INTRO TO HADOOP

Dr. Chatree Sangpachatanaruk
Computer Science Department
University of Pittsburgh

MOTIVATIONS

- Big Data!
- Storage and Analysis
  - Storage capacity increases faster than the access speeds

Year	HD Size	Transfer Speed (MB/s)	Time to read the whole drive
1990	1,370 MB (1GB)	4.4 MB/s	~ 5 minutes
2000	1 TB or 1000 GB	100 MB/s	~ 2.5 hours

  - Need parallel data access to get thing done quickly
    - 100 machines – each accesses 10 GB ~ 1.7 minutes
  - Shared access for efficiency and scalability

CHALLENGES

- Challenge of parallel data access to and from multiple disks
  - Hardware failure
- Analysis tasks need to combined data from multiple sources
  - Need a paradigm that split and merge data transparently

WHY HADOOP?

- Hadoop is an open-source software framework for storing data and running applications on clusters of commodity hardware.

- Provide massive storage for any kind of data, enormous processing power and the ability to handle virtually limitless concurrent tasks or jobs

- It consists of two main components
  - ***Hadoop Distributed File System (HDFS)***
  - ***MapReduce*** – a framework for processing data in batch (asynchronous)

HADOOP VS GRID COMPUTING

- Existing Grid Computing, ie. HPC
  - Distribute tasks to process data in a shared file system
  - Data need to move to the machines that run tasks
  - Not suitable for tasks accessing large data volumes

- Hadoop
  - Try to co-locate the data with the computing node
  - Avoid copying data around
  - Automate fault recovery

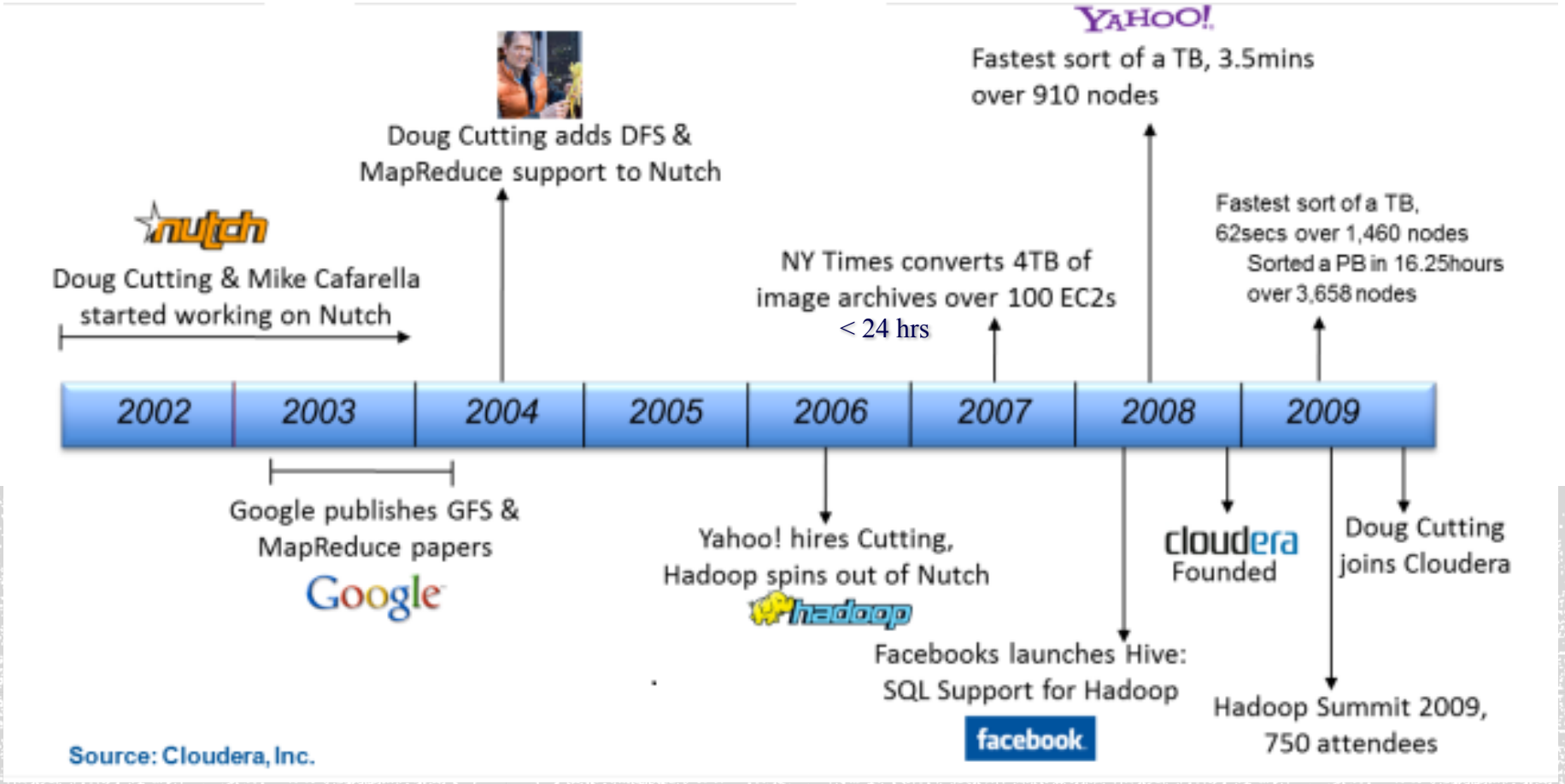
HADOOP VS VOLUNTEER COMPUTING

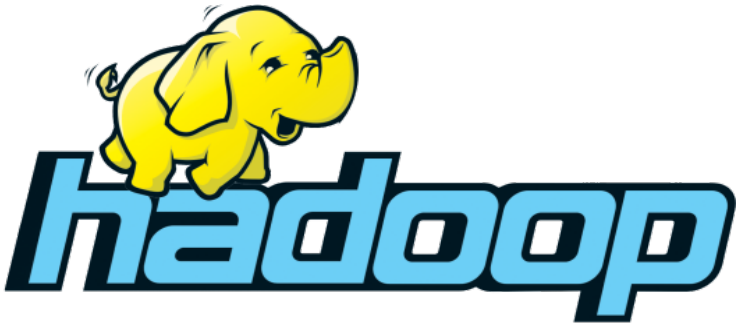
- Volunteer Computing – SETI@home
  - SETI – Search for Extra Terestrial Intelligence
  - Volunteers donate CPU cycle from their idle computers to analyze radio telescope data
  - Small chuck of data is sent to each system to analyze and the result is returned in asynchronous manner
  - To improve reliability, one chuck is sent to multiple (3) systems
  - Suitable for CPU intensive tasks

- Hadoop
  - Suitable for data intensive tasks

HADOOP HISTORY

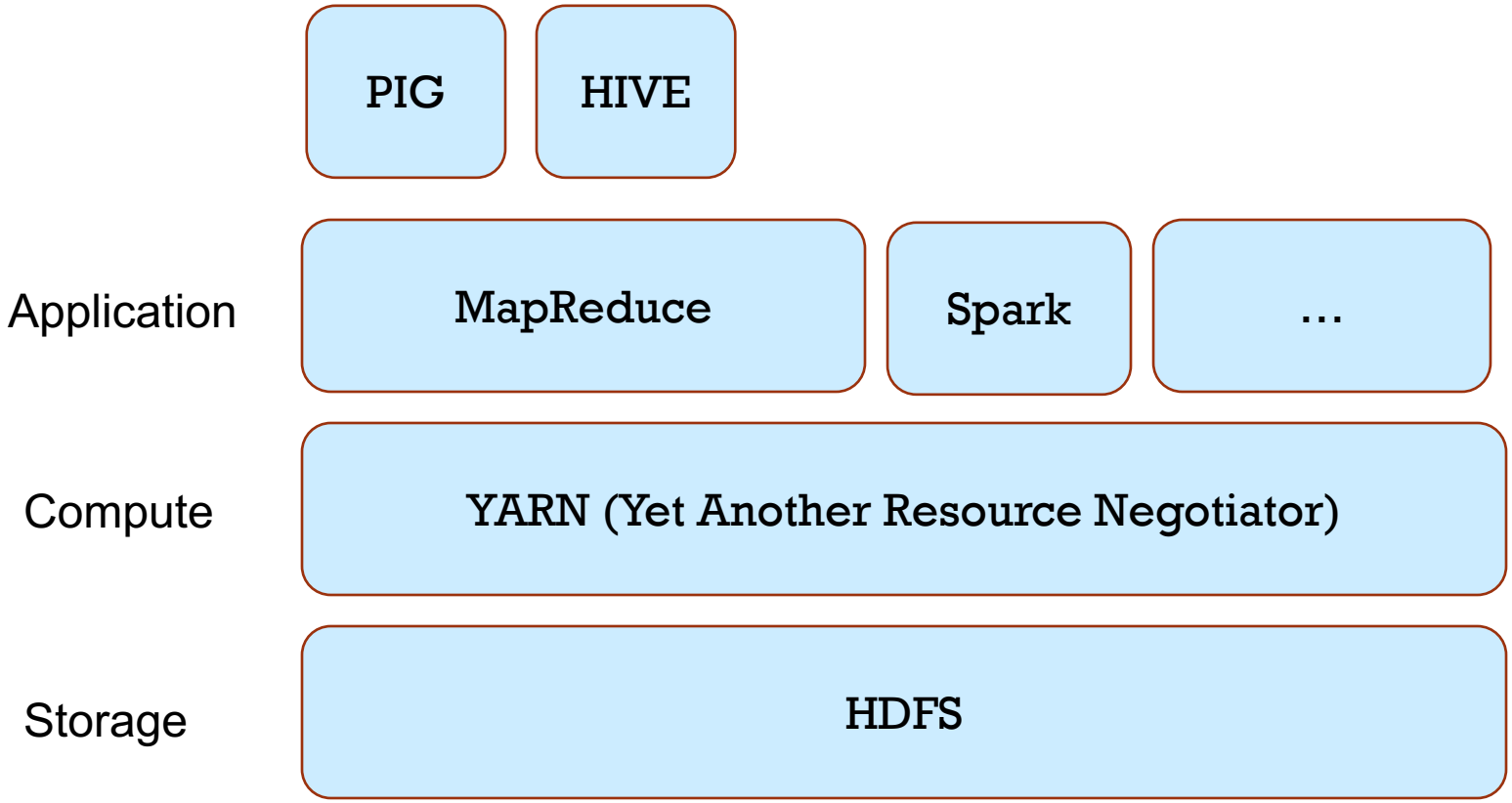
In 2008, Hadoop became the Apache top level project





"The name my kid gave a stuffed yellow elephant. Short, relatively easy to spell and pronounce, meaningless, and not used elsewhere: those are my naming criteria." … Doug Cutting

HADOOP SOFTWARE ARCHITECTURE



MAPREDUCE

- A programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a ***Hadoop*** cluster.
- It abstracts the problem from disk reads and writes, transforming it into a computation over sets for ***keys and values***
- It works as a ***batch query processor*** that can run an ad hoc query against your whole data set and get result in a reasonable time

WORD COUNT EXAMPLE

I can not do everything, but still I can do something; and because I cannot do everything, I will not refuse to do something I can do

Word	Count
And	1
Because	1
But	1
Can	4
Do	4
Everything	2
I	5
Not	3
Refuse	1
Something	2
Still	1
To	1
Will	1

WORDCOUNT PROGRAM – I

```
Define WordCount as Multiset;
  For Each Document in DocumentSet {
     T = tokenize(document);
     For Each Token in T {
     WordCount[token]++;
    }
 }
Display(WordCount);
```

Program Does NOT Scale for Large Number of Documents

WORDCOUNT PROGRAM – II

- A two-phased program can be used to speed up execution by distributing the work over several machines and combining the outcome from each machine into the final word count
- Phase I – Document Processing
  - Each machine will process a fraction of the document set
- Phase II – Count Aggregation
  - Partial word counts from individual machines are combined into the final word count

WORDCOUNT PROGRAM – II

Phase I

- Define WordCount as Multiset;
- For Each Document in DocumentSubset {
  - T = tokenize(document);
  - For Each Token in T {
    - WordCount[token]++;
- } }SendToSecondPhase(wordCount);

Phase II

- Define TotalWordCount as Multiset;
- for each WordCount Received From firstPhase {
  - MultisetAdd (TotalWordCount, WordCount);
- }

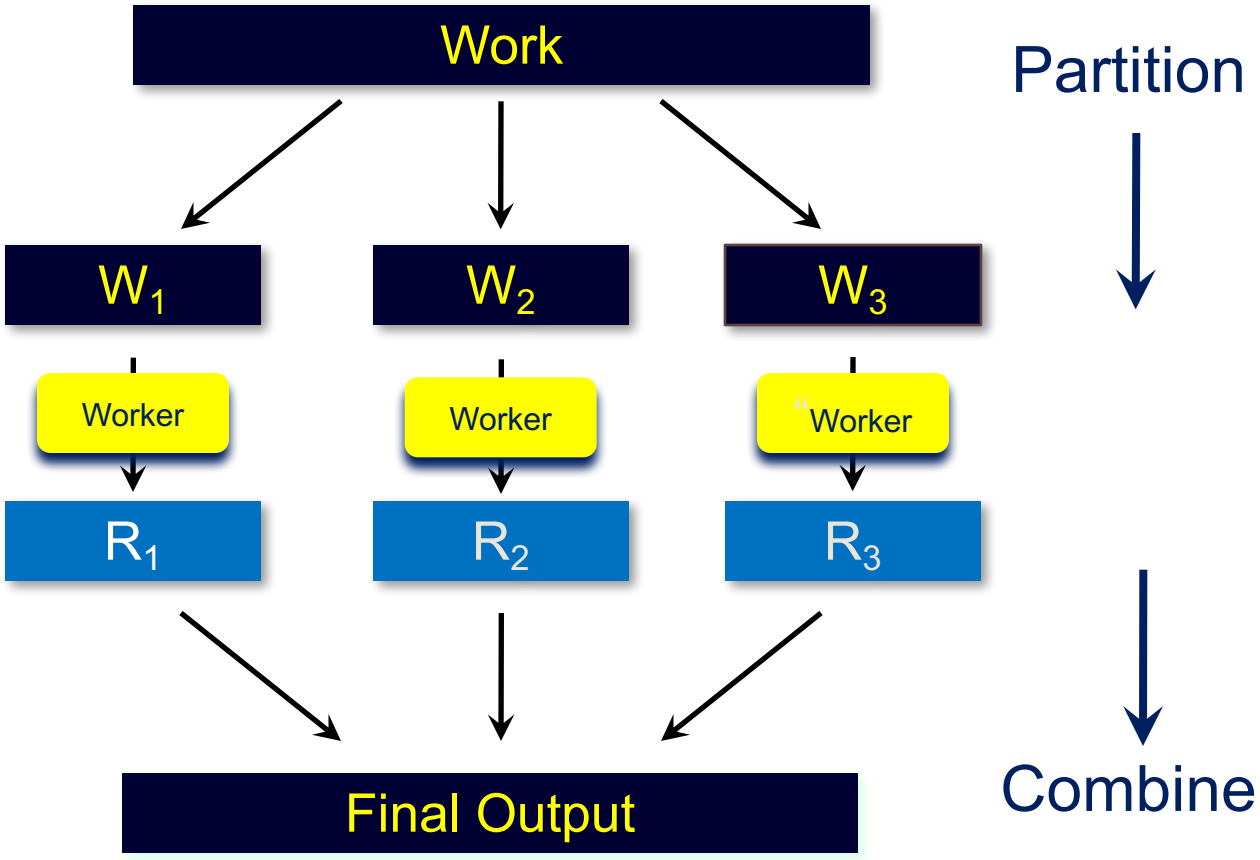
WORDCOUNT PROGRAM II – LIMITATIONS

- The program does not take into consideration the location of the documents
  - Storage server can become a bottleneck, if enough bandwidth is not available
    - Distribution of documents across multiple machines removes the central server bottleneck
- Storing WordCount and TotalWordCount in the memory is a flaw
  - When processing large document sets, the number of unique words can exceed the RAM capacity
- In Phase II, the aggregation machine becomes the bottleneck

WORDCOUNT PROGRAM – SOLUTION

- The aggregation phase must execute in a distributed fashion on a cluster of machines that can run independently
- To achieve this, functionalities must be added
  - Store files over a cluster of processing machines.
  - Design a disk-based hash table permitting processing without being limited by RAM capacity.
  - Partition intermediate data across multiple machines
  - Shuffle the partitions to the appropriate machines

DIVIDE AND CONQUER



PARALLELIZATION CHALLENGES

- How do we assign work units to workers?
- What if we have more work units than workers?
- What if workers need to share partial results?
- How do we aggregate partial results?
- How do we know all the workers have finished?
- What if workers fail?

DISTRIBUTED WORKERS COORDINATION

- Coordinating a large number of workers in a distributed environment is challenging
  - The order in which workers run may be unknown
  - The order in which workers interrupt each other may be unknown
  - The order in which workers access shared data may be unknown

MAPREDUCE
DATA-INTENSIVE PROGRAMMING MODEL

- MapReduce is a programming model for processing large sets
- Users specify the computation in terms of a **map()** and a **reduce()** function,
  - Underlying runtime system automatically parallelizes the computation across large-scale clusters of machines, and
  - Underlying system also handles machine failures, efficient communications, and performance issues.

TYPICAL LARGE-DATA PROBLEM

- At a high-level of abstraction, MapReduce codifies a generic "recipe" for processing large data set
  - Iterate over a large number of records
  - Extract something of interest from each
  - Shuffle and sort intermediate results
  - Aggregate intermediate results
  - Generate final output

Map

Reduce

Basic Tenet of MapReduce is Enabling a Functional Abstraction for the Map() and Reduce() operations

MAPREDUCE FRAMEWORK

- Programmers specify two functions:

  map (k, v) → <k', v'>*

  reduce (k', v') → <k', v'>*

  - All values with the same key are sent to the same reducer

- The execution framework supports a computational runtime environment to handle all issues related to coordinate the parallel execution of a data-intensive computation in a large-scale environment
  - Breaking up the problem into smaller tasks, coordinating workers executions, aggregating intermediate results, dealing with failures and softeare errors, …

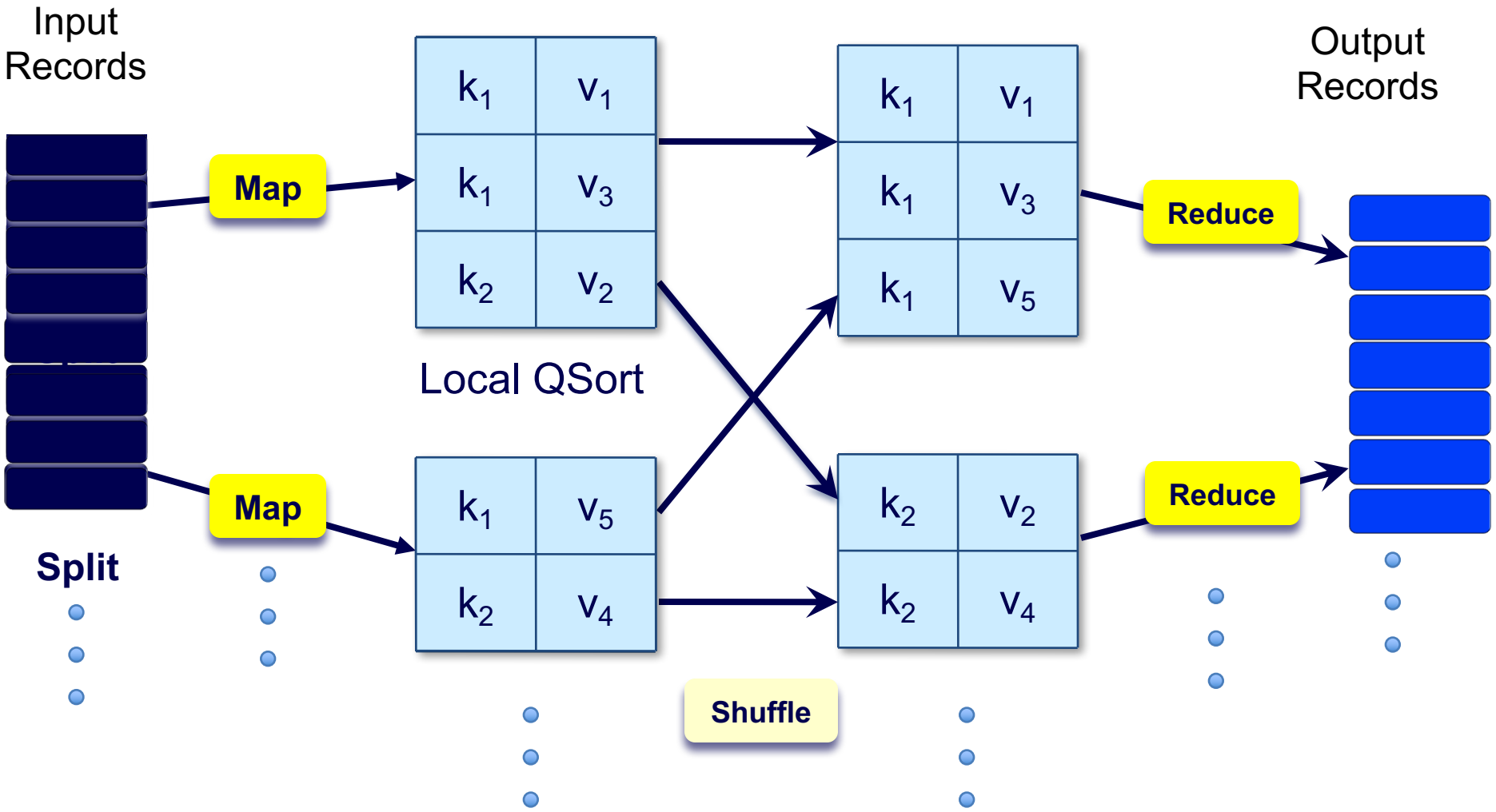
MAPREDUCE "RUNTIME" BASIC FUNCTIONS

- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles "data distribution"
  - Moves processes to data, not data to processes
- Handles synchronization among workers
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults, dynamically
  - Detects worker failures and restarts

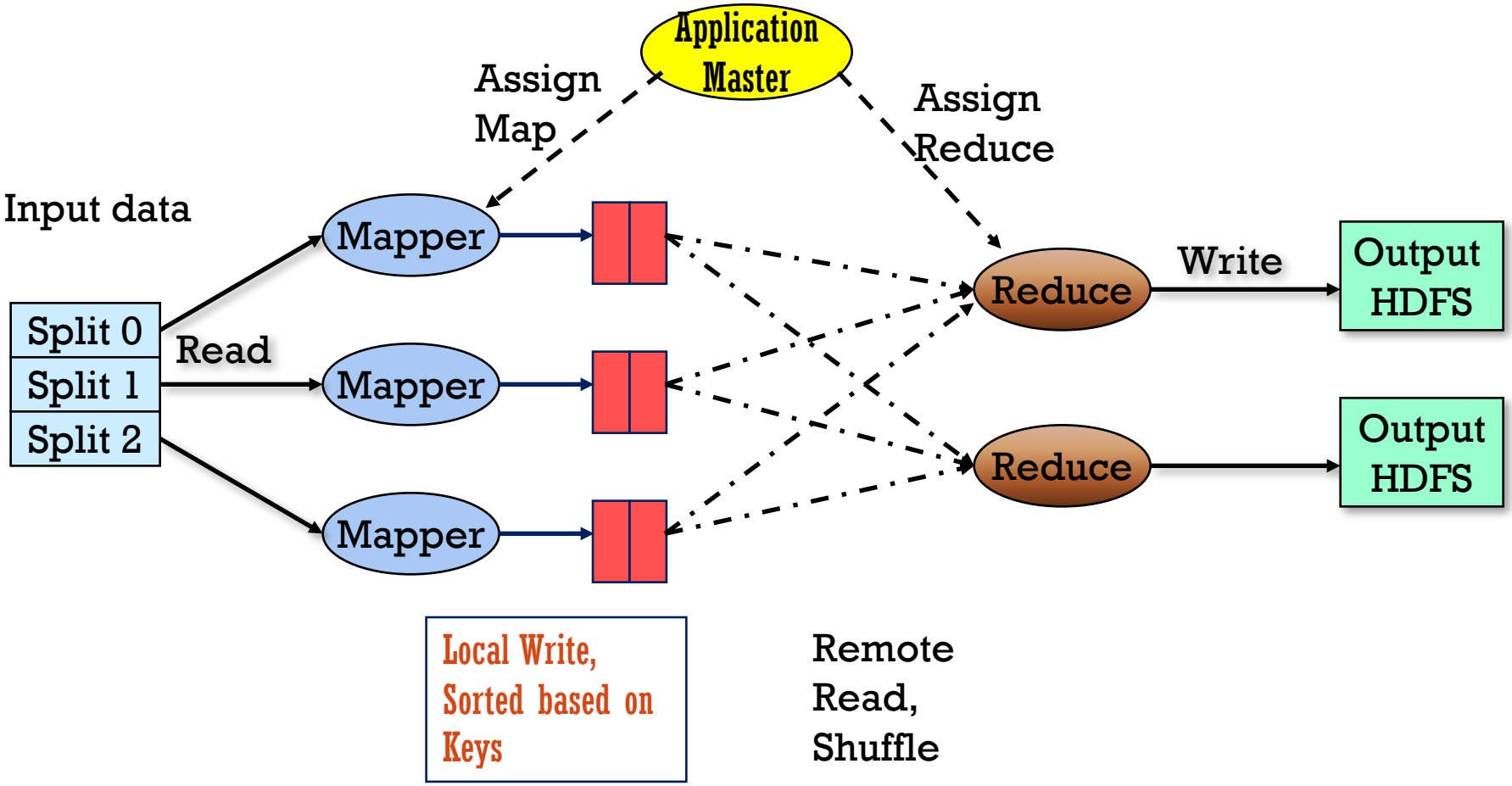
MAPREDUCE DATA FLOW

- A MapReduce job is a unit of work to be performed
  - Job consists of the **MapReduce Program**, the **Input data** and the **Configuration Information**
- The MapReduce job is divided it into two types of tasks – ***map*** tasks and ***reduce*** tasks
  - It is not uncommon for MapReduce jobs to have thousands of individual tasks to be assigned to cluster nodes
- The Input data is divided into fixed-size pieces called ***splits***
  - One map task is created for each split
  - The user-defined map function is run on each split
- Configuration information indicates where the input lies, and the output is stored

MAPREDUCE DATA FLOW



DISTRIBUTED EXECUTION OVERVIEW



FROM MAP TO REDUCE

- Select a number *R* of reduce tasks.
- Divide the intermediate keys into *R* groups,
  - Use an efficient hashing function
- Each Map task creates, at its own processor, *R* files of intermediate key-value pairs, sorted by key, and one for each Reduce task
- Its output file is transfer to the Reducer's container
- Once all Mapper tasks results are transferred to the Reducer, it merges the results and compute the final output
- The Reducer then writes the output to HDFS

MAPREDUCE – WHAT YOU NEED TO DO

- Programmers specify two functions:

  map (k, v) → <k', v'>*

  reduce (k', v') → <k', v'>*

  - All values with the same key are reduced together

- The execution framework handles everything else!

- Not quite…usually, programmers also specify:

  partition (k', number of partitions) → partition for k'

  - Often a simple hash of the key, e.g., hash(k') mod n
  - Divides up key space for parallel reduce operations

  combine (k', v') → <k', v'>*

  - Mini-reducers that run in memory after the map phase
  - Used as an optimization to reduce network traffic

WORDCOUNT PSEUDOCODE

Map

```
map(String file, String doc)
{
  List<String> T = tokenize(doc);
  for each token in T
  {
        emit ((String)token, (Int) 1);
  }
}
```

Reduce

```
reduce(String token, List<Int> values)
{
   Integer sum = 0;
   for each value in values {
        sum = sum + value;
    }
  emit ((String)token,(Integer) sum);
}
```

WORDCOUNT – MAPREDUCE VERSION

```
public class WordCount {

  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

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

    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);
      }
    }
  }

  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);
    }
  }

  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);
  }
}
```

CODE EXPLAIN - MAPPER

```
public static class TokenizerMapper
     extends Mapper<Object, Text, Text, IntWritable>{

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

  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);
    }
  }
}
```

- The Mapper implementation, via the map method, processes one line at a time, as provided by the specified *Text* Input Format
- It then splits the line into tokens separated by whitespaces, via the *StringTokenizer*, and
- emits a key-value pair of < <word>, 1>.

CODE EXPLAIN - REDUCER

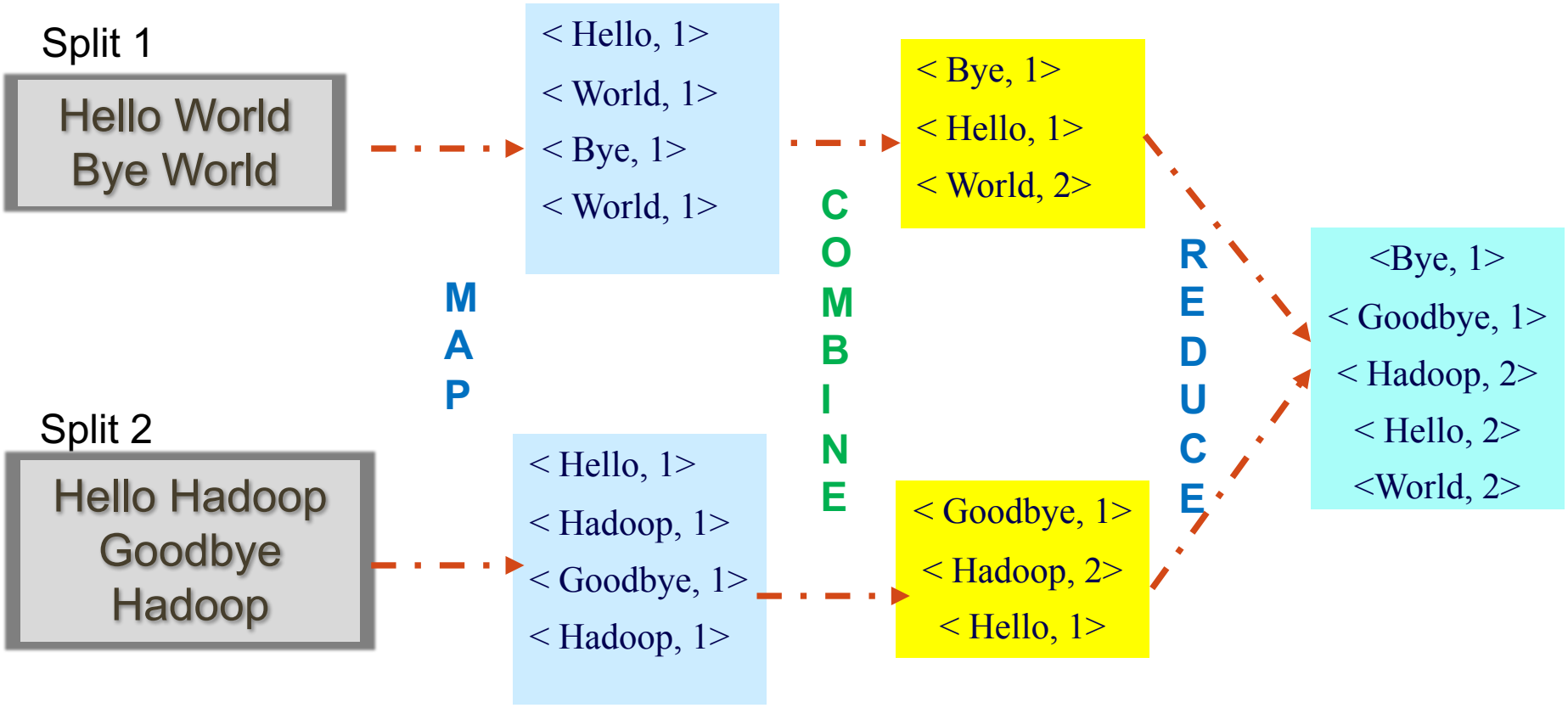
```
public static class IntSumReducer
     extends Reducer<Text,IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();

  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);
  }
}
```

- The Reducer implementation, via the reduce method just sums up the values, which are the occurrence counts for each key (i.e. words in this example).

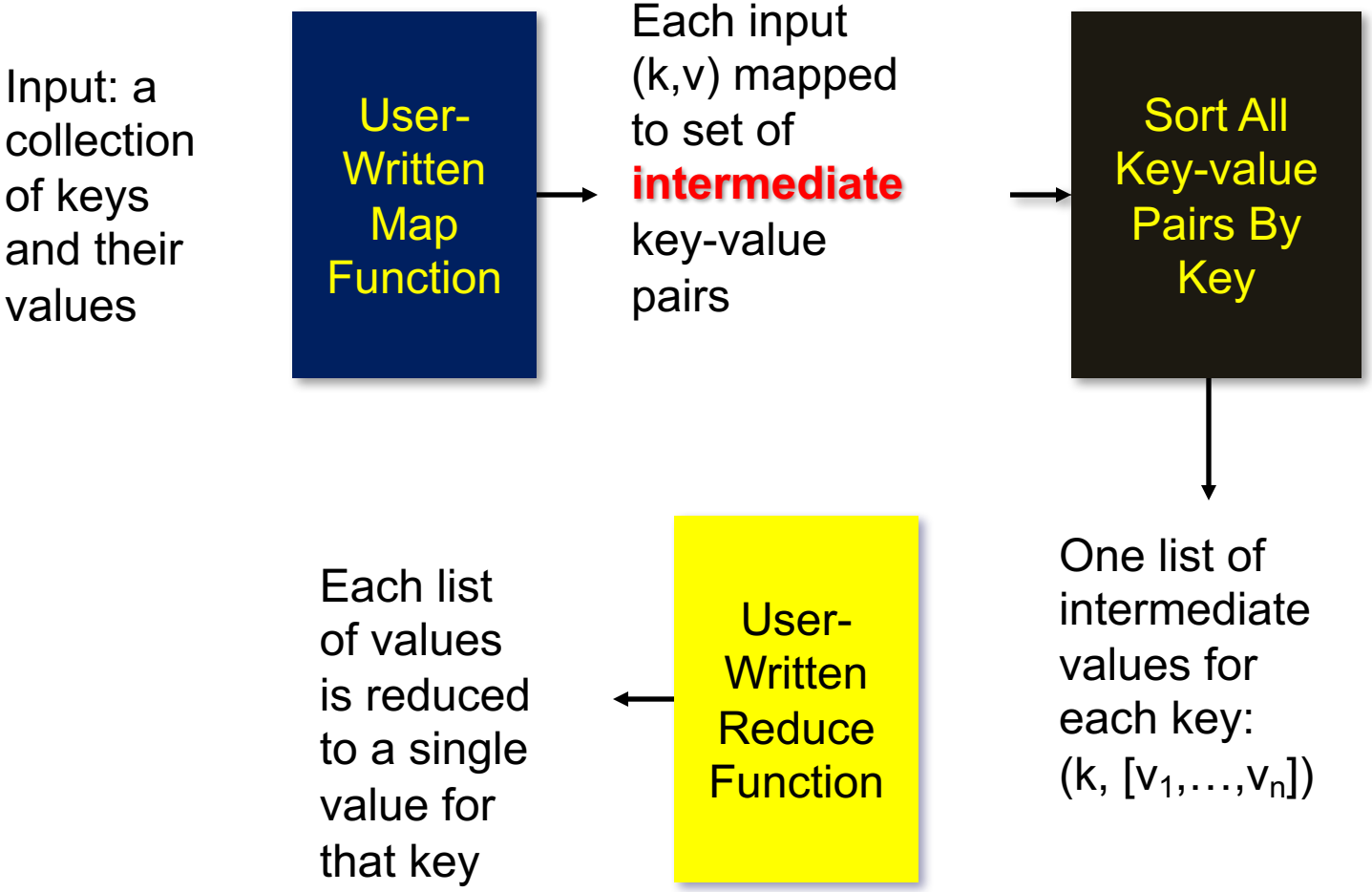
CODE EXPLAIN – MAIN

```
  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);
  }
}
```

EXAMPLE – EXECUTION FLOW

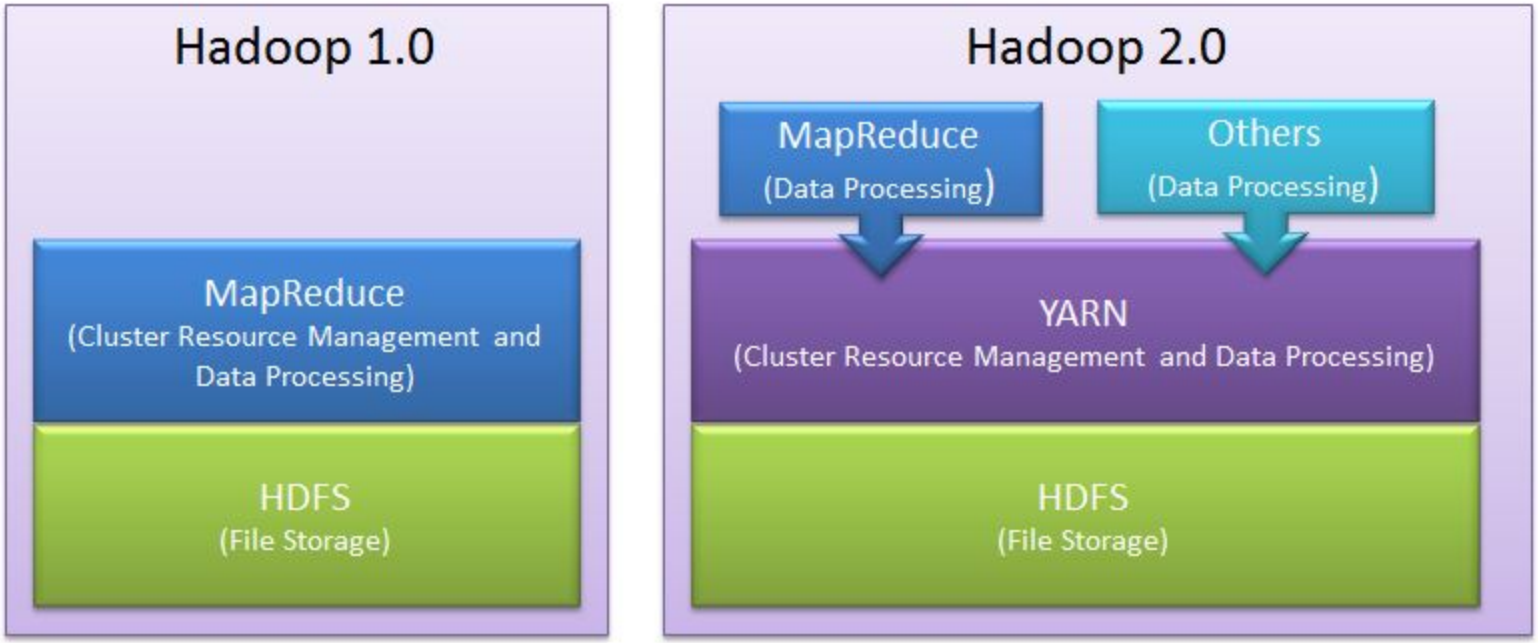


MAP-REDUCE FRAMEWORK FOR WORD COUNT

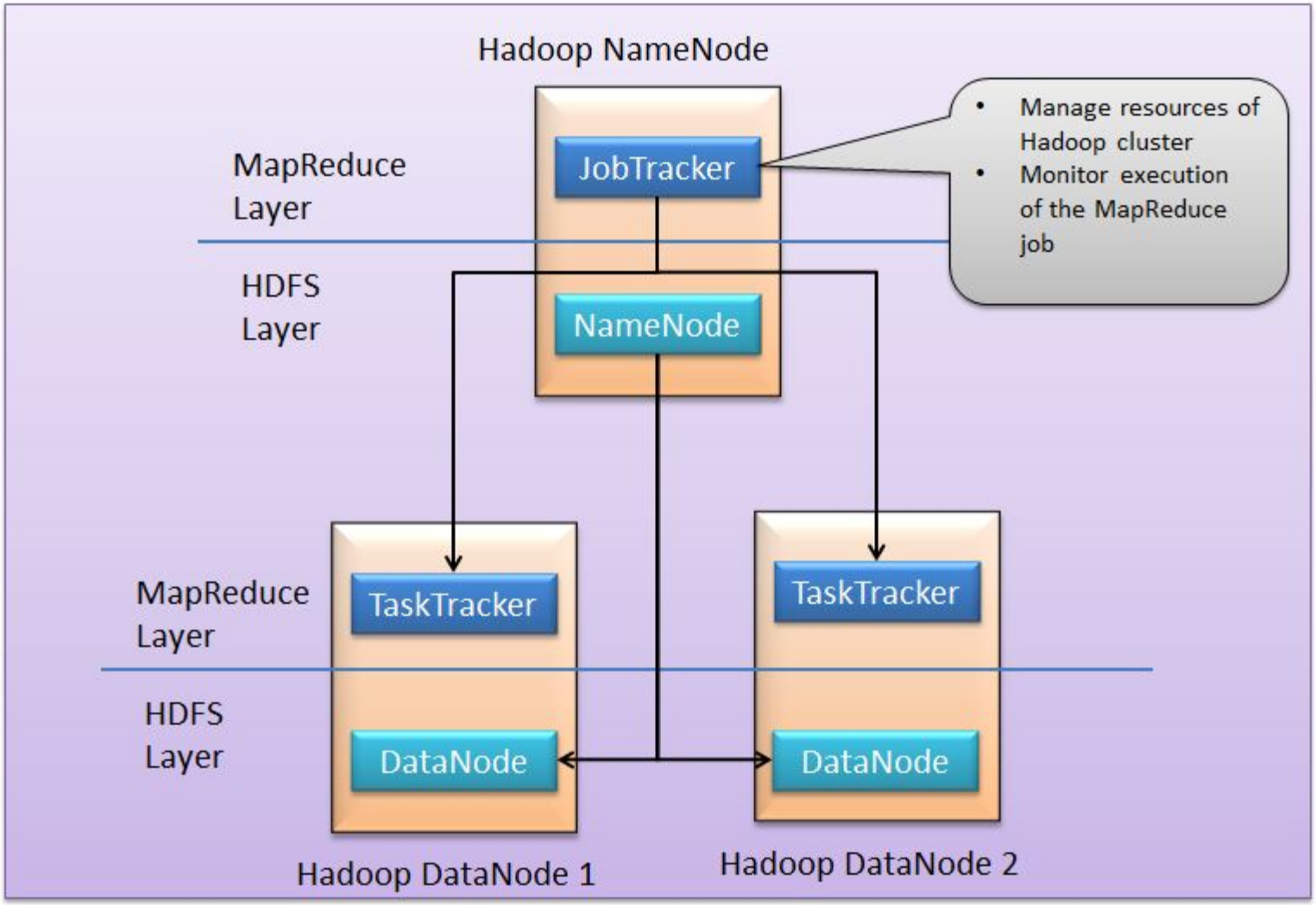


HADOOP RESOURCE MANAGEMENT

HADOOP RESOURCE MANAGEMENT



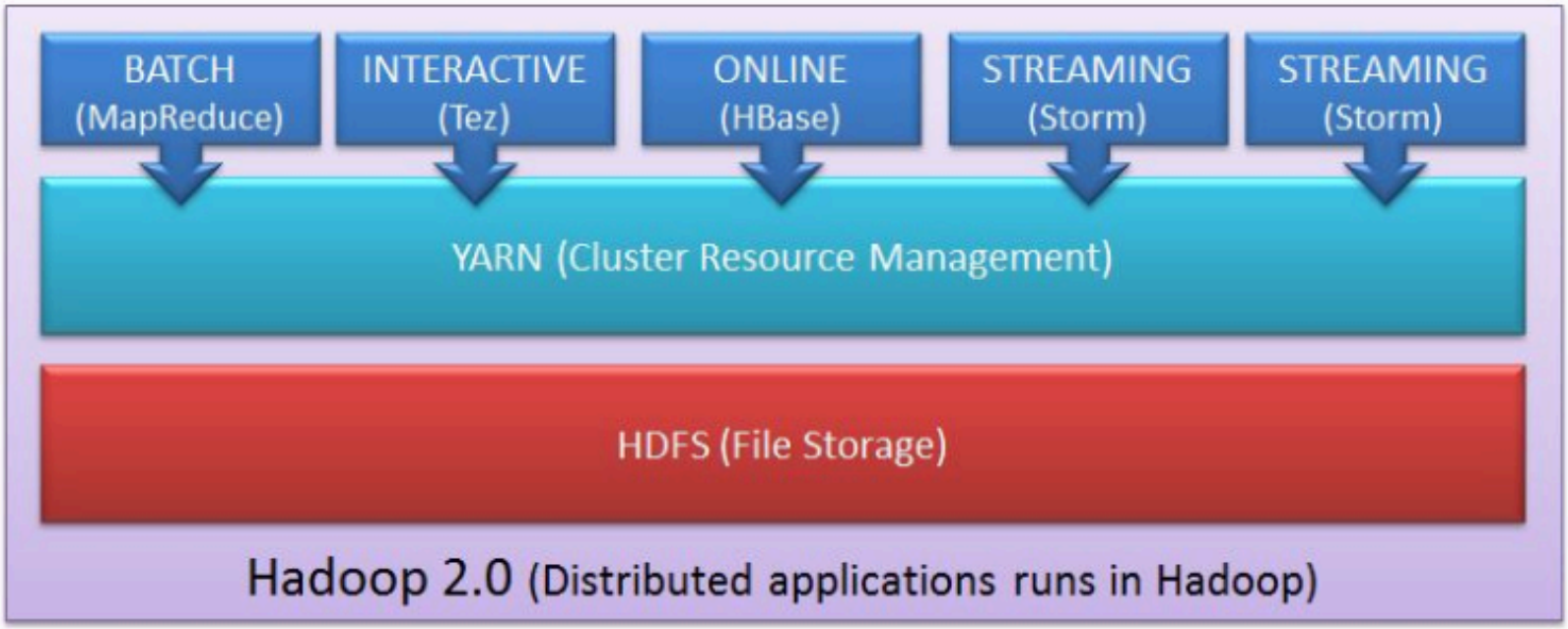
HADOOP 1.0


Hadoop NameNode
MapReduce Layer
HDFS Layer
JobTracker
NameNode
Manage resources of Hadoop cluster
Monitor execution of the MapReduce job
MapReduce Layer
HDFS Layer
TaskTracker
DataNode
TaskTracker
DataNode
Hadoop DataNode 1
Hadoop DataNode 2

HADOOP 1.0 PROBLEMS

- Scalabilty
  - JobTracker runs on a single machine and doing so many tasks
    - Resource management
    - Job and task scheduling and
    - Monitoring
- Single point of failure
- Resource utilization
  - Fixed map and reduce slots
- Limit only for MapReduce applications

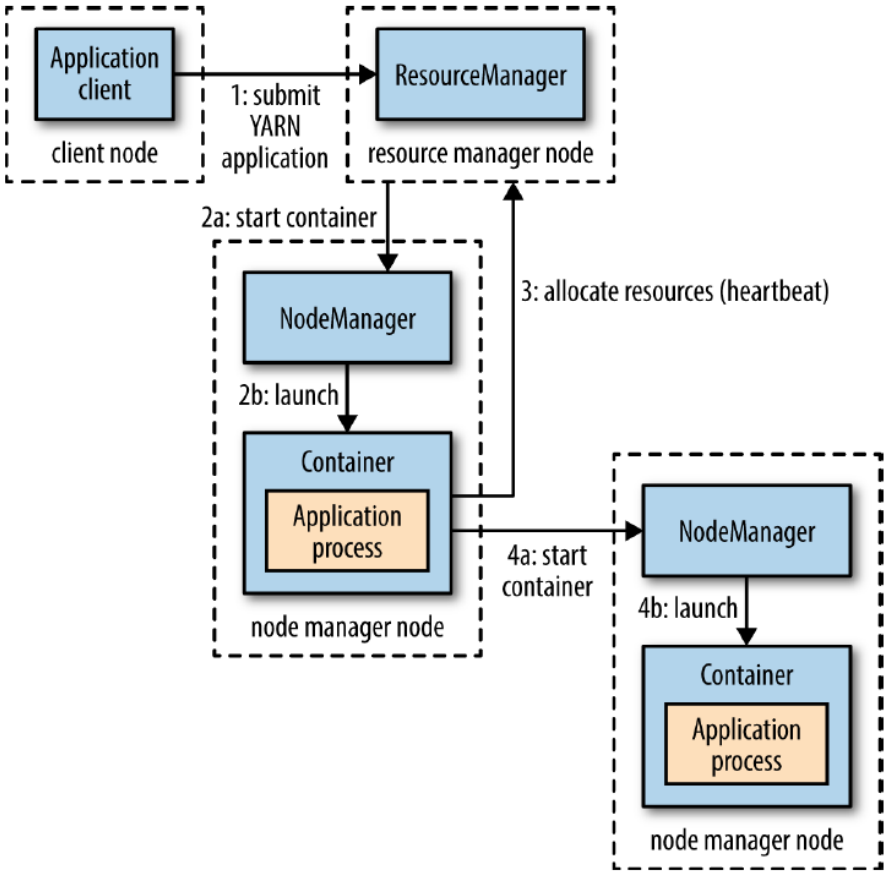
HADOOP 2.0 WITH YARN



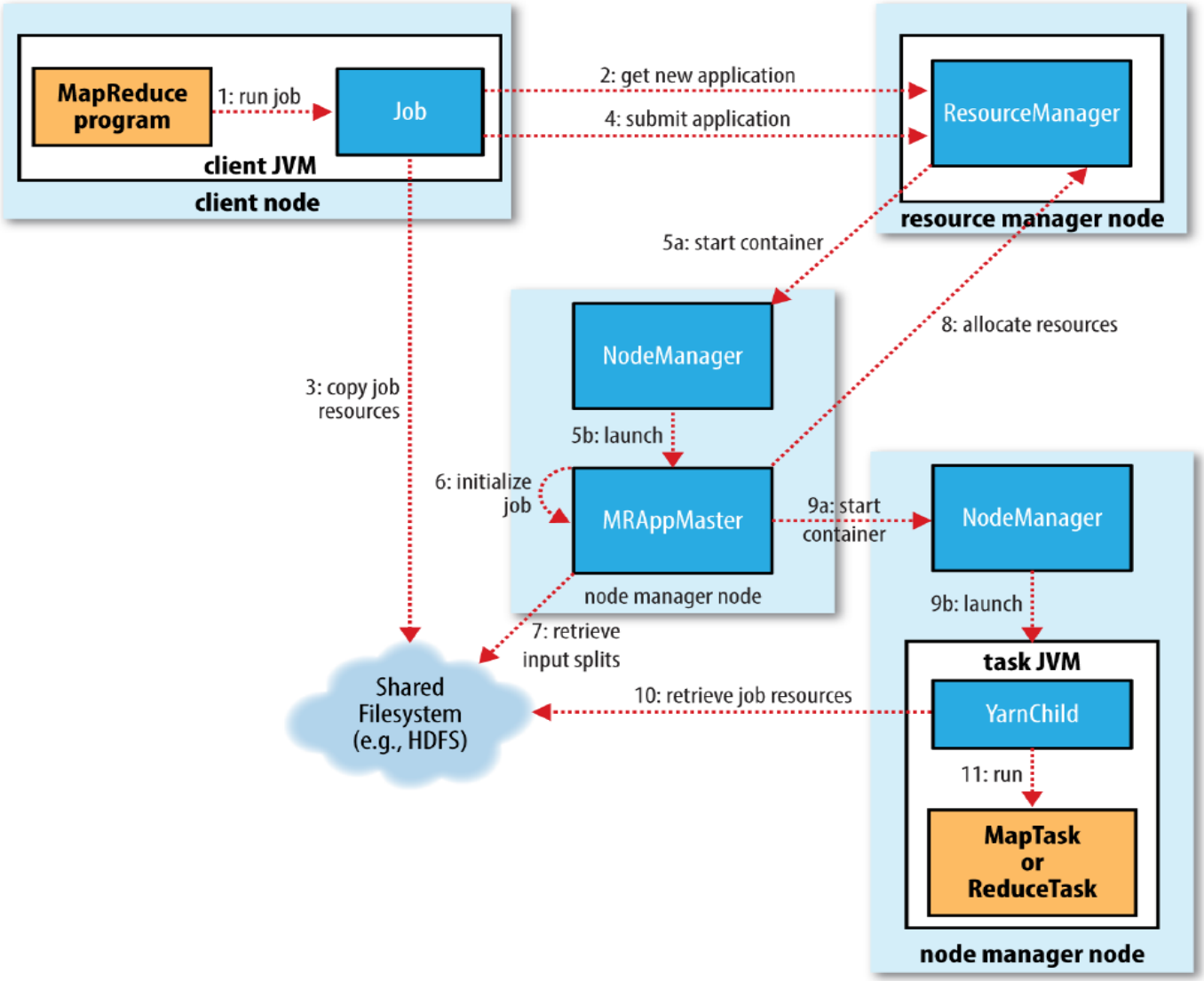
- Flexible, can run multiple types of applications
- No fixed slot limit specific for map or reduce tasks

YARN (HADOOP 2.0)

- Provide resource management via two types of long-running daemon
  - Resource Manager – one per cluster
    - Manage resource across the cluster
  - Node Managers – one per node
    - Launch and monitor job *containers*



YARN – MAPREDUCE JOB



INPUT DATA
LOCALITY AND SIZE OPTIMIZATION

INPUT DATA – SPLIT SIZE

- Fine-grained splits increase parallelism and improves fault-tolerance
  - Small splits reduce the processing time of each split and allows faster machines to process proportionally more splits over the course of the job than slower machines
  - Load-balancing can be achieved more efficiently with small splits
  - The impact of failure, when combined with load-balancing, can be reduced significantly with fine-grained splits

INPUT DIVISION – SPLIT SIZE

- Fine-grained splits increase parallelism and improves fault-tolerance
  - Small splits reduce the processing time of each split and allows faster machines to process proportionally more splits over the course of the job than slower machines
  - Load-balancing can be achieved more efficiently with small splits
  - The impact of failure, when combined with load-balancing, can be reduced significantly with fine-grained splits
- Too small splits increases the overhead of managing splits
  - Map task creation dominates the total job execution time.

DATA LOCALITY – INPUT DATA

- Data locality Optimization
  - The map task should be run on a node where the input data resides
  - The **optimal split size** is the same as the largest size of input that can be guaranteed to be stored on a **single** node.
    - What if the split size is larger than what one node can store?

DATA LOCALITY – MAP OUTPUT

- Output produced by map tasks should be stored locally, not a the distributed storage

- Map output should not be replicated to overcome failure

DATA LOCALITY – REDUCE TASKS

- Reduce tasks cannot typically take advantage of data locality

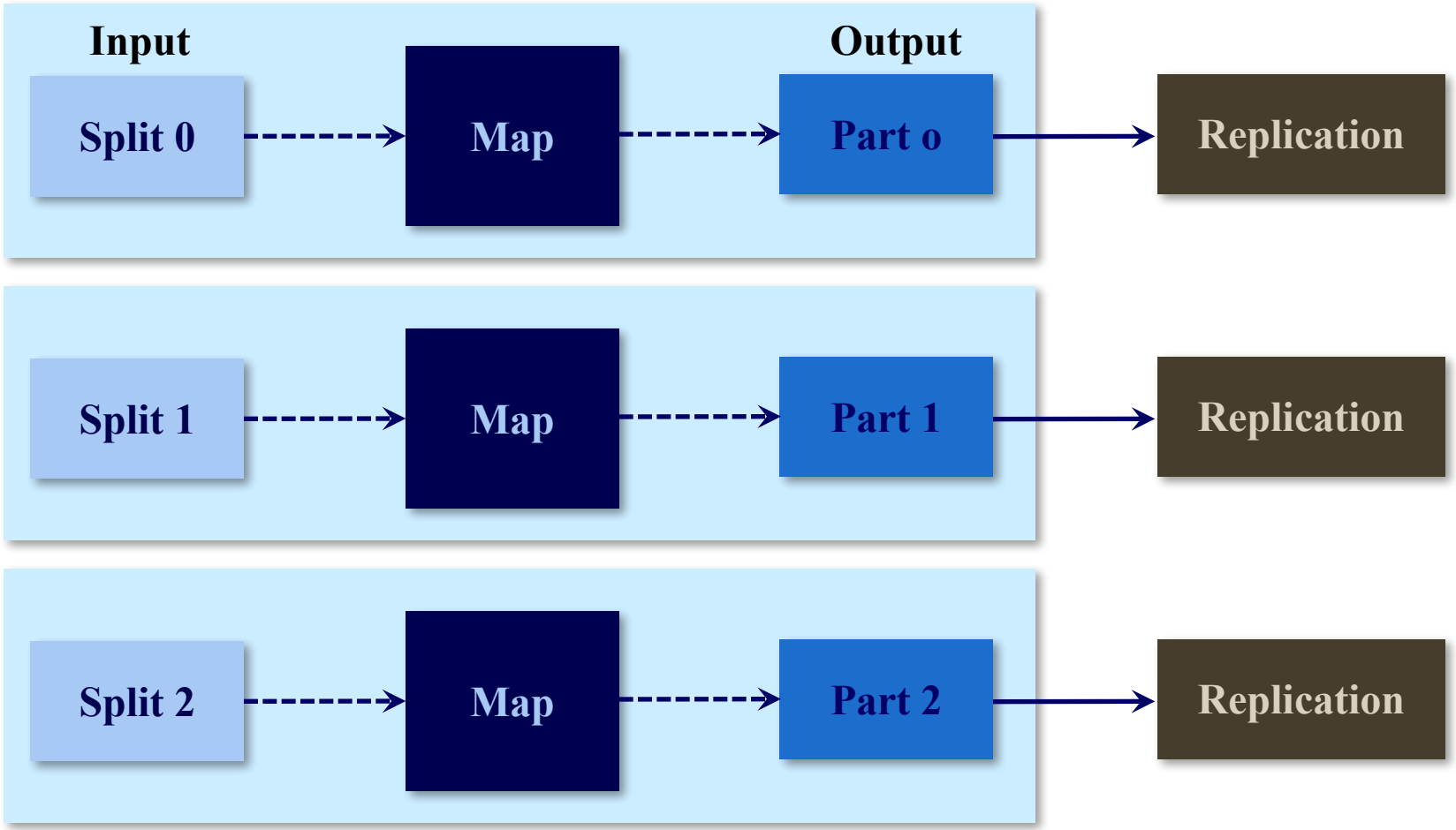
- The sorted map outputs have to be transferred across the network to the node where the reduce task is running

PARTITIONERS AND COMBINERS

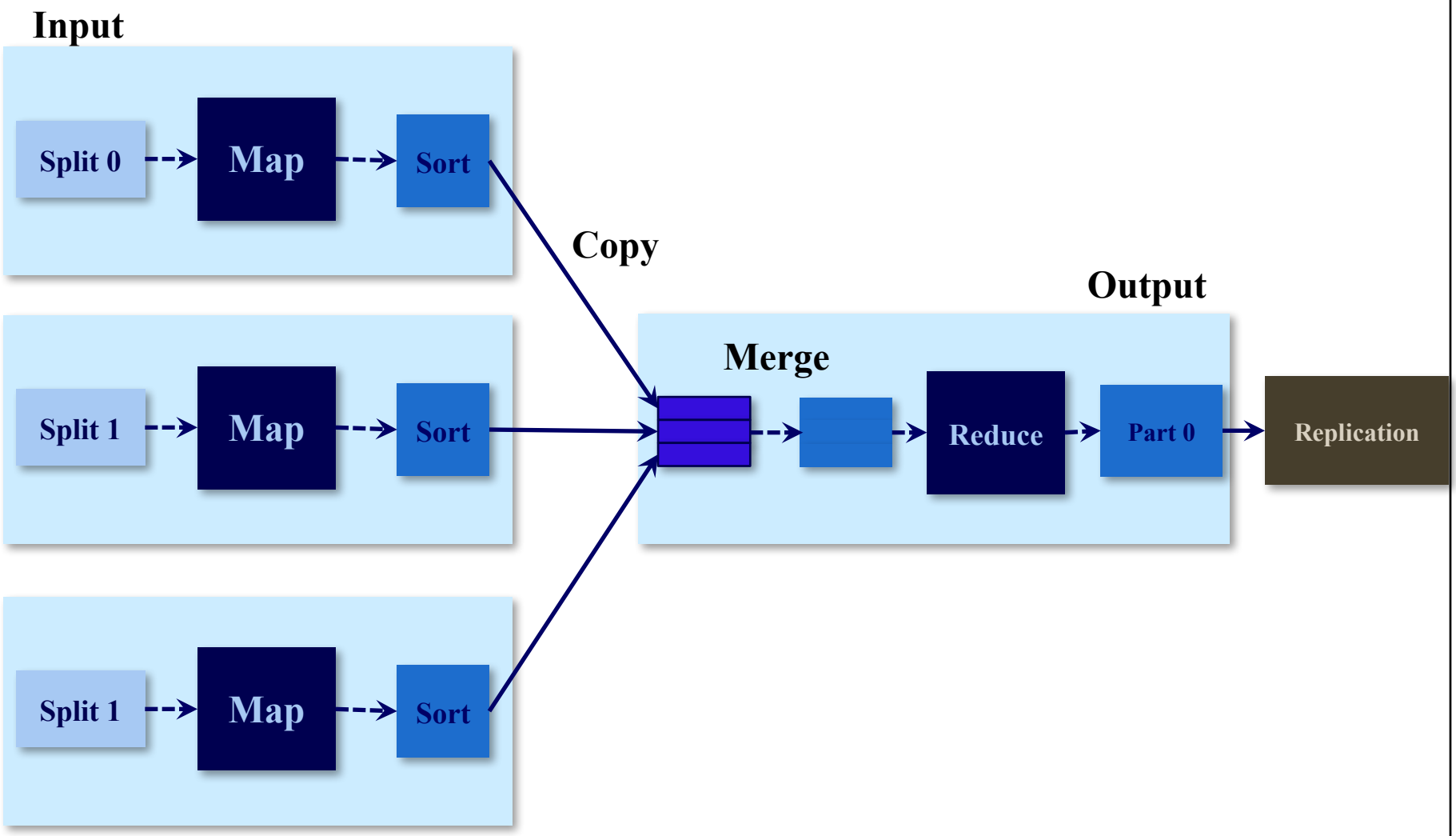
NUMBER OF REDUCE TASKS

- The number of map reducers is typically specified independently
  - It is not only governed by the size of the input, but also the type of the application
- MapReduce data flow can be specified in different ways
  - No reduce tasks
  - Singe reduce task
  - Multiple reduce tasks

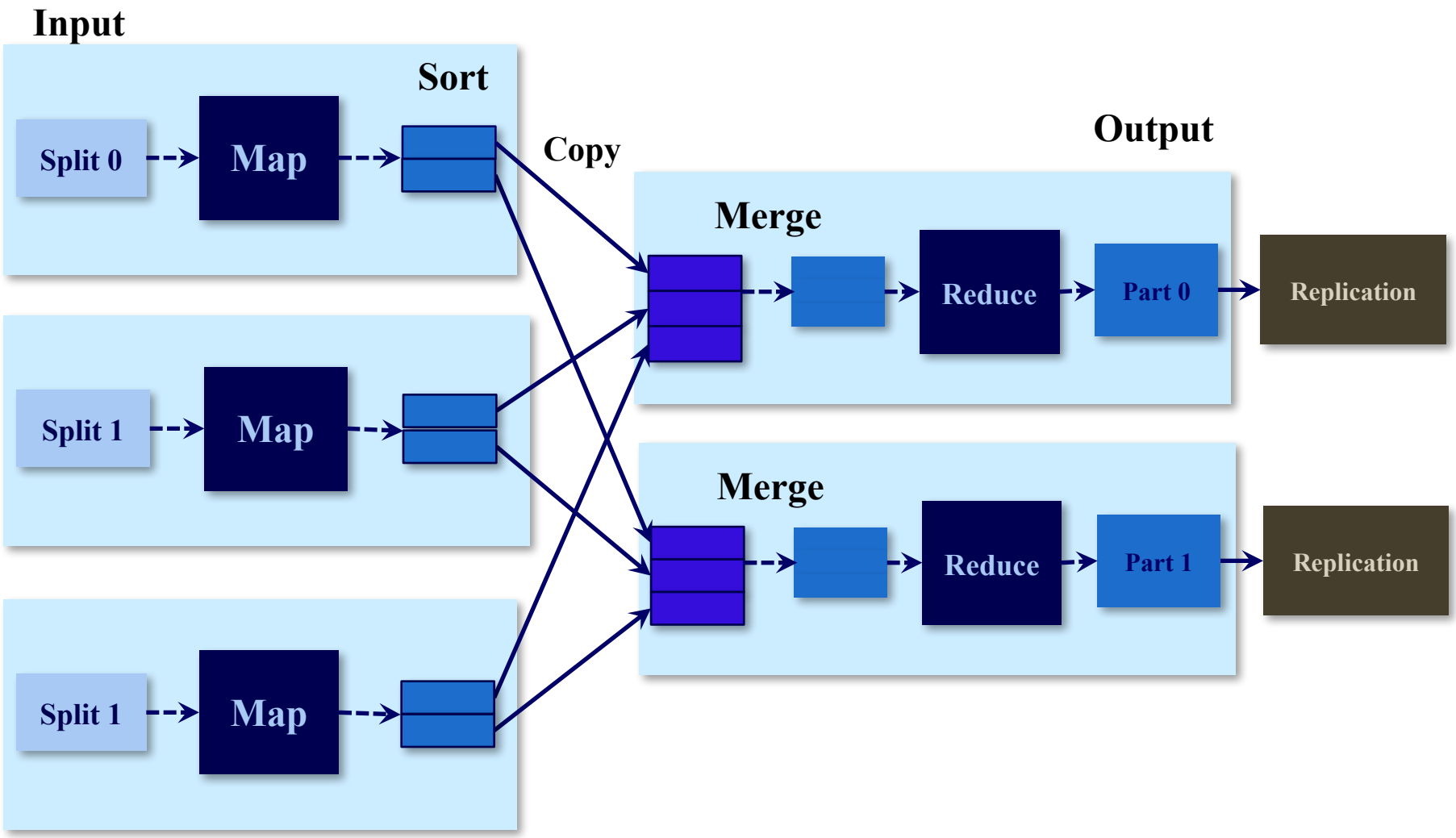
DATA FLOW – NO REDUCE TASKS



DATA FLOW – SINGLE REDUCE TASK



DATA FLOW – MULTIPLE REDUCE TASKS



DATA FLOW – MULTIPLE REDUCE TASKS

- Map tasks partition their output, each creating one partition for each reduce task
  - There can be many keys and their associate values in each partition
    - The recodes for every key are all in a single partition
  - The partition can be controlled by user-defined partition function
    - The use of a hash function typically works well
- The "Shuffle", data flow between map and reduce, is a complicated process whose tuning can have a big impact on the job execution

MAPREDUCE – COMBINERS

- Combiner functions can be used to minimize the data transferred between map and reduce tasks
- Combiner functions run on the map output
- Since it is an optimization function, there is no guarantee how many times combiners are called for a particular map output record, if at all

COMBINER EXAMPLE

- Assume that the mappers produce the following output
  - Mapper 1 – (1950, 0), (1950, 20), (1950, 10)
  - Mapper 2 – (1950, 25), (1950, 15)
- Reduce function is called with a list of all the values
  - (1950, [0, 20, 10, 25, 15]) with output (1950, 25)
- Using a combiner for each map output results in:
  - Combiner 1 – (1950, 20)
  - Combiner 2 – (1950, 25)
- Reduce function is called with (1950, [20,25]) with output (1950, 25)

COMBINER PROPERTY

- The combiner function calls can be expressed as follows:
  - Max(0, 20, 10, 25, 15) = Max(Max(0, 20, 10), Max(25, 15)) = Max(20, 25)=25
  - Max() is commonly referred to as **distributive**
- Not all function exhibit distributive property
  - Mean(0, 20, 10, 25, 15) = (0+20+10+25+15)/5=14
  - Mean(Mean(0, 20, 10), Mean(25, 15)) = Mean (10, 20) = 15
- Combiners do not replace producers
  - Producers are still needed to process recorders with the same key from different maps

REFERENCE

- Hadoop The Definitive Guide 4th Edition, Tom White,
- Data-Intensive Text Processing with MapReduce, Jimmy Lin and Chris Dyer.
- MapReduce: Simplified Data Processing on Large Clusters, Jeffrey Dean and Sanjay Ghemawat,
- The Google File System, Sanjay Ghemawat, Howard Gobioff, and Shun-TakLeung,
- MapReduce Tutorial