Introduction to Database Systems
CSE 414

Lecture 27: Map Reduce,

slides on Pig Latin

Announcements

- Last webquiz due tonight, 11 pm
- HW8 due on Friday
  - Try to make lots of progress over weekend
- Final exam:
  - Mon. 6/10, 2:30-4:20, this room
  - Comprehensive
  - Same rules as before: open textbook + 1 sheet of handwritten notes (+ midterm sheet), nothing else
- Review session:
  - Sunday, 6/9, 2 pm, Room TBD

Outline

- A clever parallel evaluation algorithm
- 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
- Assignment: learn Pig Latin for HW8 from the lecture notes, example starter code, and the Web; will discuss (too) briefly in class

A Challenge

- Have P servers (say P=27 or P=1000)
- How do we compute this query?
  $Q(x,y,z) = R(x,y),S(y,z),T(z,x)$

A Challenge

- Have P servers (say P=27 or P=1000)
- How do we compute this query?
  $Q(x,y,z) = R(x,y),S(y,z),T(z,x)$

- This computes all "triangles".
- E.g. let Follows(x,y) be all pairs of Twitter users s.t. x follows y. Let R=S=T=Follows. Then Q computes all triples of people that follow each other.

A Challenge

- Have P servers (say P=27 or P=1000)
- How do we compute this query?
  $Q(x,y,z) = R(x,y),S(y,z),T(z,x)$
- Step 1:
  - Each server sends R(x,y) to server h(y) mod P
  - Each server sends S(y,z) to server h(y) mod P

A Challenge

- Have P servers (say P=27 or P=1000)
- How do we compute this query?
  Q(x,y,z) = R(x,y),S(y,z),T(z,x)
- Step 1:
  - Each server sends R(x,y) to server h(y) mod P
  - Each server sends S(y,z) to server h(y) mod P
- Step 2:
  - Each server computes R⋈S locally
  - Each server sends [R(x,y),S(y,z)] to h(x) mod P
  - Each server sends T(z,x) to h(x) mod P

A Challenge

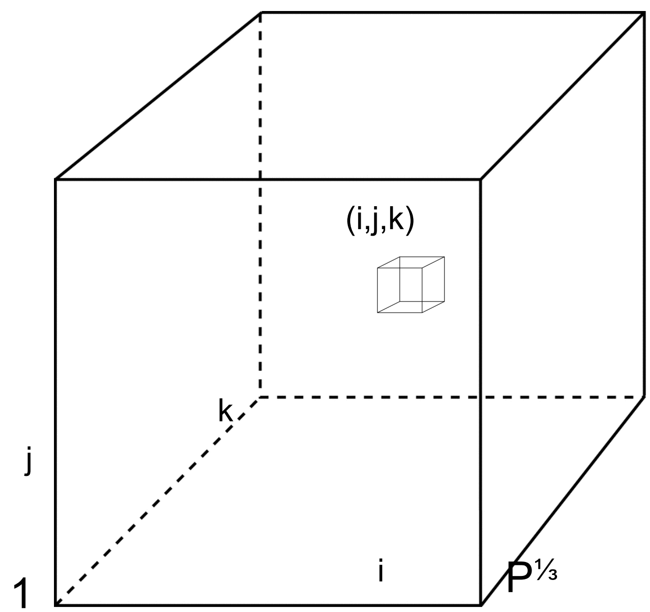
- Have P servers (say P=27 or P=1000)
- How do we compute this query?
  $Q(x,y,z) = R(x,y),S(y,z),T(z,x)$
- Step 1:
  - Each server sends R(x,y) to server h(y) mod P
  - Each server sends S(y,z) to server h(y) mod P
- Step 2:
  - Each server computes $R \bowtie S$ locally
  - Each server sends [R(x,y),S(y,z)] to h(x) mod P
  - Each server sends T(z,x) to h(x) mod P
- Final output:
  - Each server computes locally and outputs $R \bowtie S \bowtie T$

A Challenge

- Have P servers (say P=27 or P=1000)
- How do we compute this query in one step?
  $Q(x,y,z) = R(x,y),S(y,z),T(z,x)$

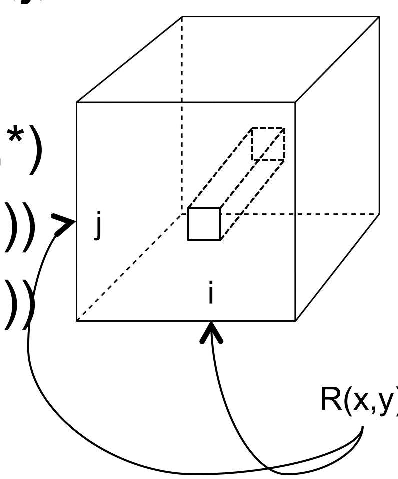
A Challenge

- Have P servers (say P=27 or P=1000)
- How do we compute this query in one step?
  $Q(x,y,z) = R(x,y),S(y,z),T(z,x)$
- Organize the P servers into a cube with side $P^{1/3}$
  - Thus, each server is uniquely identified by (i,j,k), $i,j,k \le P^{1/3}$



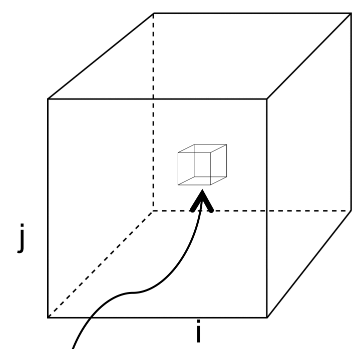
A Challenge

- Have P servers (say P=27 or P=1000)
- How do we compute this query in one step?
  $Q(x,y,z) = R(x,y),S(y,z),T(z,x)$
- Organize the P servers into a cube with side $P^{1/3}$
  - Thus, each server is uniquely identified by (i,j,k), $i,j,k \le P^{1/3}$
- Step 1:
  - Each server sends R(x,y) to all servers (h(x),h(y),*)
  - Each server sends S(y,z) to all servers (*,h(y),h(z))
  - Each server sends T(x,z) to all servers (h(x),*,h(z))



A Challenge

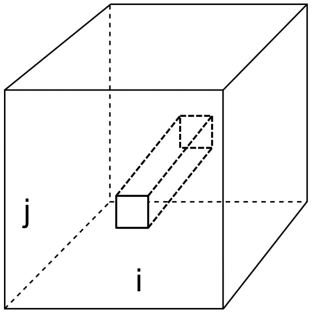
- Have P servers (say P=27 or P=1000)
- How do we compute this query in one step?
  Q(x,y,z) = R(x,y),S(y,z),T(z,x)
- Organize the P servers into a cube with side $P^{1/3}$
  - Thus, each server is uniquely identified by (i,j,k), $i,j,k \leq P^{1/3}$
- Step 1:
  - Each server sends R(x,y) to all servers (h(x),h(y),*)
  - Each server sends S(y,z) to all servers (*,h(y),h(z))
  - Each server sends T(x,z) to all servers (h(x),*,h(z))



- Final output:
  - Each server (i,j,k) computes the query R(x,y),S(y,z),T(z,x) locally

A Challenge

- Have P servers (say P=27 or P=1000)
- How do we compute this query in one step?
  Q(x,y,z) = R(x,y),S(y,z),T(z,x)
- Organize the P servers into a cube with side $P^{1/3}$
  – Thus, each server is uniquely identified by (i,j,k), $i,j,k \leq P^{1/3}$
- Step 1:
  – Each server sends R(x,y) to all servers (h(x),h(y),*)
  – Each server sends S(y,z) to all servers (*,h(y),h(z))
  – Each server sends T(x,z) to all servers (h(x),*,h(z))



- Final output:
  – Each server (i,j,k) computes the query R(x,y),S(y,z),T(z,x) locally
- Analysis: each tuple R(x,y) is replicated at most $P^{1/3}$ times

Parallel Data Processing at Massive Scale

Data Centers Today

- Data Center: 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)

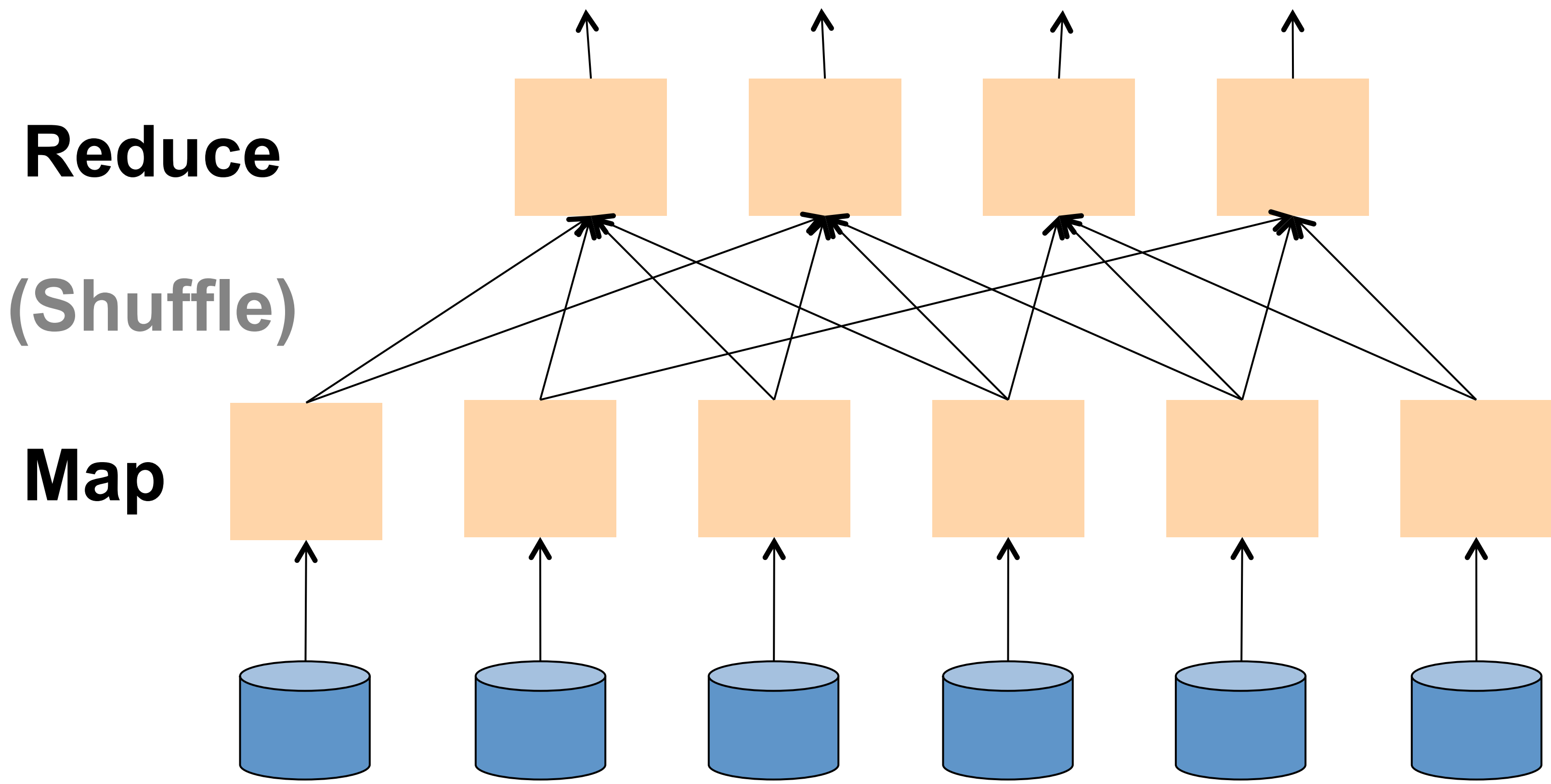
- For very large files: TBs, PBs
- Each file is partitioned into *chunks*, typically 64MB
- Each chunk is replicated several times (≥3), on different racks, for fault tolerance
- Implementations:
  - Google's DFS: GFS, proprietary
  - Hadoop's DFS: HDFS, open source

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

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)`**

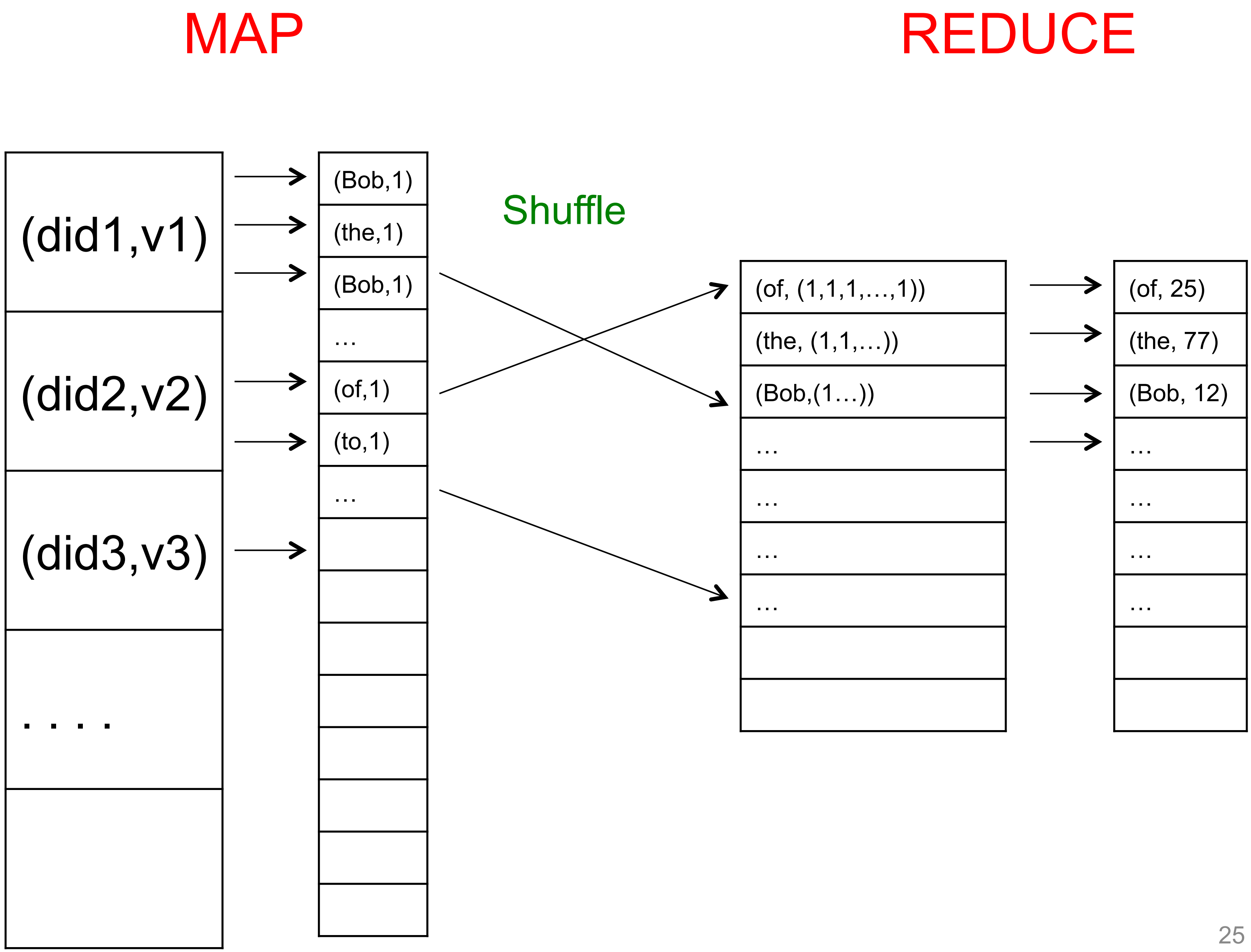
System groups all pairs with the same intermediate key, and passes the bag of values to the REDUCE function

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));
```

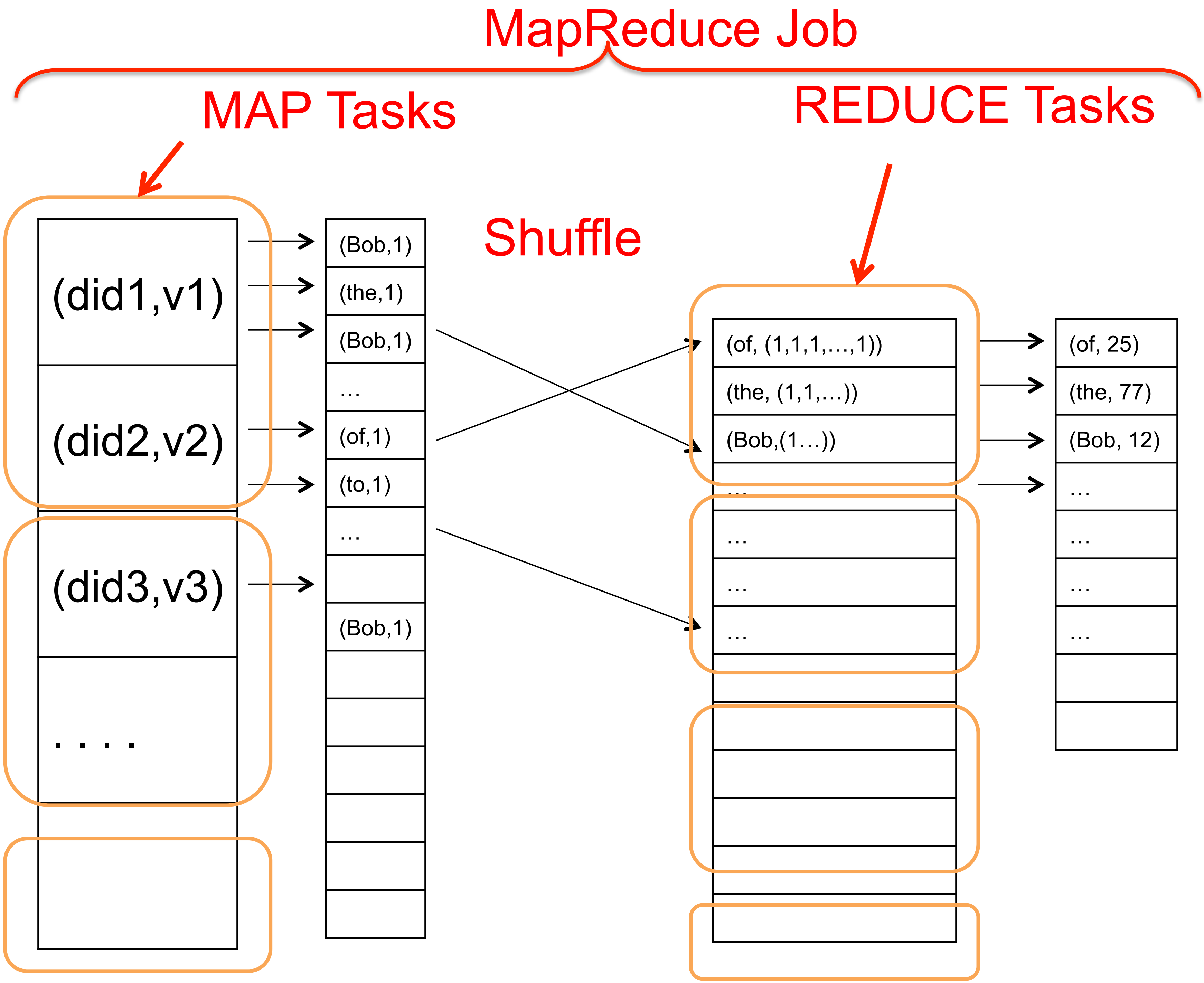

MAP
REDUCE
Shuffle
(did1,v1)
(did2,v2)
(did3,v3)
. . . .
(Bob,1)
(the,1)
(Bob,1)
…
(of,1)
(to,1)
…
(of, (1,1,1,…,1))
(the, (1,1,…))
(Bob,(1…))
…
…
…
…
(of, 25)
(the, 77)
(Bob, 12)
…
…
…
…

Jobs v.s. Tasks

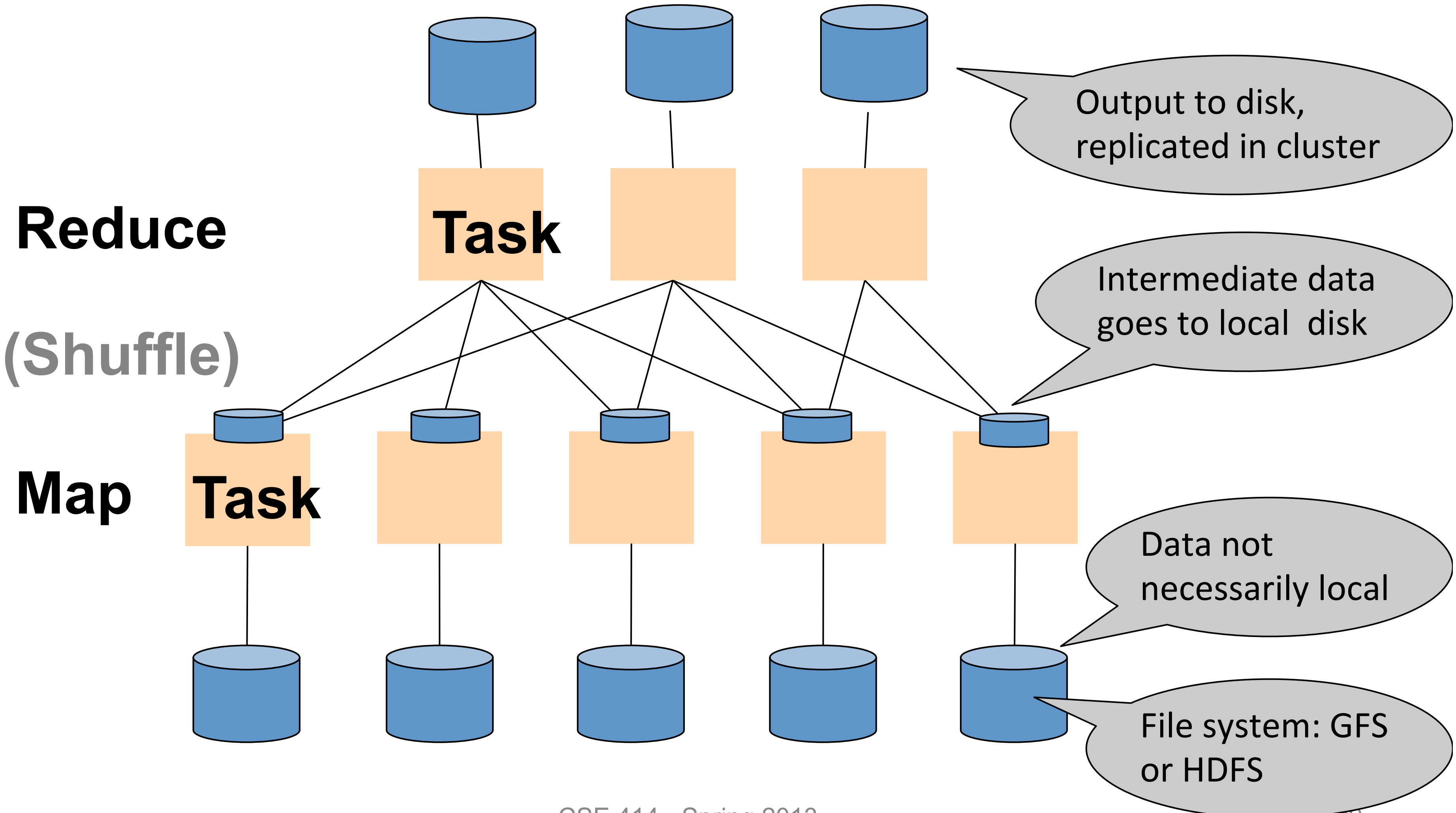
- A MapReduce Job
  - One single “query”, e.g. count the words in all docs
  - More complex queries may consists of multiple jobs
- A Map Task, or a Reduce Task
  - A group of instantiations of the map-, or reduce-function, which are scheduled on a single worker

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

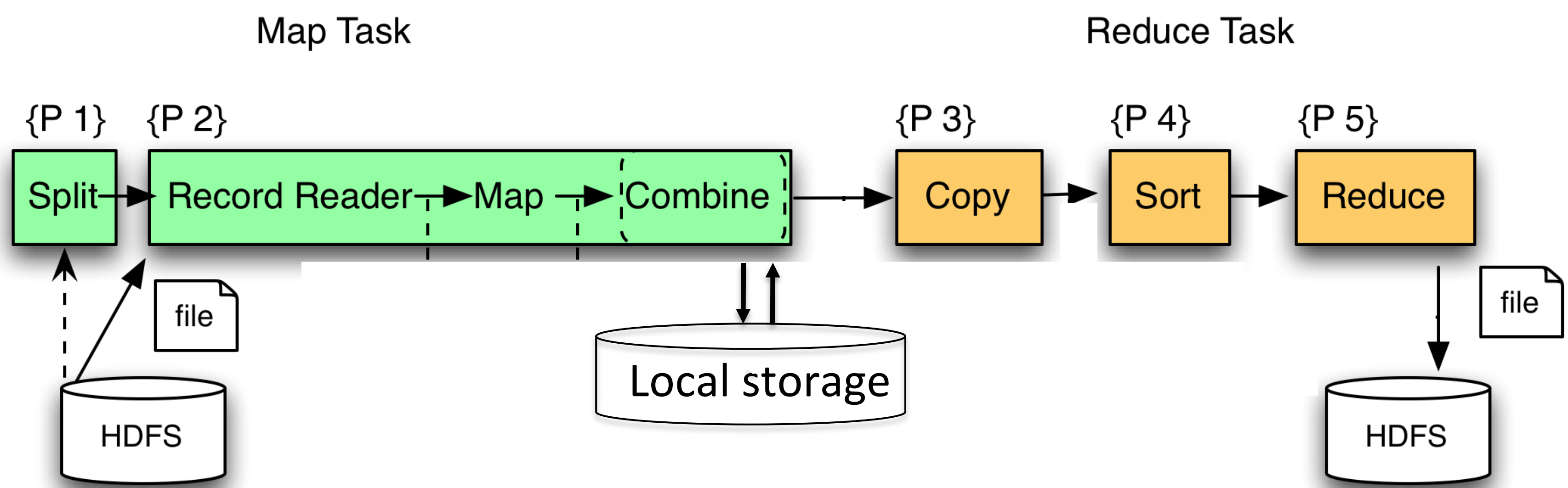

MapReduce Job
MAP Tasks
REDUCE Tasks
Shuffle
(did1,v1)
(did2,v2)
(did3,v3)
. . . .
(Bob,1)
(the,1)
(Bob,1)
…
(of,1)
(to,1)
…
(Bob,1)
(of, (1,1,1,…,1))
(the, (1,1,…))
(Bob,(1…))
…
…
…
…
(of, 25)
(the, 77)
(Bob, 12)
…
…
…
…

MapReduce Execution Details

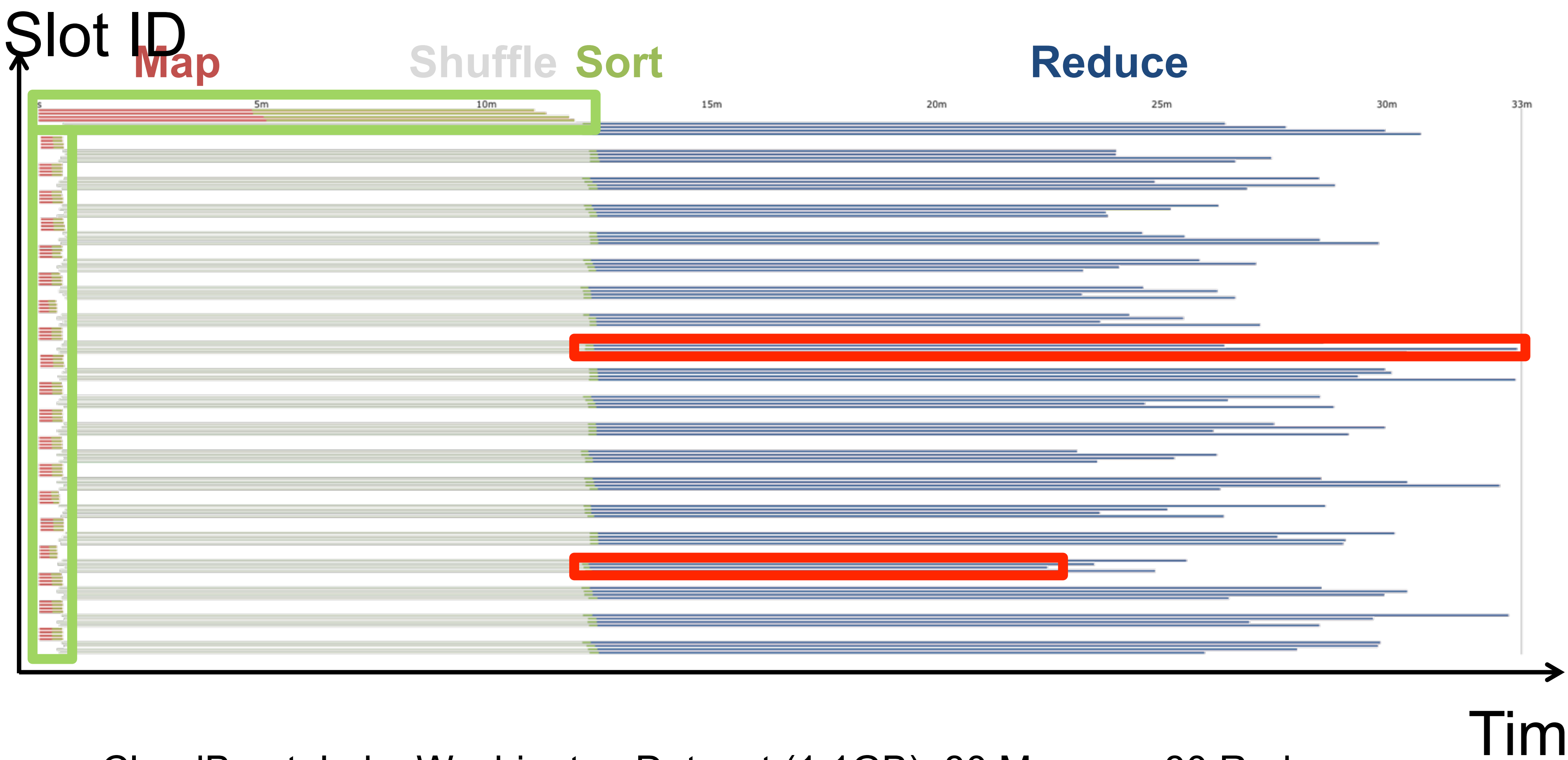


MR Phases

- 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 / Dremmel / 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!

Pig Latin Mini-Tutorial

(quick survey in class, but need to study outside in order to do homework 8)

Pig Latin Overview

- Data model = loosely typed *nested relations*
- Query model = a SQL-like, dataflow language

- Execution model:
  - Option 1: run locally on your machine; e.g. to debug
    - In HW6, debug with option 1 directly on Amazon
  - Option 2: compile into graph of MapReduce jobs, run on a cluster supporting Hadoop

Example

- Input: a table of urls:
  (url, category, pagerank)
- Compute the average pagerank of all sufficiently high pageranks, for each category
- Return the answers only for categories with sufficiently many such pages

Page(url, category, pagerank)

First in SQL…

```
SELECT category, AVG(pagerank)
FROM Page
WHERE pagerank > 0.2
GROUP BY category
HAVING COUNT(*) > 10^6
```

Page(url, category, pagerank)

…then in Pig-Latin

```
good_urls = FILTER urls BY pagerank > 0.2
groups = GROUP good_urls BY category
big_groups = FILTER groups
                          BY COUNT(good_urls) > 10^6
output = FOREACH big_groups GENERATE
                        category, AVG(good_urls.pagerank)
```

Types in Pig-Latin

- Atomic: string or number, e.g. ‘Alice’ or 55
- Tuple: (‘Alice’, 55, ‘salesperson’)
- Bag: {(‘Alice’, 55, ‘salesperson’),
  (‘Betty’,44, ‘manager’), …}
- Maps: we will try not to use these

Types in Pig-Latin

Tuple components can be referenced by number

- $0, $1, $2, …

Bags can be nested! Non 1st Normal Form

- {('a', {1,4,3}), ('c',{ }), ('d', {2,2,5,3,2})}

[Olston'2008]

$$t = \left(\text{'alice'}, \left\{ \begin{array}{c} (\text{'lakers'}, 1) \\ (\text{'iPod'}, 2) \end{array} \right\}, [\text{'age'} \rightarrow 20] \right)$$

Let fields of tuple `t` be called `f1, f2, f3`

Expression Type	Example	Value for `t`
Constant	'bob'	Independent of `t`
Field by position	$0	'alice'
Field by name	f3	['age' → 20]
Projection	f2.$0	{ ('lakers') ('iPod') }
Map Lookup	f3#'age'	20
Function Evaluation	SUM(f2.$1)	1 + 2 = 3
Conditional Expression	f3#'age'>18? 'adult':'minor'	'adult'
Flattening	FLATTEN(f2)	'lakers', 1 'iPod', 2

[Olston'2008]

Loading data

- Input data = FILES !
  - Heard that before ?

- The LOAD command parses an input file into a bag of records
- Both parser (="deserializer") and output type are provided by user

For HW6: simply use the code provided

[Olston'2008]

Loading data

```
queries = LOAD 'query_log.txt'
          USING myLoad( )
          AS (userID, queryString, timeStamp)
```

Pig provides a set of built-in load/store functions

A = LOAD 'student' USING PigStorage('\t') AS (name: chararray, age:int, gpa: float);
same as
A = LOAD 'student' AS (name: chararray, age:int, gpa: float);

[Olston'2008]

Loading data

- USING userfuction( ) -- is optional
  - Default deserializer expects tab-delimited file
- AS type – is optional
  - Default is a record with unnamed fields; refer to them as $0, $1, …
- The return value of LOAD is just a handle to a bag
  - The actual reading is done in pull mode, or parallelized

[Olston'2008]

FOREACH

```
expanded_queries =
   FOREACH queries
   GENERATE userId, expandQuery(queryString)
```

expandQuery( ) is a UDF that produces likely expansions
Note: it returns a bag, hence expanded_queries is a nested bag

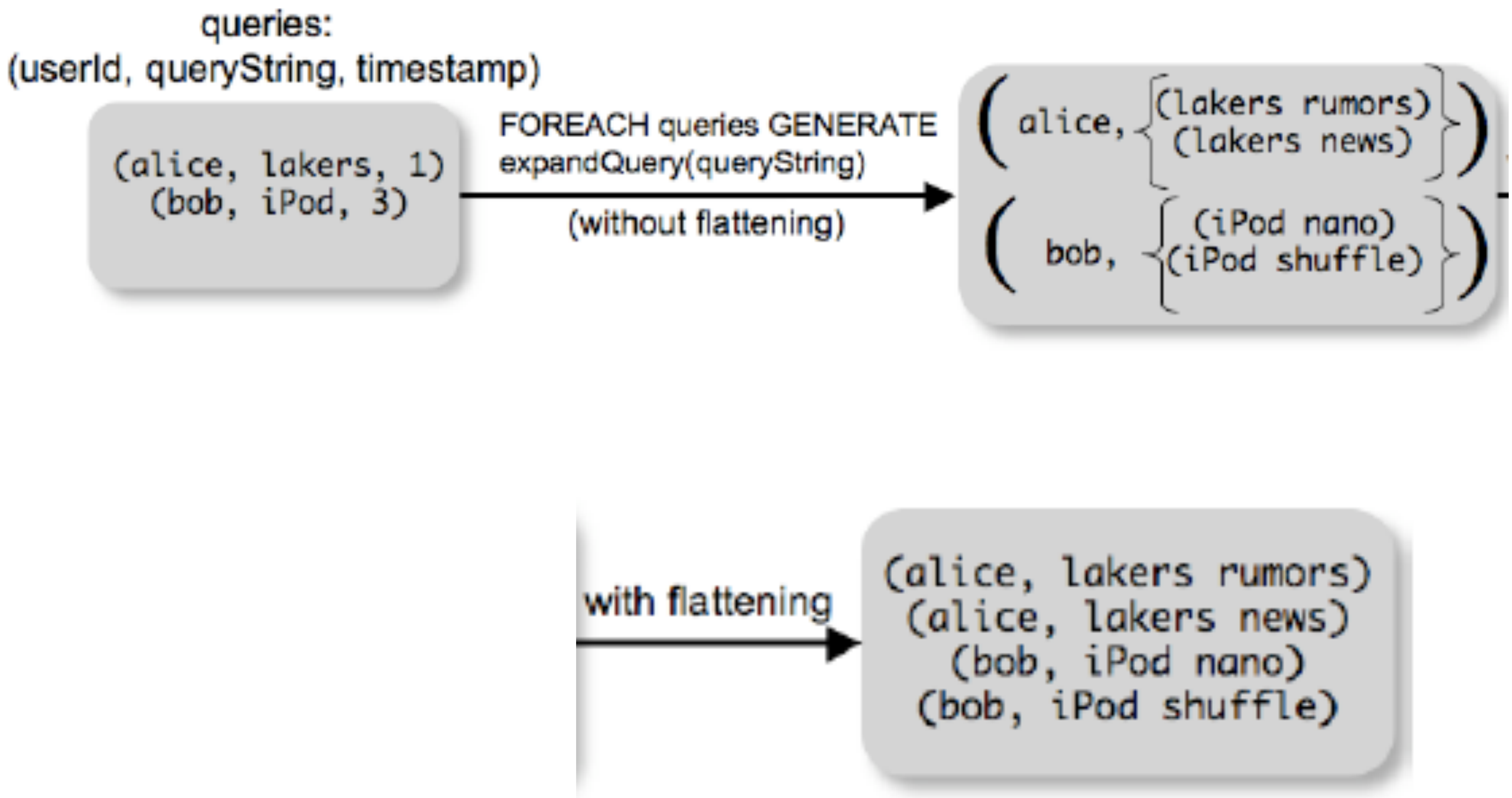
[Olston'2008]

FOREACH

```
expanded_queries =
   FOREACH queries
   GENERATE userId,
                  flatten(expandQuery(queryString))
```

Now we get a flat collection

[Olston'2008]


queries:
(userId, queryString, timestamp)
(alice, lakers, 1)
(bob, iPod, 3)
FOREACH queries GENERATE
expandQuery(queryString)
(without flattening)
alice,
(lakers rumors)
(lakers news)
bob,
(iPod nano)
(iPod shuffle)
with flattening
(alice, lakers rumors)
(alice, lakers news)
(bob, iPod nano)
(bob, iPod shuffle)

[Olston'2008]

FLATTEN

Note that it is NOT a normal function !

(that's one thing questionable about Pig-latin)

- A normal FLATTEN would do this:
  - FLATTEN({{2,3},{5},{},{4,5,6}}) = {2,3,5,4,5,6}
  - Its type is: {{T}} → {T}
- The Pig Latin FLATTEN does this:
  - FLATTEN({4,5,6}) = 4, 5, 6
  - What is its Type? {T} → T, T, T, …, T ?????

[Olston'2008]

FILTER

Remove all queries from Web bots:

```
real_queries = FILTER queries BY userId neq 'bot'
```

Better: use a complex UDF to detect Web bots:

```
real_queries = FILTER queries
               BY NOT isBot(userId)
```

[Olston'2008]

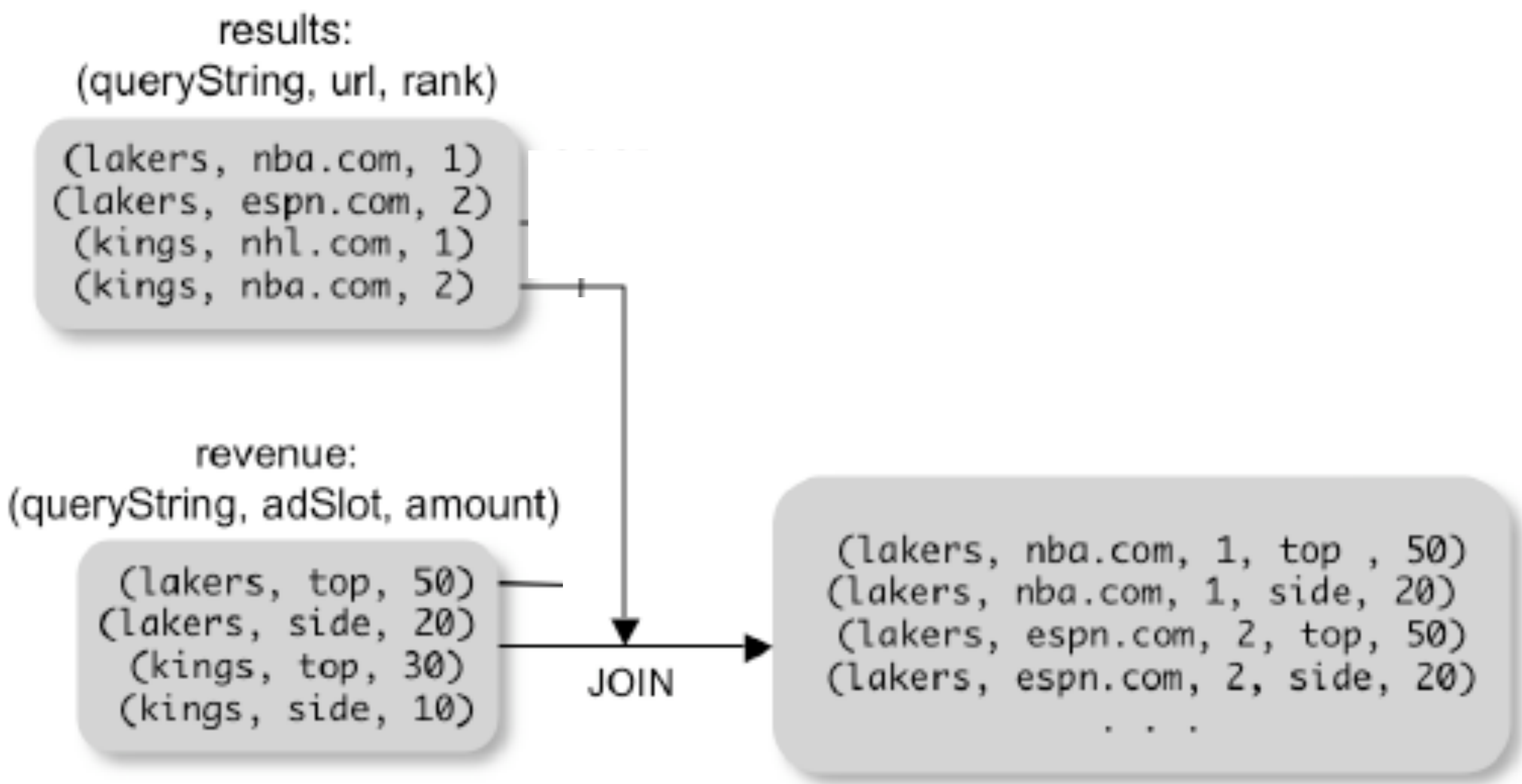
JOIN

results: {(queryString, url, position)}
revenue: {(queryString, adSlot, amount)}

```
join_result = JOIN results BY queryString
                   revenue BY queryString
```

join_result : {(queryString, url, position, adSlot, amount)}

[Olston'2008]



results:
(queryString, url, rank)
(lakers, nba.com, 1)
(lakers, espn.com, 2)
(kings, nhl.com, 1)
(kings, nba.com, 2)
revenue:
(queryString, adSlot, amount)
(lakers, top, 50)
(lakers, side, 20)
(kings, top, 30)
(kings, side, 10)
JOIN
(lakers, nba.com, 1, top , 50)
(lakers, nba.com, 1, side, 20)
(lakers, espn.com, 2, top, 50)
(lakers, espn.com, 2, side, 20)
. . .

[Olston'2008]

GROUP BY

revenue: {(queryString, adSlot, amount)}

```
grouped_revenue = GROUP revenue BY queryString
query_revenues =
    FOREACH grouped_revenue
    GENERATE queryString,
             SUM(revenue.amount) AS totalRevenue
```

grouped_revenue: {(queryString, {(adSlot, amount)})}
query_revenues: {(queryString, totalRevenue)}

[Olston'2008]

Simple MapReduce

input : {(field1, field2, field3, ...)}

```
map_result = FOREACH input
                 GENERATE FLATTEN(map(*))
key_groups = GROUP map_result BY $0
output = FOREACH key_groups
             GENERATE $0, reduce($1)
```

map_result : {(a1, a2, a3, . . .)}
key_groups : {(a1, {(a2, a3, . . .)})}

[Olston'2008]

Co-Group

results: {(queryString, url, position)}
revenue: {(queryString, adSlot, amount)}

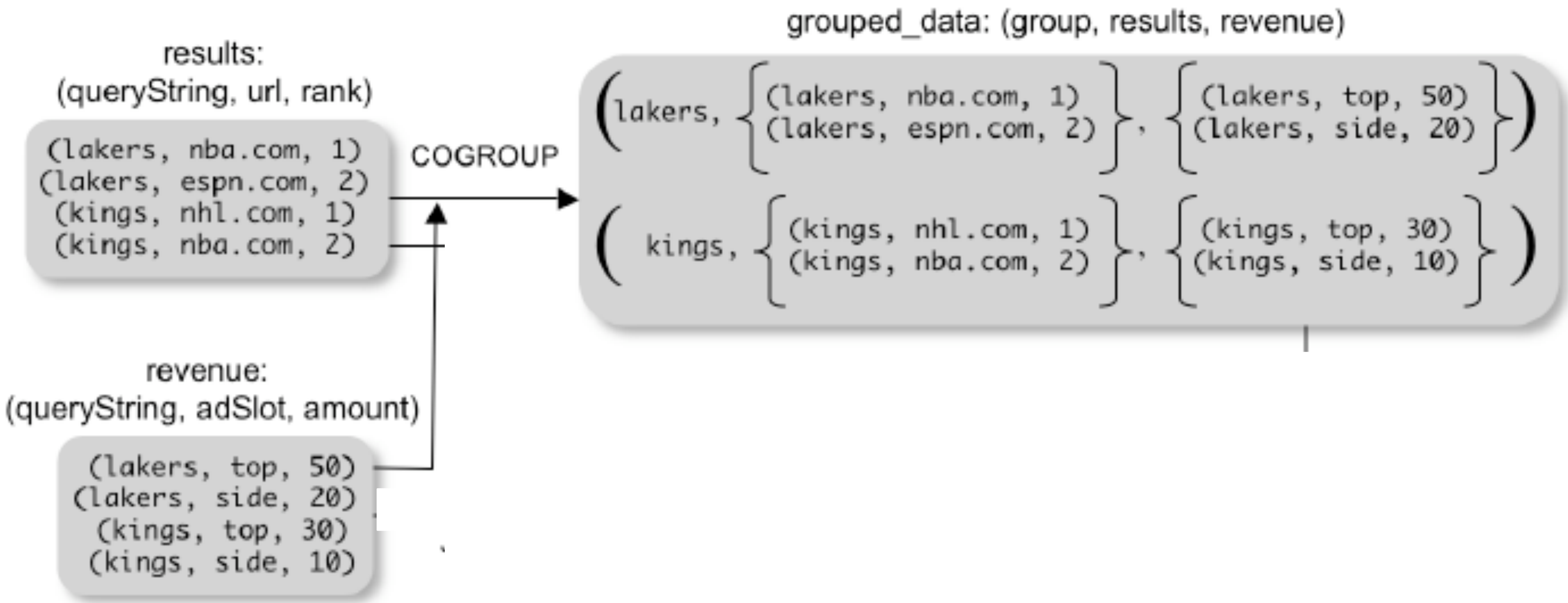
```
grouped_data =
      COGROUP results BY queryString,
                    revenue BY queryString;
```

grouped_data: {(queryString, results:{(url, position)},
                                                    revenue:{(adSlot, amount)})}

What is the output type in general ?

[Olston'2008]

Co-Group



Is this an inner join, or an outer join ?

[Olston'2008]

Co-Group

grouped_data: {(queryString, results:{(url, position)},
revenue:{(adSlot, amount)})}

```
url_revenues = FOREACH grouped_data
               GENERATE
               FLATTEN(distributeRevenue(results, revenue));
```

distributeRevenue is a UDF that accepts search re-
sults and revenue information for a query string at a time,
and outputs a bag of urls and the revenue attributed to them.

[Olston'2008]

Co-Group v.s. Join

grouped_data: {(queryString, results:{(url, position)},
revenue:{(adSlot, amount)})}

```
grouped_data = COGROUP results BY queryString,
                       revenue BY queryString;
join_result = FOREACH grouped_data
              GENERATE FLATTEN(results),
                       FLATTEN(revenue);
```

Result is the same as JOIN

[Olston'2008]

Asking for Output: STORE

```
STORE query_revenues INTO `myoutput'
USING myStore();
```

Meaning: write query_revenues to the file 'myoutput'

[Olston'2008]

Implementation

- Over Hadoop !
- Parse query:
  - Everything between LOAD and STORE → one logical plan
- Logical plan → graph of MapReduce ops
- All statements between two (CO)GROUPs → one MapReduce job

[Olston'2008]

Implementation