MapReduce

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

slide source: Jeff Dean

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

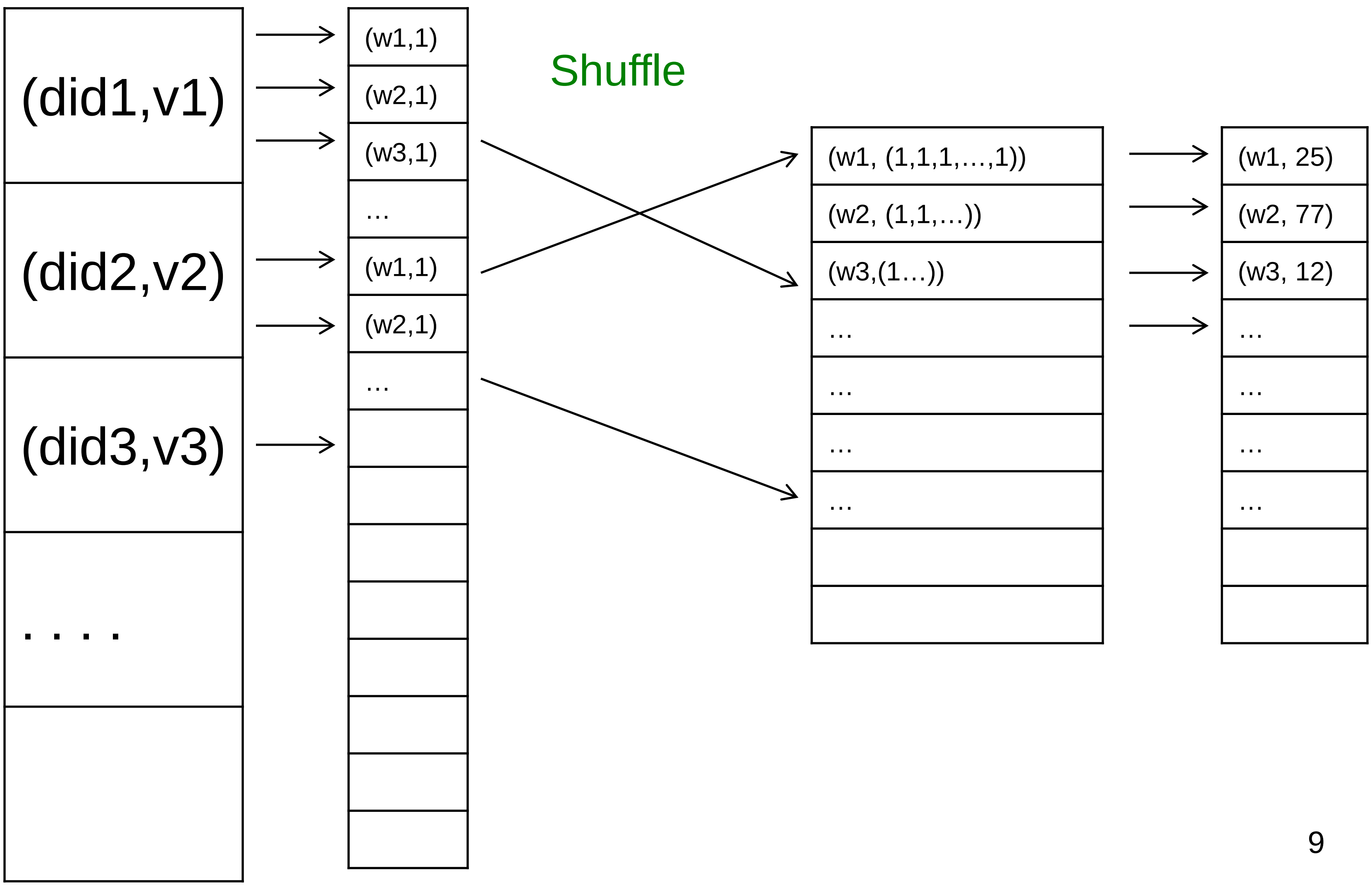
Example

- 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



(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

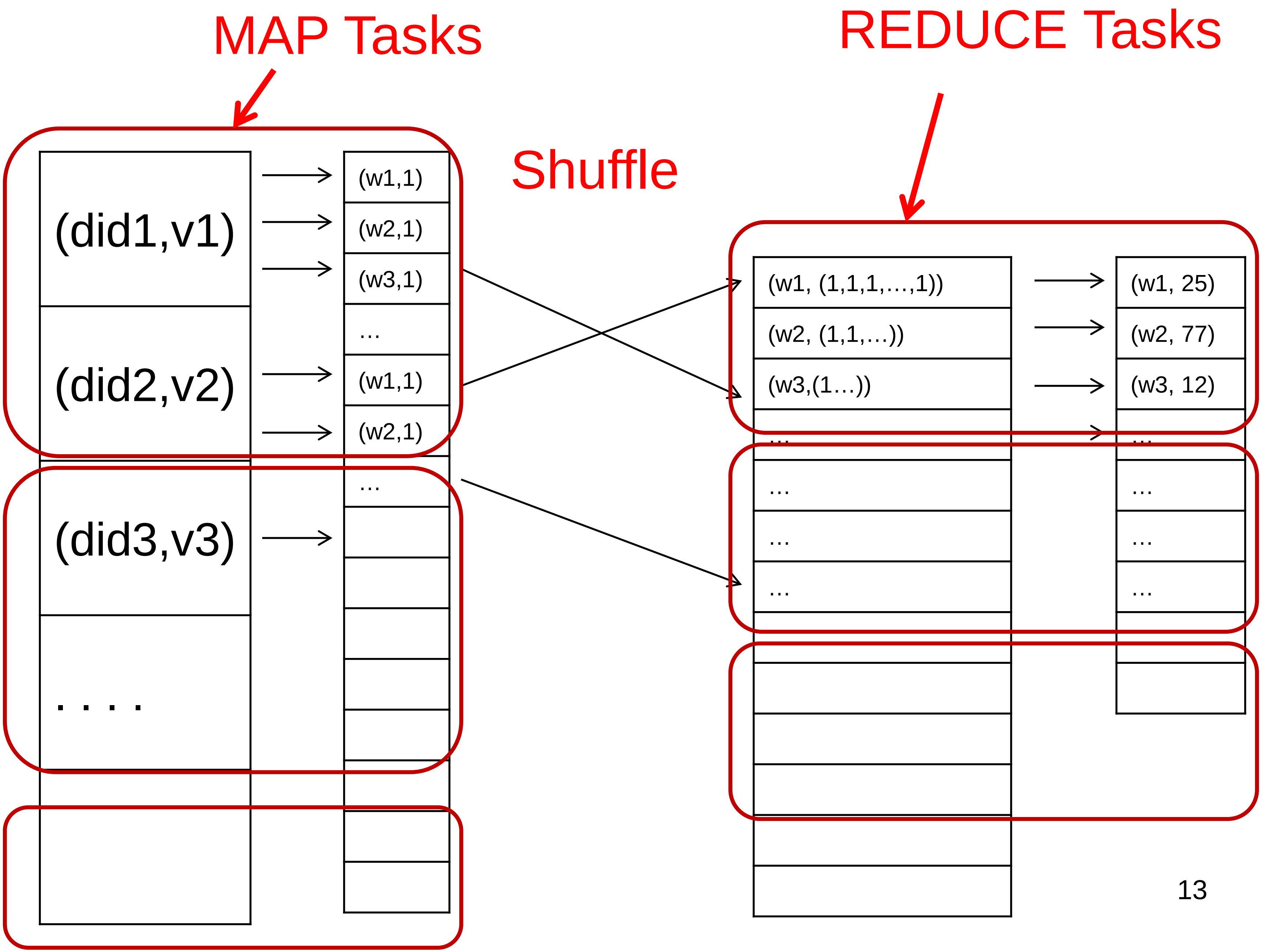
Workers

- 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

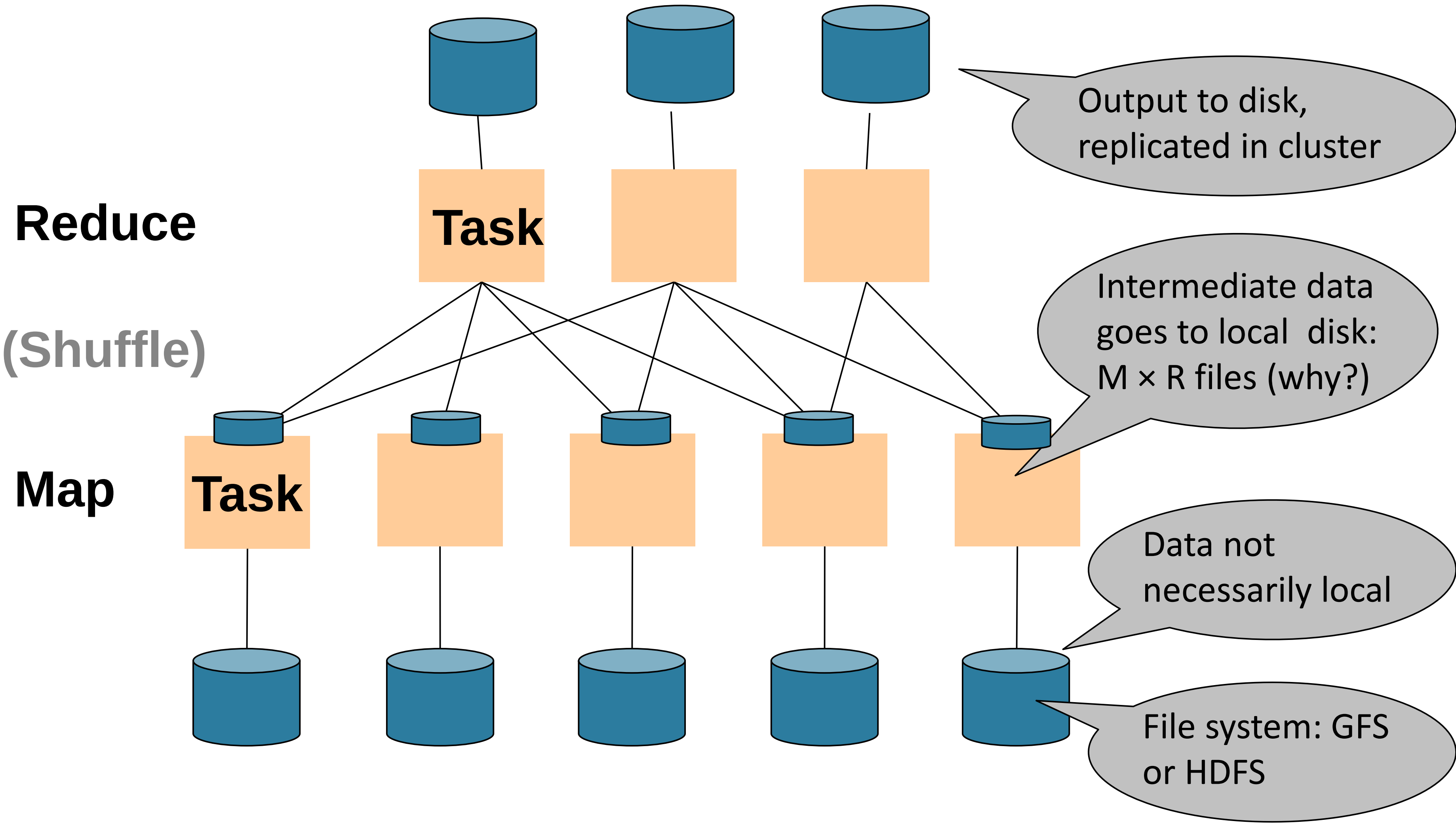
Fault Tolerance

- 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



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

- Next lecture: Spark

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
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

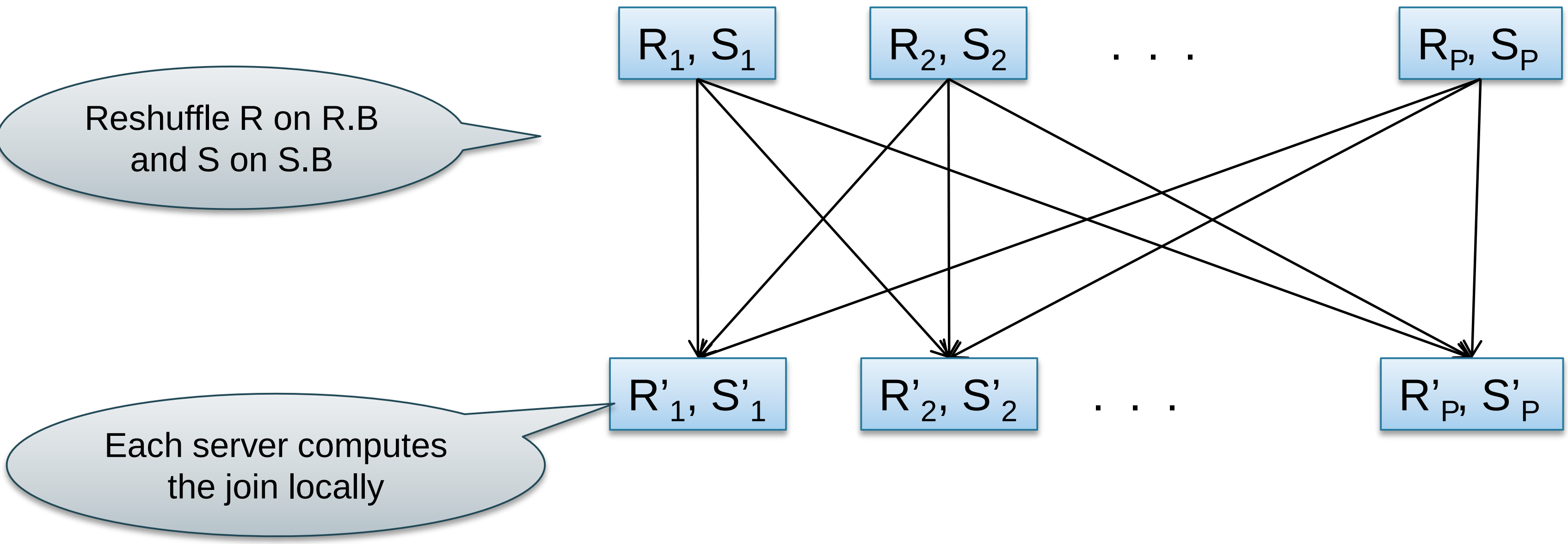
Two simple parallel join algorithms:

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

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

Partitioned Hash-Join

Initially, both R and S are horizontally partitioned



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

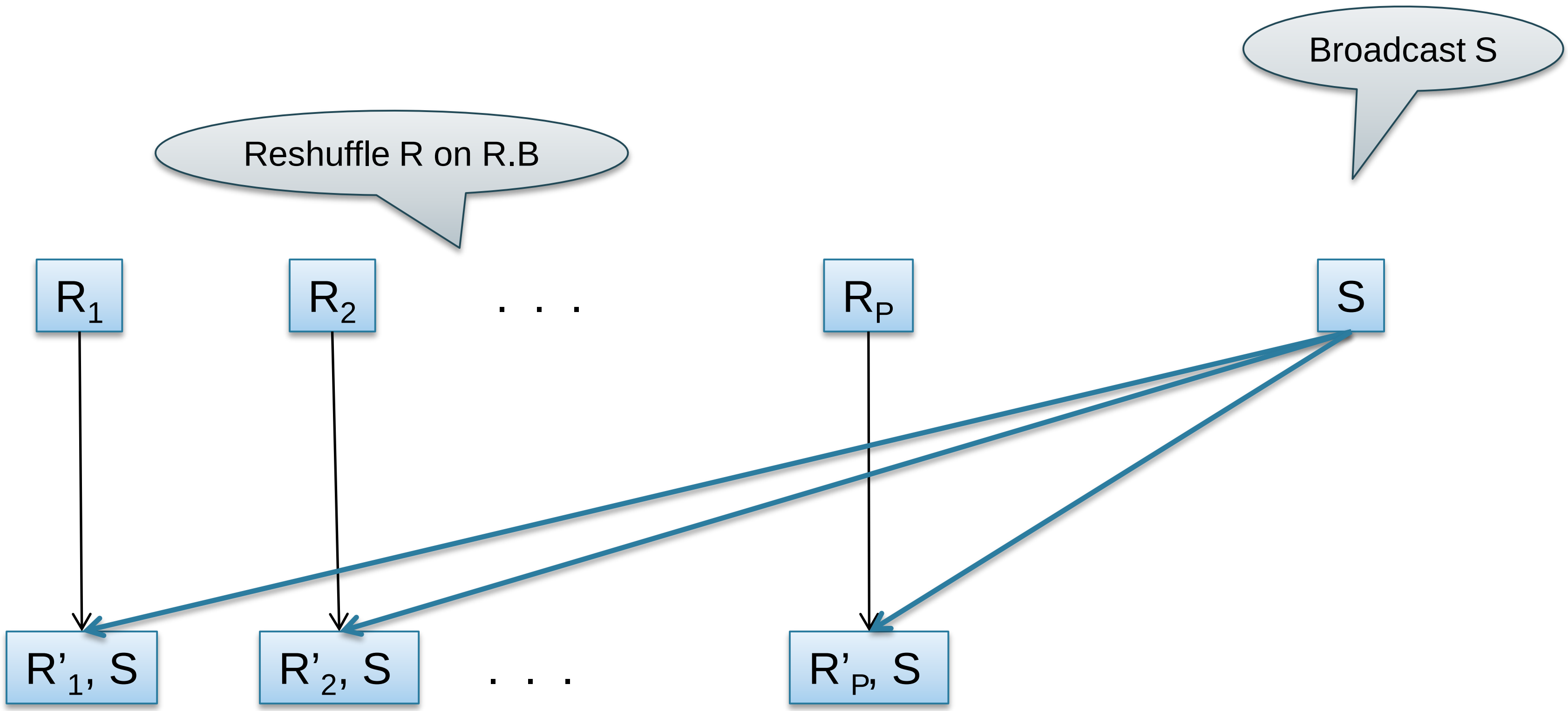
Partitioned Hash-Join

```
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.B, 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