# CSE 344

## FEBRUARY 26TH – MAP/REDUCE

# ADMINISTRIVIA

- **HW6 Due next Wednesday (Feb 28)**
- **HW4 Grades Out**
- **HW8 Out tomorrow**
  - Due Mar 9th
- **HW7 Out Wednesday**
  - Need to wait for E/R Diagrams
  - Due Mar 7th

# ADMINISTRIVIA

- **OQ6 Out Wednesday**
  - Only one left after this one
- **Course Evaluations**
  - Out Saturday

# TODAY'S LECTURE

- **Map/Reduce**
  - Applications
  - Motivation

# MOTIVATION

**We learned how to parallelize relational database systems**

**While useful, it might incur too much overhead if our query plans consist of simple operations**

**MapReduce is a programming model for such computation**

**First, let's study how data is stored in such systems**

# DISTRIBUTED FILE SYSTEM (DFS)

**For very large files: TBs, PBs**

**Each file is partitioned into *chunks*, typically 64MB**

**Each chunk is replicated several times (≥3), on different racks, for fault tolerance**

**Implementations:**

- Google's DFS:  GFS, proprietary
- Hadoop's DFS:  HDFS, open source

# MAPREDUCE

Google: paper published 2004

Free variant: Hadoop

MapReduce = high-level programming model and implementation for large-scale parallel data processing

# TYPICAL PROBLEMS SOLVED BY MR

**Read a lot of data**

**Map: extract something you care about from each record**

**Shuffle and Sort**

**Reduce: aggregate, summarize, filter, transform**

**Write the results**

Paradigm stays the same,
change map and reduce functions for different problems

# DATA MODEL

**Files!**

**A file = a bag of `(key, value)` pairs**

**A MapReduce program:**

**Input: a bag of `(inputkey, value)` pairs**

**Output: a bag of `(outputkey, value)` pairs**

# STEP 1: THE MAP PHASE

User provides the MAP-function:

Input: `(input key, value)`

Ouput: bag of `(intermediate key, value)`

System applies the map function in parallel to all `(input key, value)` pairs in the input file

# STEP 2: THE REDUCE PHASE

**User provides the REDUCE function:**

**Input:** `(intermediate key, bag of values)`

**Output: bag of output** `(values)`

**System groups all pairs with the same intermediate key, and passes the bag of values to the REDUCE function**

# EXAMPLE

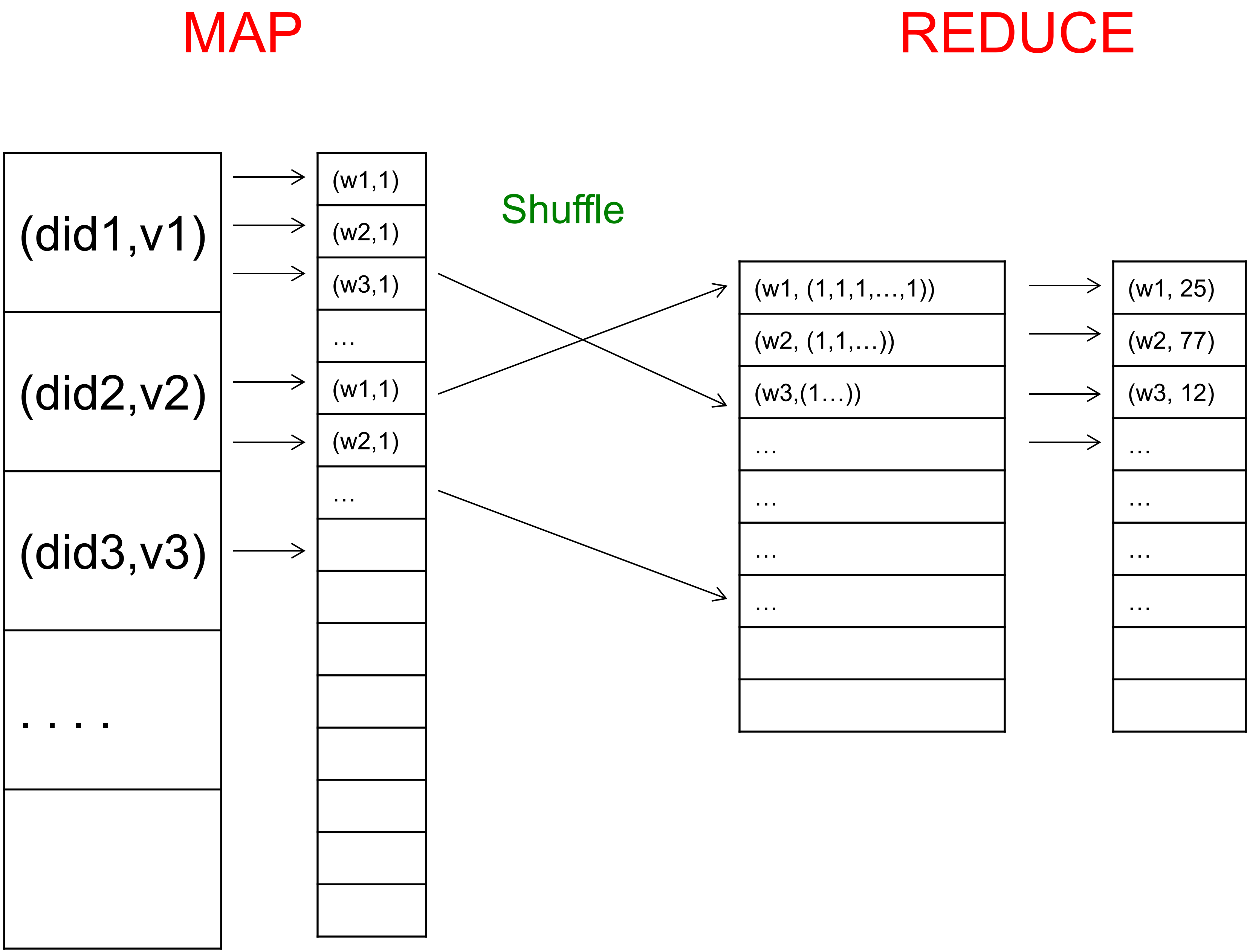
**Counting the number of occurrences of each word in a large collection of documents**

**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
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)
…
…
…
…

# JOBS VS 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

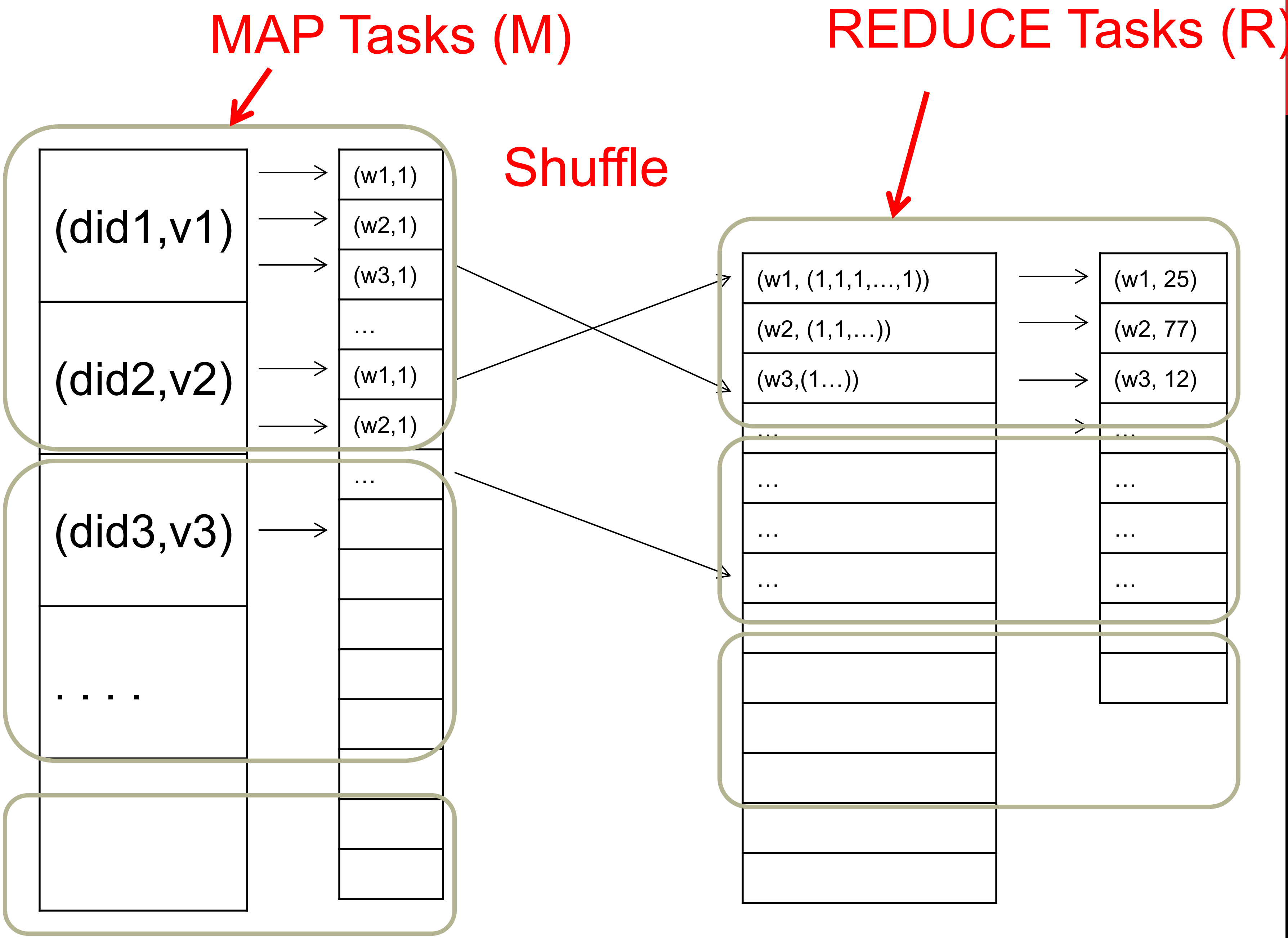
Typically there is one worker per processor, hence 4 or 8 per node

# FAULT TOLERANCE

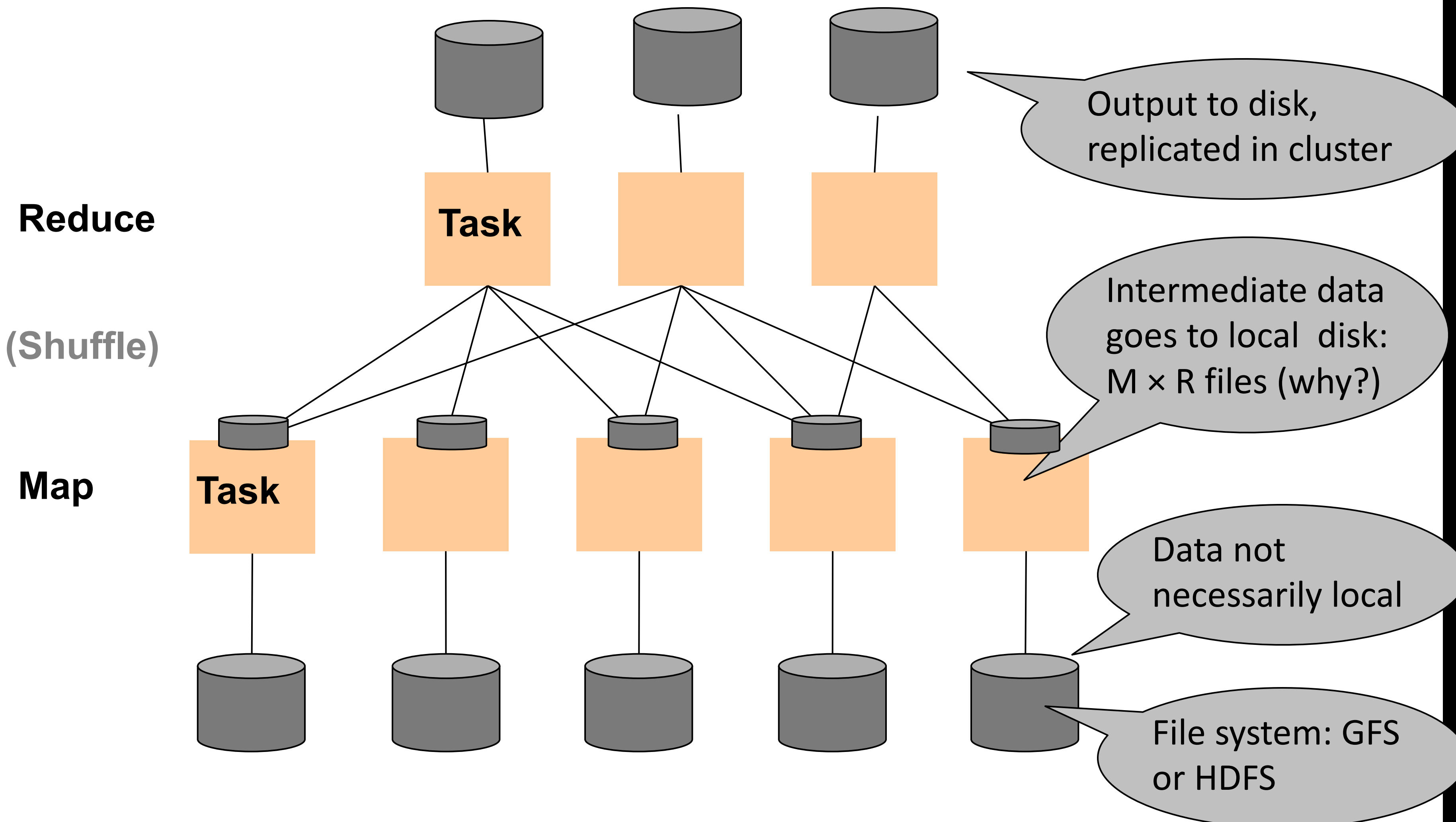
**If one server fails once every year…**
**... then a job with 10,000 servers will fail in less than one hour**

**MapReduce handles fault tolerance by writing intermediate files to disk:**

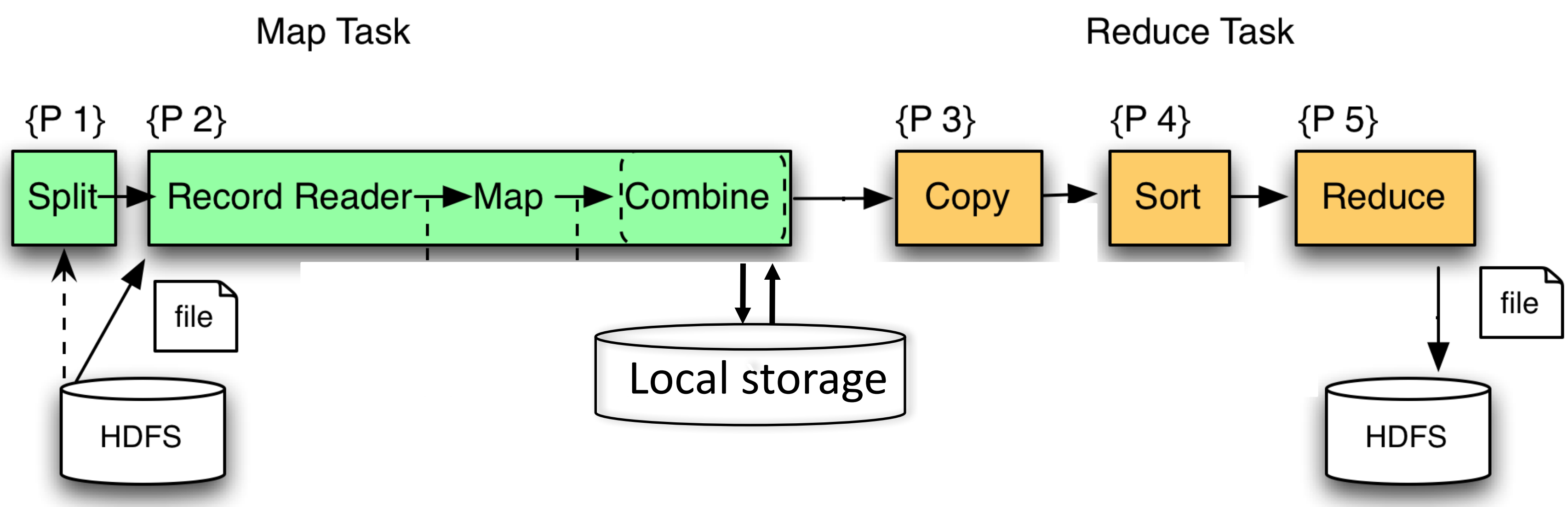
- Mappers write file to local disk
- Reducers read the files (=reshuffling); if the server fails, the reduce task is restarted on another server


MAP Tasks (M)
REDUCE Tasks (R)
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



# 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:**

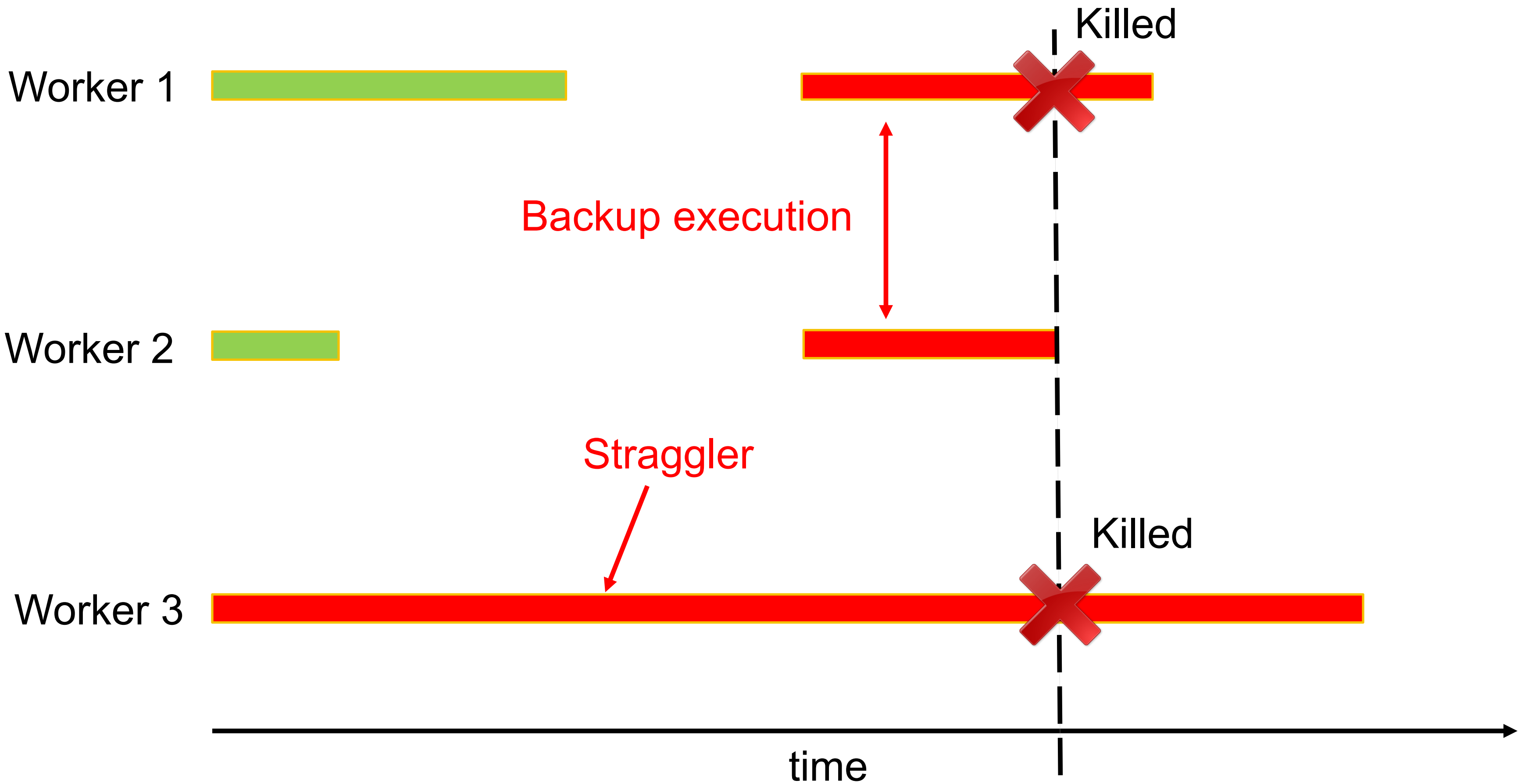
***Straggler*** **= a machine that takes unusually long time to complete one of the last tasks. E.g.:**

- 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***

# STRAGGLER EXAMPLE



# USING MAPREDUCE IN PRACTICE:

# IMPLEMENTING RA OPERATORS IN MR

# RELATIONAL OPERATORS IN MAPREDUCE

Given relations R(A,B) and S(B, C) compute:

**Selection**: $\sigma_{A=123}(R)$

**Group-by**: $\gamma_{A,sum(B)}(R)$

**Join**: $R \bowtie S$

# SELECTION $\Sigma_{A=123}(R)$

```
map(String value):
    if  value.A = 123:
      EmitIntermediate(value.key, value);
```

```
reduce(String k, Iterator values):
    for each v in values:
      Emit(v);
```

# SELECTION $\Sigma_{A=123}(R)$

```
map(String value):
    if  value.A = 123:
      EmitIntermediate(value.key, value);
```

```
reduce(String k, Iterator values):
    for each v in values:
      Emit(v);
```

No need for reduce.
But need system hacking in Hadoop
to remove reduce from MapReduce

# GROUP BY $\Gamma_{A,SUM(B)}(R)$

```
map(String value):
    EmitIntermediate(value.A, value.B);
```

```
reduce(String k, Iterator values):
    s = 0
    for each v in values:
      s = s + v
    Emit(k, v);
```

# JOIN

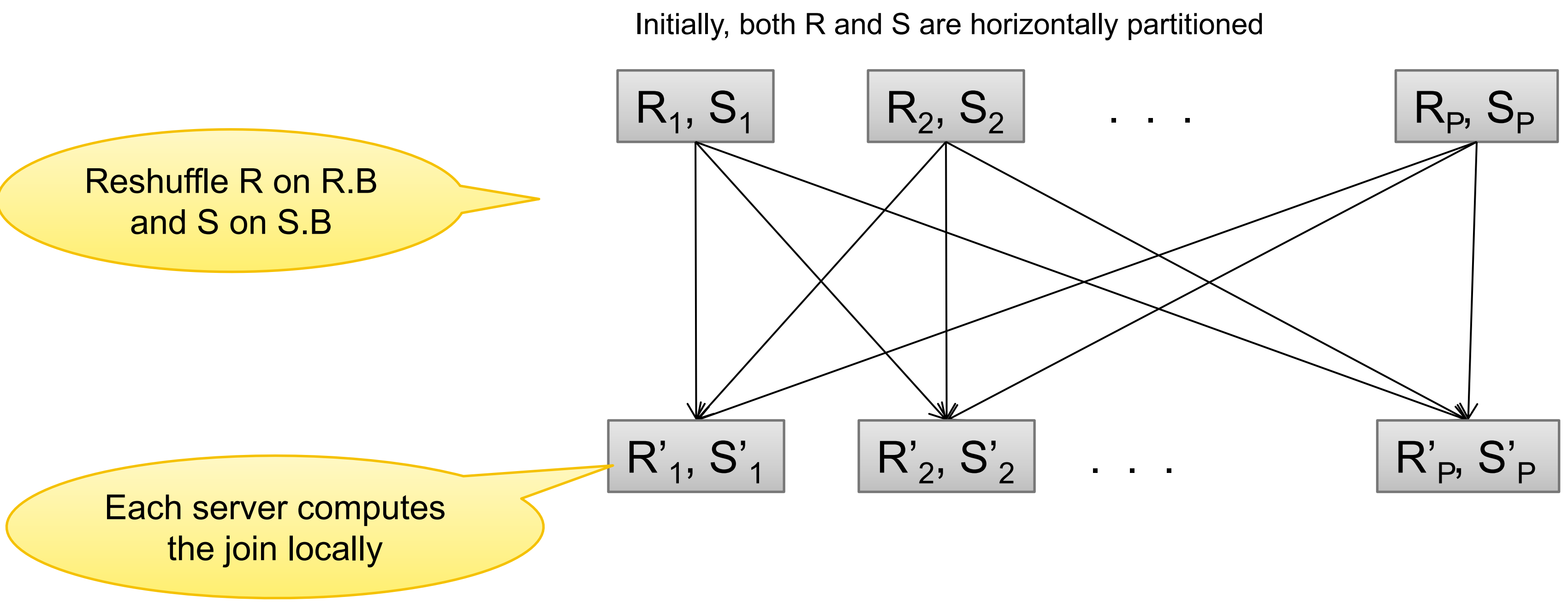
**Two simple parallel join algorithms:**

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

**Broadcast join**

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

# PARTITIONED HASH-JOIN



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

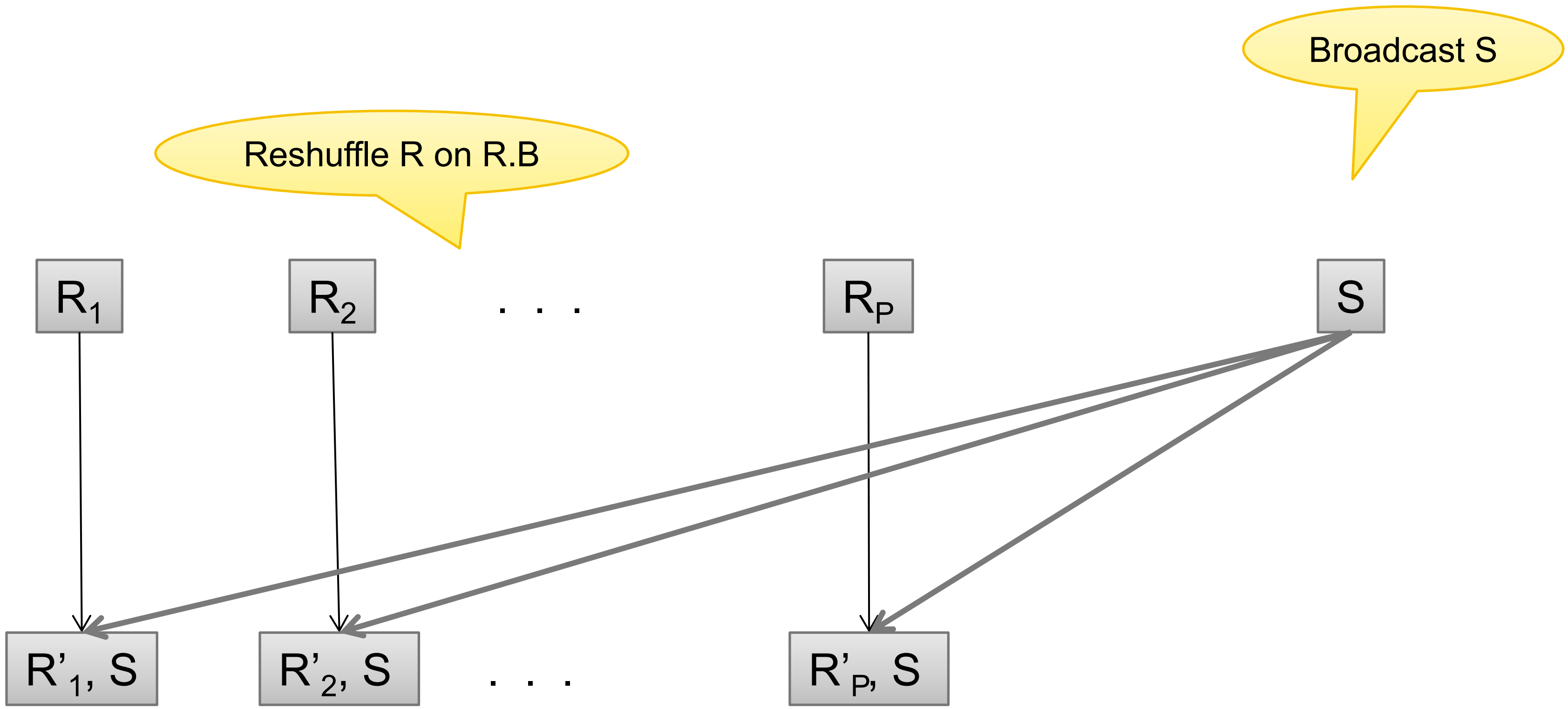
# PARTITIONED HASH-JOIN

```
map(String value):
    case value.relationName of
     'R': EmitIntermediate(value.B, ('R', value));
     'S': EmitIntermediate(value.C, ('S', value));
```

```
reduce(String k, Iterator values):
    R = empty;  S = empty;
    for each v in values:
     case v.type of:
        'R':   R.insert(v)
               'S':   S.insert(v);
    for v1 in R, for v2 in S
     Emit(v1,v2);
```

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

# BROADCAST JOIN



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

# BROADCAST JOIN

map should read several records of R: value = some group of records

```
map(String value):
    open(S); /* over the network */
    hashTbl = new()
    for each w in S:
      hashTbl.insert(w.C, w)
    close(S);

    for each v in value:
      for each w in hashTbl.find(v.B)
          Emit(v,w);
```

Read entire table S, build a Hash Table

```
reduce(…):
    /* empty: map-side only */
```

# CONCLUSIONS

MapReduce offers a simple abstraction, and handles distribution + fault tolerance

Speedup/scaleup achieved by allocating dynamically map tasks and reduce tasks to available server.  However, skew is possible (e.g., one huge reduce task)

Writing intermediate results to disk is necessary for fault tolerance, but very slow.

Spark replaces this with “Resilient Distributed Datasets” = main memory + lineage