

CS60021: Scalable Data Mining

Sourangshu Bhattacharya

In this Lecture:

- Outline:
 - Map-reduce programming in Java
 - Map reduce programming in other languages
 - Implementation details:
 - Job and tasks
 - Shuffle and sort

Hadoop Map Reduce

- ❑ Provides:
 - ❑ Automatic parallelization and Distribution
 - ❑ Fault Tolerance
 - ❑ Methods for interfacing with HDFS for colocation of computation and storage of output.
 - ❑ Status and Monitoring tools
 - ❑ API in Java
 - ❑ Ability to define the mapper and reducer in many languages through Hadoop streaming.

Wordcount program

```
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;
```

Wordcount program - Main

```
public class WordCount {  
  
    public static void main(String[] args) throws Exception {  
        Configuration conf = new Configuration();  
        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);  
        FileInputFormat.addInputPath(job, new Path(args[0]));  
        FileOutputFormat.setOutputPath(job, new Path(args[1]));  
        System.exit(job.waitForCompletion(true) ? 0 : 1);  
    }  
}
```

Wordcount program - Mapper

```
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 program - Reducer

```
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);
}
}
```

Wordcount program - running

```
export JAVA_HOME=[ Java home directory ]
```

```
bin/hadoop com.sun.tools.javac.Main WordCount.java
```

```
jar cf wc.jar WordCount*.class
```

```
bin/hadoop jar wc.jar WordCount [Input path] [Output path]
```


Wordcount in python

Mapper.py

```
#!/usr/bin/env python

import sys

# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        #
        # tab-delimited; the trivial word count is 1
        print '%s\t%s' % (word, 1)
```

Wordcount in python

Reducer.py

```
#!/usr/bin/env python

from operator import itemgetter
import sys

# maps words to their counts
word2count = {}

# input comes from STDIN
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()

    # parse the input we got from mapper.py
    word, count = line.split('\t', 1)
    # convert count (currently a string) to int
    try:
        count = int(count)
        word2count[word] = word2count.get(word, 0) + count
    except ValueError:
        # count was not a number, so silently
        # ignore/discard this line
        pass

# sort the words lexicographically;
#
# this step is NOT required, we just do it so that our
# final output will look more like the official Hadoop
# word count examples
sorted_word2count = sorted(word2count.items(), key=itemgetter(0))

# write the results to STDOUT (standard output)
for word, count in sorted_word2count:
    print '%s\t%s' % (word, count)
```

Execution code

```
bin/hadoop dfs -ls
```

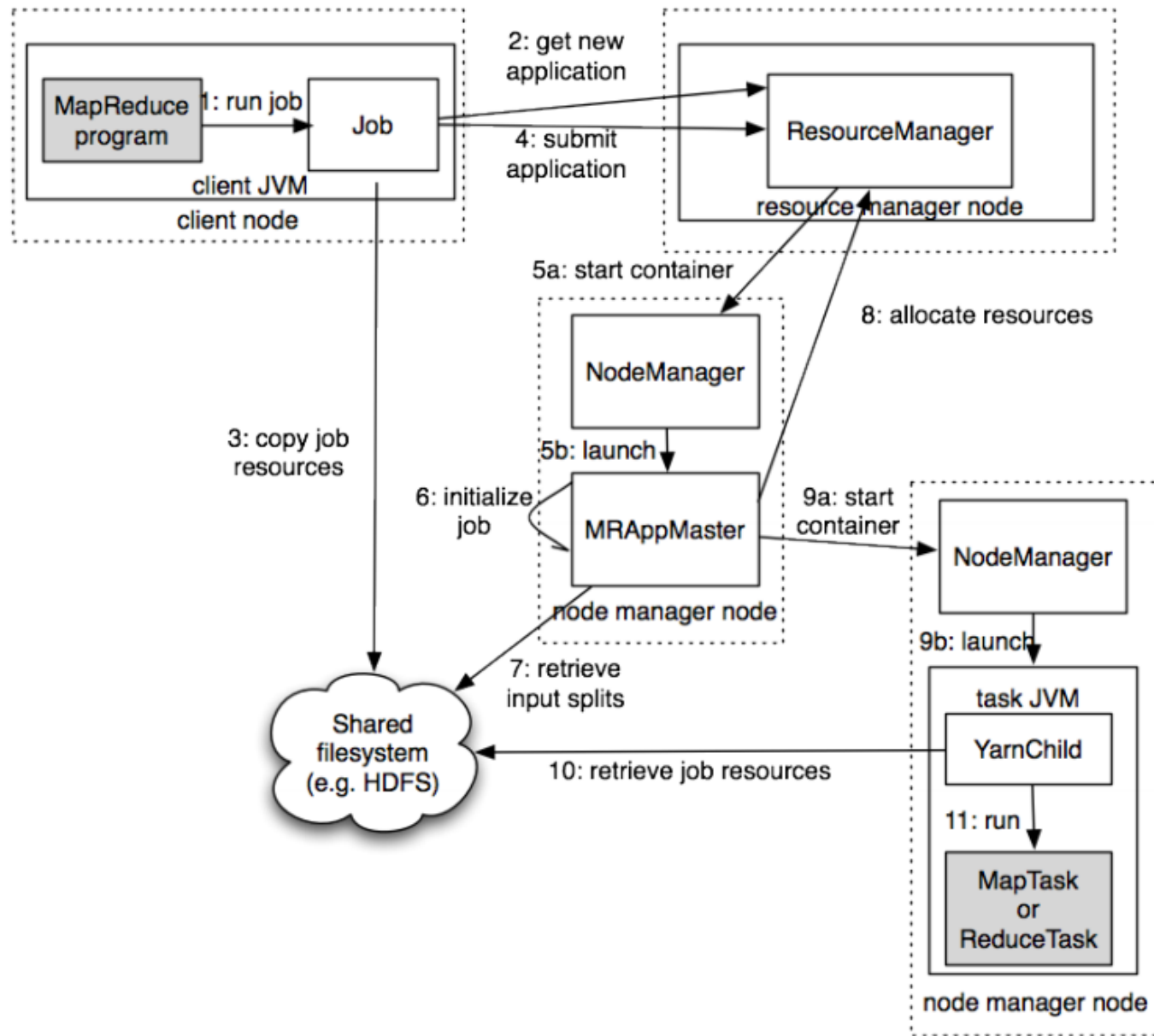
```
bin/hadoop dfs -copyFromLocal example example
```

```
bin/hadoop jar contrib/streaming/hadoop-0.19.2-streaming.jar -file  
wordcount-py.example/mapper.py -mapper wordcount-py.example/mapper.py  
-file wordcount-py.example/reducer.py -reducer wordcount-  
py.example/reducer.py -input example -output java-output
```

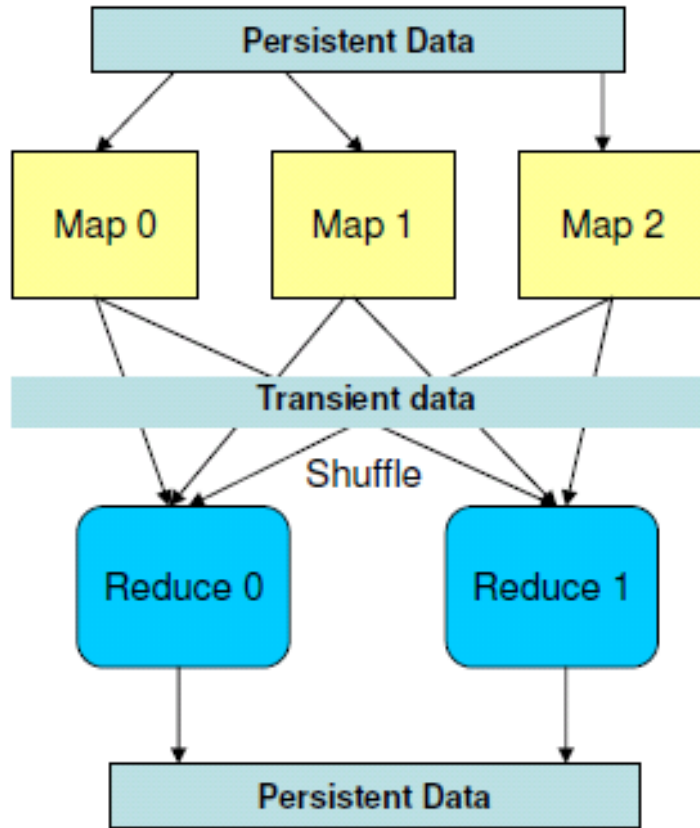
```
bin/hadoop dfs -cat java-output/part-00000
```

```
bin/hadoop dfs -copyToLocal java-output/part-00000 java-output-local
```

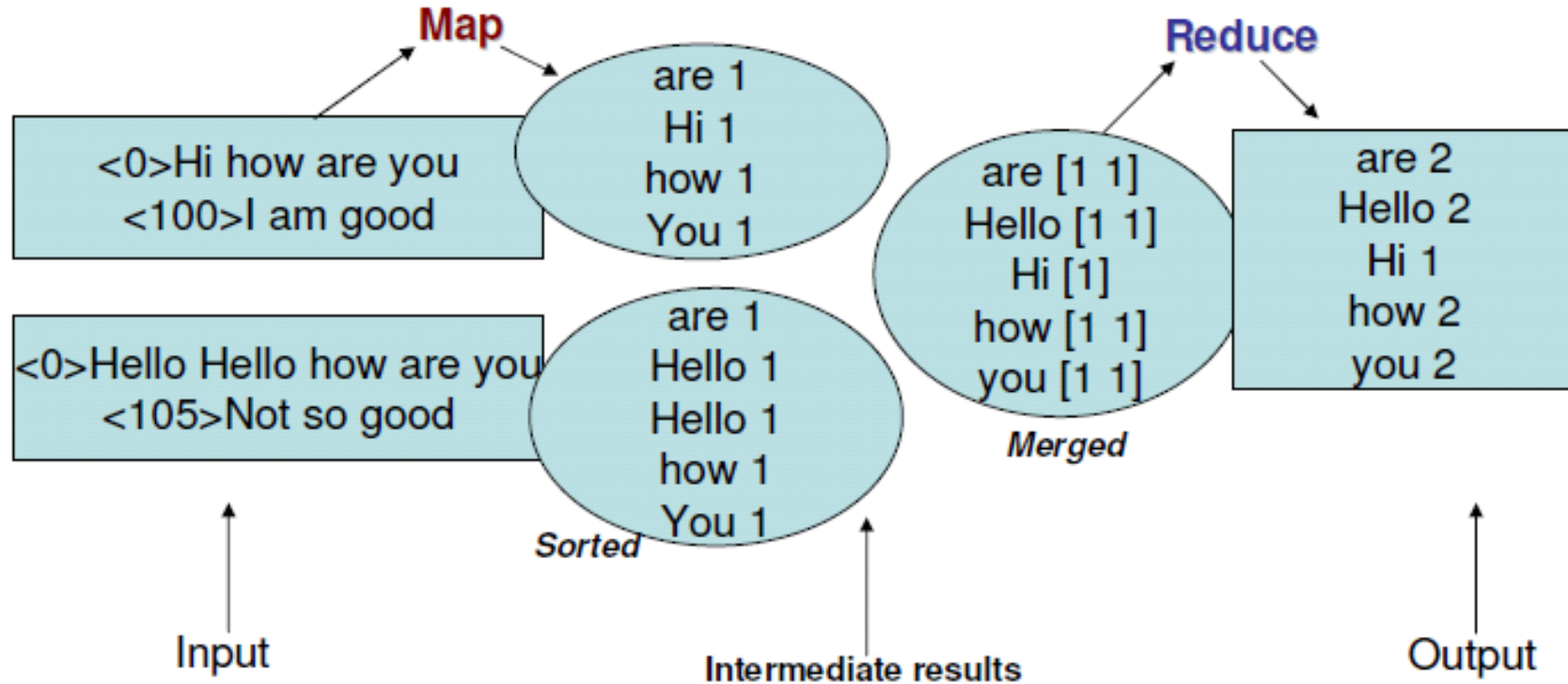
Hadoop(v2) MR job



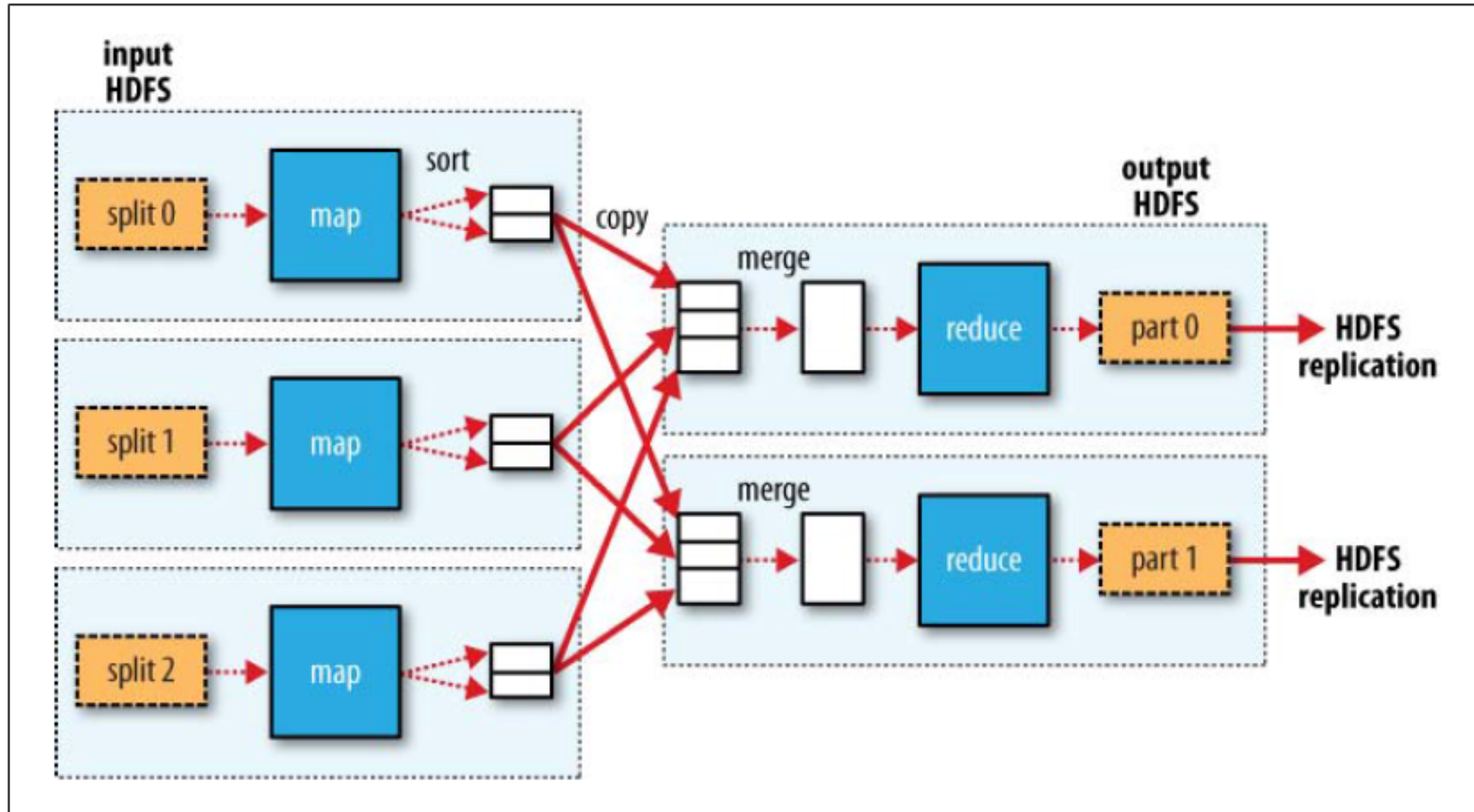
Map Reduce Data Flow



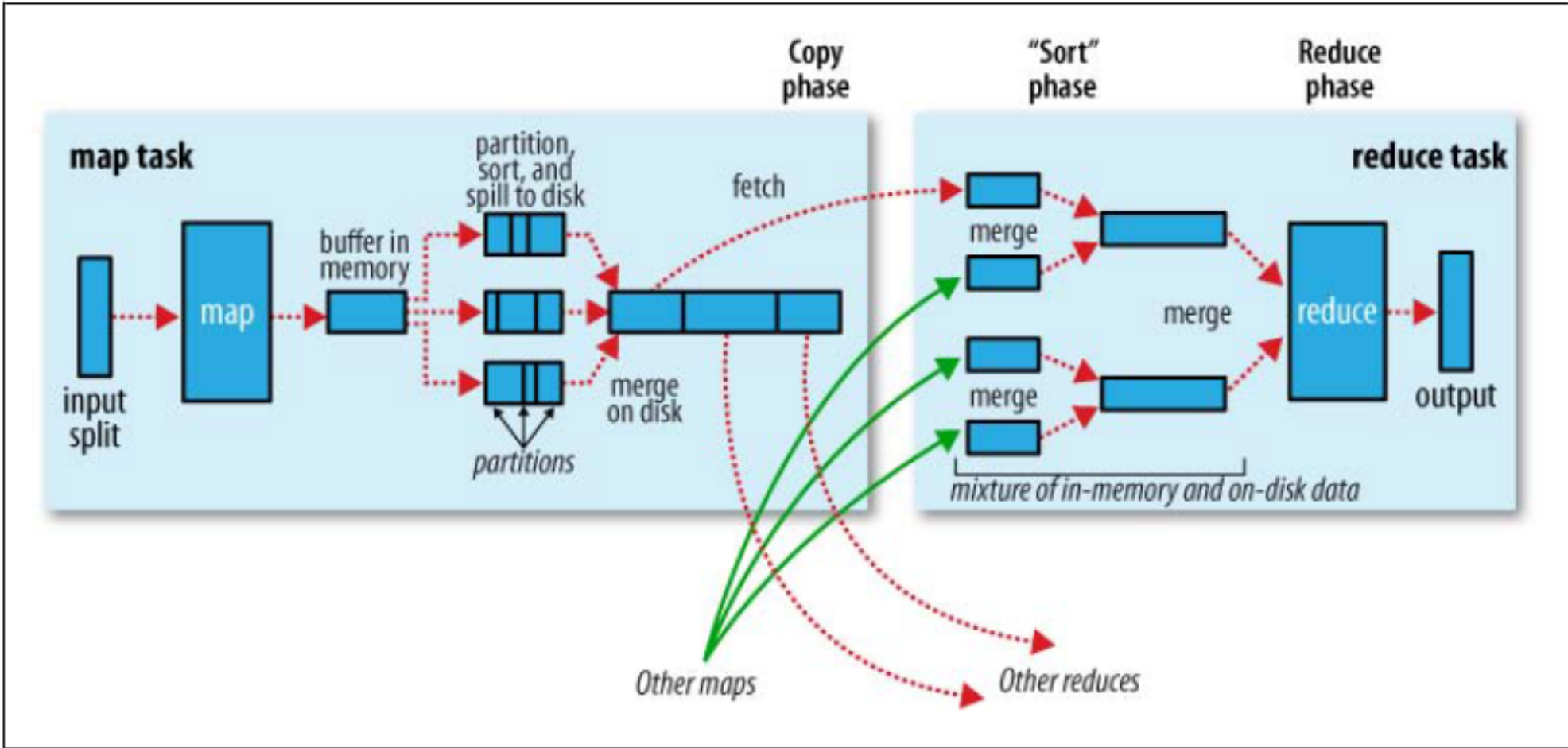
Data: Stream of keys and values



Hadoop MR Data Flow



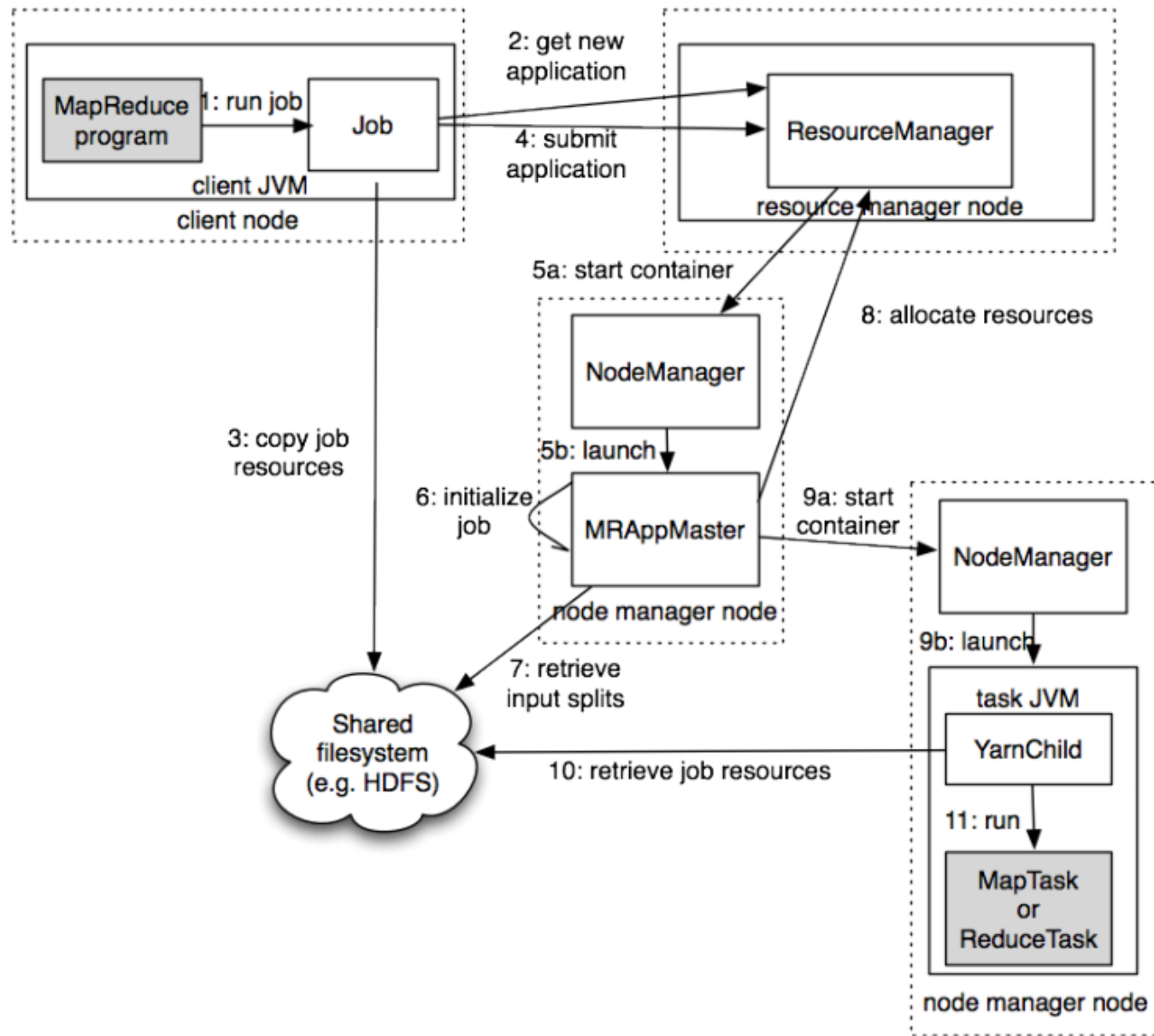
Shuffle and sort



Data Flow

- **Input and final output are stored on a distributed file system (FS):**
 - Scheduler tries to schedule map tasks “close” to physical storage location of input data
- **Intermediate results are stored on local FS of Map workers.**
- **Output of Reduce workers are stored on a distributed file system.**
- **Output is often input to another MapReduce task**

Hadoop(v2) MR job



Fault tolerance

- ❑ Comes from scalability and cost effectiveness

- ❑ HDFS:

 - ❑ Replication

- ❑ Map Reduce

 - ❑ Restarting failed tasks: map and reduce

 - ❑ Writing map output to FS

 - ❑ Minimizes re-computation

Coordination: Master

- **Master node takes care of coordination:**
 - **Task status:** (idle, in-progress, completed)
 - **Idle tasks** get scheduled as workers become available
 - When a map task completes, it sends the master the location and sizes of its R intermediate files, one for each reducer
 - Master pushes this info to reducers
- Master pings workers periodically to detect failures

Failures

Task failure

- Task has failed - report error to node manager, appmaster, client.
- Task not responsive, JVM failure - Node manager restarts tasks.

Application Master failure

- Application master sends heartbeats to resource manager.
- If not received, the resource manager retrieves job history of the run tasks.

Node manager failure

Dealing with Failures

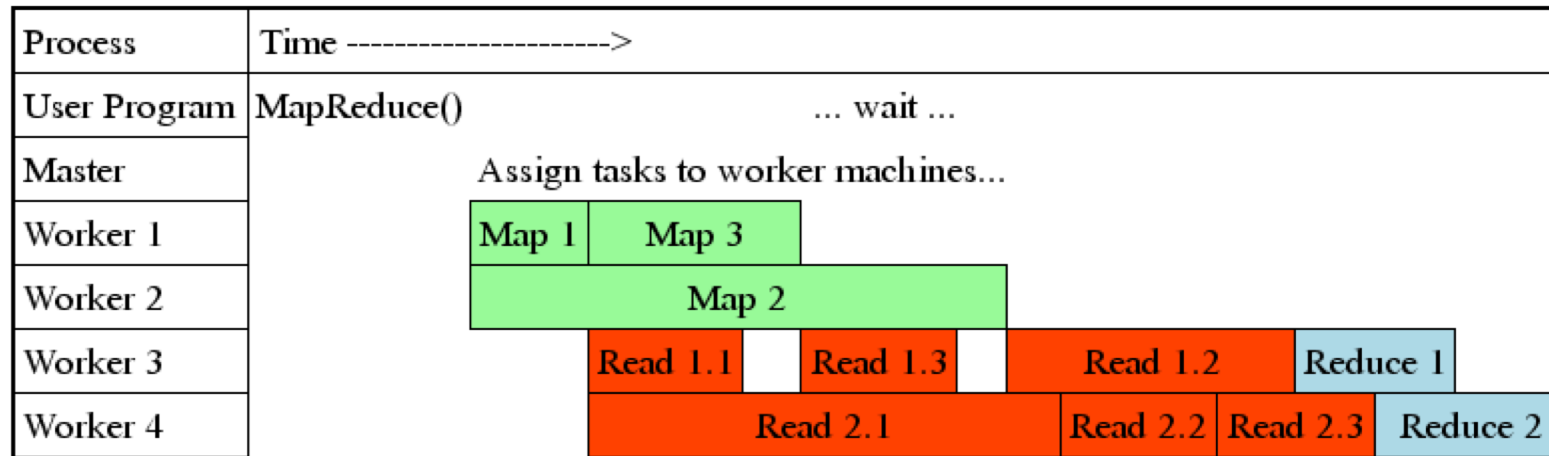
- **Map worker failure**
 - Map tasks completed or in-progress at worker are reset to idle
 - Reduce workers are notified when task is rescheduled on another worker
- **Reduce worker failure**
 - Only in-progress tasks are reset to idle
 - Reduce task is restarted
- **Master failure**
 - MapReduce task is aborted and client is notified

How many Map and Reduce jobs?

- M map tasks, R reduce tasks
- **Rule of a thumb:**
 - Make M much larger than the number of nodes in the cluster
 - One DFS chunk per map is common
 - Improves dynamic load balancing and speeds up recovery from worker failures
- **Usually R is smaller than M**
 - Because output is spread across R files

Task Granularity & Pipelining

- **Fine granularity tasks:** map tasks \gg machines
 - Minimizes time for fault recovery
 - Can do pipeline shuffling with map execution
 - Better dynamic load balancing



Refinements: Backup Tasks

- **Problem**

- Slow workers significantly lengthen the job completion time:
 - Other jobs on the machine
 - Bad disks
 - Weird things

- **Solution**

- Near end of phase, spawn backup copies of tasks
 - Whichever one finishes first “wins”

- **Effect**

- Dramatically shortens job completion time

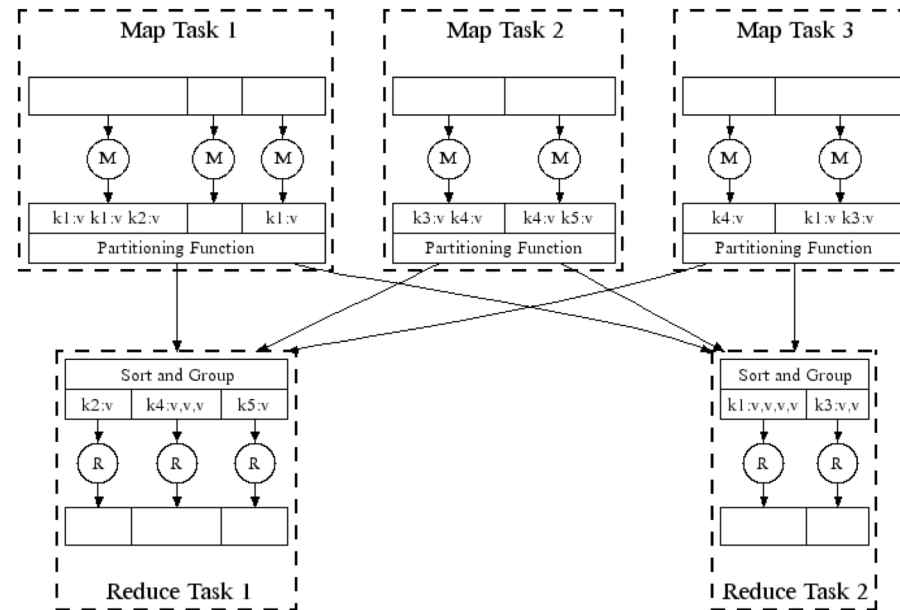
Refinement: Combiners

- Often a Map task will produce many pairs of the form $(k, v_1), (k, v_2), \dots$ for the same key k
 - E.g., popular words in the word count example

- **Can save network time by pre-aggregating values in the mapper:**

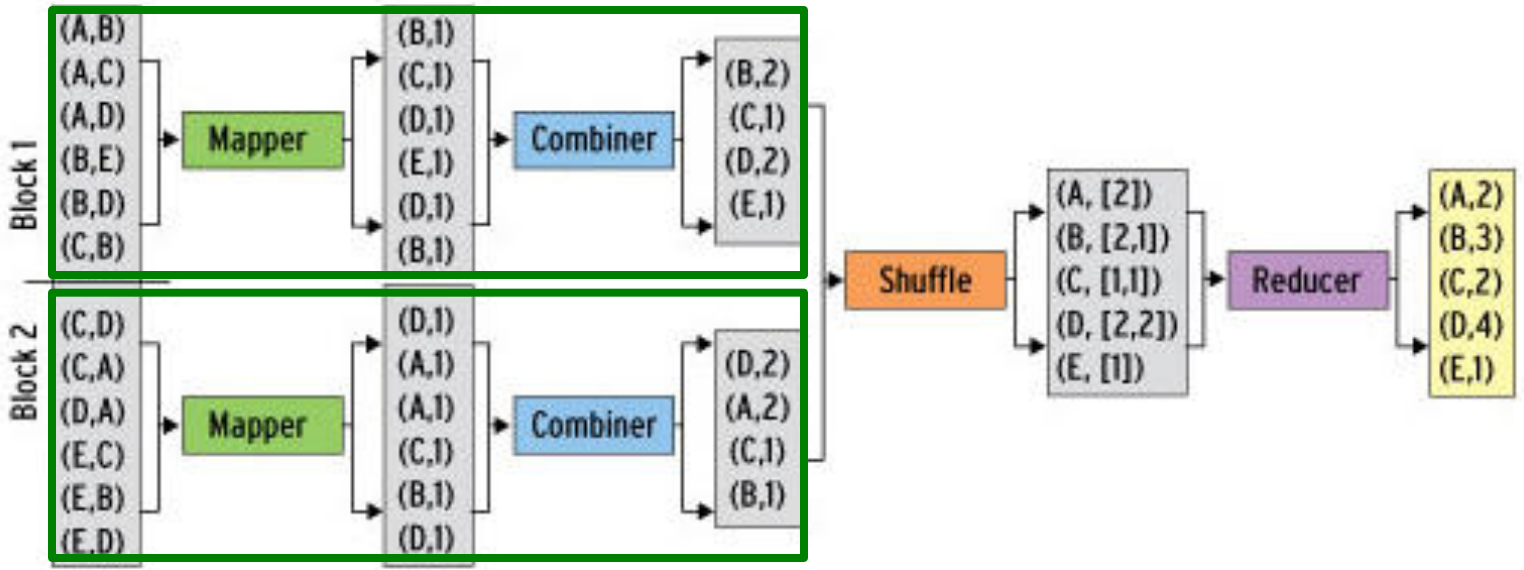
- $\text{combine}(k, \text{list}(v_1)) \rightarrow v_2$
- Combiner is usually same as the reduce function

- Works only if reduce function is commutative and associative



Refinement: Combiners

- Back to our word counting example:
 - Combiner combines the values of all keys of a single mapper (single machine):



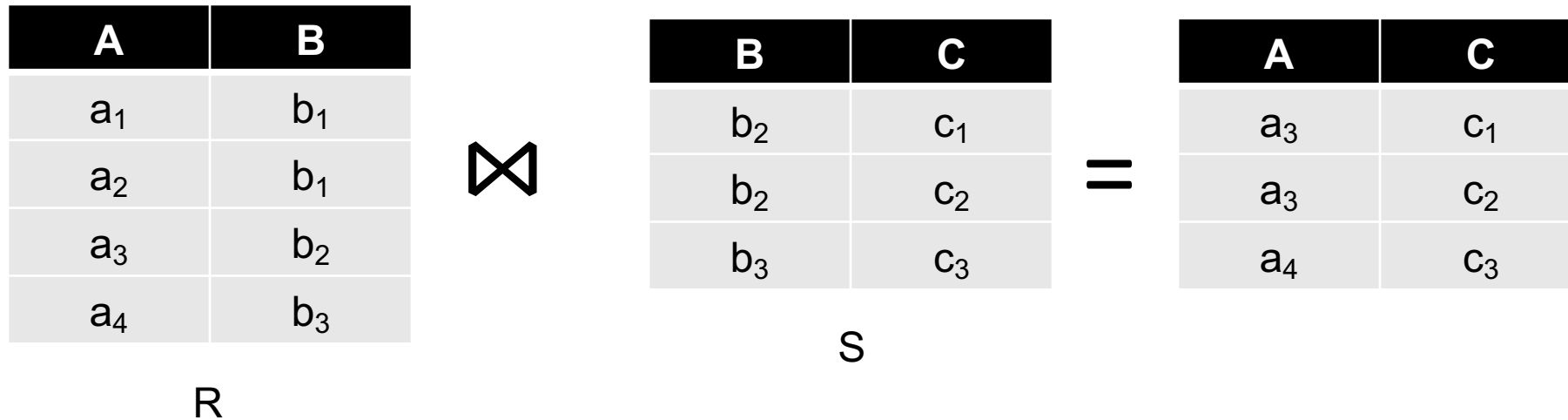
– Much less data needs to be copied and shuffled!

Refinement: Partition Function

- **Want to control how keys get partitioned**
 - Inputs to map tasks are created by contiguous splits of input file
 - Reduce needs to ensure that records with the same intermediate key end up at the same worker
- **System uses a default partition function:**
 - **$\text{hash}(\text{key}) \bmod R$**
- **Sometimes useful to override the hash function:**
 - E.g., $\text{hash}(\text{hostname}(\text{URL})) \bmod R$ ensures URLs from a host end up in the same output file

Example: Join By Map-Reduce

- Compute the natural join $R(A,B) \bowtie S(B,C)$
- R and S are each stored in files
- Tuples are pairs (a,b) or (b,c)



Map-Reduce Join

- Use a hash function h from B-values to $1\dots k$
- **A Map process turns:**
 - Each input tuple $R(a,b)$ into key-value pair $(b,(a,R))$
 - Each input tuple $S(b,c)$ into $(b,(c,S))$
- **Map processes** send each key-value pair with key b to Reduce process $h(b)$
 - Hadoop does this automatically; just tell it what k is.
- Each **Reduce process** matches all the pairs $(b,(a,R))$ with all $(b,(c,S))$ and outputs (a,b,c) .

Cost Measures for Algorithms

- **In MapReduce we quantify the cost of an algorithm using**
 1. *Communication cost* = total I/O of all processes
 2. *Elapsed communication cost* = max of I/O along any path
 3. *(Elapsed) computation cost* analogous, but count only running time of processes

Note that here the big-O notation is not the most useful
(adding more machines is always an option)

Example: Cost Measures

- **For a map-reduce algorithm:**
 - **Communication cost** = input file size + $2 \times$ (sum of the sizes of all files passed from Map processes to Reduce processes) + the sum of the output sizes of the Reduce processes.
 - **Elapsed communication cost** is the sum of the largest input + output for any map process, plus the same for any reduce process

What Cost Measures Mean

- Either the I/O (communication) or processing (computation) cost dominates
 - Ignore one or the other
- Total cost tells what you pay in rent from your friendly neighborhood cloud
- Elapsed cost is wall-clock time using parallelism

Cost of Map-Reduce Join

- **Total communication cost**
 $= O(|R| + |S| + |R \bowtie S|)$
- **Elapsed communication cost** $= O(s)$
 - We're going to pick k and the number of Map processes so that the I/O limit s is respected
 - We put a limit s on the amount of input or output that any one process can have. **s could be:**
 - What fits in main memory
 - What fits on local disk
- With proper indexes, computation cost is linear in the input + output size
 - So computation cost is like comm. cost

Conclusion:

- We have seen:
 - Map-reduce programming in Java
 - Map reduce programming in other languages
 - Implementation details:
 - Job and tasks
 - Shuffle and sort

References:

- Jure Leskovec, Anand Rajaraman, Jeff Ullman. **Mining of Massive Datasets. 2nd edition.** - *Cambridge University Press.* <http://www.mmds.org/>
- Tom White. **Hadoop: The definitive Guide.** O'Reilly Press.