Introduction to Data Management CSE 344

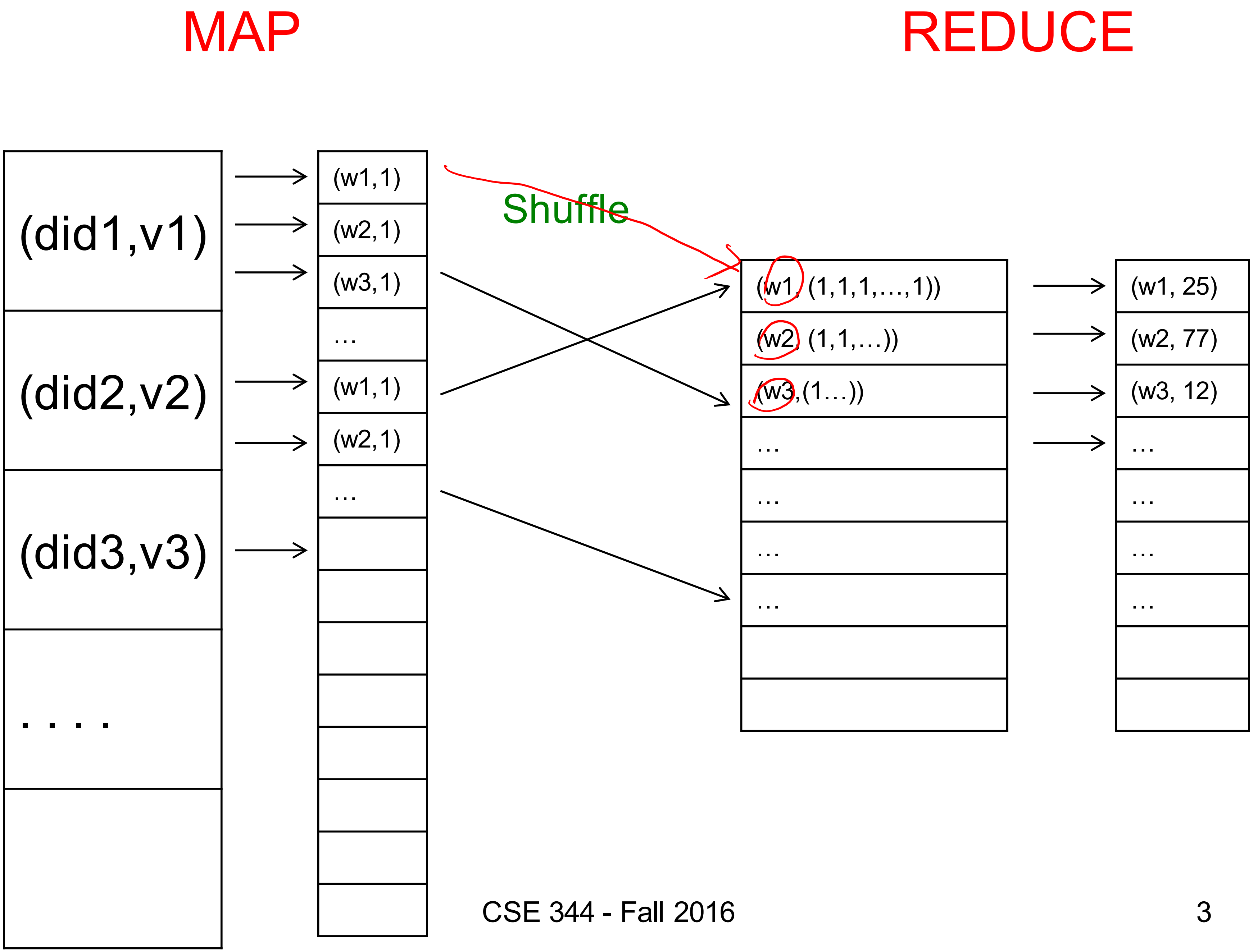
Section 9: MapReduce

Example

- Counting the number of occurrences of each word in a large collection of documents
- Each Document
  - The key = document id (did)
  - The value = set of words (word)

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
            EmitIntermediate(w, "1");
```

```
reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
            result += ParseInt(v);
    Emit(AsString(result));
```


MAP
REDUCE
(did1,v1)
(did2,v2)
(did3,v3)
. . . .
(w1,1)
(w2,1)
(w3,1)
…
(w1,1)
(w2,1)
…
Shuffle
(w1, (1,1,1,…,1))
(w2, (1,1,…))
(w3,(1…))
…
…
…
…
(w1, 25)
(w2, 77)
(w3, 12)
…
…
…
…

Jobs v.s. Tasks

- A MapReduce Job
  - One single “query,” e.g., count the words in all docs
  - More complex queries may consists of multiple jobs
- A Map Task, or a Reduce Task
  - A group of instantiations of the map-, or reduce-function, which are scheduled on a single worker

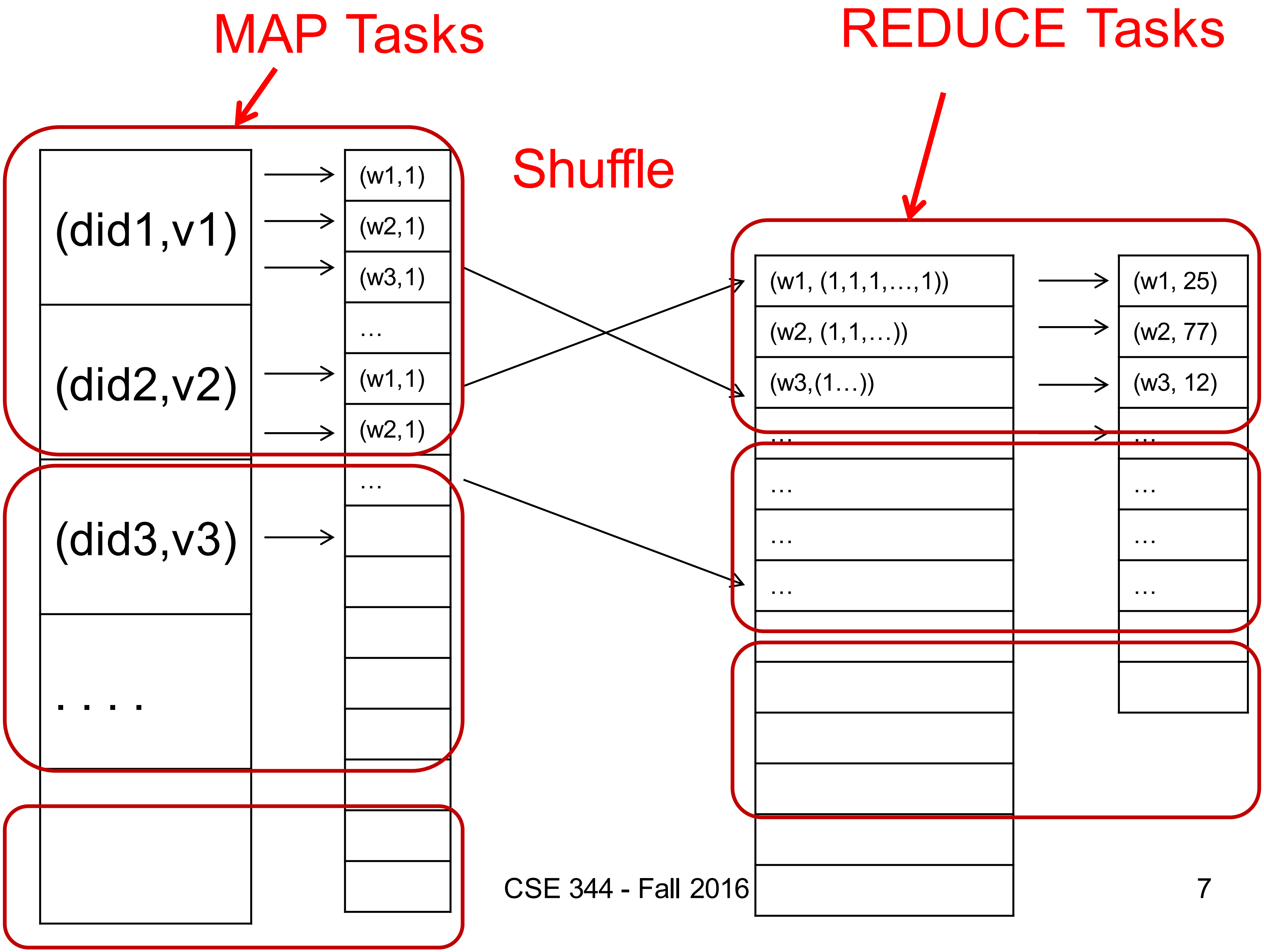
Workers

- A worker is a process that executes one task at a time
- Typically there is one worker per processor, hence 4 or 8 per node

Fault Tolerance

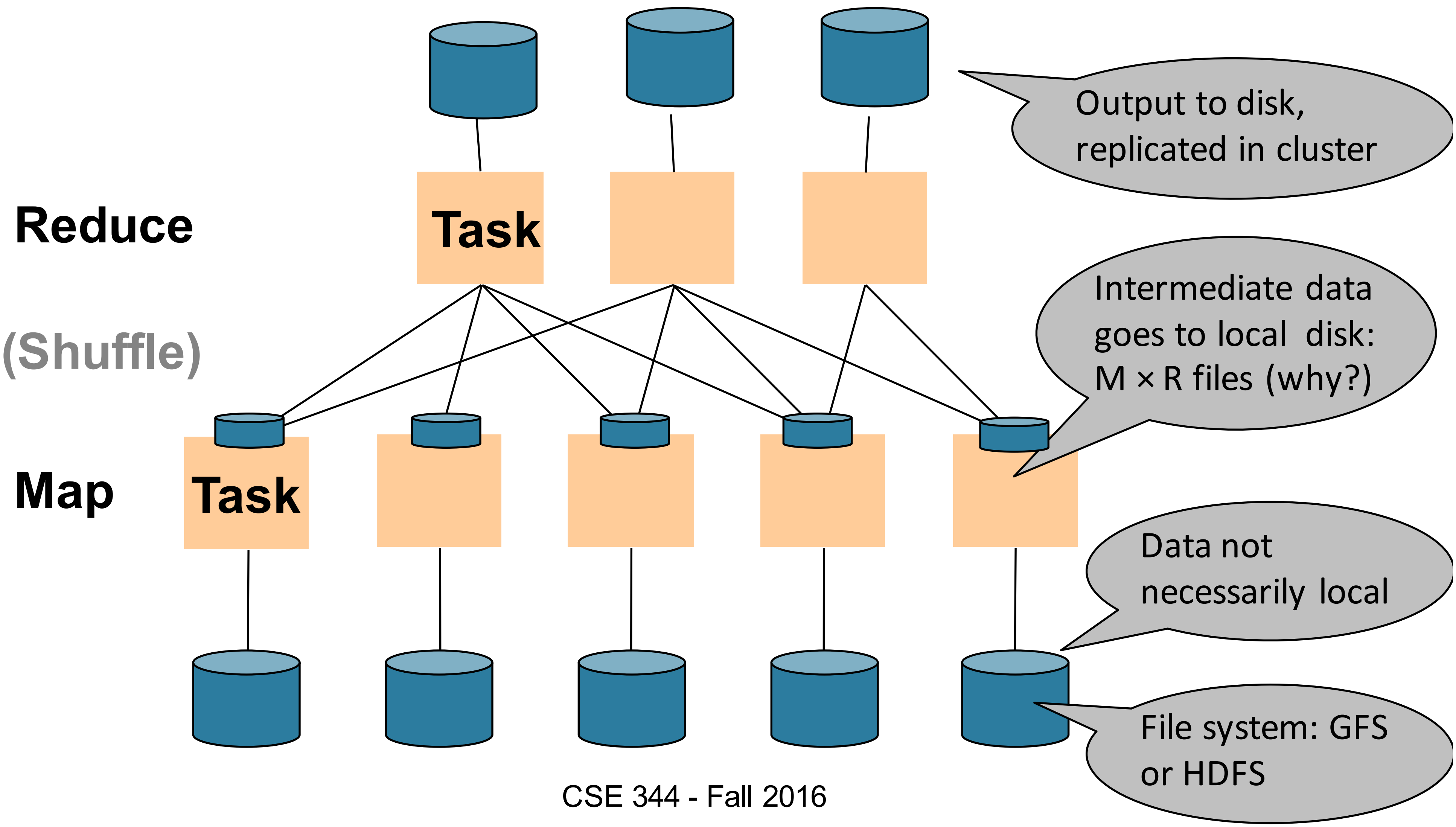
- If one server fails once every year…
  ... then a job with 10,000 servers will fail in less than one hour

- MapReduce handles fault tolerance by writing intermediate files to disk:
  – Mappers write file to local disk
  – Reducers read the files (=reshuffling); if the server fails, the reduce task is restarted on another server

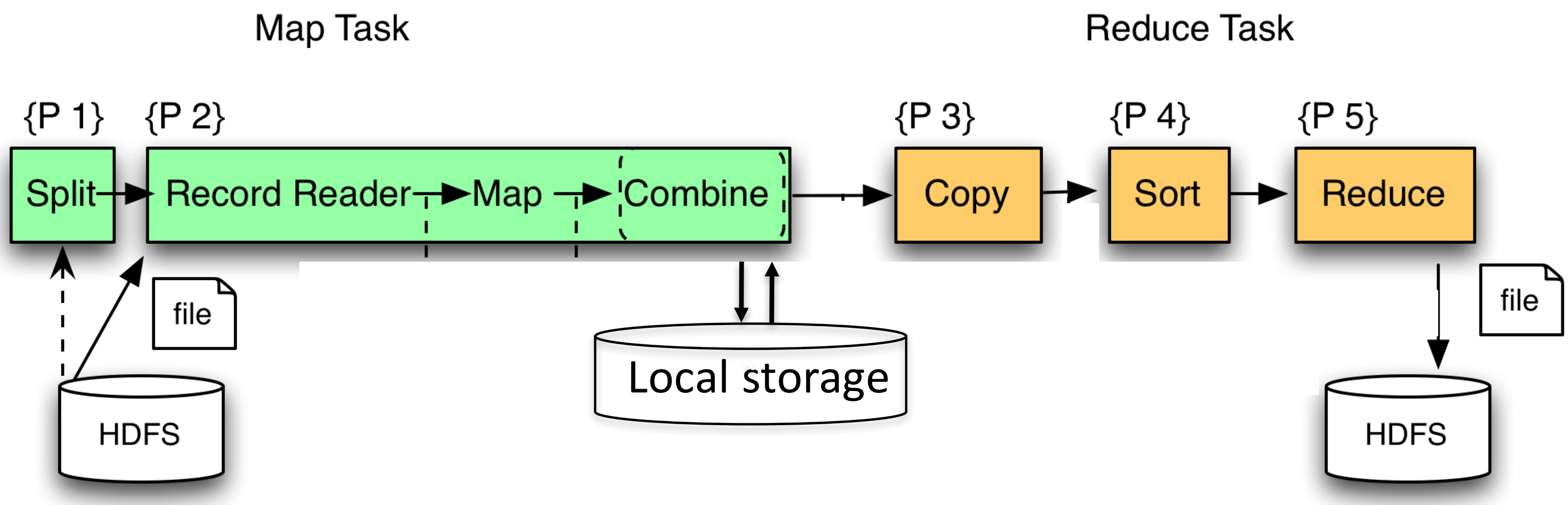


MAP Tasks
REDUCE Tasks
Shuffle
(did1,v1)
(did2,v2)
(did3,v3)
. . . . .
(w1,1)
(w2,1)
(w3,1)
…
(w1,1)
(w2,1)
…
(w1, (1,1,1,…,1))
(w2, (1,1,…))
(w3,(1…))
…
…
…
…
(w1, 25)
(w2, 77)
(w3, 12)
…
…
…
…

MapReduce Execution Details



MapReduce Phases



Top K Sort Example

- Finding the Top K most frequent words
- Each Document
  - The key = document id (did)
  - The value = set of words (word)

Discuss with each other what you think may be in the Map and Reduce phase.

Top K Sort Map Phase

```
map(String key, String value):
    // key: document name
    // value: document contents
    hashTbl = new Hashtable()
    for each word w in value:
        Integer sum = 1 + (hashTbl.find(w) == null ? 0 : hashTbl.find(w))
        hashTbl.insert(w, sum)
    String res = “”
    for each k, v in hashTbl :
        res = res + k + ”:” + v + “ “
EmitIntermediate(“dummy”, res.trim());
```

Top K Sort Reduce Phase

```
reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    hashTbl = new Hashtable()
    for each v in values:
        String[] pairs = v.split(" ")
        for each pair in pairs:
            String[] kv = pair.split(":")
            Integer sum = parseInt(kv[1]) +
                (hashTbl.find(kv[0]) == null ? 0 : hashTbl.find(kv[0]))
            hashTbl.insert(kv[0], sum)
    treeMap = sortByValue(hashTbl)
    for each k, v in treeMap:
        count = count + 1
        Emit(k, v);
        if count equals 20: break
```

Implementation of Map Phase

```
public static class TopNMapper extends Mapper<Object, Text, Text, IntWritable> {
    private Map<String, Integer> countMap = new HashMap<>();
    @Override
    public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
        String cleanLine = value.toString().toLowerCase().replaceAll("[_|$#<>\\^=\\[\\]\\*/\\\\,;,.\\-:()?!\"']", " ");
        StringTokenizer itr = new StringTokenizer(cleanLine);
        while (itr.hasMoreTokens()) {
            String word = itr.nextToken().trim();
            if (countMap.containsKey(word)) {
                countMap.put(word, countMap.get(word)+1);
            }
            else {
                countMap.put(word, 1);
            }
        }
    }
    @Override
    protected void cleanup(Context context) throws IOException, InterruptedException {
        for (String key: countMap.keySet()) {
            context.write(new Text(key), new IntWritable(countMap.get(key)));
        }
    }
}
```

source: https://github.com/andreaiacono/

Implementation of Reduce Phase

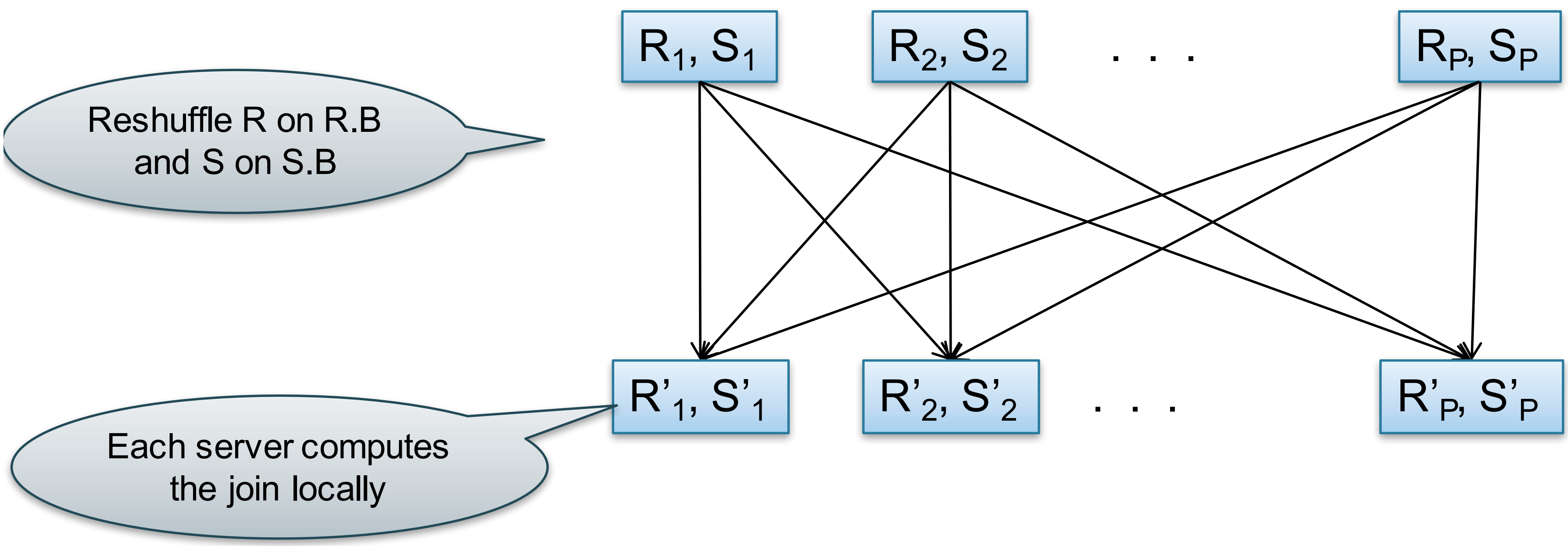
```
public static class TopNReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    private Map<Text, IntWritable> countMap = new HashMap<>();
    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        countMap.put(new Text(key), new IntWritable(sum));
    }
    @Override
    protected void cleanup(Context context) throws IOException, InterruptedException {
        Map<Text, IntWritable> sortedMap = MiscUtils.sortByValues(countMap);
        int counter = 0;
        for (Text key: sortedMap.keySet()) {
            if (counter ++ == 20) {
                break;
            }
            context.write(key, sortedMap.get(key));
        }
    }
}
```

source: https://github.com/andreaiacono/

$R(A,B) \bowtie_{B=C} S(C,D)$

Partitioned Hash-Join

Initially, both R and S are horizontally partitioned



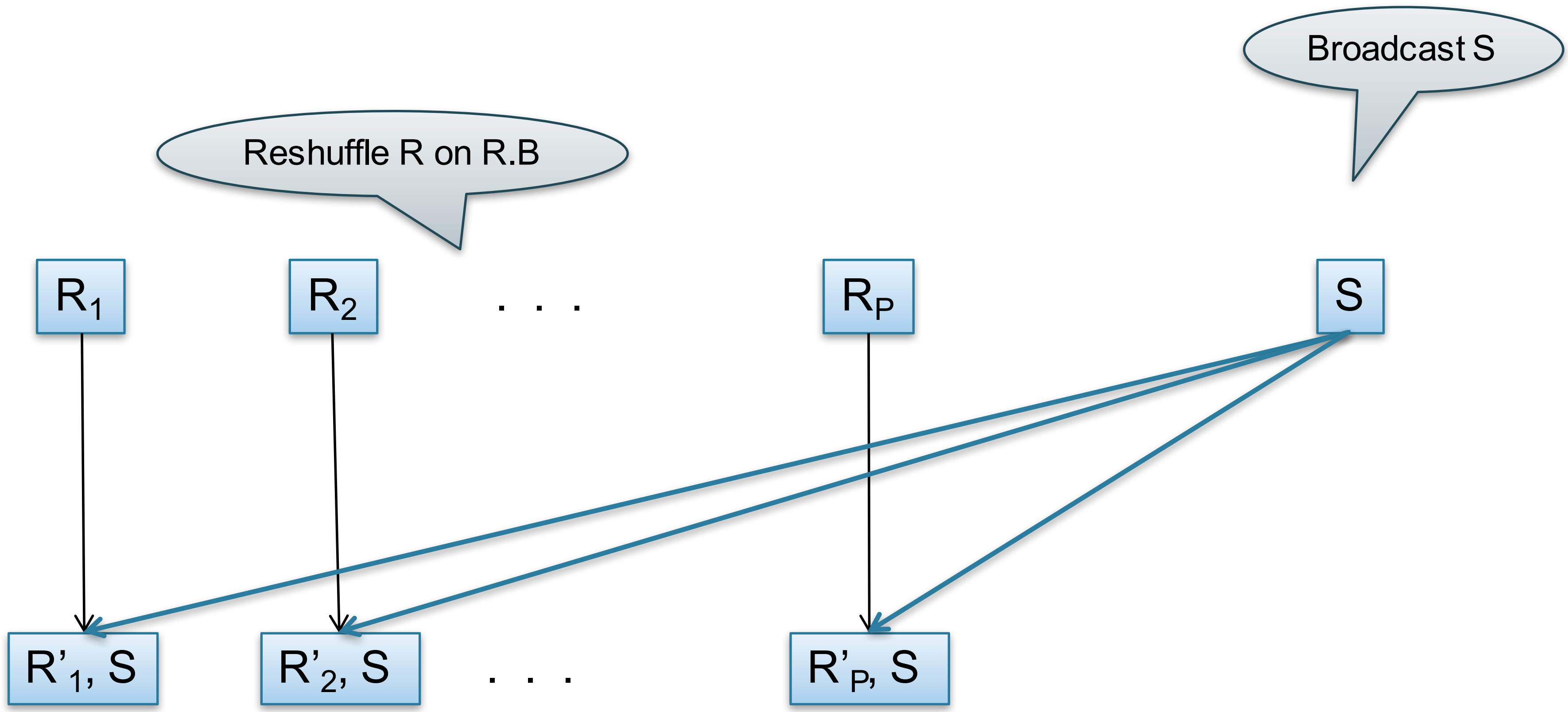
Implementing Join in MR

Two simple parallel join algorithms:

- Partitioned hash-join (we saw it, will recap)
- Broadcast join

$R(A,B) \bowtie_{B=C} S(C,D)$

Broadcast Join



Implementation

- There is one master node
- Master partitions input file into *M splits*, by key
- Master assigns *workers* (=servers) to the *M map tasks*, keeps track of their progress
- Workers write their output to local disk, partition into *R regions*
- Master assigns workers to the *R reduce tasks*
- Reduce workers read regions from the map workers' local disks

Interesting Implementation Details

Worker failure:

- Master pings workers periodically,
- If down then reassigns the task to another worker

Interesting Implementation Details

Backup tasks:

- *Straggler* = a machine that takes unusually long time to complete one of the last tasks. Eg:
  - Bad disk forces frequent correctable errors (30MB/s → 1MB/s)
  - The cluster scheduler has scheduled other tasks on that machine
- Stragglers are a main reason for slowdown
- Solution*: pre-emptive backup execution of the last few remaining in-progress tasks*

Parallel Data Processing @ 2010

Issues with MapReduce

- Difficult to write more complex queries
- Need multiple MapReduce jobs: dramatically slows down because it writes all results to disk
- We will talk about Spark in the next lecture