

Introduction to Data Management

CSE 344

Lecture 26: Parallel Databases and MapReduce

HW8

- MapReduce (Hadoop) w/ declarative language (Pig)
- Cluster will run in Amazon's cloud (AWS)
 - Give your credit card
 - Click, click, click... and you have a MapReduce cluster
- We will analyze a real 0.5TB graph
- Processing the entire data takes hours
 - Required problems: queries on a subset only
 - Extra credit problem #4: entire data
- Tomorrow's Quiz Sections
 - Step by step instructions on how to connect to AWS
 - Don't miss!

Amazon Warning

- “We **HIGHLY** recommend you remind students to turn off any instances after each class/session – as this can quickly diminish the credits and start charging the card on file. **You are responsible for the overages.**”
- “AWS customers can now use **billing alerts** to help monitor the charges on their AWS bill. You can get started today by visiting your [Account Activity page](#) to enable monitoring of your charges. Then, you can set up a billing alert by simply specifying a bill threshold and an e-mail address to be notified as soon as your estimated charges reach the threshold.”

Outline

- Today: Query Processing in Parallel DBs
- Next Lecture: Parallel Data Processing at Massive Scale (MapReduce)
 - Reading assignment:
Chapter 2 (Sections 1,2,3 only) of Mining of Massive Datasets, by Rajaraman and Ullman
<http://i.stanford.edu/~ullman/mmds.html>

Review

- Why parallel processing?
- What are the possible architectures for a parallel database system?
- What are speedup and scaleup?

Basic Query Processing: Quick Review in Class

Basic query processing **on one node**.

Given relations $R(A,B)$ and $S(B, C)$, **no indexes**, how do we compute:

- **Selection:** $\sigma_{A=123}(R)$
- **Group-by:** $\gamma_{A, \text{sum}(B)}(R)$
- **Join:** $R \bowtie S$

Basic Query Processing: Quick Review in Class

Basic query processing **on one node**.

Given relations $R(A,B)$ and $S(B, C)$, **no indexes**, how do we compute:

- **Selection:** $\sigma_{A=123}(R)$
 - Scan file R, select records with $A=123$
- **Group-by:** $\gamma_{A, \text{sum}(B)}(R)$
 - Scan file R, insert into a hash table using attr. A as key
 - When a new key is equal to an existing one, add B to the value
- **Join:** $R \bowtie S$
 - Scan file S, insert into a hash table using attr. B as key
 - Scan file R, probe the hash table using attr. B

Parallel Query Processing

How do we **compute** these operations on a shared-nothing parallel db?

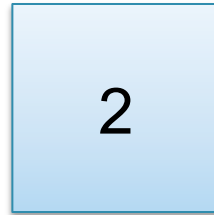
- **Selection:** $\sigma_{A=123}(R)$ (that's easy, won't discuss...)
- **Group-by:** $\gamma_{A, \text{sum}(B)}(R)$
- **Join:** $R \bowtie S$

Before we answer that: how do we **store** R (and S) on a shared-nothing parallel db?

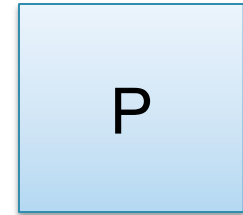
Horizontal Data Partitioning

Data:

Servers:



...

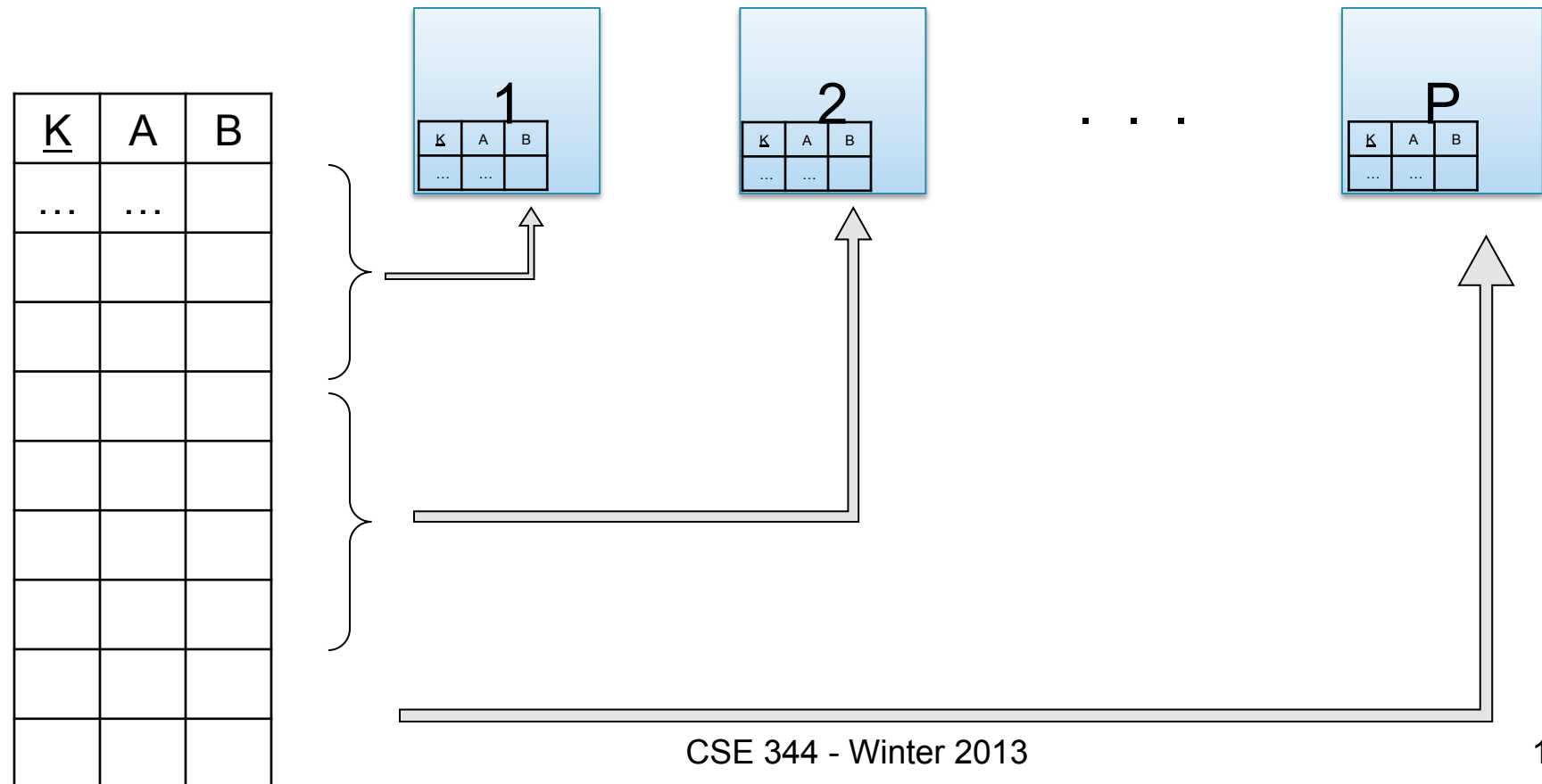


<u>K</u>	A	B
...	...	

Horizontal Data Partitioning

Data:

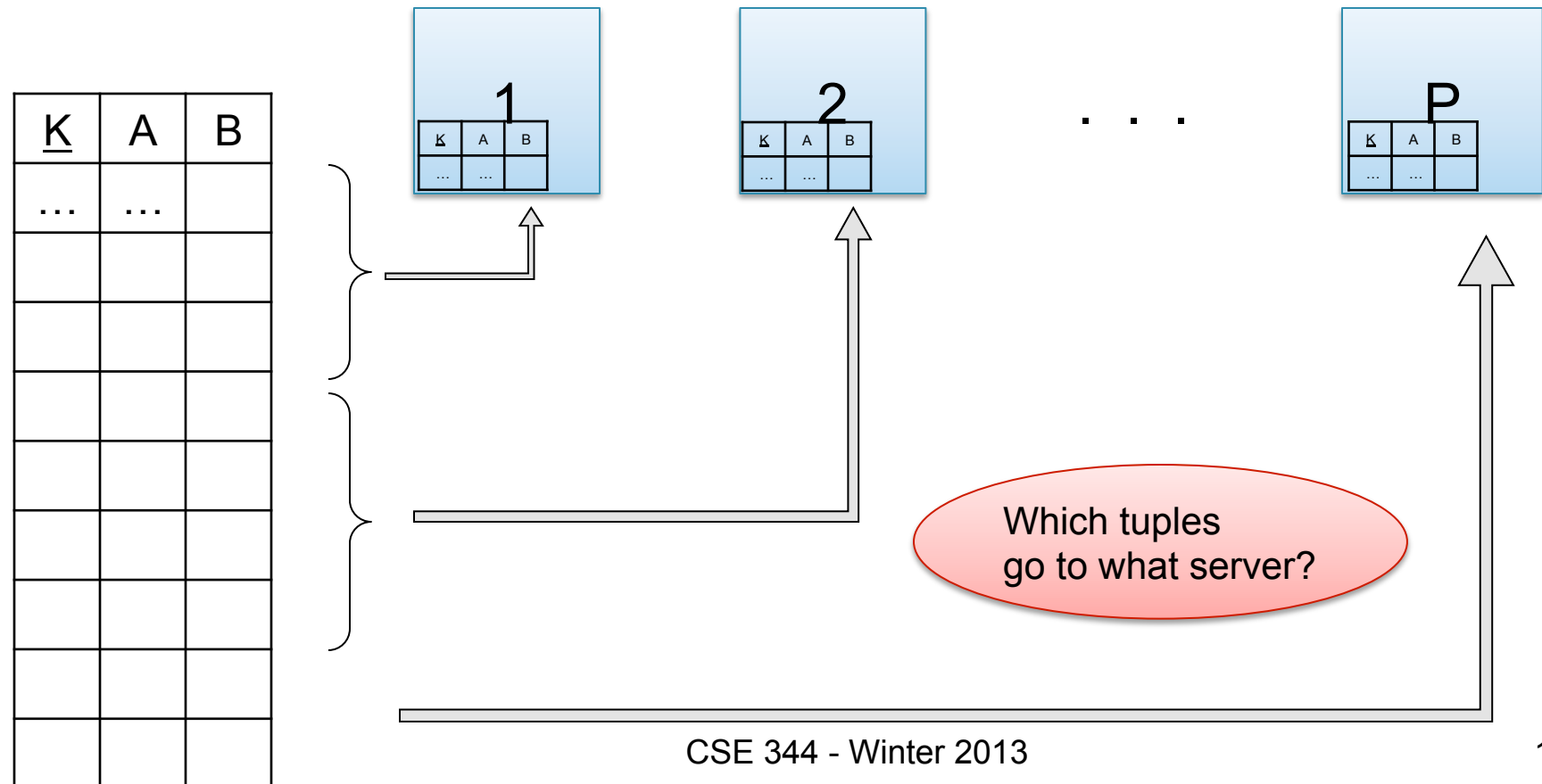
Servers:



Horizontal Data Partitioning

Data:

Servers:



Horizontal Data Partitioning

- **Block Partition:**
 - Partition tuples arbitrarily s.t. $\text{size}(R_1) \approx \dots \approx \text{size}(R_P)$
- **Hash partitioned on attribute A:**
 - Tuple t goes to chunk i , where $i = h(t.A) \bmod P + 1$
- **Range partitioned on attribute A:**
 - Partition the range of A into $-\infty = v_0 < v_1 < \dots < v_P = \infty$
 - Tuple t goes to chunk i , if $v_{i-1} < t.A < v_i$

Parallel GroupBy

Data: $R(\underline{K}, A, B, C)$

Query: $\gamma_{A, \text{sum}(C)}(R)$

Discuss in class how to compute in each case:

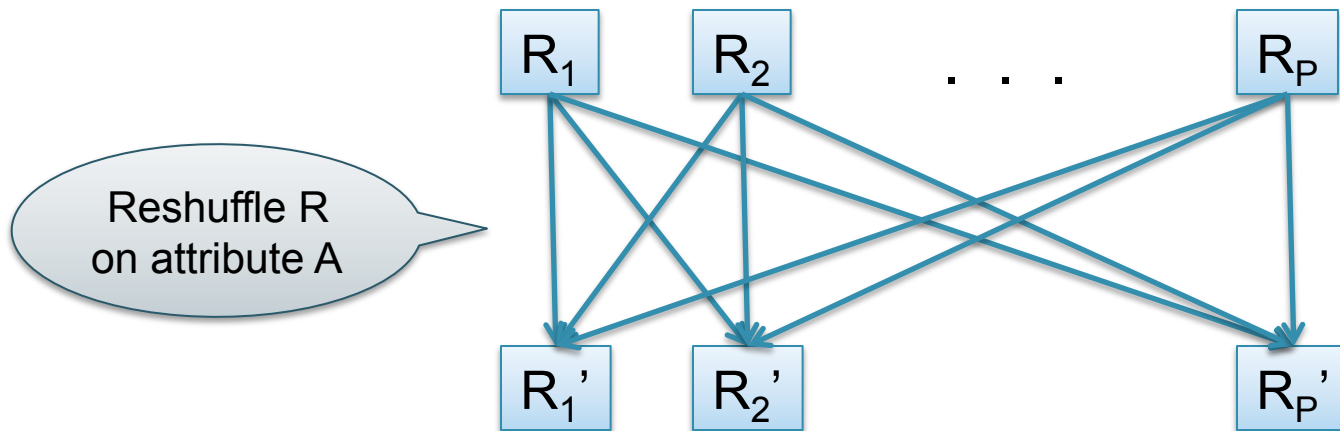
- R is hash-partitioned on A
- R is block-partitioned
- R is hash-partitioned on K

Parallel GroupBy

Data: $R(\underline{K}, A, B, C)$

Query: $\gamma_{A, \text{sum}(C)}(R)$

- R is block-partitioned or hash-partitioned on K



Parallel Join

- **Data:** $R(\underline{K1}, A, B)$, $S(\underline{K2}, B, C)$
- **Query:** $R(\underline{K1}, A, B) \bowtie S(\underline{K2}, B, C)$

Initially, both R and S are horizontally partitioned on K1 and K2

R_1, S_1

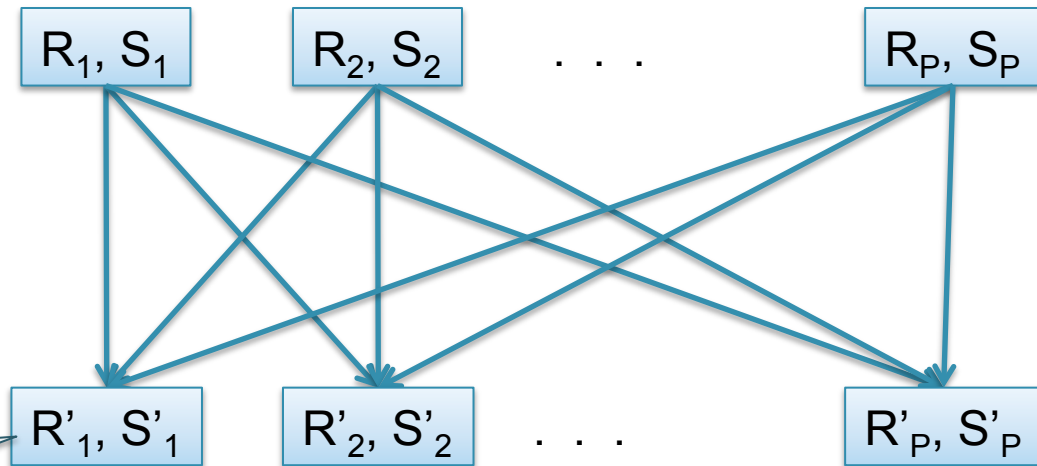
R_2, S_2

R_P, S_P

Parallel Join

- **Data:** $R(\underline{K1}, A, B)$, $S(\underline{K2}, B, C)$
- **Query:** $R(\underline{K1}, A, B) \bowtie S(\underline{K2}, B, C)$

Initially, both R and S are horizontally partitioned on K1 and K2



Reshuffle R on R.B
and S on S.B

Each server computes
the join locally

Speedup and Scaleup

- Consider:
 - Query: $Y_{A,\text{sum}(C)}(R)$
 - Runtime: dominated by reading chunks from disk
- If we double the number of nodes P , what is the new running time?
- If we double both P and the size of R , what is the new running time?

Speedup and Scaleup

- Consider:
 - Query: $Y_{A, \text{sum}(C)}(R)$
 - Runtime: dominated by reading chunks from disk
- If we double the number of nodes P , what is the new running time?
 - Half (each server holds $\frac{1}{2}$ as many chunks)
- If we double both P and the size of R , what is the new running time?
 - Same (each server holds the same # of chunks)

Uniform Data v.s. Skewed Data

- Let $R(\underline{K}, A, B, C)$; which of the following partition methods may result in **skewed** partitions?
- **Block partition**
- **Hash-partition**
 - On the key K
 - On the attribute A

Uniform Data v.s. Skewed Data

- Let $R(\underline{K}, A, B, C)$; which of the following partition methods may result in **skewed** partitions?

- **Block partition**

Uniform

- **Hash-partition**

- On the key K
- On the attribute A

Uniform

May be skewed

Assuming good hash function

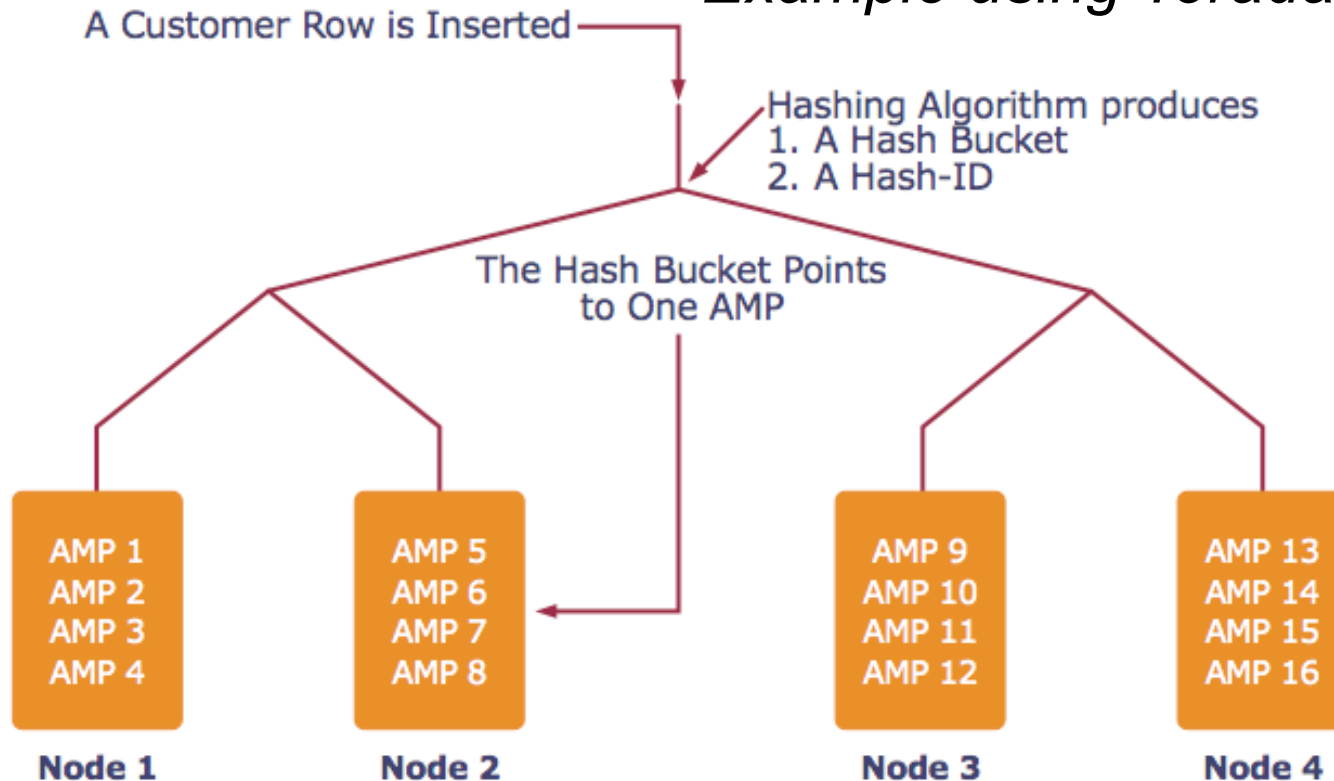
E.g. when all records have the same value of the attribute A , then all records end up in the same partition

Parallel DBMS

- Parallel query plan: tree of parallel operators
 - Intra-operator parallelism**
 - Data streams from one operator to the next
 - Typically all cluster nodes process all operators
- Can run multiple queries at the same time
 - Inter-query parallelism**
 - Queries will share the nodes in the cluster
- Notice that user does not need to know how his/her SQL query was processed

Loading Data into a Parallel DBMS

Example using Teradata System

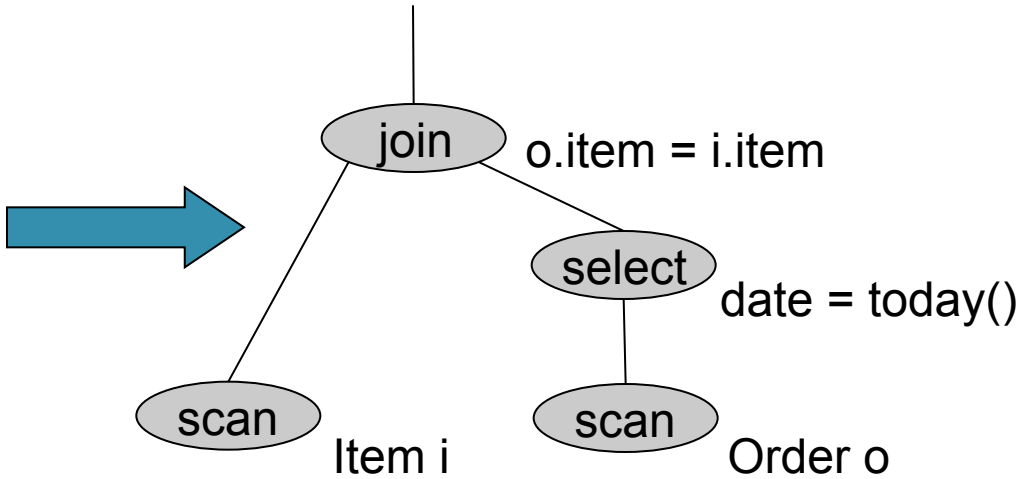


AMP = "Access Module Processor" = unit of parallelism

Example Parallel Query Execution

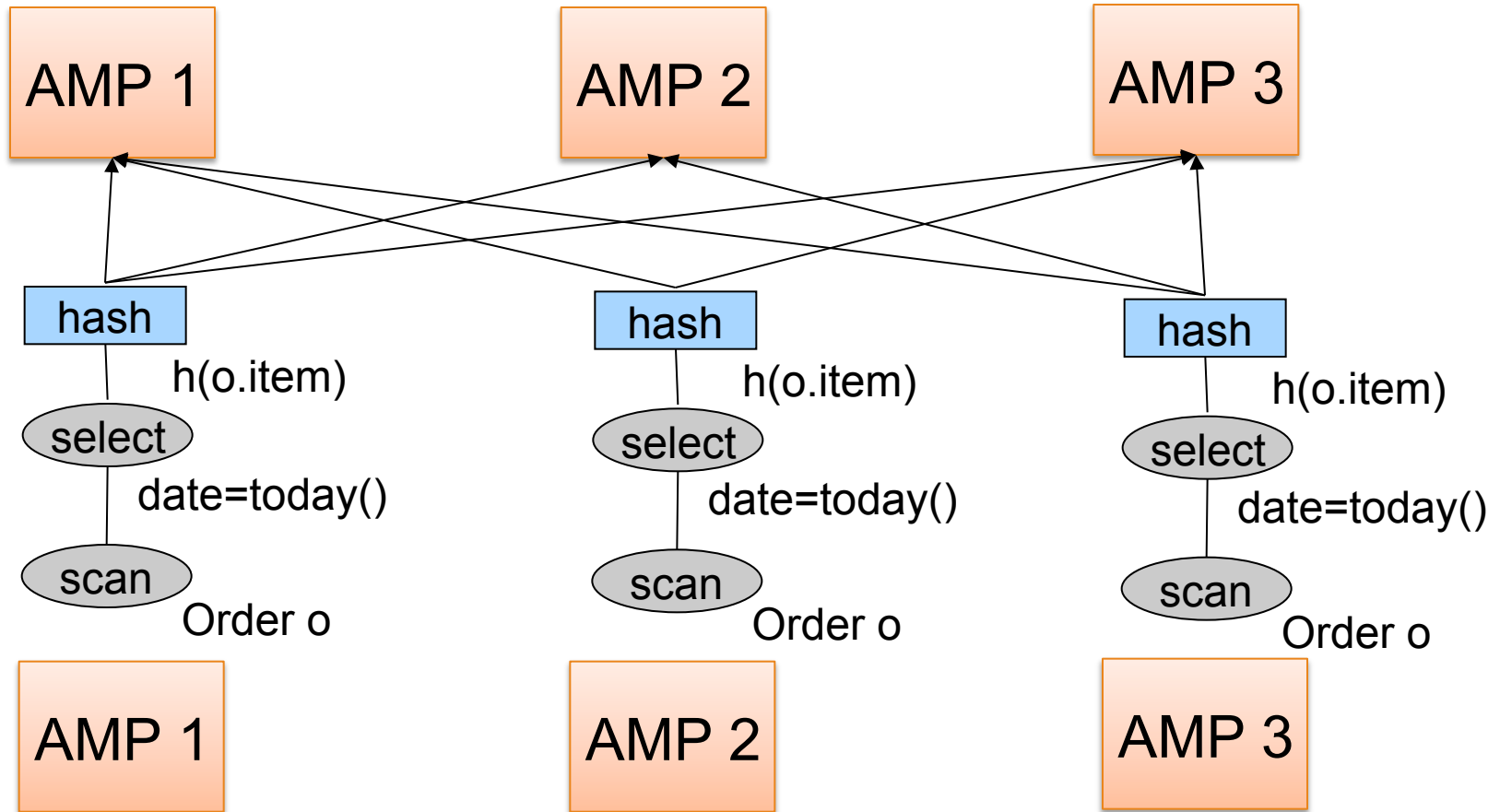
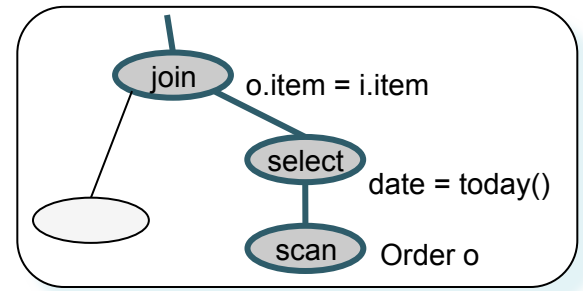
Find all orders from today, along with the items ordered

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

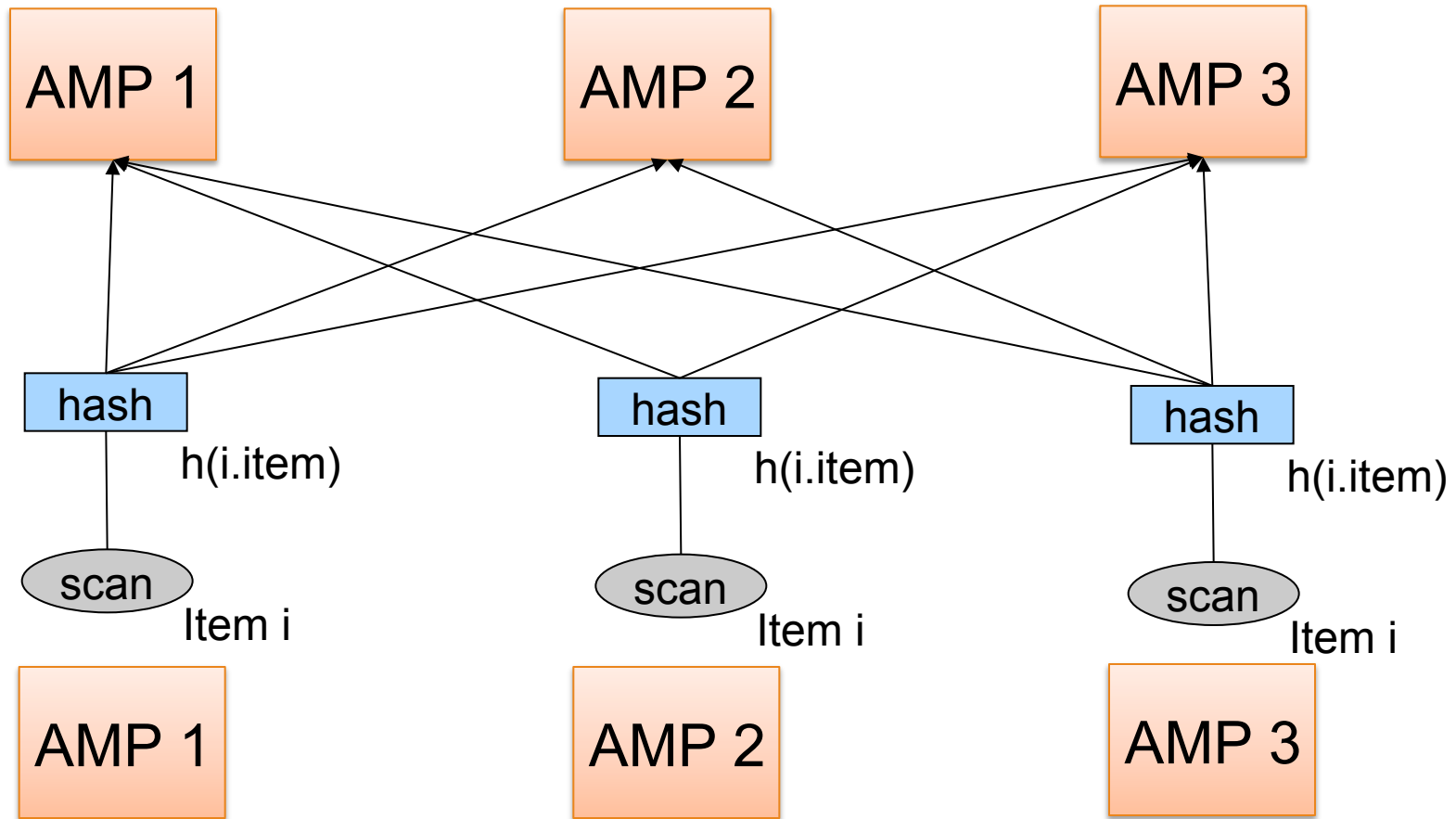
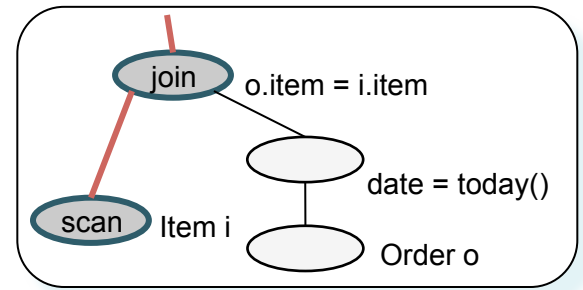


Order(oid, item, date), Line(item, ...)

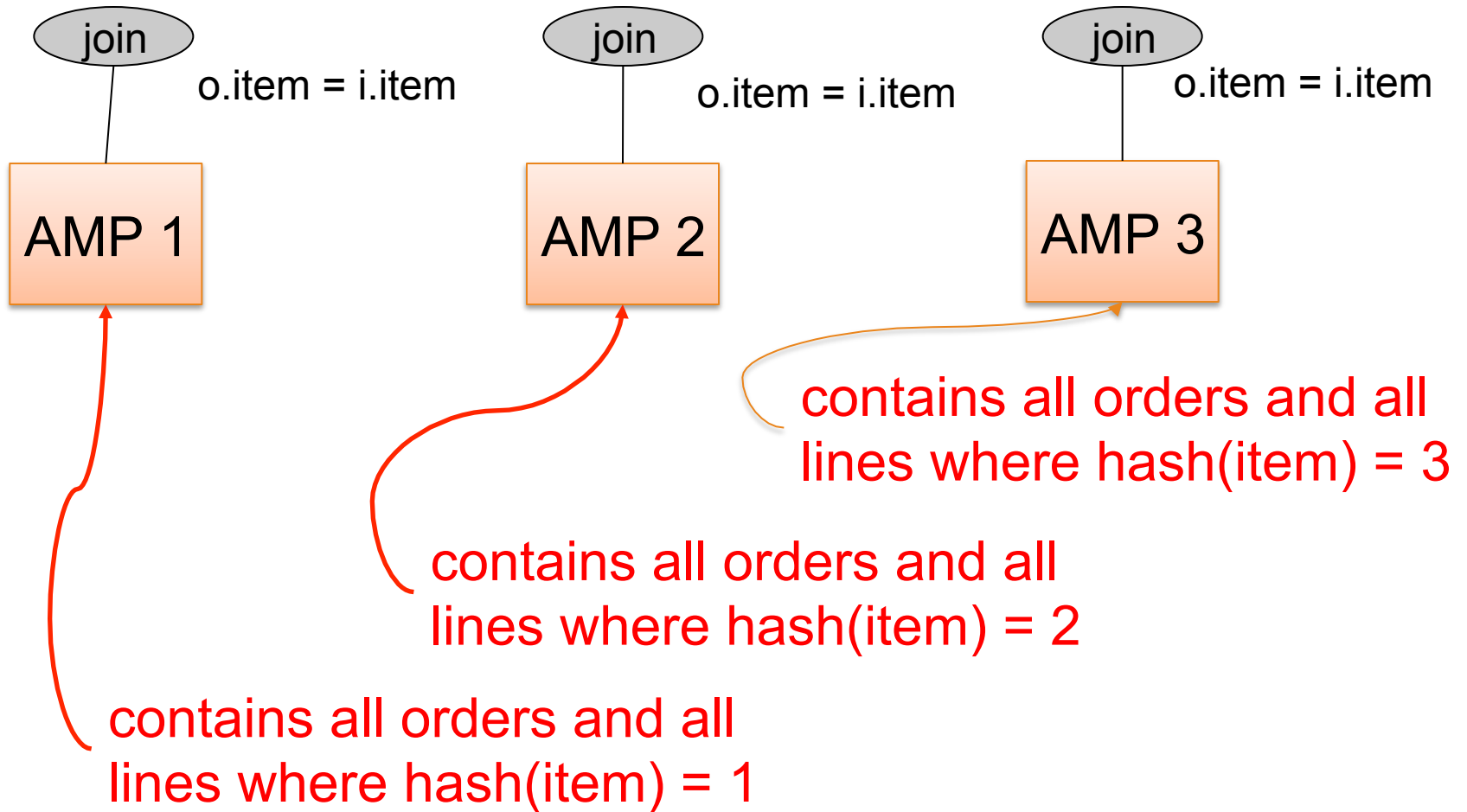
Example Parallel Query Execution



Example Parallel Query Execution



Example Parallel Query Execution



Parallel Dataflow Implementation

- Use relational operators unchanged
- Add a special *shuffle* operator
 - Handle data routing, buffering, and flow control
 - Inserted between consecutive operators in the query plan
 - Two components: ShuffleProducer and ShuffleConsumer
 - Producer pulls data from operator and sends to n consumers
 - Producer acts as driver for operators below it in query plan
 - Consumer buffers input data from n producers and makes it available to operator through getNext interface
- You will use this extensively in 444

Parallel Data Processing at Massive Scale

Data Centers Today

- Large number of commodity servers, connected by high speed, commodity network
- Rack: holds a small number of servers
- Data center: holds many racks

Data Processing at Massive Scale

- Want to process petabytes of data and more
- Massive parallelism:
 - 100s, or 1000s, or 10000s servers
 - Many hours
- Failure:
 - If medium-time-between-failure is 1 year
 - Then 10000 servers have one failure / hour

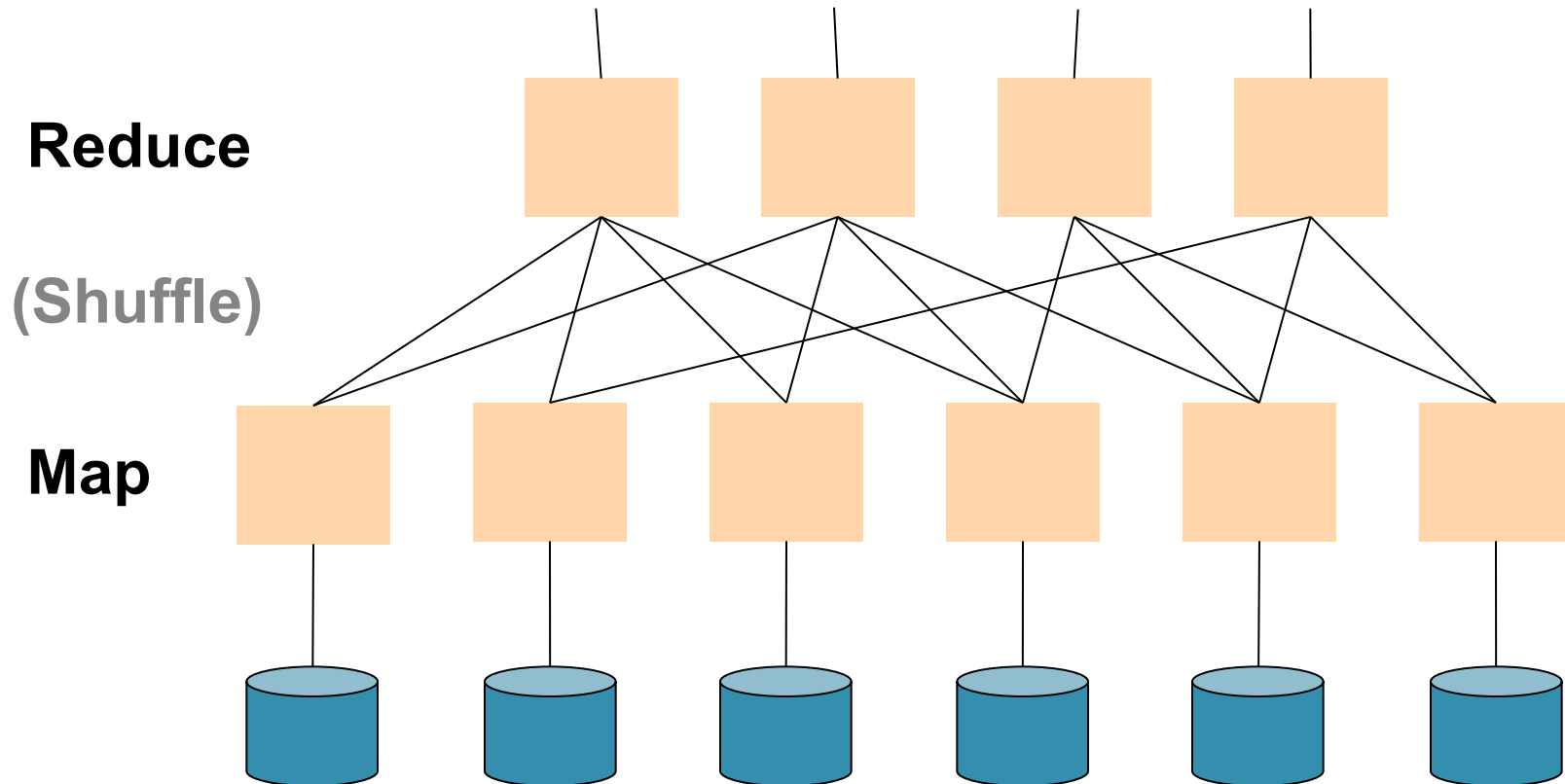
Distributed File System (DFS)

- 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

Observation: Your favorite parallel algorithm...



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

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 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

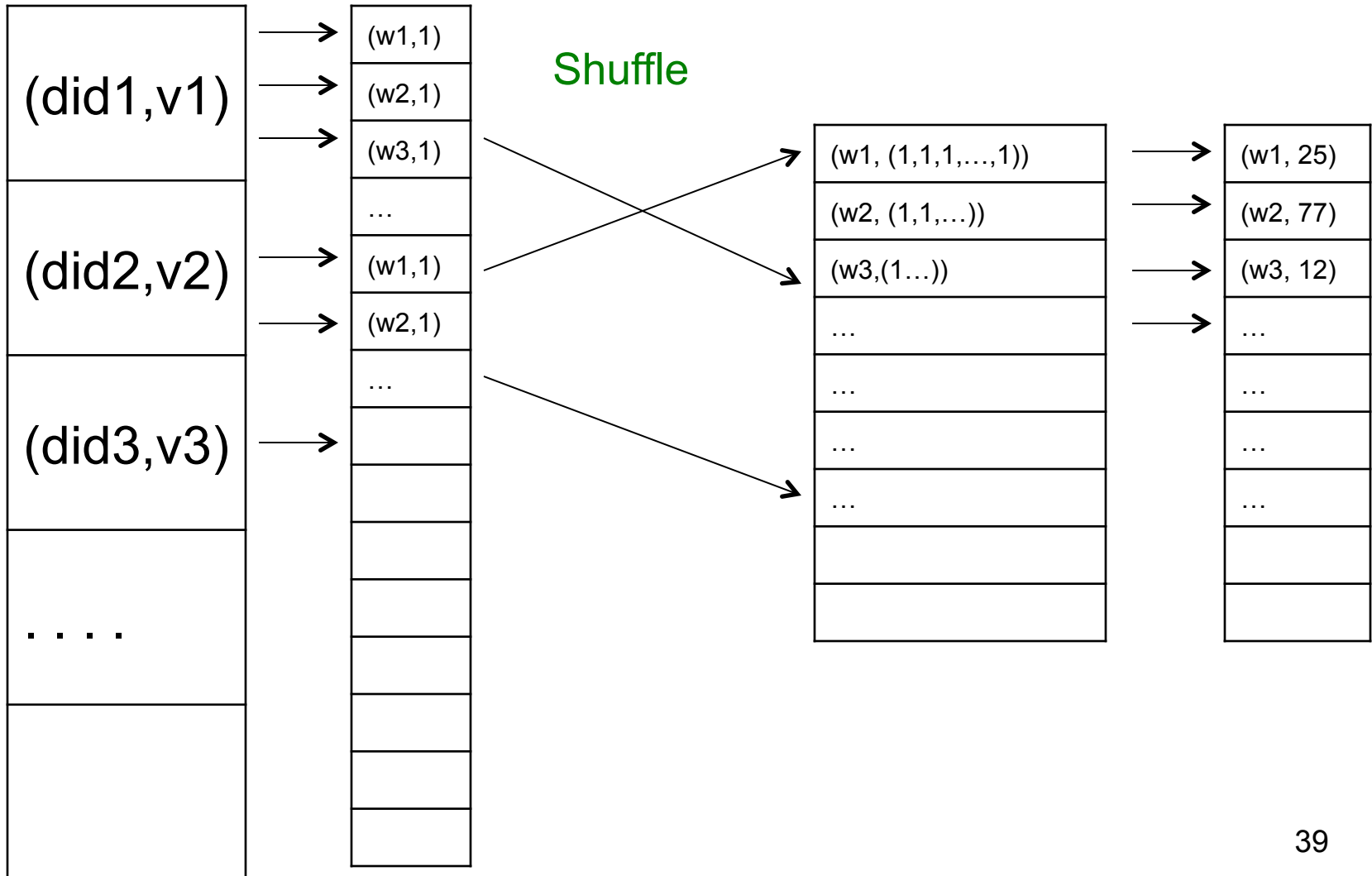
- Counting the number of occurrences of each word in a large collection of documents
- Each Document
 - The **key** = document id (**did**)
 - The **value** = set of words (**word**)

```
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



Jobs v.s. Tasks

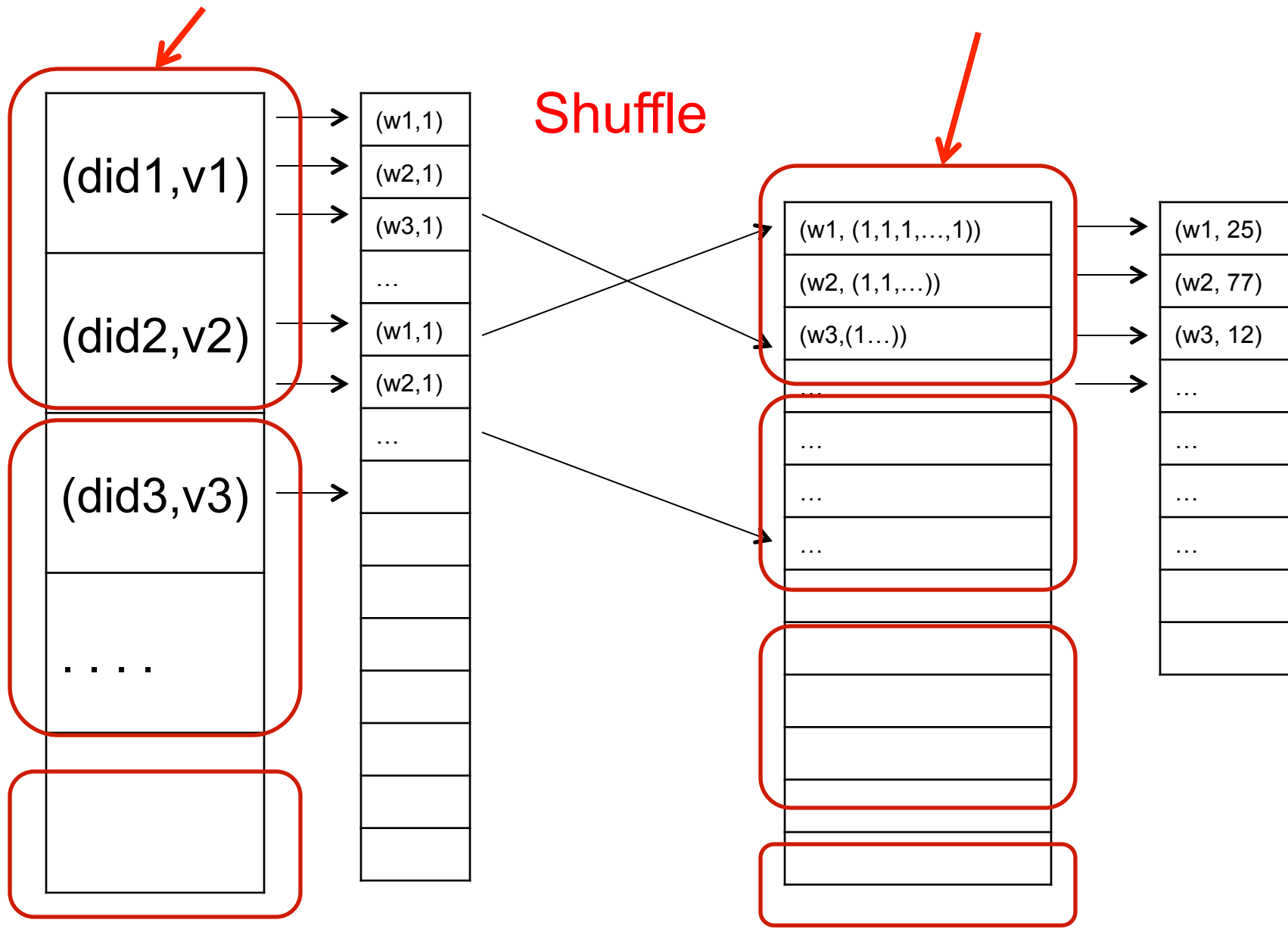
- A **MapReduce Job**
 - One single “query”, e.g. count the words in all docs
 - More complex queries may consists of multiple jobs
- A **Map Task**, or a **Reduce Task**
 - A group of instantiations of the map-, or reduce-function, which are scheduled on a single worker

Workers

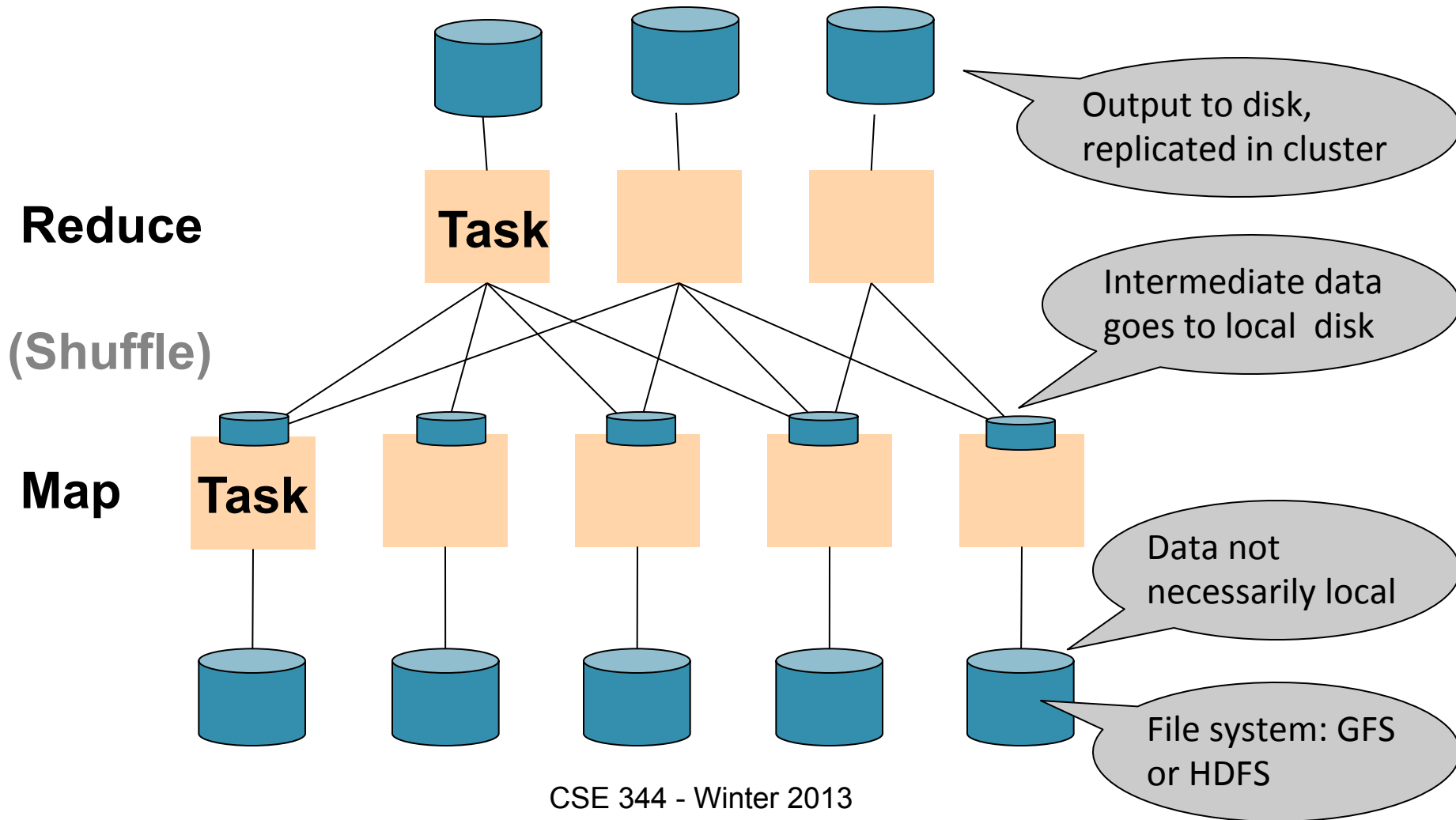
- A **worker** is a process that executes one task at a time
- Typically there is one worker per processor, hence 4 or 8 per node

MAP Tasks

REDUCE Tasks

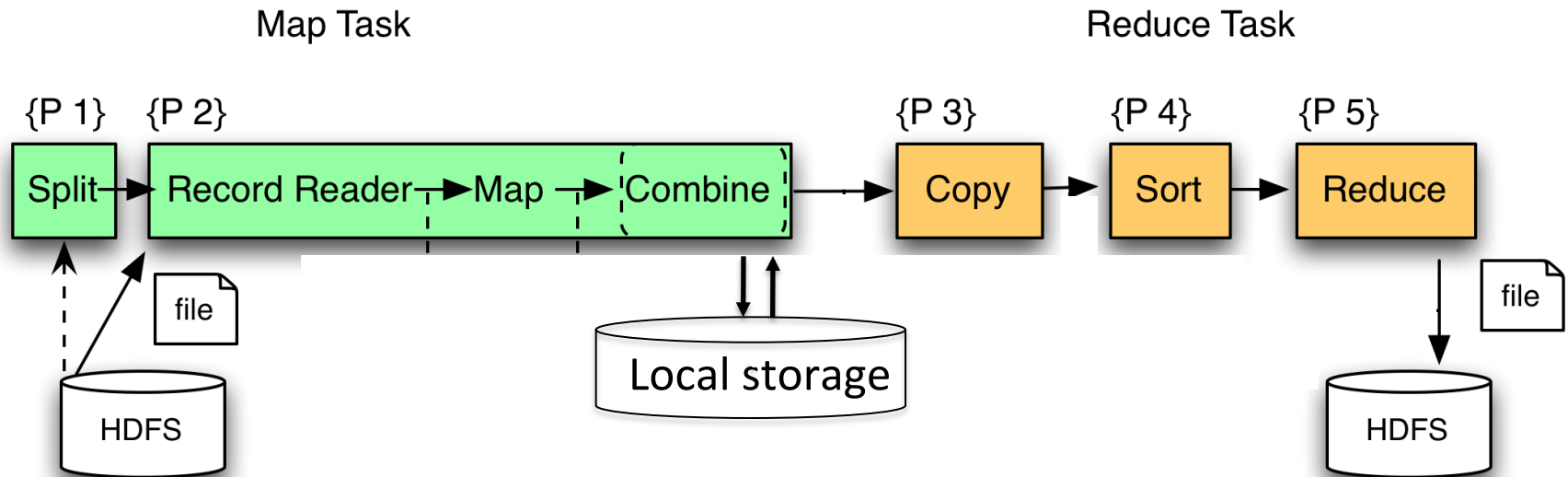


MapReduce Execution Details



MR Phases

- Each Map and Reduce task has multiple phases:



Example: CloudBurst



CloudBurst. Lake Washington Dataset (1.1GB). 80 Mappers 80 Reducers.

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*

MapReduce Summary

- Hides scheduling and parallelization details
- However, very limited queries
 - Difficult to write more complex queries
 - Need multiple MapReduce jobs
- Solution: declarative query language

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

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!

Review: Parallel DBMS

Figure 5 - Master server performs global planning and dispatch

