Introduction to Data Management
CSE 344

Lecture 24: MapReduce

Announcements

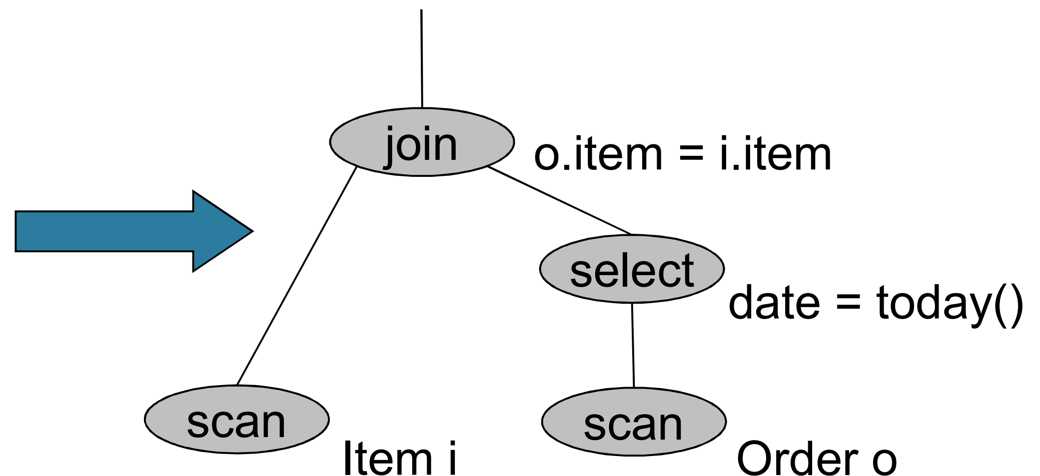
- HW7 due yesterday
  - With 2 late days due tomorrow
- Final next Friday (in class)
  - 60 min (will give 75). Shorter than examples.
  - Monday will be last lecture covered by exam.
- Evaluation: https://uw.iasystem.org/survey/179899

Order(oid, item, date), Line(item, …)

Putting it Together: Example Parallel Query Plan

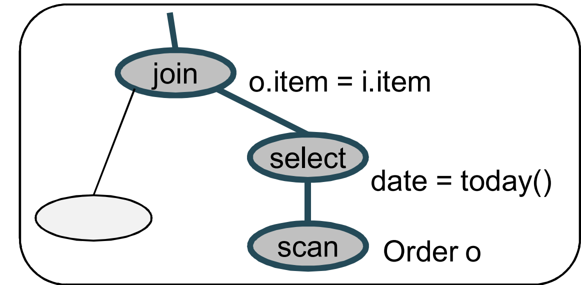
Find all orders from today, along with the items ordered

```
SELECT *
  FROM Order o, Line i
 WHERE o.item = i.item
   AND o.date = today()
```



Order(oid, item, date), Line(item, …)

Example Parallel Query Plan



Node 1 Node 2 Node 3

hash h(o.item)
select date=today()
scan Order o

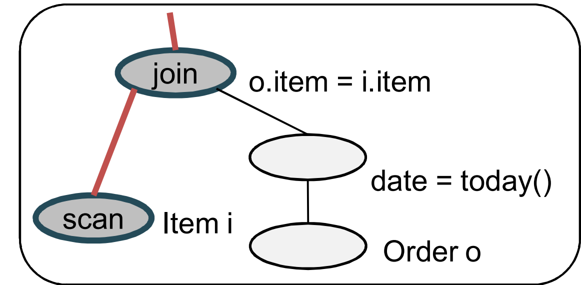
hash h(o.item)
select date=today()
scan Order o

hash h(o.item)
select date=today()
scan Order o

Node 1 Node 2 Node 3

Order(oid, item, date), Line(item, …)

Example Parallel Query Plan



Node 1 Node 2 Node 3

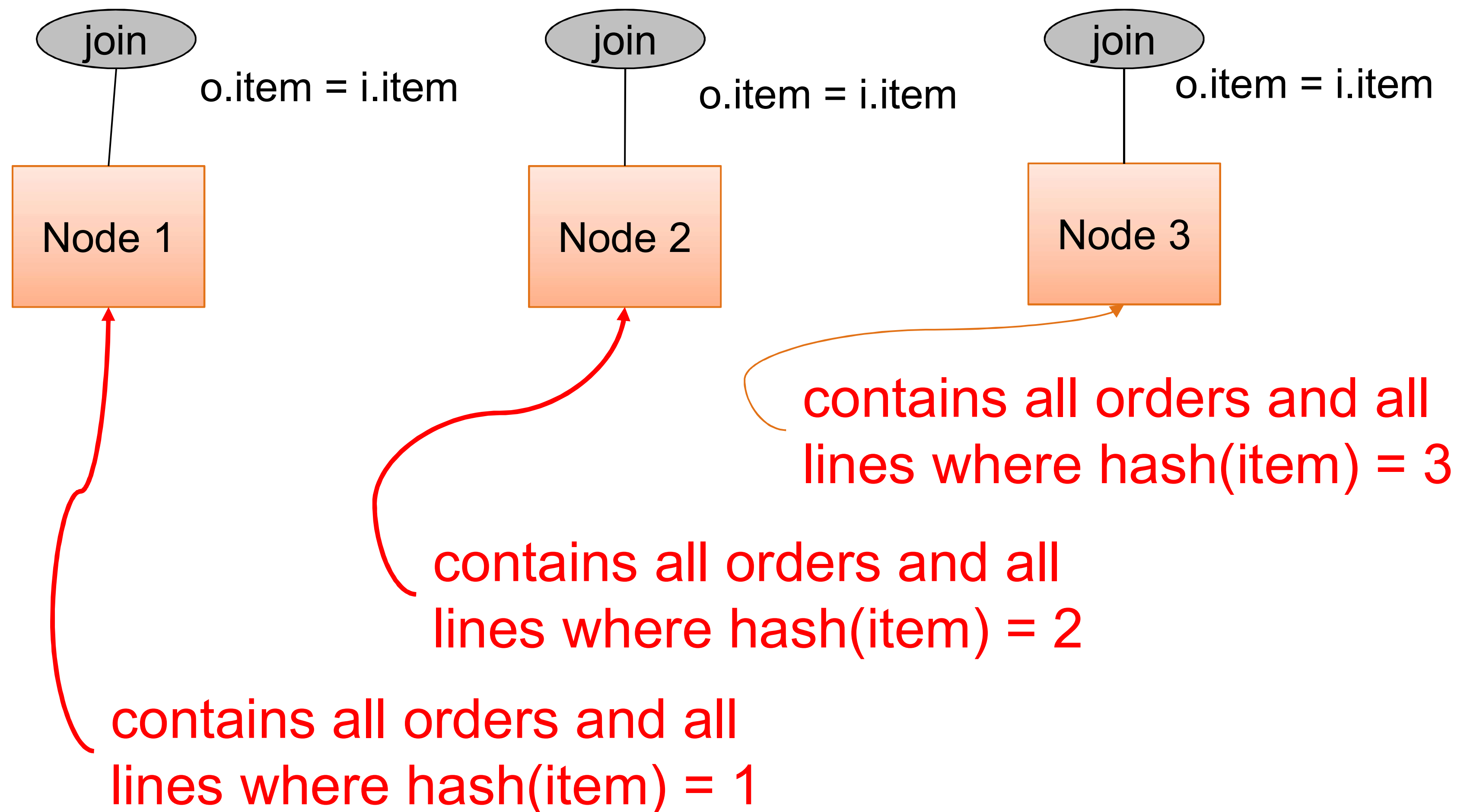
hash h(i.item) hash h(i.item) hash h(i.item)

scan Item i scan Item i scan Item i

Node 1 Node 2 Node 3

Order(oid, item, date), Line(item, …)

Example Parallel Query Plan



What's wrong with the relational data model?

- From last class - NoSQL

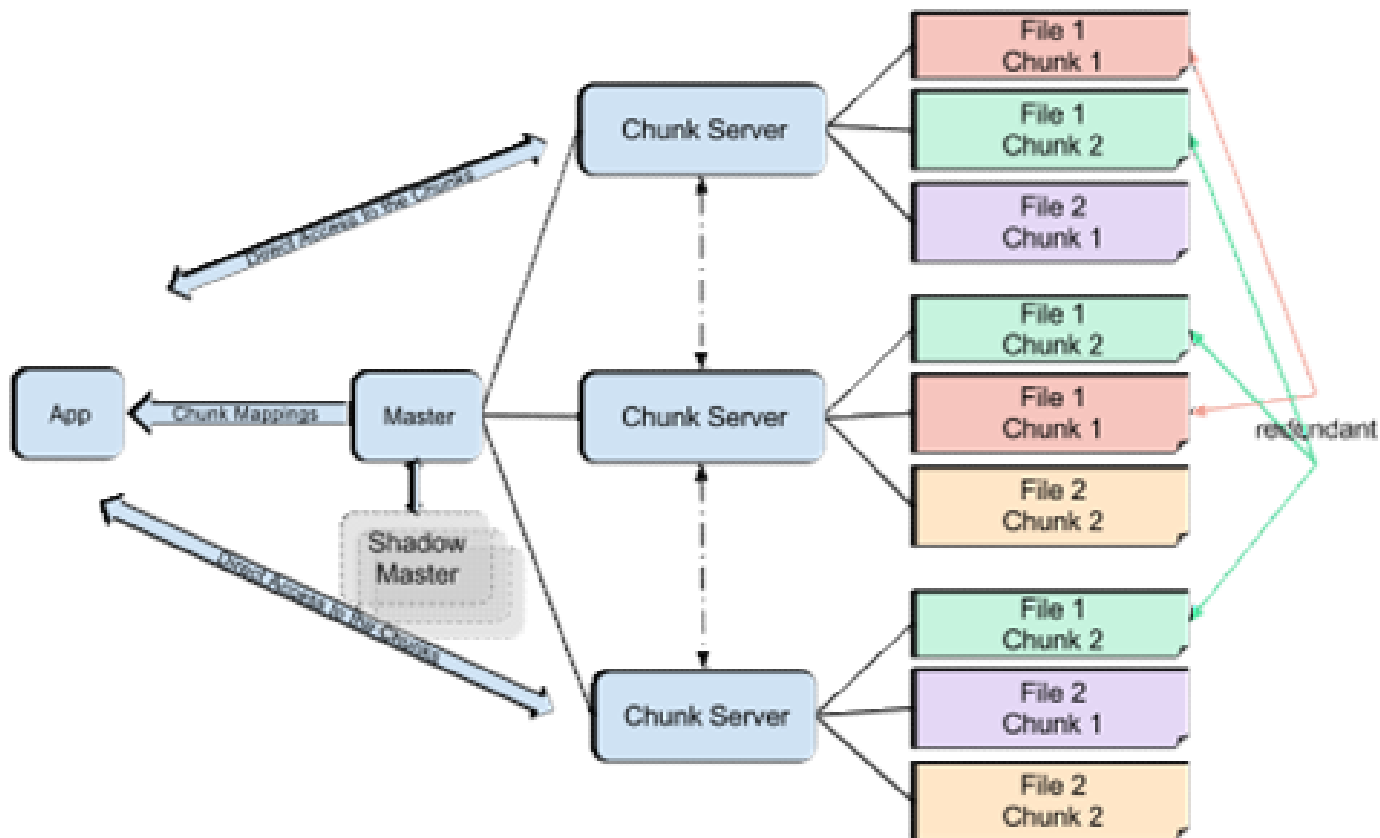
  ↳ nice not to have a schema

- For parallel data processing:
  - Want to control both data distribution and query processing
  - Want simpler programming model
    - "I don't want to learn SQL!" (non 344 student)
  - Fault tolerance is important

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

Distributed File System (DFS)



The Problem

- Want to calculate in link counts for the entire web.

- What you have:
  - 30 billion webpages stored in GFS
  - In 100MB chunks on 10,000 nodes

- What you want:
  - List of pairs url : count (in bound link count)

The Solution

- Step 1
  - For each html document create keys {url -> count}
  - Distributed across all GFS nodes
- Step 2
  - Partition keys on url over cluster.
- Step 3
  - On each node sum up the total count of inbound links for each URL.

This is Map Reduce

Map Reduce Data Model

Started by Google in 2004

Instance: Files containing (key, value) pairs

Schema: None!

- just like other key-value data models

Query language: a MapReduce program:

- Input: a bag of (key, value) pairs
- Output: a bag of (key, value) pairs
- Implementation in Java (Hadoop), Python, Go, …

Lifecycle of a MR Program

1. Read a lot of data and parse into (key, value) pairs
2. Map: extract something you care about from each (key, value) pair
3. Shuffle output from mappers
   - done internally by implementation
4. Reduce: aggregate, summarize, filter, transform
5. Write the results to files

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

Step 2: the MAP Phase

User provides the MAP-function:

- Input: **(key, value)**
- Output: bag of **(intermediate key, value)**

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

Step 3: the shuffle phase

- System groups all pairs generated by MAPpers with the same intermediate key
- Passes the bag of values to the REDUCE function in next stage
- Example: given map output:
  (“a”, 1), (“b”, 1), (“a”, 2), (“c”, 1), (“c”, 5)

  Shuffle produces the output:
  (“a”, [1,2]), (“b”, [1]), (“c”, [1,5]) and partitions
- This is just another (key, value) pair!

Step 4: the REDUCE Phase

User provides the REDUCE function:

- Input: **(intermediate key, bag of values)**
- Output: bag of output **(values)**

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 "1"s
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

Map Reduce Data Model

Key Points:

Instance: Files containing (key, value) pairs

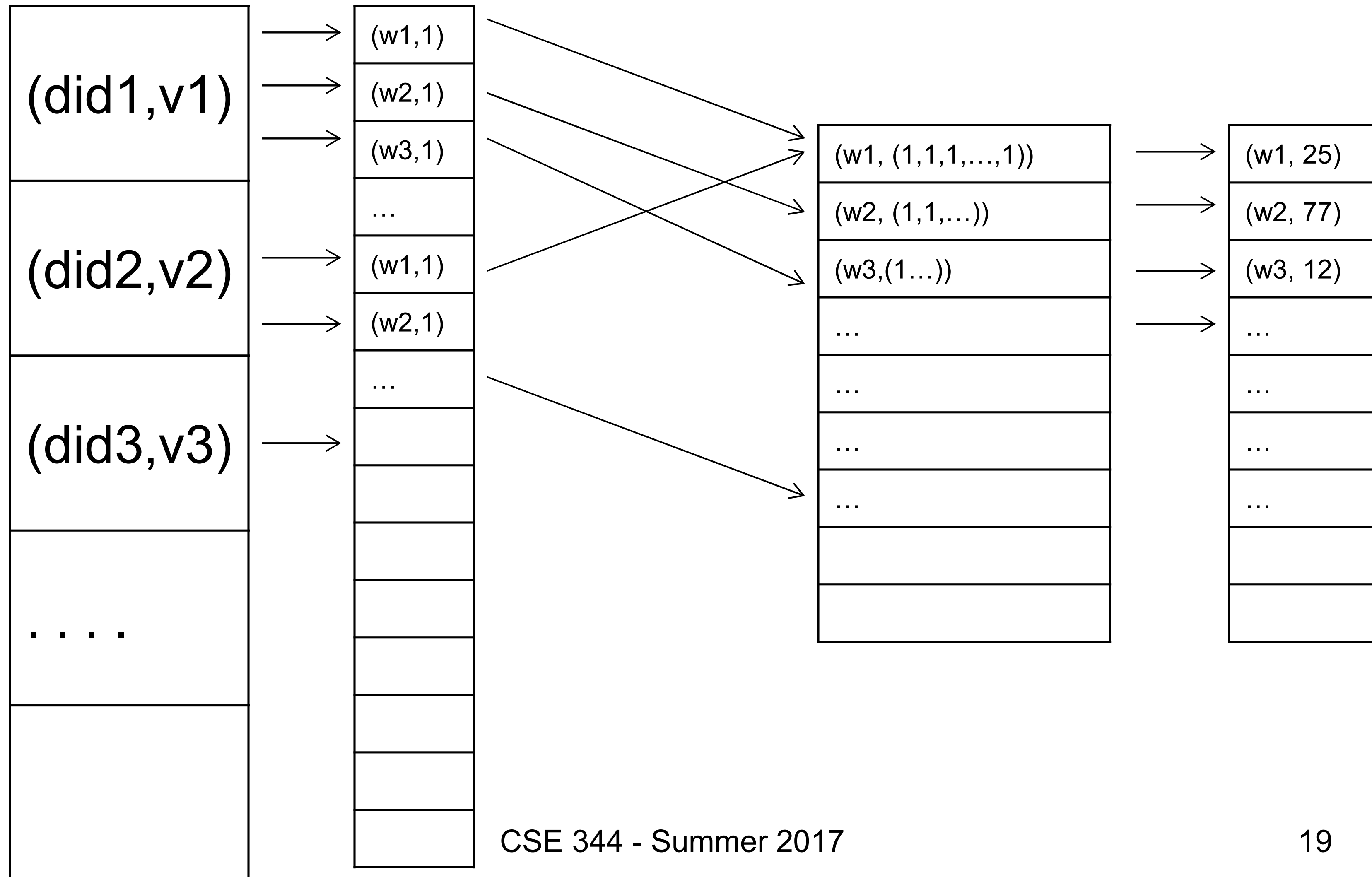
Schema: None!

- just like other key-value data models

Query language: a MapReduce program (No SQL):

- Input: a bag of (key, value) pairs
- Output: a bag of (key, value) pairs
- Implementation in Java (Hadoop), Python, Go, …

Illustration

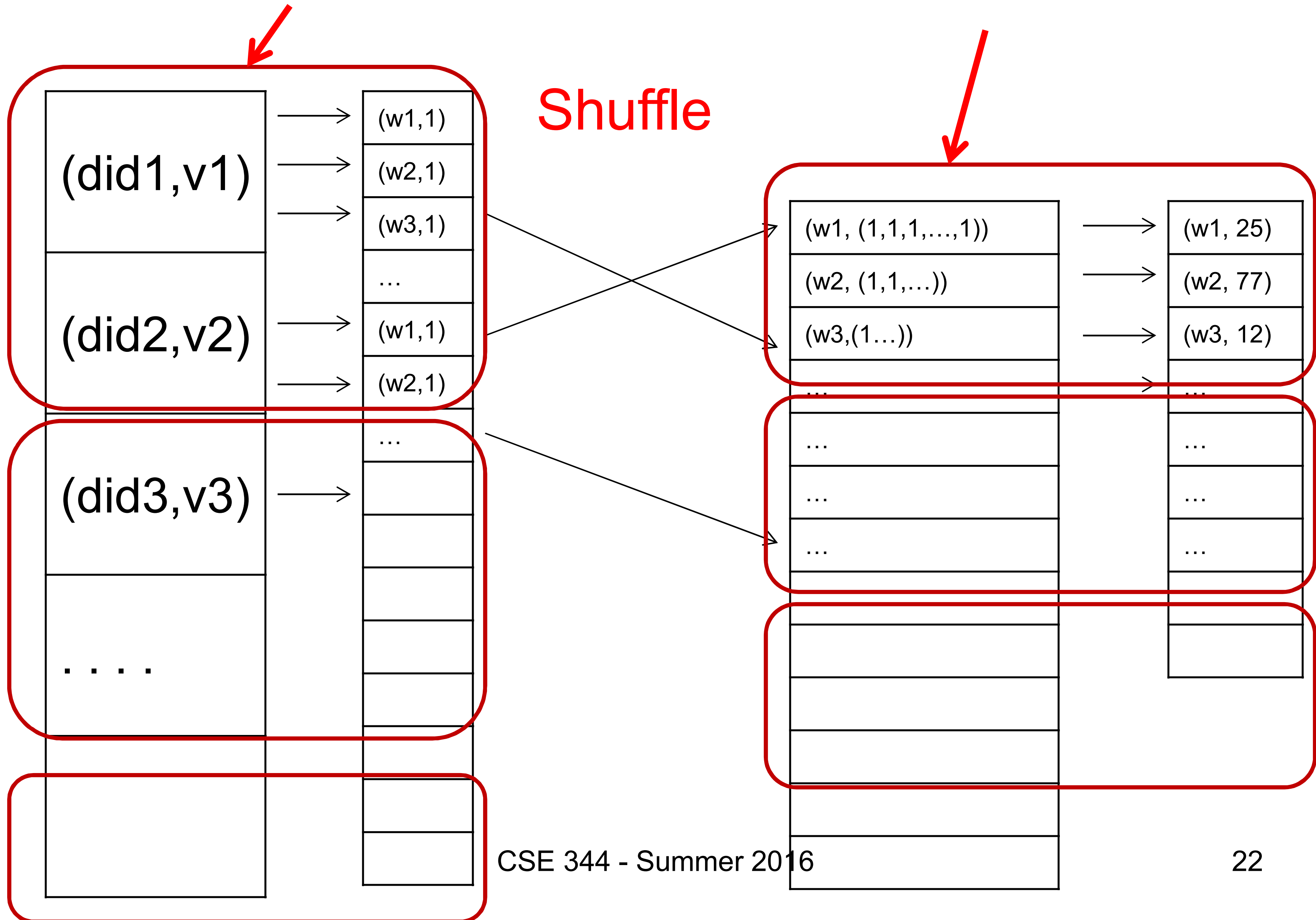


Jobs v.s. Tasks

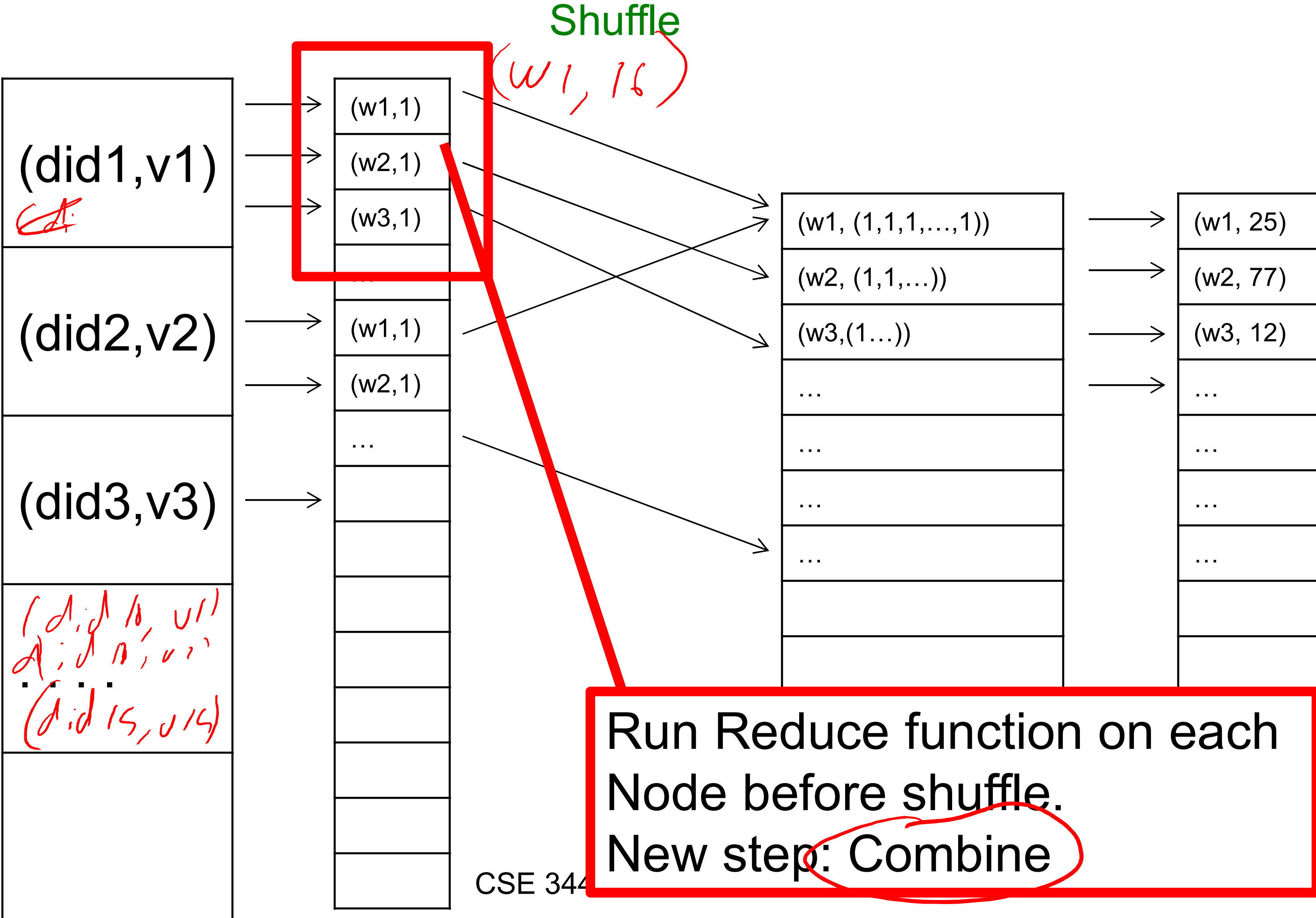
- A MapReduce Job
  - One single "query," e.g., count the words in all docs
  - More complex queries may consists of multiple jobs
- A Map Task, or a Reduce Task
  - A group of instantiations of the map-, or reduce-function, which are scheduled on a single worker

Workers

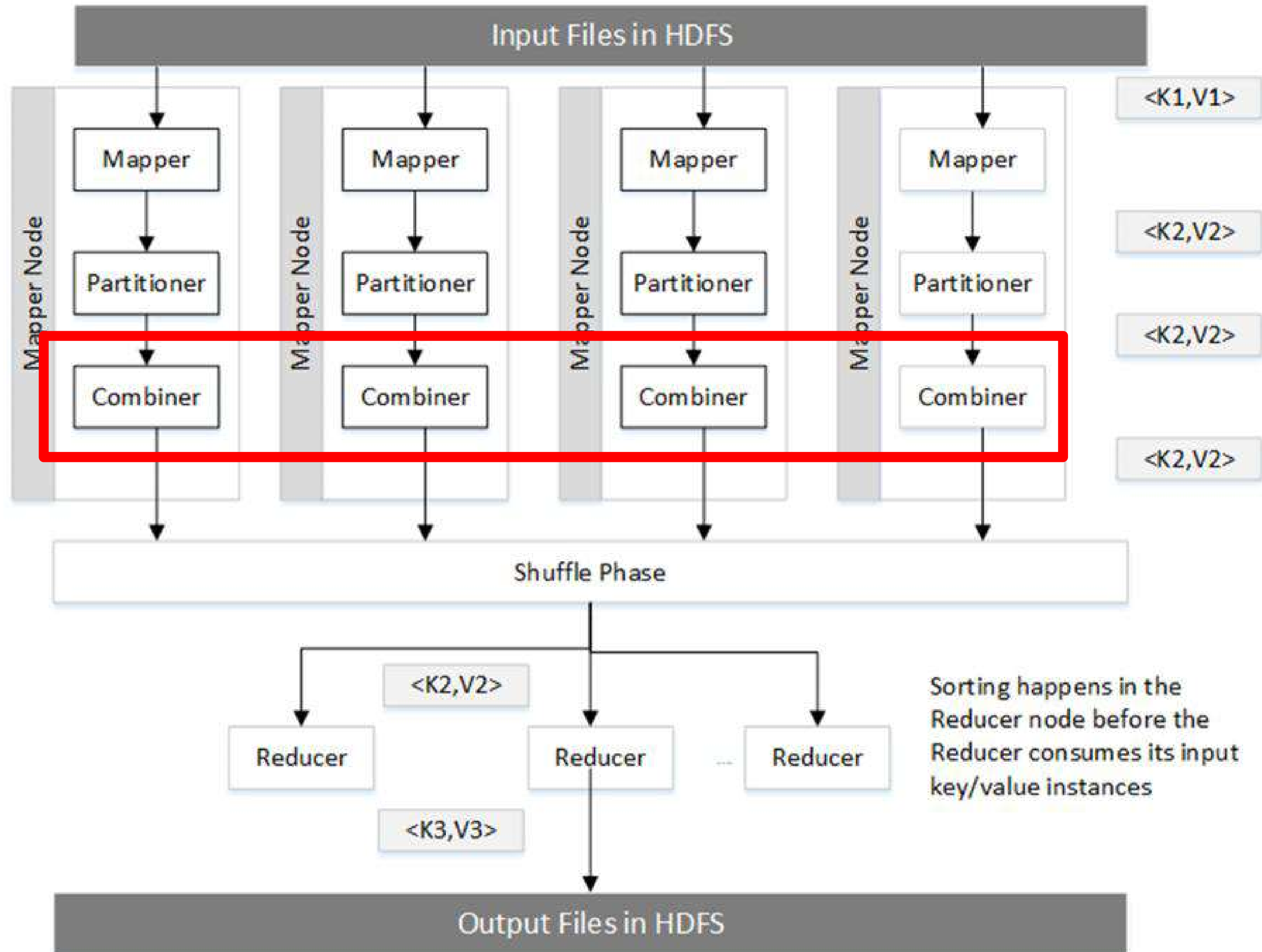
- A worker is a process that executes one task at a time
- Typically there is one worker per processor, hence 4 or 8 per node


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)
…
…
…
…

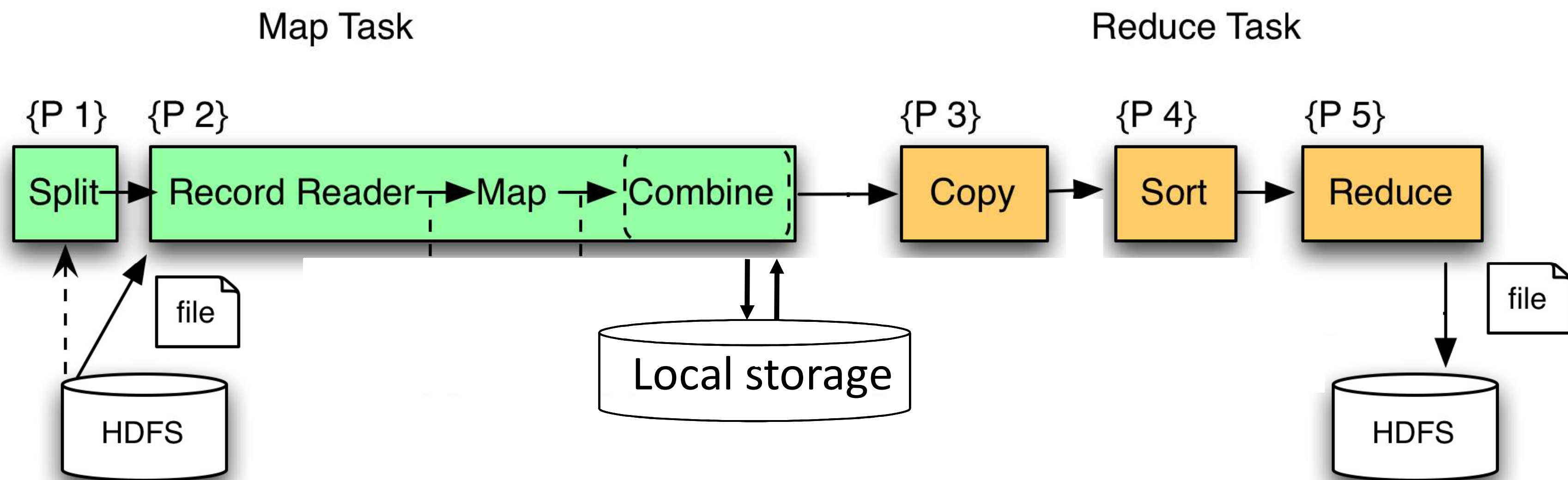
How can we optimize the shuffle?



How can we optimize the shuffle?



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

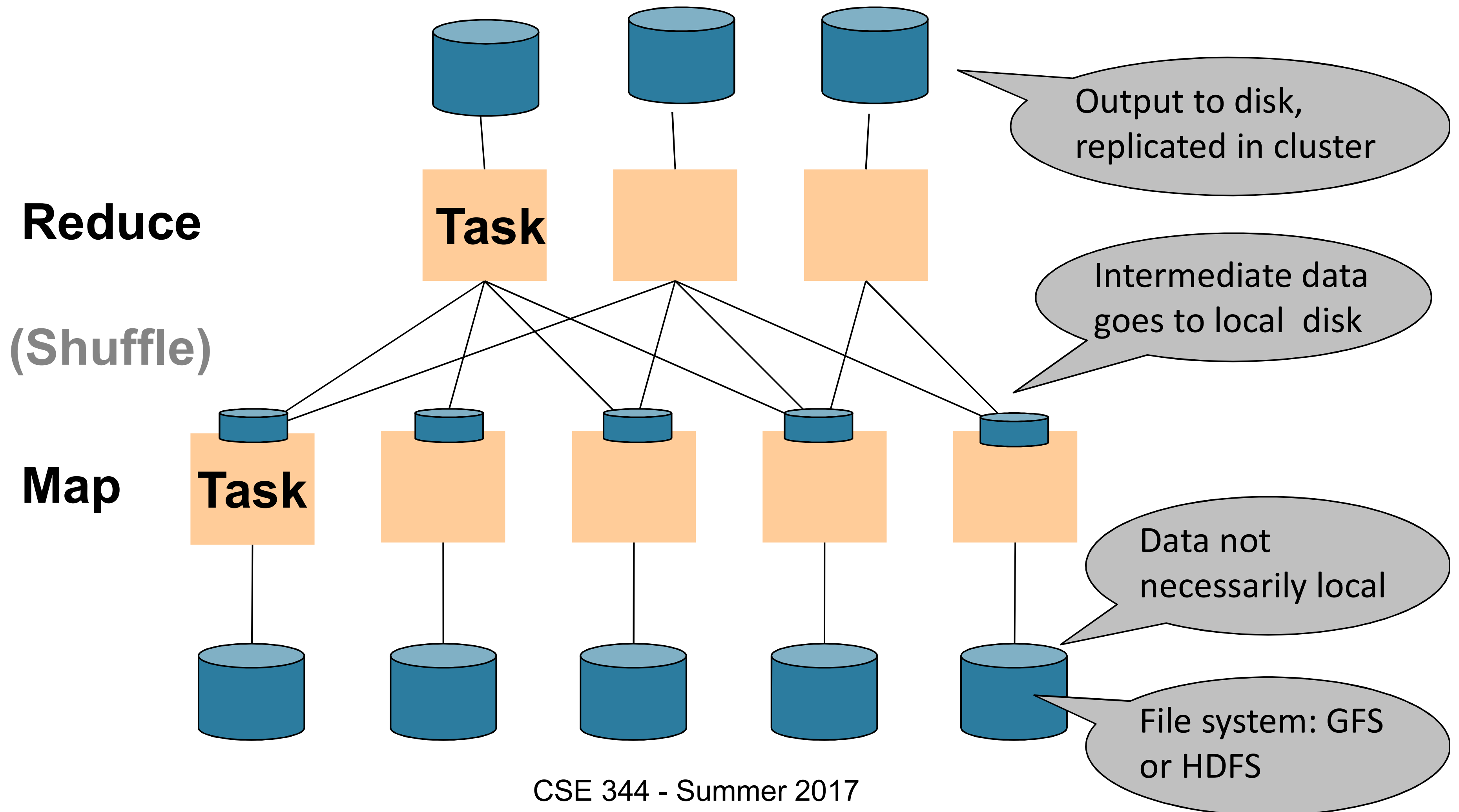
Worker failure:

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

Fault Tolerance

- If one server fails once every year, how long before a job running on 10,000 servers fails
  - 1h 10 min! (10,000 / (365 * 24))
- 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

MapReduce Execution Details



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*

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: $R \bowtie S$

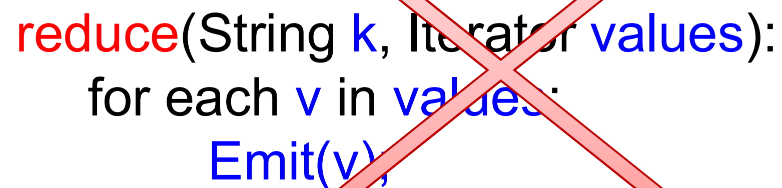
Selection $\sigma_{A=42}(R)$

```
map(String relationName, Tuple t):
  if t.A == 42:
    EmitIntermediate(relationName, t);
```

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

Selection $\sigma_{A=42}(R)$

```
map(String relationName, Tuple t):
  if t.A == 42:
    EmitIntermediate(relationName, t);
```



- Reduce isn't really needed
- But MR requires reduce functions

Group By $\gamma_{A,sum(B)}(R)$

```
map(String relationName, Tuple t):
    EmitIntermediate(t.A, t.B);
```

t.A

[list of B's]

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

s

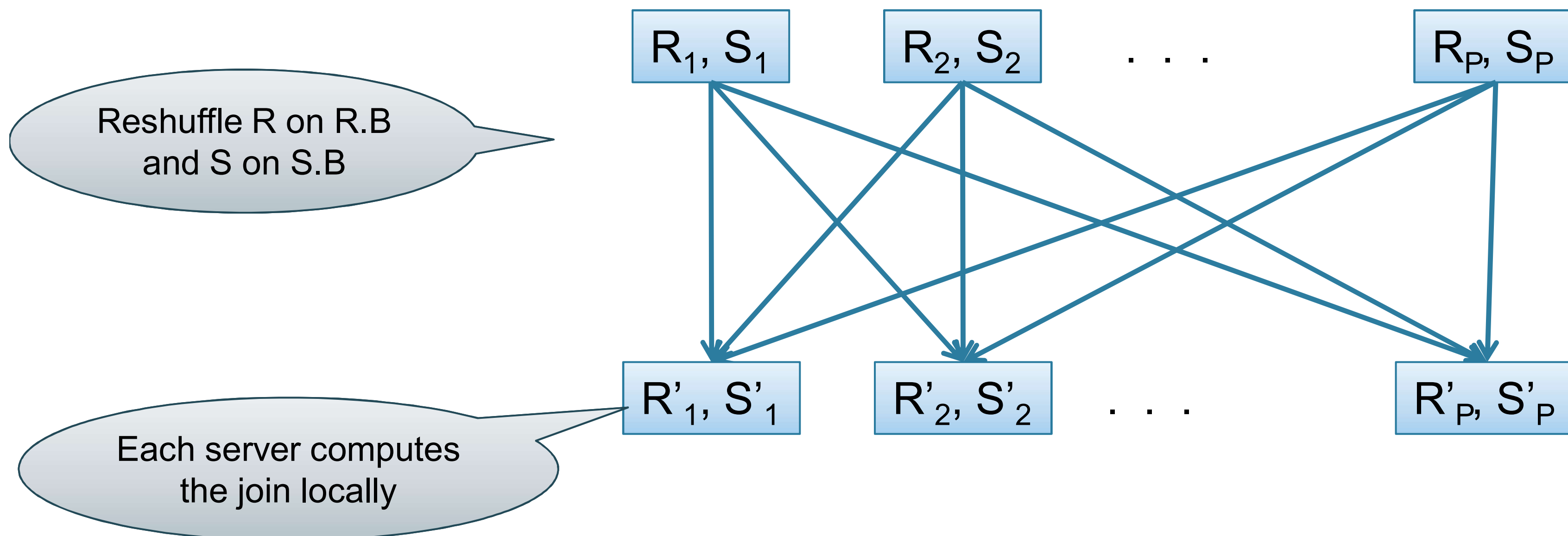
Implementing Join in MR

Two parallel join algorithms that we have seen:

- Partitioned hash-join
- Broadcast join

Parallel Execution of RA Operators: Partitioned Hash-Join

- Data: R(<u>K1</u>,A, B), S(<u>K2</u>, B, C)
- Query: R(<u>K1</u>,A,B) ⋈ S(<u>K2</u>,B,C)
  - Initially, both R and S are partitioned on K1 and K2



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

Partitioned Hash-Join in MR

```
map(String relationName, Tuple t):
  switch (relationName):
    case ‘R’: EmitIntermediate(t.B, IntKey(‘R’, value));
    case ‘S’: EmitIntermediate(t.C, IntKey(‘S’, value));
```

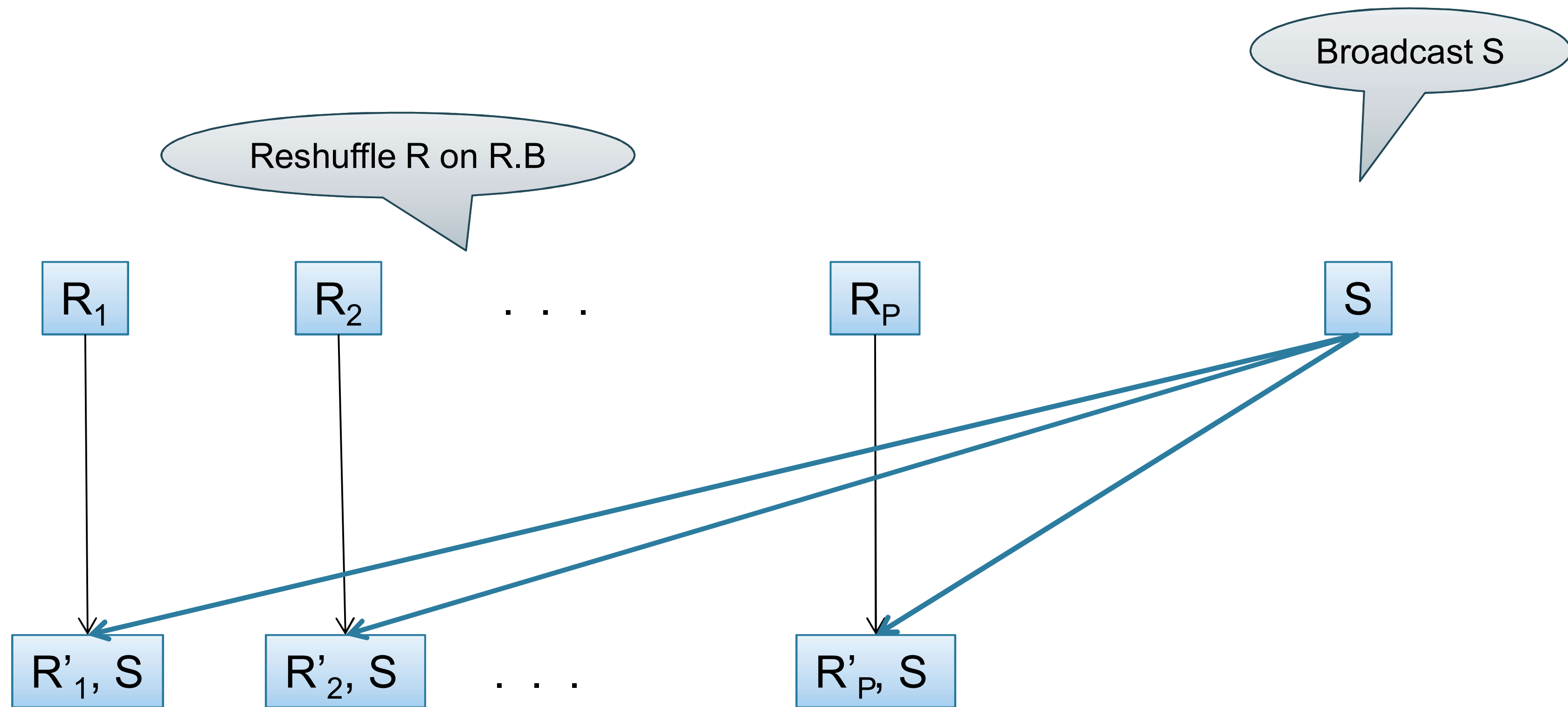
Or call hash(t.B)

```
reduce(String k, Iterator values):
  R = [];  S = [];
  for each v in values:
    switch (v.relationName):
      ‘R’: R.insert(v);
      ‘S’: S.insert(v);
  for r in R, for s in S
      Emit(Tuple(r,s));
```

All tuples here must join

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

Broadcast Join



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

Broadcast Join in MR

```
map(Tuple [] rs):
  S = readFromNetwork();
  ht = new Hashtable()
  for each w in S:
    ht.insert(w.C, w)

  for each r in ts:
    for each s in ht.find(r.B):
      Emit(Tuple(r,s));
```

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

Read entire table S, build a Hash Table

Broadcast

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

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

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.

- Next time: Spark replaces this with "Resilient Distributed Datasets" = main memory + lineage