Hadoop MapReduce

DSCI 55x

Wensheng Wu

Hadoop

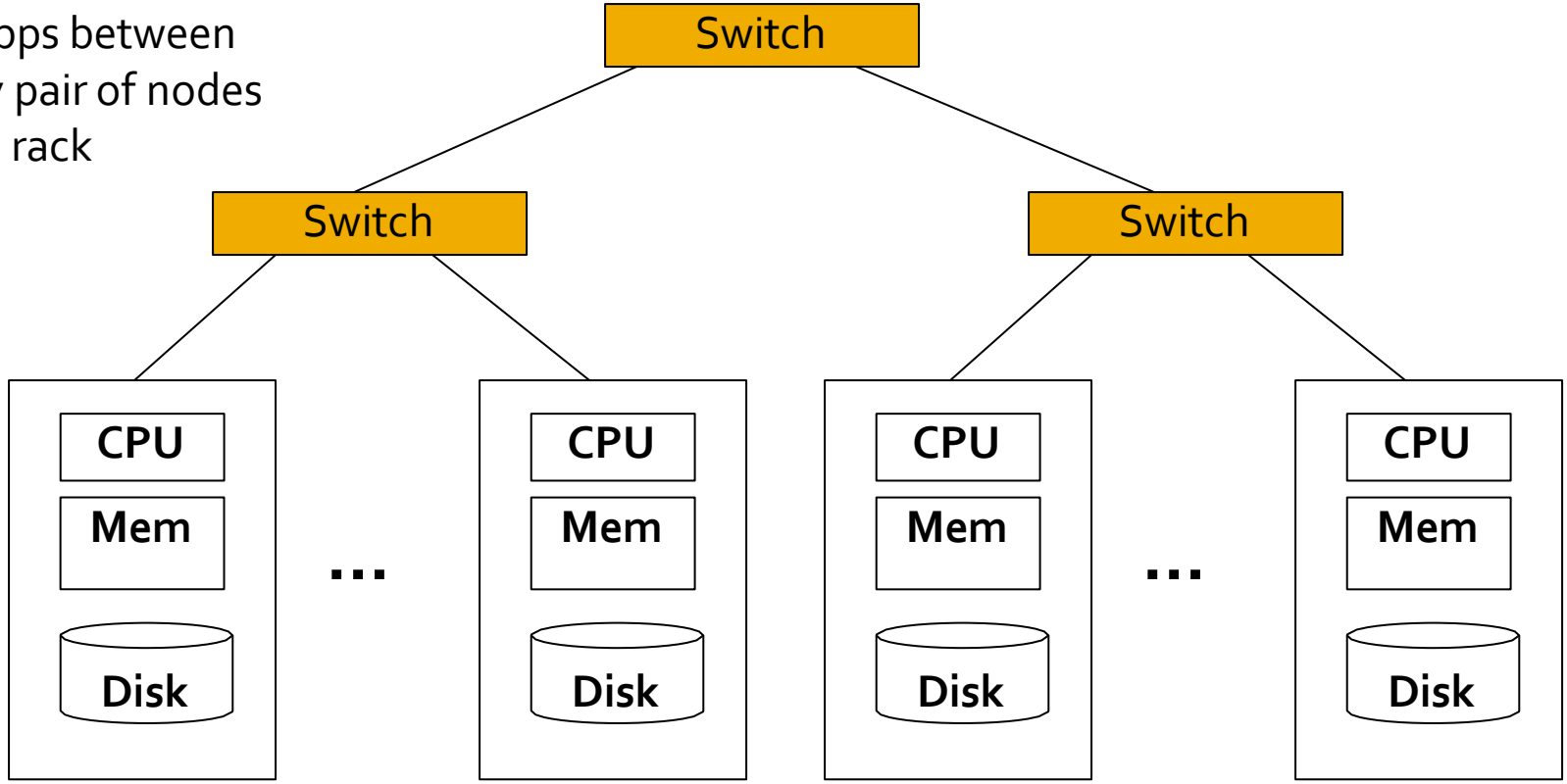
- A large-scale distributed batch-processing infrastructure

- Large-scale:
  - Handle a large amount of data and computation
- Distributed:
  - Distribute data & work across a number of machines
- Batch processing
  - Process a series of jobs without human intervention

Cluster Architecture

2-10 Gbps backbone between racks

1 Gbps between any pair of nodes in a rack



Each rack contains 16-64 nodes

In 2011 it was guestimated that Google had 1M machines, http://bit.ly/Shn0RO

Roadmap

- Hadoop architecture 
  - HDFS
  - MapReduce
- MapReduce implementation
- Compile & run MapReduce programs

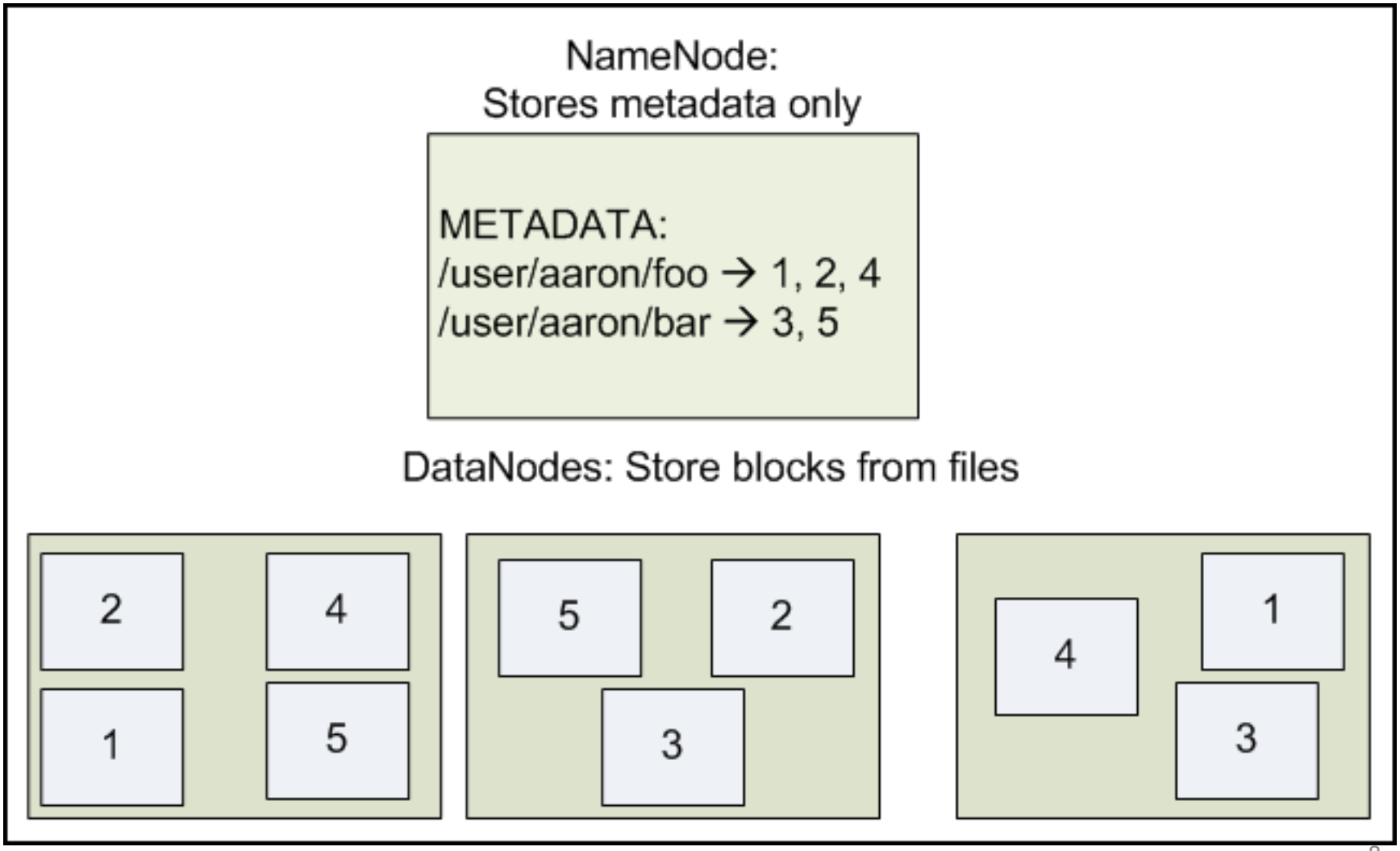
Key components

- HDFS (Hadoop distributed file system)
  - Distributed data storage with high reliability
- MapReduce
  - A parallel, distributed computational paradigm
  - With a simplified programming model

HDFS

- Data are distributed among multiple data nodes
  - Data nodes may be added on demand for more storage space
- Data are replicated to cope with node failure
  - Typically replication factor = 2/3
- Requests can go to any replica
  - Removing the bottleneck (in single file server)

HDFS architecture



HDFS has …

- A single NameNode, storing meta data:
  - A hierarchy of directories and files
  - Attributes of directories and files
  - Mapping of files to blocks on data nodes
- A number of DataNodes:
  - Storing contents/blocks of files

Compute nodes

- Data nodes are compute nodes too
- Advantage:
  - Allow schedule computation close to data

HDFS also has …

- A SecondaryNameNode
  - Maintaining checkpoints of NameNode
  - For recovery

- In a single-machine setup
  - all nodes correspond to the same machine

Metadata in NameNode

- NameNode has an inode for each file and dir
- Record attributes of file/dir such as
  - Permission
  - Access time
  - Modification time
- Also record mapping of files to blocks

Mapping information in NameNode

- E.g., file /user/aaron/foo consists of blocks 1, 2, and 4

- Block 1 is stored on data nodes 1 and 3
- Block 2 is stored on data nodes 1 and 2
- …

Block size

- HDFS: 128MB
  - Much larger than disk block size (4KB)

- Why larger size in HDFS?
  - Reduce metadata required per file
  - Fast streaming read of data (since larger amount of data are sequentially laid out on disk)
  - Good for workload with largely sequential read of large file

Roadmap

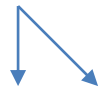
- Hadoop architecture
  - HDFS
  - MapReduce 
- MapReduce implementation
  - Map & reduce functions and tasks
  - Shuffling & merging
  - Input and output format
  - Combiner
- Compile & run MapReduce programs

MapReduce job

- A MapReduce job consists of a number of
  - Map tasks
  - Reduce tasks
  - (Internally) shuffle tasks

Map, reduce, and shuffle tasks

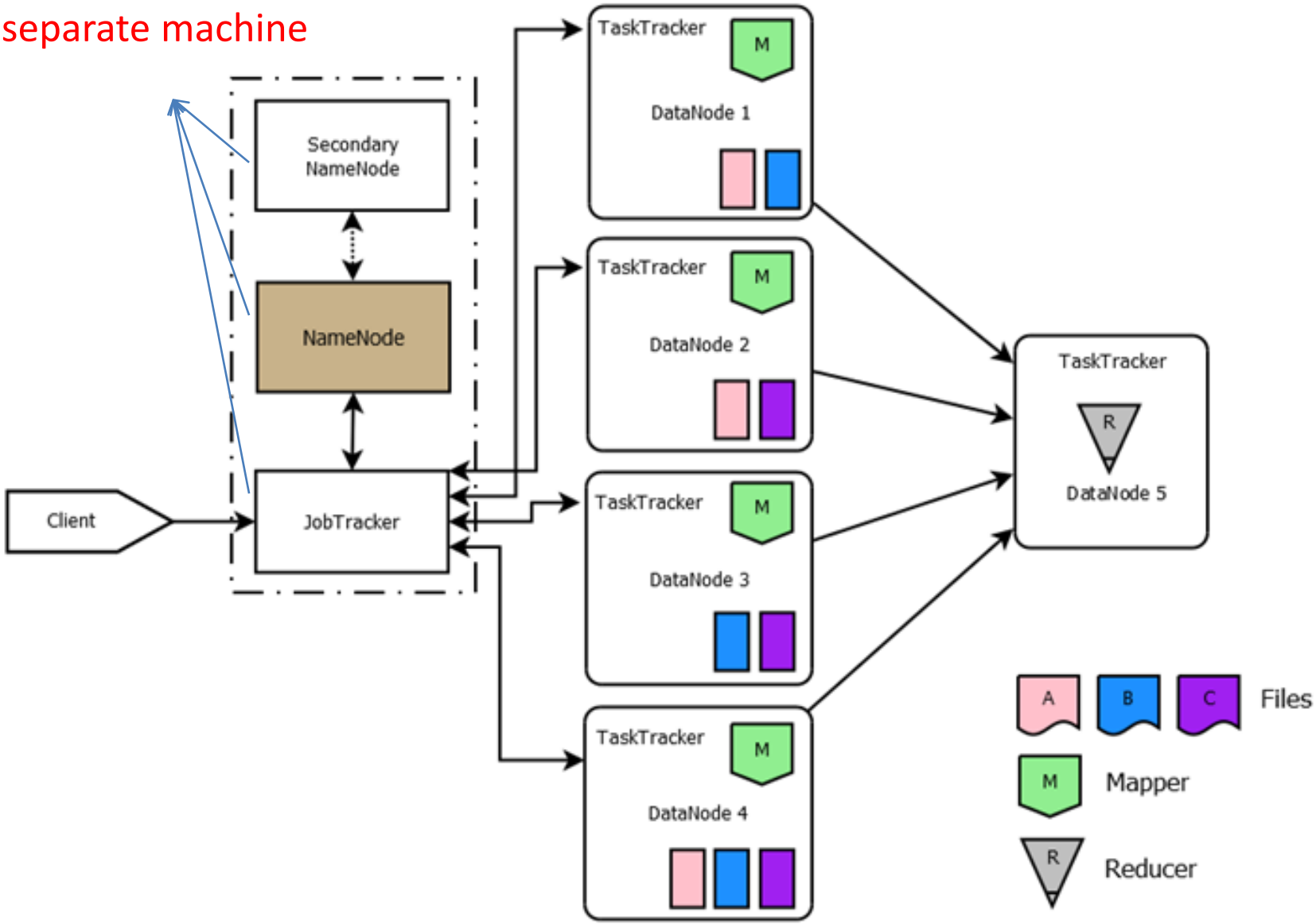
- Map task performs data transformation
- Reduce task combines results of map tasks
- Shuffle task sends output of map tasks to right reduce tasks
  - M1 M2 (=> key-values)



  - R1 R2

Hadoop cluster

Each may be run on
a separate machine



Job tracker

- Takes requests from clients (MapReduce programs)
- Ask name node for location of data
- Assign tasks to task trackers near the data
  - Compared to: bring data to computation
- Reassign tasks if failed

Task tracker

- Accept (map, reduce, shuffle) tasks from job trackers
- Send heart beats to job trackers: I am alive
- Monitor status of tasks and notify job tracker

Roadmap

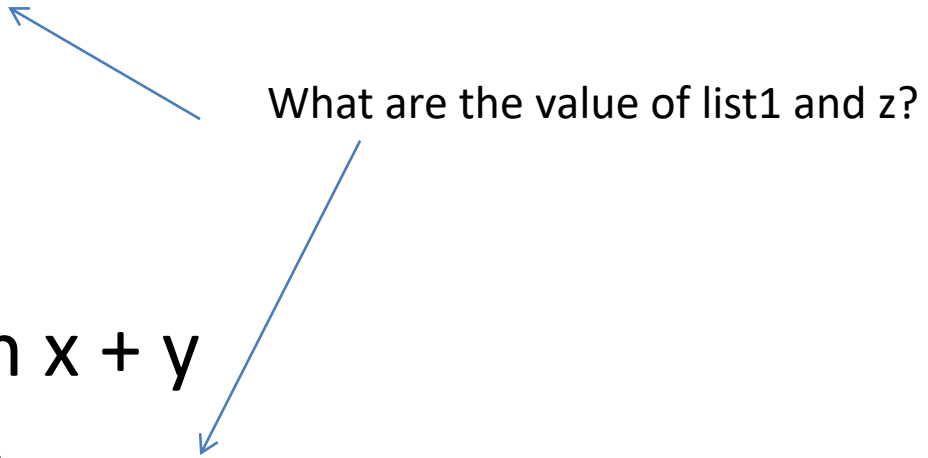
- Hadoop architecture
  - HDFS
  - MapReduce
- MapReduce implementation
  - Map & reduce functions and tasks 
  - Shuffling & merging
  - Input and output format
  - Combiner
- Compile & run MapReduce programs

Roots in functional programming

- Functional programming languages:
  - Python, Lisp (list processor), Scheme, Erlang, Haskell
- Two functions:
  - Map: mapping a list => list
  - Reduce: reducing a list => value
- map() and reduce() in Python
  - https://docs.python.org/2/library/functions.html#map

map() and reduce() in Python

- list = [1, 2, 3]
- def sqr(x): return x ** 2
- list1 = map(sqr, list)



- def add(x, y): return x + y
- z = reduce(add, list)

Lambda function

- Anonymous function (not bound to a name)

- list = [1, 2, 3]

- list1 = map(lambda x: x ** 2, list)
- z = reduce(lambda x, y: x + y, list)

How is reduce() in Python evaluated?

- z = reduce(f, list) where f is add function

- Initially, z (an accumulator) is set to list[0]
- Next, repeat z = add(z, list[i]) for each i > 0
- Return final z

- Example: z = reduce(add, [1, 2, 3])
  - i = 0, z = 1; i = 1, z = 3; i = 2, z = 6

Python 3

- map() returns an iterator
  - g = map(lambda x: x* 2, [1,2,3])
  - next(g)

- reduce() moved to functools
  - import functools as fc
  - fc.reduce(lambda U, x: U + x, [1,2,3])
  - fc.reduce(lambda U, x: U + x, [1,2,3], 0)

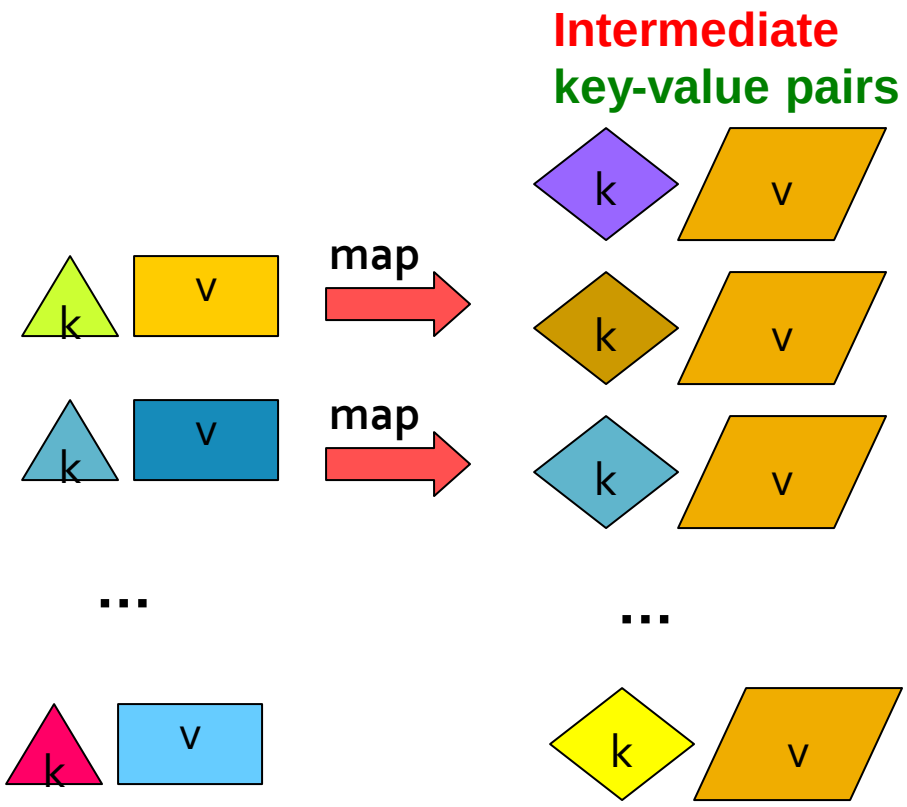
MapReduce

- Map function:
  - Input: <k, v> pair
  - Output: a list of <k', v'> pairs // ('LA', 2), ('LA', 3)
- Reduce function: ('LA', [2, 3])
  - Input: <k', list of v's> (note k's are output by map)
  - Output: a list of <k'', v''> pairs

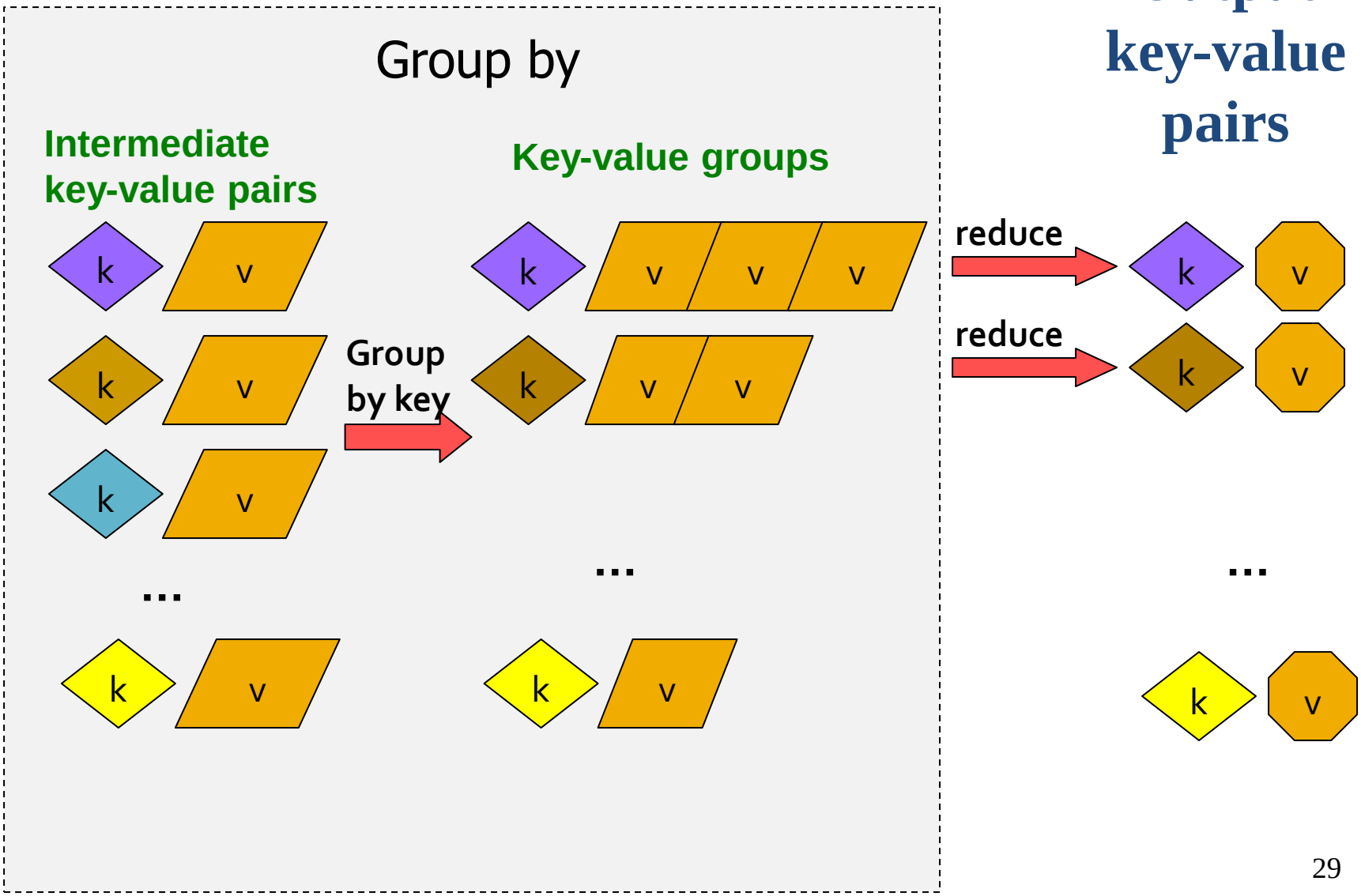
MapReduce: The Map Step

Input: key-value pairs

Output: (intermediate) key-value pairs



MapReduce: The Reduce Step

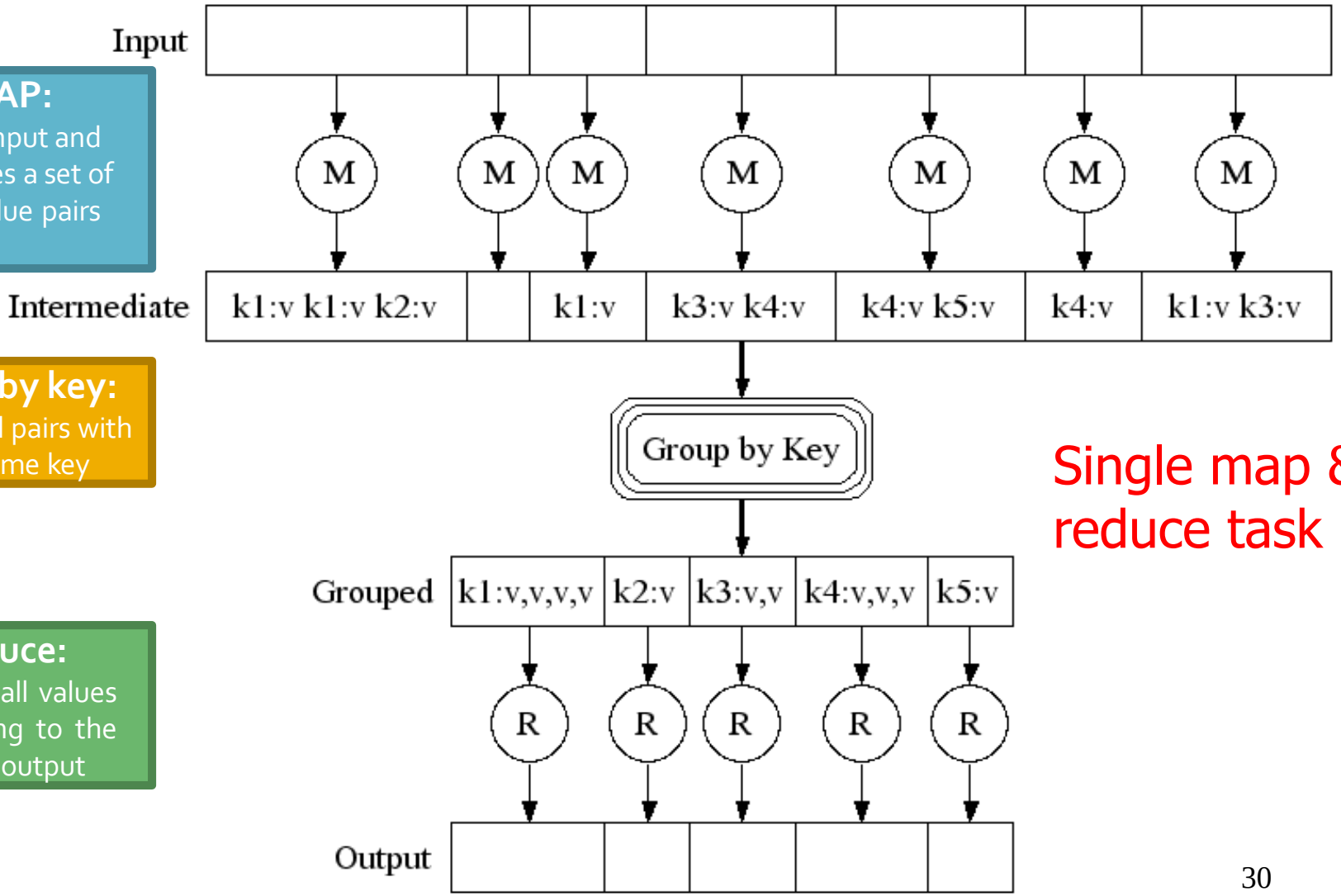


Map-Reduce: A diagram

MAP:
Read input and produces a set of key-value pairs

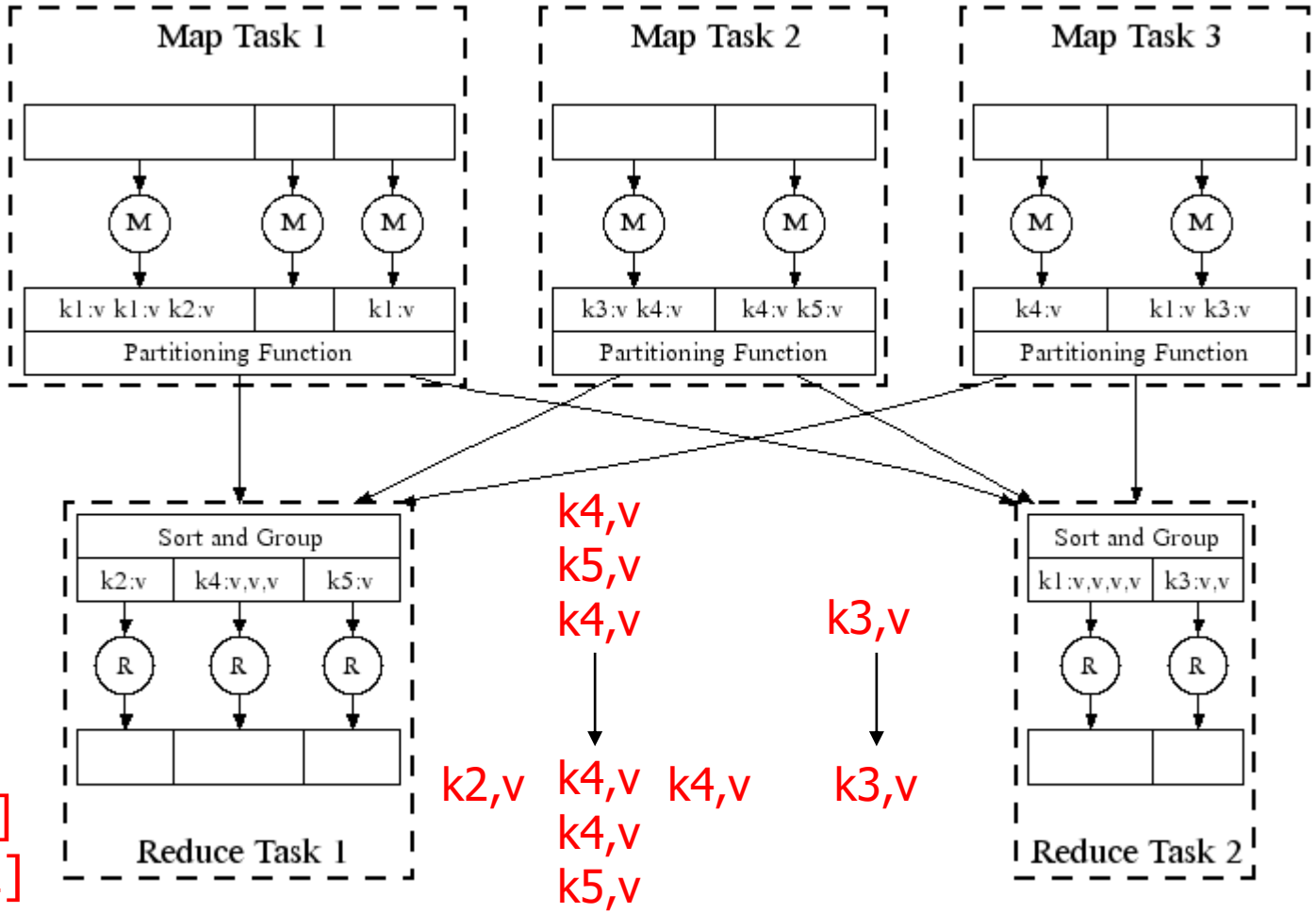
Group by key:
Collect all pairs with the same key

Reduce:
Reduce all values belonging to the key and output



Single map & reduce task

Map-Reduce: In Parallel



Multiple map & reduce tasks

Example: WordCount

- Counting the number of occurrences of words in a collection of documents

- helloworld.txt (stored under an input directory, among possible other documents)
  - hello world
  - hello this world
  - hello hello world

Example: WordCount

- Map function:
  - Input: <offset of line, line> // line = a line of text in a document
  - Output: for each word in line, output <word, 1>

- Reduce function:
  - Input: <word, list of 1’s>
  - Output: <word, count> where count is the number of 1's in the input list

Word Count Using MapReduce

Pseudocode

```
map(key, value):
// key: line offset; value: line content
for each word w in value:
      output (w, 1)
```

```
reduce(key, values):
// key: a word; values: an iterator over counts
      result = 0
      for each count v in values:
            result += v
      output (key, result)
```

Group by

- System groups the intermediate key-value pairs from map tasks by key

- E.g., <hello, 1> <hello, 1> <hello, 1> <this, 1>

  => <hello, [1, 1, 1]>, <this, [1]>

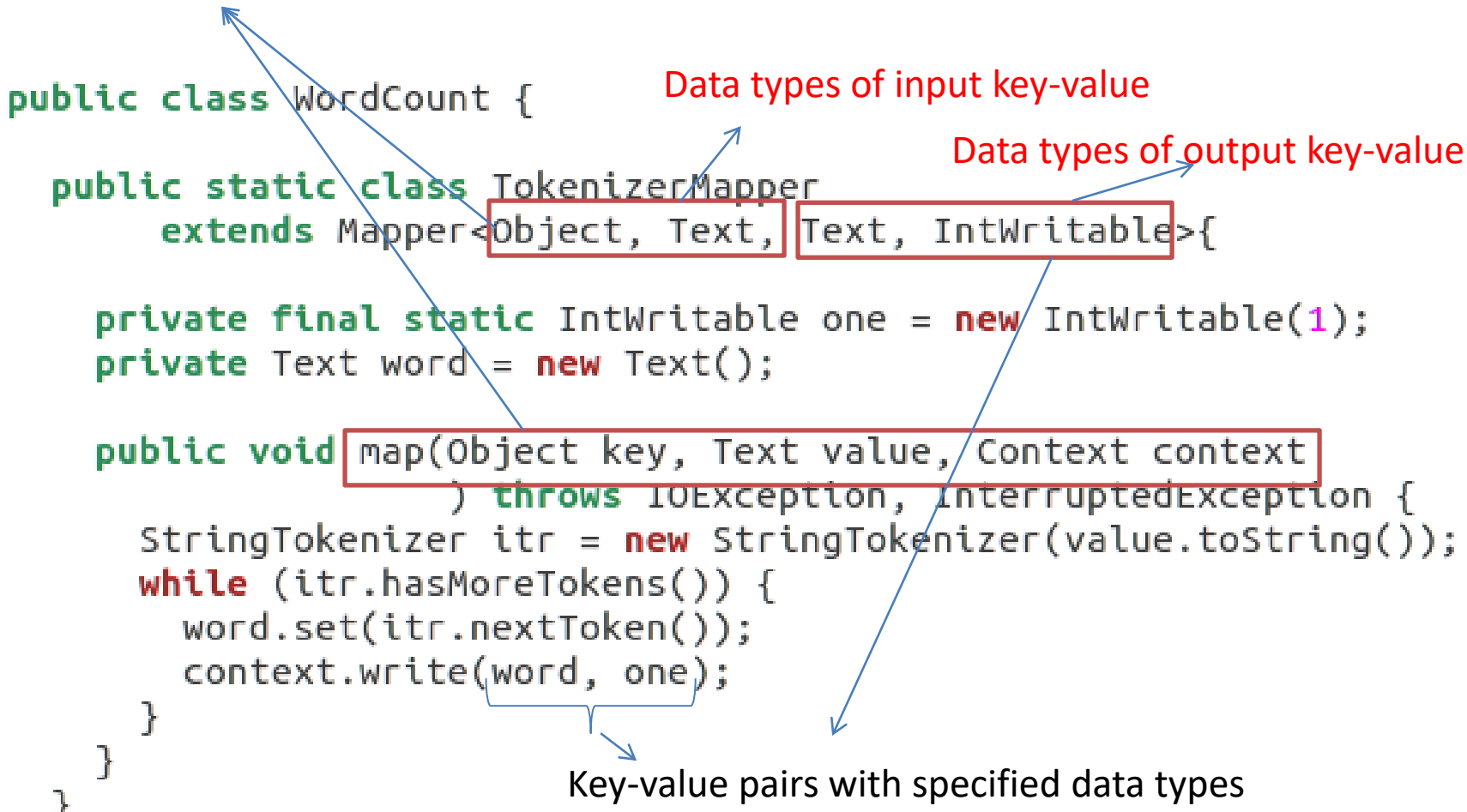
Example: WordCount

- hadoop jar wc.jar WordCount input output

- Output:
  - hello 4
  - this 1
  - world 3

WordCount: Mapper

Object can be replaced with LongWritable




```
public class WordCount {

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

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

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }
```

WordCount: Reducer

Data types of input key-value
Should be the same as output data types of mapper

Data types of output key-value

```
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);
  }
}
```

A list of values

Checking map input

```
public void map(Object key, Text value, Context context
                ) throws IOException, InterruptedException {
 System.out.println("map input: key=" + key + ", value=" + value);

  StringTokenizer itr = new StringTokenizer(value.toString());
   while (itr.hasMoreTokens()) {
     word.set(itr.nextToken());
     context.write(word, one);
   }
}
```

- map input: key=0, value=hello world
- map input: key=12, value=hello this world
- map input: key=29, value=hello hello world

Checking reduce input

```
public void reduce(Text key, Iterable<IntWritable> values,
                   Context context
                   ) throws IOException, InterruptedException {

  System.out.print("reduce input: key=" + key + ", values=");

  int sum = 0;
  for (IntWritable val : values) {
    System.out.print(val + " ");
    sum += val.get();
  }

  System.out.println();
```

- reduce input: key=hello, values=1 1 1 1
- reduce input: key=this, values=1
- reduce input: key=world, values=1 1 1

Map and reduce tasks in Hadoop

- A node may run multiple map/reduce tasks
- Typically, one map task per input split (chunk of data)

“输入分片”！！

- One reduce task per partition of map output
  - E.g., partition by key range or hashing

After the map tasks have completed, their outputs are partitioned (grouped) before being sent to the reduce tasks. Each partition of the mapped output data is processed by one reduce task.
The partitioning can be done based on different strategies:
a. "Partition by key range": The mapped output is partitioned based on ranges of the key values. For example, all keys starting with 'a' go to one partition, 'b' to another, and so on.
b. "Partition by hashing": The mapped output is partitioned by applying a hash function on the keys. This ensures that records with the same key are sent to the same partition (and hence the same reduce task).

Mapper and Reducer

- Each map task runs an instance of Mapper
  - Mapper has a map function
  - Map task invokes the map function of the Mapper once for each input key-value pair
- Each reduce task runs an instance of Reducer
  - Reducer has a reduce function
  - Reduce task invokes the reduce function of the Reducer once for every different intermediate key

Reduce function

- Input: a key and an iterator over the values for the key
- Values are NOT in any particular order
- Reduce function is called once for every different key (received by the reduce task)

Roadmap

- Hadoop architecture
  - HDFS
  - MapReduce
- MapReduce implementation
  - Map & reduce functions and tasks
  - Shuffling task
  - Input and output format
  - Combiner
- Compile & run MapReduce programs

Shuffling

- Process of distributing intermediate key-values to the right reduce tasks

- It is the only communication among map and reduce tasks
  - Individual map tasks do not exchange data directly with other map tasks
  - They are not even aware of existence of their peers

Shuffling

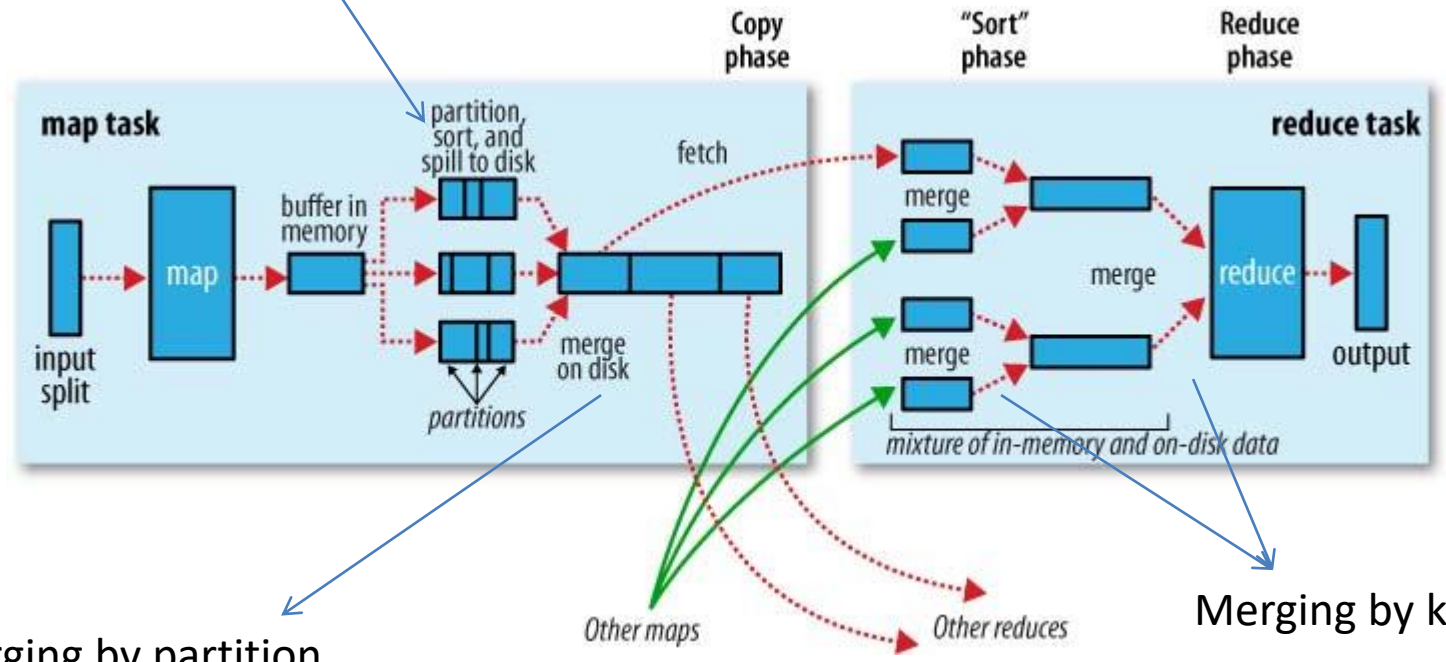
- Begins when a map task completed on a node
- All intermediate key-value pairs with the same key are sent to the same reducer task
- Partitioning method defined in Partitioner class
  - Default rule: partition by hashing the key

Internals of shuffling

- Map side
  - Partition, sort, spill & merge
- Reduce side
  - Fetch & merge

Shuffling process

Keys in the same partition are sorted (keys from different partitions may not be)



Merging by partition

Merging by key

From "Hadoop: The Definitive Guide" by T. White.

Map side

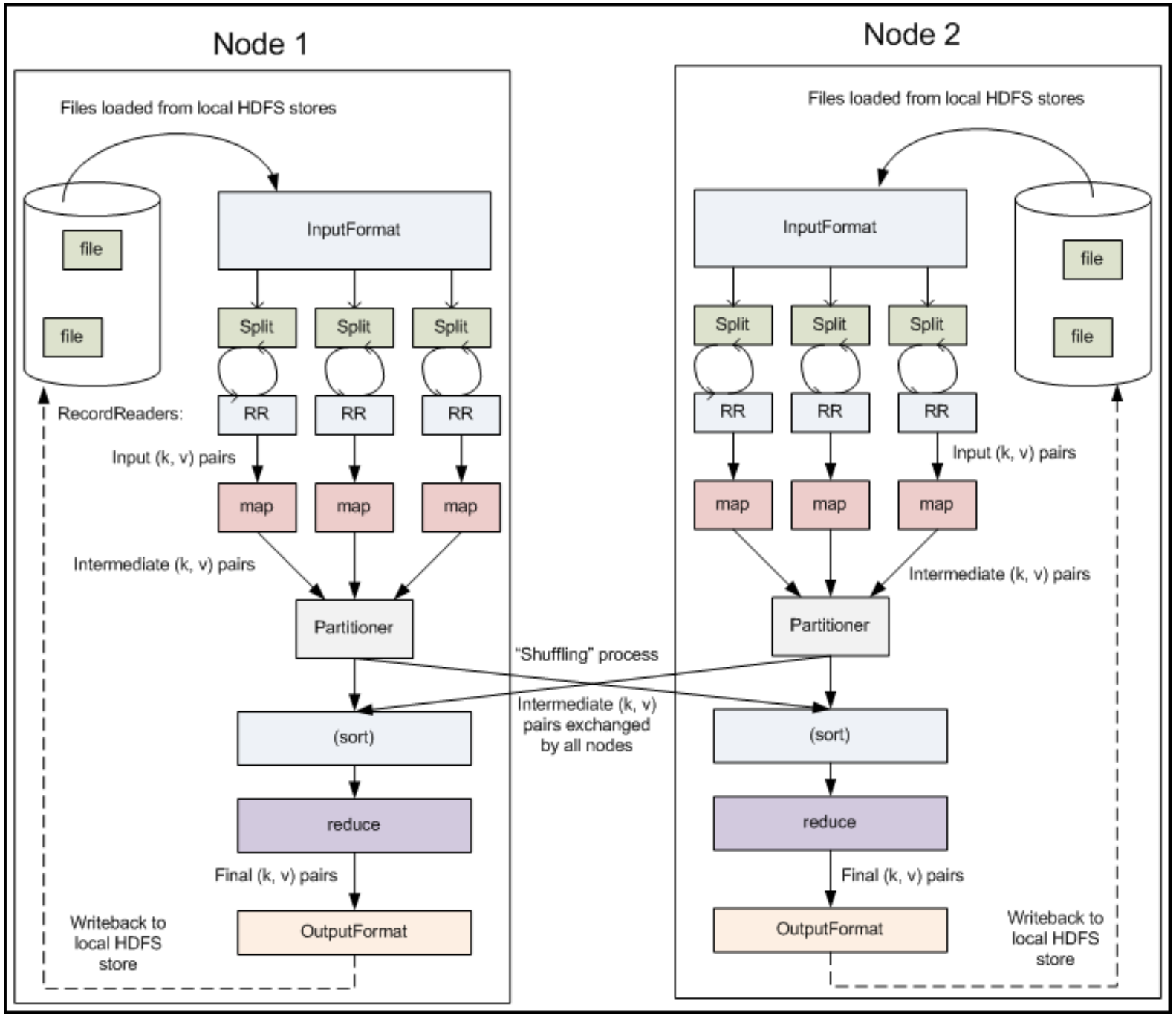
- Partition data in the buffer into R parts
  - R = # of reduce tasks
- Sort data in each partition by key
- Spill/write data in the buffer to disk
- Merge the spills
- Notify job tracker: output complete

Reduce side

- Task tracker notified by job tracker: data ready
- Fetch/copy data from map side
- Merge the data
  - Some data may sit on disk once fetched
  - This depends on the buffer size
- Figure out groups from sorted data

Roadmap

- Hadoop architecture
  - HDFS
  - MapReduce
- MapReduce implementation
  - Map & reduce functions and tasks
  - Shuffling & merging
  - Input and output format 
  - Combiner
- Compile & run MapReduce programs


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

InputFormat

- Determine how input files are split and read
- Defined in the Java interface InputFormat
- Job:
  - Split input file into chunks called InputSplits
  - Implement RecordReader to read data from splits

InputFormat implementations

- FileInputFormat (input from files in given dirs)
- DBInputFormat (input data from a database)
- CombineFileInputFormat (input data by combining multiple files)
- …

FileInputFormat

- Job:
  - Takes paths to files
  - Read all files in the paths
  - Divide each file into one or more InputSplits
- Subclasses:
  - TextInputFormat
  - KeyValueTextInputFormat
  - SequenceFileInputFormat

Subclasses of FileInputFormat

InputFormat:	Description:	Key:	Value:
TextInputFormat	Default format; reads lines of text files	The byte offset of the line	The line content
KeyValueTextInputFormat	Parses lines into key, value pairs	Everything up to the first tab character	The remainder of the line
SequenceFileInputFormat	A Hadoop-specific high-performance binary format	user-defined	user-defined

Use non-default input format

- If input file contains tab-separated key-value pairs, e.g.,
  - John 5
  - David 6
  - …

Then you can set the input format explicitly using:

- job.setInputFormatClass(KeyValueTextInputFormat.class);
  - Both key and value are of type "Text"

InputSplits

- If a file is big, multiple splits may be created
  - Typical split size = 128MB
- A map task is created for each split
  - i.e., a chunk of some input file

RecordReader (RR)

- InputFormat defines an instance of RR
  - E.g., TextInputFormat provides LineRecordReader
- LineRecordReader
  - Form a key-value pair for every line of file
  - Data type for key: LongWritable; value: Text
- Reader is repeatedly called
  - Until all data in the split are processed

OutputFormat

- Define the format of output from Reducers
  - Output stored in a file
- Defined in the Java interface OutputFormat
- Implemention: FileOutputFormat
  - Subclasses: TextOutputFormat, SequenceFileOutputFormat

OutputFormat

OutputFormat:	Description
TextOutputFormat	Default; writes lines in "key \t value" form
SequenceFileOutputFormat	Writes binary files suitable for reading into subsequent MapReduce jobs

Outputs

- All Reducers write to the same directory
  - Each writes a separate file, named part-r-nnnnn
  - r: output from Reducers
  - nnnnn: partition id associated with reduce task
- Output directory
  - Set by FileOutputFormat.setOutputPath() method
- OutputFormat defines a RecordWriter
  - which handles the write

WordCount: setting up job

```
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 = Job.getInstance(conf, "word count");
  job.setJarByClass(WordCount.class);
  job.setMapperClass(TokenizerMapper.class);
  job.setCombinerClass(IntSumReducer.class);
  job.setReducerClass(IntSumReducer.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
  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);
}
```

Take multiple directories as input

Set output key and value types for both map and reduce tasks.
If Mapper has different types, use setMapOutputKeyClass and setMapOutputValueClass

setJarByClass

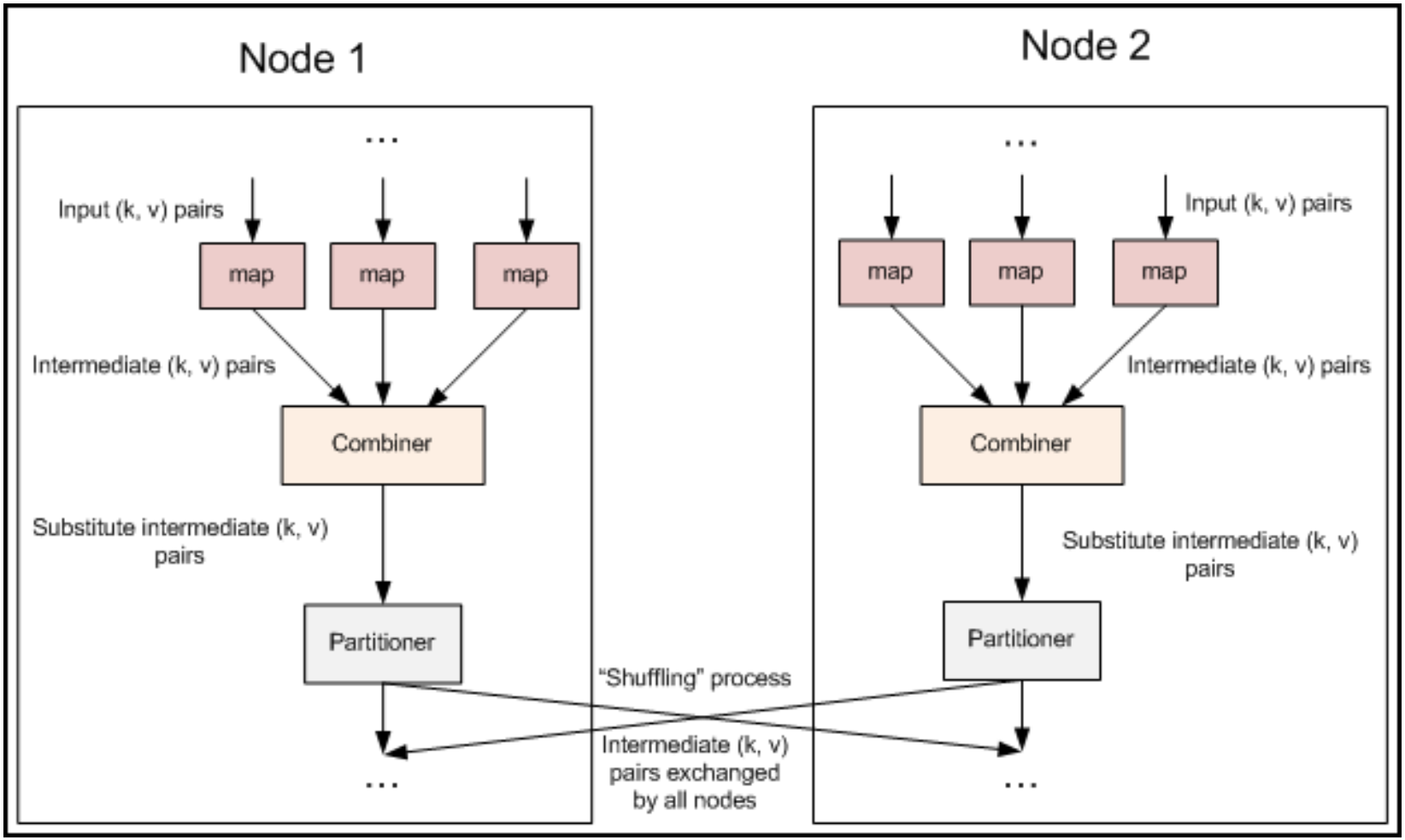
- Mapper and reducer code may be in a different jar
  - E.g., a jar in Hadoop class search path
  - (wc.jar contains only the job submission code)

- setJarByClass tells Hadoop where to find jar containing mapper and reducer code
  - By specifying its class name (e.g., WordCount)

Roadmap

- Hadoop architecture
  - HDFS
  - MapReduce
- MapReduce implementation
  - Map & reduce functions and tasks
  - Shuffling & merging
  - Input and output format
  - Combiner 
- Compile & run MapReduce programs

Combiner


Node 1
Node 2
Input (k, v) pairs
map
map
map
Intermediate (k, v) pairs
Combiner
Substitute intermediate (k, v) pairs
Partitioner
"Shuffling" process
Intermediate (k, v) pairs exchanged by all nodes
Input (k, v) pairs
map
map
map
Intermediate (k, v) pairs
Combiner
Substitute intermediate (k, v) pairs
Partitioner

Combiner

- Run on the node running the Mapper
  - Perform local (or mini-) reduction
- Combine Mapper results
  - Before they are sent to the Reducers
  - Reduce communication costs
- E.g., may use a combiner in WordCount
  - (cat, 1), (cat, 1), (cat, 1) => (cat, 3)
  - One key-value pair per unique word

Without combiner

- Mapper 1 outputs:
  - (cat, 1), (cat, 1), (cat, 1), (dog, 1)
- Mapper 2 outputs:
  - (dog, 1), (dog, 1), (cat, 1)
- Suppose there is only one Reducer
  - It will receive: (cat, [1, 1, 1, 1]), (dog, [1, 1, 1])

Implementing combiner

- May directly use the reduce function
  - If it is commutative and associative
  - Meaning operations can be grouped & performed in any order
- Operation 'op' is commutative if:
  - A op B = B op A
- Op is associative if:
  - A op (B op C) = (A op B) op C

Example: without combiner

- Consider two map tasks
  - M1 => 1, 2, 3 for some key x
  - M2 => 4, 5 for the same key
- Reducer adds all values for x
  - Result = (((1 + 2) + 3) + 4) + 5

Example: with combiner

- M1 => 1, 2, 3 => combiner: (1 + 2) + 3 => 6
- M2 => 4, 5 => combiner: 4 + 5 => 9

- Reducer now 6 + 9,
  - I.e., (((1 + 2) + 3) + (4 + 5))
  - Question: is it the same as ((((1 + 2) + 3) + 4) + 5)?

- Yes, since '+' is associative

Example: with combiner

- M1 => 1, 2, 3 => combiner: (1 + 2) + 3 => 6
- M2 => 4, 5 => combiner: 4 + 5 => 9

- Reducer may also compute 9 + 6,
  - I.e., (4 + 5) + ((1 + 2) + 3)
  - Since values may arrive at reducer in any order
  - Question: is it the same as (((1 + 2) + 3) + 4) + 5?

- Yes, since '+' is also commutative

General requirements

- To use reduce function 'f' for a combiner
  - Consider a set of values S and its subsets S_1, …, S_k
  - It must be that: f(S) = f(f(S_1), …, f(S_k))
    - sum(S) = sum(sum(S1), sum(S2))
    - avg([1,2,3,4,5]) = avg(avg([1,2,3]), avg([4,5]))

    - avg([1,2,3,4,5]) ?= f(sum_cnt([1,2,3]), sum_cnt([4,5]))
    - f: adding up sum => global sum; local cnt => global cnt
    - Combiners: [1,2,3] => ('LA', (6,3)); [4,5] => ('LA',(9, 2))
    - Reducers: ('LA', [(6,3), (9,2)]) => 'LA', 15/5 = 3
- E.g., in WordCount:
  - f = sum
  - S = a list of integers

Commutative and associative

- Examples
  - Sum
  - Max
  - Min
- Non-examples
  - Count
  - Average
  - Median

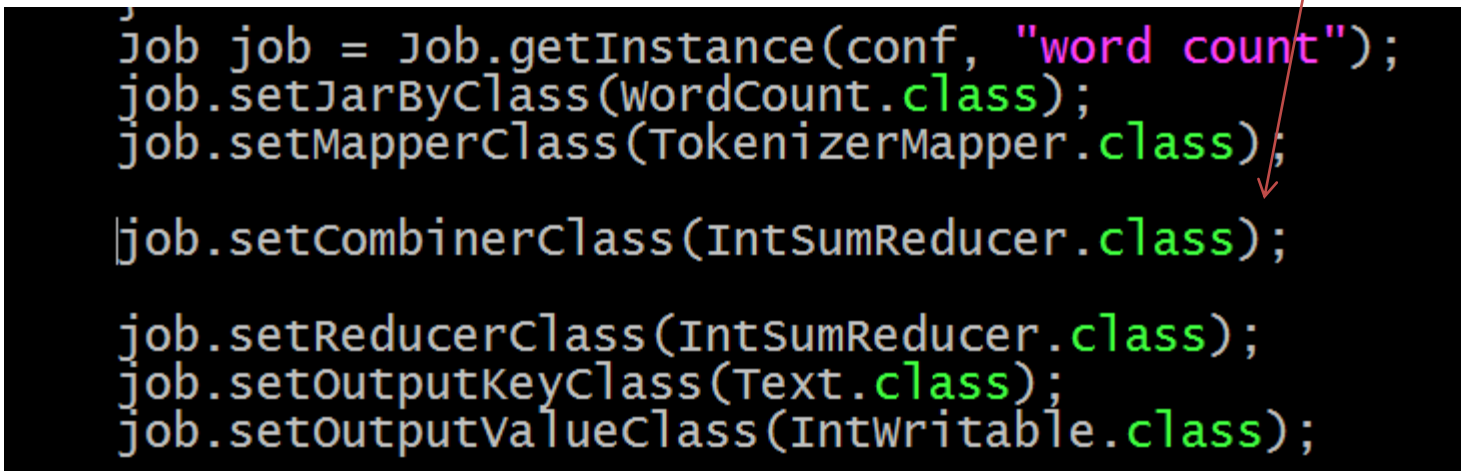
Custom combiner

- Key & value data type of both input & output
  - Should be same as that of the output of Mapper
  - (Also the same as that for input to Reducer)

- So if Mapper outputs (Text, IntWritable), then:
  - public static class MyCombiner
    extends Reducer<Text, IntWritable,
    Text, IntWritable> {
    …
    }

Enabling combiner

- job.setCombinerClass(IntSumReducer.class)
  - To use reduce function for combiner



Roadmap

- Hadoop architecture
- MapReduce framework
- Compile & run MapReduce programs
  - On Amazon EC2

Hadoop installation

- Install the Hadoop package
  - Log into your EC2 instance and then execute:
    - wget https://downloads.apache.org/hadoop/common/hadoop-3.3.1/hadoop-3.3.1.tar.gz

- Might want to remove installation package (~200MB) to save space

Install java sdk

- sudo yum install java-1.8.0-devel
  - 1.8 is needed for spark

Setup environment variables

- Edit ~/.bashrc by adding the following:
  - export JAVA_HOME=/usr/lib/jvm/java
  - export HADOOP_CLASSPATH=${JAVA_HOME}/lib/tools.jar
  - export HADOOP_HOME=/home/ec2-user/hadoop-3.3.1
  - export PATH=${JAVA_HOME}/bin:${HADOOP_HOME}/bin:${PATH}

This assumes that you installed hadoop right under home directory

- source ~/.bashrc
  - This is to get the new variables in effect
  - Or you may also log out and log in again

Run Hadoop in standalone mode

- Comment out <property> element (if exists) in <your hadoop installation directory>/etc/hadoop/core-site.xml as shown below
  -

```
<configuration>
<!--property>
        <name>fs.defaultFS</name>
        <value>hdfs://localhost:9000</value>
</property-->

</configuration>
~
```

Examples

- You may find example codes here:
  - /home/ec2-user/hadoop-3.3.1/share/hadoop/mapreduce/sources/hadoop-mapreduce-examples-3.3.1-sources.jar
- unzip hadoop-mapreduce-examples-3.3.1-sources.jar
- Find WordCount.java and more under
  - org/apache/hadoop/examples/

WordCount.java

- Copy WordCount.java to a working directory of your choice
  - E.g., ~/dsci551

- Comment out first line:
  - // package org.apache.hadoop.examples;

Compile & run

- Go to the directory that has WordCount.java
- hadoop com.sun.tools.javac.Main WordCount.java
- jar cf wc.jar WordCount*.class
- hadoop jar wc.jar WordCount input-hello output-hello

More examples

- Two input splits
- Multiple reducers
- SQL implemented in MapReduce
  - Selection, projection, group by, having
  - Join

Two-split example

- Now input directory has two files

  => Two splits (hence two map tasks) generated, one for each file

```
[ec2-user@ip-172-31-52-194 inf551]$ ls input-hello2/
helloworld1.txt  helloworld2.txt
[ec2-user@ip-172-31-52-194 inf551]$ cat input-hello2/helloworld1.txt
hello world
hello this world
hello hello world
[ec2-user@ip-172-31-52-194 inf551]$ cat input-hello2/helloworld2.txt
hello that world
hello this world
hello hello that world
```

```
16/11/14 22:26:32 INFO input.FileInputFormat: Total input paths to proce
ss : 2
16/11/14 22:26:32 INFO mapreduce.JobSubmitter: number of splits:2
16/11/14 22:26:32 INFO mapreduce.JobSubmitter: Submitting tokens for job
: job_local462978203_0001
```

Processing one split (Mapper)

- Split for "helloworld2.txt"
  - This shows map function is called 3 times
  - One for each line of text

```
16/11/14 22:26:33 INFO mapred.MapTask: Processing split: file:/home/ec2-
user/hadoop-2.7.3/inf551/input-hello2/helloworld2.txt:0+57
16/11/14 22:26:33 INFO mapred.MapTask: (EQUATOR) 0 kvi 26214396(10485758
4)
16/11/14 22:26:33 INFO mapred.MapTask: mapreduce.task.io.sort.mb: 100
16/11/14 22:26:33 INFO mapred.MapTask: soft limit at 83886080
16/11/14 22:26:33 INFO mapred.MapTask: bufstart = 0; bufvoid = 104857600
16/11/14 22:26:33 INFO mapred.MapTask: kvstart = 26214396; length = 6553
600
16/11/14 22:26:33 INFO mapred.MapTask: Map output collector class = org.
apache.hadoop.mapred.MapTask$MapOutputBuffer
map input: key=0, value=hello that world
map input: key=17, value=hello this world
map input: key=34, value=hello hello that world
```

Processing one split (combiner)

- This shows input key-values for combiner
  - Note combiner uses the same reduce function

```
16/11/14 22:26:33 INFO mapred.MapTask: kvstart = 26214396(104857584); kv
end = 26214360(104857440); length = 37/6553600
reduce input: key=hello, values=1 1 1 1
reduce input: key=that, values=1 1
reduce input: key=this, values=1
reduce input: key=world, values=1 1 1
16/11/14 22:26:33 INFO mapred.MapTask: Finished spill 0
```

Process the other split (Mapper)

- helloworld1.txt

```
16/11/14 22:26:33 INFO mapred.MapTask: Processing split: file:/home/ec2-
user/hadoop-2.7.3/inf551/input-hello2/helloworld1.txt:0+47
16/11/14 22:26:33 INFO mapred.MapTask: (EQUATOR) 0 kvi 26214396(10485758
4)
16/11/14 22:26:33 INFO mapred.MapTask: mapreduce.task.io.sort.mb: 100
16/11/14 22:26:33 INFO mapred.MapTask: soft limit at 83886080
16/11/14 22:26:33 INFO mapred.MapTask: bufstart = 0; bufvoid = 104857600
16/11/14 22:26:33 INFO mapred.MapTask: kvstart = 26214396; length = 6553
600
16/11/14 22:26:33 INFO mapred.MapTask: Map output collector class = org.
apache.hadoop.mapred.MapTask$MapOutputBuffer
map input: key=0, value=hello world
map input: key=12, value=hello this world
map input: key=29, value=hello hello world
```

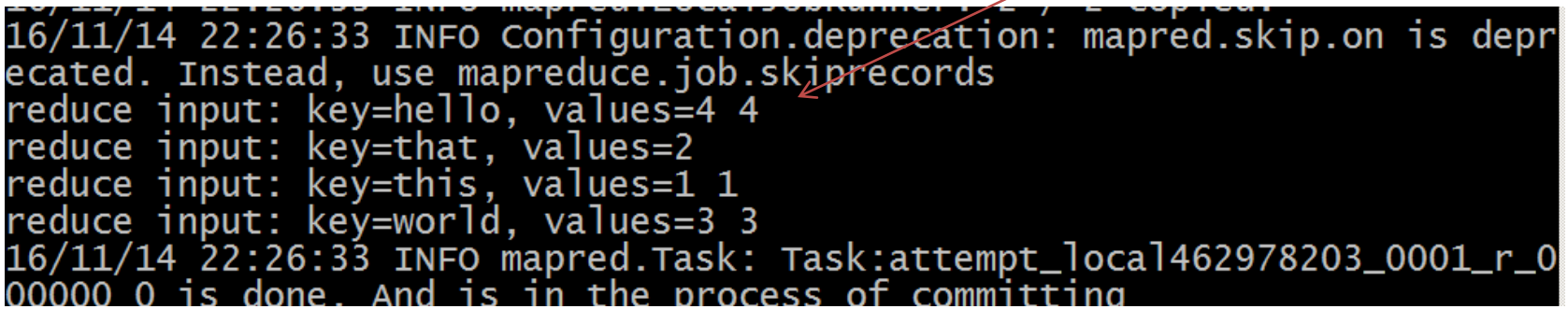
Process the other split (combiner)

- This shows the input to the combiner for the 2nd Map task

```
16/11/14 22:26:33 INFO mapred.MapTask: kvstart = 26214396(104857584); kv
end = 26214368(104857472); length = 29/6553600
reduce input: key=hello, values=1 1 1 1
reduce input: key=this, values=1
reduce input: key=world, values=1 1 1
```

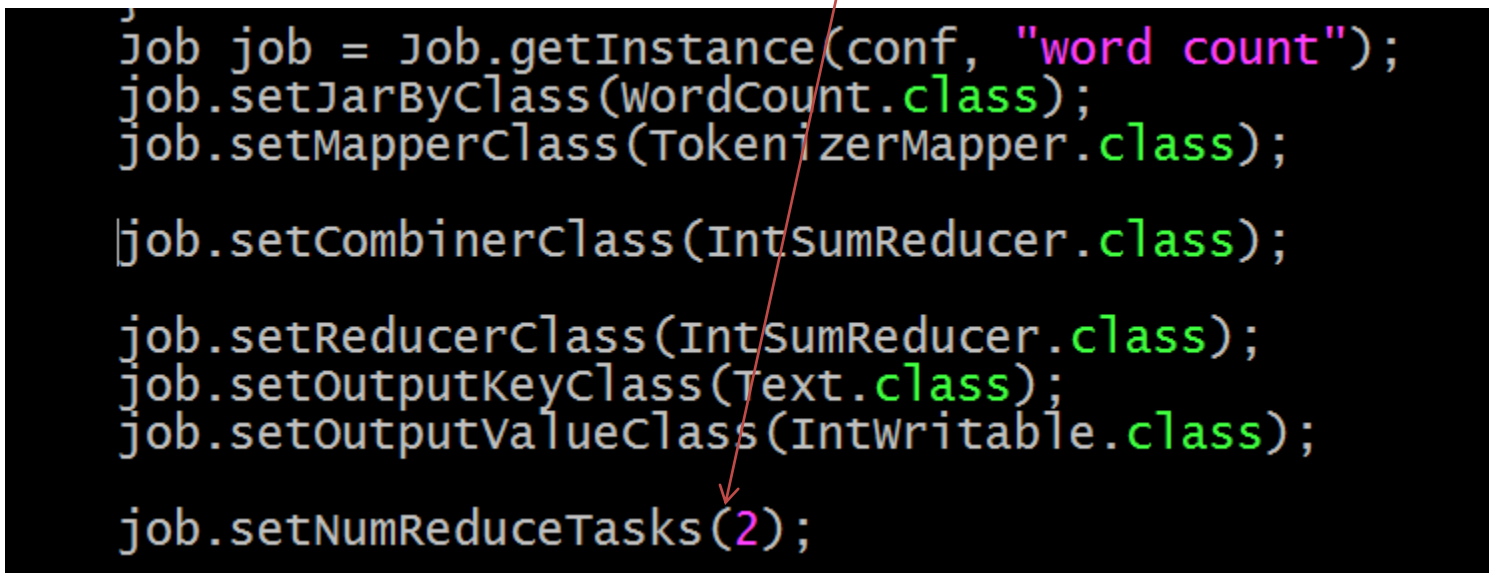
Reducer input (one Reducer)

- Assume only one Reducer is used
  - Note the input values now contain local counts



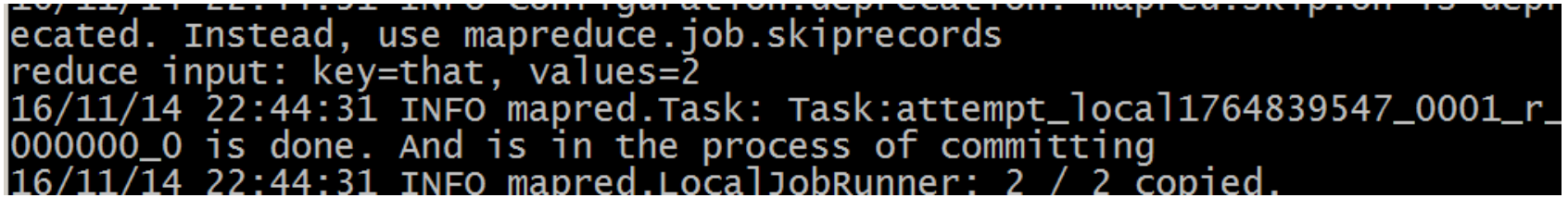
Setting number of Reducers

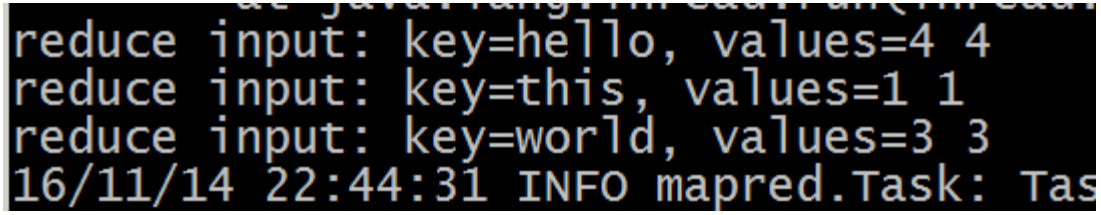
- job.setNumReduceTasks(2);
  - Two reduce tasks



Two-Reducer case

- Note "that" is in one partition
- "hello", "this", "world" in the other



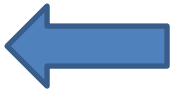


Two output files

- Part-r-00000 for partition 00000
- Part-r-00001 for partition 00001

```
[ec2-user@ip-172-31-52-194 output-hello2-r2]$ ls
part-r-00000  part-r-00001  _SUCCESS
[ec2-user@ip-172-31-52-194 output-hello2-r2]$ cat part-r-00000
that    2
[ec2-user@ip-172-31-52-194 output-hello2-r2]$ cat part-r-00001
hello   8
this    2
world   6
```

More examples

- Two input splits
- Multiple reducers
- SQL implemented in MapReduce 
  - Selection, projection, group by, having
  - Join

Example

100,John,25,M
200,Mary,23,F
300,David,23,M
400,Bill,26,M
500,Jennifer,20,F
600,Maria,28,F

```
Map(key, value):
toks = tokenize(value, ',')
name = toks[1]
gender = toks[3]

If (gender == 'M'):
  output(name, 1)
```

(John, 1)
(David, 1)
…

```
SELECT name
FROM Emp
WHERE gender = 'M'
```

Example

100,John,25,M
200,Mary,23,F
300,David,23,M
400,Bill,26,M
500,Jennifer,20,F
600,Maria,28,F

```
Map(key, value):
 toks = tokenize(value, ',')
 gender = toks[3]
 name = toks[1]

 if (gender == 'M'):
     output(name, 1)
```

```
SELECT name
FROM Emp
WHERE gender = 'M'
```

Example

100,John,25,M
200,Mary,23,F
300,David,23,M
400,Bill,26,M
500,Jennifer,20,F
600,Maria,28,F

```
Map(key, value):
  Toks = tokenize(value, ‘,’)
  name =  toks[1]
  gender = toks[3]

  if gender == 'M':
     output(name, 1)
```

No need for reduce

```
SELECT name
FROM Emp
WHERE gender = 'M'
```

Example

- Employee.txt

Id, name, age, gender

100,John,25,M
200,Mary,23,F
300,David,23,M
400,Bill,26,M
500,Jennifer,20,F
600,Maria,28,F

```
Map(key, value):
    Toks = tokenize(value, ‘,’)
    Age = Toks[2]
    Gender = Toks[3]

    if (Age > 22):
      output(gender, 1)
```

(M,1)
(F, 1)
(M,1)
(M,1)
(F,1)

⇒ (M, [1,1,1])
⇒ (F, [1,1])

M, 3
F, 2

```
SELECT gender, count(*)
FROM Emp
WHERE age > 22
GROUP BY gender
Having count(*) > 2
```

Example

- Employee.txt

Id, name, age, gender

100,John,25,M
200,Mary,23,F
300,David,23,M
400,Bill,26,M
500,Jennifer,20,F
600,Maria,28,F

```
Map(key, value):
    Toks = tokenize(value, ‘,’)
    Age = Toks[2]
    Gender = Toks[3]

    if (Age > 22):
      output(gender, 1)
```

```
SELECT gender, count(*)
FROM Emp
WHERE age > 22
GROUP BY gender
Having count(*) > 2
```

```
Reduce(key, values):
   cnt = sum(values) // expand it yourself
   if (cnt > 2):
     output(key, cnt)
```

M, 3
F, 2

Join

- R(A, B)⋈ S(A, C)

- Map:
  - r(a, b) => (a, ('R', b))
  - s(a, c) => (a, ('S', c))

- Reduce:
  - Joining every R tuple with every S tuple with same key
  - (a, [('R', b), ('S', c1), ('S', c2)]) => (a, (b, c1)), (a, (b, c2))

Join

- Dangling tuples:
  - Key with values from only one relation
  - (a, [('R', b)]) => left dangling
  - (a, [('S', c)]) => right dangling

Implementation

- Each relation stored as a text file
  - In different input directories (say R and S)
  - E.g., R/tuples.txt, S/tuples.txt
- bin/hadoop jar join.jar R S output
- Need to use MultipleInputs class
  - org.apache.hadoop.mapreduce.lib.input.MultipleInputs

Implementation

- MultipleInputs.addInputPath(job,
  new Path(args[0]),
  KeyValueTextInputFormat.class,
  RMapper.class);

- tuples.txt contains tab-separated key-value tuples
  - john 25
  - mary 36
  - …
- RMapper handles:
  - r(a, b) => (a, ('R', b))

Removing verbose messages

- Modify:
  - /home/ec2-user/hadoop-3.1.4/etc/Hadoop/hadoop-env.sh

```
# Default log4j setting for interactive commands
# Java property: hadoop.root.logger
export HADOOP_ROOT_LOGGER=ERROR,console
```

Resources & readings

- MapReduce tutorial from Apache:
  - https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

- MapReduce tutorial from Yahoo! – module 4
  - https://developer.yahoo.com/hadoop/tutorial/module4.html

Readings

- J. Dean and S. Ghemawat, MapReduce: simplified data processing on large clusters," Communications of the ACM, vol. 51, pp. 107-113, 2008.