CSE 444: Database Internals

Lectures 21

MapReduce

Announcements

- HW5 due tonight!

- Lab5 due on Wednesday

- Next lab: choice of
  - Lab4 Query Optimization, or
  - Lab6 Parallel Databases

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)

Outline

- Review high-level MR ideas from 344

- Discuss implementation in greater detail

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

- Designed to simplify task of writing parallel programs
  - A simple programming model that applies to many large-scale computing problems

- Hides messy details in MapReduce runtime library:
  - Automatic parallelization
  - Load balancing
  - Network and disk transfer optimizations
  - Handling of machine failures
  - Robustness
  - **Improvements to core library benefit all users of library!**

content in part from: Jeff Dean

Data Processing at Massive Scale

- Want to process petabytes of data and more

- 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

- Common implementation is to store files in a highly scalable file system such as GFS/HDFS
  - GFS: Google File System
  - HDFS: Hadoop File System

  - Each data file is split into M blocks (64MB or more)
  - Blocks are stored on random machines & replicated
  - Files are append only

Observation: Your favorite parallel algorithm…

Reduce

(Shuffle)

Map
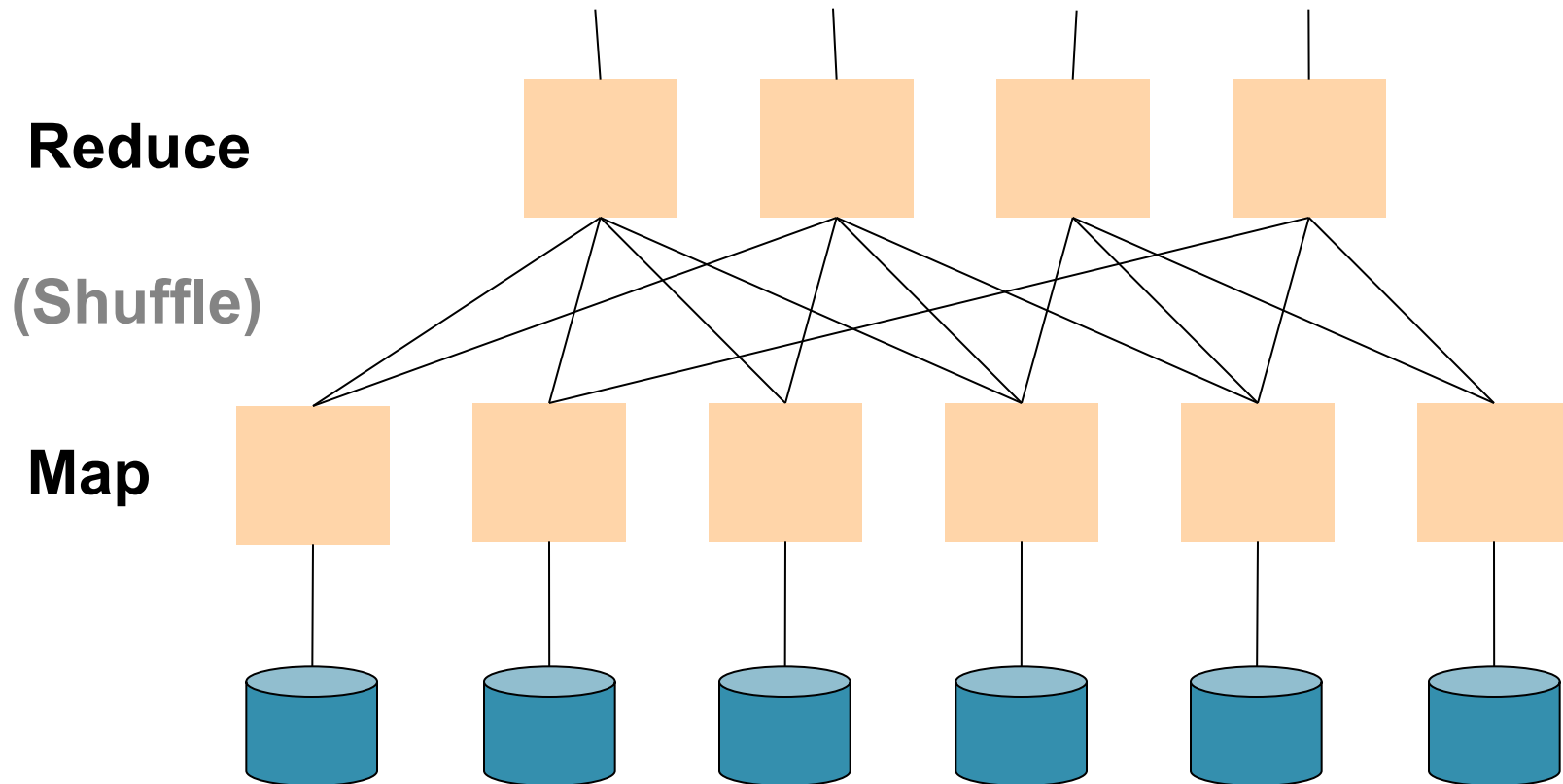
Typical Problems Solved by MR

- 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

Outline stays the same, map and reduce change to fit the problem

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 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)`**
- Output (Hadoop): bag of **`(output key, values)`**

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

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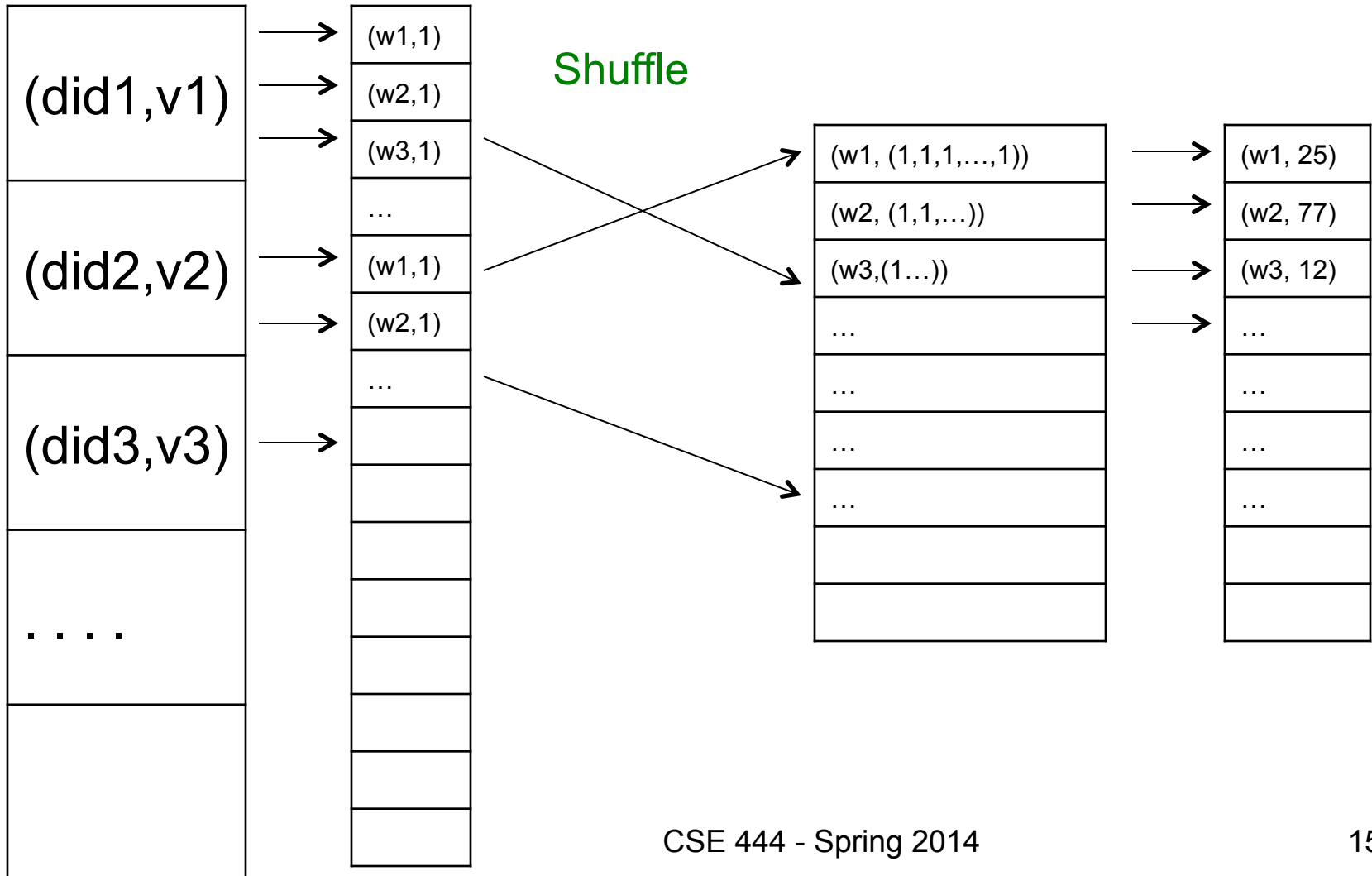
- Counting the number of occurrences of each word in a large collection of documents

- 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…))

…

…

…

…

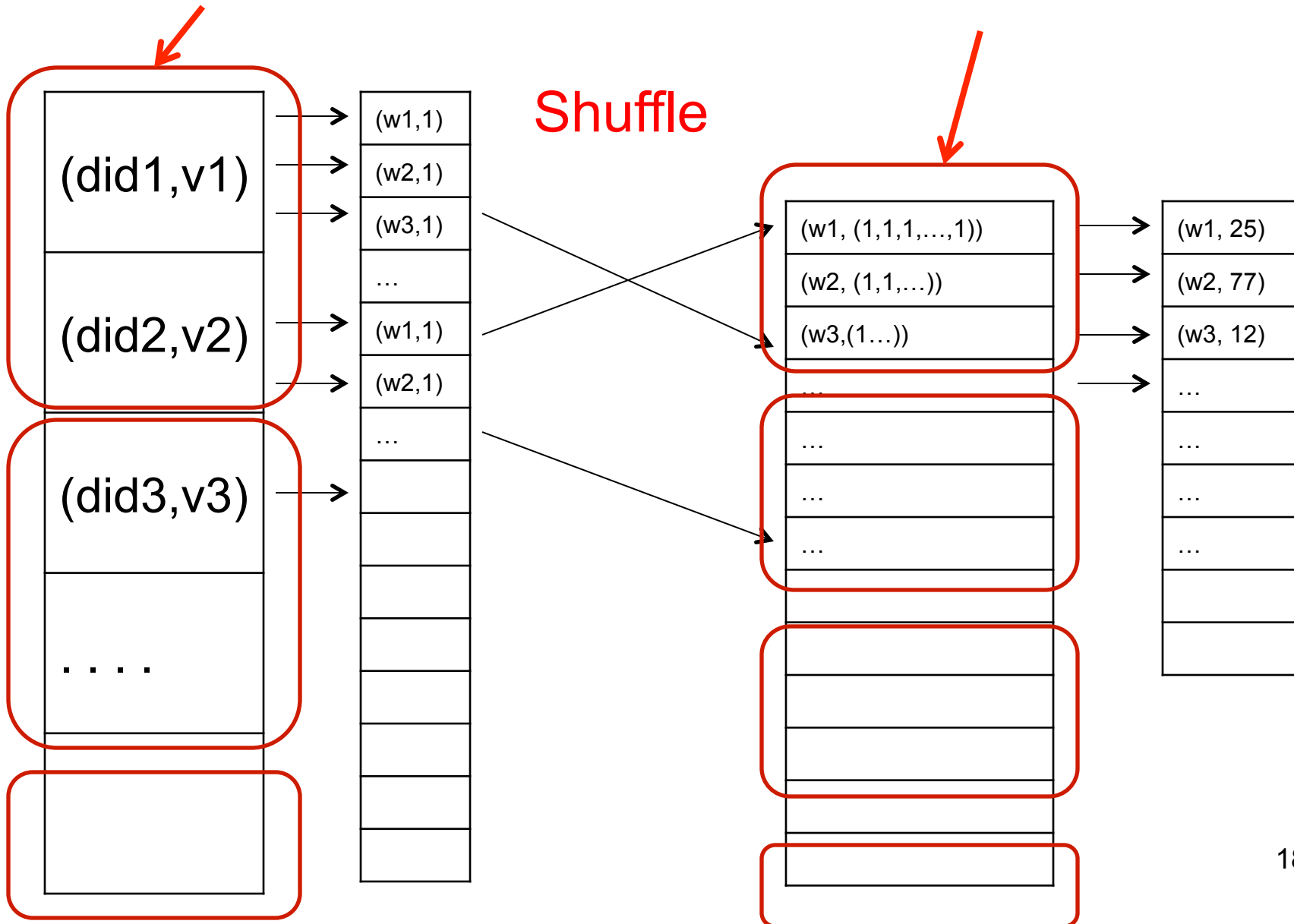(w1, 25)

(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
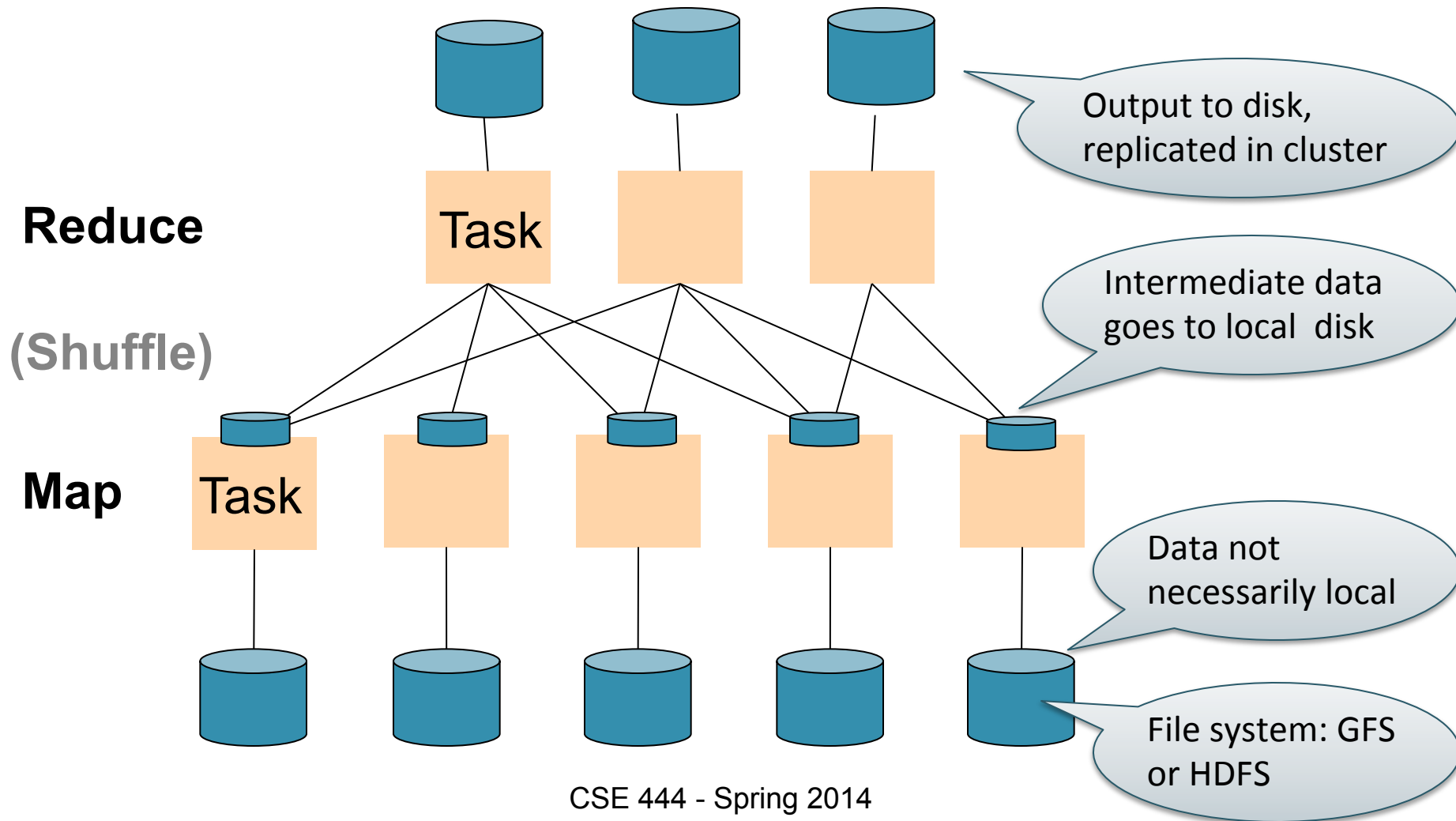
Workers

- 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

- Often talk about "slots"
  - E.g., Each server has 2 map slots and 2 reduce slots
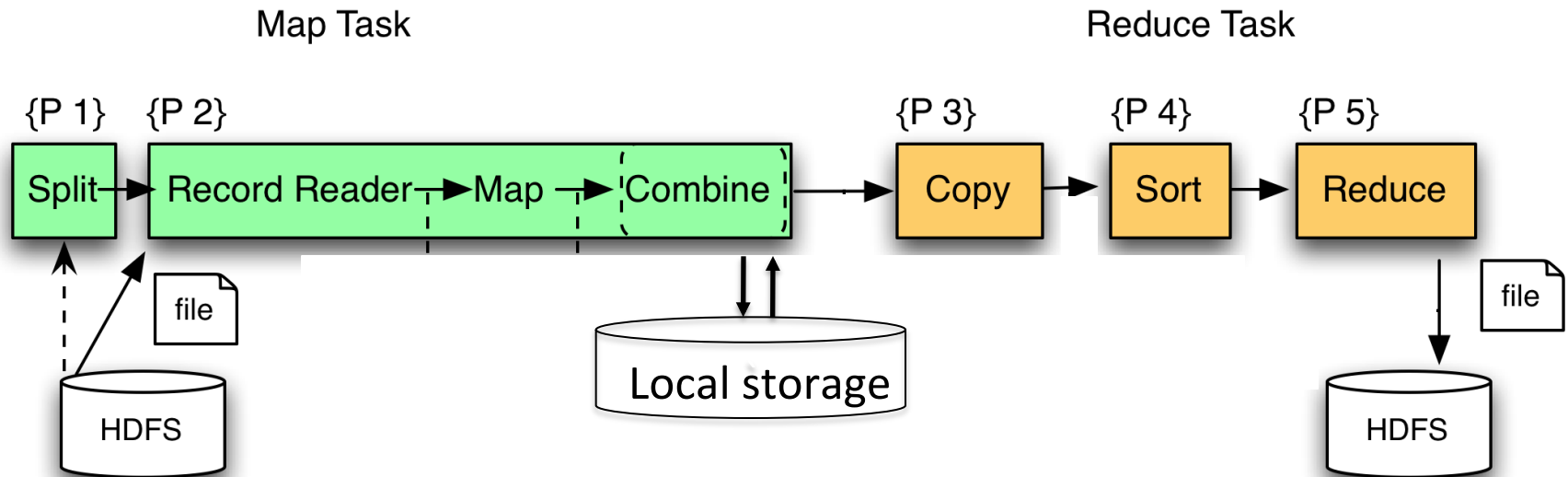
MAP Tasks

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)

...

...

...

...

Parallel MapReduce Details

Reduce

(Shuffle)

Map

Task

Task

Output to disk, replicated in cluster

Intermediate data goes to local disk

Data not necessarily local

File system: GFS or HDFS

CSE 444 - Spring 2014
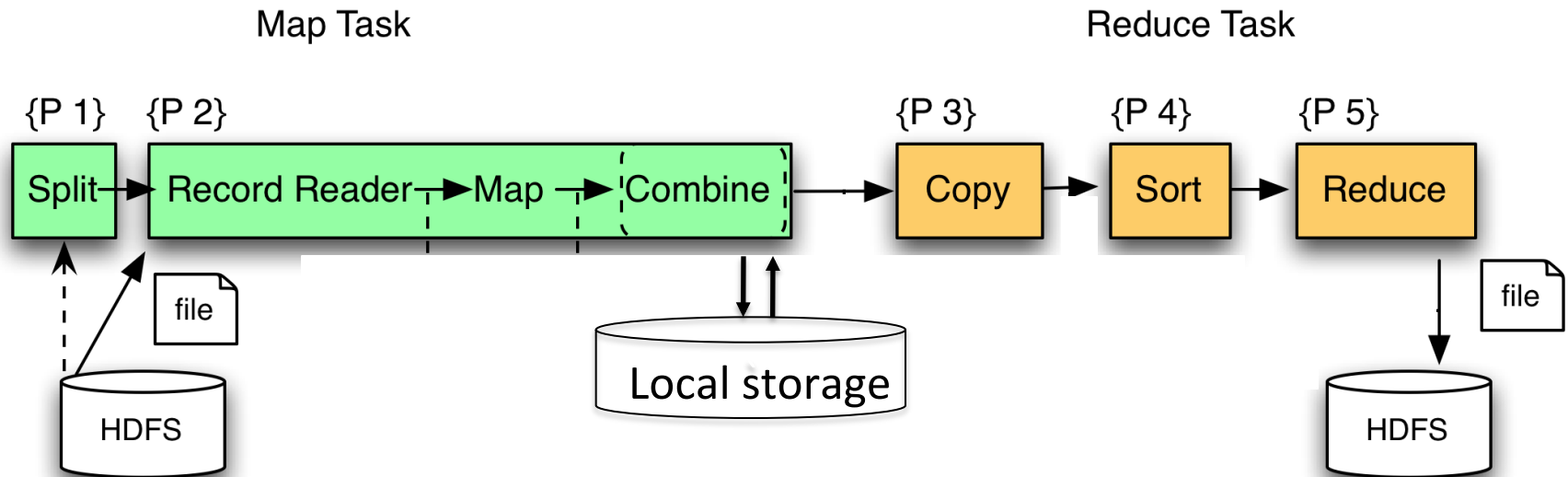
MapReduce Implementation

- There is one master node

- Input file gets partitioned further into *M' splits*

  – Each split is a contiguous piece of the input file

- 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*

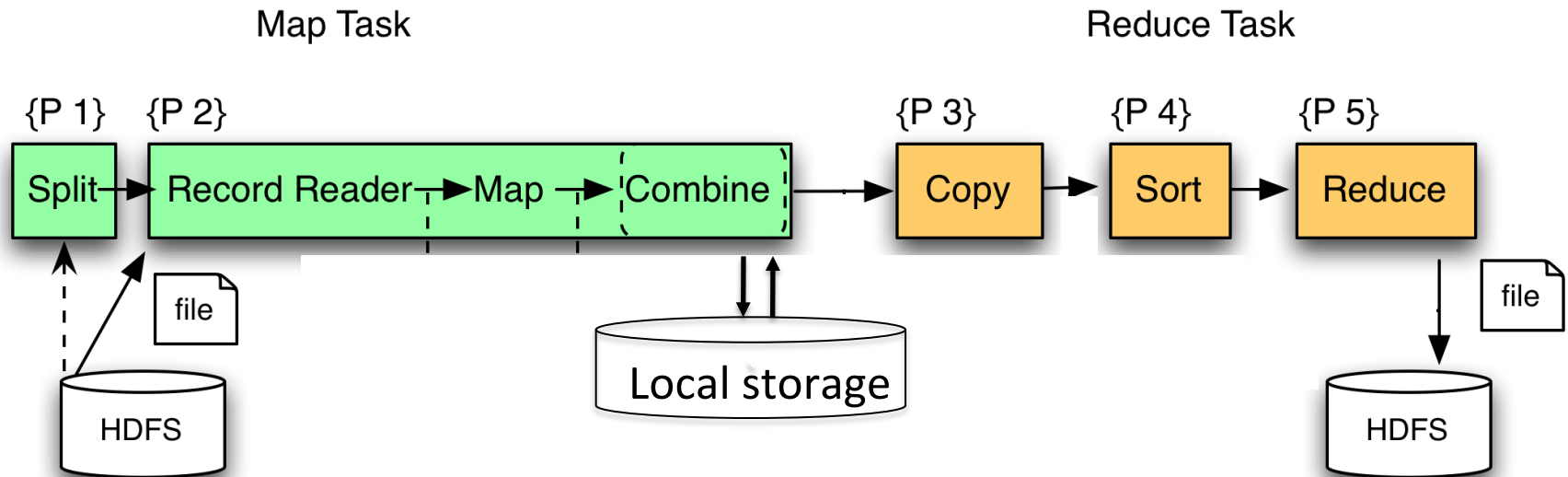- Reduce workers read regions from the map workers' local disks

MapReduce Phases

Map Task

Reduce Task

{P 1} {P 2}

{P 3} {P 4} {P 5}

Split → Record Reader → Map → Combine → Copy → Sort → Reduce

file

HDFS

Local storage

file

HDFS

MapReduce Phases

Q: If we compute an aggregate, when can we use a combiner?

Map Task

Reduce Task

{P 1}   {P 2}   {P 3}   {P 4}   {P 5}

Split → Record Reader → Map → Combine → · · Copy → Sort → Reduce

file

HDFS

Local storage

file

HDFS

MapReduce Phases

Q: If we compute an aggregate, when can we use a combiner?

Map Task

Reduce Task

{P 1} {P 2}   {P 3}   {P 4}   {P 5}

Split → Record Reader → Map → Combine → Copy → Sort → Reduce

file

HDFS

Local storage

file

HDFS

A: When the aggregate operator is distributive, or algebraic

Interesting Implementation Details

- Worker failure:
  - Master pings workers periodically,
  - If down then reassigns its task to *another* worker
  - (≠ a parallel DBMS restarts whole query)

- How many map and reduce tasks:
  - Larger is better for load balancing
  - But more tasks also add overheads
 - (≠ 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*

Skew

PageRank Application
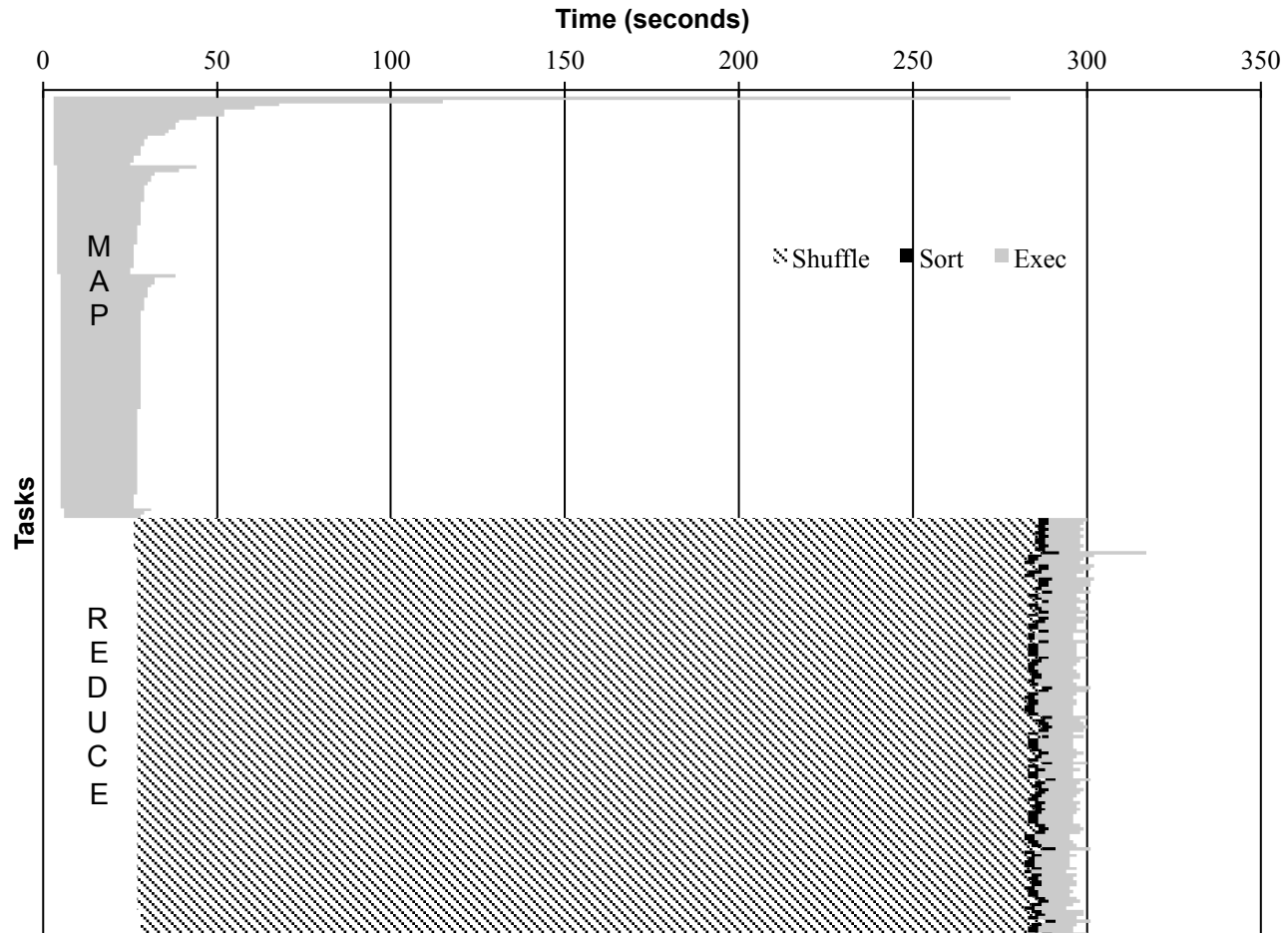
Parallel DBMS vs MapReduce

- ## 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.

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!

The State of MapReduce Systems

- Lots of extensions to address limitations
  - Capabilities to write DAGs of MapReduce jobs
  - Declarative languages (see 344)
  - Ability to read from structured storage (e.g., indexes)
  - Etc.
- Most companies use both types of engines (MR and DBMS), with increased integration
- Potential replacement to MR: Spark

Declarative Languages on MR

- PIG Latin (Yahoo!)
  - New language, like Relational Algebra
  - Open source
- HiveQL (Facebook)
  - SQL-like language
  - Open source
- SQL / Tenzing (Google)
  - SQL on MR
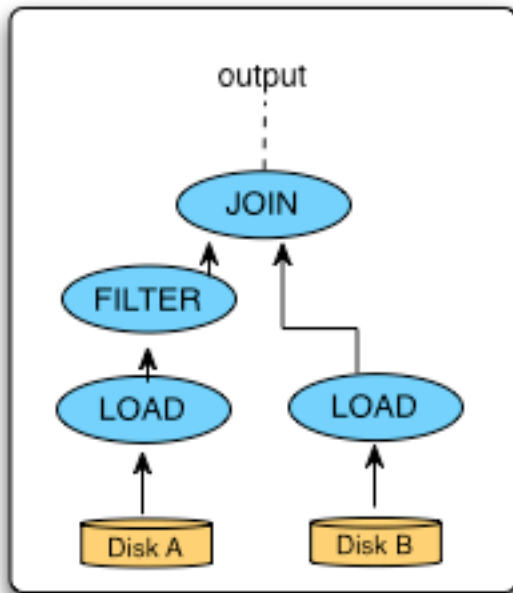 - Proprietary
  - Morphed into BigQuery

Relational Queries over MR

- Query → query plan
- Each operator → one MapReduce job
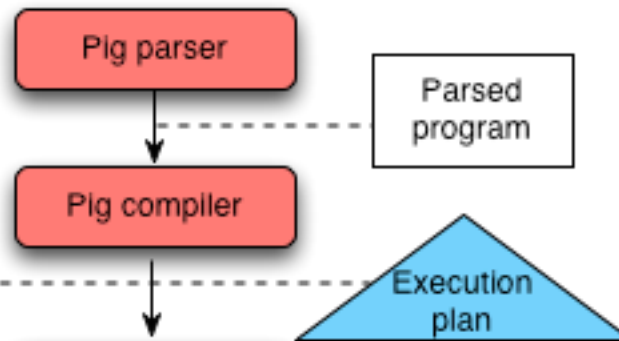
- Example: the Pig system

Background: Pig system


```
A = LOAD 'file1' AS (sid,pid,mass,px:double);
B = LOAD 'file2' AS (sid,pid,mass,px:double);
C = FILTER A BY px < 1.0;
D = JOIN C BY sid,
          B BY sid;
STORE g INTO 'output.txt';
```

Doc(key, word)

GroupBy in MapReduce

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) ⋈ S(B,C) using MR?

Joins in MapReduce

- If MR is GROUP-BY plus AGGREGATE, then how do we compute R(A,B) ⋈ 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(a1,b),R(a2,b),…,S(b,c1),S(b,c2),…})
    - Output = {R(a1,b),R(a2,b),…} × {S(b,c1),S(b,c2),…}
    - In practice: improve the reduce function (next…)

Hash Join in MR

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

Pages

Users

Hash Join in MR

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

Pages

Users

Hash Join in MR

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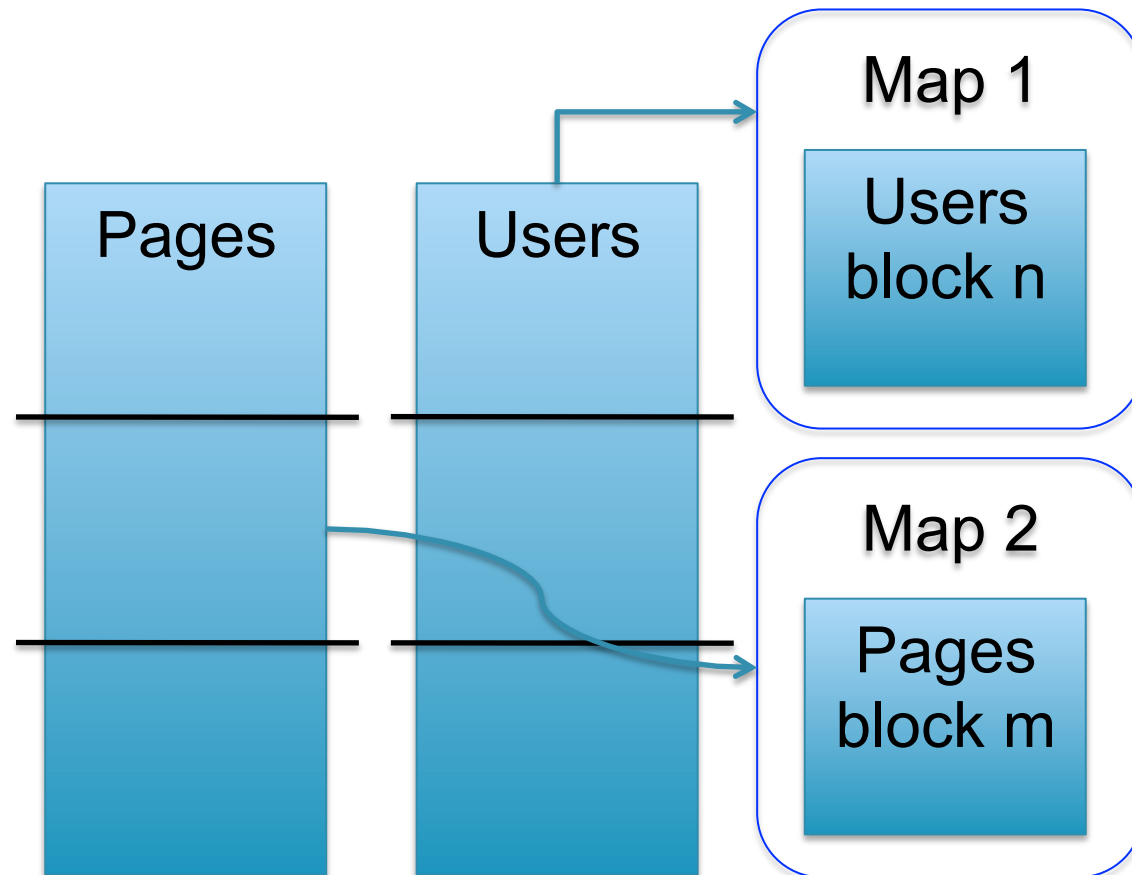
Hash Join in MR

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

Map 1

Users block n

Means: it comes from relation #1

(1, user)

Pages

Users

Map 2

Pages block m

Means: it comes from relation #2

(2, userName)

Hash Join in MR

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

Pages

Users

Map 1

Users block n

Map 2

Pages block m

(1, user)

(2, userName)

Reducer 1

(1, fred)
(2, fred)
(2, fred)

Reducer 2

(1, jane)
(2, jane)
(2, jane)

Hash Join in MR

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (userName, url);
Jnd = join Users by name, Pages by userName;
```
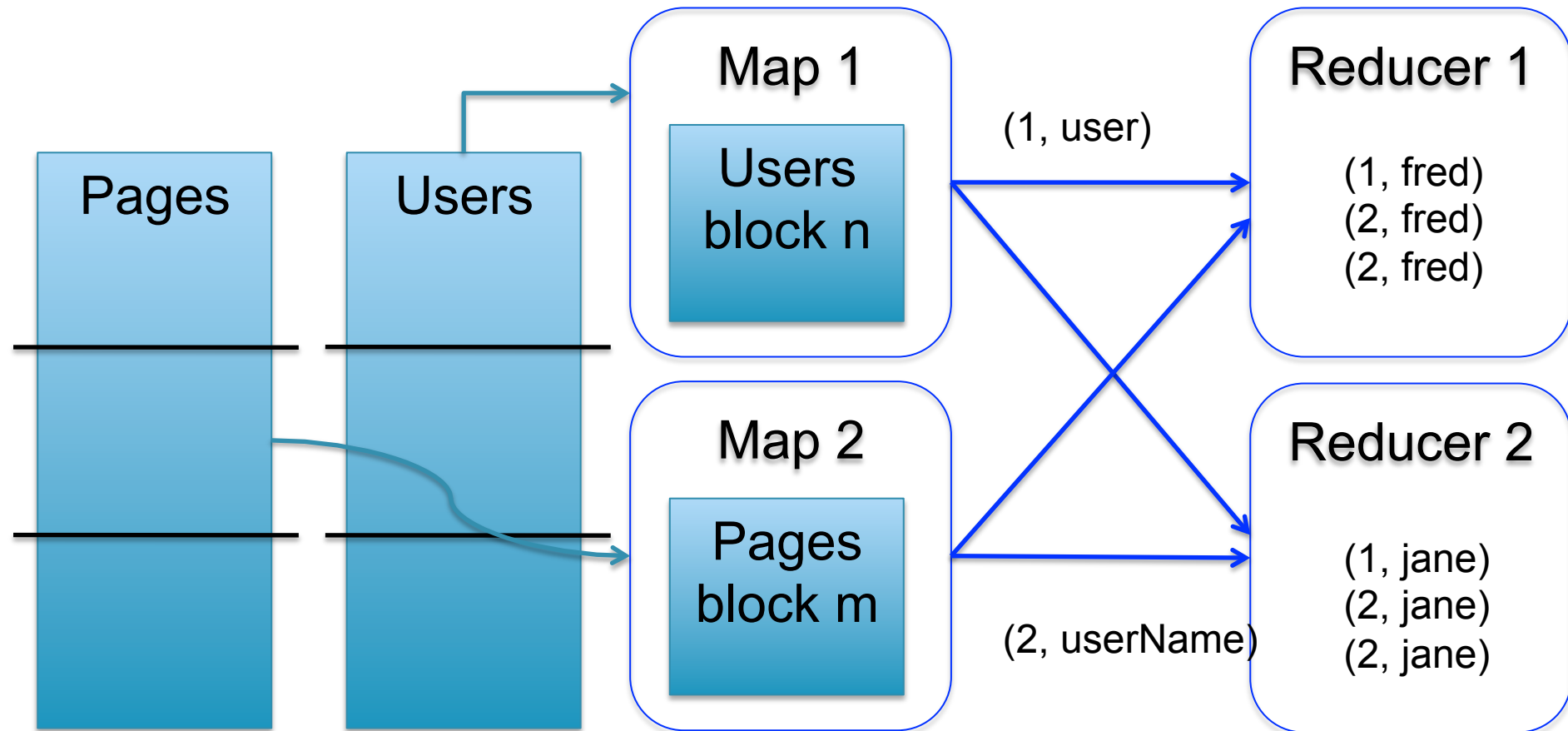
map([String key], String value):
    // value.relation is either 'Users' or 'Pages'
    if value.relation='Users':
        EmitIntermediate(value.name, (1, value));
    else // value.relation='Pages':
        EmitIntermediate(value.userName, (2, value));

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);

Relying entirely on the MR system to do the hashing

Hash 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(h(value.name), (1, value));
    else // value.relation='Pages':
        EmitIntermediate(h(value.userName), (2, value));

Controlling the hash function

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
      if v1.name=v2.user: Emit(v1,v2);

CSE544 - S

Broadcast Join in MR

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

Pages

Users

Broadcast Join in MR

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

Pages

Users

Broadcast Join in MR

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

Pages

Users

Map 1

Map 2

Broadcast Join in MR

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

No need to copy Pages

Broadcast Users

Pages

Users

Map 1

Pages block 1

Users

Map 2

Pages block 2

Users

Broadcast Join in MR

Write the Map and Reduce functions (in class):

Matrix Multiplication v.s. Join

Dense matrices:

$$\begin{bmatrix} 6 & 6 & 0 \\ 1 & 0 & 0 \\ 2 & 0 & 6 \end{bmatrix} = \begin{bmatrix} 0 & 3 & 3 \\ 1 & 0 & 0 \\ 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 3 \\ 0 & 2 & 0 \\ 2 & 0 & 0 \end{bmatrix}$$

forall i,k do
    $C[i,k] = \Sigma_j \ A[i,j] * B[j,k]$

Matrix Multiplication v.s. Join

Dense matrices:

Sparse matrices as relations:

$$\begin{bmatrix} 6 & 6 & 0 \\ 1 & 0 & 0 \\ 2 & 0 & 6 \end{bmatrix} = \begin{bmatrix} 0 & 3 & 3 \\ 1 & 0 & 0 \\ 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 3 \\ 0 & 2 & 0 \\ 2 & 0 & 0 \end{bmatrix}$$

forall i,k do
    C[i,k] = Σ_j A[i,j] * B[j,k]

B(j,k,v)

j	k	v
1	1	1
1	3	3
2	2	1
3	1	2

A(i,j,v)

i	j	v
1	2	3
1	3	3
2	1	1
3	1	2

SELECT A.i, B.k, sum(A.v*B.v)
FROM A, B
WHERE A.j=B.j
GROUP BY A.i,B.i

Matrix Multiplication v.s. Join

Dense matrices:

Sparse matrices as relations:

$$\begin{bmatrix} 6 & 6 & 0 \\ 1 & 0 & 0 \\ 2 & 0 & 6 \end{bmatrix} = \begin{bmatrix} 0 & 3 & 3 \\ 1 & 0 & 0 \\ 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 3 \\ 0 & 2 & 0 \\ 2 & 0 & 0 \end{bmatrix}$$

B(j,k,v)

j	k	v
1	1	1
1	3	3
2	2	1
3	1	2

A(i,j,v)

i	j	v
1	2	3
1	3	3
2	1	1
3	1	2

forall i,k do
    C[i,k] = Σ$_j$ A[i,j] * B[j,k]

SELECT A.i, B.k, sum(A.v*B.v)
FROM A, B
WHERE A.j=B.j
GROUP BY A.i.B.i

Matrix multiplication = a join + a group by