Cloud Programming: Lecture4 – MapReduce Parallel Programming

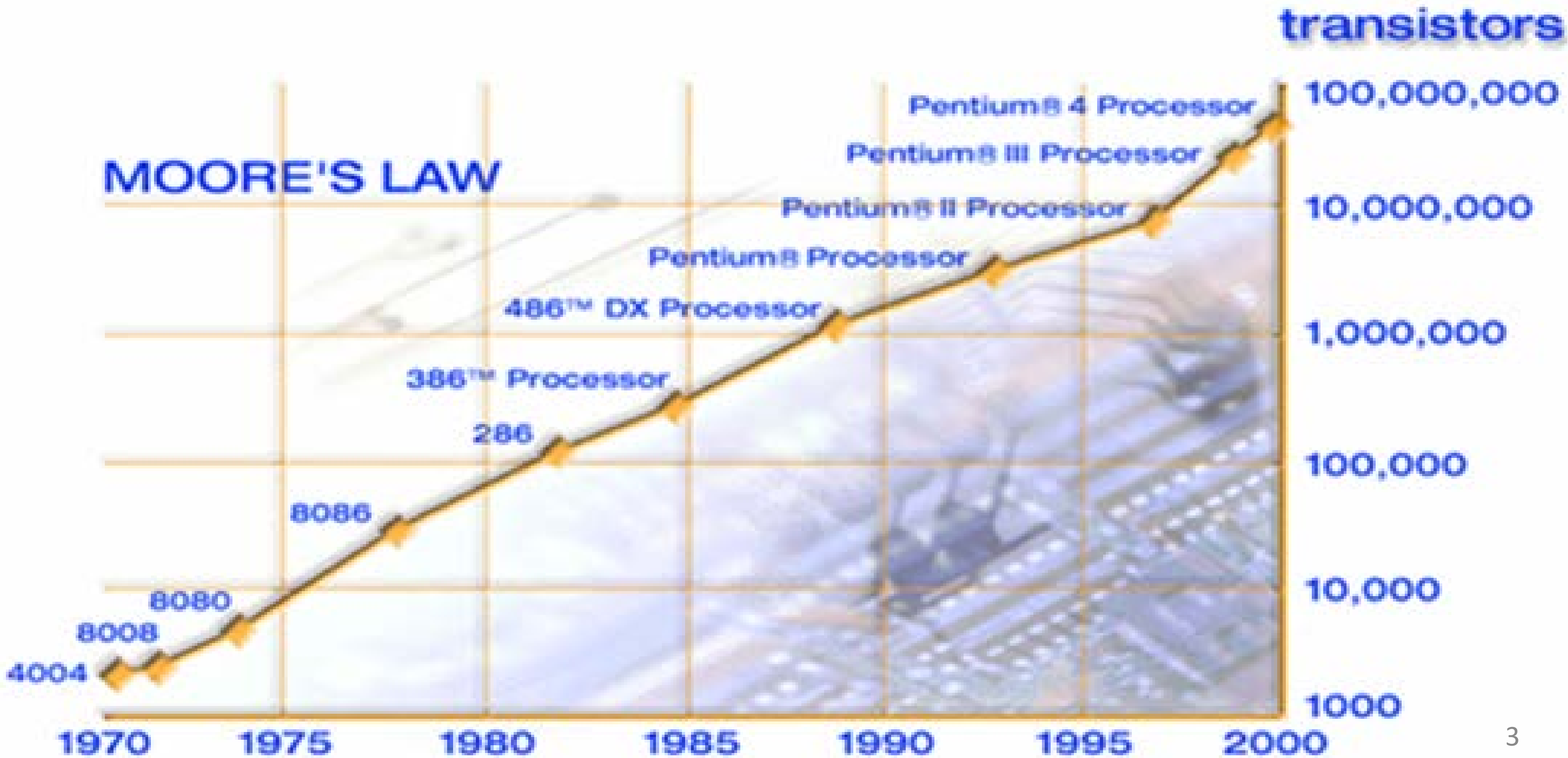
National Tsing-Hua University
2015, Spring Semester

Outline

- Distributed Computing Overview
- MapReduce Framework
- MapReduce(Hadoop) Programming

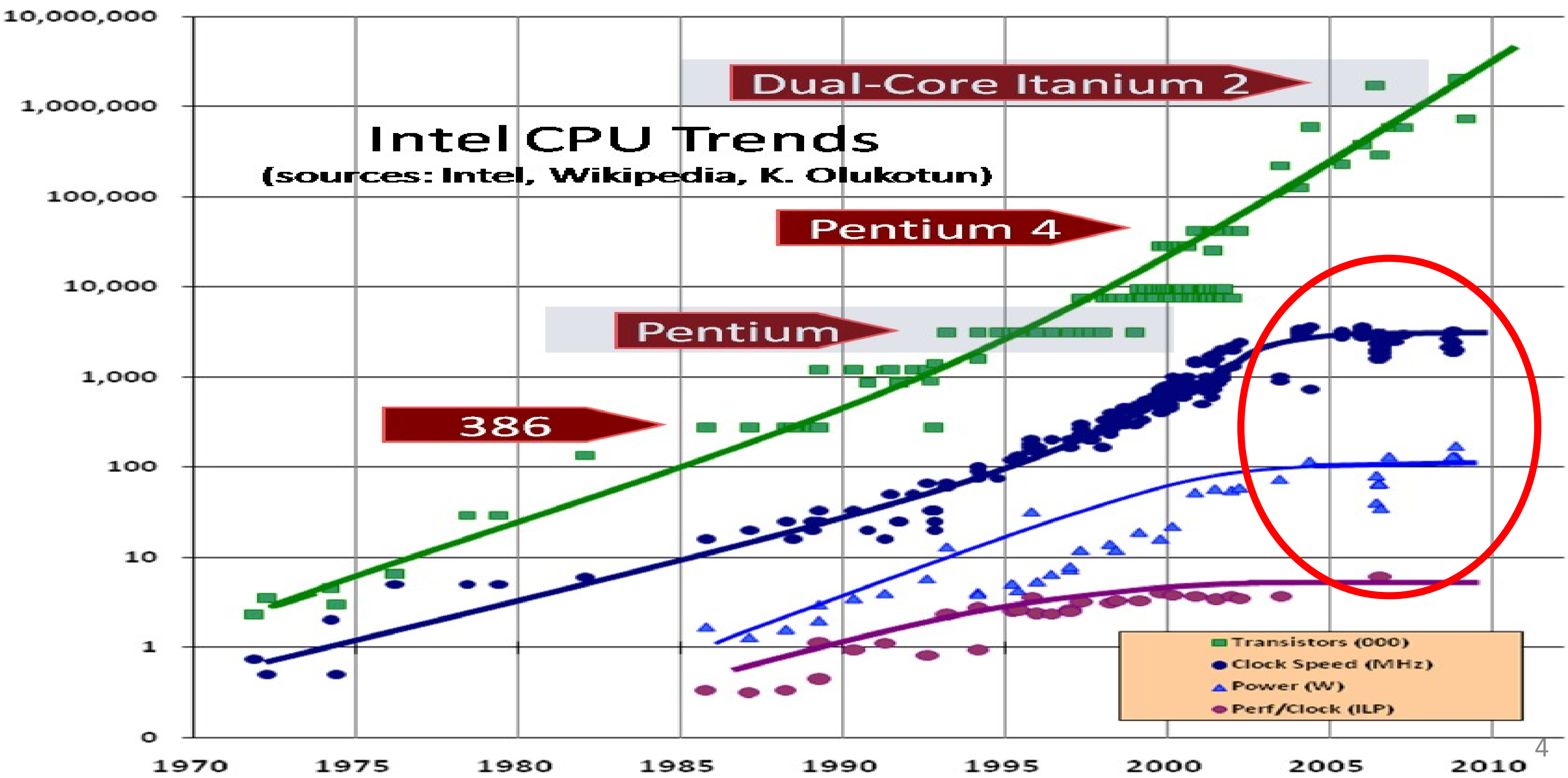
Moore's Law

- The observation that, over the history of computing hardware, the number of transistors on integrated circuits doubles approximately every two years



The Death of CPU Scaling

- Increase of transistor density ≠performance
  - The power consumption and clock speed improvements collapsed



Trend of Parallel Computers

Single-Core Era

Enabled by:
- Moore's Law
- Voltage Scaling

Constraint by:
- Power
- Complexity

Assembly → C/C++→Java …

Muti-Core Era

Enabled by:
- Moore's Law
- SMP

Constraint by:
- Power
- Parallel SW
- Scalability

Pthread → OpenMP …

Distributed System Era

Enabled by:
- Networking

Constraint by:
- Synchronization
- Comm. overhead

MPI → MapReduce …

Heterogeneous Systems Era

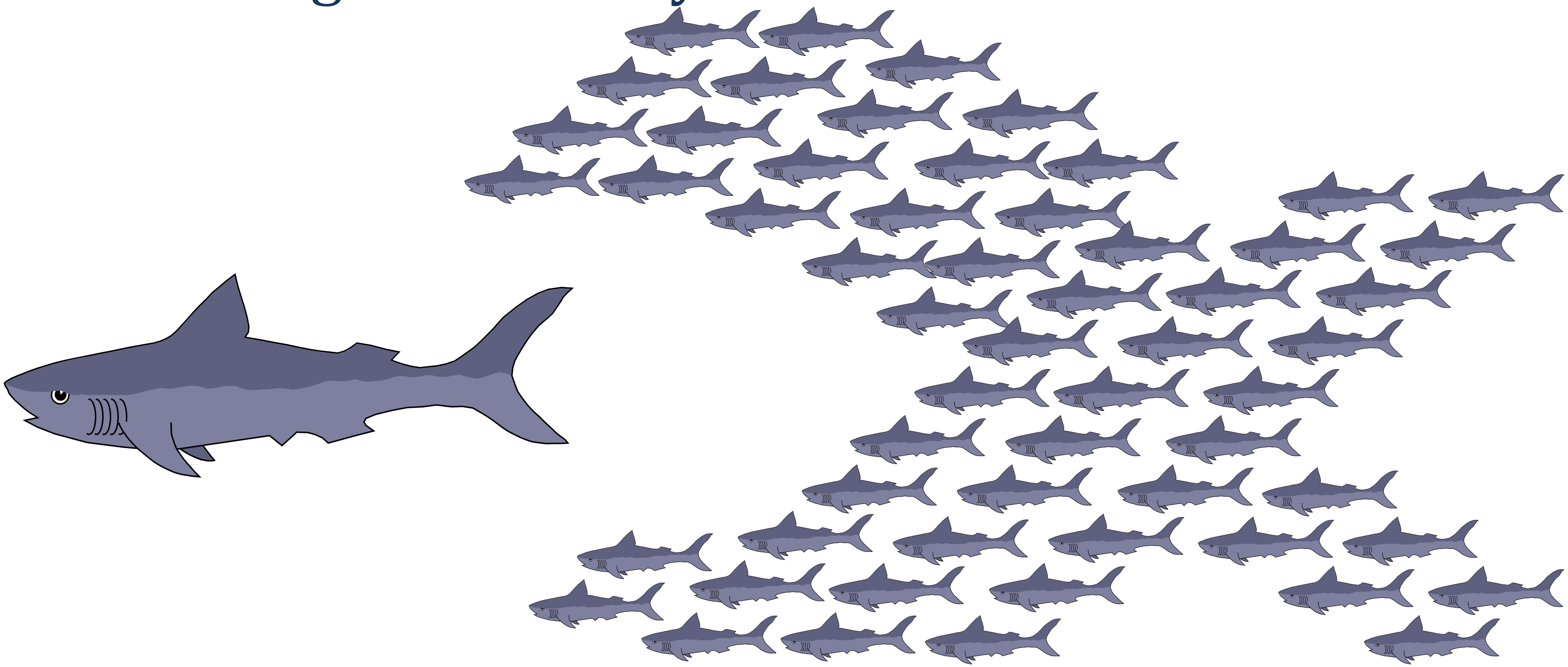
Enabled by:
- Abundant data parallelism
- Power efficient GPUs

Constraint by:
- Programming models
- Comm. overhead

Shader → CUDA →OpenCL …

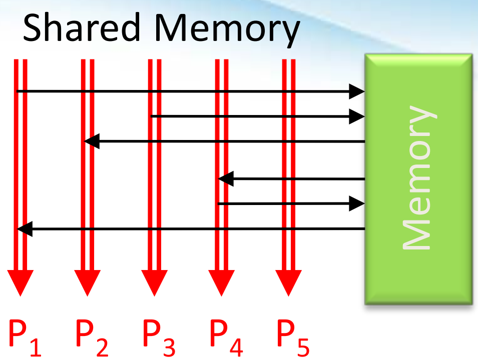
Distributed Computing

- A computer system in which several *interconnected computers* share the computing tasks assigned to the system

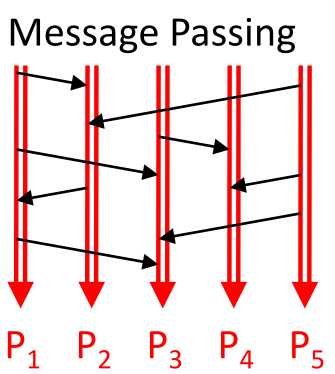


Parallel Programming Models

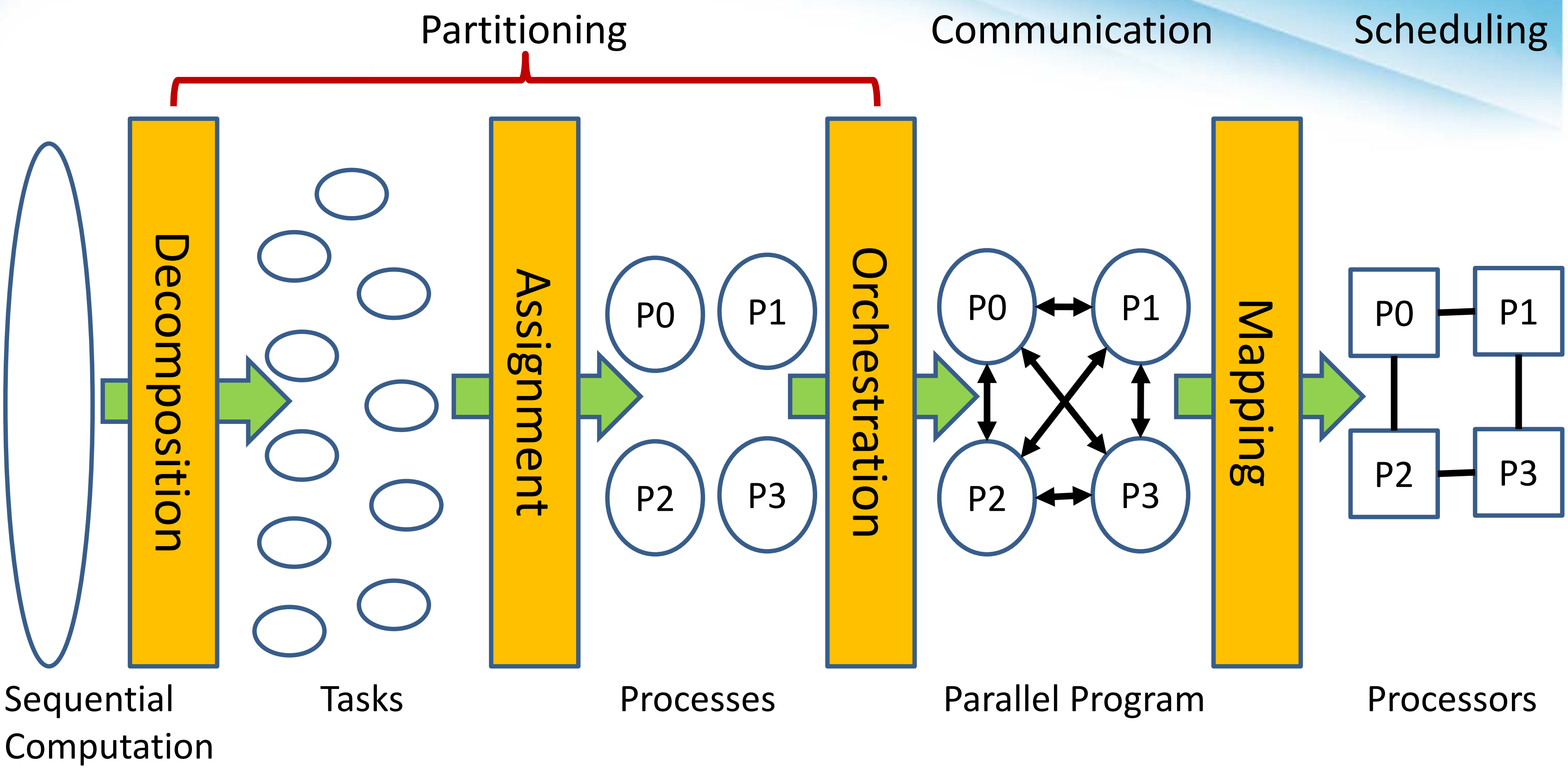
- Shared memory
  - Communicate through shared memory space
  - For multi-core processor
    - Scale-up by adding more cores
  - Languages:
    - Pthread, CUDA, OpenCL



- Message passing
  - Communicate through memory copy
  - For distributed computing
    - Scale-out by adding more nodes
  - Language:
    - MPI, Socket, RPC



4 Common Steps to Create a Parallel Program



Parallelization Challenges

- How do we assign work units to workers? (Partition)
- What if we have more work units than workers? (Scheduling)
- What if workers need to share or aggregate partial results? (Communication)
- How do we know all the workers have finished? (Termination)
- **What if workers die? (Fault Tolerance)**

Common Theme?

- Parallelization problems arise from:
  - Communication between workers (e.g., to exchange state)
  - Access to shared resources (e.g., data)
- Thus, we need a synchronization mechanism

Where the rubber meets the road

- Concurrency is difficult to reason about
- Concurrency is even more difficult to reason about
  - At the scale of datacenters (even across datacenters)
  - In the presence of failures
  - In terms of multiple interacting services
- Not to mention debugging…
- The reality:
  - Lots of one-off solutions, custom code
  - Write you own dedicated library, then program with it
  - Burden on the programmer to explicitly manage everything

What's the point?

- It's all about the right level of abstraction
  - The von Neumann architecture has served us well, but is no longer appropriate for the multi-core/cluster environment
- Hide system-level details from the developers
  - No more race conditions, lock contention, etc.
- Separating the *what* from *how*
  - Developer specifies **what are the computations** need to be performed
  - Execution framework ("runtime") handles **how to execute the computations**

The datacenter *is* the computer!

Solution: Parallel Execution Framework

- Goal:
  - Make it easier for developers to write efficient parallel and distributed applications as **sequential program without considering synchronization or concurrency problem**.
- Approach
  - Define a programming model that explicitly forces developer to consider the data parallelism and data flow of the computation
  - Functional programming meets distributed computing
  - System automatically handle execution problems including resource allocation, scheduling, distribution, and fault tolerance.
- Examples:
  - MapReduce, Dryad, SPARK, GLADE, STORM, CIEL, etc.

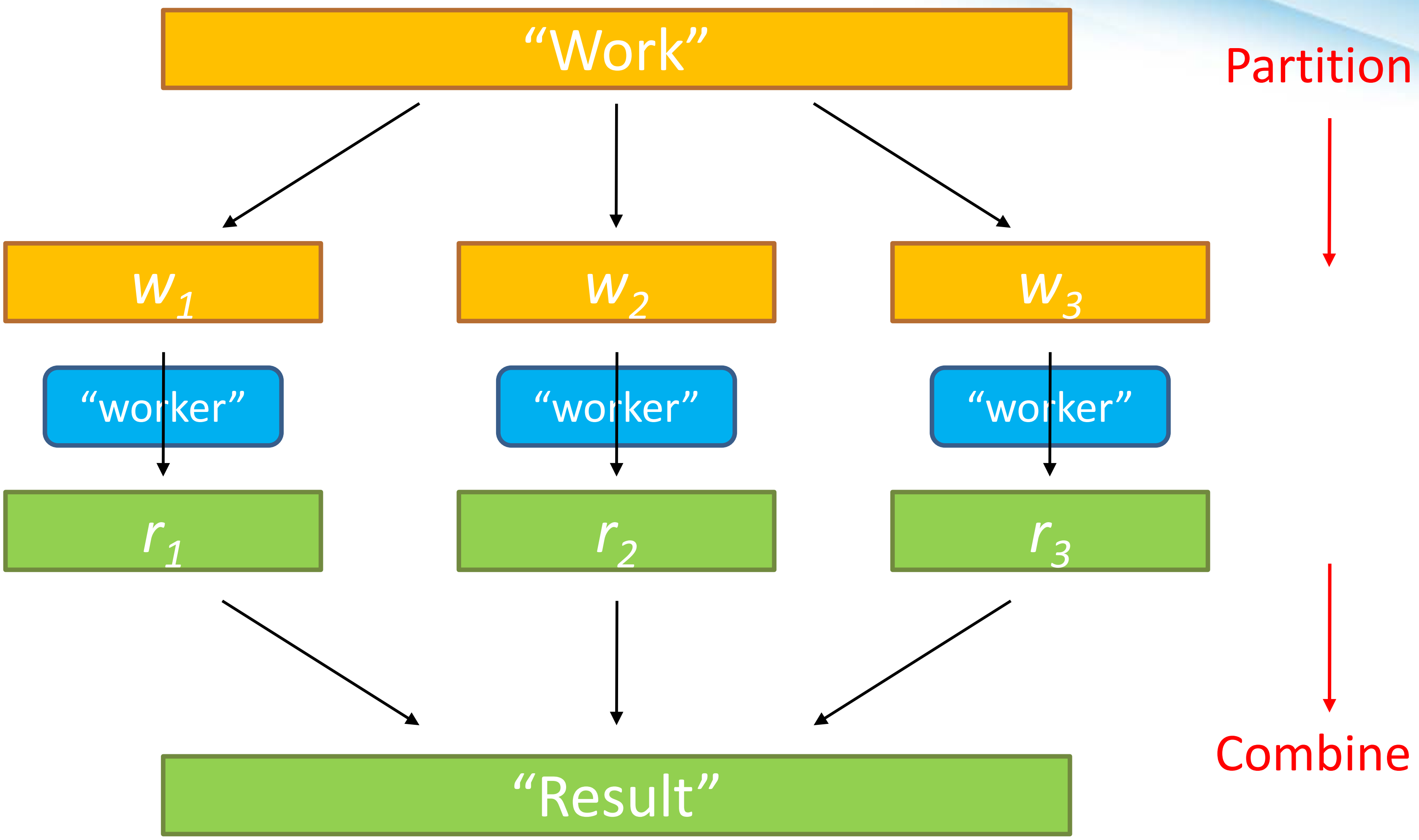
Outline

- Distributed Computing Overview
- MapReduce Framework
- MapReduce(Hadoop) Programming

MapReduce

- Developed by ***Google*** to process PB of data per data using datacenters (published in OSDI'04)
  - Program written in this functional style are **automatically parallelized and executed** on machines
- ***Hadoop*** is the open source (JAVA) implemented by Yahoo
- MapReduce has several meanings
  - A programming model
  - A implementation
  - A system architecture

Start with the Simplest Solution: Divide and Conquer



Typical Large-Data Problem

1. Iterate over a large number of records
2. Extract something of interest from each record
3. Shuffle and sort intermediate results
4. Aggregate intermediate results
5. Generate final output

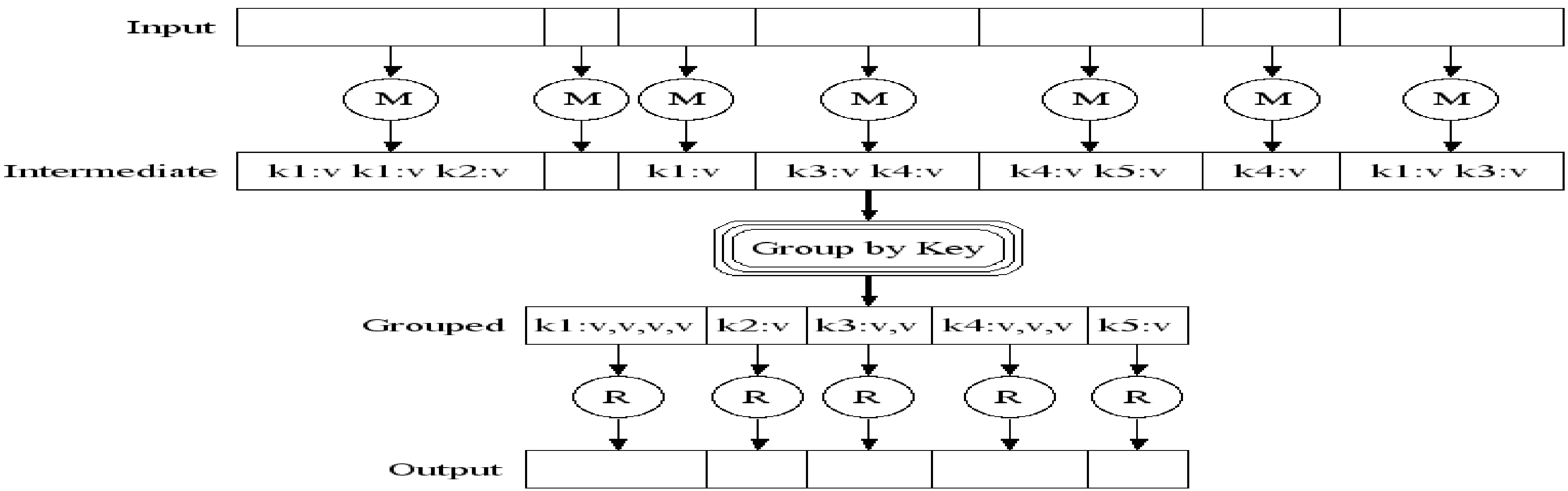
Map

Reduce

Key idea: provide **a functional abstraction** for these two operations

MapReduce Programming Model

- A parallel programming model (divide-conquer)
  - Map: processes a key/value pair to generate a set of intermediate key/value pairs
  - Reduce: merges all intermediate values associated with the same intermediate key



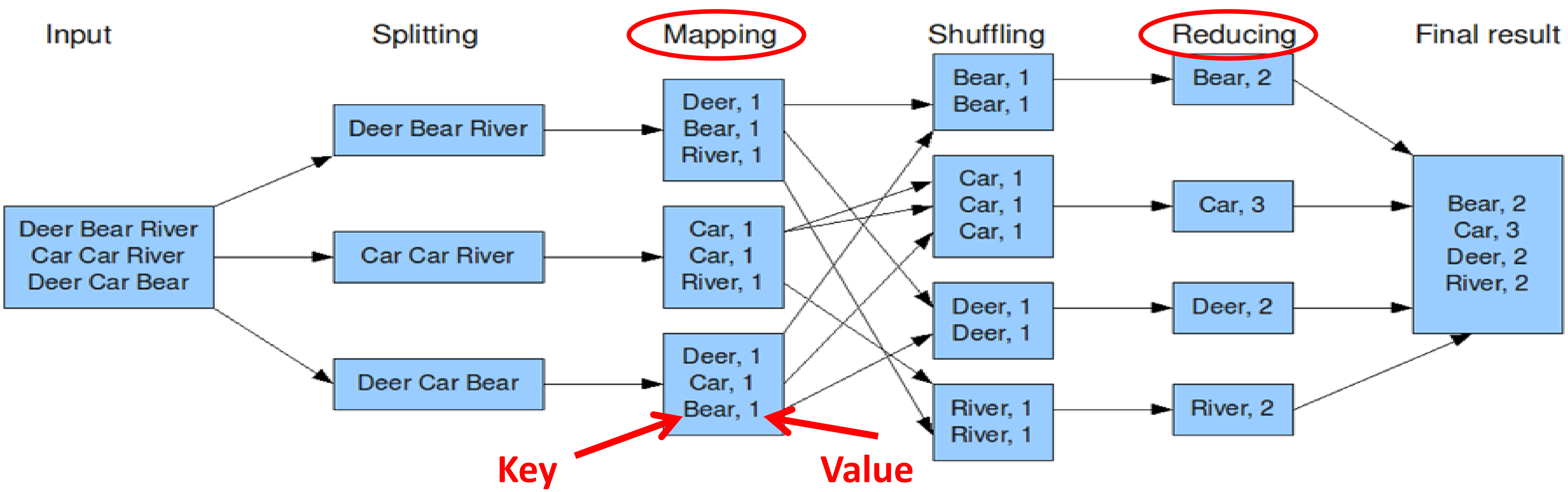
MapReduce Word Count Example

- User specify the ***map*** and ***reduce*** functions

```
Map(String docid, String text):
    for each word w in text:
        Emit(w, 1);
```

```
Reduce(String term, Iterator<Int> values):
    int sum = 0;
    for each v in values:
        sum += v;
        Emit(term, value);
```

The overall MapReduce word count process



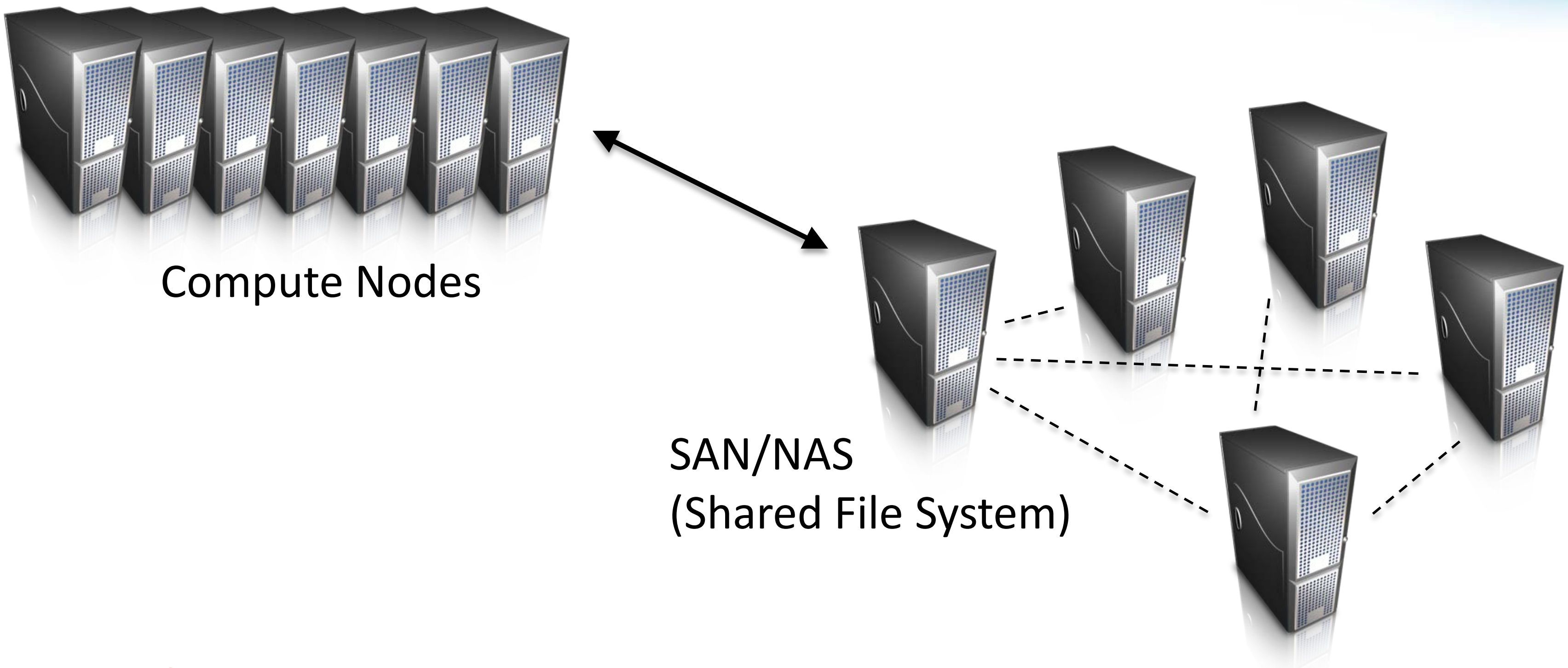
- The execution framework handles everything else…

What's "everything else"?

MapReduce “Runtime”

- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles “data distribution”
  - Moves processes to data
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects worker failures and restarts
- Everything happens on top of a **distributed FS**

How do we get data to the workers?



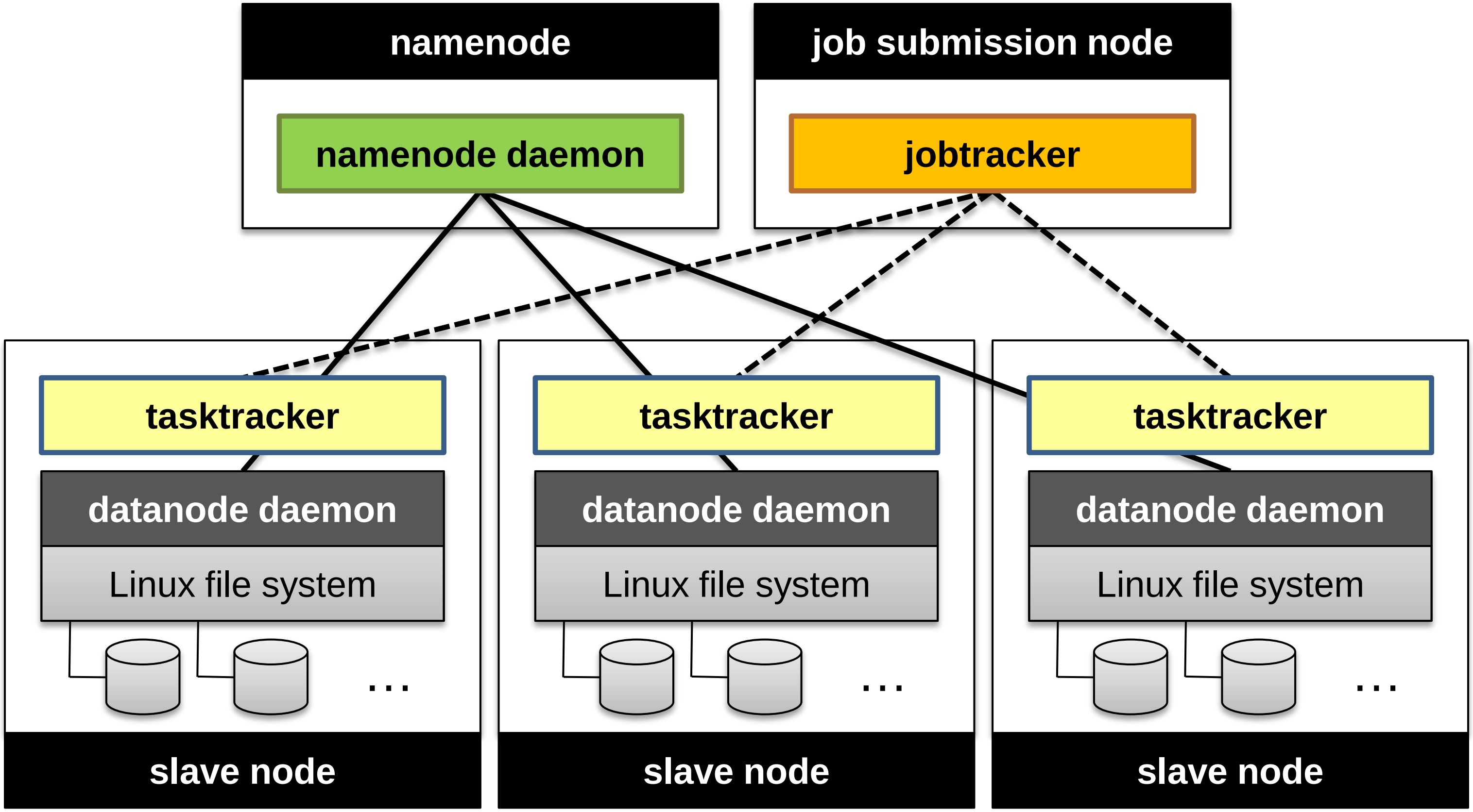
What's the problem here?

Distributed File System

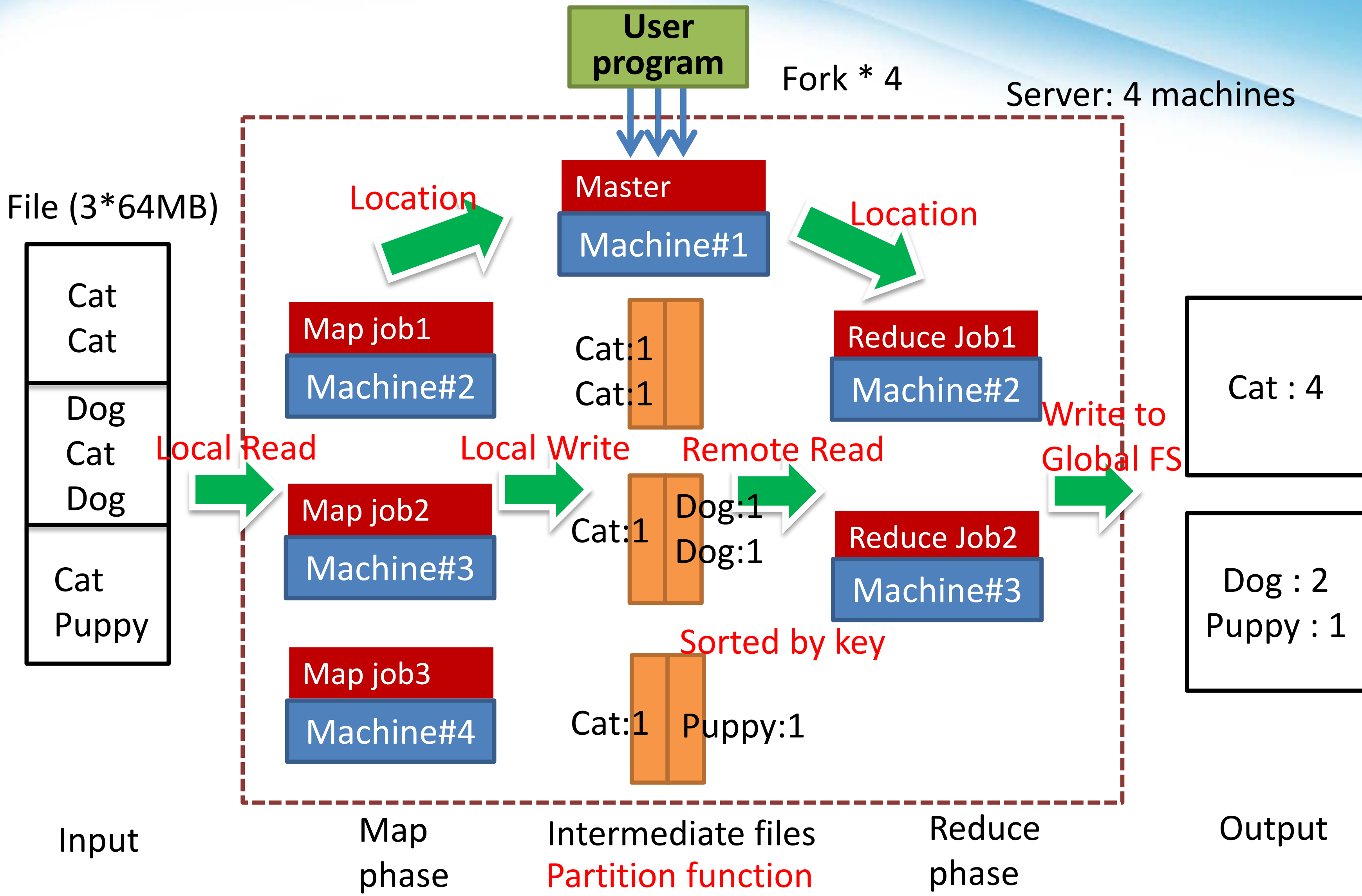
- Don't move data to workers…
  move workers to the data!
  - A node act as both compute and storage node
  - Store data on the local disks of nodes in the cluster
  - Start up the workers on the node that has the data local
- A distributed file system is the answer
  - GFS (Google File System) for Google's MapReduce
  - HDFS (Hadoop Distributed File System) for Hadoop

Putting everything together…

- Hadoop:
  - Namenode (Master in GFS): file metadata server
  - Job/Task tracker: MapReduce engine



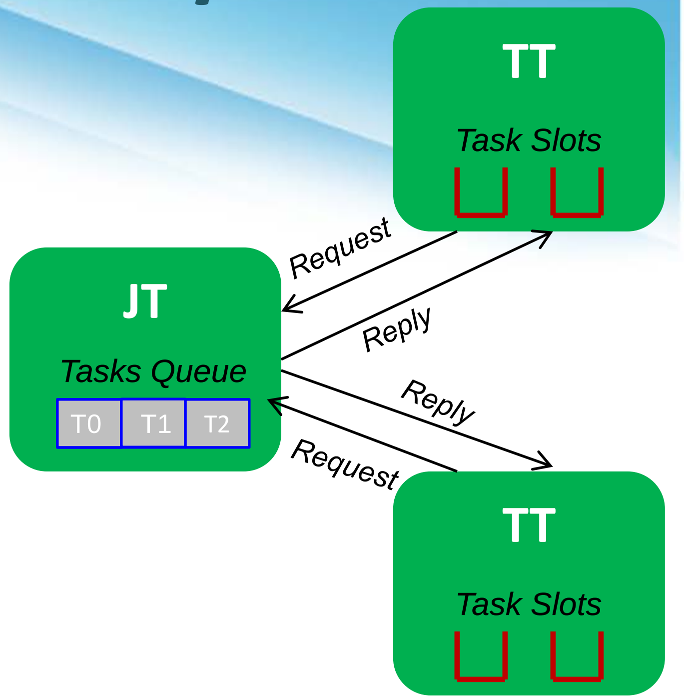
MapReduce in Action



Job Scheduling in MapReduce

- In MapReduce, an application is represented as a *job*
- A job encompasses multiple map and reduce tasks
- MapReduce in Hadoop comes with a choice of schedulers:
  - The default is the *FIFO scheduler* which schedules jobs in order of submission
  - There is also a multi-user scheduler called the *Fair scheduler* which aims to give every user a fair share of the cluster capacity over time

Task Scheduling in MapReduce



- MapReduce adopts a *master-slave architecture*
- The master node in MapReduce is referred to as *Job Tracker* (JT)
  - Implement a scheduler
- Each slave node in MapReduce is referred to as *Task Tracker* (TT)
  - Has a fixed number of mapper slots and reducer slots
- MapReduce adopts a *pull scheduling* strategy rather than a *push one*:
  - Triggered by the heartbeat message from task tracker

Map and Reduce Task Scheduling

- Every TT sends a *heartbeat message* periodically to JT encompassing a request for a map or a reduce task to run

I. Map Task Scheduling:
   - JT satisfies requests for map tasks via attempting to schedule mappers in the *vicinity* of their input splits (i.e., it considers locality)
   - Multiple level of locality: node level, rack level, datacenter level

II. Reduce Task Scheduling:
   - However, JT simply assigns the next yet-to-run reduce task to a requesting TT regardless of TT’s network location and its implied effect on the reducer’s shuffle time (i.e., it does not consider locality)

Fault Tolerance in Hadoop

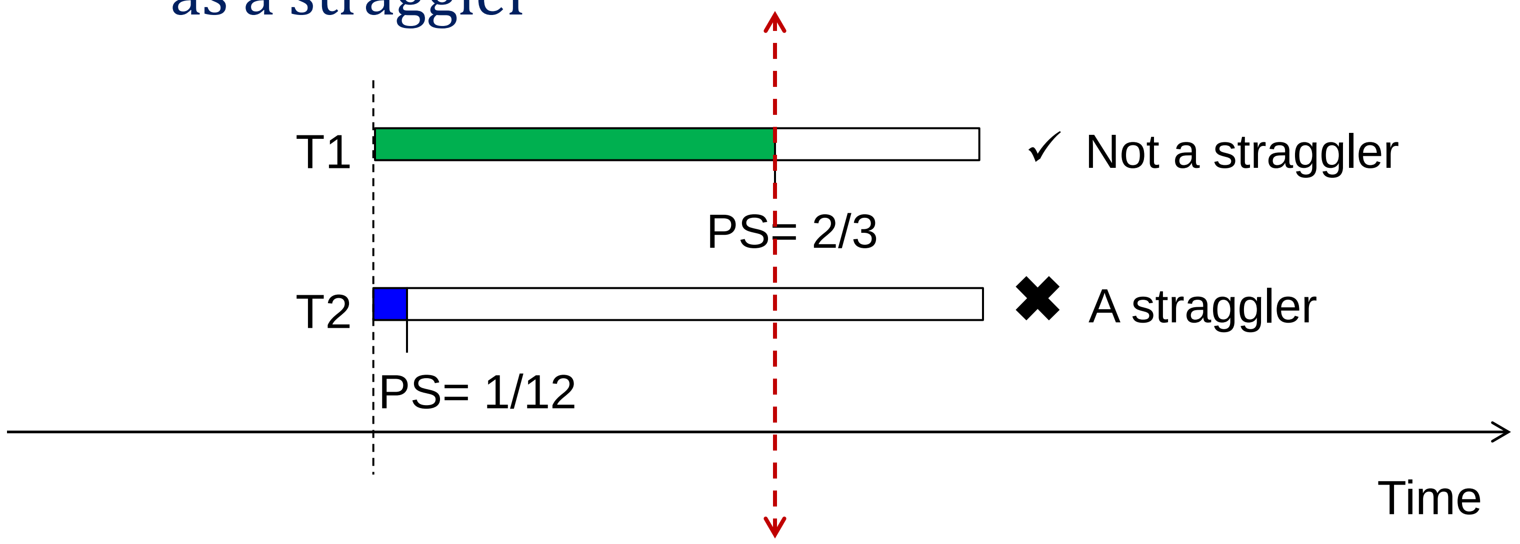
- MapReduce can guide jobs toward a successful completion even when jobs are run on a large cluster where probability of failures increases
- The primary way that MapReduce achieves fault tolerance is through *restarting tasks*
- If a TT fails to communicate with JT for a period of time (by default, 1 minute in Hadoop), JT will assume that TT in question has crashed
  - If the job is still in the map phase, JT asks another TT to re-execute *all Mappers that previously ran at the failed TT*
  - If the job is in the reduce phase, JT asks another TT to re-execute *all Reducers that were in progress on the failed TT*

Speculative Execution

- A MapReduce job is dominated by the slowest task
- MapReduce attempts to locate slow tasks (*stragglers*) and run redundant (*speculative*) tasks that will optimistically commit before the corresponding stragglers
- This process is known as *speculative execution*
- Only one copy of a straggler is allowed to be speculated
- Whichever copy (among the two copies) of a task commits first, it becomes the definitive copy, and the other copy is killed by JT

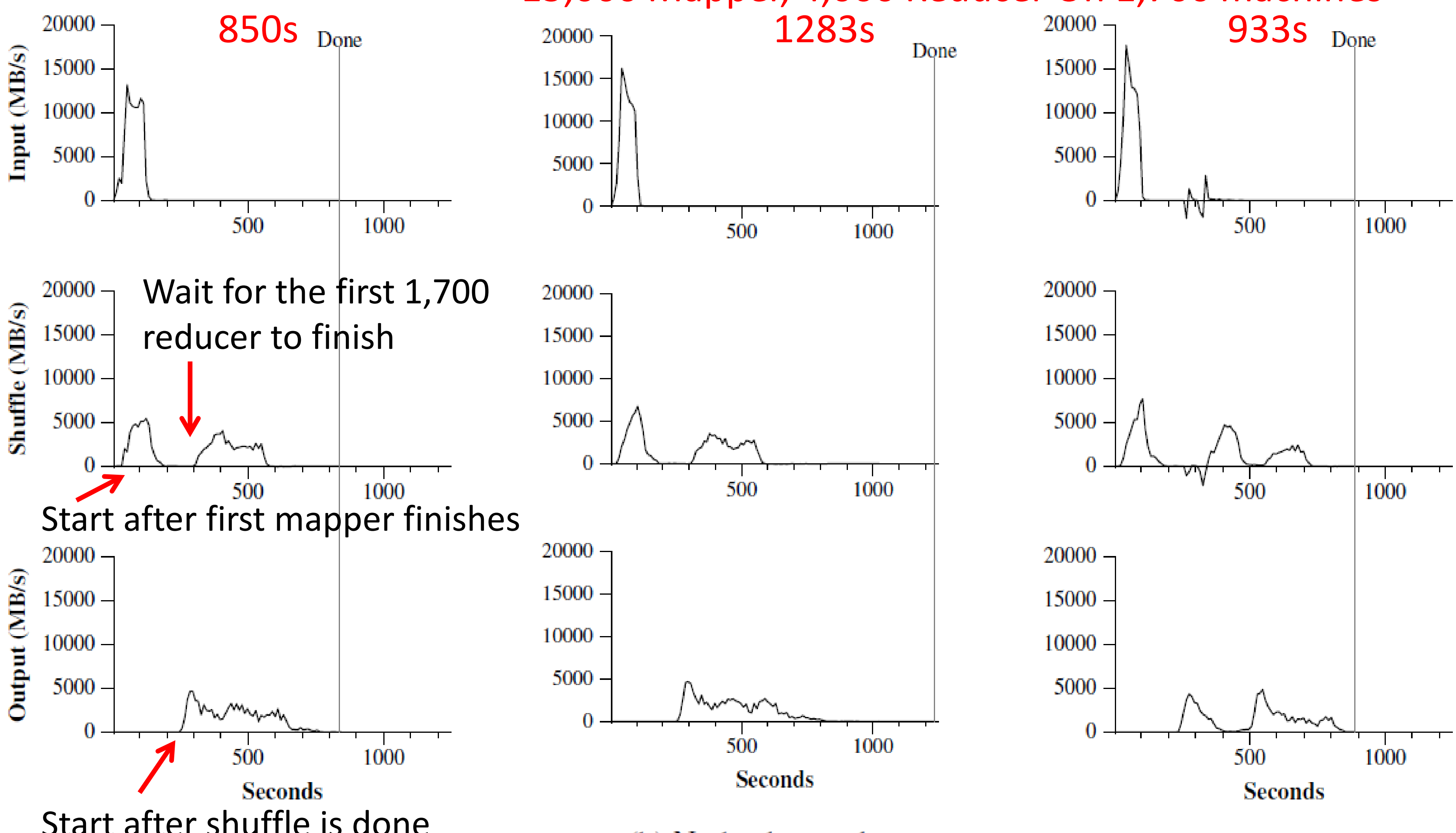
Locating Stragglers

- How does Hadoop locate stragglers?
  - Hadoop monitors each task progress using a *progress score* between 0 and 1
  - If a task's progress score ***is less than*** (average – 0.2), and the task has run for at least 1 minute, it is marked as a straggler



Performance

15,000 Mapper, 4,000 Reducer On 1,700 machines



(a) Normal execution

(b) No backup tasks

(c) 200 tasks killed

Comparison With Traditional Models

Aspect	Shared Memory	Message Passing	MapReduce
Communication	Implicit (via loads/stores)	Explicit Messages	Limited and Implicit
Synchronization	Explicit	Implicit (via messages)	Immutable (K, V) Pairs
Hardware Support	Typically Required	None	None
Development Effort	Lower	Higher	Lowest
Tuning Effort	Higher	Lower	Lowest

What Makes MapReduce Unique?

- MapReduce is characterized by:

  1. Its simplified programming model which allows the user to quickly write and test distributed systems

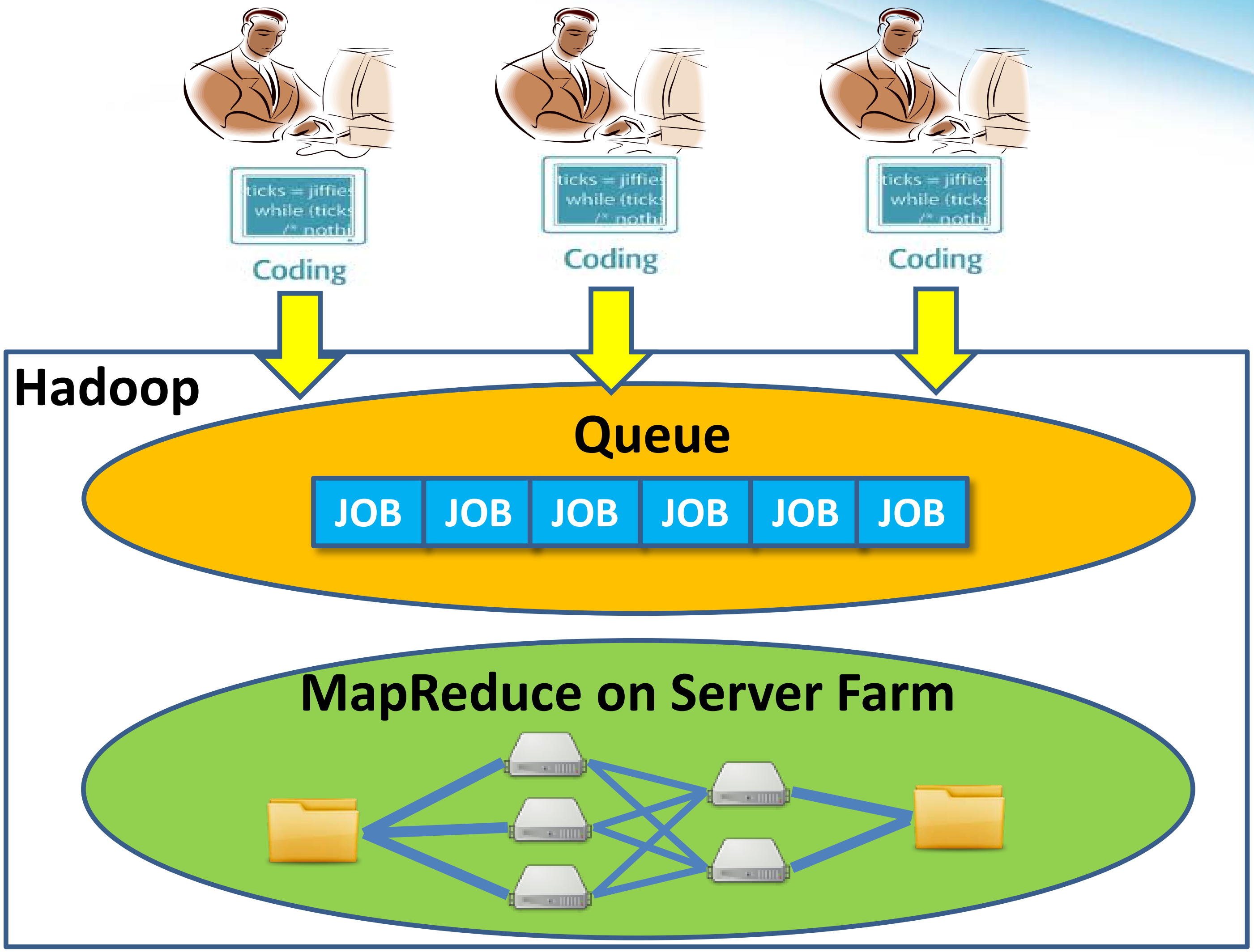
  2. Its efficient and automatic distribution of data and workload across machines. Moving process to data.

  3. Seamless scalability. Specifically, after a Mapreduce program is written and functioning on 10 nodes, very little-if any- work is required for making that same program run on 1000 nodes

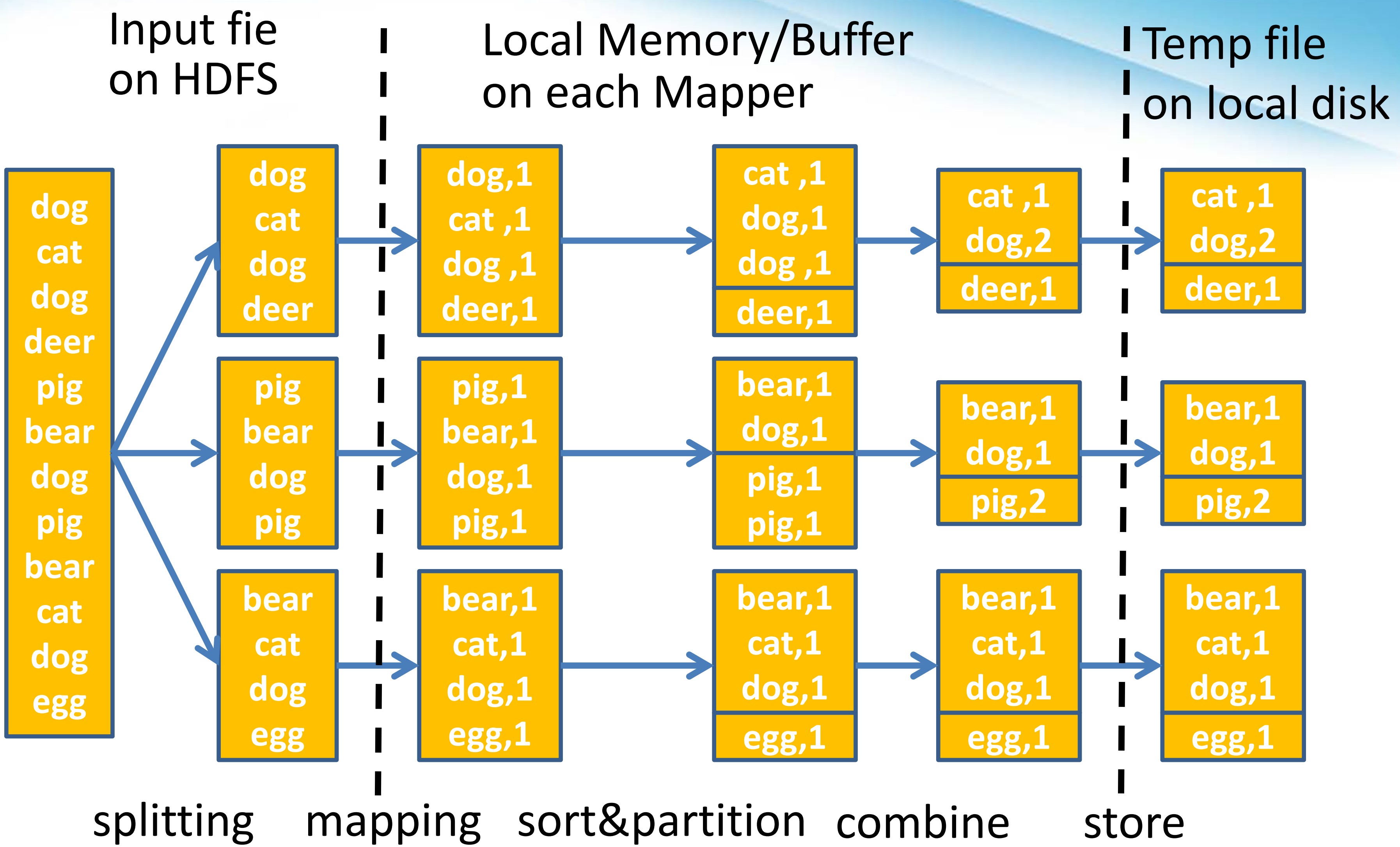
Outline

- Distributed Computing Overview
- MapReduce Framework
- MapReduce(Hadoop) Programming
  - Hadoop MapReduce Overview
  - Hadoop Job Configuration
  - Data Format and Data Type
  - Hadoop MapReduce Process
  - Performance Profiling & Tuning

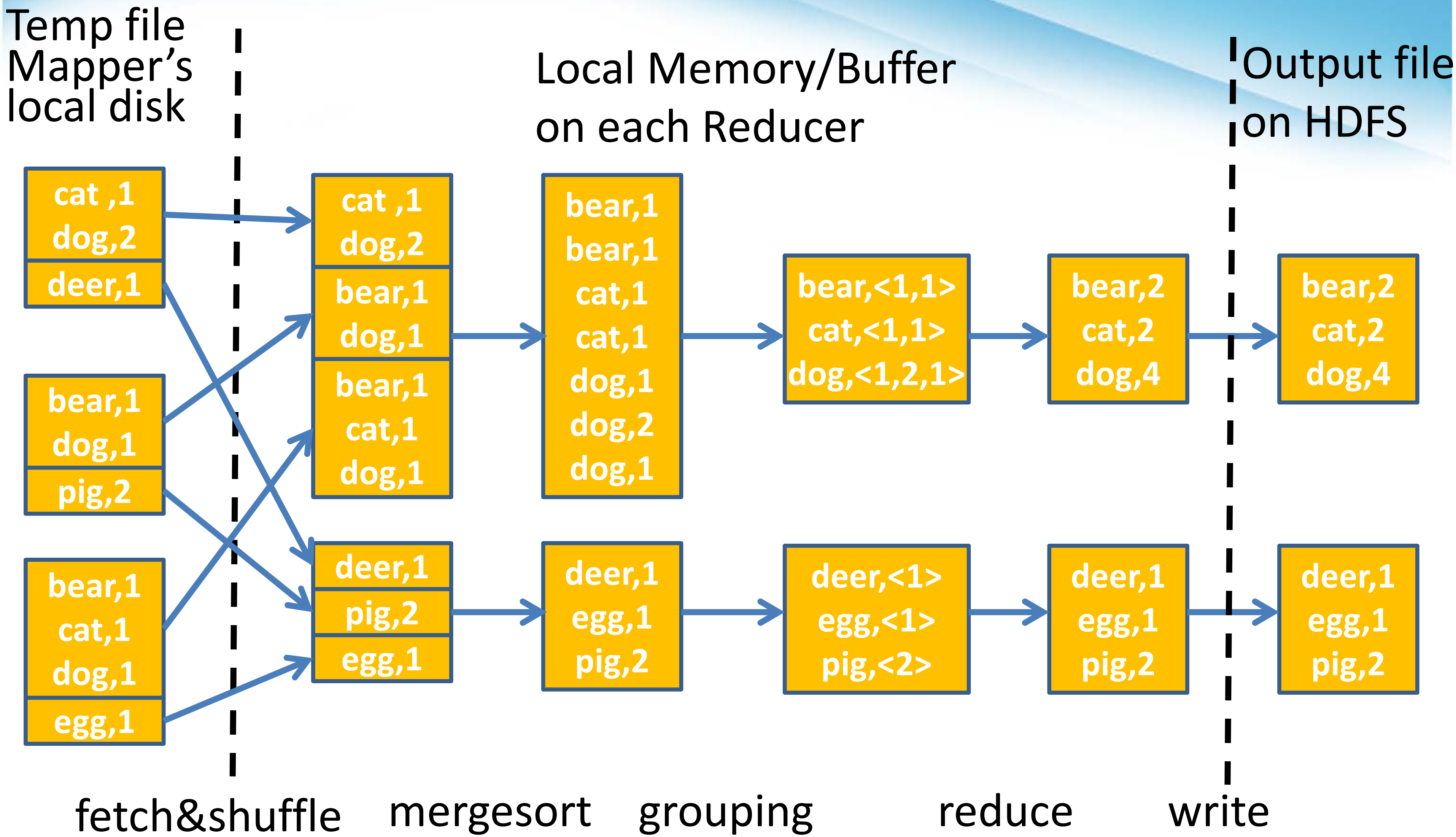
Hadoop Platform



Map Phase


Input fie
on HDFS
Local Memory/Buffer
on each Mapper
Temp file
on local disk
dog cat dog deer pig bear dog pig bear cat dog egg
dog cat dog deer
pig bear dog pig
bear cat dog egg
dog,1 cat ,1 dog ,1 deer,1
pig,1 bear,1 dog,1 pig,1
bear,1 cat,1 dog,1 egg,1
cat ,1 dog,1 dog ,1 | deer,1
bear,1 dog,1 | pig,1 pig,1
bear,1 cat,1 dog,1 | egg,1
cat ,1 dog,2 | deer,1
bear,1 dog,1 | pig,2
bear,1 cat,1 dog,1 | egg,1
cat ,1 dog,2 | deer,1
bear,1 dog,1 | pig,2
bear,1 cat,1 dog,1 | egg,1
splitting
mapping
sort&partition
combine
store

Reduce Phase



Hadoop Implementation

- Hadoop release ***1.0.3***
  - ***More compatible with Hbase, Hive, etc.***
- Java Language
  - Based on inheritance and interface
- API in "***org.apache.hadoop.mapred***" package
  - Not interchangeable with the "***org.apache.hadoop.mapreduce***" package
- Official Tutorial
  - https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html

Important Hadoop Classes

- **JobConf**
  - the primary interface for a user to describe a map-reduce job to the Hadoop framework for execution
- **Mapper**
  - maps input <K,V> pairs to intermediate <K,V> pairs
- **Reducer**
  - reduces intermediate values to a smaller set of values
- **Partitioner**
  - partitions the key of intermediate <K,V> pairs to reducer
- **Combiner**
  - combine map-outputs <K,V> pairs before being sent to reducers
- **RecordReader/RecordWriter**
  - Read input file & write output file
- **Reporter**
  - A facility for Map-Reduce applications to report progress and update counters, status information etc

Outline

- Hadoop MapReduce Overview
- Hadoop Job Configuration
- Data Format and Data Type
- Hadoop MapReduce Main Components
- Performance Profiling & Tuning

JobConf: WordCount Example

```
public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");

    conf.setMapperClass(Map.class);
    conf.setReducerClass(Reduce.class);
    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));
    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    JobClient.runJob(conf);
}
```

JobConf Class

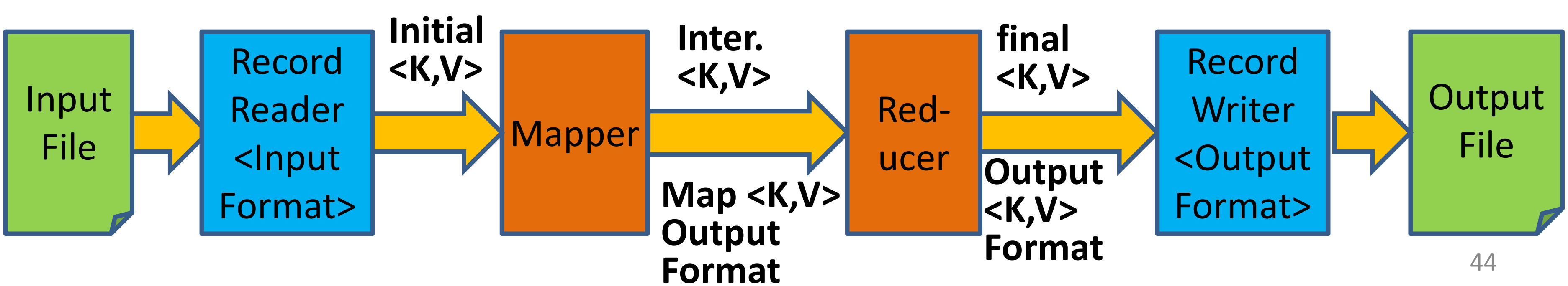
Method	Description
setJobName(String name)	Set the user-specified job name
setJobPriority(JobPriority prio)	High/Low/Normal/Very_High/Very_Low
setQueueName (String queueName)	Set the name of the queue to which this job should be submitted.
setNumMapTasks(int n)	Set the number of map tasks for this job.
setNumReduceTasks(int n)	Set the requisite number of reduce tasks for this job.
setMaxMapAttempts(int n)	Set the number of maximum attempts that will be made to run a map task.
setCompressMapOutput (boolean compress)	Should the map outputs be compressed before transfer?
setMapDebugScript (String mDbgScript)	Set the debug script to run when the map tasks fail.

JobConf Class

Method	Description
setMapperClass (Class<? extends Mapper> theClass)	Set the Mapper class for the job
setReducerClass (Class<? extends Reducer> theClass)	Set the Reducer class for the job.
setPartitionerClass (Class<? extends Partitioner> theClass)	Set the Partitioner class used to partition Mapper-outputs to be sent to the Reducers.
setCombinerClass (Class<? extends Reducer> theClass)	combine map-outputs before being sent to the reducer
setOutputKeyComparatorClass (Class<? extends RawComparator> theClass)	Set the RawComparator comparator used to compare keys.
setOutputValueGroupingComparator (Class<? extends RawComparator> theClass)	Set the user defined RawComparator comparator for grouping keys in the input to the reduce.

JobConf Class

		Description
Void	**setInputFormat** (Class<? extends InputFormat> theClass)	Set the InputFormat implementation for the map-reduce job
Void	**setOutputFormat** (Class<? extends OutputFormat> theClass)	Set the OutputFormat implementation for the map-reduce job.
Void	**setOutputKeyClass**(Class<?> theClass)	Set the key class for the job output data.
Void	**setOutputValueClass**(Class<?> theClass)	Set the value class for job outputs.
Void	**setMapOutputKeyClass**(Class<?> theClass)	Set the key class for the map output data. Same type as final output if not specify
Void	**setMapOutputValueClass**(Class<?> theClass)	Set the value class for the map output data Same type as final output if not specify



Outline

- Hadoop MapReduce Overview
- Hadoop Job Configuration
- Data Format and Data Type
  - Data Input/Output Format
  - Key-Value Pair Data Type
- Hadoop MapReduce Process
- Performance Profiling & Tuning

Input/Output Format Class

- The MapReduce operates **exclusively** on <K, V> pairs
  - It views the job input as a set of <key, value> pairs and produces a set of <key, value> pairs as the output of the job
- InputFormat: parse input file into a set of <key, value>
  - **TextInputFormat**: Keys are the position in the file, and values are the line of text.
  - **KeyValueTextInputFormat:** Each line is divided into key and value parts by a separator byte. If no such a byte exists, the key will be the entire line and value will be empty.
- OutputFormat: write a set of <key, value> to output file
  - TextOutputFormat<K, V>: writes plain text: key, value, and "\r\n".

Key-Value Pair Class

- Hadoop MapReduce **uses typed data at all times** when it interacts with user-provided Mappers and Reducers
- ***Key***: Instance of *WritableComparable*, because key will be sorted by the framework
- ***Value***: Instance of *Writable*, because value must be mutable by the framework
- WritableComparable interface
  - BooleanWritable, BytesWritable, DoubleWritable, FloatWritable, IntWritable, LongWritable, Text, NullWritable
- Writable but not WritableComparable
  - ArrayWriable

Custom Value Types

- *Value in* 3-dimensional coordinate

  struct point3d { float x; float y; float z; }

- Implement *Writable* interface

```
public class Point3D implements Writable {
    public float x; public float y; public float z;
    public Point3D(float x, float y, float z)  { this.x = x; this.y = y; this.z = z; }
    public void write(DataOutput out) throws IOException {
            out.writeFloat(x); out.writeFloat(y); out.writeFloat(z);
    }
    public void readFields(DataInput in) throws IOException
            { x = in.readFloat(); y = in.readFloat(); z = in.readFloat();
    public String toString()  {
            return Float.toString(x)+","+ Float.toString(y)+","+Float.toString(z);}
}
```

Call when writing to file

Call when reading from file

Custom Key Types

- *Key in* 3-dimensional coordinate

  struct point3d { float x; float y; float z; }

- Implement all functions in the ***writable*** interface
  - write(), readFields()
- Implement additional functions in the ***writablecomparable*** interface
  - compareTo(): used for **sorting**
  - equals(): used for **grouping**
  - hashCode(): used for **partitioning**

```
public class Point3D implements WritableComparable {
    public float distanceFromOrigin() {
        return (float)Math.sqrt(x*x + y*y + z*z);
    }
    public int compareTo(Point3D other) {
        float myDistance = distanceFromOrigin();
        float otherDistance = other.distanceFromOrigin();
        return Float.compare(myDistance, otherDistance);
    }
    public boolean equals(Object o) {
        Point3D other = (Point3D)o;
        return this.x == other.x && this.y == other.y && this.z ==
other.z; }
    public int hashCode() {
        return Float.floatToIntBits(x) ^ Float.floatToIntBits(y) ^
        Float.floatToIntBits(z);
    }
}
```

Use Case Example

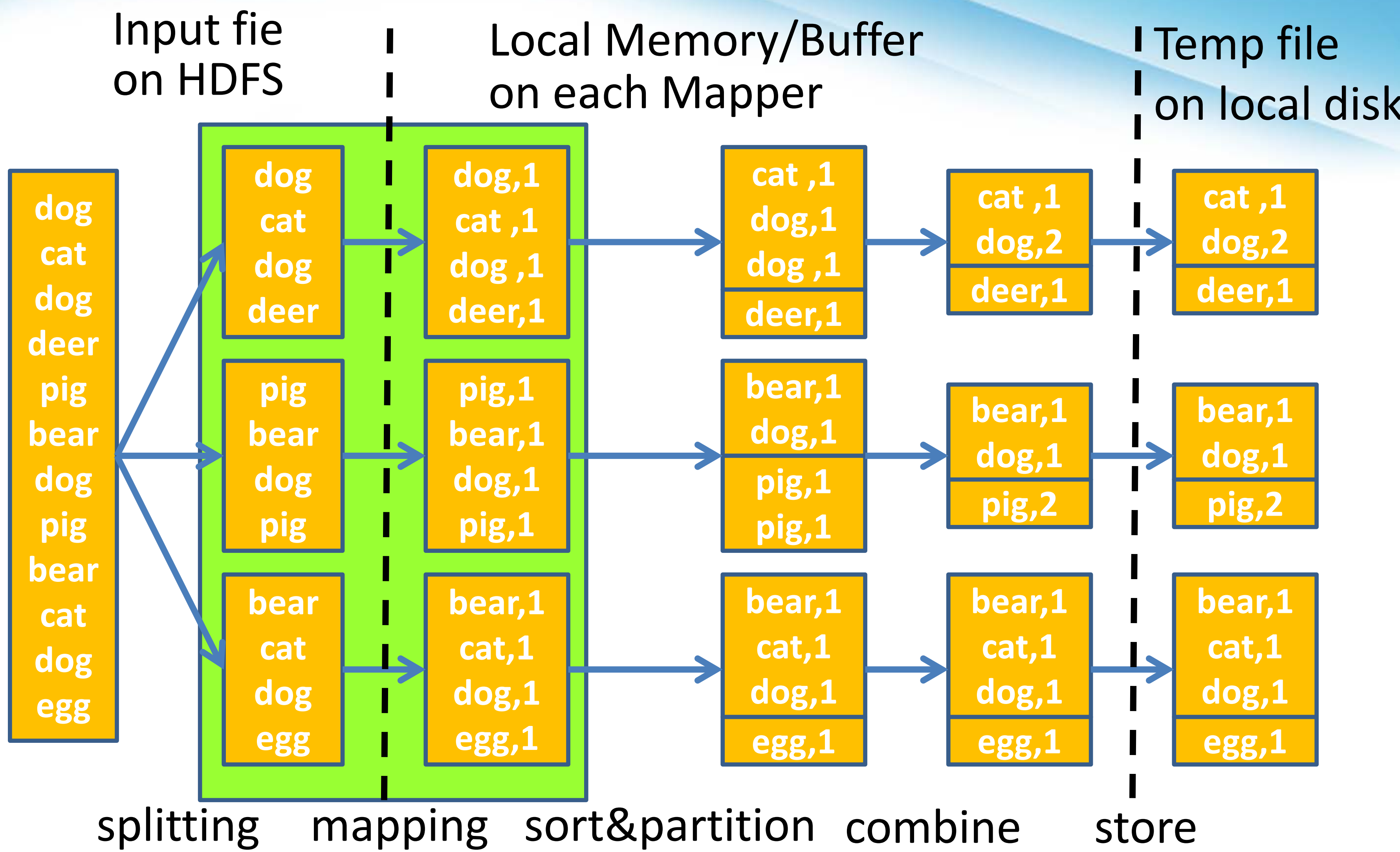
- Given a list of 3D-coordinates, sort them in order in each of the output file:
  - **key type**: Point3D
  - **Value type**: NullWritable
  - **Mapper**: map each line to {<x,y,z>, Null}
  - **Reducer**: write key to file

➔Data is sorted automatically by Key in the MapReduce process

Outline

- Hadoop MapReduce Overview
- Hadoop Job Configuration
- Data Format and Data Type
- Hadoop MapReduce Process
- Performance Profiling & Tuning

Map Phase: Mapper



Map Phase: Mapper

- Mapper maps input key/value pairs to a set of intermediate key/value pairs
  - The transformed intermediate records do **NOT** need to be the same type as the input records.
  - A given input pair may map to **zero** or **many** output pairs.
- Each key/value pair is applied with a map function:
  - **map**(WritableComparable, Writable, OutputCollector, Reporter)
  - **<WritableComparable, Writable>** are the input key-value pairs generated by the **InputFormat class**
  - **outputCollector.collect(K, V)** collects output key-value pairs

Map Phase: Mapper

- Constructor:
  - MapReduceBase is a class
  - Mapper is a interface
  - <LongWritable, Text, Text, IntWritable> is the **generic type of the interface** to prevent runtime type casting error

```
public static class Map extends MapReduceBase implements
  Mapper<LongWritable, Text, Text, IntWritable> {
  ..............................
}
```

Type of input Key&value from the InputFormat

Type of output Key&value from the Mapper

Mapper: WodCount Example

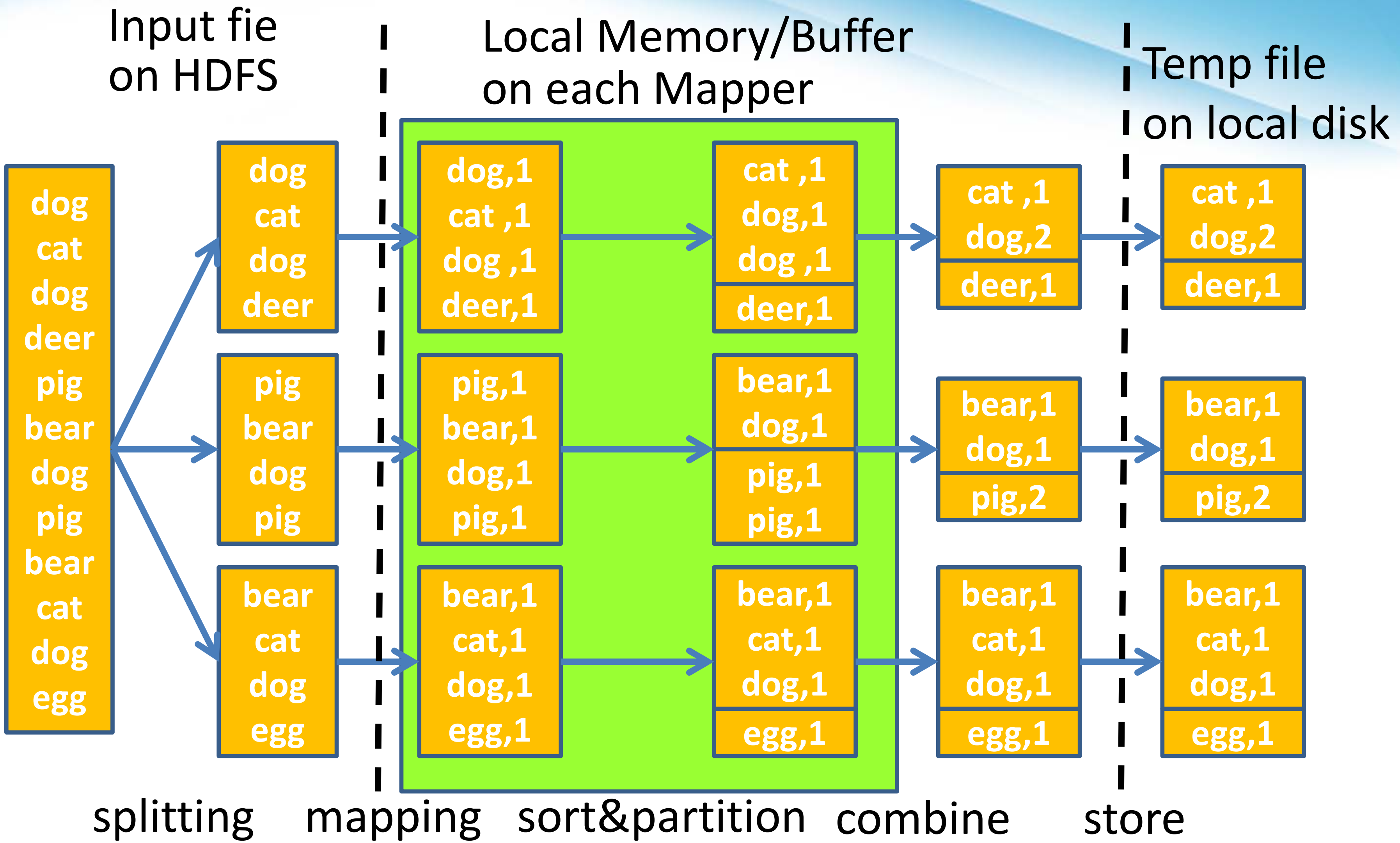
```
public static class Map extends MapReduceBase implements
  Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value,
                    OutputCollector<Text, IntWritable> output,
                    Reporter reporter) throws IOException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {   // each line has multiple words
                word.set(tokenizer.nextToken());
                output.collect(word, one);
        }
    }
}
```

Recall JobConf setting:

```
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(IntWritable.class);
conf.setMapperClass(Map.class);
conf.setInputFormat(TextInputFormat.class);
```

Map Phase: Partitioner



Map Phase: Partitioner

- Partitioner decides which intermediate (K,V) pair is sent to which reducer
- The total number of partitions is the same as the number of reduce tasks for the job.
- Default partitioner: "**HashPartitioner**"
- Write a custom Partitioner:

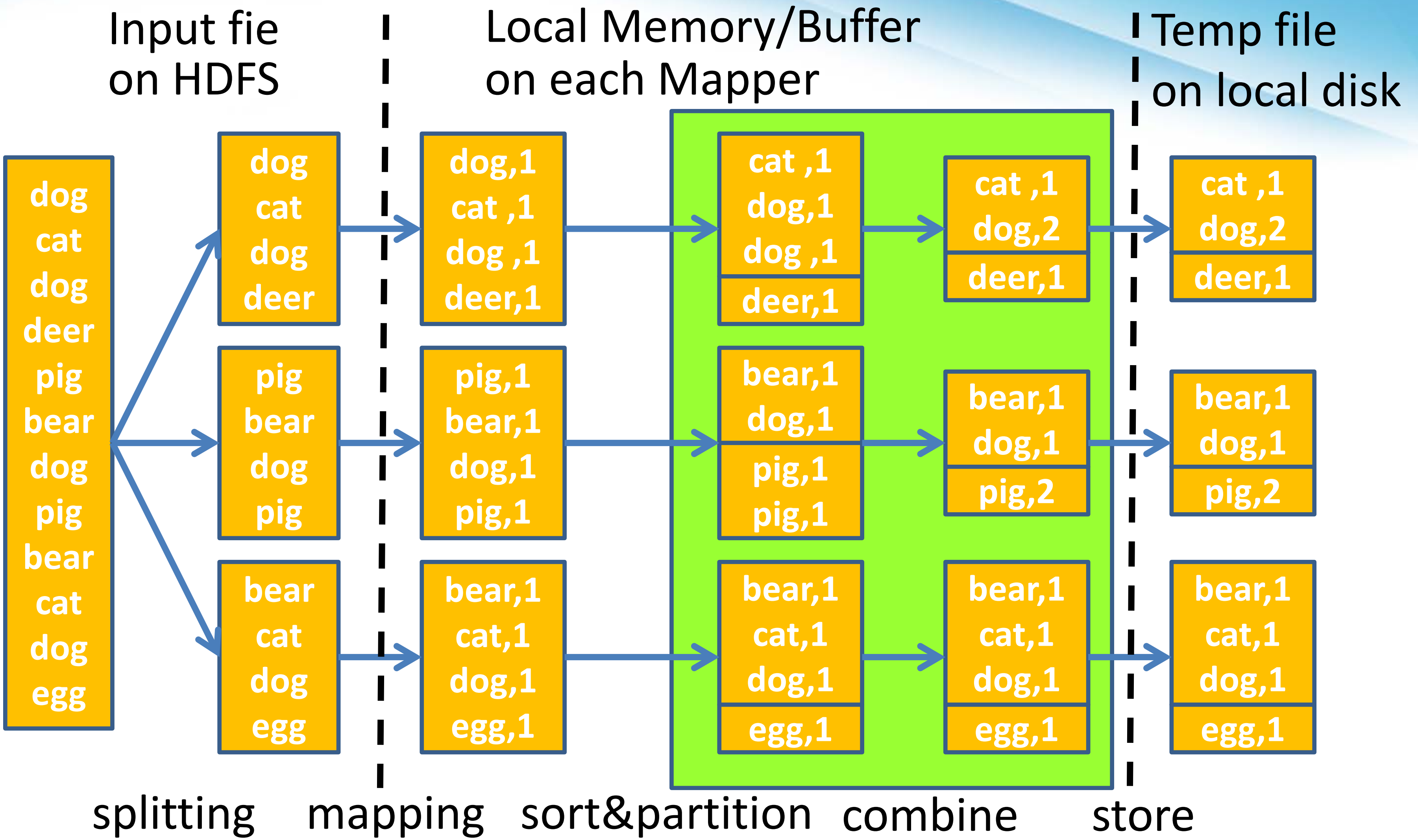
```
public class MyPartitioner implements Partitioner<Point3D, Writable> {
    public int getPartition(TextPair key, Writable value, int numPart) {
        return Math.abs(key.hashCode()) % numPart;
    }
}
public void configure(JobConf job) { }
```

Must have it!!!

JobConf setting:

```
conf.setPartitionerClass(MyPartitioner.class);
```

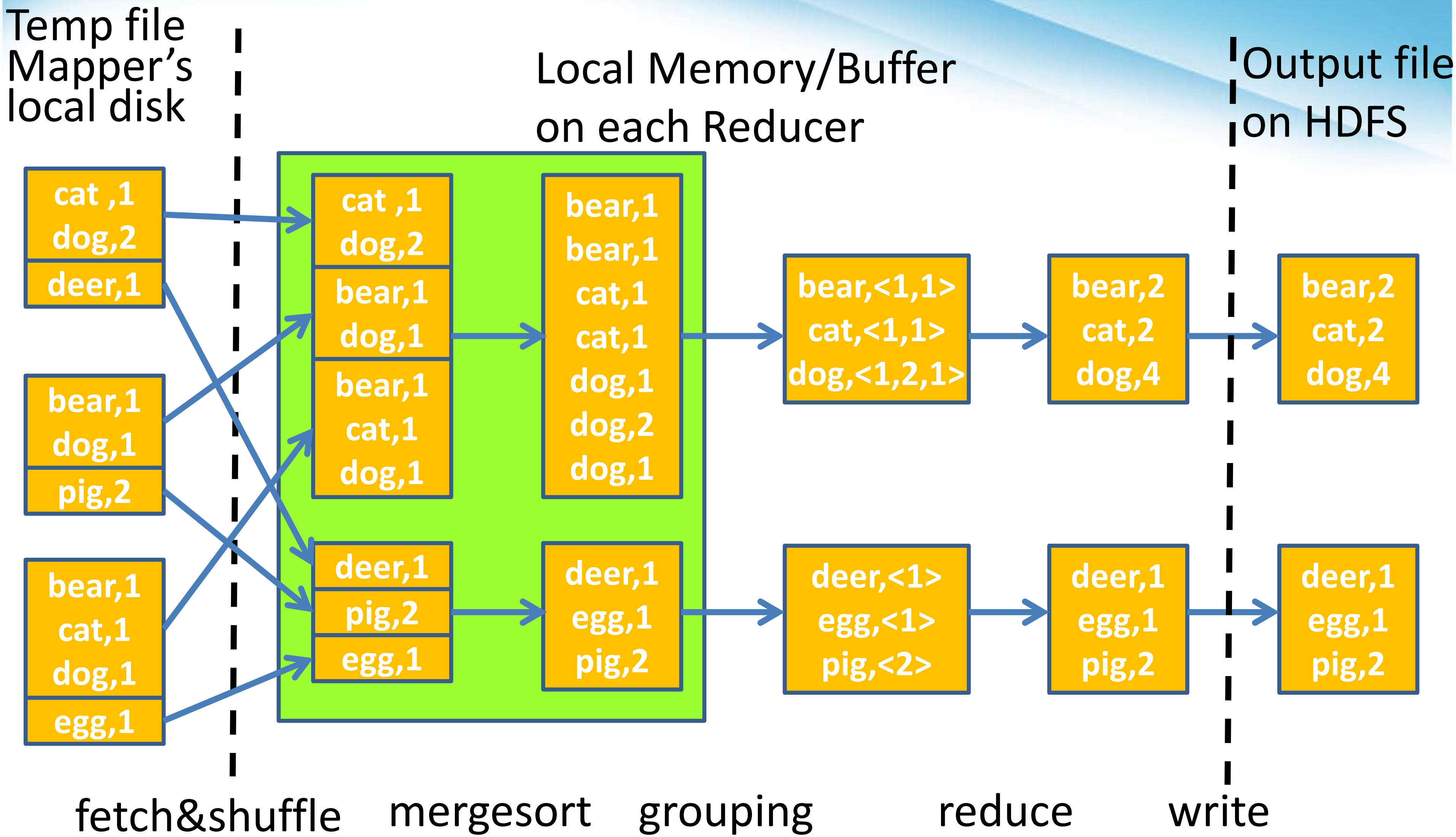
Map Phase: Combiner



Combiner

- An OPTIONAL **optimization** step in mapping phase
  - Combiner combines map-outputs before being sent to the reducers ➔ reduce intermediate file size and transfer time
  - Combiner could be run **many** or **ZERO** time ➔ program results can't depend on combiner
  - <K,V> data type must be the **same** for INPUT & OUTPUT
    - Reducer can emit a different output type to file
  - Reducer and combiner could be but NOT ALWAYS the same
    - E.g.: compute the avg of each key
    - MEAN({1,2,3,4,5}) ≠ MEAN(MEAN({1,2}),MEAN({3,4,5}))
  - Some problem can be difficult to apply combiner
    - E.g.: Find the median value of each key

Reduce Phase: Merge&Sort



Mergesort: OutputKeyComparator

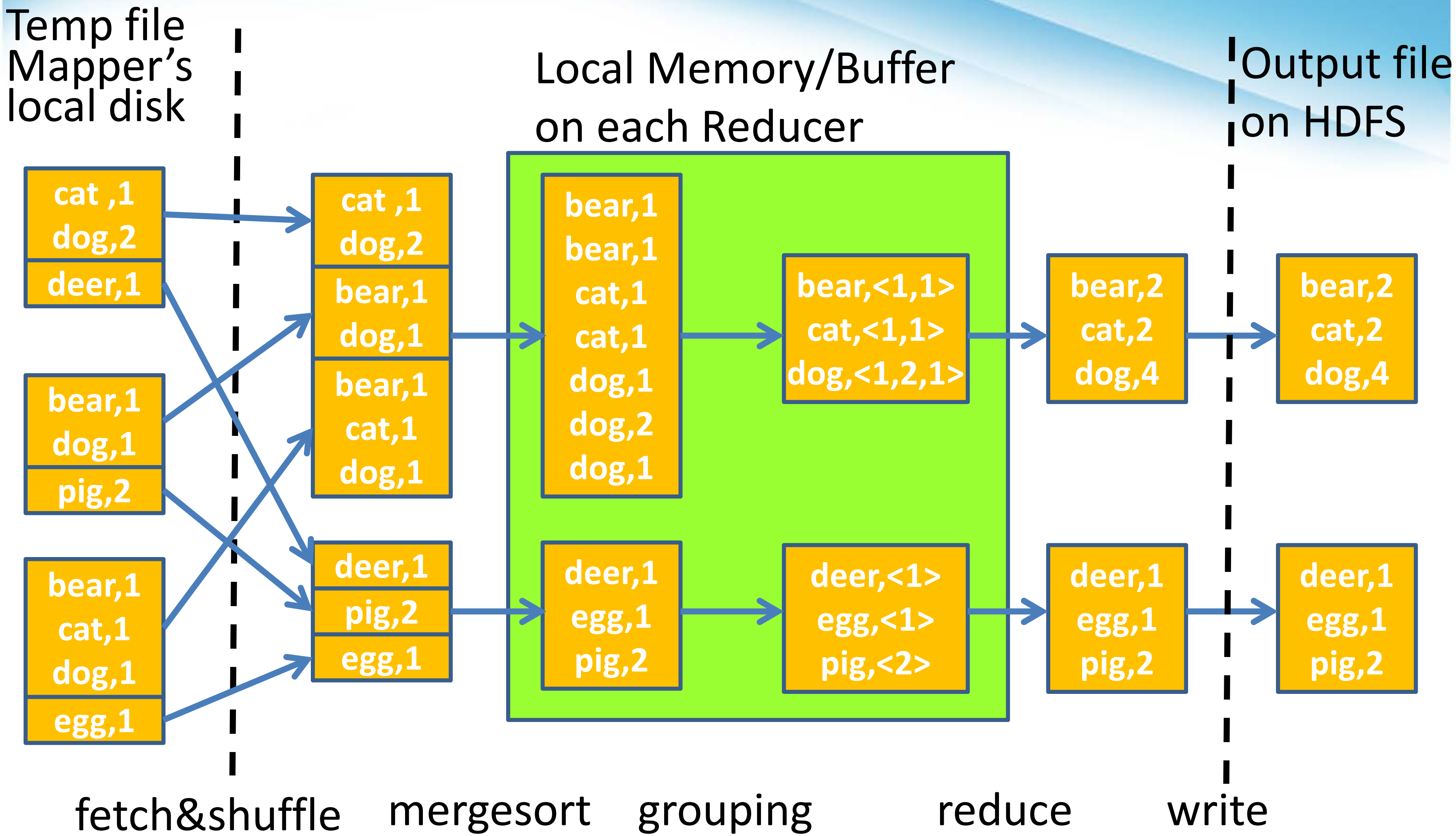
- Mergesort is used by the framework to effectively merge the output from mappers, and sort the result in one stage
- <K,V> pairs are sorted by a comparator class called the **OutputKeyComparator**
  - The comparator can be set by "**JobConf.setOutputKeyComparatorClass**"
  - The comparator must implement the "***rawComparator***" **interface** *or extend "**writeComparator**" **class***
    - Override the function: **compare**

KeyComparator Example

- Let keys in the form of <string1>:<string2>
- Sort keys in the ascending order of <string1>

```
public static class KeyComprator extends WritableComparator {
    protected KeyComprator() { super(Text.class, true); }
    public int compare(WritableComparable w1,
                                    WritableComparable w2) {
        Text t1 = (Text) w1;
        Text t2 = (Text) w2;
        String[] t1Items = t1.toString().split(":");
        String[] t2Items = t2.toString().split(":");
        return t1Items[0].compareTo(t2Items[0]);
    }
}
```

Reduce Phase: Grouping



Reduce Phase: Grouping: OutputValueGroupingComparator

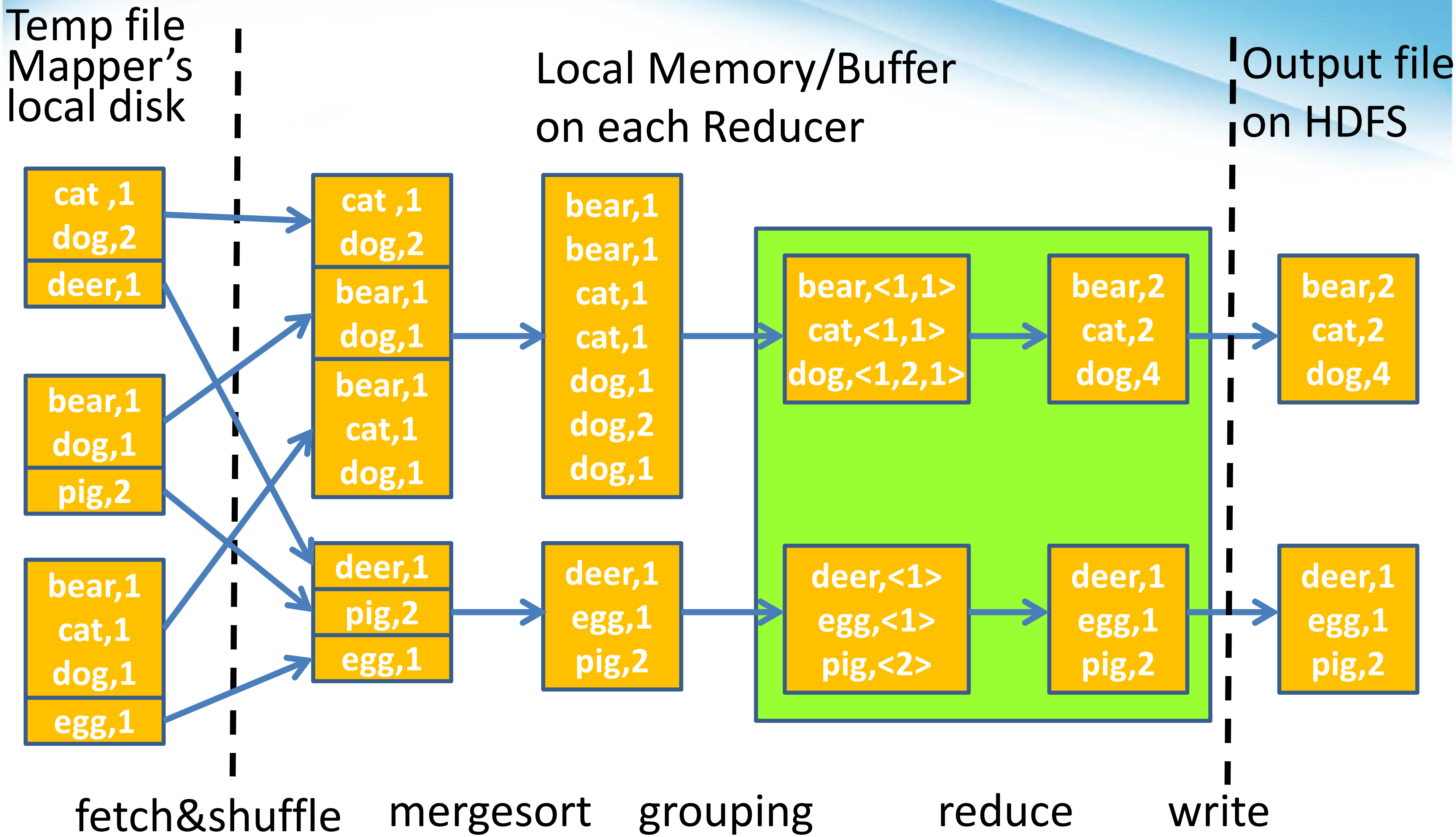
- The comparator set by "JobConf.setOutputValueGroupingComparator"
- Grouping:
  - <K,V> pairs compared as equal are grouped together
  - If multiple keys in the same group, "**OutputKeyComparator**" is used to decide the key for the group
- Example:
  - Input: <A1, V1>, <A2, V2>, <A3, V3>, <B1, V4>, <B2, V5>
  - Grouping comparator to just compare the first letter:
    (A1, {V1,V2,V3}); (B1, {V4,V5});

GroupComparator Example

```
public static class  groupComprator extends WritableComparator {
    protected GroupComprator() { super(Text.class, true); }
    public int compare(WritableComparable w1,
                       WritableComparable w2) {
        Text t1 = (Text) w1;                  Text t2 = (Text) w2;
        int t1char = t1.charAt(0);            int t2char = t2.charAt(0);
        if (t1char < t2char) return -1;
        else if (t1char > t2char) return 1;
        else return 0;
    }
}
```

- Could also override "**compare**(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2)" which de-serializes the key in ***buffer*** then call the above function

Reduce Phase: Reducer



Reduce Phase: Reducer

- Reducer reduces a set of intermediate values which share a key to a smaller set of values.
- Each **group** of (K,V) pair applied with a reduce func:
  - reduce(WritableComparable, **Iterator**, OutputCollector, Reporter)
- Constructor

```
public static class Reduce extends MapReduceBase
    implements Reducer<Text, IntWritable, Text, IntWritable>
{
        ......
}
```

Type of input Key&value from the Mapper

Type of output Key&value from the Reducer

Reducer: WordCount Example

```
public static class Reduce 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));
    }
}
```

Outline

- Hadoop MapReduce Overview
- Hadoop Job Configuration
- Data Format and Data Type
- Hadoop MapReduce Process
- Performance Profiling & Tuning
  - Reporter
  - How many maps
  - How many reduces

Reporter

- The Hadoop system records a set of metric counters for each job that it runs
  - E.g.: # input records mapped, # bytes reads from or writes to HDFS, etc…
- The ***Reporter*** object passed in to your Mapper and Reducer classes can be used to record & update those metric counters
  - The **values are aggregated by the master node** of the cluster, so they are "thread-safe" in this manner
  - **Counters are incremented** through the **Reporter.incrCounter()** method
  - the values of all the counters will be **printed to stdout** when the job completes

Reporter

- Count the number of "A" vs. "B" records seen by the mapper

```
public class MyMapper extends MapReduceBase
    implements Mapper<Text, Text, Text, Text> {
      static enum RecordCounters { TYPE_A, TYPE_B, TYPE_UNKNOWN };
      public boolean isTypeA(Text input) { ... }
      public boolean isTypeB(Text input) { ... }
      public void map(Text key, Text val, OutputCollector<Text, Text> output,
          Reporter reporter) throws IOException {
             if (isTypeA(key)) reporter.incrCounter(RecordCounters.TYPE_A, 1);
             else if (isTypeB(key)) reporter.incrCounter(RecordCounters.TYPE_B, 1);
             else   reporter.incrCounter(RecordCounters.TYPE_UNKNOWN, 1);
    }
}
```

How many Maps?

- The number of maps is usually driven by the total size of the inputs, that is, the total **number of blocks of the input files**
- The right level of parallelism for maps seems to be around 10-100 maps per-node
- **JobConf.setNumMapTasks**(int n)

How many reduces?

- The right number of reduces seems to be 0.95 or 1.75 multiplied by <#nodes x #reduce_task/node>
- With 0.95 all of the reduces can launch immediately
- With 1.75 the faster nodes will finish their first round of reduces and launch a second wave of reduces doing a much better job of load balancing.
- Increasing the number of reduces increases the framework overhead, but increases load balancing and lowers the cost of failures.
- **JobConf.setNumReduceTasks**(int n)

Reference

- Distributed system lecture slides from Gregory Kesden
- Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 2004), pages 137-150
- http://hadoop.apache.org/common/docs/r1.0.3/mapred_tutorial.html
- http://hadoop.apache.org/common/docs/r1.0.3/api/org/apache/hadoop/mapred/JobConf.html
- http://developer.yahoo.com/hadoop/tutorial/module5.html