

# Database System Internals

# MapReduce

Paul G. Allen School of Computer Science and Engineering  
University of Washington, Seattle

# Parallel Data Processing

## OLAP: Online Analytical Processing

- Big queries: joins, group-by, large data
- No updates
- Use parallelism/distribution to improve performance
- Challenge: optimize ONE query

## OLTP: Online Transaction Processing

- Big data, but simple query: many simple updates
- Distribute data to support large workloads
- Challenge: ACID or something weaker

# This lecture

Data model?

Relational

Scaleup goal?

OLAP

Architecture?

Shared-Nothing

# This lecture

Data model?

Relational



text/kv-pairs

Scaleup goal?

OLAP

Architecture?

Shared-Nothing

# References

- **MapReduce: Simplified Data Processing on Large Clusters.** Jeffrey Dean and Sanjay Ghemawat. OSDI'04
- Mining of Massive Datasets, by Rajaraman and Ullman, <http://i.stanford.edu/~ullman/mmds.html>
  - Map-reduce (Section 20.2);
  - Chapter 2 (Sections 1,2,3 only)

# A Note

- MapReduce is obsolete now  
Interesting only from a historical perspective
- It has had an important influence, still visible today, but newer systems do a better job at adopting traditional database principles:
  - Spark
  - Snowflake -- standard highly distributed SQL

# Map Reduce Review

- Google: [Dean 2004]
- Open source implementation: Hadoop
  
- MapReduce = high-level programming model and implementation for large-scale parallel data processing

# MapReduce Motivation

- Not designed to be a DBMS
- But to simplify task of writing parallel programs
  - Simple programming model that applies to many problems
- Hides messy details in runtime library:
  - Automatic parallelization
  - Load balancing
  - Network and disk transfer optimizations
  - Handling of machine failures
  - Robustness

content in part from: Jeff Dean

# Data Processing at Massive Scale

- Massive parallelism:
  - 100s, or 1000s, or 10000s servers (think data center)
  - Many hours
  
- Failure:
  - If medium-time-between-failure is 1 year
  - Then 10000 servers have one failure / hour

# Data Storage: GFS/HDFS

- MapReduce job input is a file
- Distributed file system:
  - GFS: Google File System
  - HDFS: Hadoop File System
- File is split into “blocks” or “chunks”: 64MB or so
- Blocks are replicated & stored on random machines
- Files are append only

# MapReduce: 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)`
- Output: `bag of (intermediate key, value)`

System applies 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:
  - Original MR paper: bag of output (values)
  - Hadoop: bag of (output key, values)

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

# Example

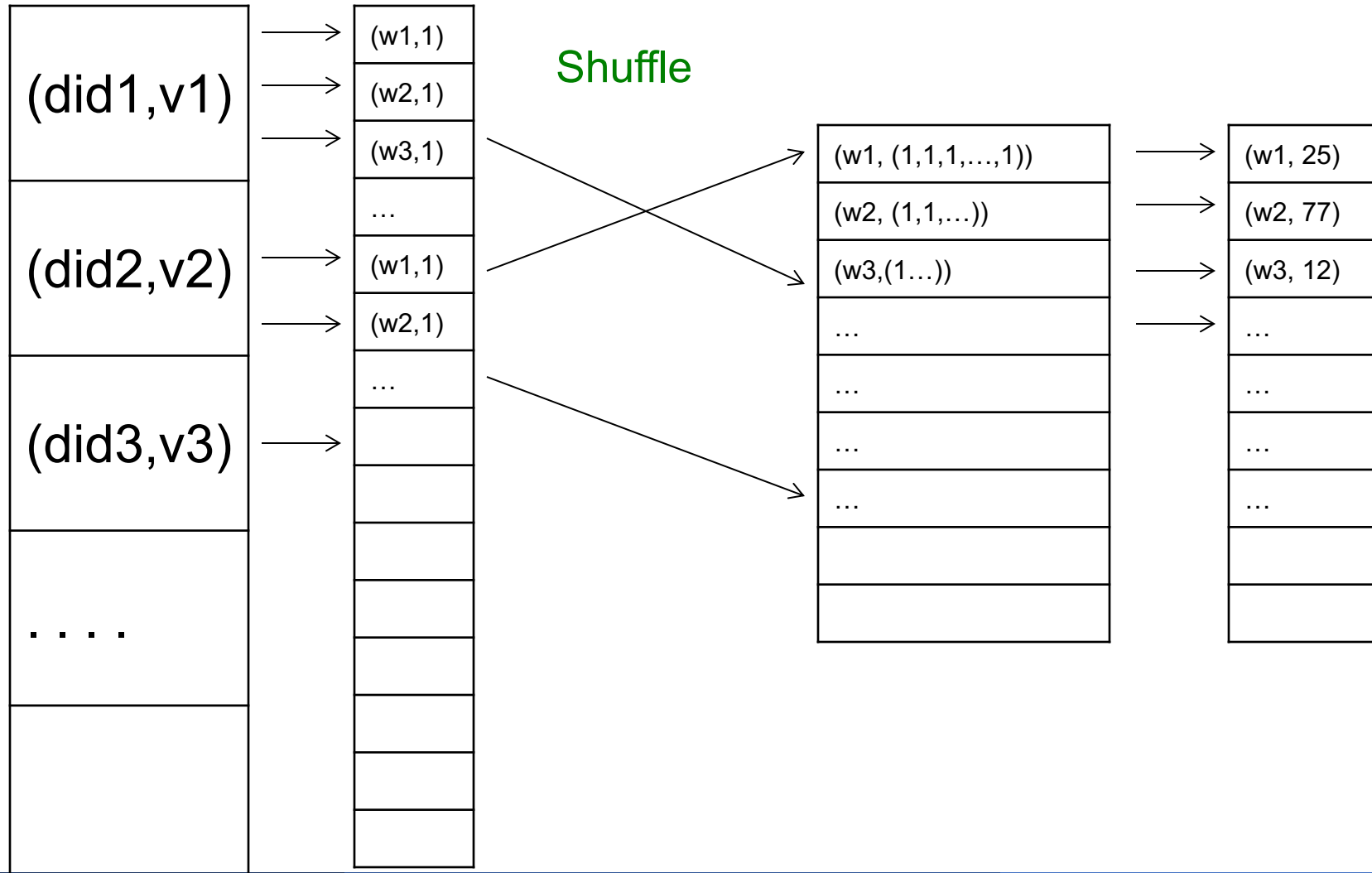
- Counting the number of occurrences of each word in a large collection of documents
- Each Document
  - The **key** = document id (**did**)
  - The **value** = set of words (**word**)

```
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



# Jobs vs. Tasks

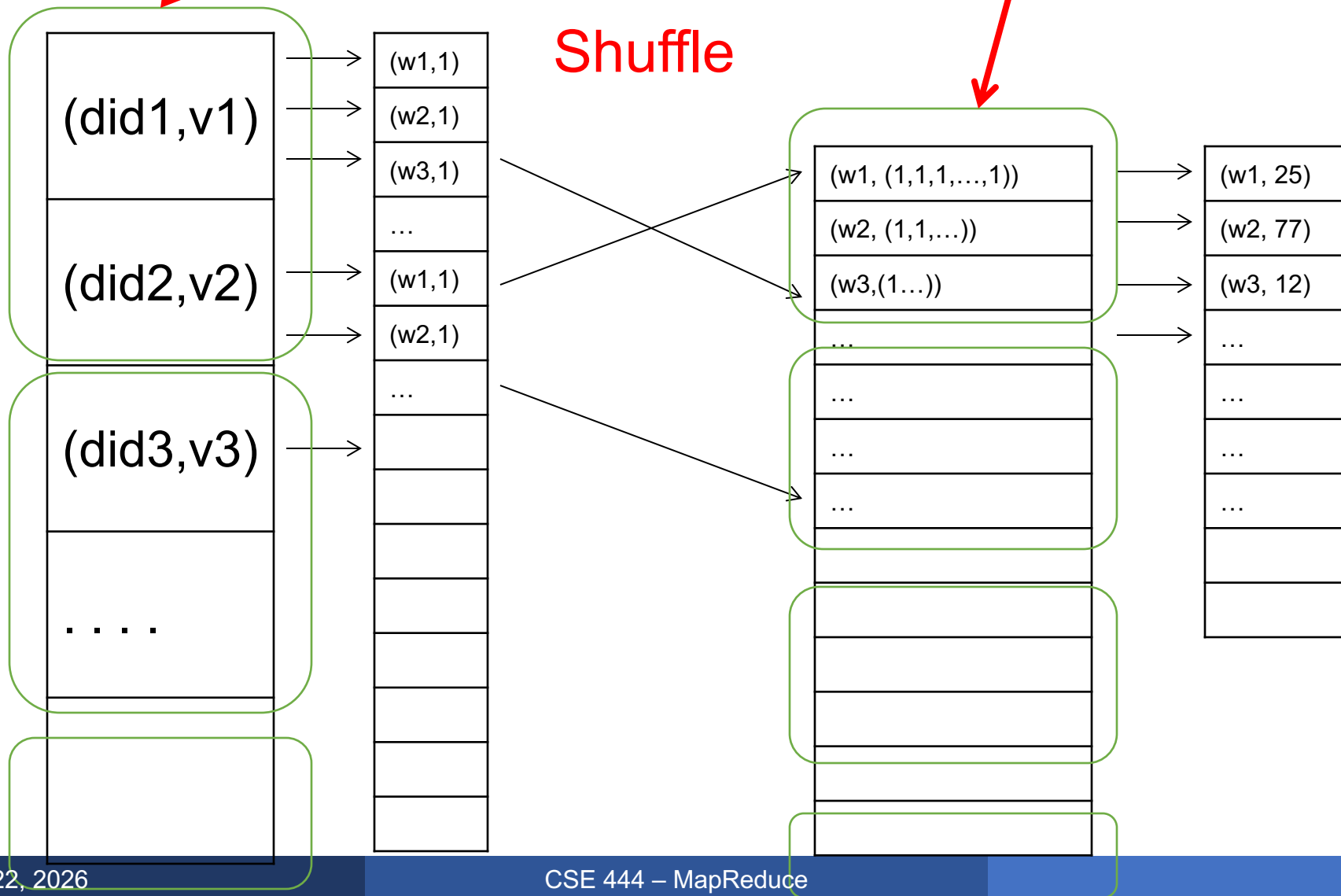
- A **MapReduce Job**
  - One single “query”, e.g. count the words in all docs
  - More complex queries may consists of multiple jobs
- A **Map Task**, or a **Reduce Task**
  - A group of instantiations of the map-, or reduce-function, which are scheduled on a single worker

# Workers

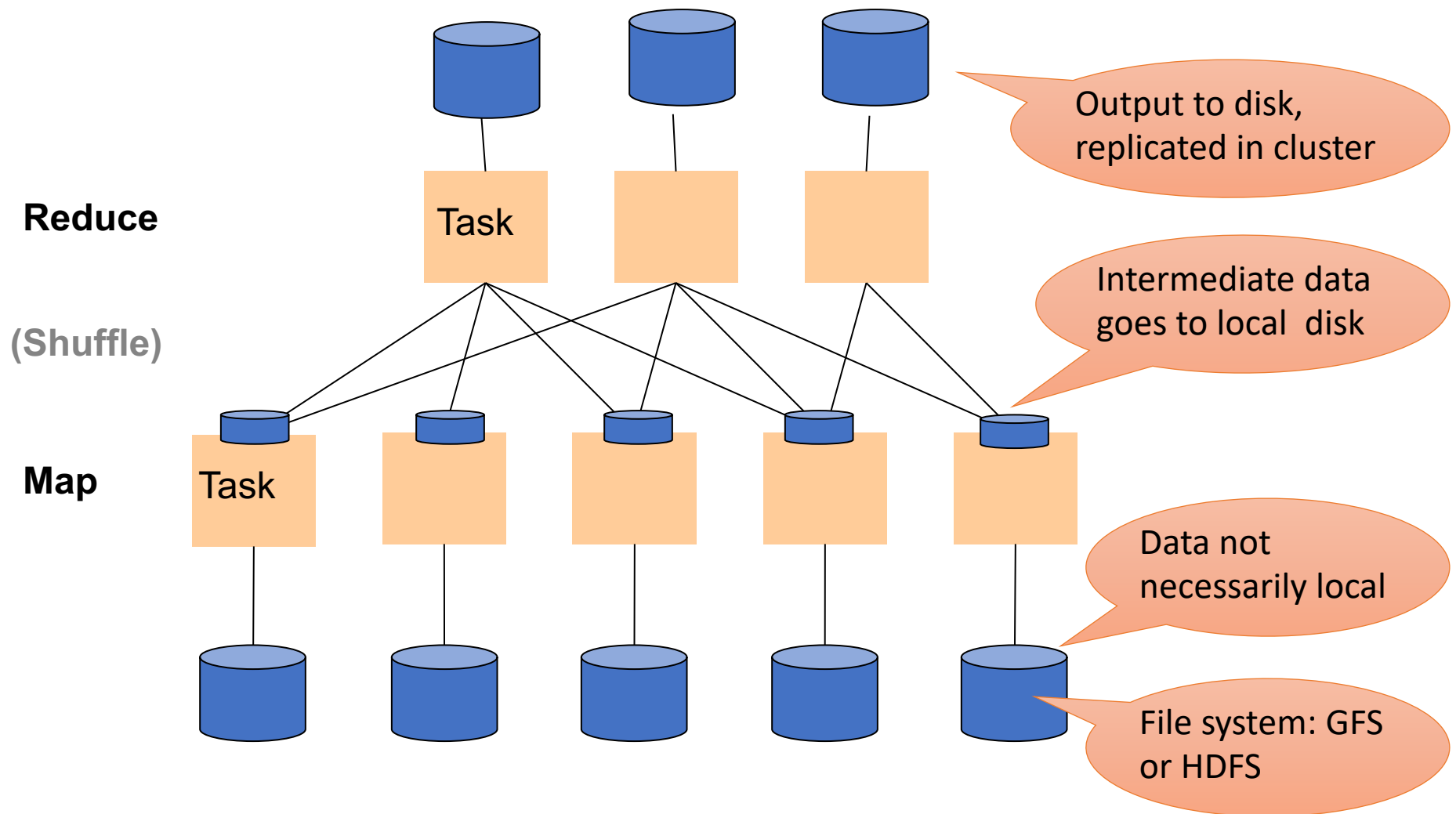
- A **worker** is a process that executes one task at a time
- Typically there is one worker per processor, hence 4 or 8 per node
- Often talk about “slots”
  - E.g., Each server has 2 map slots and 2 reduce slots

MAP Tasks

REDUCE Tasks



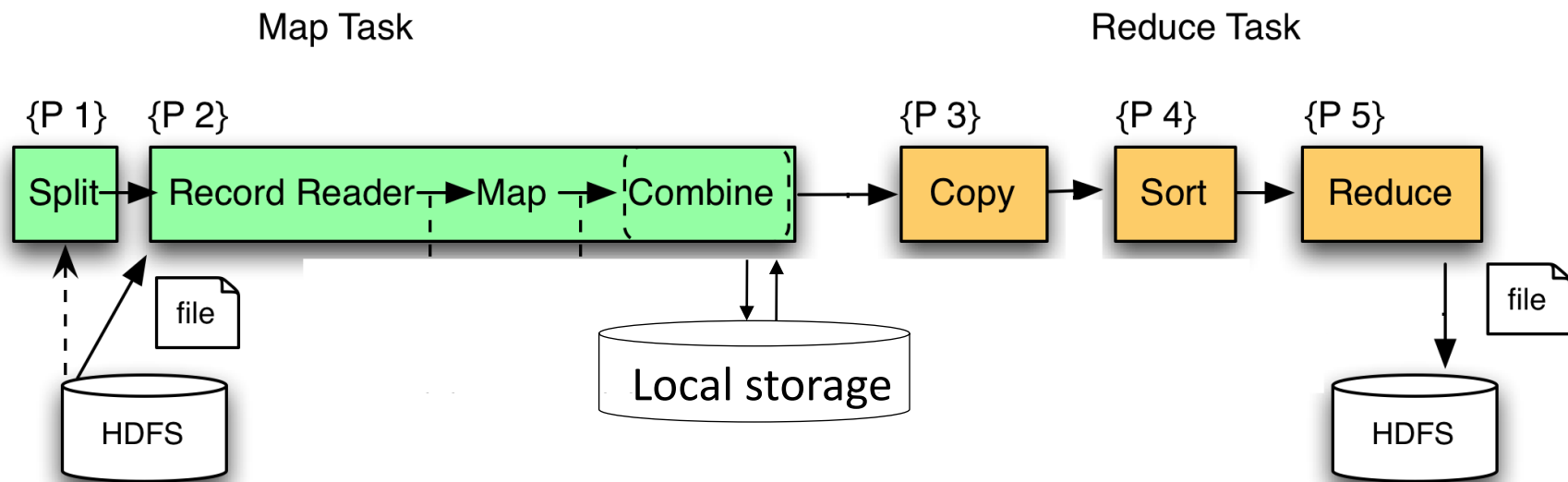
# Parallel MapReduce Details



# MapReduce Implementation

- There is one master node
- Input file gets partitioned further into  $M'$  splits
  - Each split is a contiguous piece of the input file
  - By default splits correspond to blocks
- Master assigns *workers* (=servers) to the  $M'$  *map tasks*, keeps track of their progress
- Workers write their output to local disk
- Output of each map task is partitioned into  $R$  *regions*
- Master assigns workers to the  $R$  *reduce tasks*
- Reduce workers read regions from the map workers' local disks

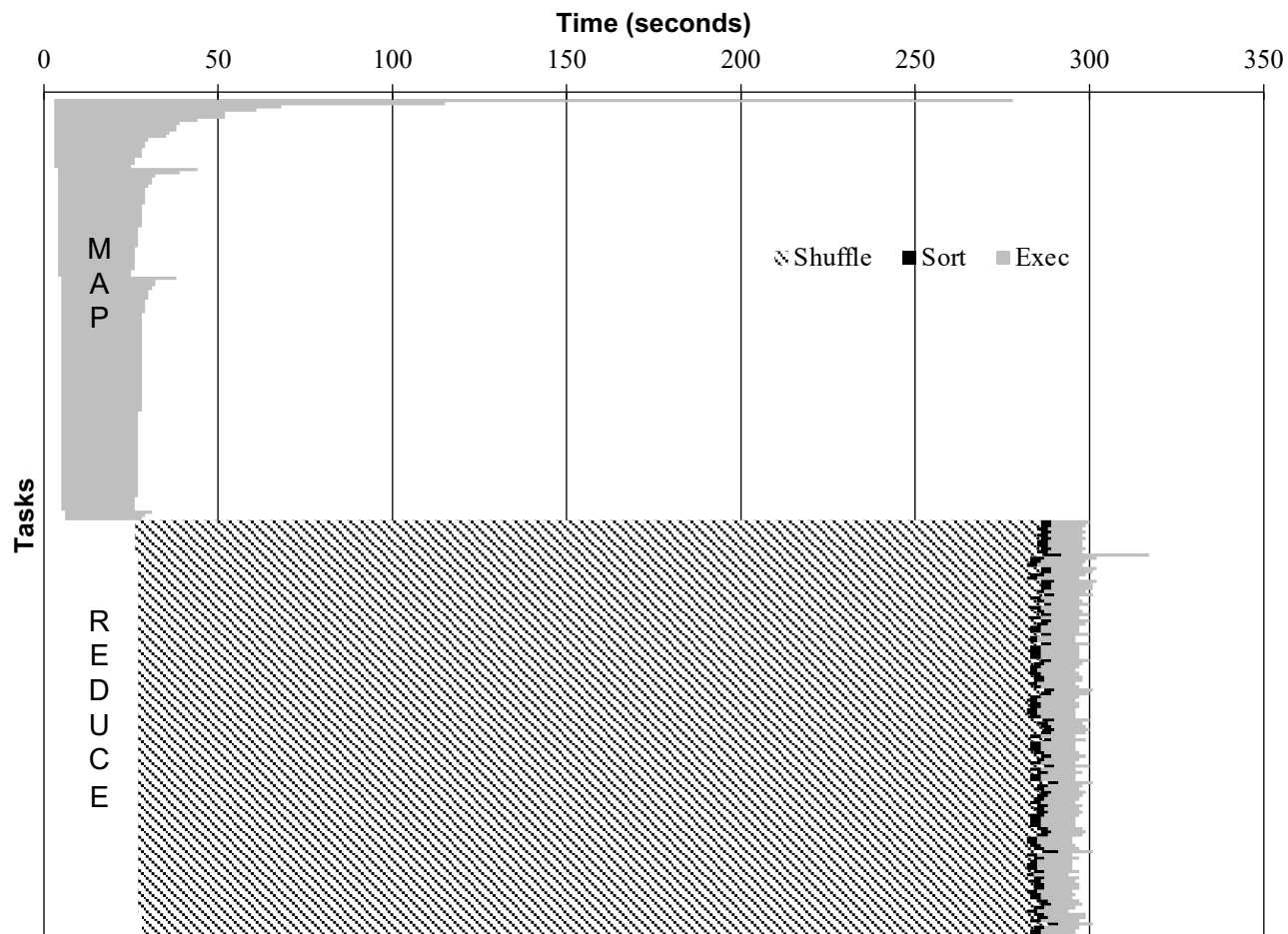
# MapReduce Phases



# Skew

## PageRank Application

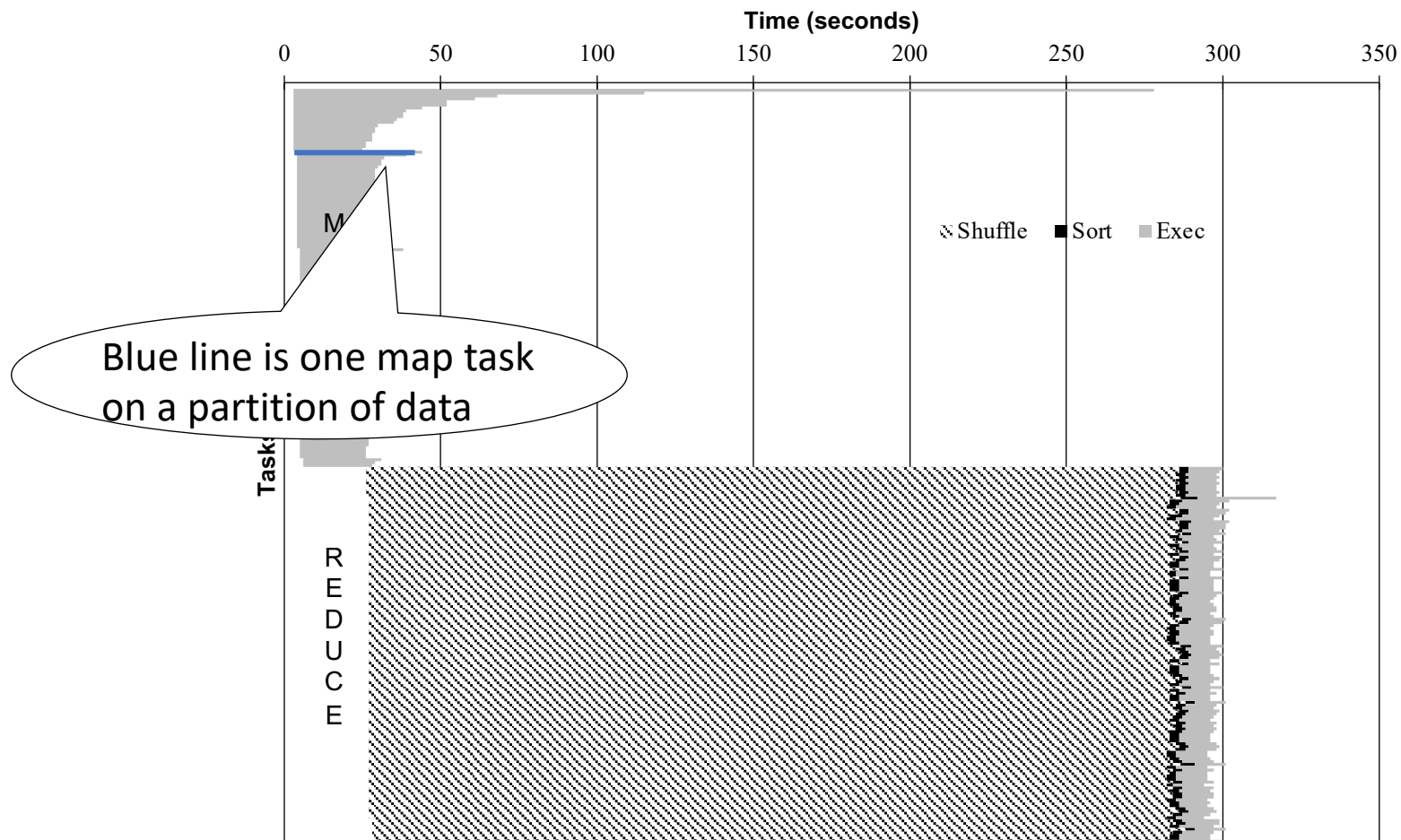
- Reduce tasks do not begin until all map tasks are finished



# Skew

## PageRank Application

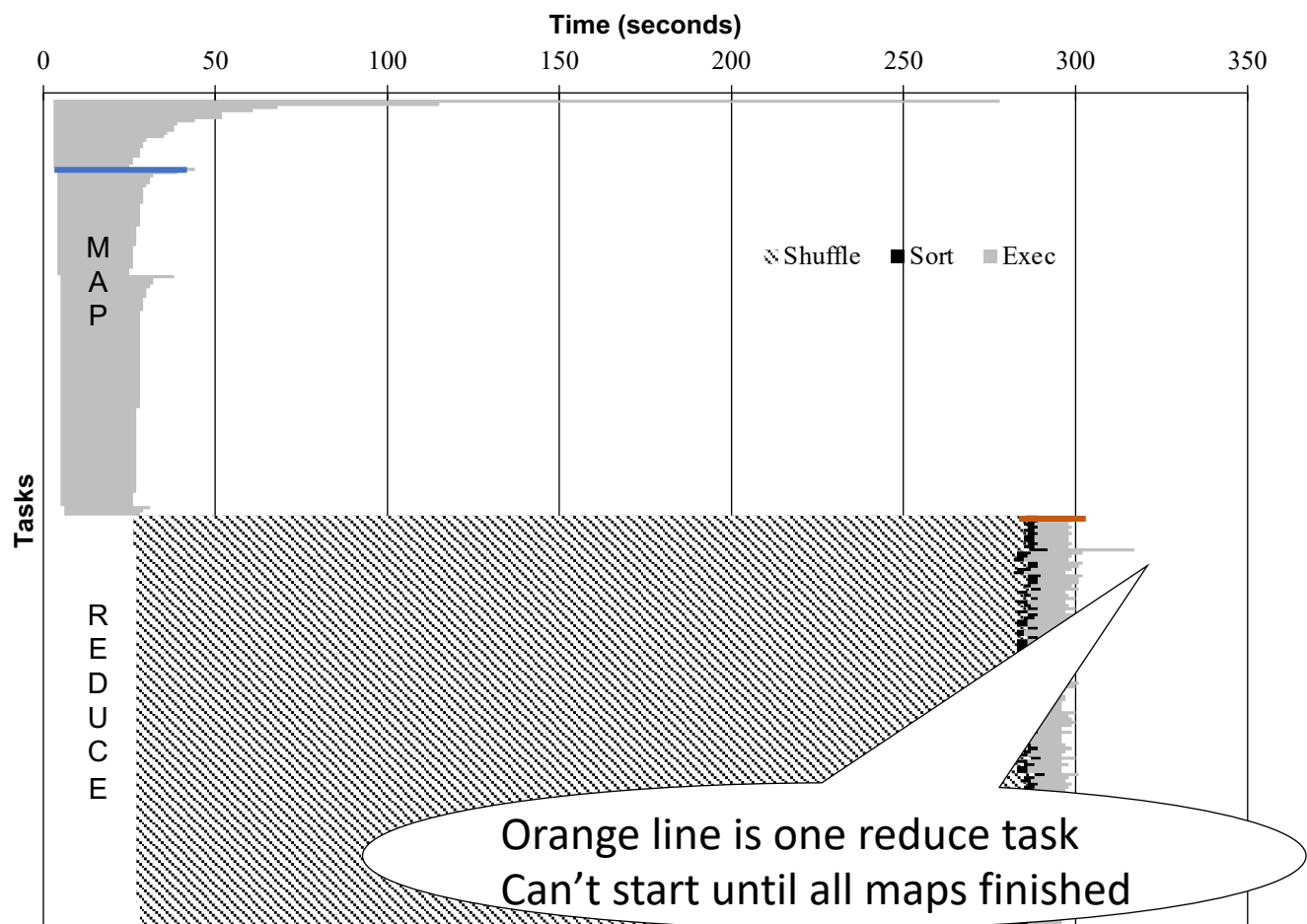
- Reduce tasks do not begin until all map tasks are finished



# Skew

## PageRank Application

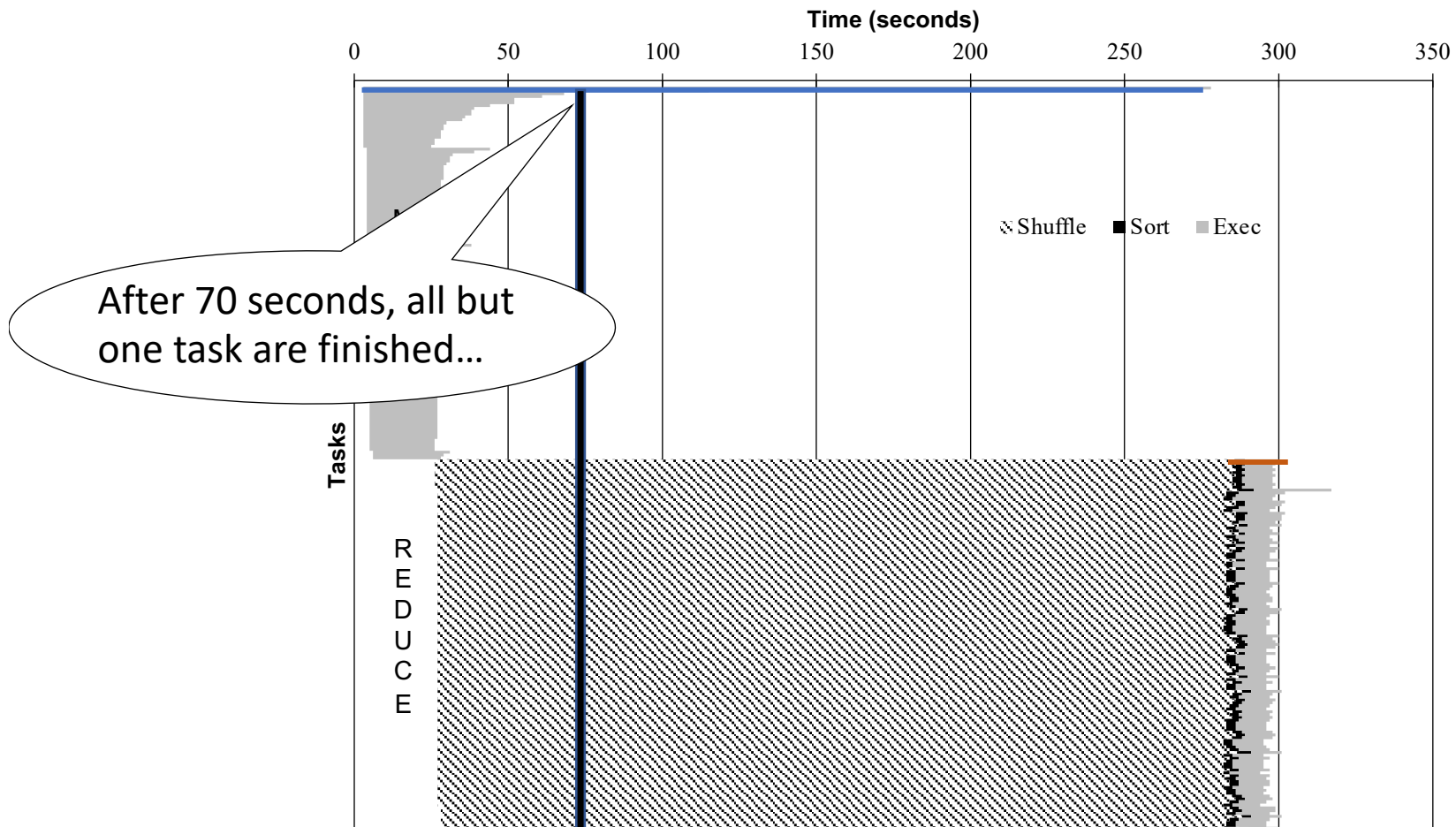
- Reduce tasks do not begin until all map tasks are finished



# Skew

## PageRank Application

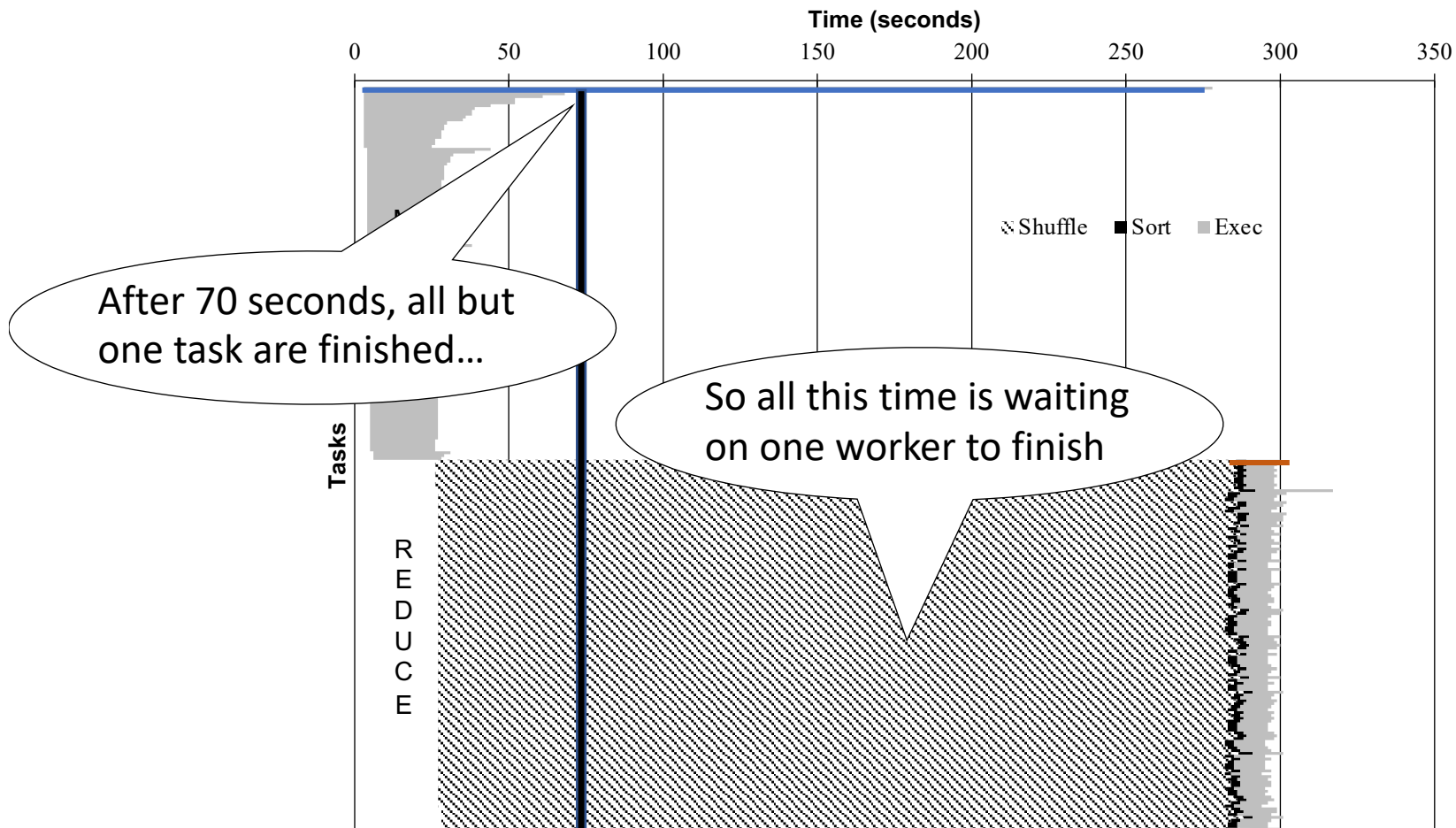
- Reduce tasks do not begin until all map tasks are finished



# Skew

## PageRank Application

- Reduce tasks do not begin until all map tasks are finished



# Hadoop dashboard example (“datadog”)

☆ Hadoop - Overview

Enable dashboard time controls...

\$cluster \$node \$application \$job

ESC /



# Interesting Implementation Details

- Worker failure:
  - Master pings workers periodically,
  - If down then reassigns its task to **another worker**
  - ( $\neq$  a parallel DBMS restarts whole query)
  
- How many map and reduce tasks:
  - Larger is better for load balancing
  - But more tasks also add overheads
  - ( $\neq$  parallel DBMS spreads ops across all nodes)

# 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*

# The State of MapReduce Systems

- Lots of extensions to address limitations
  - Capabilities to write DAGs of MapReduce jobs
  - Declarative languages
  - Ability to read from structured storage (e.g., indexes)
  - Etc.
- Most companies use both types of engines (MR and DBMS), with increased integration
- New systems emerged which improve over MapReduce: e.g. Spark

# Relational Queries over MR

- Query → query plan
- Each operator → one MapReduce job

# GroupBy in MapReduce

Doc(key, word)

MapReduce IS A GroupBy!

**MAP**=GROUP BY, **REDUCE**=Aggregate

```
SELECT word, sum(1)
FROM Doc
GROUP BY word
```

# Joins in MapReduce

- If MR is GROUP-BY plus AGGREGATE, then how do we compute  $R(A,B) \bowtie S(B,C)$  using MR?

# Joins in MapReduce

- If MR is GROUP-BY plus AGGREGATE, then how do we compute  $R(A,B) \bowtie S(B,C)$  using MR?
- Answer:
  - Map: group R by R.B, group S by S.B
    - Input = either a tuple  $R(a,b)$  or a tuple  $S(b,c)$
    - Output =  $(b,R(a,b))$  or  $(b,S(b,c))$  respectively
  - Reduce:
    - Input =  $(b,\{R(a_1,b),R(a_2,b),\dots,S(b,c_1),S(b,c_2),\dots\})$
    - Output =  $\{R(a_1,b),R(a_2,b),\dots\} \times \{S(b,c_1),S(b,c_2),\dots\}$
    - In practice: improve the reduce function (next...)

# Join in MR

Users(name, age)  
Pages(userName, url)

```
Users = load 'users' as (name, age);  
Pages = load 'pages' as (userName, url);  
Jnd = join Users by name, Pages by userName;
```

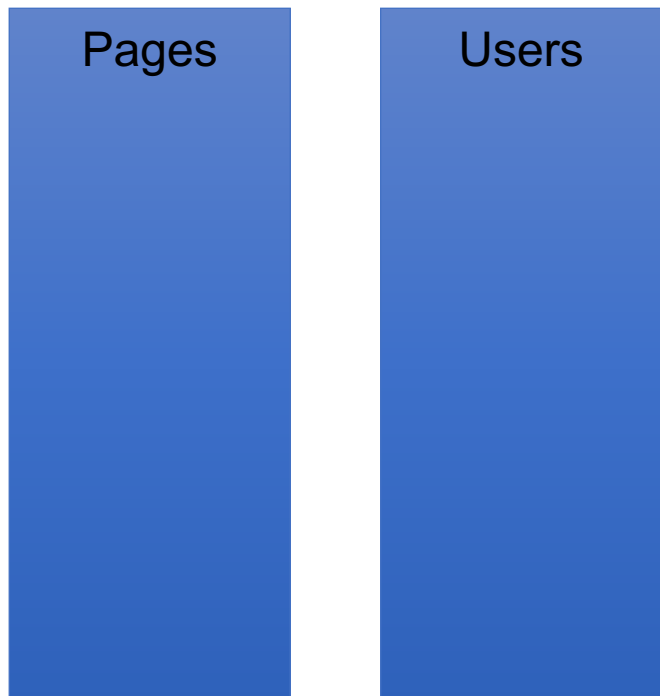
```
map([String key], String value):  
  // value.relation is either 'Users' or 'Pages'  
  if value.relation='Users':  
    EmitIntermediate(value.name, (1, value));  
  else // value.relation='Pages':  
    EmitIntermediate(value.userName, (2, value));
```

```
reduce(String user, Iterator values):  
  Users = empty; Pages = empty;  
  for each v in values:  
    if v.type = 1: Users.insert(v)  
    else Pages.insert(v);  
  for v1 in Users, for v2 in Pages  
    Emit(v1,v2);
```

# Join in MR

Users(name, age)  
Pages(userName, url)

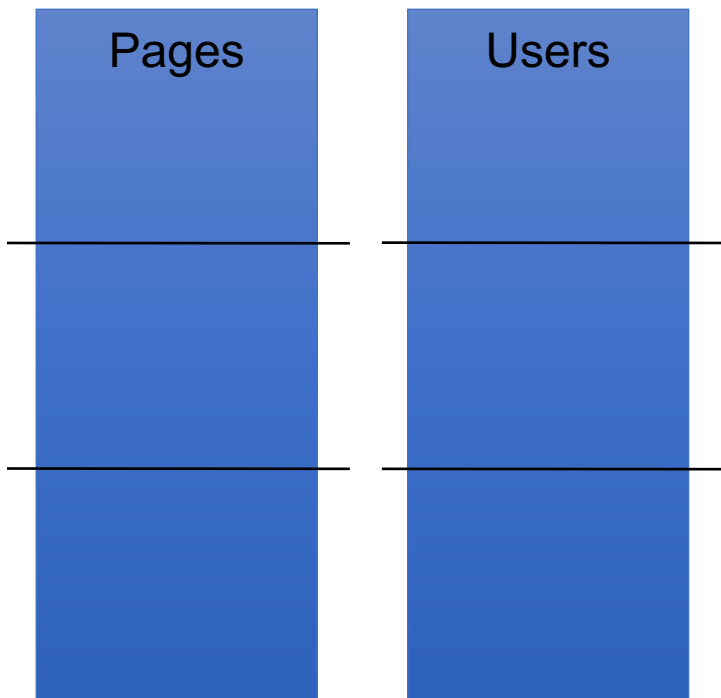
```
Users = load 'users' as (name, age);  
Pages = load 'pages' as (userName, url);  
Jnd = join Users by name, Pages by userName;
```



# Join in MR

Users(name, age)  
Pages(userName, url)

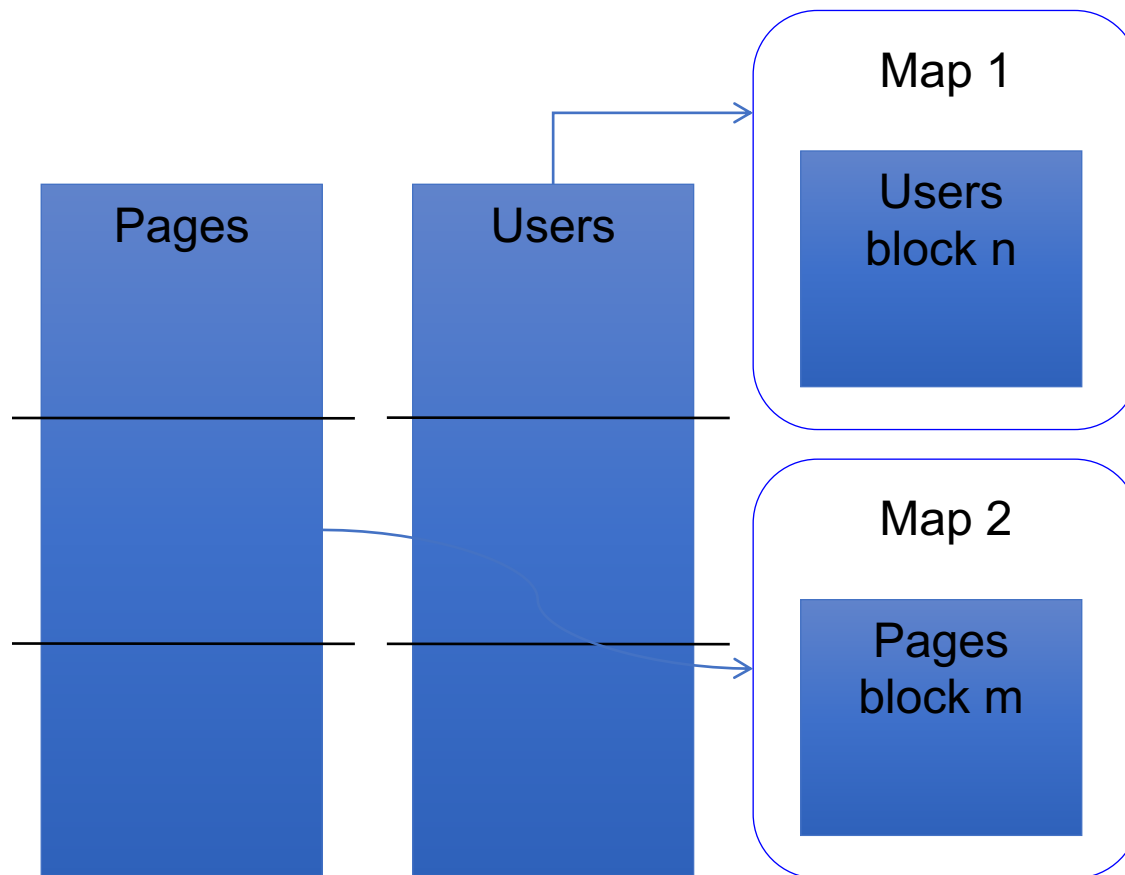
```
Users = load 'users' as (name, age);  
Pages = load 'pages' as (userName, url);  
Jnd = join Users by name, Pages by userName;
```



# Join in MR

Users(name, age)  
Pages(userName, url)

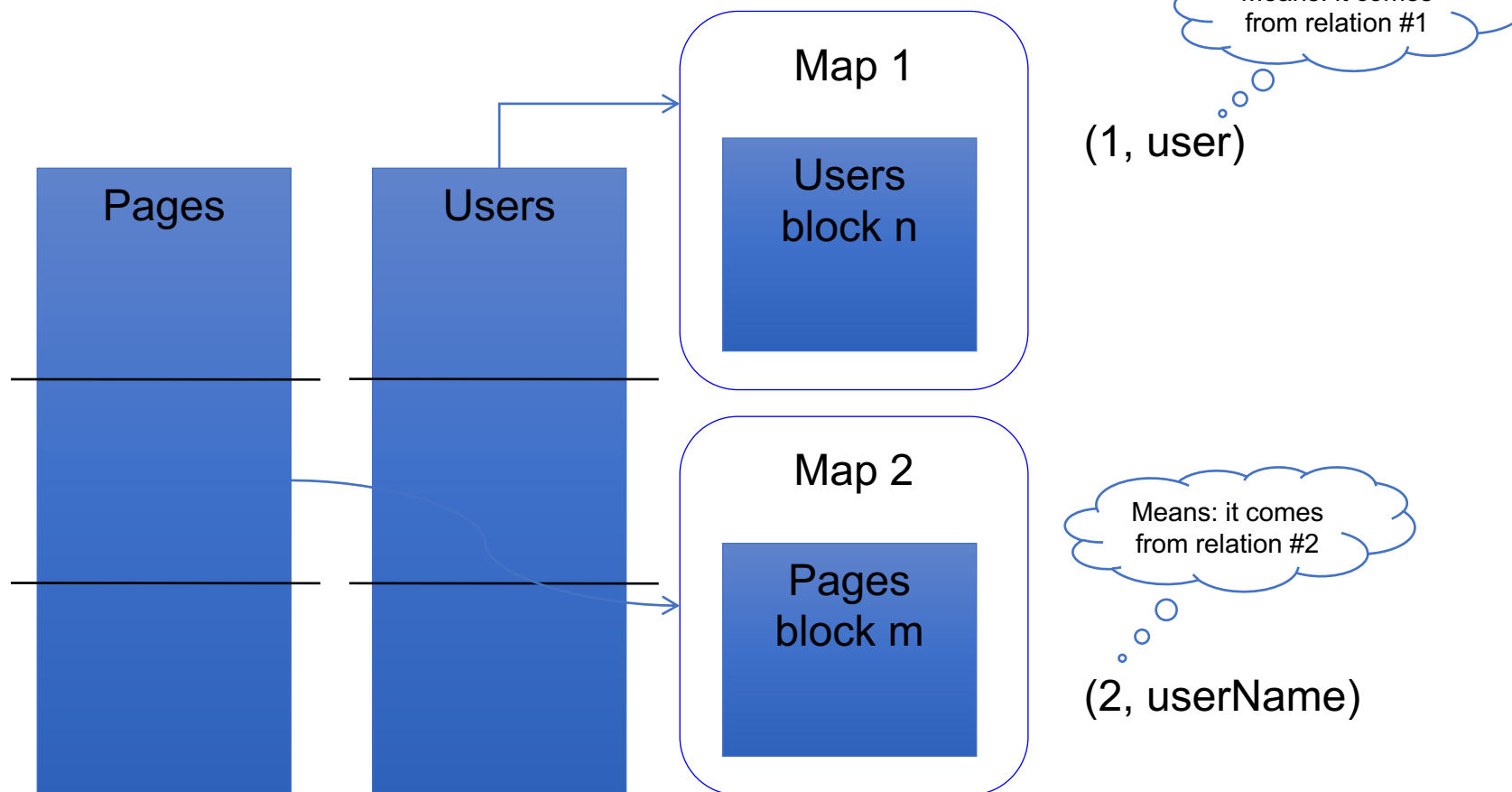
```
Users = load 'users' as (name, age);  
Pages = load 'pages' as (userName, url);  
Jnd = join Users by name, Pages by userName;
```



# Join in MR

Users(name, age)  
Pages(userName, url)

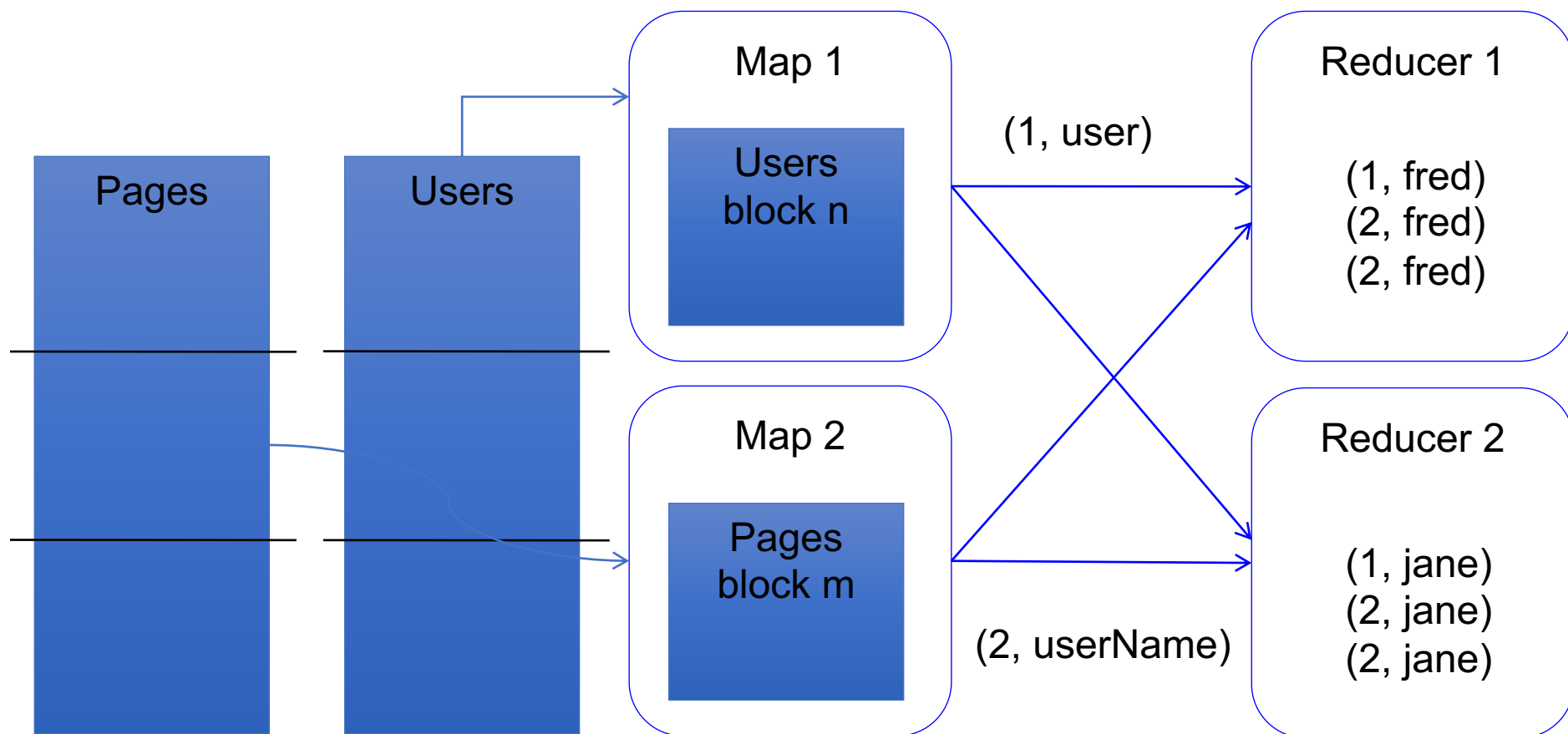
```
Users = load 'users' as (name, age);  
Pages = load 'pages' as (userName, url);  
Jnd = join Users by name, Pages by userName;
```



# Join in MR

Users(name, age)  
Pages(userName, url)

```
Users = load 'users' as (name, age);  
Pages = load 'pages' as (userName, url);  
Jnd = join Users by name, Pages by userName;
```



# Parallel DBMS vs MapReduce

## Parallel DBMS

- Relational data model and schema
- Declarative query language: SQL
- Many pre-defined operators: relational algebra
- Can easily combine operators into complex queries
- Query optimization, indexing, and physical tuning
- Streams data from one operator to the next without blocking
- **Can do more than just run queries: Data management**
  - Updates and transactions, constraints, security, etc.

*MapReduce: A major step backwards* article by David DeWitt

# Parallel DBMS vs MapReduce

## MapReduce

- Data model is a file with key-value pairs!
- No need to “load data” before processing it
- Easy to write user-defined operators
- Can easily add nodes to the cluster (no need to even restart)
- Uses less memory since processes one key-group at a time
- Intra-query fault-tolerance thanks to results on disk
- Intermediate results on disk also facilitate scheduling
- Handles adverse conditions: e.g., stragglers
- **Arguably more scalable... but also needs more nodes!**

*MapReduce: A major step backwards* article by David DeWitt