BITS Pilani
Hyderabad Campus

BITS Pilani presentation

D. Powar
Lecturer,
BITS-Pilani, Hyderabad Campus

innovate achieve lead

BITS Pilani
Hyderabad Campus

SSZG527

Cloud Computing

Agenda:

- Hadoop components and importance of MapReduce
- Understanding MapReduce various logical steps
- Exploring the word count java program in detail
- Summary of MapReduce facts

MapReduce

- MapReduce is a software framework for easily running applications which processes large amount of data in parallel on large clusters having thousands of nodes of commodity hardware in a reliable and fault-tolerant manner

Hadoop components and importance of MapReduce

- MapReduce is fundamental building block in Hadoop
- Provides Framework for Massive parallel processing
- Provides scalability
- Programmer can focus on their program, and the framework takes care of the details of parallelization, fault-tolerance, locality optimization, load balancing
- **Paradigm shift:** In MapReduce programming model, computation goes to data rather than data coming to program. Processing takes place where data is.

Home work

1. Hadoop frame work - based on white paper published by Google in 2004

2. "MapReduce: Simplified data processing on large clusters" by Jeffrey Dean and Sanjay Ghemawat

MapReduce??

- Origin from Google, [OSDI'04]
- A simple programming model - distributed programming frame work (works on divide and conquer)
- Used for processing and generating large data sets
- Functional model
- For large-scale data processing
  - Exploits large set of commodity computers
  - Executes process in distributed manner
  - Offers high availability

Motivation

Lots of demands for very large scale data processing

A certain common themes for these demands

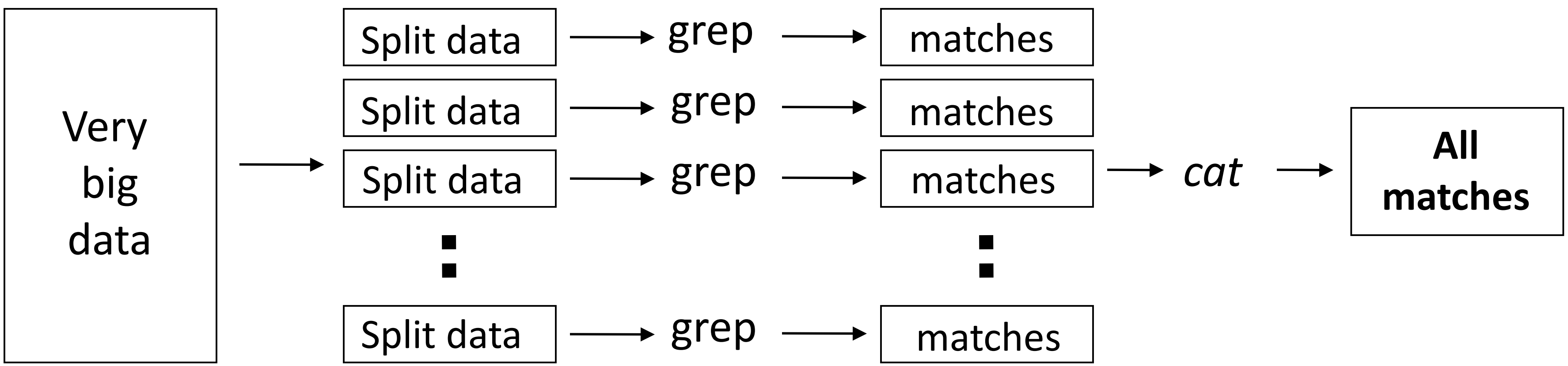
- Lots of machines needed (scaling)
- Two basic operations on the input
  - Map
  - Reduce

World Record

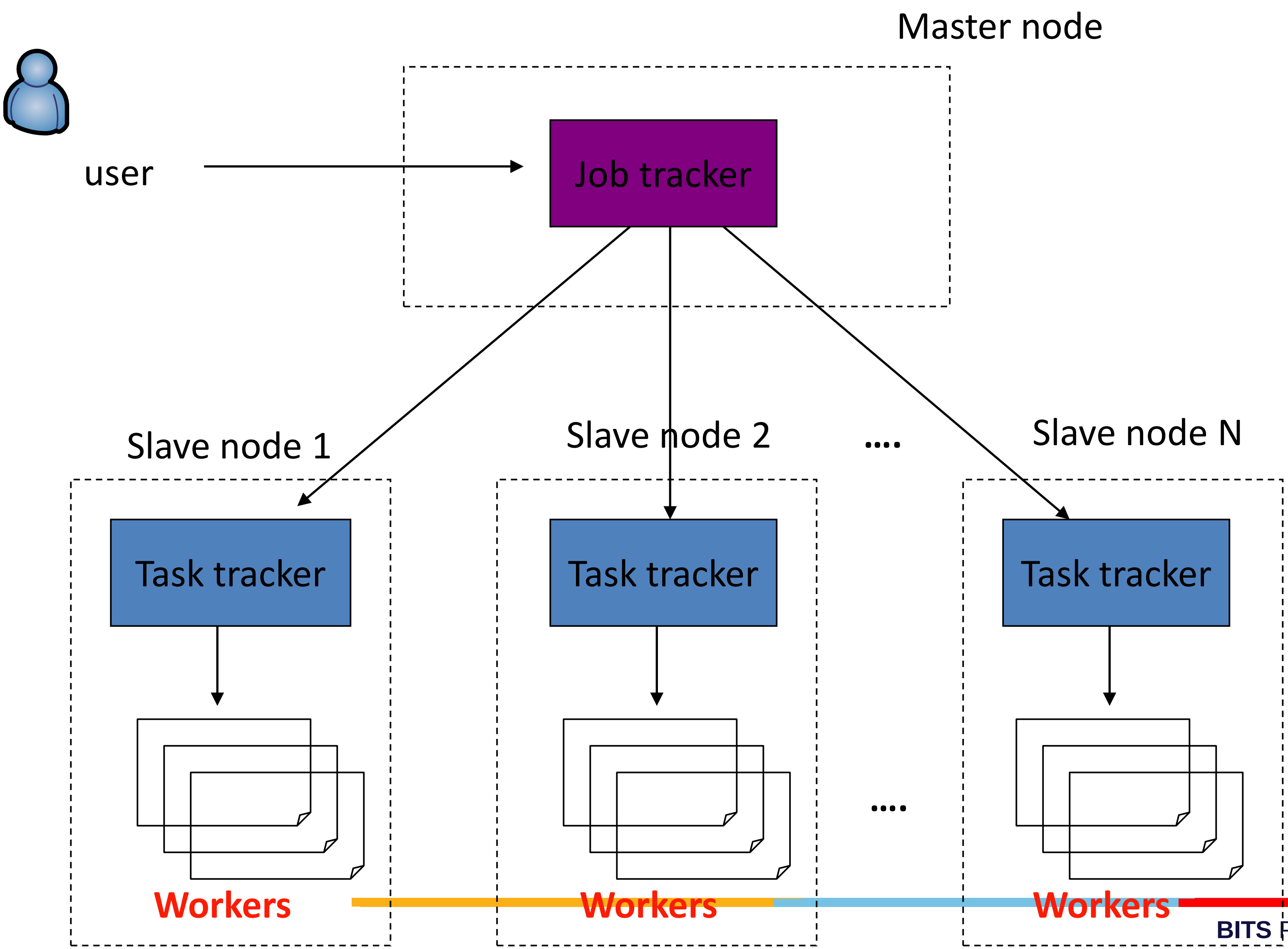
Record Setting Hadoop in the Cloud

- (Hadoop) We are proud to announce that we were able to run the Hadoop TeraSort benchmark to **sort 1TB of data** in a world-record setting time of **54 seconds on a 1003-node cluster** that Google graciously provided for our use. Of the 1003 instances, 998 instances ran the tasks, and 5 instances were used for control (e.g., ran the JobTracker, Zookeeper, etc.)

Distributed Grep



Architecture overview



The Job Tracker:

- Central authority for the complete MapReduce cluster and responsible for scheduling and monitoring MapReduce jobs
- Responds to client request for job submission and status

The Task Tracker:

- Workers that accepts map and reduce tasks from job tracker, launches them and keeps track of their progress, reports the same to job tracker.
- Keeps track of resource usage of tasks and kills the tasks that overshoots their memory limits

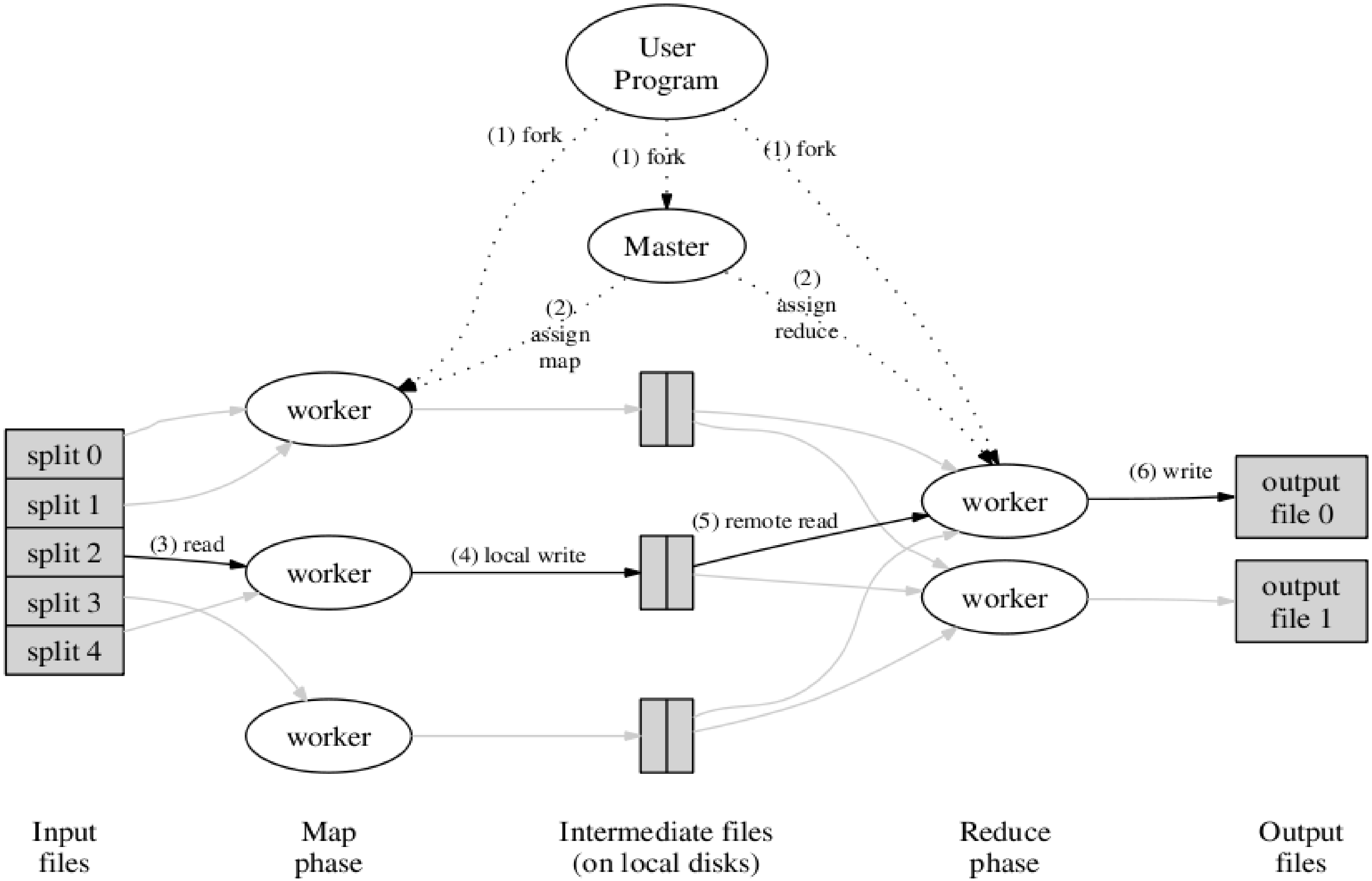
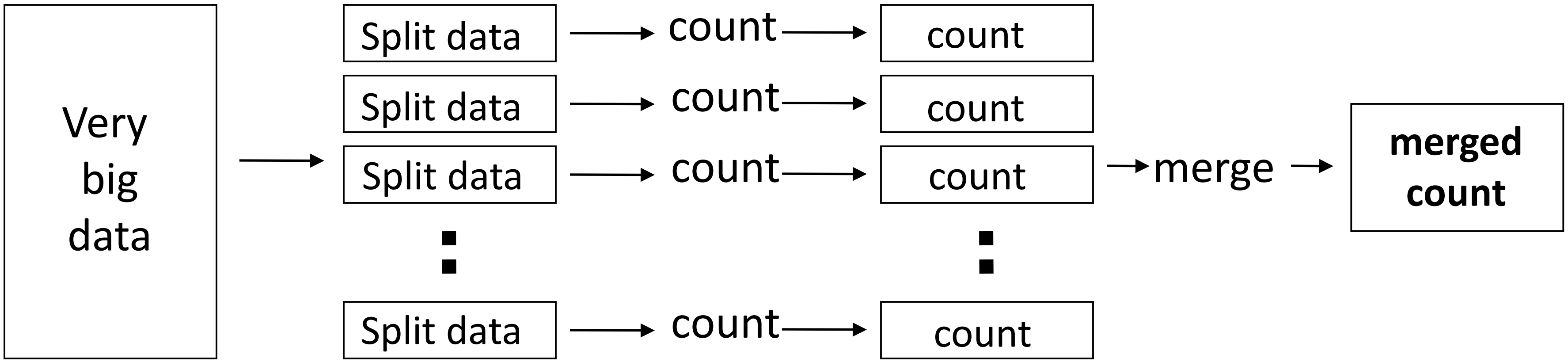


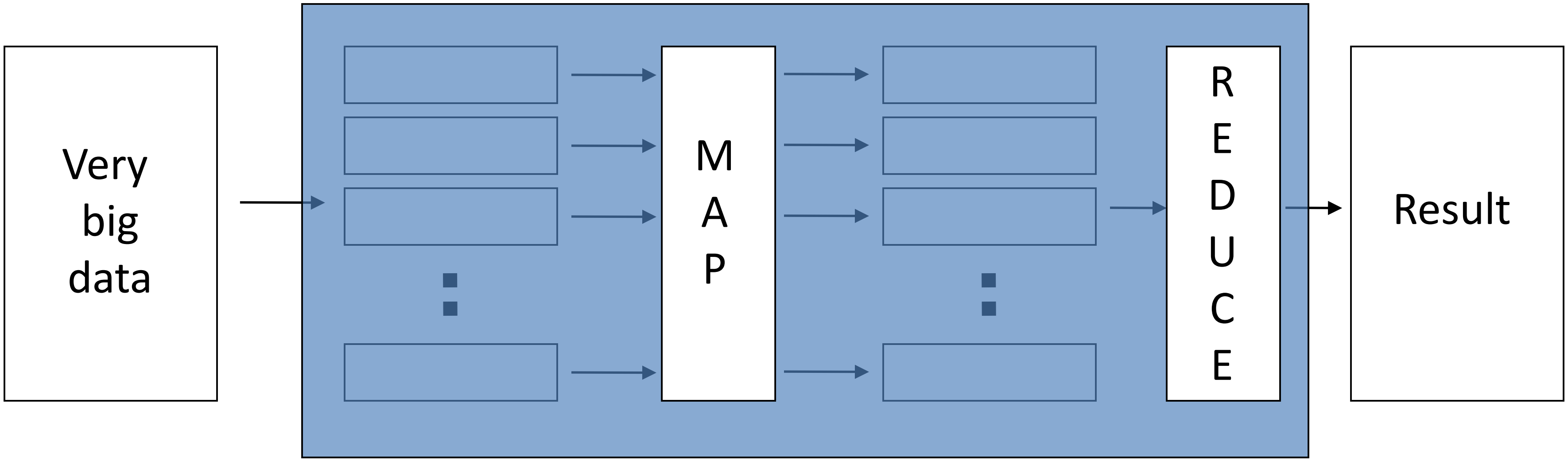
Figure 1: Execution overview

Ref: Jeffrey Dean and Sanjay Ghemawat

Distributed Word Count



Map+Reduce



Map:

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

Reduce

- Accepts intermediate key/value pair
- Emits output key/value pair

Typical Hadoop Cluster

Challenges of Cloud Environment

Cheap nodes fail, especially when you have many

- Mean time between failures for 1 node = 3 years
- MTBF for 1000 nodes = 1 day
- **Solution:** Build fault tolerance into system

Commodity network = low bandwidth

- **Solution:** Push computation to the data

Programming distributed systems is hard

- **Solution:** Restricted programming model: users write data-parallel "map" and "reduce" functions, system handles work distribution and failures

Flow of MapReduce

1. Define Inputs
2. Define Map function
3. Define Combiner function
4. Define Reduce function
5. Define output

MapReduce Programming Model

Data type: key-value *records*

Map function:

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

Reduce function:

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

Examples

let map(k,v) =emit (k.toUpper(), v.toUpper())

- **(“foo”, “bar”) -> (“FOO”,”BAR”)**
- **(“key2”,”data”) -> (“KEY2”,”DATA”)**

let map(k,v)= foreach char c in v :emit (k,c)

- **(“A”,”cats”)->(“A”,”c”),(“A”,”a”),(“A”,”t”),(“A”,”s”)**
- **(“B”,”hi”) ->(“B”,”h”), (“B”,”i”)**

let map(k,v)= if (isPrime(v)) then emit (k,v)

- **(“foo”,7) -> (“foo”,7)**
- **(“test”,10) -> (nothing)**

let map(k,v)= emit(v.length,v)

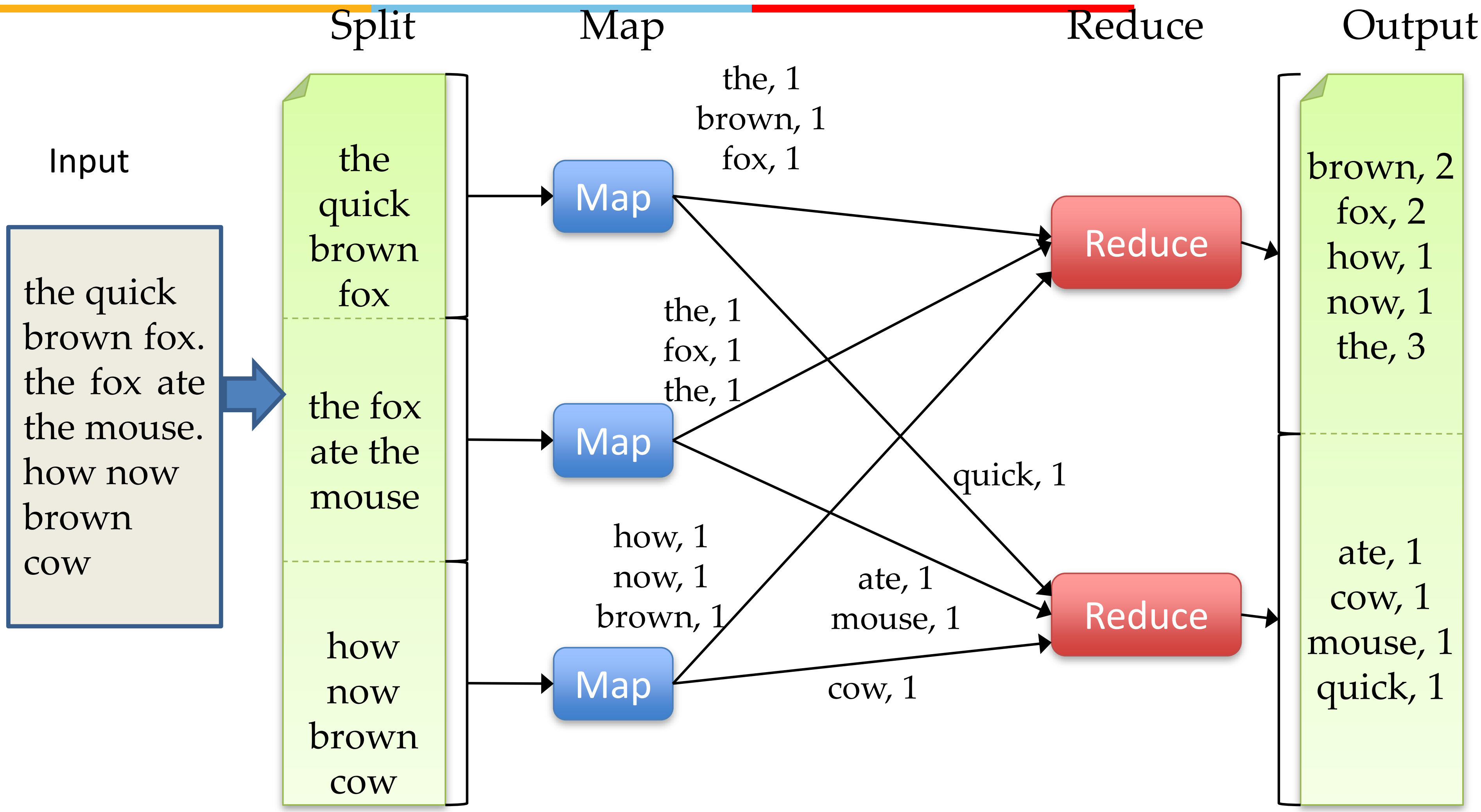
- **(“hi”,”test”)->(4,”test”)**
- **(“x”,”quick”) ->(5,”quick”)**

Example: Word Count


```
def mapper(line):
    foreach word in line.split():
        output(word, 1)


def reducer(key, values):
    output(key, sum(values))
```

Word Count Execution



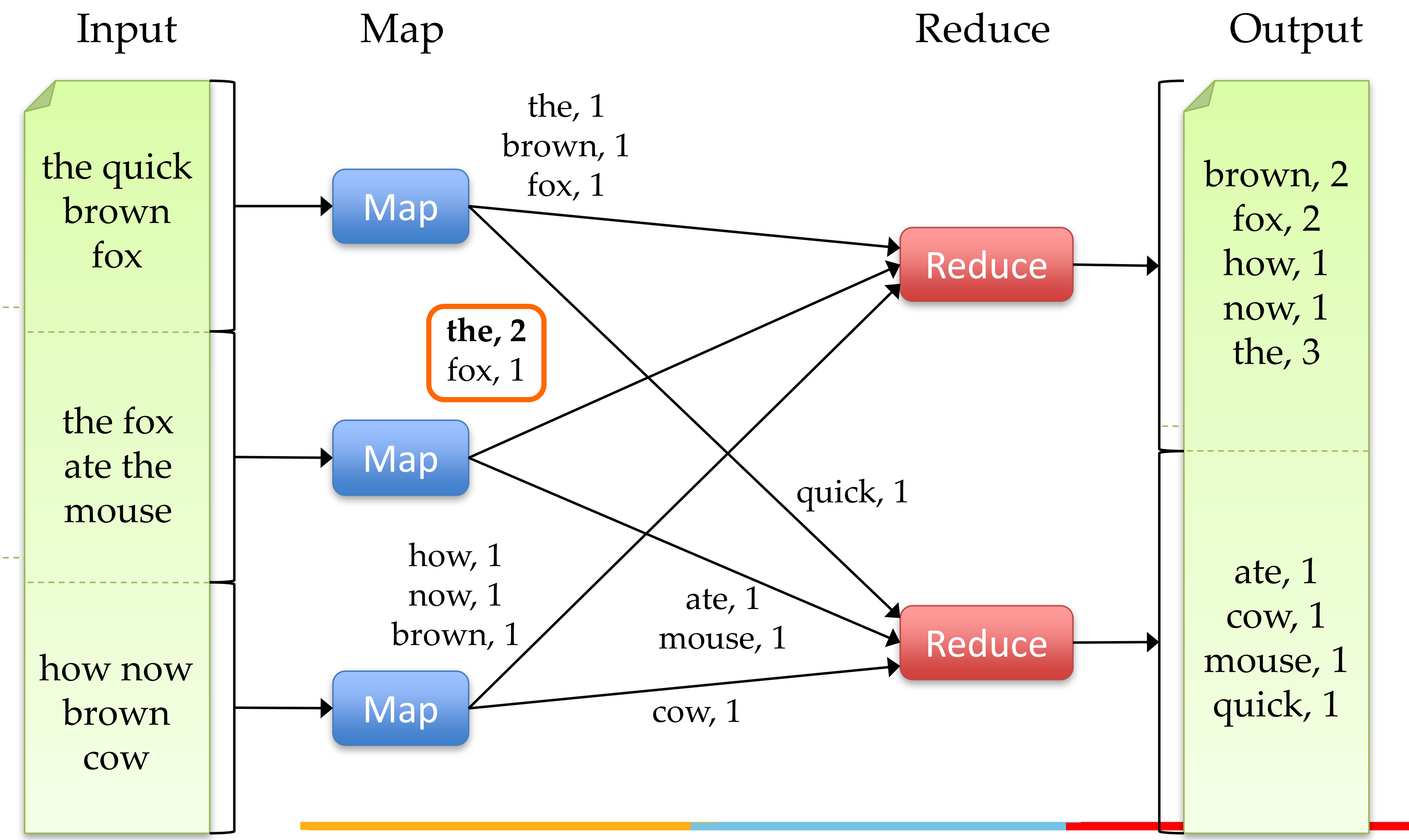
An Optimization: The Combiner

- Local reduce function for repeated keys produced by same map
- For associative ops. like sum, count, max
- Decreases amount of intermediate data

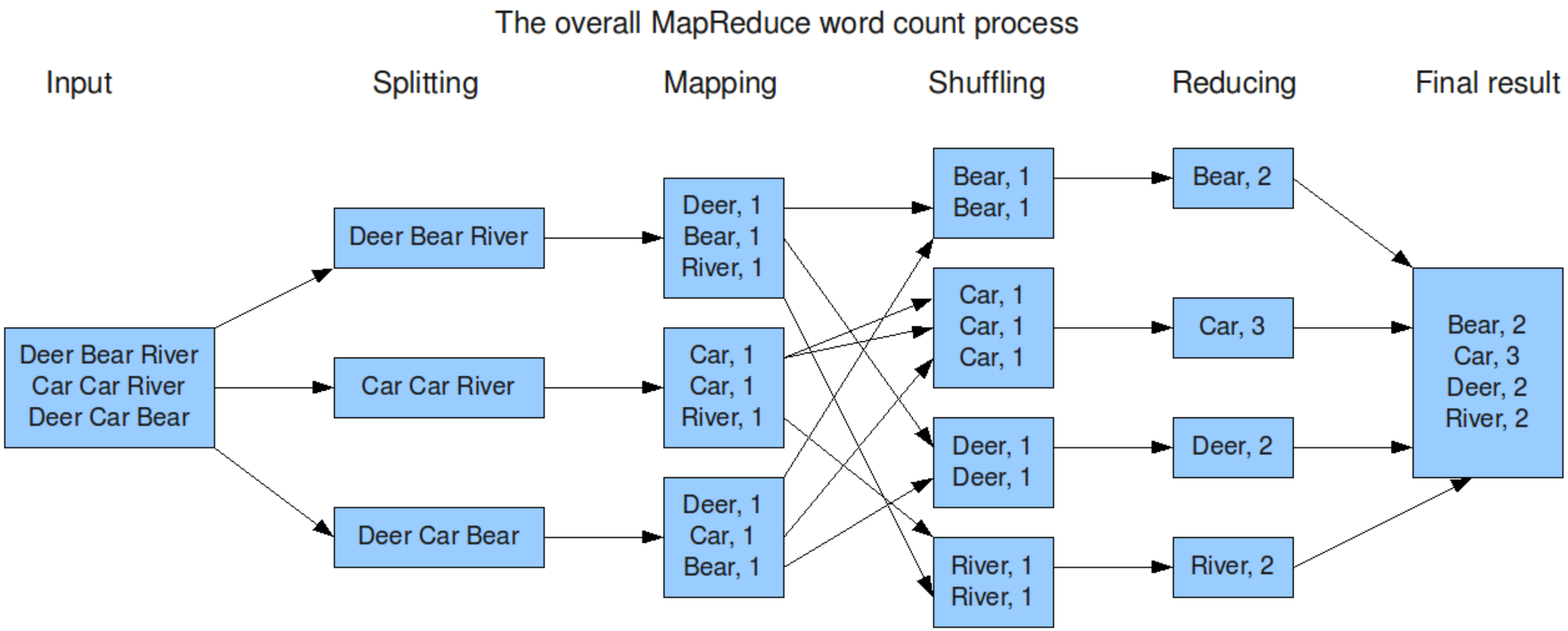
- Example: local counting for Word Count:

```
def combiner(key, values):
    output(key, sum(values))
```

Word Count with Combiner



Overall Word Count Execution (2)



Word Count implementation (java)

Objective:

- To count number of distinct words in each file

wordcount.java

```
public class WordCount {

 public static class Map extends 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,InWritable> output, Reporter reporter )
    throws IOException, InterruptedException {
      String line = value.toString();
      StringTokenizer tokenizer = new StringTokenizer(line);
      while (tokenizer.hasMoreTokens()) {
         word.set(tokenizer.nextToken());
                 output.collect(word, one);
      }
   }
 }
public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {
   public void reduce(Text key, Iterable<IntWritable> values, OutputCollector<Text,InWritable> output,
   Reporter reporter ) throws IOException, InterruptedException {
      int sum = 0;
      while(values.hasNext()){
         sum += values.next().get();
       }
      output.collect(key, new IntWritable(sum));
   }
 }
```

```
public static void main(String[] args) throws Exception {

   JobConf job = new JobConf(WordCount.class);
   job.setJobName("wordcount")

   job.setOutputKeyClass(Text.class);
   job.setOutputValueClass(IntWritable.class);

   job.setMapperClass(Map.class);
   job.setCombinerClass(Reducer.class)
   job.setReducerClass(Reduce.class);

   job.setInputFormatClass(TextInputFormat.class);
   job.setOutputFormatClass(TextOutputFormat.class);

   FileInputFormat.addInputPath(job, new Path(args[0]));
   FileOutputFormat.setOutputPath(job, new Path(args[1]));

   JobClient.runJob(job);
 }
} // WordCount class end
```

Running the application

Step 1: compile your program.java and create a jar

Step 2: Place the files in appropriate HDFS directory

Step 2: Run the application

/user/WILP/wordcount/input/file01 (Hello WILP students)

/user/WILP/wordcount/input/file02 (How are you! Bye for now)

$bin/hadoop jar wc.jar WordCount /user/WILP/wordcount/input /user/WILP/wordcount/output

Output:

cat /user/WILP/wordcount/output/part-r-00000

are 1
Bye 1
For 1
Hello 1
How 1
Now 1
Students 1
You! 1
WILP 1

Word Count example code (java)

http://hadoop.apache.org/docs/stable/mapred_tutorial.html

http://wiki.apache.org/hadoop/WordCount

MapReduce Execution Details

- Mappers preferentially scheduled on same node or same rack as their input block
  - Minimize network use to improve performance

- Mappers save outputs to local disk before serving to reducers
  - Allows recovery if a reducer crashes
  - Allows running more reducers than # of nodes

Fault Tolerance in MapReduce

1. If a task crashes:
   – Retry on another node
     • OK for a map because it had no dependencies
     • OK for reduce because map outputs are on disk
   – If the same task repeatedly fails, fail the job or ignore that input block

Fault Tolerance in MapReduce

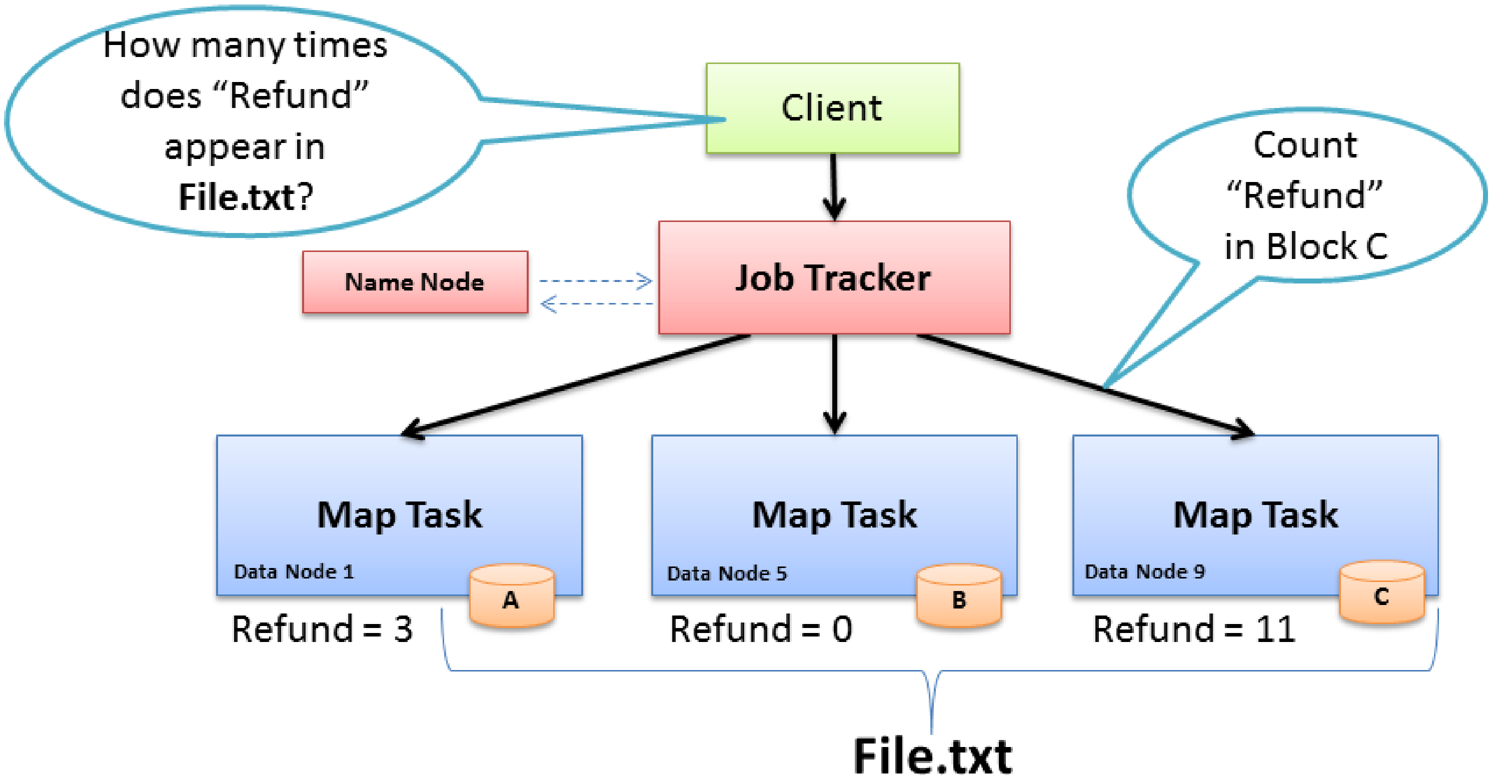
2. If a node crashes:
   - Relaunch its current tasks on other nodes
   - Relaunch any maps the node previously ran
     - Necessary because their output files were lost along with the crashed node

Fault Tolerance in MapReduce

3. If a task is going slowly (straggler):
   - Launch second copy of task on another node
   - Take the output of whichever copy finishes first, and kill the other one

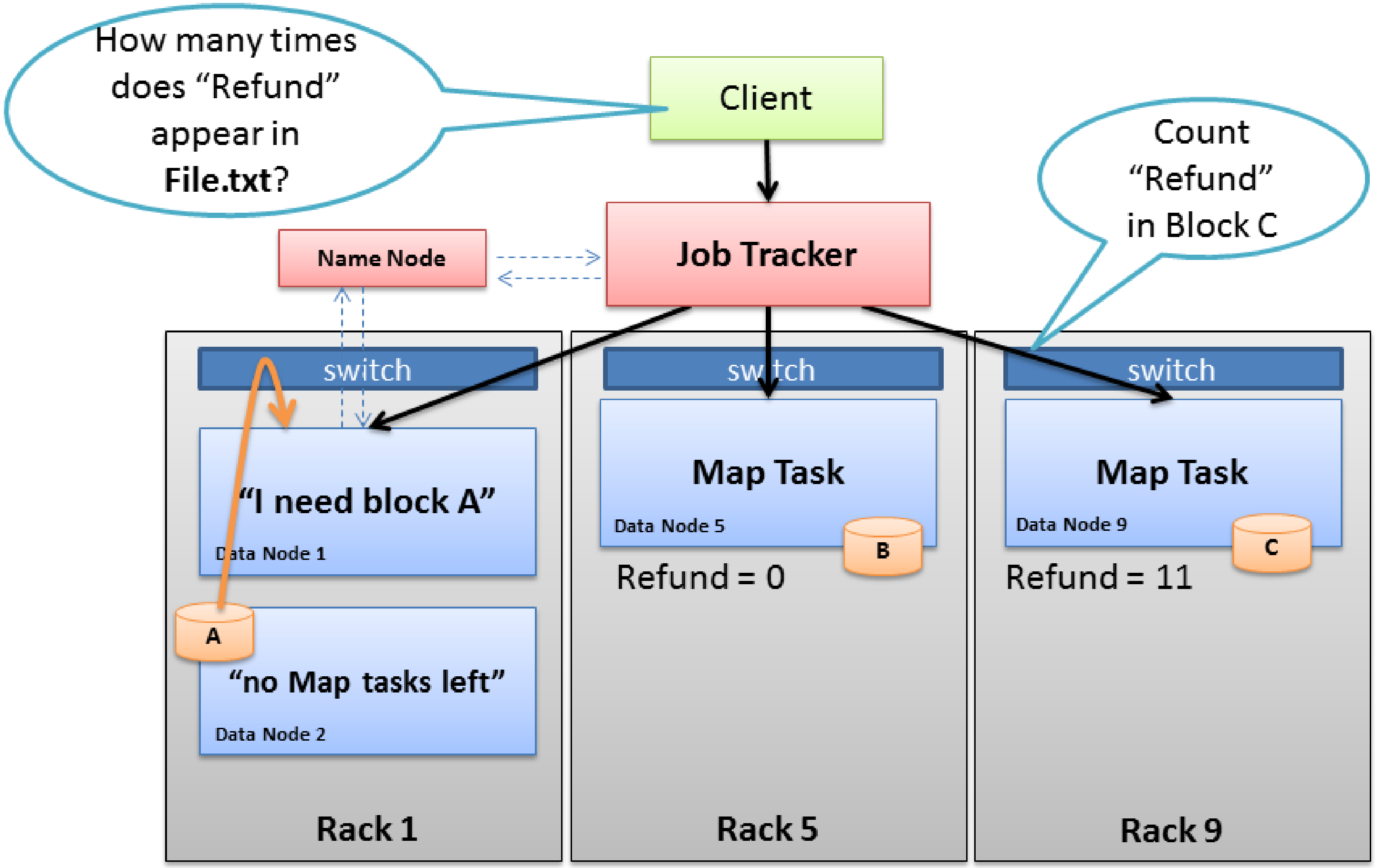
- Critical for performance in large clusters (many possible causes of stragglers)

MapReduce (Map task)



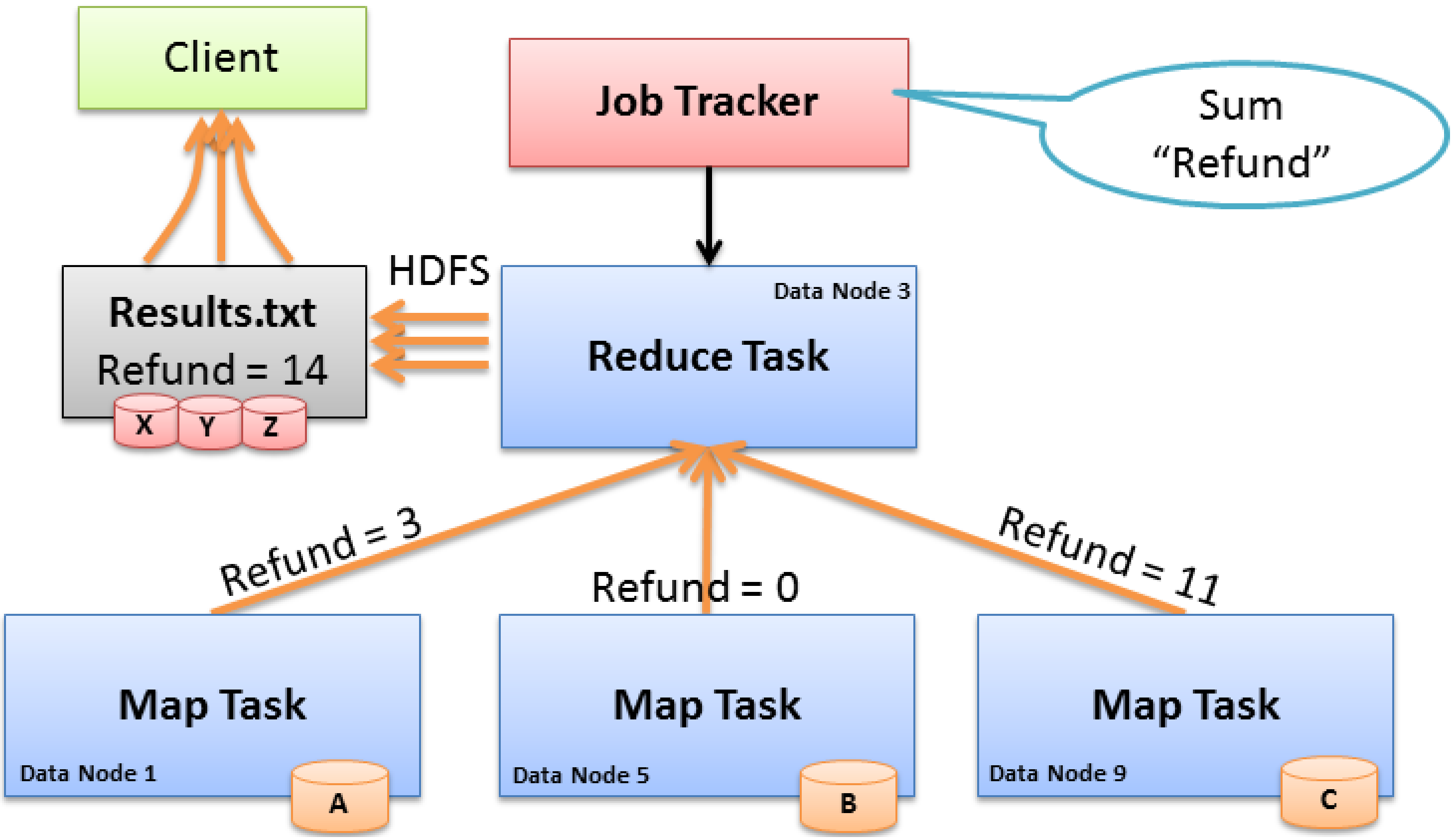
- **Map:** “Run this computation on your local data”
- Job Tracker delivers Java code to Nodes with local data

What if data is not local?



- Job Tracker tries to select Node in same rack as data
- Name Node rack awareness

MapReduce (Reduce task)



- **Reduce:** "Run this computation across Map results"
- Map Tasks <u>send output data to Reducer over the network</u>
- Reduce Task data output <u>written to and read from HDFS</u>

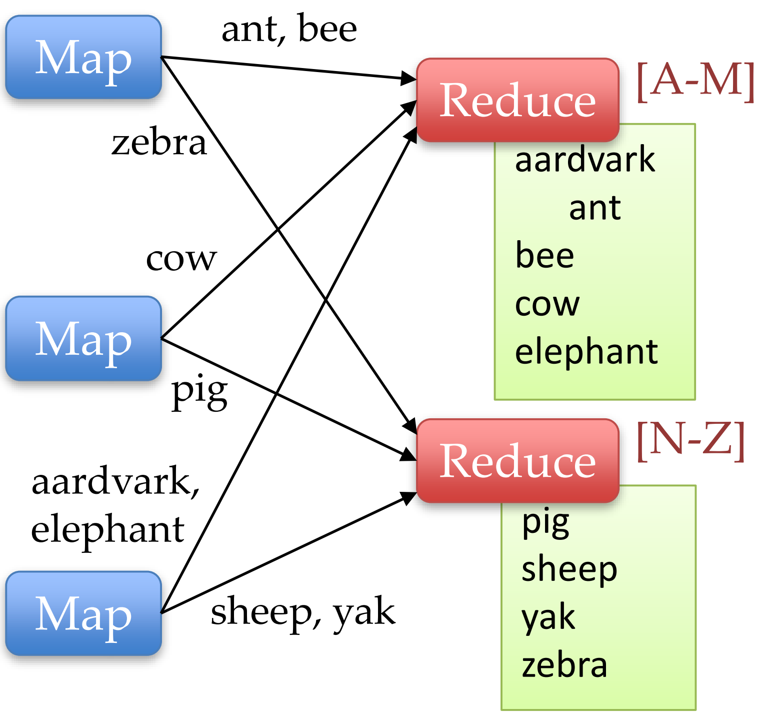
Examples

Sort

Input: (key, value) records

Output: same records, sorted by key

Trick: Pick partitioning function p such that $k_1 < k_2 \Rightarrow p(k_1) < p(k_2)$



Inverted Index

- **Input:** (filename, text) records
- **Output:** list of files containing each word

- **Map:**

```
foreach word in text.split():
    output(word, filename)
```

- **Combine:** uniquify filenames for each word

- **Reduce:**

```
def reduce(word, filenames):
    output(word, sort(filenames))
```

Inverted Index Example

hamlet.txt

to be or
not to be

to, hamlet.txt
be, hamlet.txt
or, hamlet.txt
not, hamlet.txt

12th.txt

be not
afraid of
greatness

be, 12th.txt
not, 12th.txt
afraid, 12th.txt
of, 12th.txt
greatness, 12th.txt

afraid, (12th.txt)
be, (12th.txt, hamlet.txt)
greatness, (12th.txt)
not, (12th.txt, hamlet.txt)
of, (12th.txt)
or, (hamlet.txt)
to, (hamlet.txt)

Summary of MapReduce facts

- Number of Map's: depends on Input data size, usually 10-100 per node.

  SetNumMapTasks(int) can be used to set it higher

- Number of Reduce's : Its legal to have zero Reducer if no reduction is desired

  setNumReduceTasks(int)

- The Hadoop framework is in Java, but it supports the streaming, thus making it possible to write the MapReduce in other languages like .Net, C#, etc

Summary

- MapReduce's data-parallel programming model hides complexity of distribution and fault tolerance
- Principal philosophies:
  - *Make it scale*, so you can throw hardware at problems
  - *Make it cheap*, saving hardware, programmer and administration costs (but necessitating fault tolerance)
- Hive and Pig further simplify programming
- MapReduce is not suitable for all problems, but when it works, it may save you a lot of time