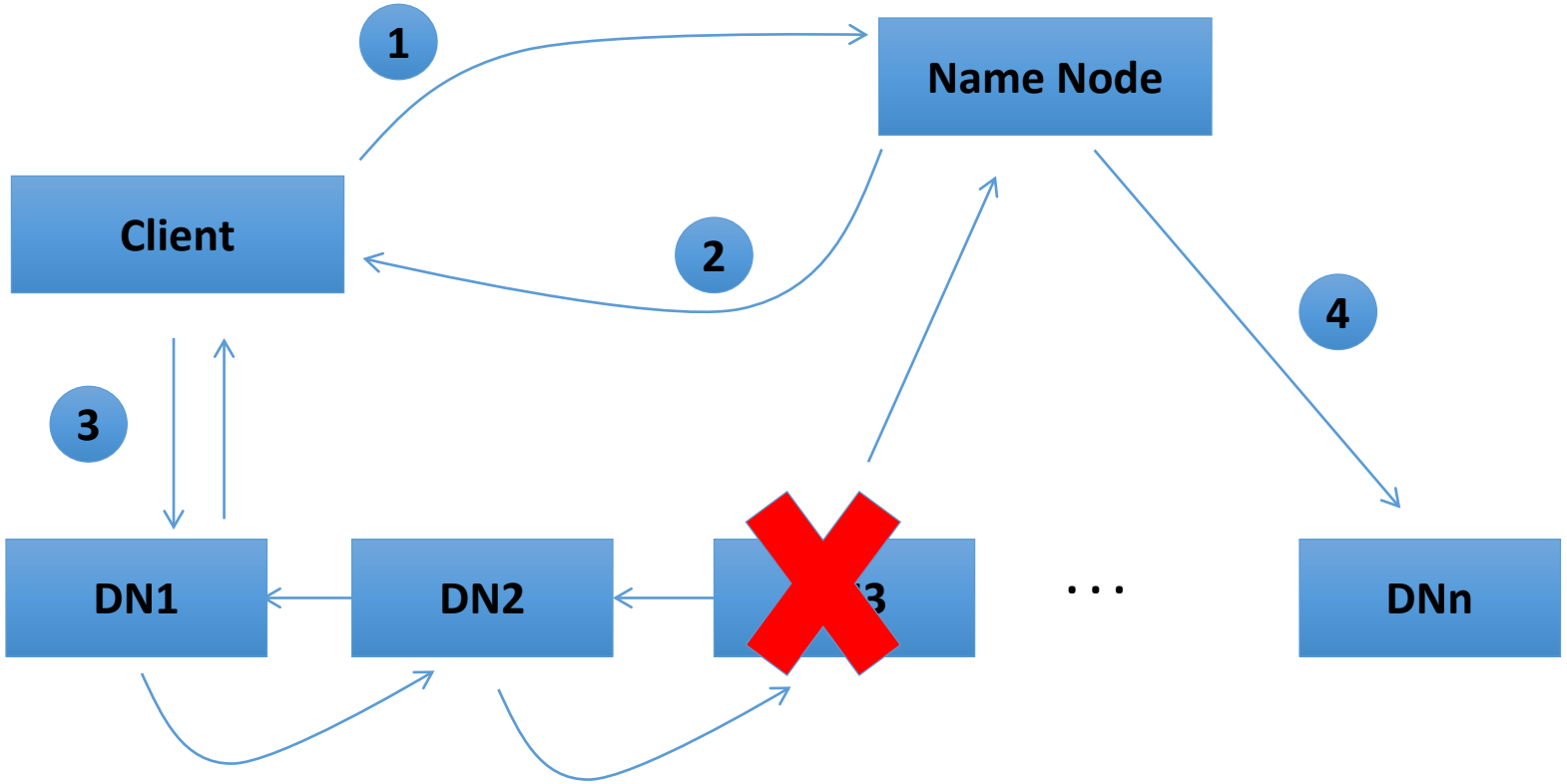
- HDFS Block size is by default **128 MB**, why it is much larger than regular file system block?
- Reasons:
  - Minimize overhead: disk seek time is almost constant
  - Example: seek time: 10 ms, file transfer rate: 100MB/s, overhead (seek time/a block transfer time) is 1%, what is the block size?
  - 100 MB (HDFS→ 128 MB)

HDFS Inside: Read



1. Client connects to NN to read data
2. NN tells client where to find the data blocks
3. Client reads blocks directly from data nodes (without going through NN)
4. In case of node failures, client connects to another node that serves the missing block

HDFS Inside: Write



1. Client connects to NN to write data
2. NN tells client write these data nodes
3. Client writes blocks directly to data nodes with desired replication factor
4. In case of node failures, NN will figure it out and replicate the missing blocks

HDFS Command Line

- Hadoop Shell

```
[dwang5@disc01 ~]$ hadoop fs
Jsage: java FsShell
           [-ls <path>]
           [-lsr <path>]
           [-df [<path>]]
           [-du [-s] [-h] <path>]
           [-dus <path>]
           [-count[-q] <path>]
           [-mv <src> <dst>]
           [-cp <src> <dst>]
           [-rm [-skipTrash] <path>]
           [-rmr [-skipTrash] <path>]
           [-expunge]
           [-put <localsrc> ... <dst>]
           [-copyFromLocal <localsrc> ... <dst>]
           [-moveFromLocal <localsrc> ... <dst>]
           [-get [-ignoreCrc] [-crc] <src> <localdst>]
           [-getmerge <src> <localdst> [addnl]]
           [-cat <src>]
           [-text <src>]
           [-copyToLocal [-ignoreCrc] [-crc] <src> <localdst>]
           [-moveToLocal [-crc] <src> <localdst>]
           [-mkdir <path>]
           [-setrep [-R] [-w] <rep> <path/file>]
           [-touchz <path>]
           [-test -[ezd] <path>]
           [-stat [format] <path>]
```

Hadoop MapReduce

What is MapReduce?

- Terms are borrowed from Functional Language (e.g., Lisp)
- Sum of squares:
- (map square ‘(1 2 3 4))
  - Output: (1 4 9 16)
  - [processes each record sequentially and independently]

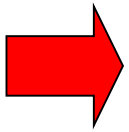
- (reduce + ‘(1 4 9 16))
  - (+ 16 (+ 9 (+ 4 1)))
  - Output: 30
  - [processes set of all records in batches]

- Let’s consider a sample application: Wordcount
  - You are given a huge dataset (e.g., Wikipedia dump or all of Shakespeare’s works) and asked to list the count for each of the words in each of the documents therein

Map

- Process individual records to generate intermediate key/value pairs.

Welcome Everyone
Hello Everyone

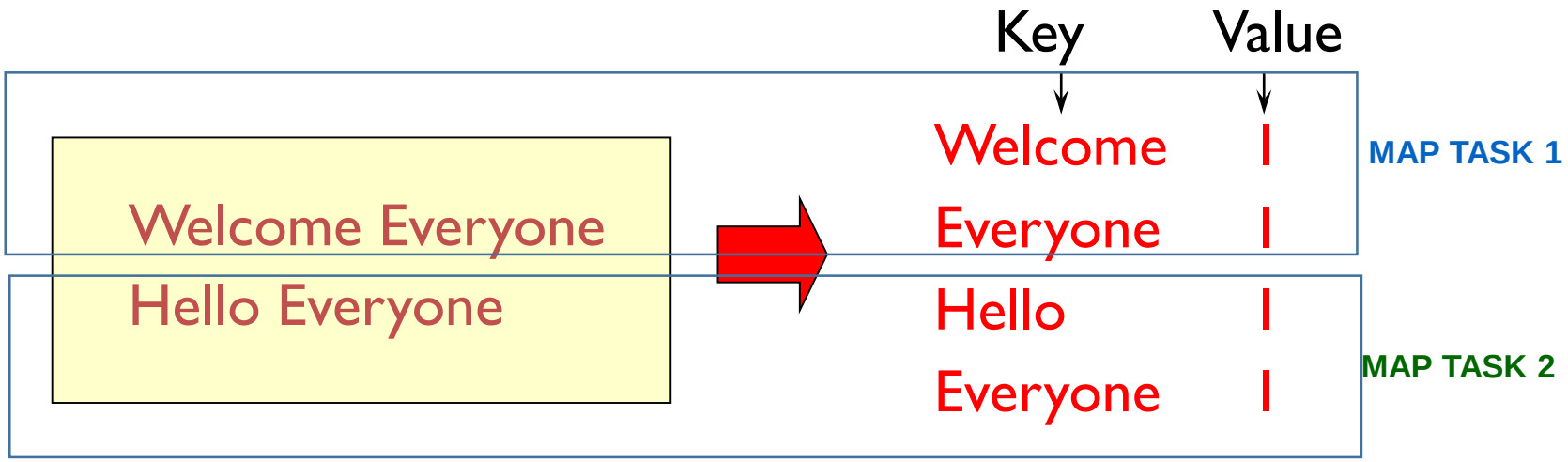


Key	Value
Welcome	1
Everyone	1
Hello	1
Everyone	1

Input <filename, file text>

Map

- Parallelly Process individual records to generate intermediate key/value pairs.



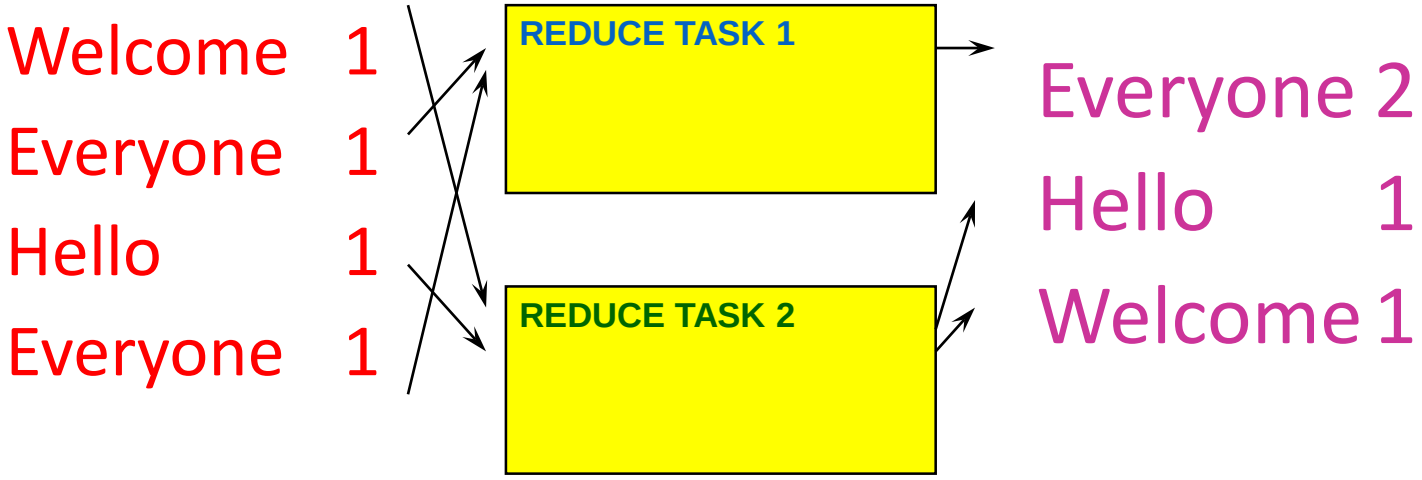
Input <filename, file text>

Reduce

- Reduce processes and merges all intermediate values associated per key

Reduce

- Each key assigned to one Reduce
- Parallelly Processes and merges all intermediate values by partitioning keys



- Popular: Hash partitioning, i.e., key is assigned to reduce # = hash(key)%number of reduce servers

Hadoop Code - Map

```
public static class MapClass extends MapReduceBase implements
                Mapper<LongWritable, Text, Text, IntWritable> {

  private final static IntWritable one = new IntWritable(1);

  private Text word = new Text();


  public void map( LongWritable key, Text value, OutputCollector<Text,
IntWritable> output, Reporter reporter) throws IOException {

    String line = value.toString();

    StringTokenizer itr = new StringTokenizer(line);

    while (itr.hasMoreTokens()) {

            word.set(itr.nextToken());

            output.collect(word, one);

    }

}
```

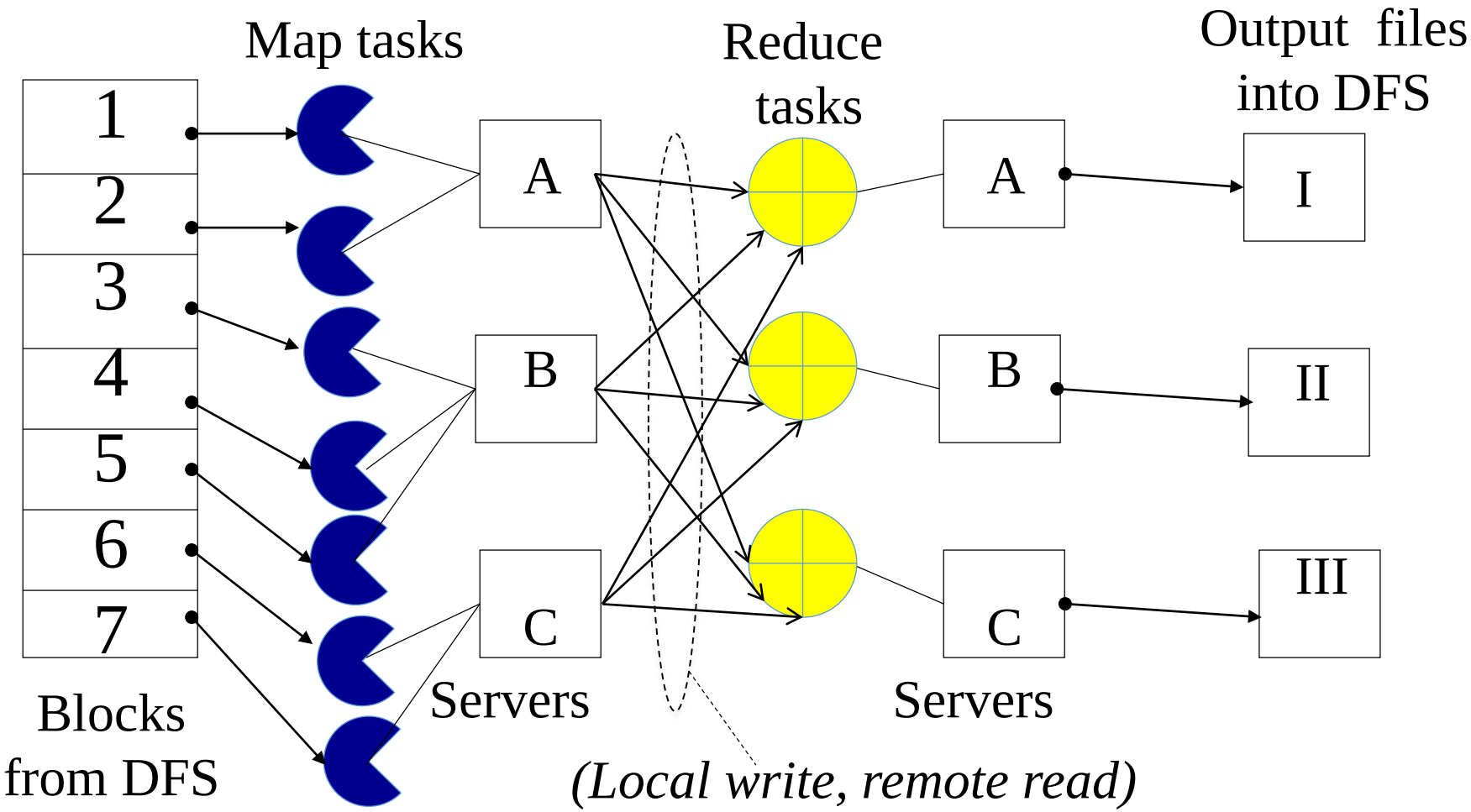
Hadoop Code - Reduce

```
public static class ReduceClass extends MapReduceBase implements
Reducer<Text, IntWritable, Text, IntWritable> {

  public void reduce(Text key, Iterator<IntWritable> values,
OutputCollector<Text, IntWritable> output, Reporter reporter)
throws IOException {

        int sum = 0;

        while (values.hasNext()) {

          sum += values.next().get();

        }

        output.collect(key, new IntWritable(sum));

  }

}
```

Hadoop Code - Driver

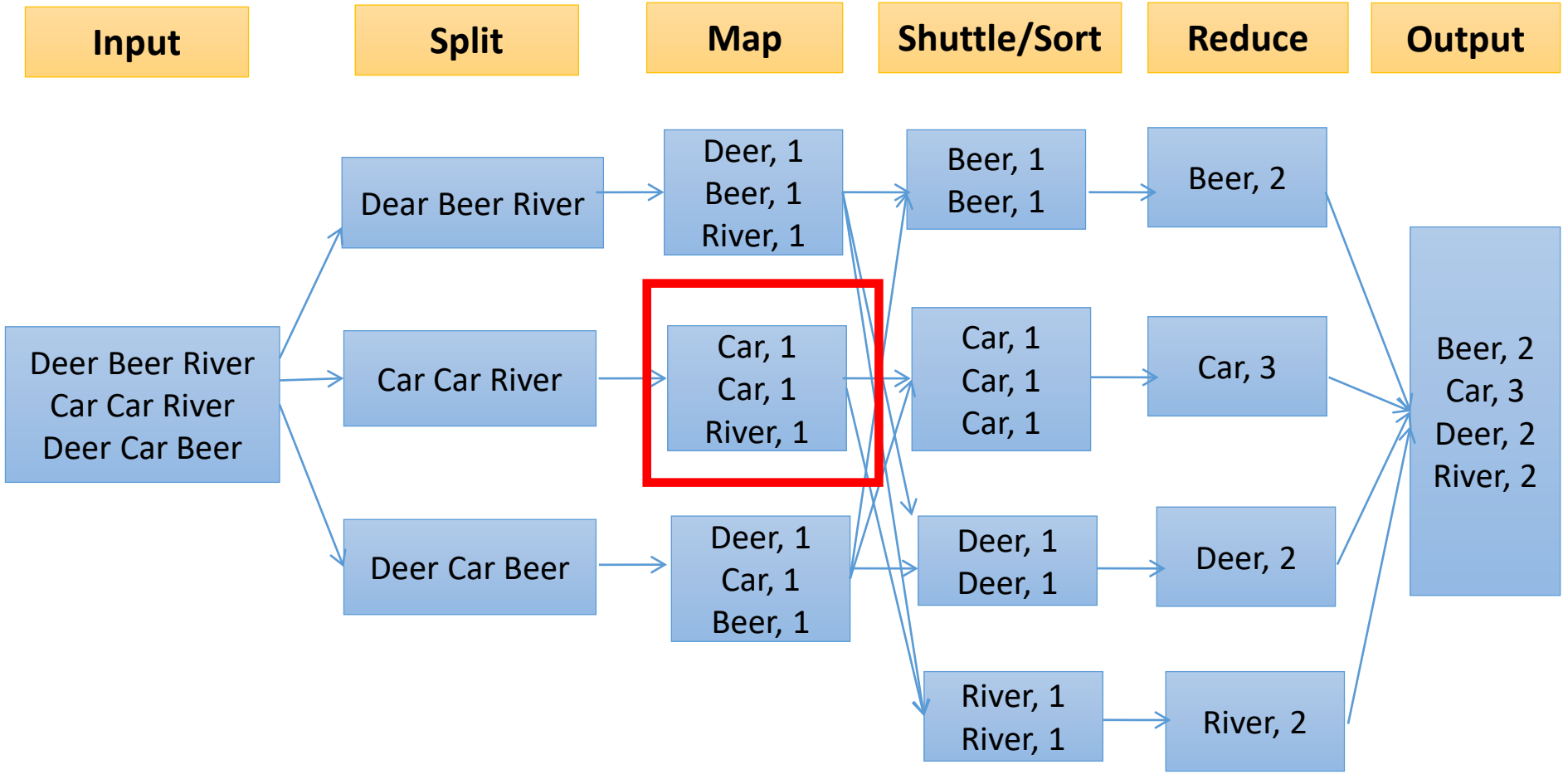
```
// Tells Hadoop how to run your Map-Reduce job
public void run (String inputPath, String outputPath) throws Exception
{
  // The job. WordCount contains MapClass and Reduce.
  JobConf conf = new JobConf(WordCount.class);
  conf.setJobName(”mywordcount");
  // The keys are words
  (strings) conf.setOutputKeyClass(Text.class);
  // The values are counts (ints)
  conf.setOutputValueClass(IntWritable.class);
  conf.setMapperClass(MapClass.class);
  conf.setReducerClass(ReduceClass.class);
  FileInputFormat.addInputPath(conf, newPath(inputPath));
  FileOutputFormat.setOutputPath(conf, new Path(outputPath));
  JobClient.runJob(conf);
}
```

MapReduce Data Flow



Resource Manager (assigns maps and reduces to servers)
Idea: Bring computations to data!

MapReduce Example: Word Count

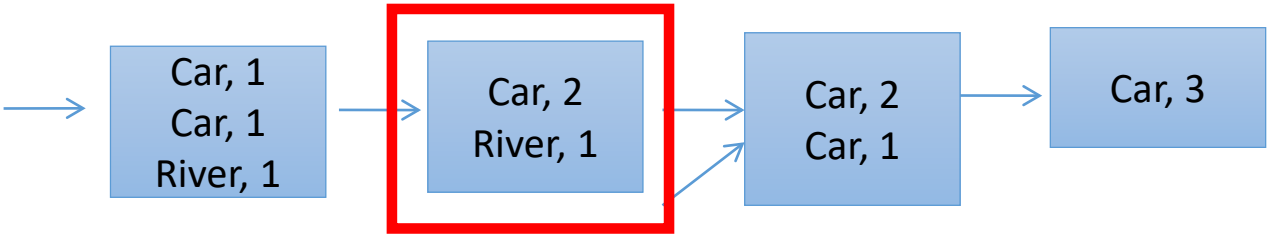


Q: Do you see any place we can improve the efficiency?

Local aggregation at mapper will be able to improve MapReduce efficiency.

MapReduce: Combiner

- Combiner: do local aggregation/combine task at mapper



- Q: What are the benefits of using combiner:
  - Reduce memory/disk requirement of Map tasks
  - Reduce network traffic
- Q: Can we remove the reduce function?
  - No, reducer still needs to process records with same key but from different mappers
- Q: How would you implement combiner?
  - It is the same as Reducer!

MapReduce In-class Exercise

- Functions that can use combiner are called distributive:
  - Distributive: Min/Max(), Sum(), Count(), TopK()
  - Non-distributive: Mean(), Median(), Rank()