
# Big Data

## Map Reduce Programming model & Architecture

**Dr. H.L. Phalachandra**

Department of Computer Science and Engineering

**phalachandra@pes.edu**

Leveraging Slides of Dr. K.V. Subramaniam


## What we have learnt so far..

- Large amounts of data to be processed.
- We have HDFS as a distributed store.
- We need to distribute the processing also.

## What is Map Reduce ?

# BIG DATA

## Map Reduce Programming model and Architecture

### Why do we do Map-Reduce ?

- It's the processing engine Apache Hadoop was originally written in
- It's the processing engine which can process extremely large data
- We are going to
  - Study Map-Reduce paradigm or the programming model for Map-Reduce
  - Study Hadoop architecture or how Map-Reduce works internally
    - Open Source implementation of Map-Reduce

# BIG DATA

## What is MapReduce ?


- Origin from Google, [OSDI'04] & built on princ
- Hadoop is the adoption of open-source implementation by Yahoo (now Apache project)
- MapReduce is the processing component of Hadoop with a programming model and processes massive amount of data with advantages of
  - *Parallelly* processing the data in a distributed computational environment and thus *increases processing performance*
  - *Data Locality –ability to process data where it, is by moving processing to data location rather than moving data which is more expensive*
- Its an execution framework for large-scale data processing
  - Which distributes the computing across distributed commodity servers and then pulls the results together
  - Programmer writes code as if writing for a single machine but the framework executes the process in distributed manner
  - Distributed implementation that hides all the messy details
    - Fault tolerance (thus provides HA)
    - I/O scheduling
    - parallelization

# BIG DATA

## Map Reduce - Motivation

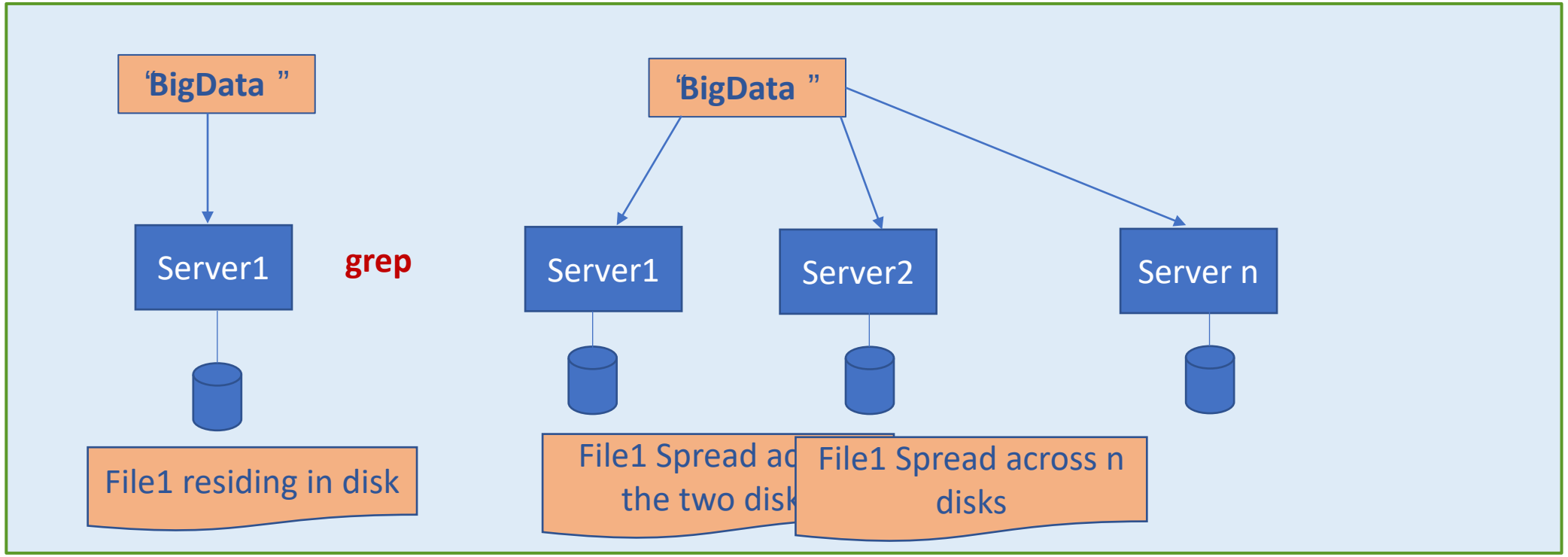

- Lots of demands for very large scale data processing leading to the need for Scaling
- Scale out not up - Symmetric multi-processing machines (SMP) with large number of processor sockets (100s), large shared memory (GBs) are significantly more expensive as compared to Cluster of low end machines which are 4x more cost effective but lots of machines needed
- Build levels of abstraction which supports beneficial division of labor
  - A simple Programming model/Interface : A map + reduce
    (separating the abstraction of ***what*** from ***how*** of data intensive processing)
  - An application programmer develops using well defined interfaces and passes on the data **(what)** and hence is isolated from system-level details like locking, starvation, etc.
  - The framework (Map-Reduce) which is designed once and verified for correctness, takes care of the **How**. This includes the system-level details of writing distributed programs with details of threads, processes, machines Code that runs concurrently and needing to factor in Deadlocks, race conditions, etc.

# Big Data: Illustrative Example leading to Map Reduce

# BIG DATA

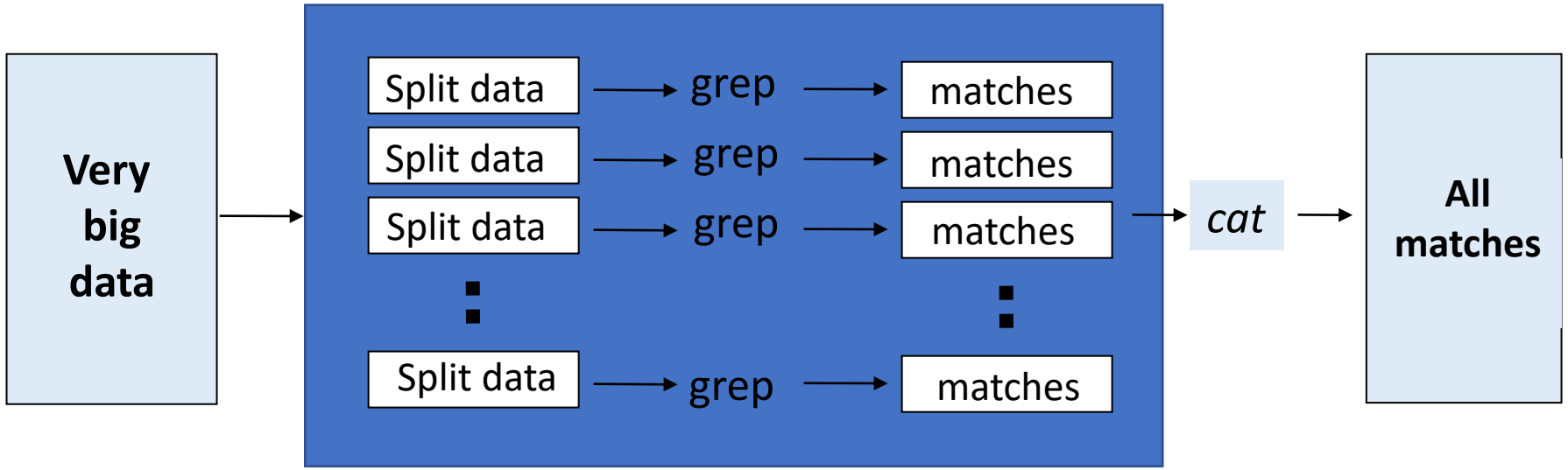
## A MapReduce Example

- Consider a very large text file and you want to determine if a word exists in this file. BigData? EngFile1
  - Say if the file is stored in HDFS across two machines.
  - How will you search if BigData is present in the file?



# BIG DATA

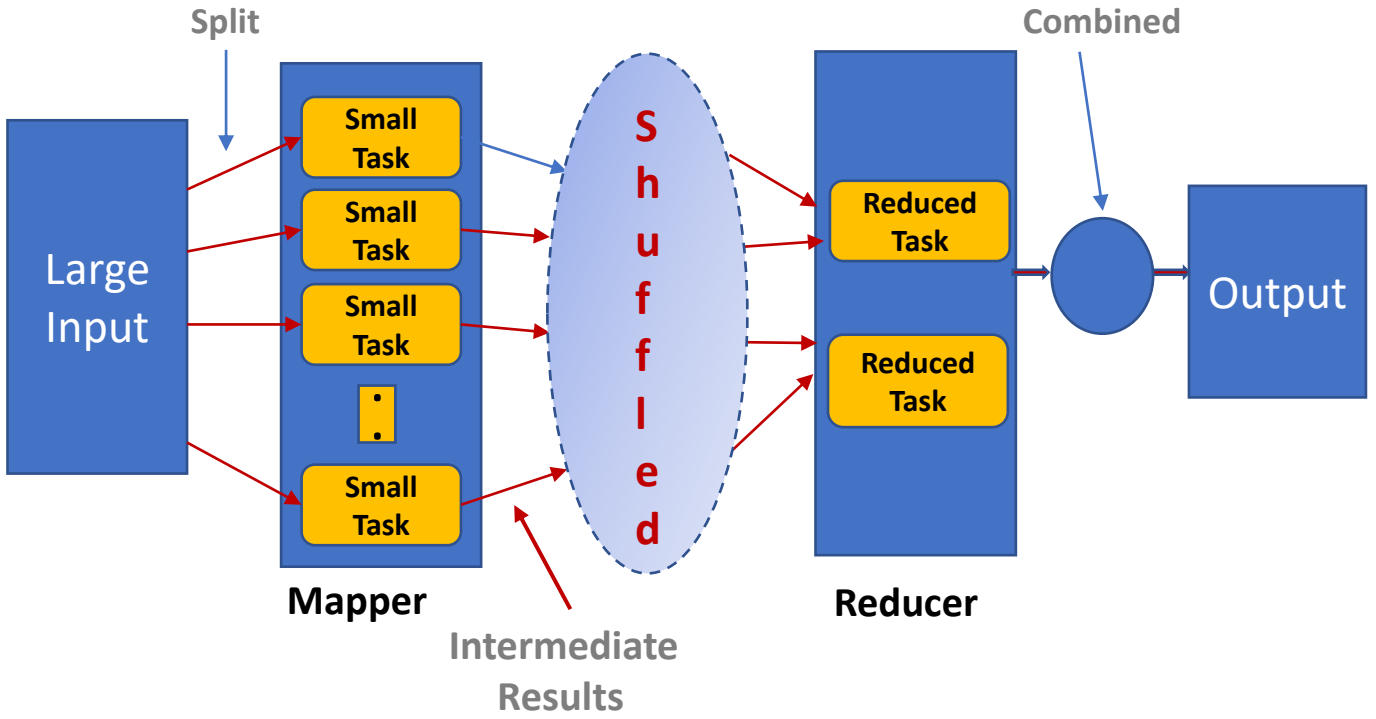
## Map Reduce: Distributed Grep-Solution



# BIG DATA

## Map Reduce - Insight

- Uses Divide and conquer –Partitions large problem into smaller subproblems
- Workers work on sub-problems in parallel (could be threads in a core or cores in multi-core processor, multiple processor in a machine, machines in a cluster) and produce intermediate results
- Intermediate results from workers are combined to form the final result
- Basic operations on the input
  - Map
  - Reduce

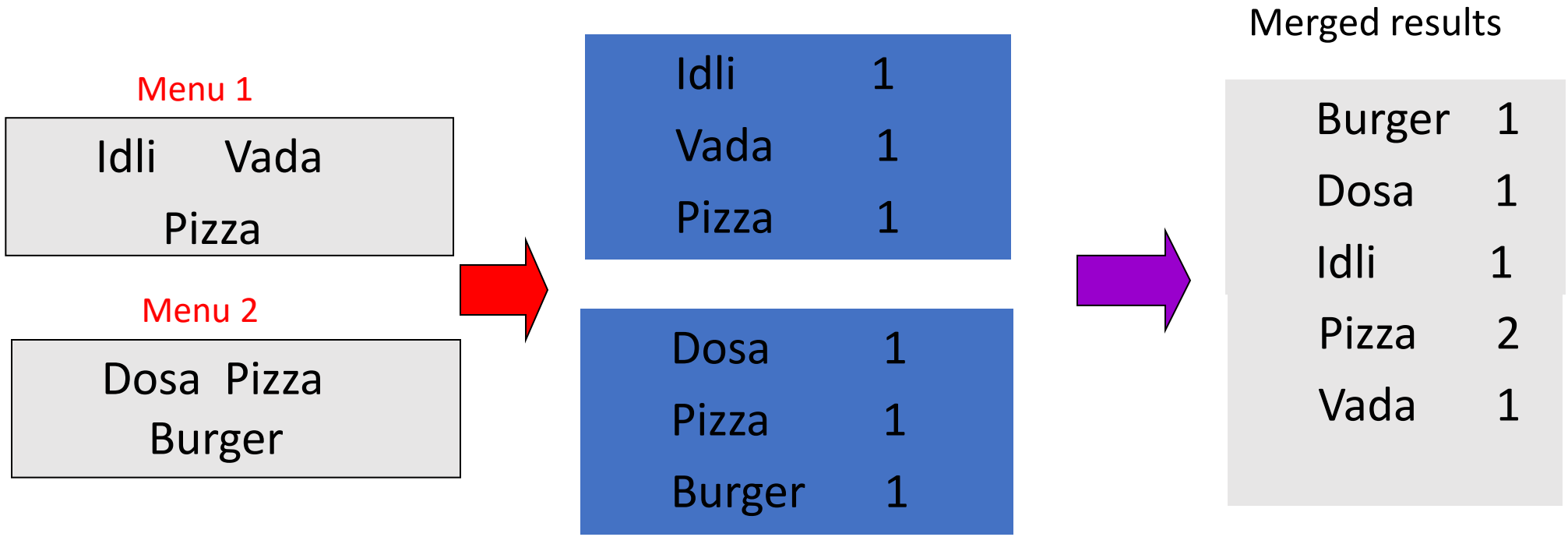



- To understand what Map/Reduce really are..

# Lets consider a Distributed Word Count

# BIG DATA

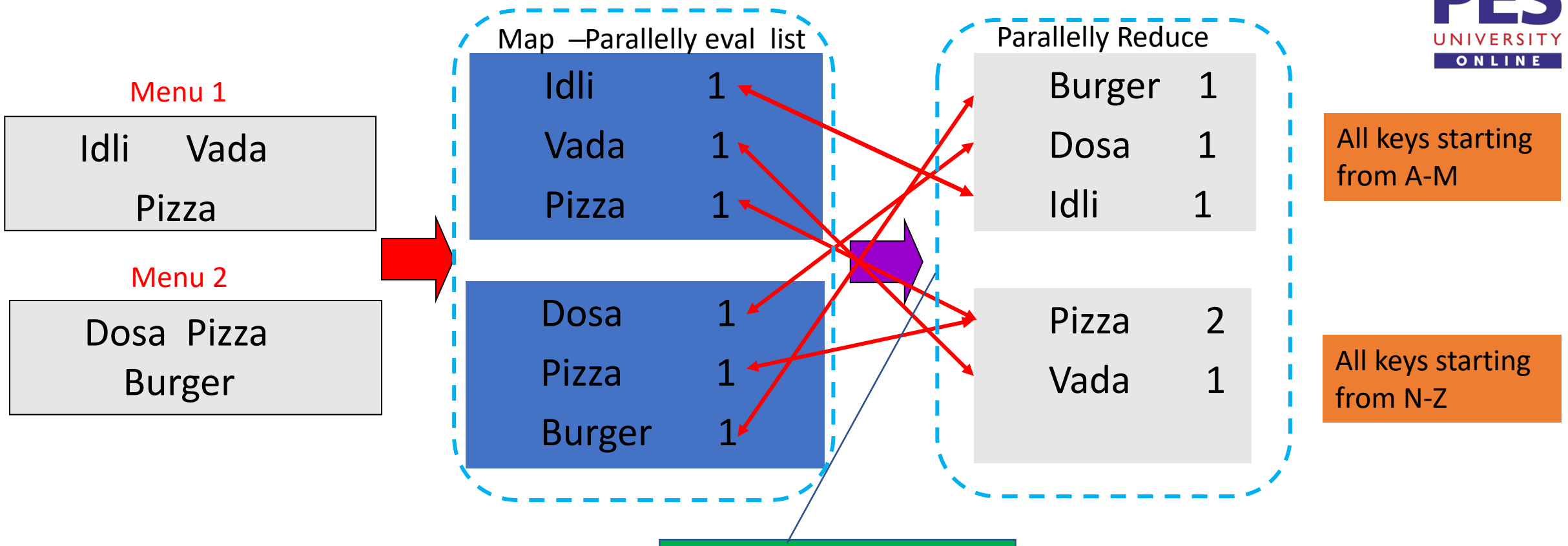
## Example: find the number of restaurants offering each item?




- Suppose we need parallelism of the merge program.
- Would the earlier approach work?
- What do we need to do?

# BIG DATA

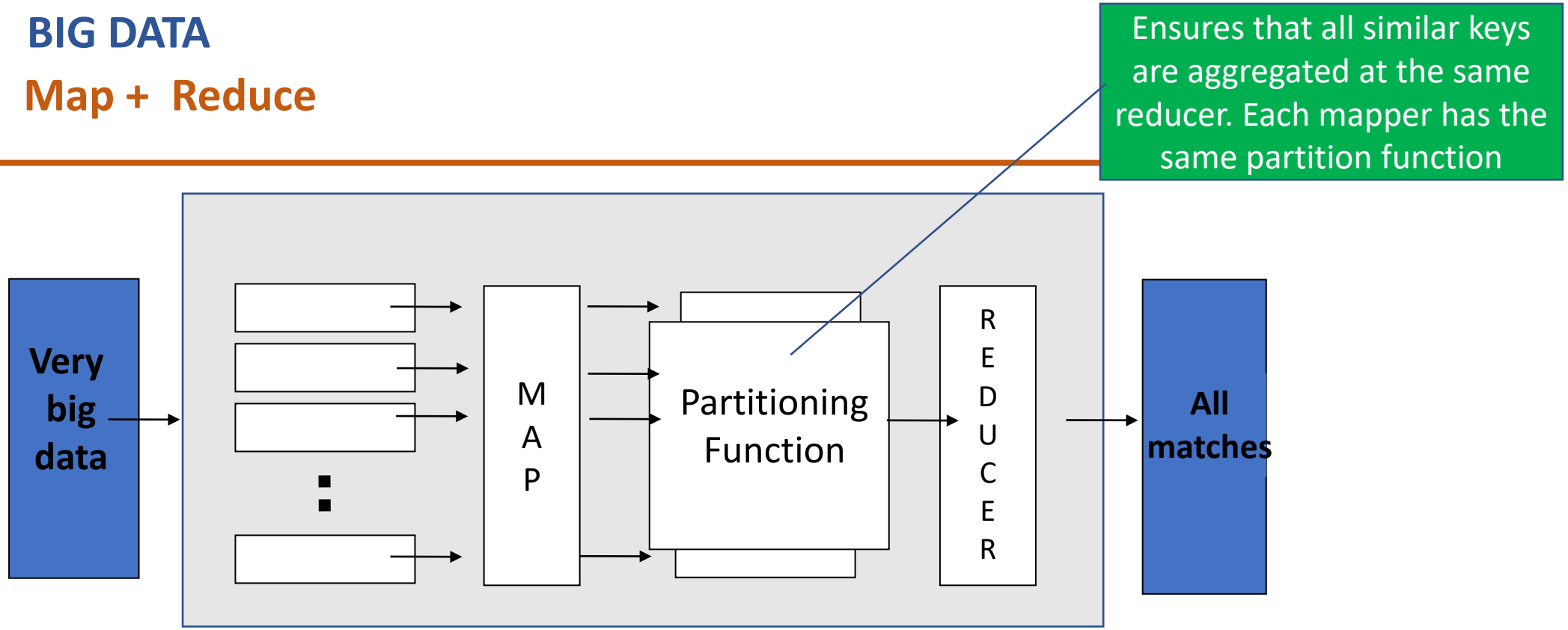
## Distributed Word Count




- Treat output of first program
- While partitioning the keys in

# BIG DATA

## Map + Reduce




- Map:
  - Accepts *input* key/value pair
  - Emits *intermediate* key/value pair

- Reduce :
  - Accepts *intermediate* key/value* pair
  - Emits *output* key/value pair

# Map Reduce: A look at the code

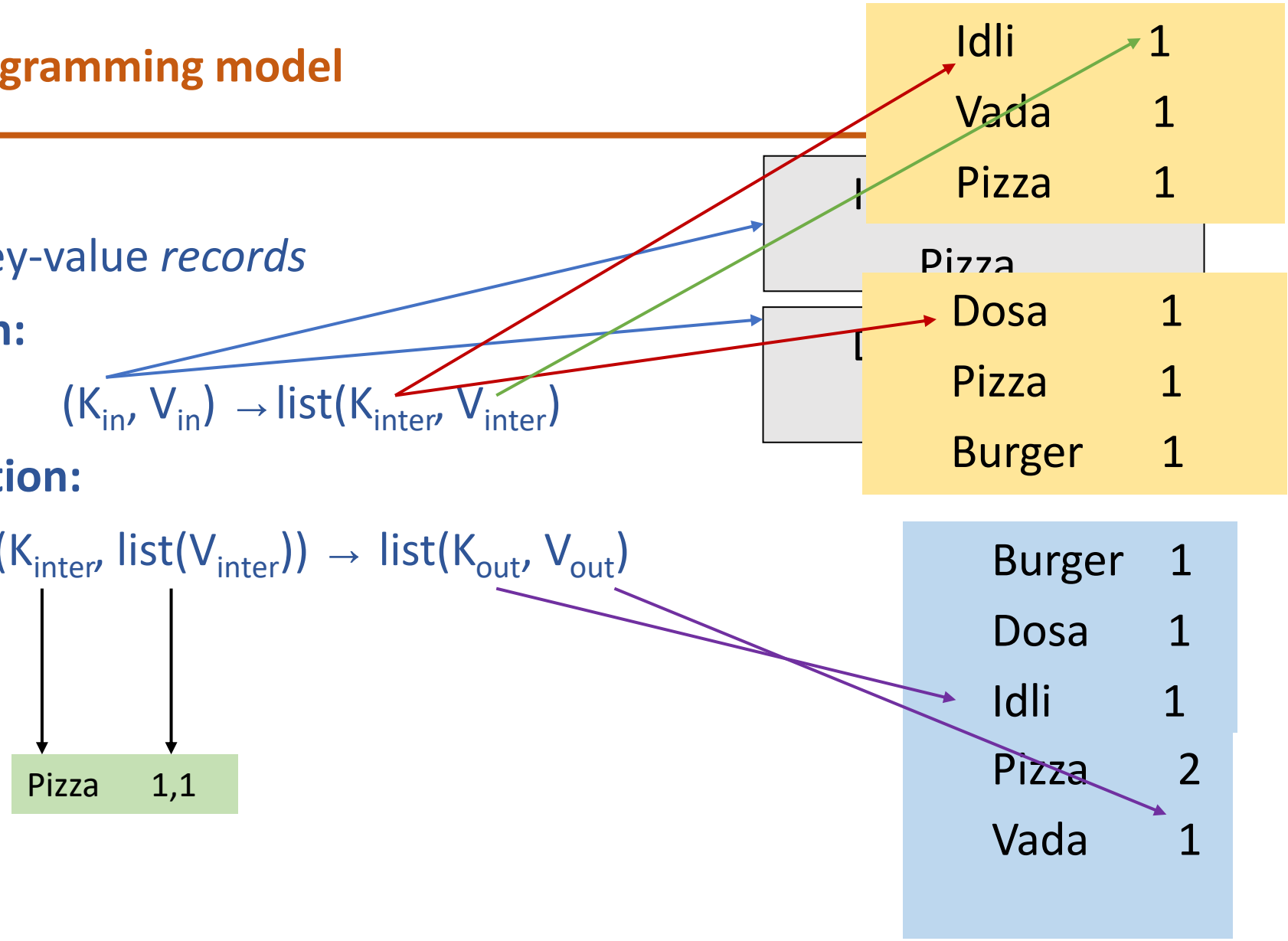
# Map Reduce Programming model

- **Data type:** key-value *records*
- **Map function:**

$$(K_{in}, V_{in}) \rightarrow list(K_{inter}, V_{inter})$$

- **Reduce function:**

$$(K_{inter}, list(V_{inter})) \rightarrow list(K_{out}, V_{out})$$



# Map Reduce- Mapper for Word Count

```
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);
     }
   }
 }
```

(Key , value)

List (Key ,value)

$(K_{in}, V_{in}) \rightarrow list(K_{inter}, V_{inter})$

Hadoop data type

# BIG DATA

## Map Reduce -Reducer for Word Count

```
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);
   }
 }
```

Key ,List (value)

List (Key ,value)

$(K_{inter}, list(V_{inter})) \rightarrow list(K_{out}, V_{out})$

# BIG DATA

## Map Reduce - Main Program for Word Count

```
public static void main(String[] args) throws Exception {
 Configuration conf = new Configuration();
 String[] otherArgs = new GenericOptionsParser(conf, args).
                           getRemainingArgs();
 if (otherArgs.length < 2) {
   System.err.println("Usage: wordcount <in> [<in>...] <out>");
   System.exit(2);
 }
 Job job = new Job(conf, "word count");
 job.setJarByClass(WordCount.class);
 job.setMapperClass(TokenizerMapper.class);

 job.setReducerClass(IntSumReducer.class);
 job.setOutputKeyClass(Text.class);
 job.setOutputValueClass(IntWritable.class);
 for (int i = 0; i < otherArgs.length - 1; ++i) {
   FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
 }
 FileOutputFormat.setOutputPath(job,
   new Path(otherArgs[otherArgs.length - 1]));
 System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

**Set Mapper and Reducer class**

- This does not state the machine where this needs to be run in
- This also does not state how the input needs to be distributed across mappers

# Map Reduce: Sample Exercise

**Context : Search**

- We have a set of files each representing a web site
- Every file contains a line number and information as records.
- We are looking or searching for a pattern in the file
- Output should be all the line numbers which has the pattern

**Input:** file(lineNumber, line) records and pattern

**Output:** lines matching a given pattern

**What will be the mapper and reducer? What will be the keys?**

**Map:**

**Reduce:**

## Map Reduce , Search - Solution

**Input:** file (lineNumber, line) records and pattern

**Output:** lines matching a given pattern

**Map:**

```
if(line matches pattern):
        output(linenumber)
```

**Reduce:** identity function

- Alternative: no reducer (map-only job)

## Map Reduce , Functions in the Model

- **Map**
  - Process a key/value pair to generate intermediate key/value pairs
  - Sorts all key/value pairs before sending to reducer
- **Reduce**
  - Merge all intermediate values associated with the same key
  - Runs after all Map tasks are finished (why?)
- **Partition**
  - By default : **`hash(key) mod R`** （Well balanced)
  - There are cases where this can be more complex

# Map Reduce: Revision exercise

## Map Reduce, Sort

**Input:** (key, value) records

**Output:** same records, sorted by key

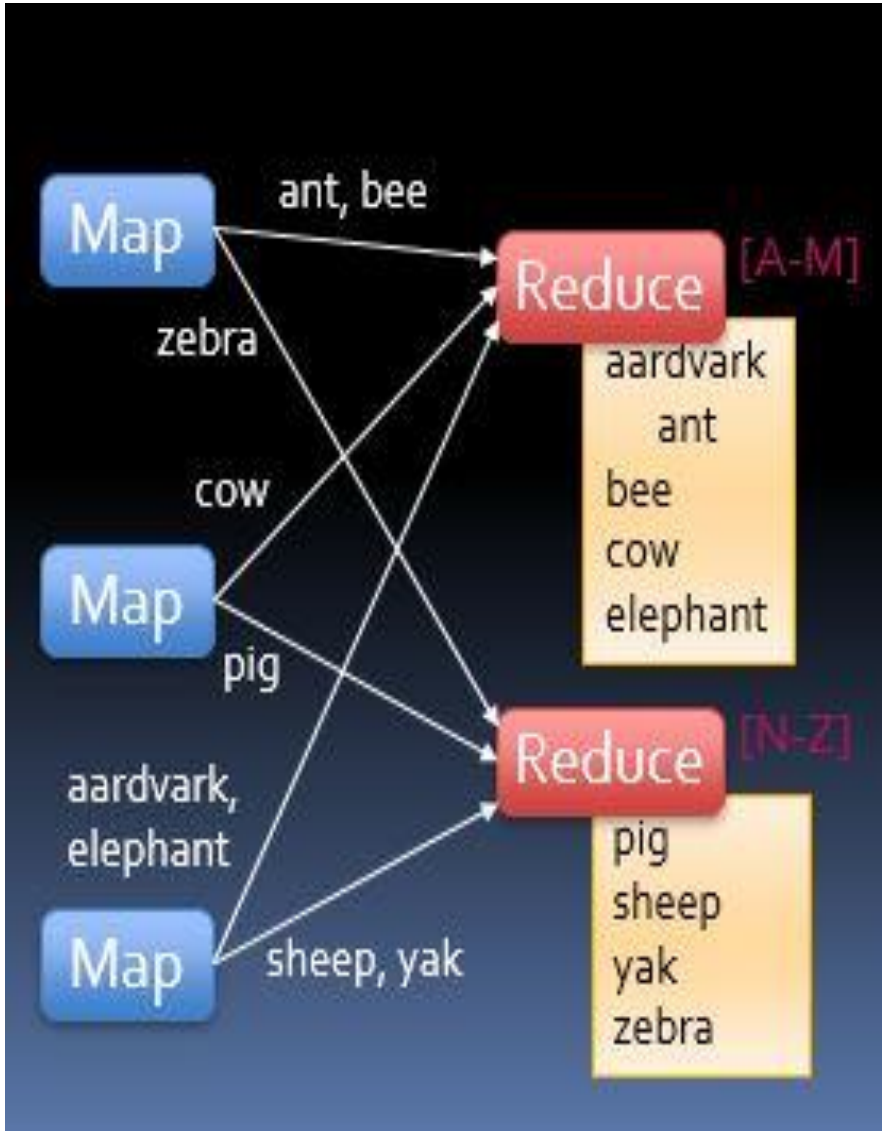
**What will be the mapper and reducer?**
**What will be the partition function?**

**Map:**

**Reduce:**

# Map Reduce, Sort - Solution

- **Input**: (key, value) records
- **Output**: same records, sorted by key
- **Map**: identity function
- **Reduce**: identity function
- **Trick**: Pick partitioning function *p* so that
  $k_1 < k_2 \Rightarrow p(k_1) < p(k_2)$
- Works because map sorts output keys.

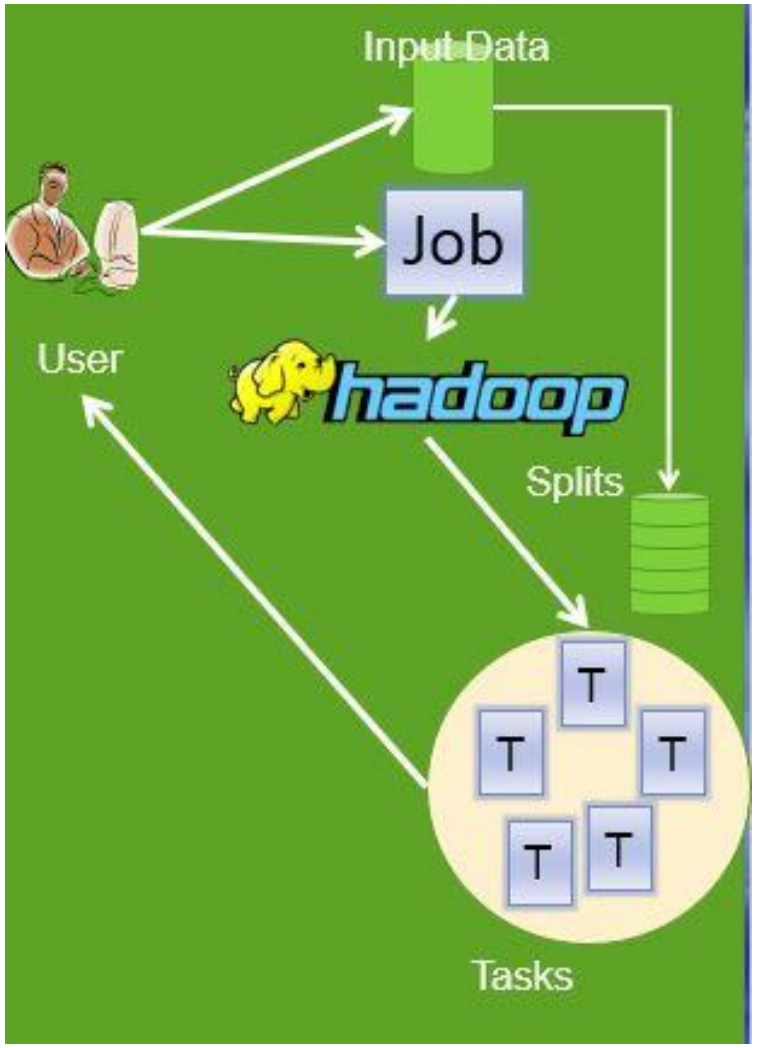


# Map Reduce: Job Submission Flow

# BIG DATA

## Map Reduce , Hadoop Flow.

- User submits job
  - input data, MapReduce program, and configuration information
- How is parallelization achieved?
  - Divide input into smaller chunks –called **input splits**
- Hadoop divides jobs into tasks
  - Map tasks, reduce tasks
  - One map task per split
  - Tasks run in parallel



# BIG DATA

## MapReduce Flow for A SINGLE REDUCE Task

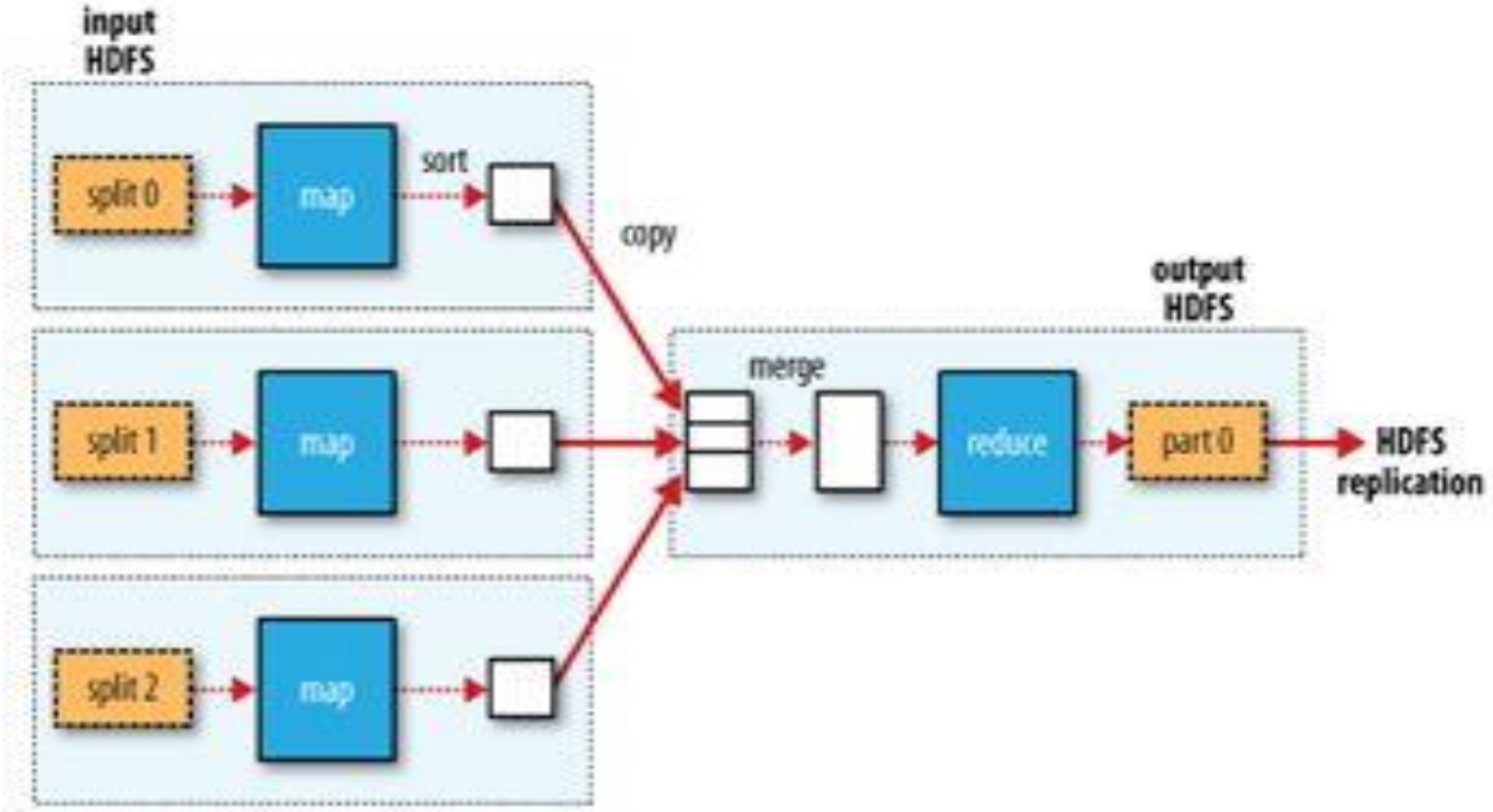



Figure: Map reduce data flow for single reducer task.

# Map Reduce:
# Exercise : Job Submission Flow with two Reducers

## MapReduce Flow

- How will it work when there are two reducers?
- Where will the outputs be?

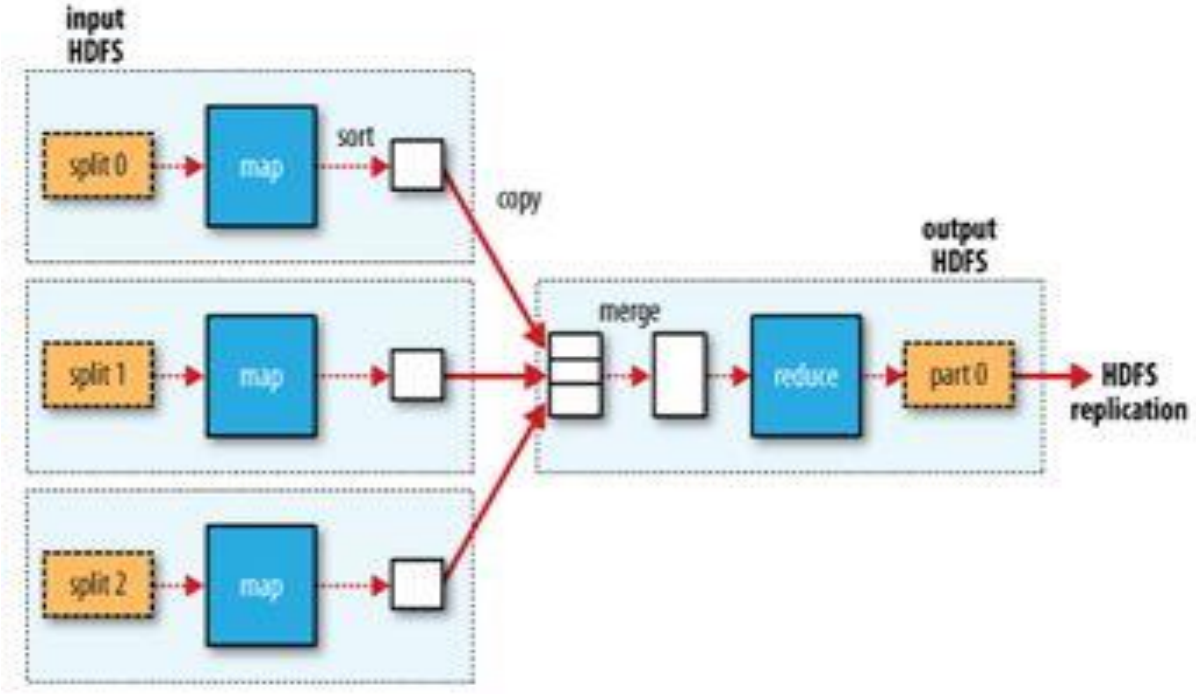
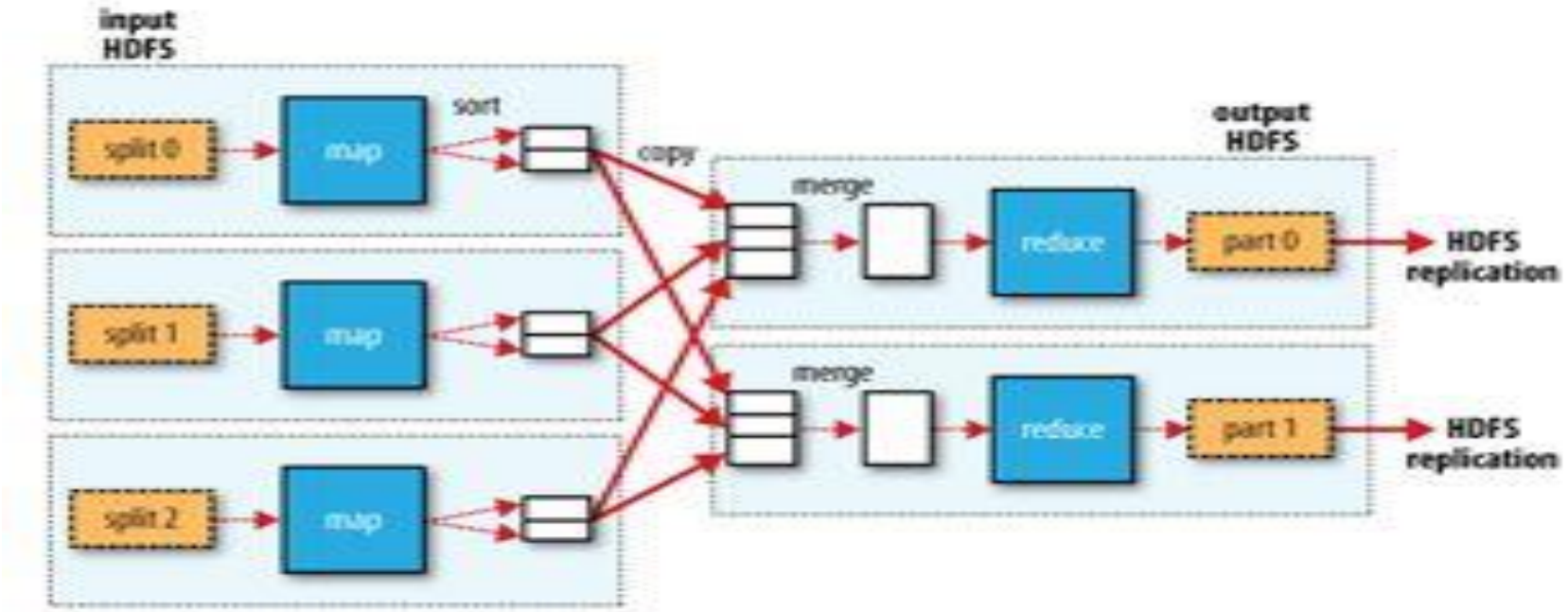



**Figure: Map reduce data flow for single reducer task.**

# BIG DATA

## MapReduce Flow for MULTIPLE REDUCE Tasks

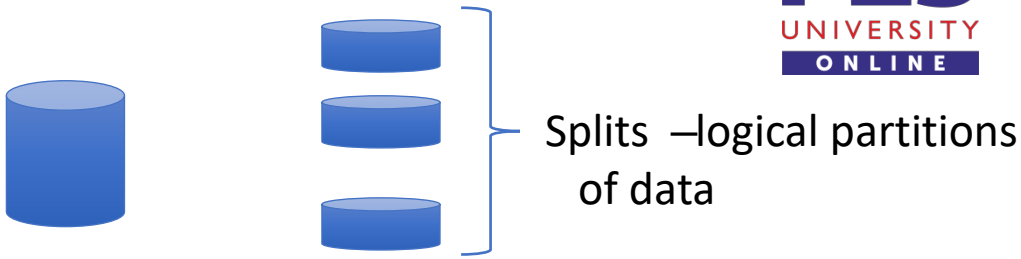



- Outputs are created on two different nodes and have to be merged.
- But available through HDFS on any node

# Map Reduce: Splits

## Map Reduce Split size considerations

- Split size proportional to parallelism
- Small split size
  - Advantages
    - Large #splits
    - Increased parallelism
    - Increased load balancing
  - Disadvantages
    - the overhead of managing the splits and of map task creation
    - begins to dominate the total job execution time.




Optimal split size == size of HDFS block (128MB) (default) on Hadoop v2

# Map Output

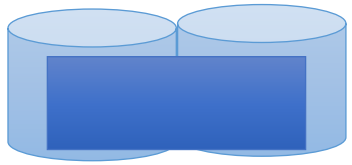



## Split == Block size

- All data required for Map
  - In the same node
  - No inter-node data transfer is required

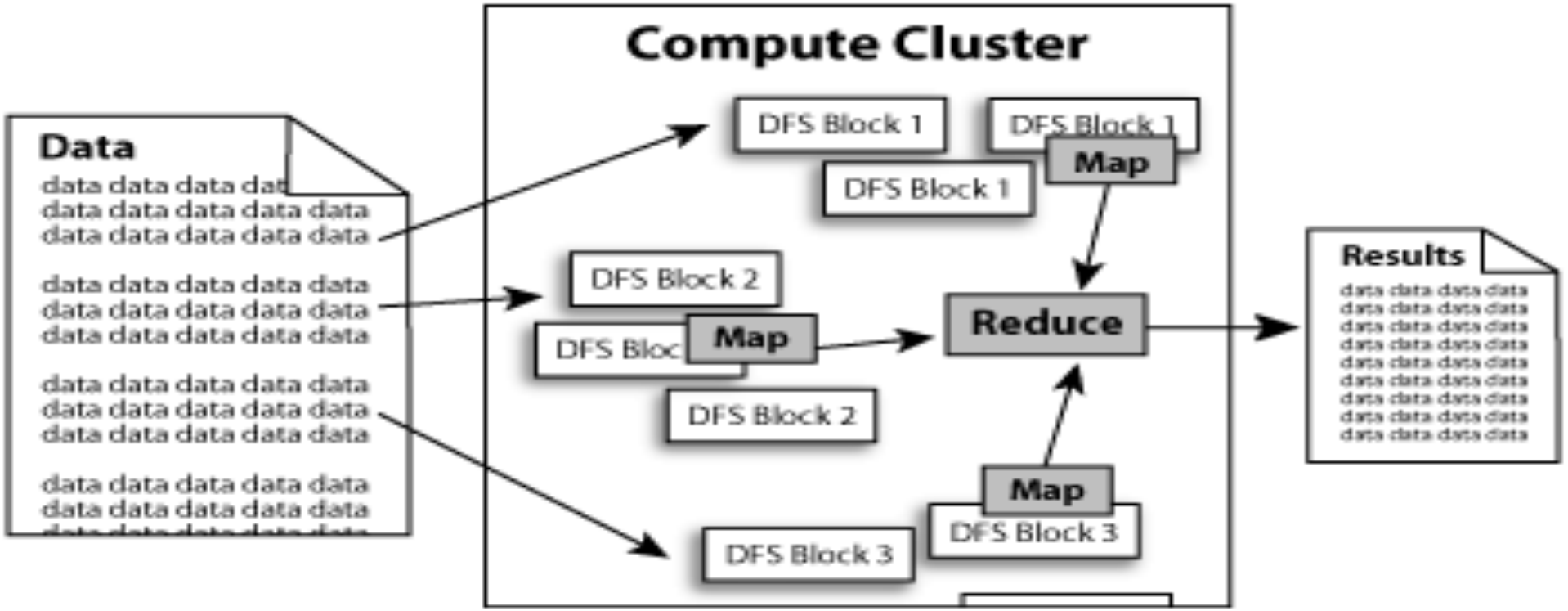
## Split != block size

- Data transfer across multiple nodes
- Impacts performance

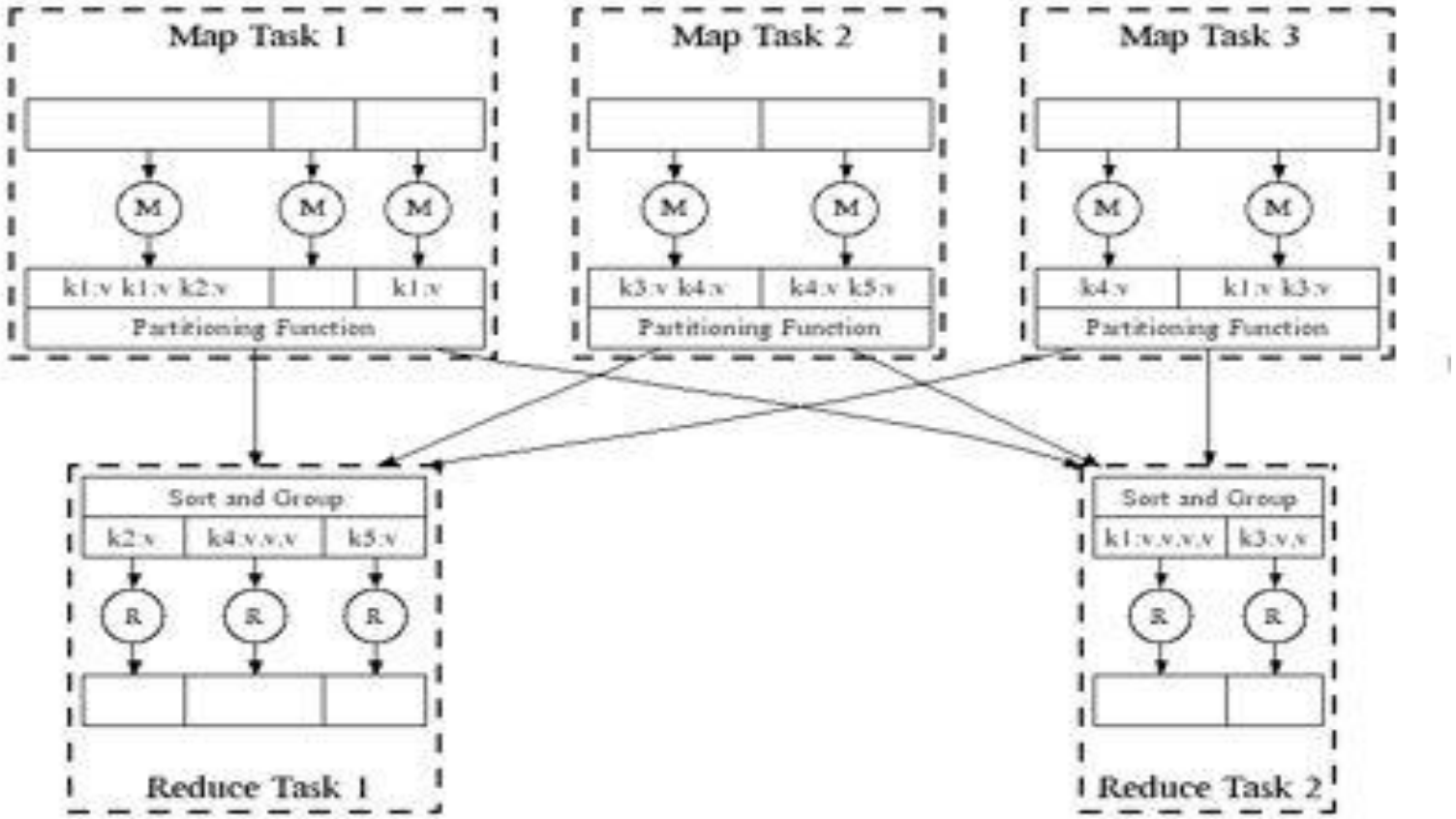


# BIG DATA

## Traditional: Move Data to Compute



# Big Data: Move Compute to Data




Parallel Execution

## Map Output


- **Where is Map output written to?**
  - Local disk and not HDFS
  - Why? Temporary output to be discarded after reduce.

- **Failure**
  - If the node running the map task fails
    - before the output has been consumed by reducer
  - Automatically rerun map task on another node

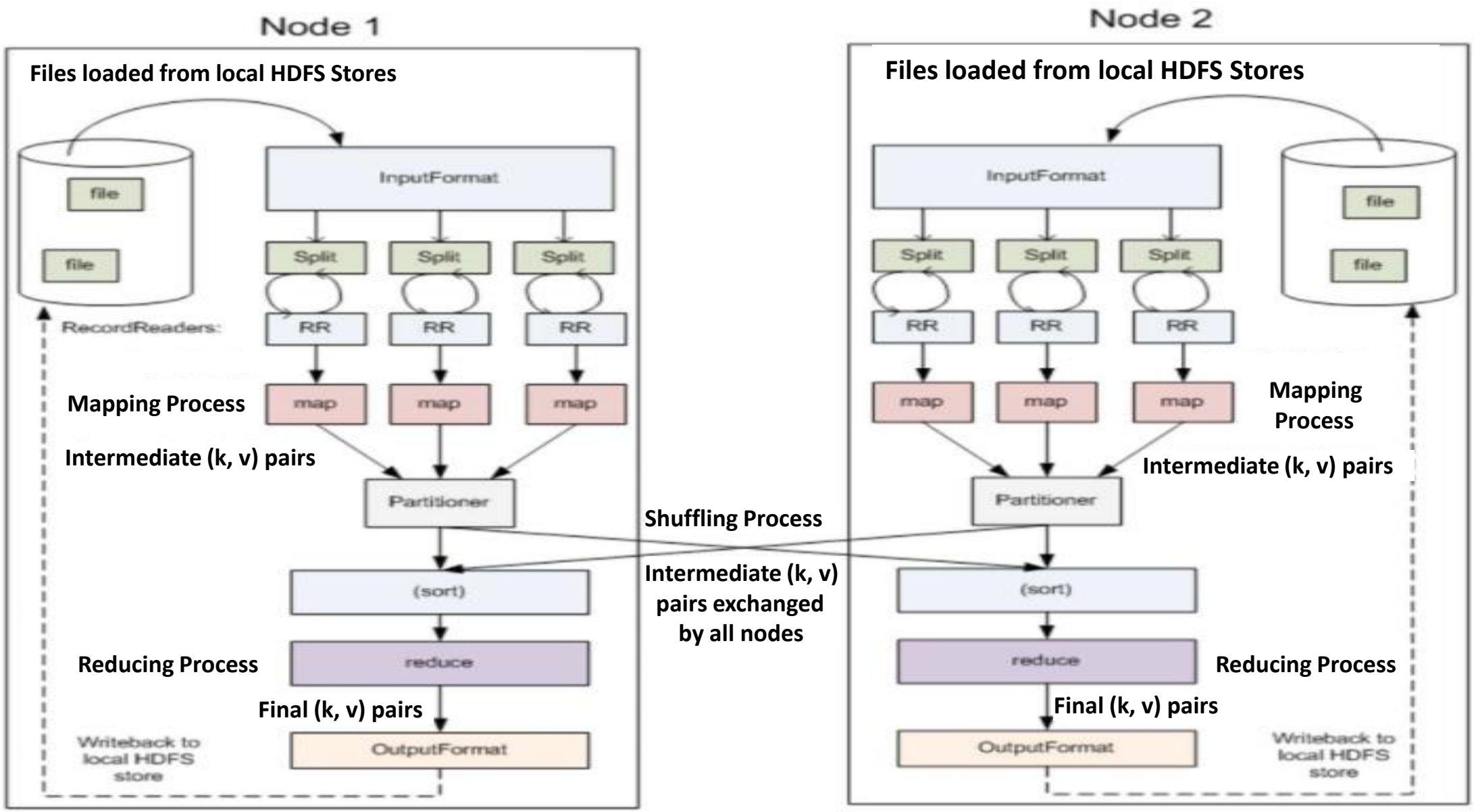
## Reduce Tasks

- Reduce tasks don't have the advantage
  - the input to a single reduce task is normally the output from all mappers.
- Sorted map outputs
  - have to be transferred across the network
  - Where to?
    - To the node where the reduce task is running
    - Merge data from different mappers
    - Then passed to the user-defined reduce function.
- The output of the reduce is normally stored in HDFS for reliability.
- Where is the reduce output stored
  - 1 on the local node where the reduce happens
  - Other replicas on off-rack nodes.
  - Consumes network bandwidth

# Summary: MapReduce

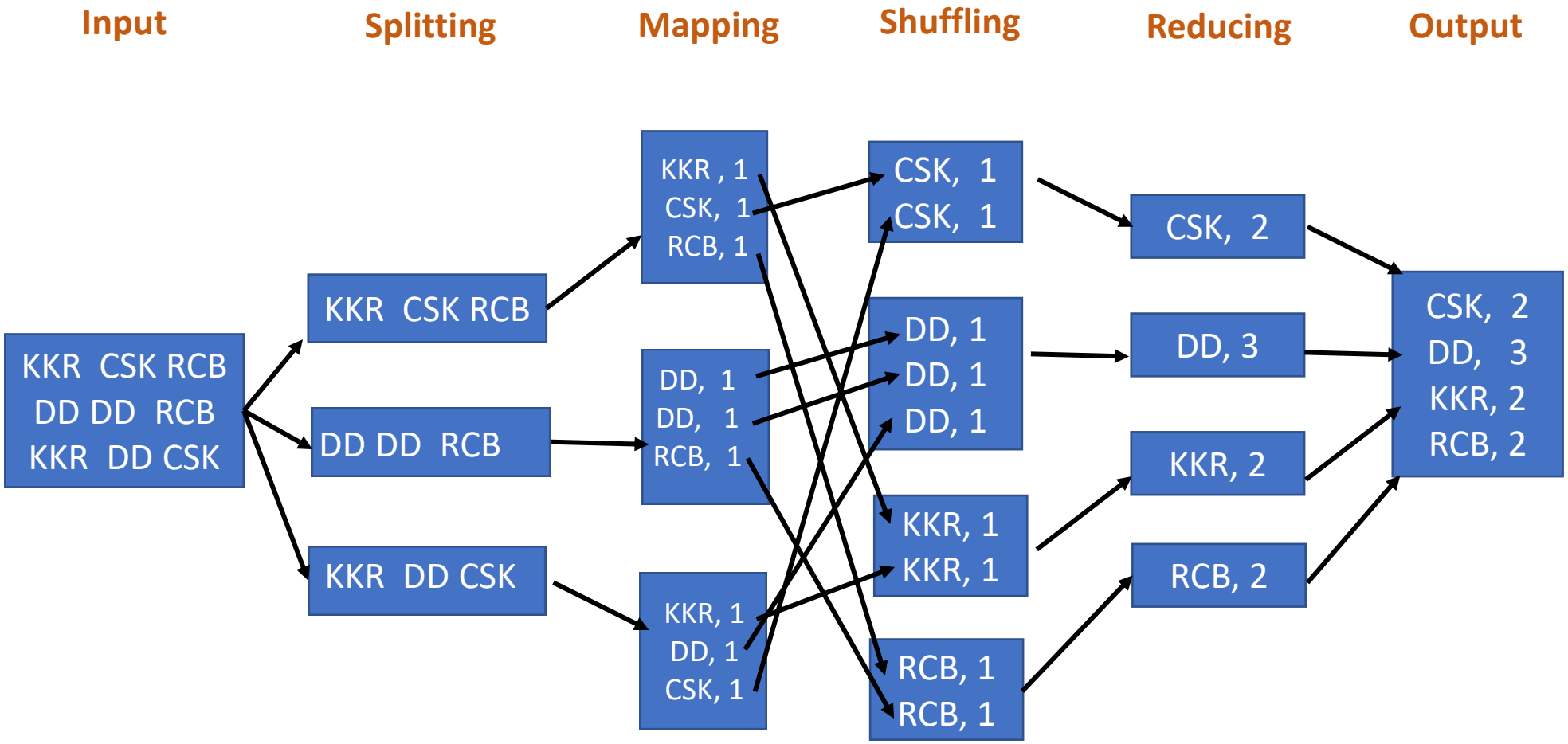
1. Parallelize Map: easy! each map task is independent of the other!
   - All Map output records with same key assigned to same Reducer
2. Transfer data from Map to Reduce:
   - Shuffle data
   - All Map output records with same key assigned to same Reduce task
   - use partitioning function, e.g., hash(key)%number of reducers
3. Parallelize Reduce: easy! each reduce task is independent of the other!
4. Implement Storage for Map input, Map output, Reduce input, and Reduce output
   - Map input: from distributed file system (in our case HDFS)
   - Map output: to local disk (at Map node); uses local file system (in our case Linux FS)
   - Reduce input: from (multiple) remote disks; uses local file systems
   - Reduce output: to distributed file system

# Map Reduce: Working


Node 1
Files loaded from local HDFS Stores
file
file
InputFormat
Split
Split
Split
RecordReaders:
RR
RR
RR
map
map
map
Mapping Process
Intermediate (k, v) pairs
Partitioner
Shuffling Process
Intermediate (k, v) pairs exchanged by all nodes
(sort)
reduce
Reducing Process
Final (k, v) pairs
OutputFormat
Writeback to local HDFS store
Node 2
Files loaded from local HDFS Stores
InputFormat
file
file
Split
Split
Split
RR
RR
RR
map
map
map
Mapping Process
Intermediate (k, v) pairs
Partitioner
(sort)
reduce
Reducing Process
Final (k, v) pairs
OutputFormat
Writeback to local HDFS store

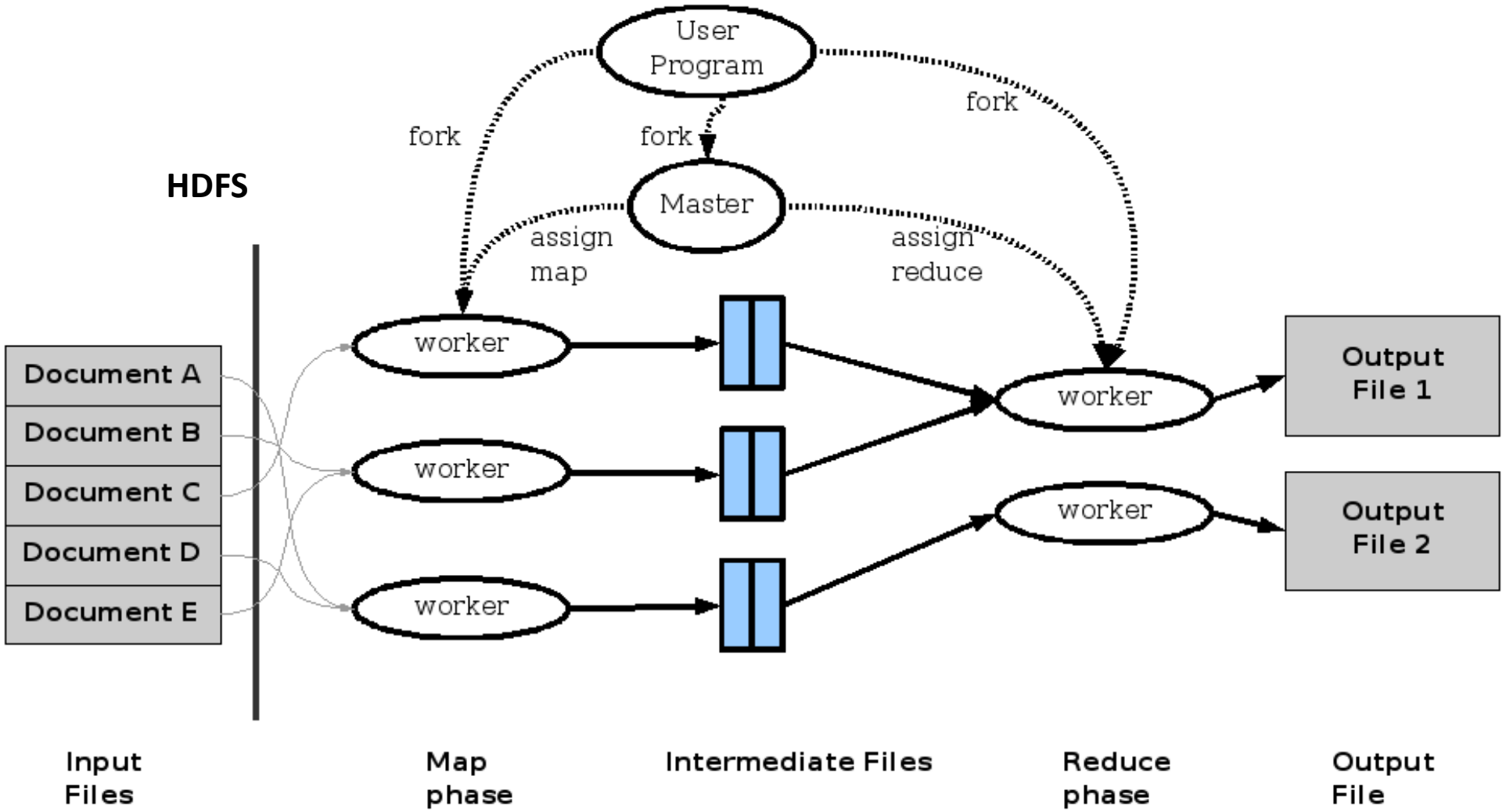
# BIG DATA

## Shuffling - example

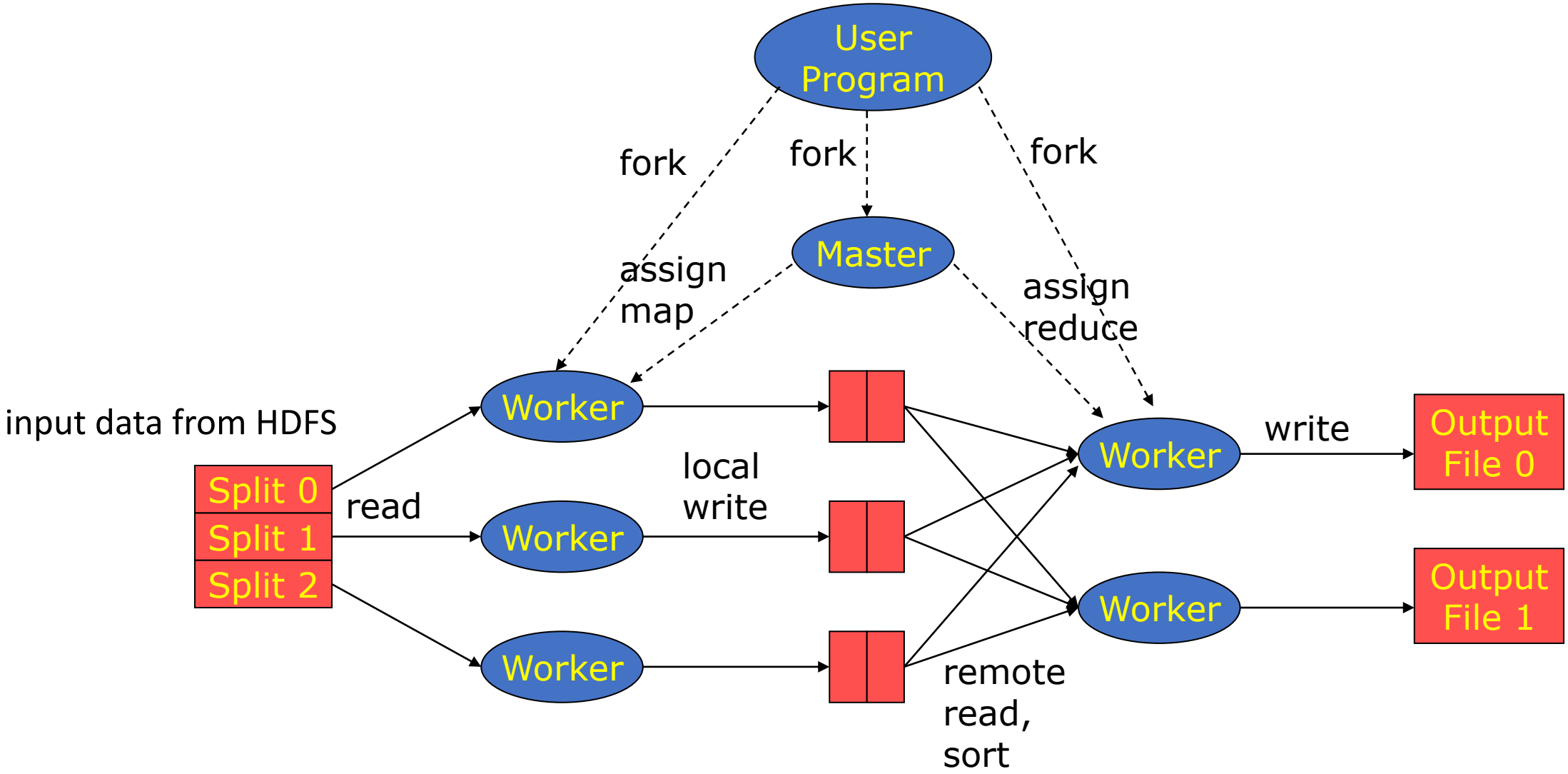



The overall Map-Reduce word count Process

# MapReduce: Execution overview



# Distributed Execution Overview

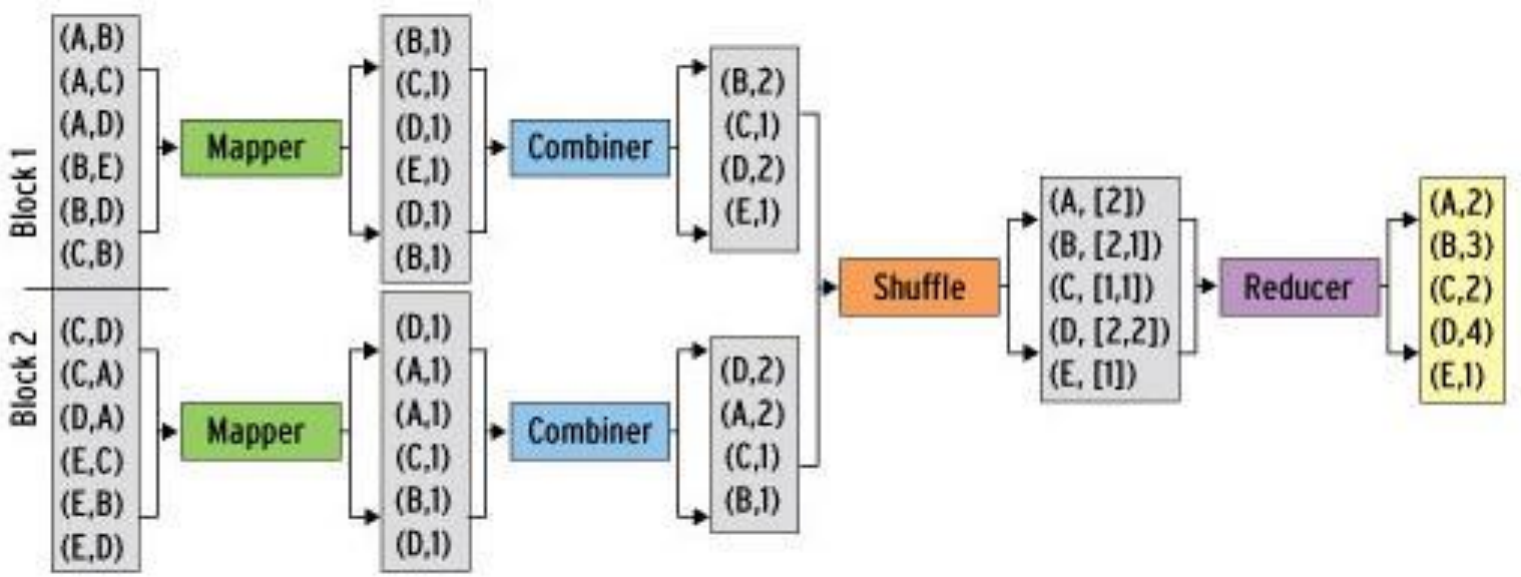



From Jeff Ullman's course sli

# Map Reduce: Combiners

# BIG DATA

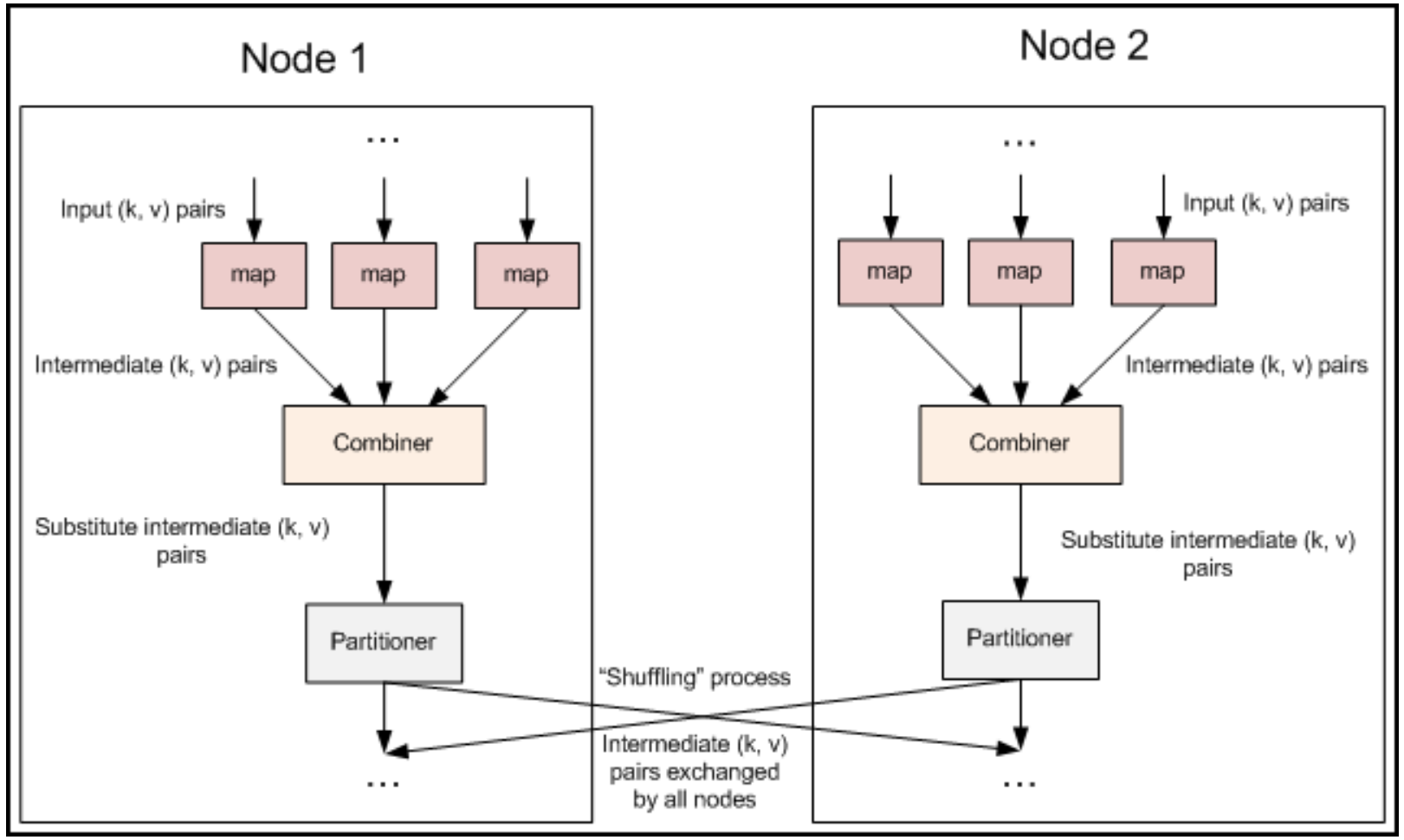
## MapReduce –Combiners - motivation




- Combine multiple map outputs before doing a reduce
- Can write a combiner function in program
  - Combiner will be run before reduce
- Mini-reducer

# BIG DATA

## Combiner –when does it run?



# BIG DATA

## Map Reduce - Main Program for Word Count

```
public static void main(String[] args) throws Exception {
 Configuration conf = new Configuration();
 String[] otherArgs = new GenericOptionsParser(conf, args).
                           getRemainingArgs();
 if (otherArgs.length < 2) {
   System.err.println("Usage: wordcount <in> [<in>...] <out>");
   System.exit(2);
 }
 Job job = new Job(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);
 for (int i = 0; i < otherArgs.length - 1; ++i) {
   FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
 }
 FileOutputFormat.setOutputPath(job,
   new Path(otherArgs[otherArgs.length - 1]));
 System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

**Combiner is set here**

# Review

## Review Question

- Suppose
  - We have a 2 GB file
  - Split size is 128 MB
  - We have 4 disks (one per node)
- How many splits are there?
- How many splits per disk?
- How many map tasks?
- How many map tasks per node?
- How many reduce tasks?

## Review Question - Solution

- Suppose
  - We have a 2 GB file
  - Split size is 128 MB
  - We have 4 disks (one per node)
- How many splits are there? - 16
- How many splits per disk? - 4
- How many map tasks? - 16
- How many map tasks per node? - 4
- How many reduce tasks? –User specified

# BIG DATA

## Sample Problem

Block 1

Far out in the uncharted backwaters of the unfashionable end of the Western Spiral arm of the Galaxy lies a small unregarded yellow sun. Orbiting this at a distance of roughly ninety-eight mi

Block 2

llion miles is an utterly insignificant little blue-green planet whose ape-descended life forms are so amazingly primitive that they still think digital watches are a pretty neat idea

- You run Word count using Hadoop on this data
- We know each block is an input split
- And each split is processed by a different mapper
- Do we get the right result?
- How will you solve this?

# BIG DATA

## Solution

Block 1

Far out in the uncharted backwaters of the unfashionable end of the Western Spiral arm of the Galaxy lies a small unregarded yellow sun. Orbiting this at a distance of roughly ninety-eight mi

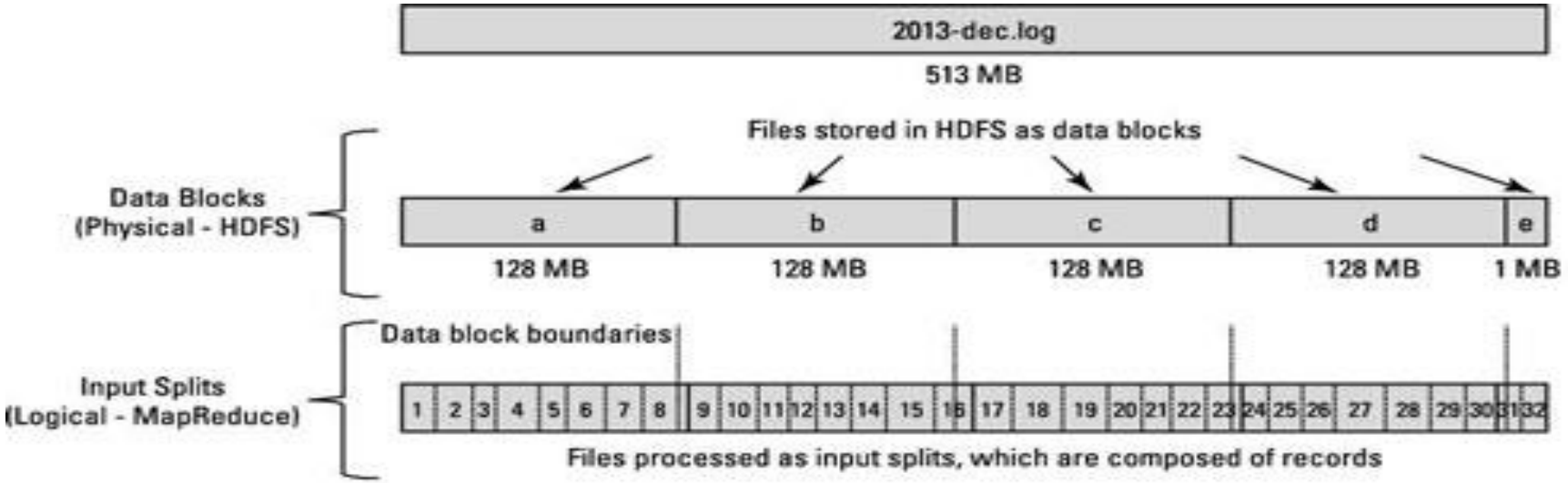
Block 2

llion miles is an utterly insignificant little blue-green planet whose ape-descended life forms are so amazingly primitive that they still think digital watches are a pretty neat idea

- Each split further broken into records
- In this case the records would look like
  - Far
  - Out
  - i n . ...
- How to handle incomplete records?
  - Mapper 1 will process million
  - By looking for EOR separator.

# BIG DATA

## Review Questions




map: (K1, V1) → list(K2, V2)
reduce: (K2, list(V2)) → list(K3, V3)

Record boundary → \n by default

Q: How can this be changed?

# Review Questions

## Review Questions

- Questions from T1, LOR 2.4

## Sample questions

- How many mappers and reducers will get started when trying to process a 230 MB file with Hadoop v2?
  - Ans: Block size = 128MB, so there will be two blocks. Assuming one block per split there will be 2 mappers
    - #reducers is configurable.
- Where is a combiner executed?
  - On the mapper.
- Write mappers and reducer pseudo code showing keys for counting #unique words in a file?
  - Similar to word count. Just that reducer does not have to write the count.

# Additional Notes, References and Videos

## Notes

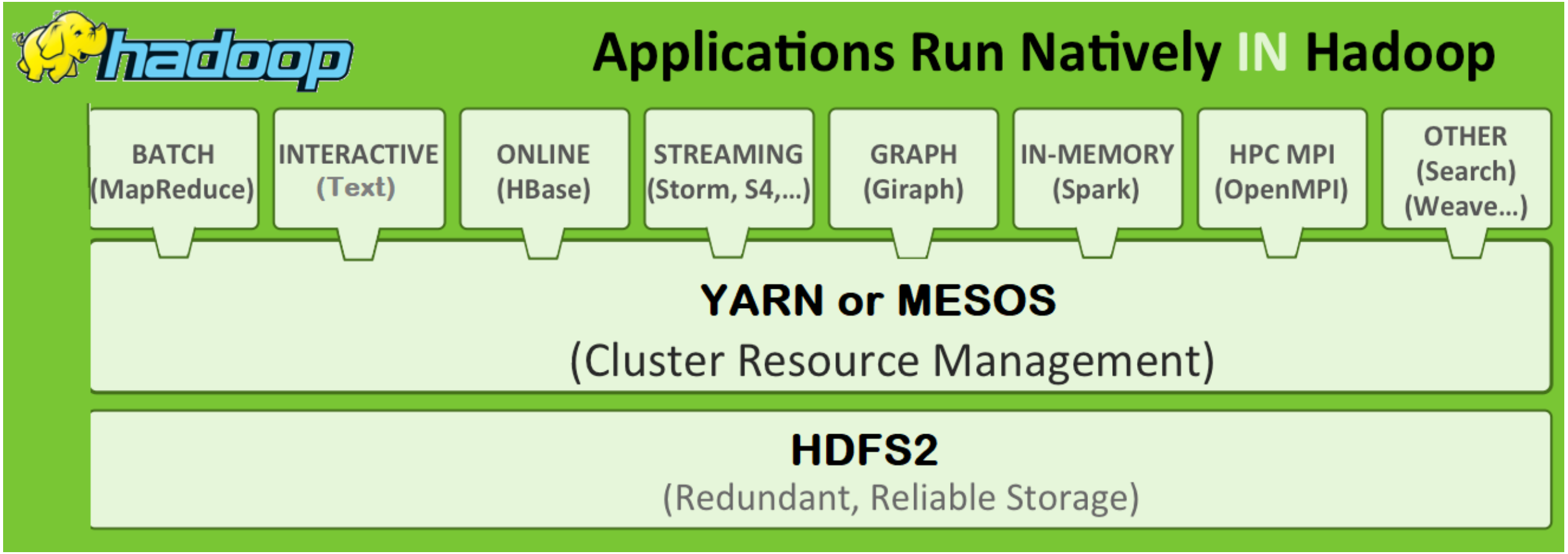
- Chapter 2.4 from T1 –Rajkamal
- Tom whites book is an excellent reference for the programming component.


# THANK YOU

**H.L. Phalachandra**

Dept. of Computer Science and Engineering

**phalachandra@pes.edu**


hadoop
Applications Run Natively IN Hadoop
BATCH (MapReduce)
INTERACTIVE (Text)
ONLINE (HBase)
STREAMING (Storm, S4,…)
GRAPH (Giraph)
IN-MEMORY (Spark)
HPC MPI (OpenMPI)
OTHER (Search) (Weave…)
YARN or MESOS
(Cluster Resource Management)
HDFS2
(Redundant, Reliable Storage)