Cloud Programming: Lecture4 – MapReduce Parallel Programming

National Tsing-Hua University

2016, Spring Semester

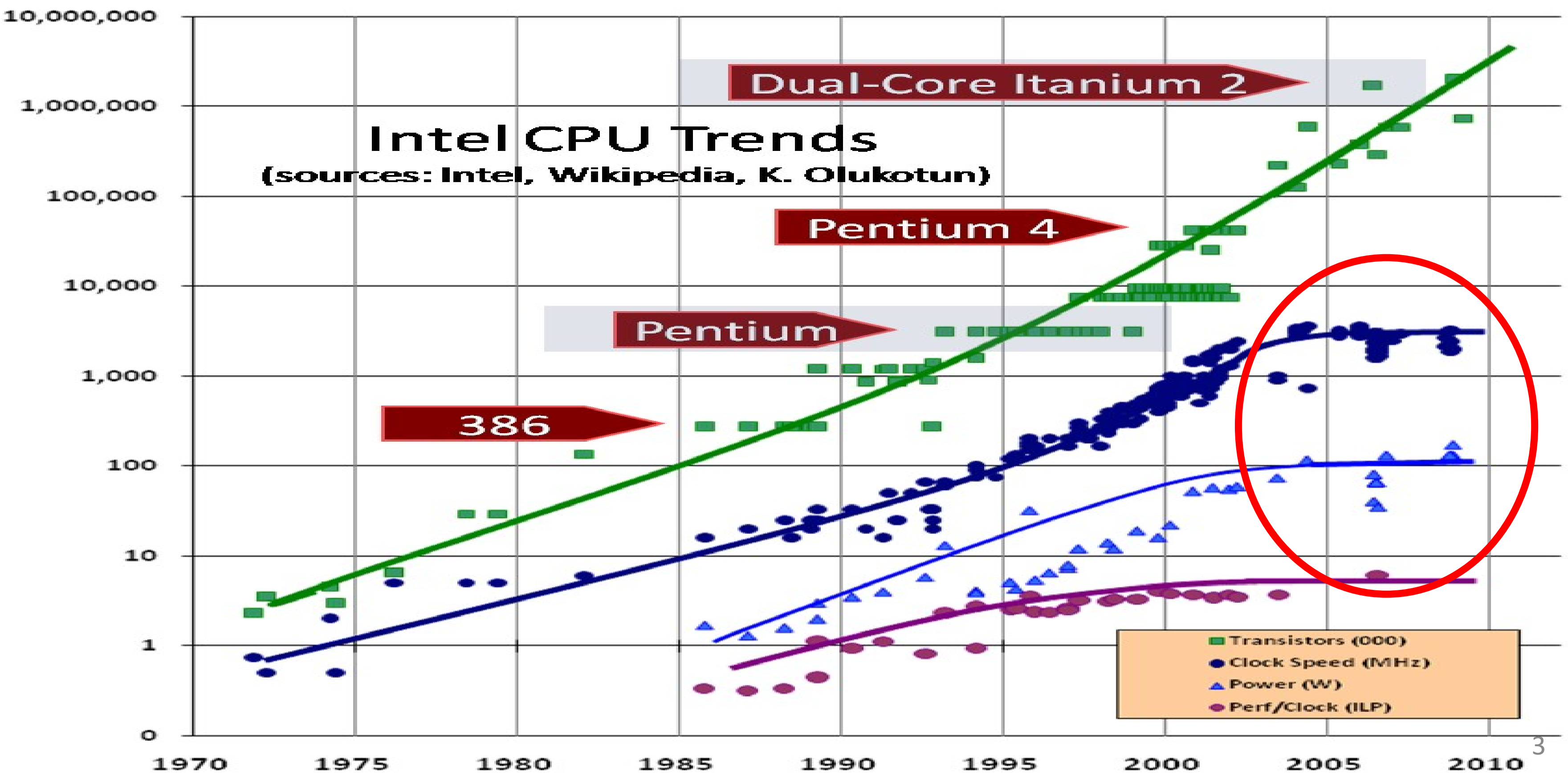
國立清華大學

Outline

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

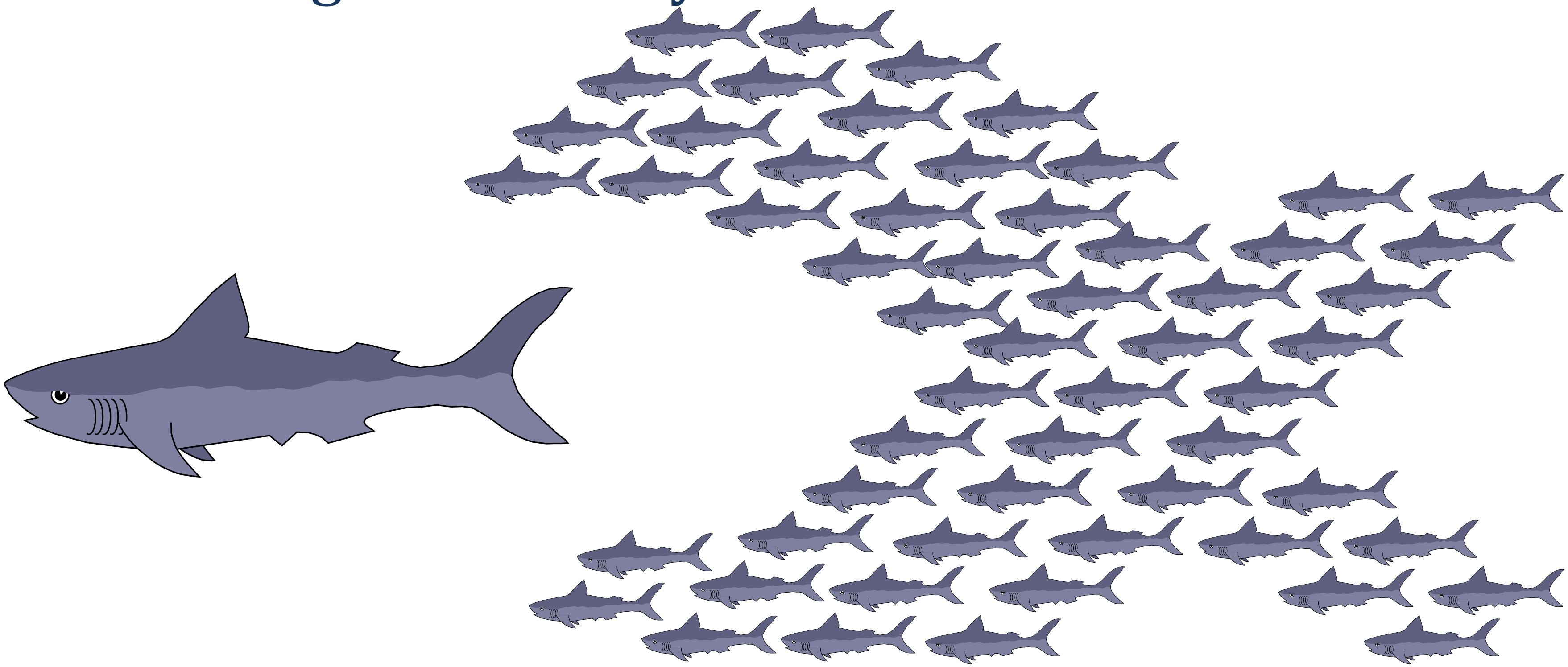
The Death of CPU Scaling

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

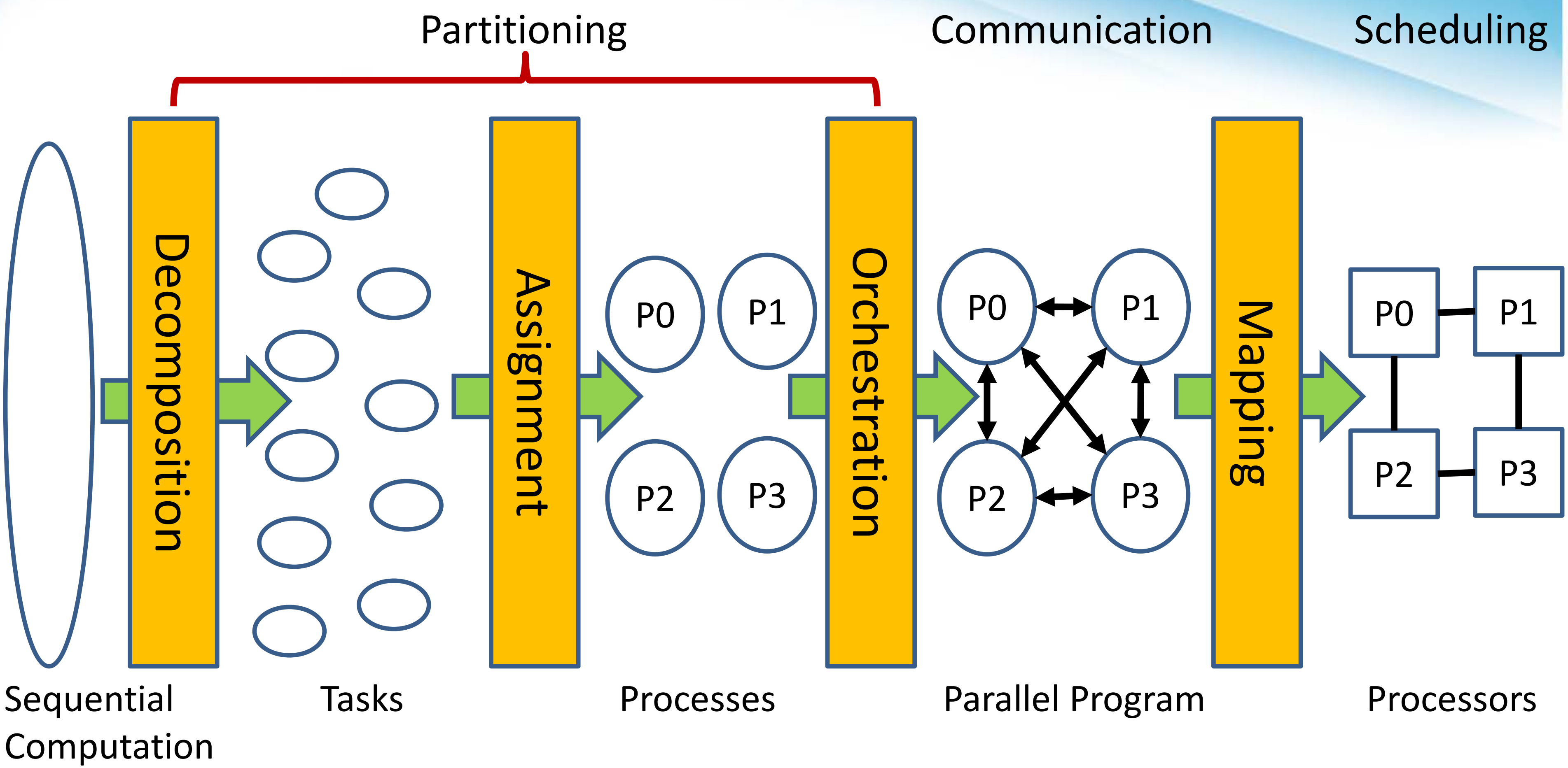


Distributed Computing

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



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

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.

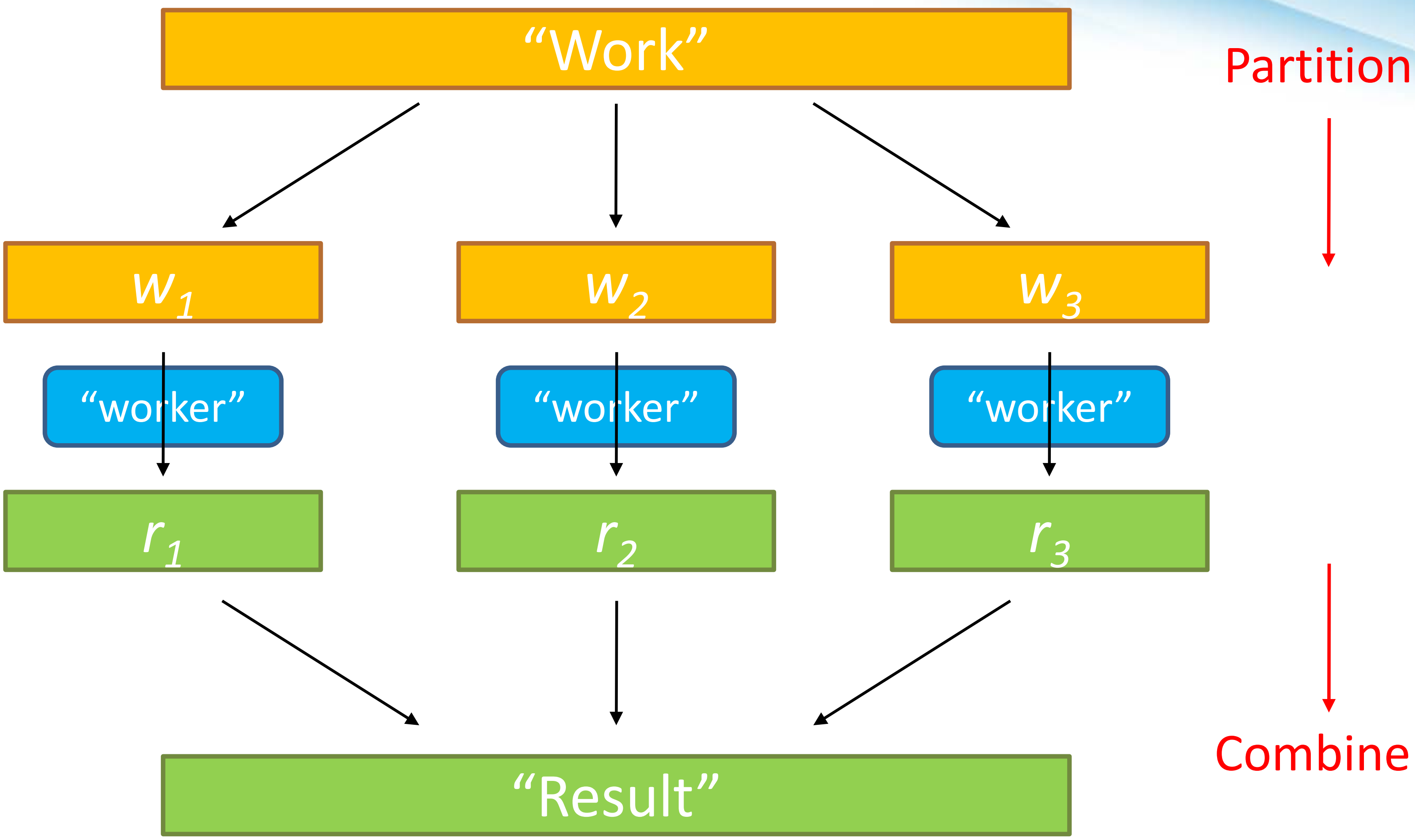
Outline

- 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

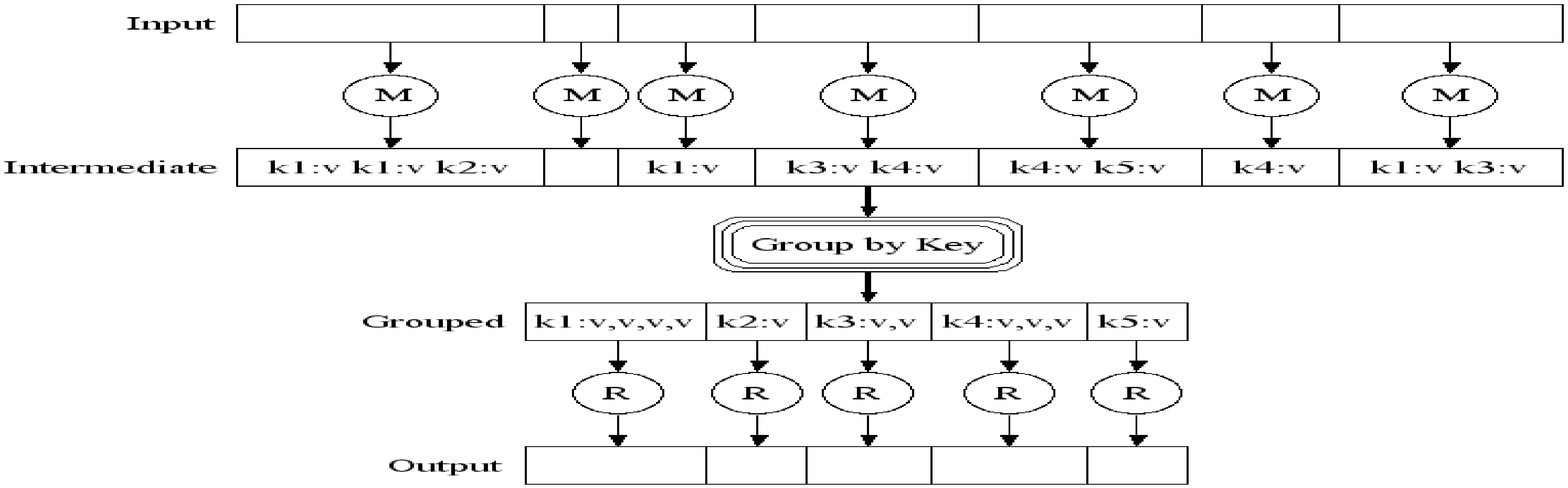
Map

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

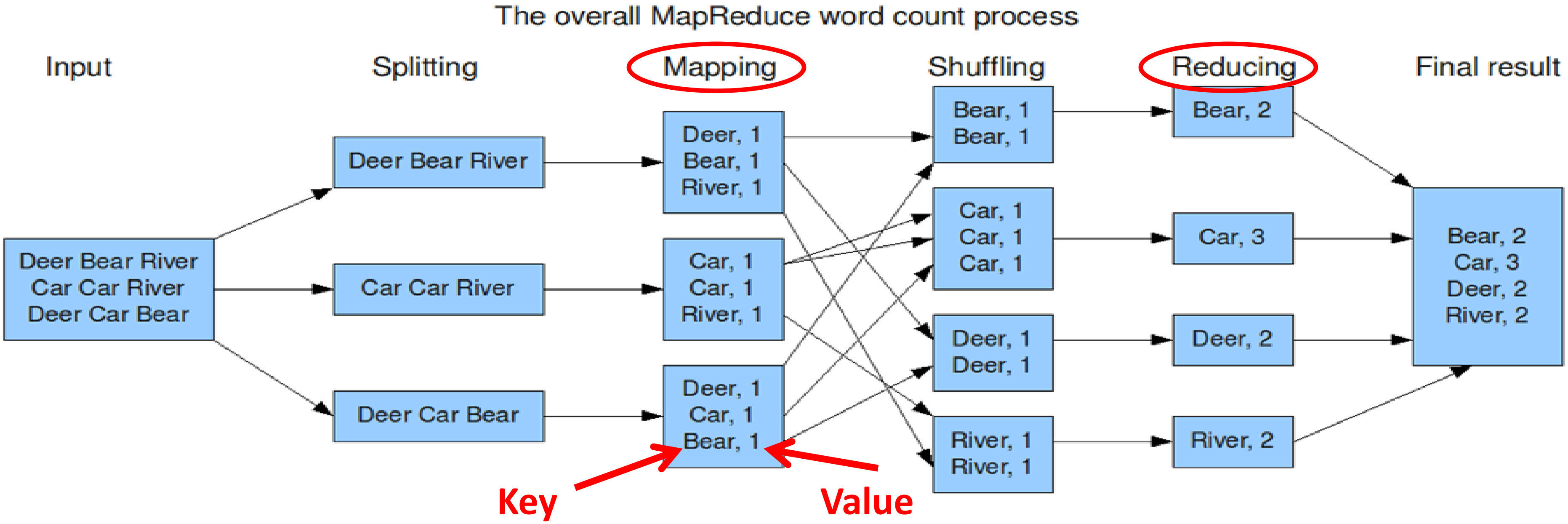


MapReduce Word Count Example

- 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);
```



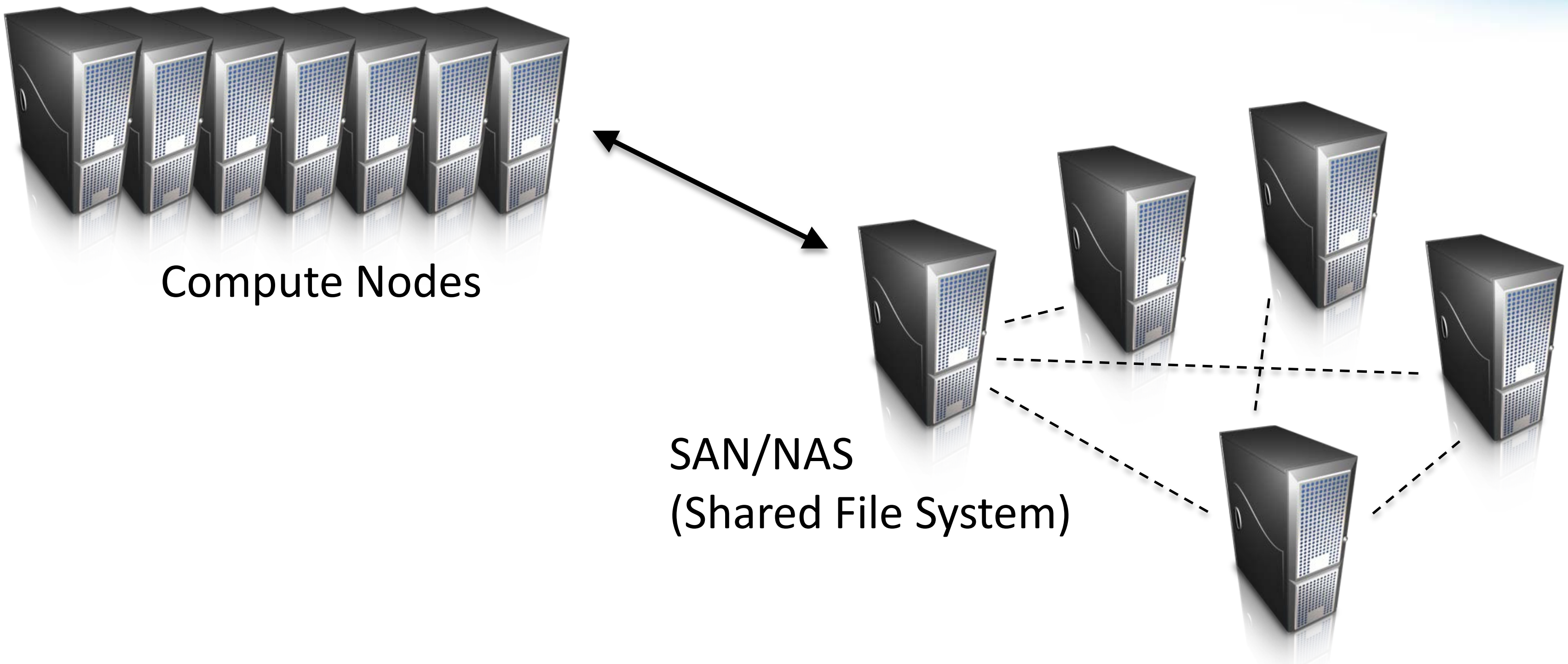
- The execution framework handles everything else…

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?



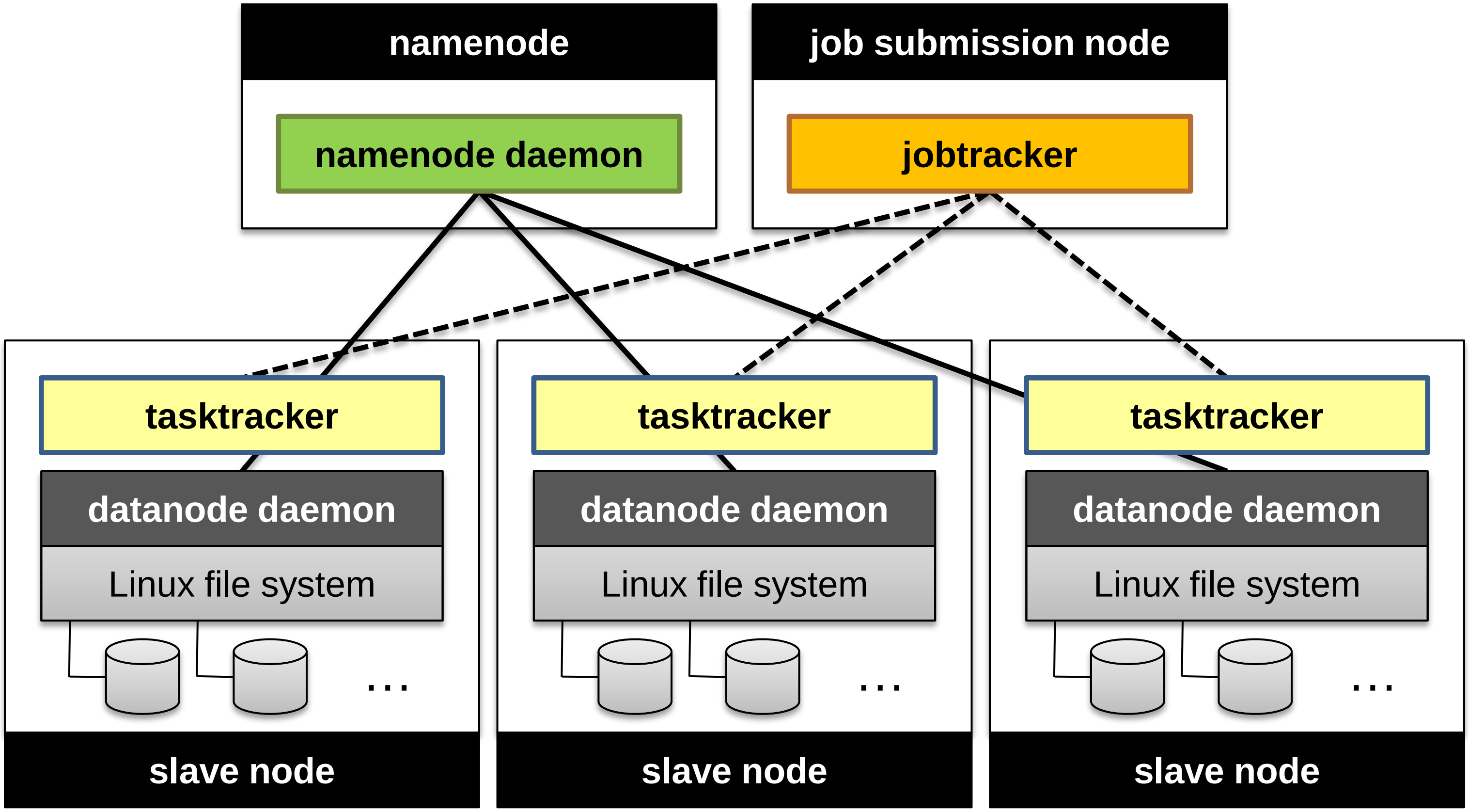
What's the problem here?

Distributed File System

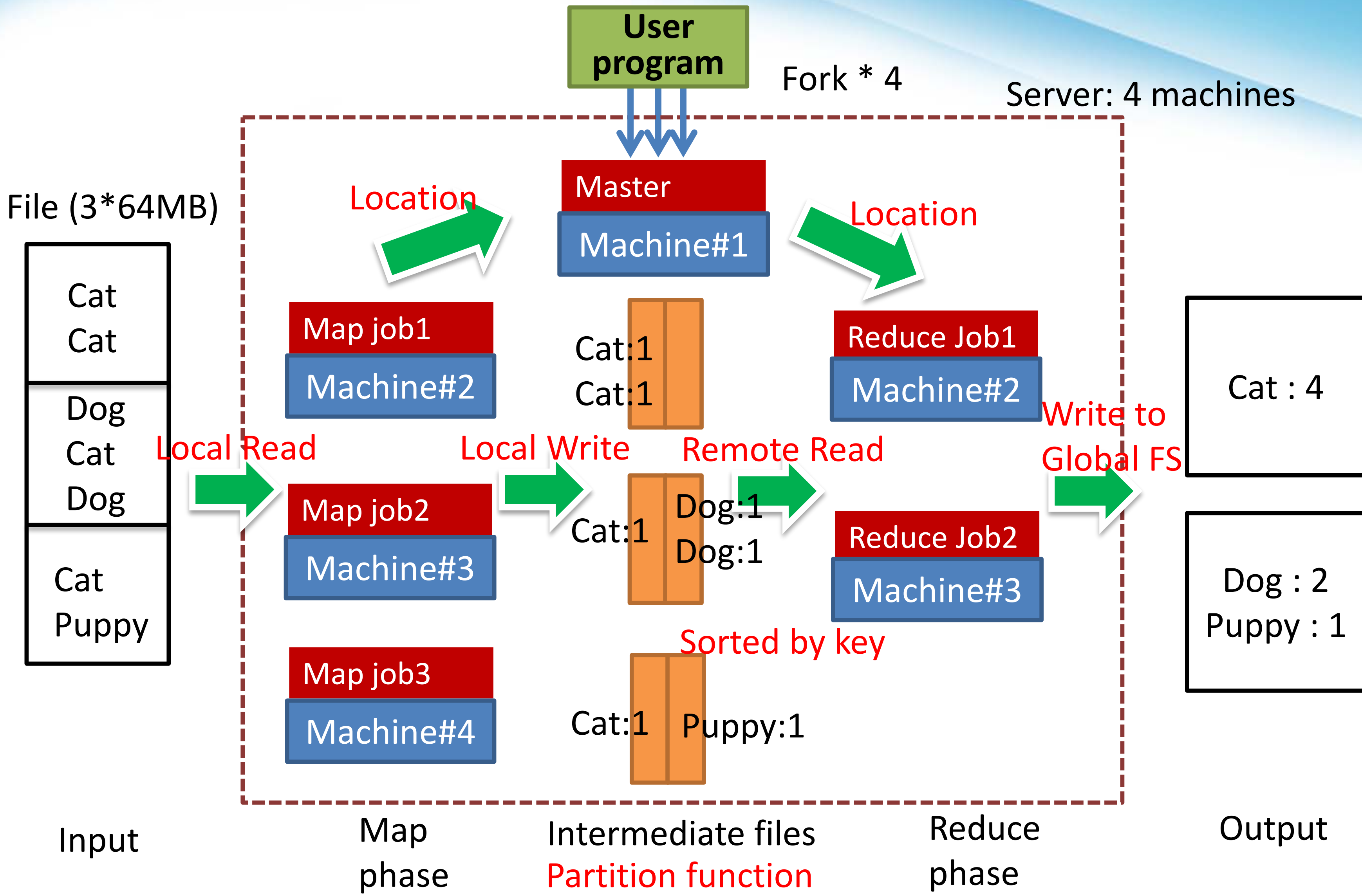
- Don’t move data to workers…
  move workers to the data!
  - A node act as both compute and storage node
  - Store data on the local disks of nodes in the cluster
  - Start up the workers on the node that has the data local
- A distributed file system is the answer
  - GFS (Google File System) for Google’s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

Putting everything together…

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



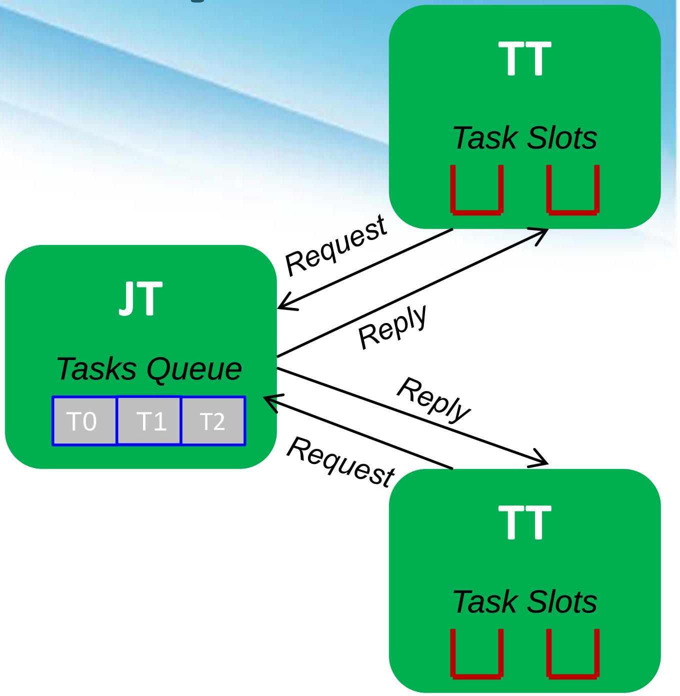
MapReduce in Action



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

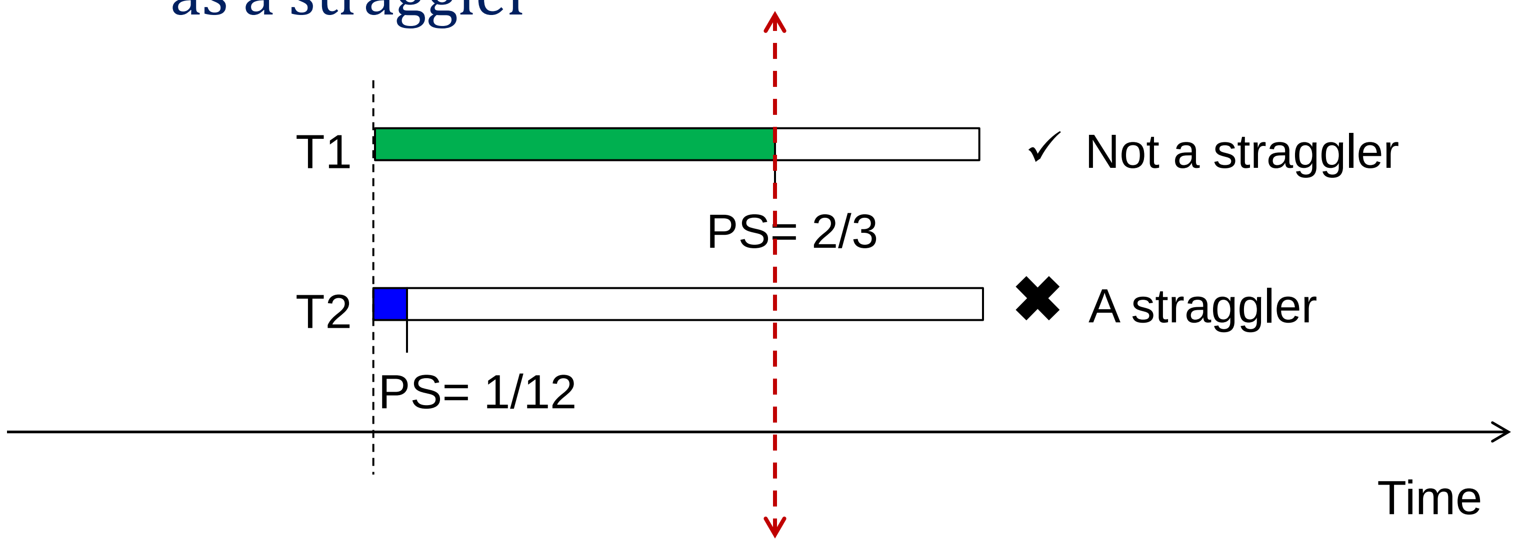
- MapReduce can guide jobs toward a successful completion even when jobs are run on a large cluster where probability of failures increases
- The primary way that MapReduce achieves fault tolerance is through *restarting tasks*
- If a TT fails to communicate with JT for a period of time (by default, 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

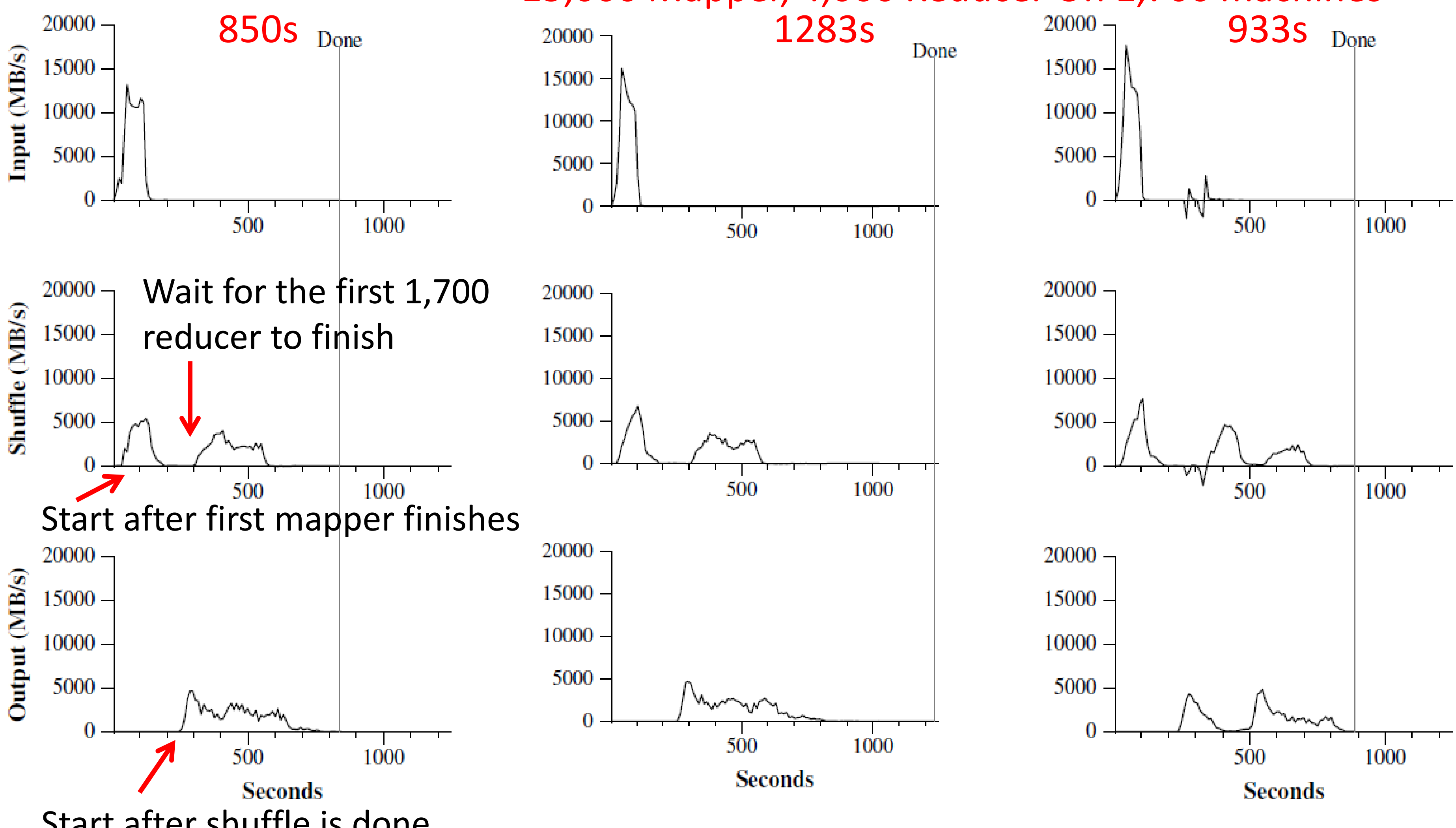
Locating Stragglers

- 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

What Makes MapReduce Unique?

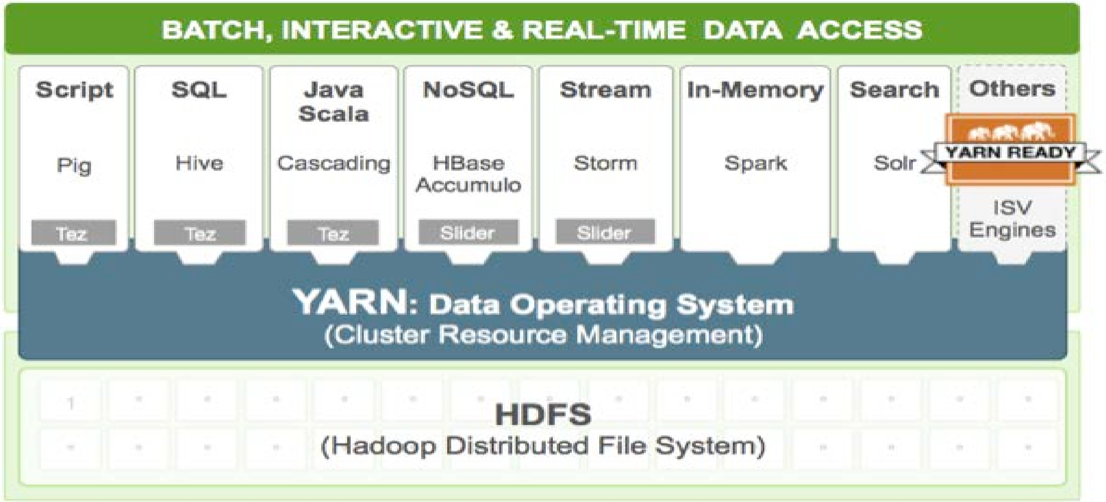
- MapReduce is characterized by:
  1. Its simplified programming model which allows the user to quickly write and test distributed systems
  2. Its efficient and automatic distribution of data and workload across machines. Moving process to data.
  3. Seamless scalability. Specifically, after a Mapreduce 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
  - WordCount Example
  - Advanced Features
  - Custom Key Example
  - SecondarySort Example

Hadoop Implementation

- Hadoop release 2.x
  - *New version with YARN*



- Java Language
  - Based on **inheritance and interface**
- Official Tutorial
  - https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

Basic HDFS Commands

Command	Description
-ls <args>	List directory
-mkdir <paths>	Create a directory
-put <localsrc> <HDFS_dest_Path>	Upload files
-get <hdfs_src> <localdst>	Download file
-cat <path[filename]>	See content of files
-cp <source> <dest>	Copy files in HDFS
-rm <arg>	Remove files or directories
-tail <path[filename]>	Display last few lines of a file
-getmerge [hdfs_src_dir] [hdfs_dst_file]	Merge files (from reducers)

- $/bin/hadoop fs [command]
- Ref: https://hadoop.apache.org/docs/r2.7.1/hadoop-project-dist/hadoop-common/FileSystemShell.html

Main Hadoop Classes

- **Configureation**
  - Hadoop **cluster** configuration
- **Job**
  - 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

Import hadoop package

Import classes in "***org.apache.hadoop.mapreduce***" package

- **import java.io.IOException; import java.util.StringTokenizer;**
- **import org.apache.hadoop.conf.Configuration;**
- **import org.apache.hadoop.fs.Path;**
- **import org.apache.hadoop.io.IntWritable;**
- **import org.apache.hadoop.io.Text;**
- **import org.apache.hadoop.mapreduce.Job;**
- **import org.apache.hadoop.mapreduce.Mapper;**
- **import org.apache.hadoop.mapreduce.Reducer;**
- **import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;**
- **Import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;**
- **[Other necessary classes called by your code ……]**

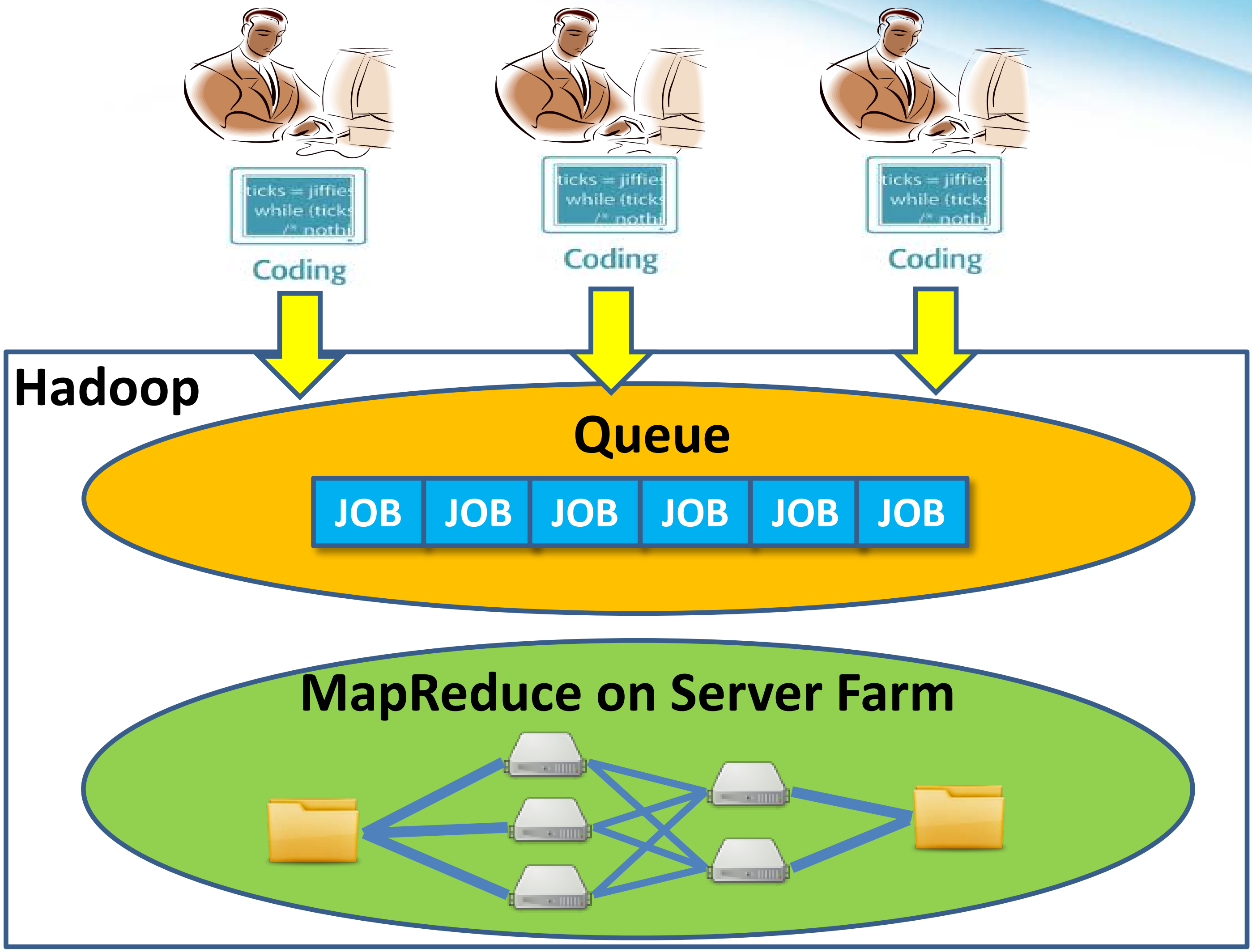
Notice:

- **"mapreduce" package it not interchangable with "mapred" package**
- **Prevent using "Deprecated" methods**

https://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapreduce/Job.html

JOB CLASS

Hadoop Runtime

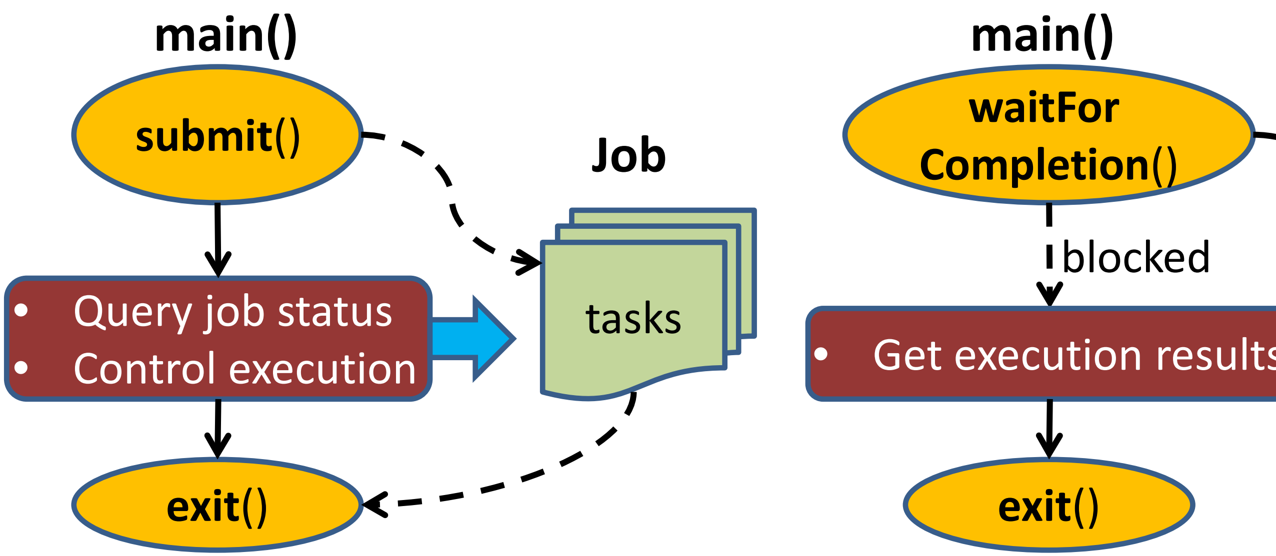


Job Class

- configure a job
  - Specify the class for mapper, reducer, combiner, etc.
- submit the job
  - **Submit the job to the cluster and return immediately**
  - **Or submit the job to the cluster and wait for it to finish**
- control its execution
  - Set the number of max attempts to run a reduce or map task.
  - Set scheduling priority.
  - Kill the running job, or specific task.
  - Turn speculative execution on or off for this job.
- query its state.
  - Get the *progress* of the job's map-tasks or reduce-tasks (between 0 and 1).
  - Returns the current state of the Job.
  - Get start time of the job.
  - Check if the job completed successfully.

Job Creation & Submission

Method	Description
getInstance (**Configuration** conf, **String** jobName)	Creates a new job with a given jobName.
setJarByClass(**Class**<?> cls)	Set the Jar by finding where a given class came from.
submit()	Submit the job to the cluster and return immediately. (non-blocking call)
waitForCompletion(boolean verbose)	Submit the job to the cluster and wait for it to finish. (blocking call)



Query & Control Job Execution

Method	Description
getStartTime()	Get start time of the job.
getFinishTime()	Get finish time of the job.
getStatus()	Returns a JobStatus object contain all the current job state info
mapProgress() **reduceProgress**()	Get the *progress* of the job. , as a float between 0.0 and 1.0.
getCounters()	Gets the counters object for this job
isComplete()	Check if the job is finished or not.

Method	Description
setPriority(JobPriority prio)	High/Low/Normal/Very_High/Very_Low
setNumReduceTasks(int n)	Set the requisite number of reduce tasks for this job. (notice: no method for map tasks)
setSpeculativeExecution(boolean flag)	Turn speculative execution on or off for this job.
killJob()	Kill the running job.
killTask(TaskAttemptID taskId)	Kill indicated task attempt.

WordCount Example

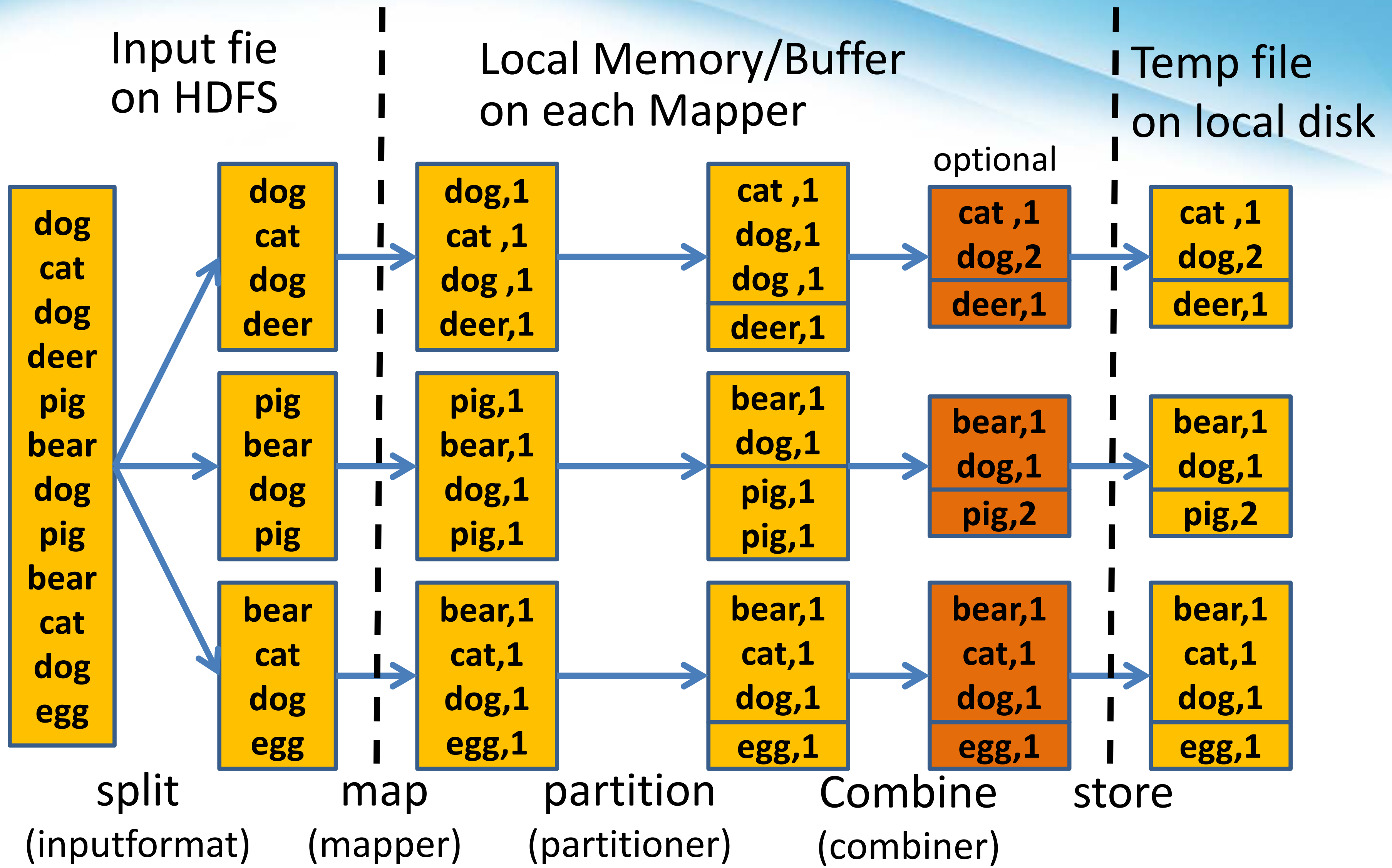
```
public class WordCount {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, “wordcount”);
        job.setJarByClass(WordCount.class);
        job.waitForCompletion(true); // Submit the job and wait for it to finish
    }
}
```

```
public class WordCount {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, “wordcount”);
        job.setJarByClass(WordCount.class);
        job.submit(); // Submit the job and return immediately
        while(job. isComplete()==false) {
            System.out.println(mapProgress());
        }
    }
}
```

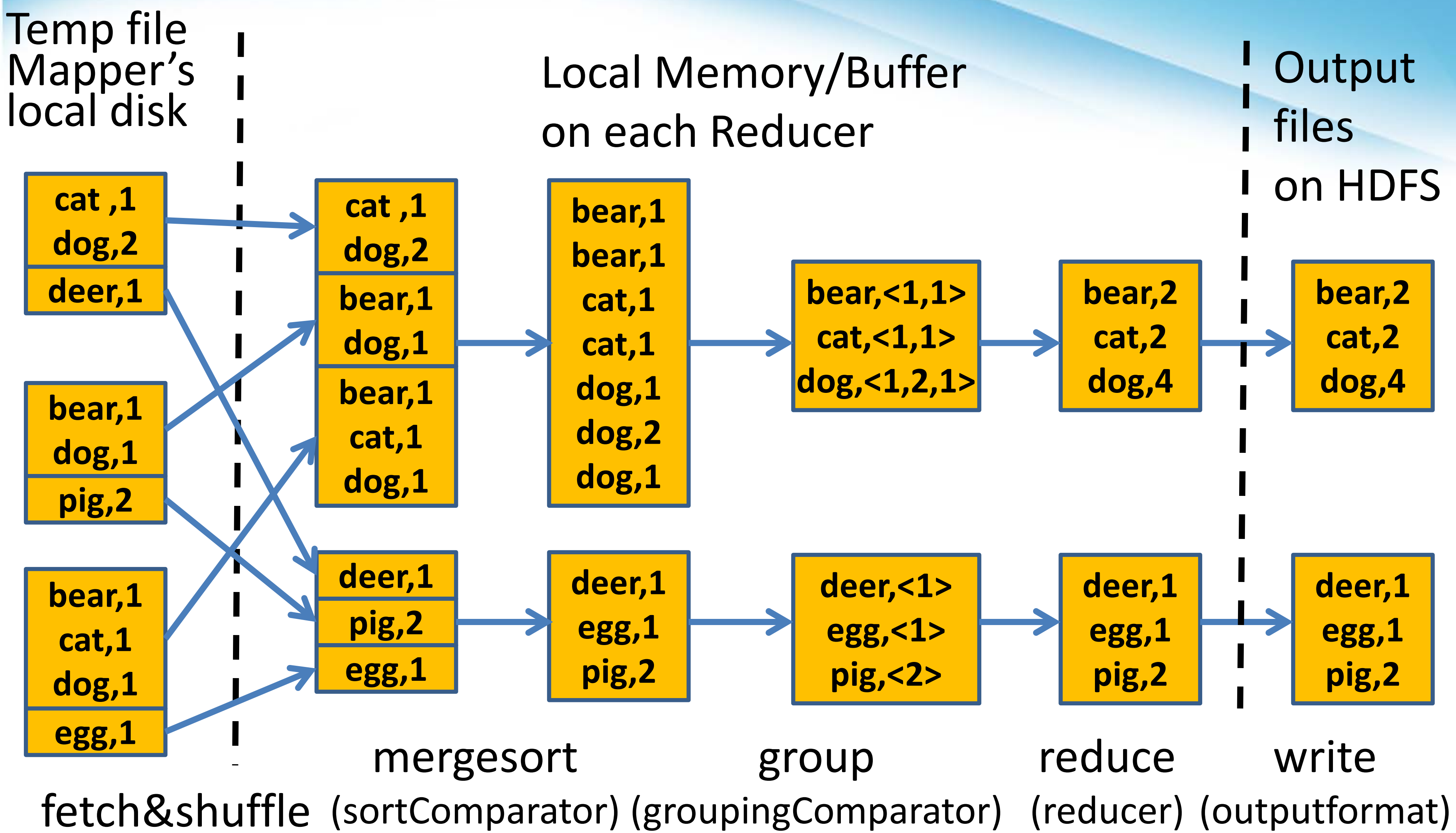
Job Configuration on Compute Functions

Method	Description
setMapperClass (Class<? extends Mapper>)	Set the Mapper class for the job
setReducerClass (Class<? extends Reducer>)	Set the Reducer class for the job.
setPartitionerClass (Class<? extends Partitioner>)	partition Mapper-outputs to be sent to the reducers
setCombinerClass (Class<? extends Reducer>)	combine map-outputs before being sent to the reducer It is an optional function during execution
setGroupingComparatorClass (Class<? extends RawComparator>)	Define the comparator that controls which keys are grouped together for a single call to reducer
setSortComparatorClass (Class<? extends RawComparator>)	Define the comparator that controls how the keys are sorted before they are passed to the reducer

Map Phase Steps

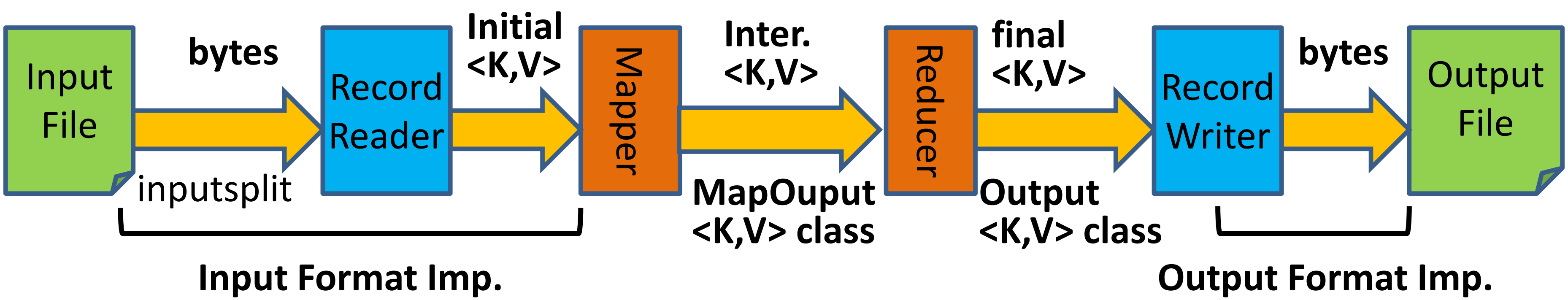


Reduce Phase Steps



Job Configuration on Data Type

Method	Description
setInputFormatClass()	Set the InputFormat **implementation** for the job
setMapOutputKeyClass()	Set the key class for the map output data **Same type as final output if not specify**
setMapOutputValueClass()	Set the value class for the map output data **Same type as final output if not specify**
setOutputKeyClass()	Set the key class for the job output data
setOutputValueClass()	Set the value class for job outputs
setOutputFormatClass()	Set the OutputFormat **implementation** for the job



How many Map/Reduce Tasks?

- The number of map tasks is controlled by the implementation of **inputsplit** in **inputFormat**
  - Default is to split by the **block size of files in HDFS**
  - But it can also be overwritten to split differently
- The number of reduce tasks is controlled by the job configuration: job.setNumReduceTasks(int n)
  - The right number of reduces seems to be 0.95 or 1.75 multiplied by #reduce_slots
  - More reducer → higher framework overhead, better load balancing and lowers failure cost.

WORDCOUNT EXAMPLE

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**: writes plain text: key, value, and "\r\n".

Key-Value Pair Class

- Both ***key*** and ***value*** must implement *Writiable* interface
  - *A* serializable object which implements a simple, efficient, serialization protocol, based on **DataInput** and **DataOutput**
- ***Key*** also implements the interface of *WritableComparable*
  - Because key is sorted by the framework
- Default supported types includes:
  - BooleanWritable, BytesWritable, DoubleWritable, FloatWritable, IntWritable, LongWritable, Text, NullWritable

Main()

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, “world count”);
    job.setJarByClass(WordCount.class);

    job.setMapperClass(Tokenizer.class);
    job.setCombiner(IntSum.class);
    job.setReducerClass(IntSum.class);

    //FileInputFormat is the base class for all file-based InputFormats
    FileInputFormat.addInputPaths(job, new Path(args[0]));
    FileOutputFormat.addOutputPath(job, new Path(args[1]));

    job.setInputFormat(TextInputFormat.class); // inputs are texts
    job.setOutputFormat(TextOutputFormat.class); // outputs are texts

    job.setOutputKeyClass(Text.class); // intermediate key is text
    job.setOutputValueClass(IntWritable.class); // intermediate value is integer

    job.waitForCompletion(true); // Submit the job and wait for it to finish
}
```

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, Context)
  - **<WritableComparable, Writable>** are the input key-value pairs generated by the **InputFormat class**
  - **context.write(K, V)** collects output key-value pairs

Mapper

Input <K,V> type from InputFormat

Default <K,V> type for final output

```
public static class Tokenizer 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 {
          String line = value.toString();
          StringTokenizer iter = new StringTokenizer(line);
          while (iter.hasMoreTokens()) {   // each line has multiple words
                   word.set(iter.nextToken());
                   context.write(word, one);
          }
    }
}
```

<K,V> must be private var to the class

Set the var value
Don't declare a new var here

```
main():
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    job.setMapperClass(Tokenizer.class);
    job.setInputFormat(TextInputFormat.class);
```

Reducer

- Reducer reduces a set of intermediate values which share a key to a smaller set of values.
  - 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 **group** of (K,V) pair applied with a reduce func:
  - reduce(WritableComparable, **Iterator<**Writable**>**, Context)
  - **WritableComparable** is the input key-value pairs generated by the **mapper class**
  - **Iterator<**Writable**>** is the **list of values grouped by the same key**
  - **context.write(K, V)** collects output key-value pairs

Reducer

Output <K,V> type

```
public static class IntSum extends
          Reducer<Text, IntWritable, Text, IntWritable>
{
   private IntWritable result = new IntWritable();
   public void reduce(Text key, Iterator<IntWritable> values,
          Context context) throws IOException, InterruptedException
   {
     int sum = 0;
     while (values.hasNext())  sum += values.next().get();
     result.set(sum);
     context.write(key, result);
   }
}
```

Set the var value
Don’t declare a new var here

```
main():
   job.setOutputKeyClass(Text.class);
   job.setOutputValueClass(IntWritable.class);
   job.setReducerClass(IntSum.class);
   job.setOutputFormat(TextInputFormat.class);
```

Compile & Execution

- Input files:
  - $ bin/hadoop fs -cat /user/joe/wordcount/input/file01

    Hello World, Bye World!

  - $ bin/hadoop fs -cat /user/joe/wordcount/input/file02

    Hello Hadoop, Goodbye to hadoop.

- Compile WordCount.java and create a jar
  - $ bin/hadoop com.sun.tools.javac.Main WordCount.java
  - $ jar cf wc.jar WordCount*.class
- Run applications
  - bin/hadoop jar wc.jar WordCount /user/joe/wordcount/input /user/joe/wordcount/output
- Check output
  - $ bin/hadoop fs -cat /user/joe/wordcount/output/part-r-00000

Bye 1
Goodbye 1
Hadoop, 1
Hello 2
World! 1
World, 1
hadoop. 1
to 1

Compilation & Execution

- Compile WordCount.java and create a jar
  - $ javac -classpath `yarn classpath` -d . WordCount.java
  - $ jar cf wc.jar WordCount*.class
- Run applications
  - $ bin/hadoop jar wc.jar WordCount /user/hadoop/wordcount/input /user/hadoop/wordcount/output
- Check output
  - $ bin/hadoop fs -cat /user/hadoop/wordcount/output/part-r-00000

Bye 1
Goodbye 1
Hadoop, 1
Hello 2
World! 1
World, 1
hadoop. 1
to 1

One output file per reducer

Print Out Message

- Hadoop comes with preconfigured log4j

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
```

- define logger inside your mappers, or any other class

```
private static final Log LOG = LogFactory.getLog(MyClass.class);
```

- Log your info

```
LOG.info("My message");
```

- Check your log through YARN

```
yarn logs -applicationId application_XXXXX_XXXX
```

Custom value types

Combiner

Partitioner

Counter

GroupingComparator

SortComparator

DistributedCache

ADVANCED PROG.

Custom Value Types

- *Value in* 3-dimensional coordinate

  struct point3d { float x; float y; float z; }
- Implement *Writable* interface
  - write: data serialization
  - readFields: data de-serialization

```
public class Point3D implements Writable {
    private float x; private float y; private 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(); }
}
```

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**
    - Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
  - hashCode(): used for **partitioning**

Custom Key Types

```
public class Point3D implements WritableComparable <Point3D> {
    private float x; private float y; private float z;
    public 3DPoint (){x=0.0f; y=0.0f; z=0.0f;}
    public void set(float x, float y, float z)  { this.x = x; this.y = y; this.z = z; }
    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 int hashCode() {
            return Float.floatToIntBits(x) ^ Float.floatToIntBits(y) ^
            Float.floatToIntBits(z);
    }
    // overwrite other methods in Writable interface:  write & readFields
}
```

Point3D Sorting Example

```
public class TestPoint3D {
  public static class TokenizerMapper
      extends Mapper<Object, Text, Point3D, NullWritable>
  {
    private Point3D point = new Point3D();
    public void map(Object key, Text value, Context context
                ) throws IOException, InterruptedException {
        String line  = value.toString();
        String[] tokens = line.split(",");
        float x = Float.parseFloat(tokens[0]);
        float y = Float.parseFloat(tokens[1]);
        float z = Float.parseFloat(tokens[2]);
        point.set(x,y,z);
        context.write(point, NullWritable.get());
      }
  }
```

Input file:
0,0,0
1,0,2
4,4,4
2,2,2

Output file:
0,0,0
1,0,2
2,2,2
4,4,4

```
main():
    job.setOutputKeyClass(Point3D.class);
    job.setOutputValueClass(NullWritable.class);
    job.setMapClass(Tokenizer.class);
```

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

Custom value types

Combiner

Partitioner

Counter

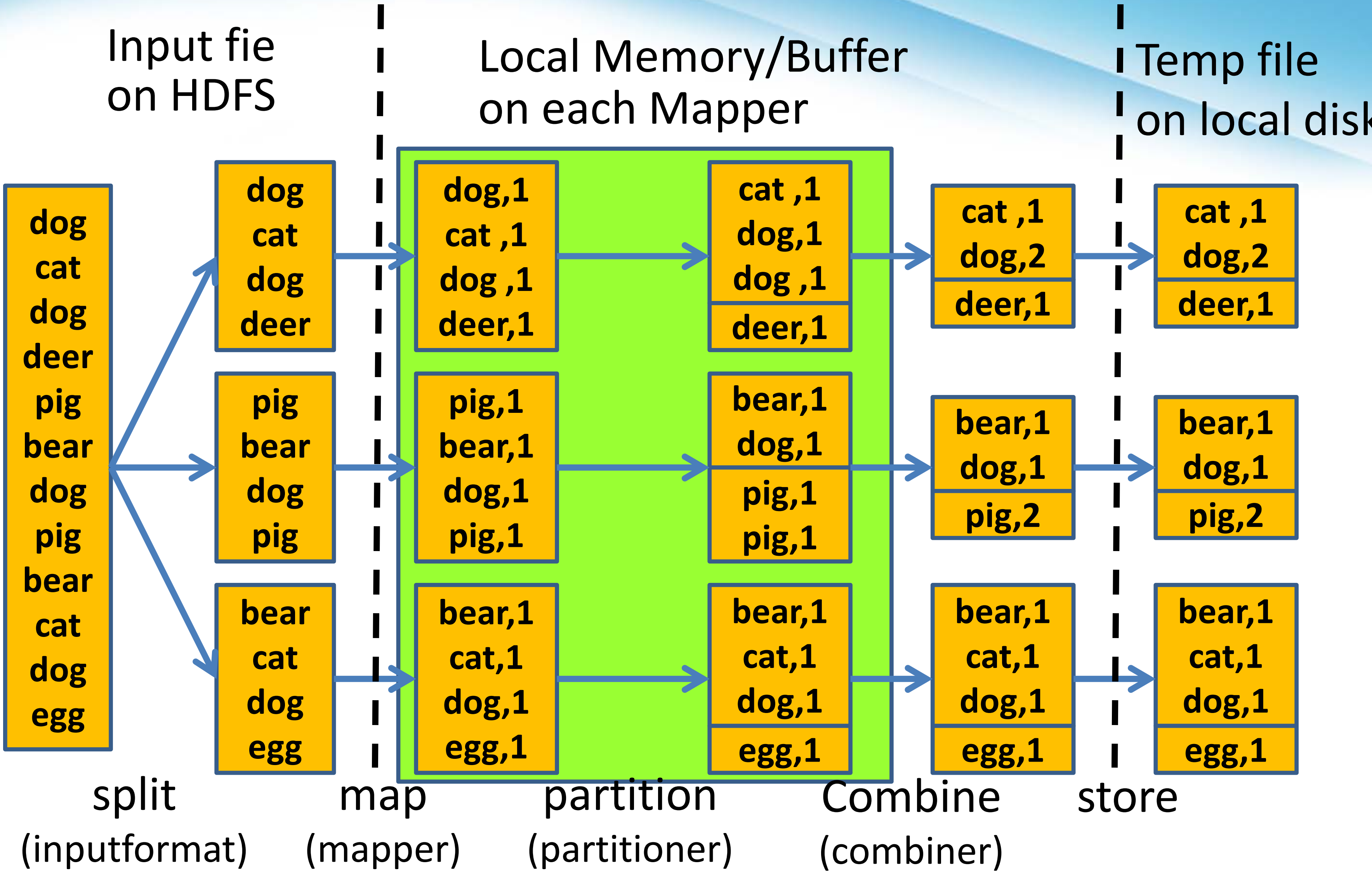
GroupingComparator

SortComparator

DistributedCache

ADVANCED PROG.

Map Phase: Partitioner



Map Phase: Partitioner

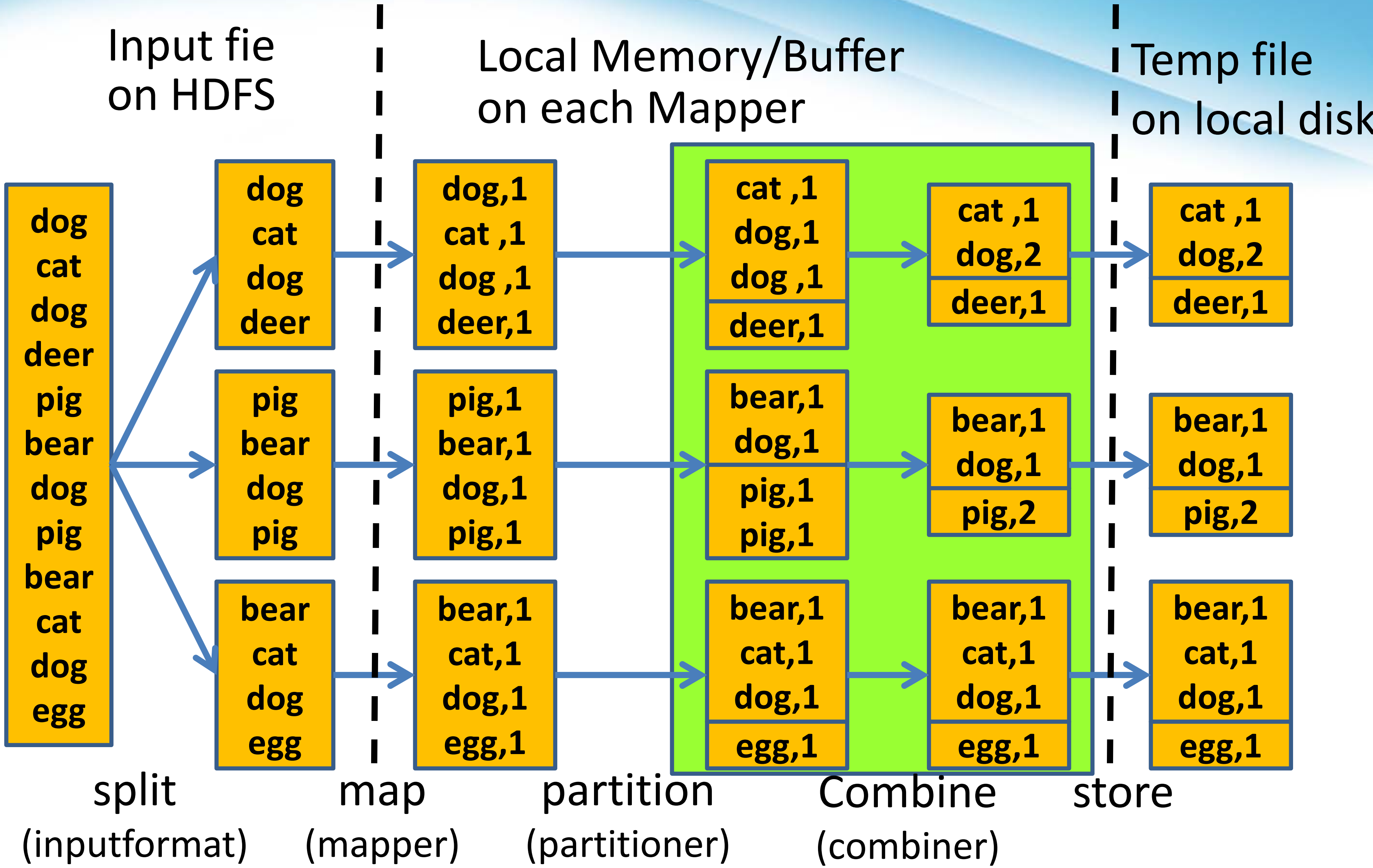
- Partitioner decides which intermediate (K,V) pair is sent to which reducer
- The total number of partitions is the same as the number of reduce tasks for the job.
- Default partitioner: "**HashPartitioner**"
- Write a custom Partitioner:

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

Total number of partitions

```
main(){
   job.setPartitionerClass(MyPartitioner.class);
}
```

Map Phase: Combiner



Map Phase: 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

Counters

- What is a counters class?
  - It represents a set of global counters, defined either by the Map-Reduce framework or applications.
  - Each Counter can be of any Enum type.
  - Counters are bunched into Counters.Groups
  - It is automatically aggregated over Map/Reduce phases
- What is it for?
  - It is used to determine if and how often a particular event occurred during a job execution.
    - E.g.: count I/O bytes, memory usage, etc.
  - Operations API:
    - **incrCounter**(**Enum**<?> key, long amount)
    - **incrCounter**(**String** group, **String** counter, long amount)

Example

- Define a *enum* type for a counter under main()

```
static enum SYMBOL_COUNTER
{  COMMA,  COLON,  EXCLAMATION,  STOP};
```

- Call the incremental method in any class

```
context.getCounter(SYMBOL_COUNTER.COMMA).increment(1);
```

- Find your counter after the job completes

```
job.waitForCompletion(true);
Counters counters = job.getCounters();
Counter c1 = counters.findCounter(SYMBOL_COUNTER.COMMA);
System.out.println(c1.getDisplayName()+":"+c1.getValue());
```

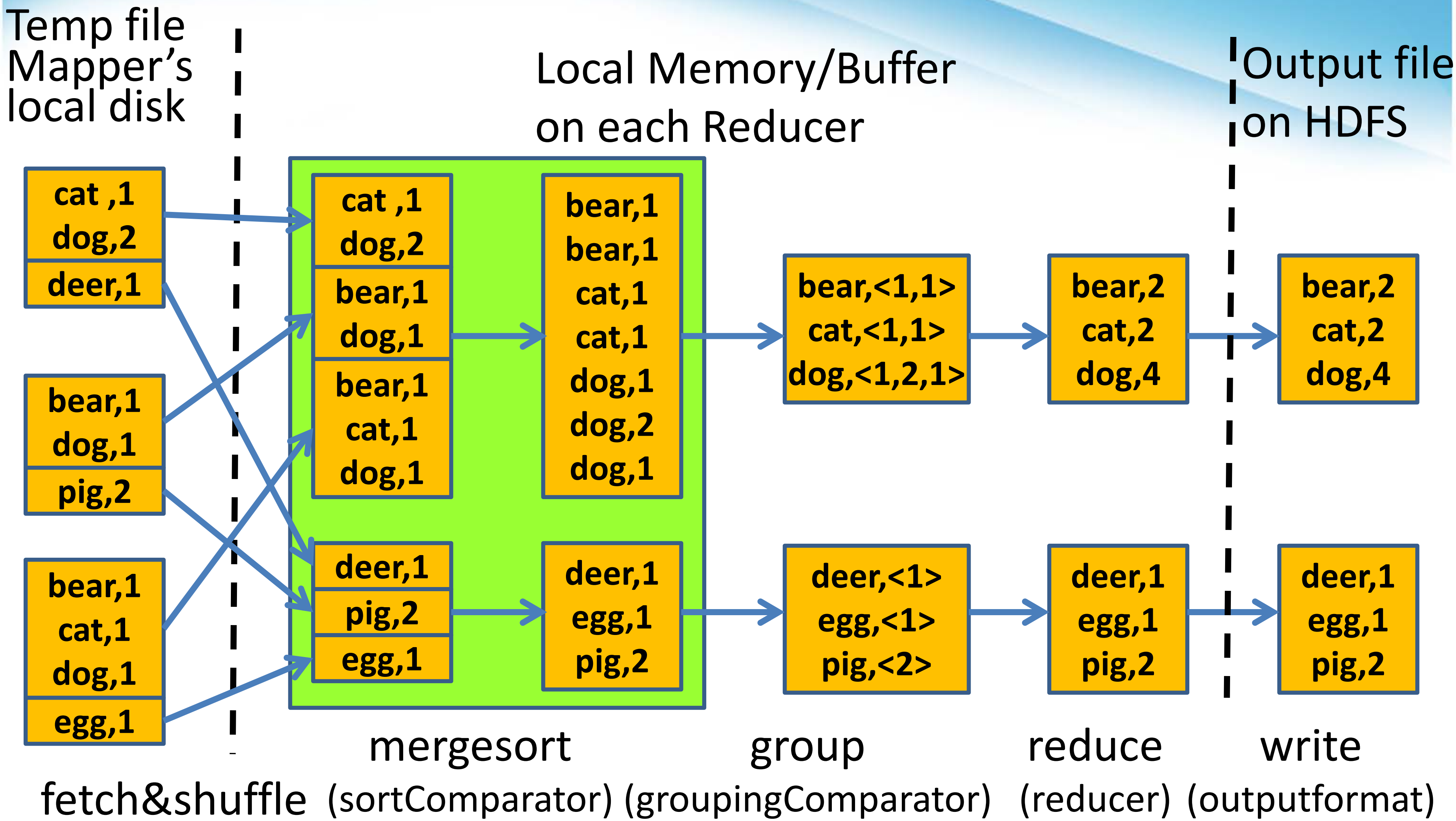
- Also print out automatically to standout

```
TestPoint3D$SYMBOL_COUNTER
        COMMA=6
File Input Format Counters
        Bytes Read=39
File Output Format Counters
        Bytes Written=87
```

Custom value types
Combiner
Partitioner
Counter
GroupingComparator
SortComparator
DistributedCache

ADVANCED PROG.

Reduce Phase: sortComparator



Reduce Phase: sortComparator

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

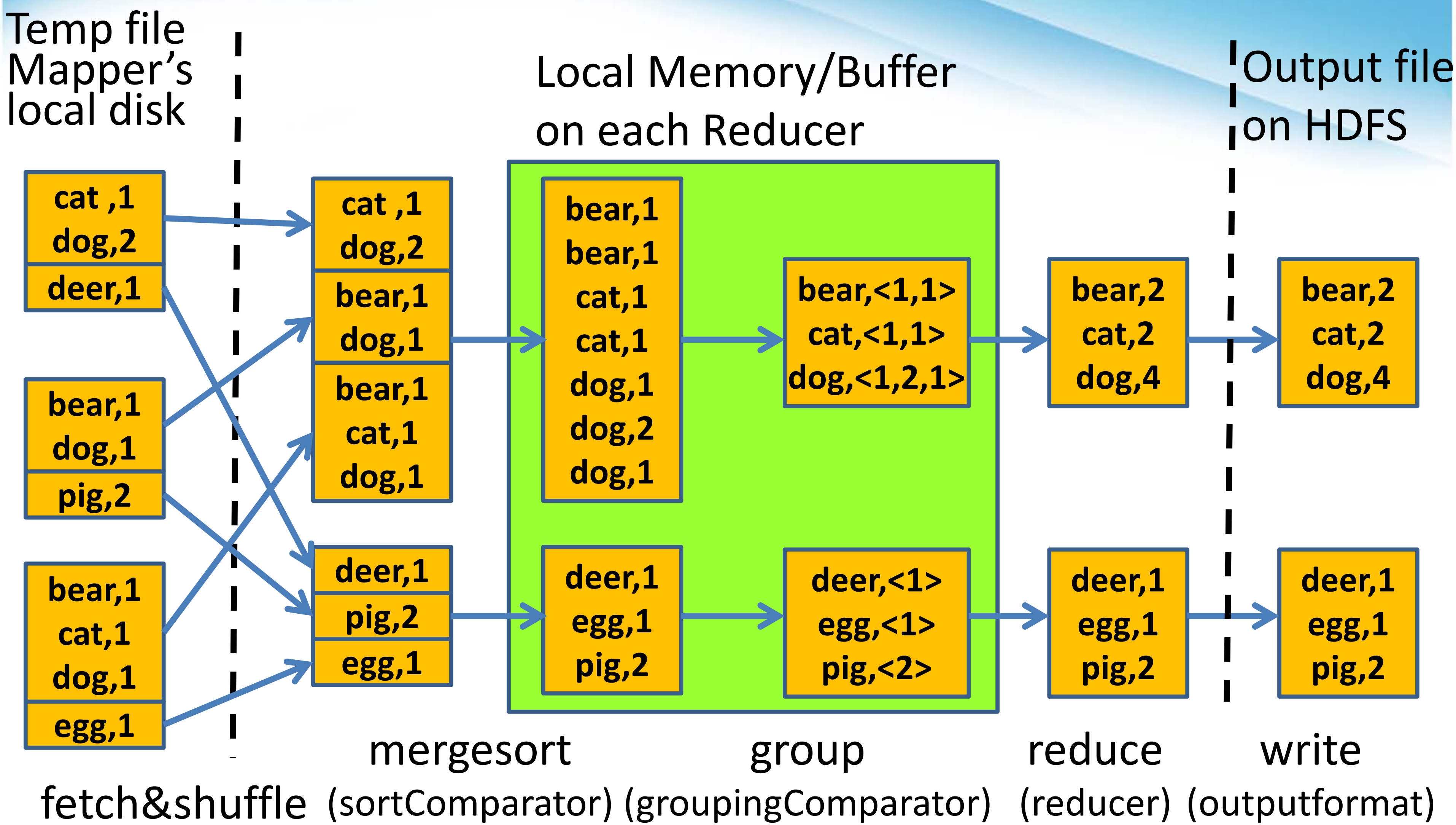
Reduce Phase: sortComparator

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

```
public static class MySortComprator extends WritableComparator {
    protected MySortComprator() { 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]);
    }
}
```

```
main(){
   job.setSortComparatorClass(MySortComparator.class);
}
```

Reduce Phase: groupingComparator



Reduce Phase: groupingComparator

- <K,V> pairs are grouped together if their ***keys*** are compared as equal by using a comparator called **groupingComparator**
  - The comparator can be set by "**job.setGroupingComparatorClass()**"
  - The comparator must implement the "***rawComparator***" **interface** *or extend "**writeComparator**" **class***
    - Override the function: **compare**
- If multiple keys in the same group, "**sortComparator**" is used to decide the key for the group
  - Input: <A1, V1>, <A2, V2>, <A3, V3>, <B1, V4>, <B2, V5>
  - Grouping comparator to just compare the first letter
  - Output: (A1, {V1,V2,V3}); (B1, {V4,V5});

Reduce Phase: groupingComparator

- Only compare the first letter

```
public static class  MyGroupComp extends WritableComparator {
    protected MyGroupCom() { 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;
     }
}
```

```
main(){
   job.setGroupingComparatorClass(MyGroupComp.class);
}
```

Custom value types

Combiner

Partitioner

Counter

GroupingComparator

SortComparator

DistributedCache

ADVANCED PROG.

DistributedCache

- What is it?
  - A facility provided by the Map-Reduce framework to **cache read-only files** (text, archives, jars etc.) needed by applications on compute nodes
  - The framework will copy the necessary files on to the slave node before execution, and remove them automatically after execution
- What is it for?
  - Distribute dictionary text for mapper and reducer
  - Map-side join: cache the smaller table
  - As a rudimentary software distribution mechanism: jar files
- How to specify the cached files?
  - The files are specified via urls (hdfs:// or http://) of a **file system**
  - The url must be accessible by every machine in the cluster

Example

- Copy the requisite files to the FileSystem:

```
$ bin/hadoop fs -copyFromLocal [local_src_files] [hdfs_dst_dir]
```

- Setup the application's job in main()

```
job.addCacheFile(new URI("/myapp/lookup.dat"));
job.addCacheArchive(new URI("/myapp/map.zip");
job.addFileToClassPath(new Path("/myapp/mylib.jar"));
```

- Use the cached files in the Mapper or Reducer through the context object and

```
import org.apache.hadoop.fs.FileSystem;
import java.net.URI;
import java.io.BufferedReader;
import java.io.InputStreamReader;
```

```
Configuration conf = context.getConfiguration();
FileSystem fs = FileSystem.get(conf);
URI[] cacheFiles = context.getCacheFiles();
Path filePath = new Path(cacheFiles[0].getPath());
BufferedReader bf = new BufferedReader(
        new InputStreamReader(fs.open(filePath)));
```

SECONDARYSORT EXAMPLE

SecondarySort

- What is SecondarySort?
  - **Sorting values** associated with a key in the reduce phase
- Examples:
  - Input: A dump of the temperature data with 4 columns
    year, month, day, daily_temperature

    2012, 01, 01, 5
    2012, 01, 02, 45
    2012, 01, 03, 35
    …
    2001, 11, 01, 46
    2001, 11, 02, 47
    2001, 11, 03, 48

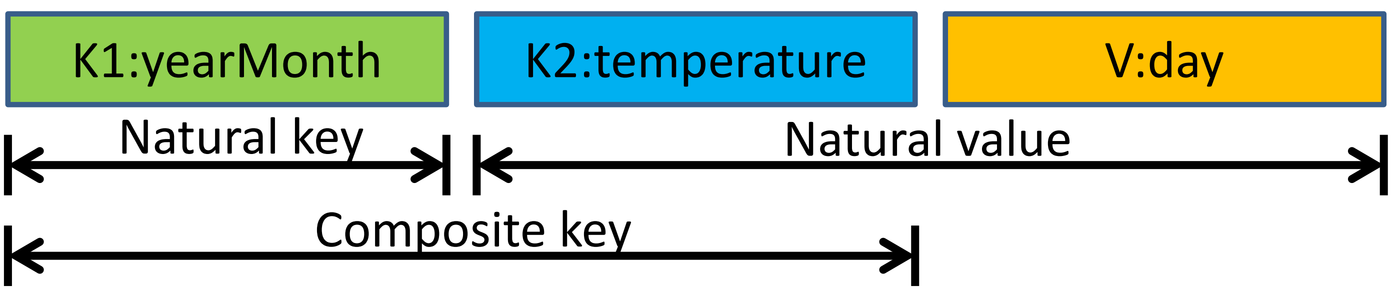
  - Output: The temperature for every year-month with the values sorted

    2012-01: 5, 35, 45
    2001-11, 46, 47, 48
    ….

SecondarySort

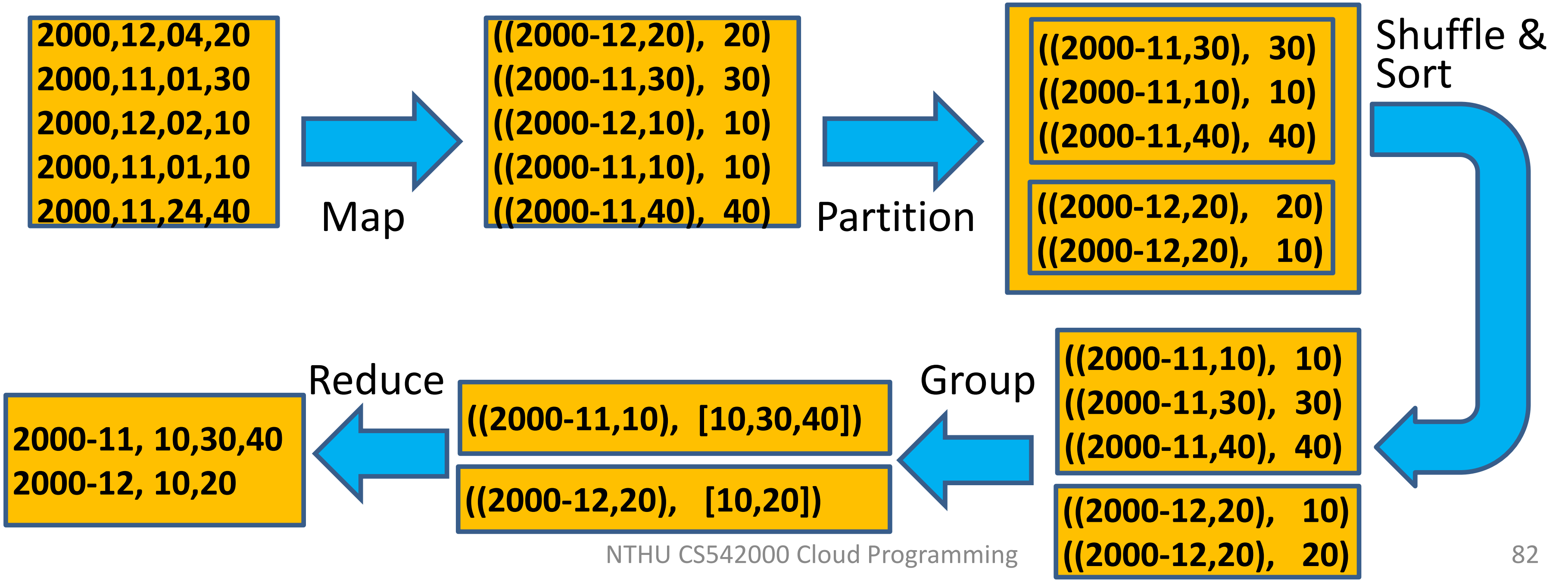
- Soltion1:
  - having the reducer **buffer all of the values** for a given *key*
  - then doing an **in-reducer sort** on the values

  ➔might cause the reducer to run out of memory
- Solution2:
  - Trick MapReduce to sort the reducer values
  - *Value-to-Key Conversion* design pattern: “Creating a **composite key** by adding a part of, or the entire value to, the natural key to achieve your sorting objectives”



SecondarySort

- Implementation details:
  - Map Output Key: {yearMonth}+{temperature}
  - Map Output Value: temperature
  - Partitioner: by yearMonth
  - sortComparator: by yearMonth and then ascending temp.
  - groupingComparator: by yearMonth



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
- Hadoop tutorial:
  - https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html