Introduction to Data Management CSE 344

Lecture 23 and 24:

Map Reduce

+ slides on Pig Latin

Announcements

- HW8 due next Thursday (1 late day allowed)
  - Start early and read all instructions carefully
  - Prob#4 may take > 4 hours just to run

- Review session on BCNF / ER diagram -- Monday, CSE 303, by Vaspol, 4:30-5:30 pm

Outline

- 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 not discuss in class

Map-Reduce

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 (8-64).
  - nodes in a rack are connected by a network, typically gigabit Ethernet

- Data center: holds many racks
  - Racks are connected by network with higher bandwidth or a switch

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
  - We do not want to abort and restart the computation every time a component fails

- Solution:
  - (1) keep redundant data, duplicate file at several nodes
  - (2) Divide computation into tasks, now only tasks need to be restarted

Distributed File System (DFS)

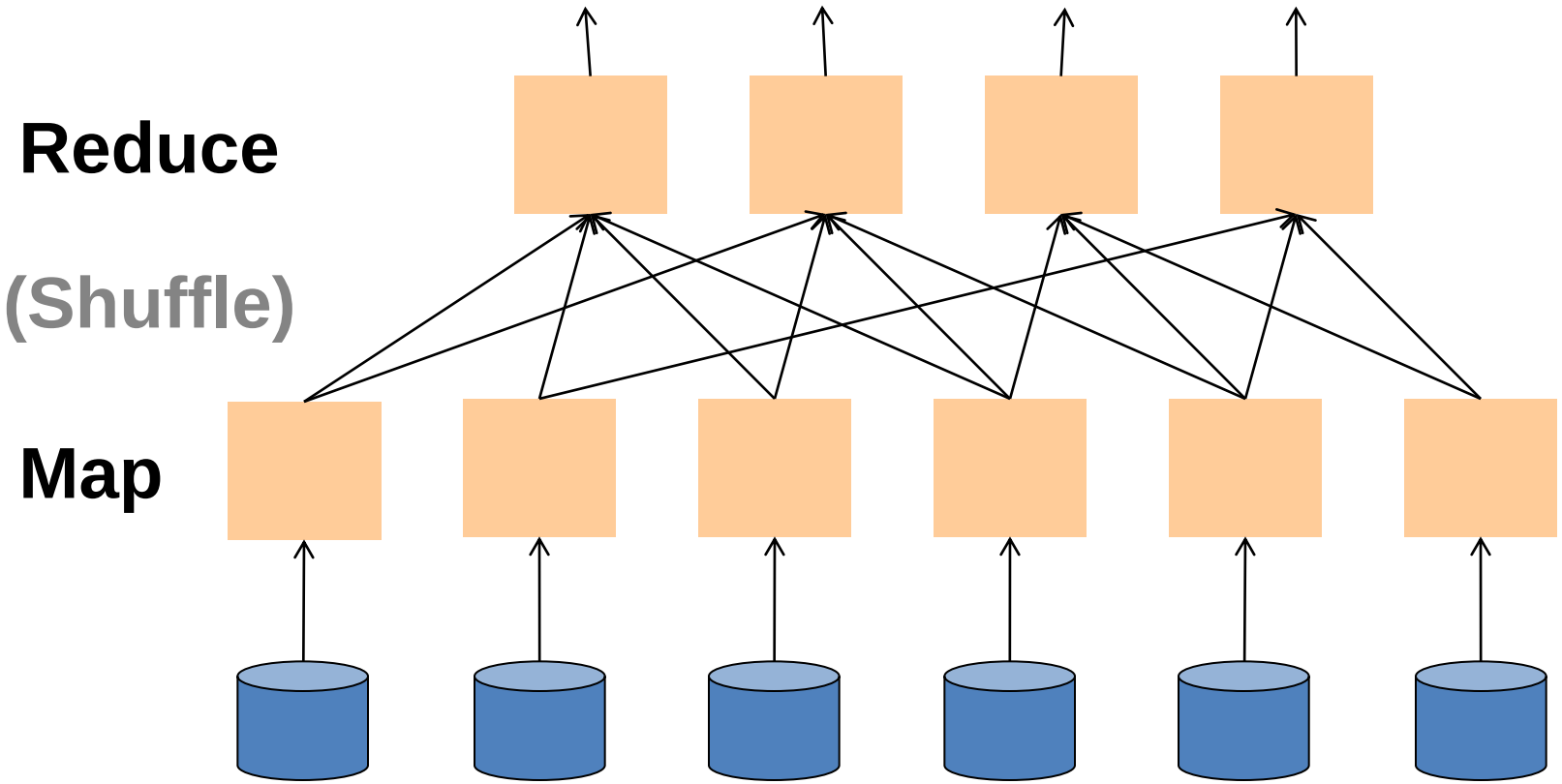
- For very large files (TBs, PBs) that are not updated frequently
- Each file is partitioned into *chunks*, typically 64MB
- Each chunk is replicated several times (≥3), on different racks, for fault tolerance
- Implementations:
  - Google's DFS: GFS, proprietary
  - Hadoop's DFS: HDFS, open source

MapReduce

- Google: paper published 2004
- Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI 2004.
- Free variant: Hadoop

- MapReduce = high-level programming model and implementation for large-scale parallel data processing

Map Reduce Framework



You only need to write the Map and Reduce functions
System will take care of co-ordination, parallel execution, failures

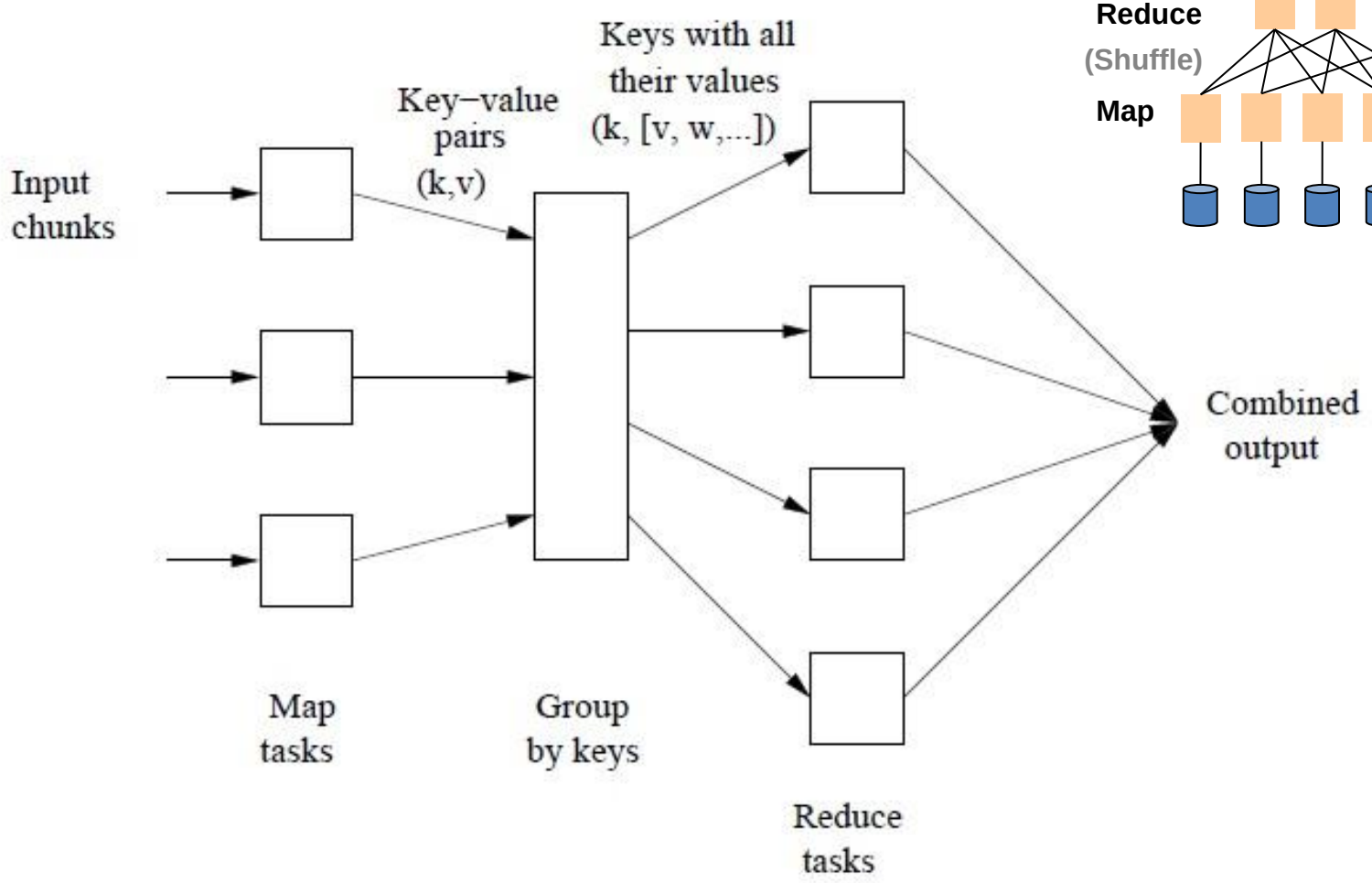
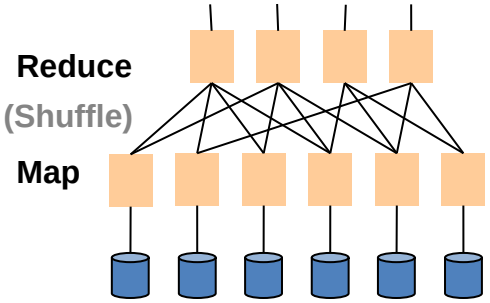


Figure 2.2: Schematic of a MapReduce computation

Fig from the reading assignment

Typical Problems Solved by MR



- Read a lot of data
- Map: extract something (value) you care about from each record (and index with a key)
- Shuffle and Sort
  - same key goes to same reducer node along with its value
- Reduce: aggregate, summarize, filter, transform (values with the same key)
- Write the results

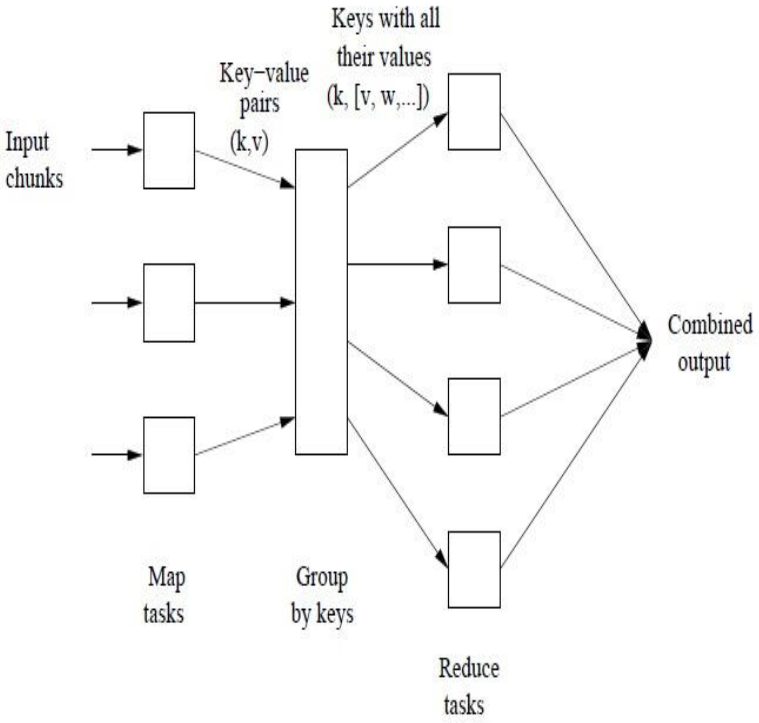
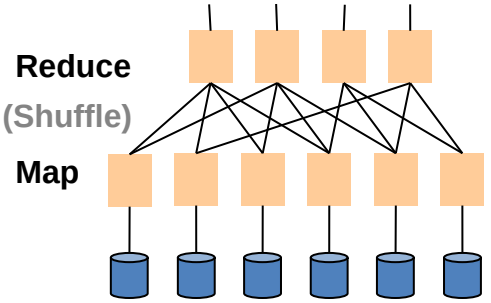


Figure 2.2: Schematic of a MapReduce computation

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

Data Model



Files! Input for Map tasks

A file = a bag of **`(key, value)`** pairs

A MapReduce program (everything is in key-value form):

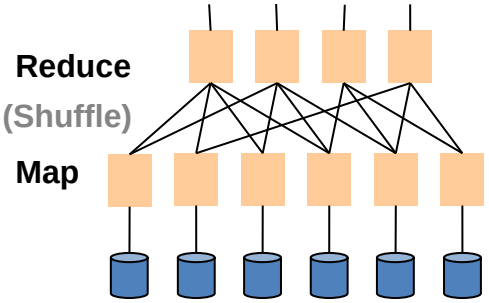
- Input to Map: a bag of **`(inputkey, value)`** pairs from files
- Outout of Map = Input to Reduce

  `(intermediate key, value)` pairs from files

  This is what you have to define carefully
- Output of Reduce: a bag of **`(outputkey, value)`** pairs

- Sometimes inputkey - outputkey are not mentioned explicitly

Step 1: the MAP Phase



User provides the MAP-function:

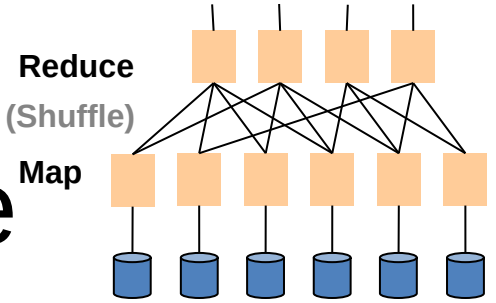
- Input: **`(input key, value)`**
- Ouput:
  bag of **`(intermediate key, value)`**

Values are typically different

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

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

Step 2: the REDUCE Phase



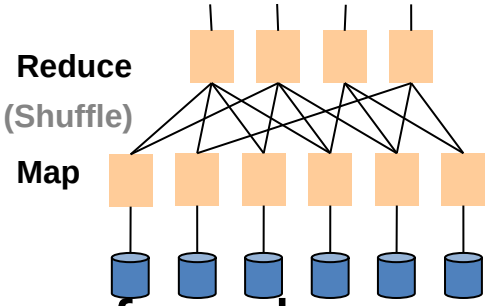
System groups all pairs with the same intermediate key, and passes the bag of values to the REDUCE function

User provides the REDUCE function:

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

  - `Outputkey` is the same as the `intermediatekey` and is often omitted

Example

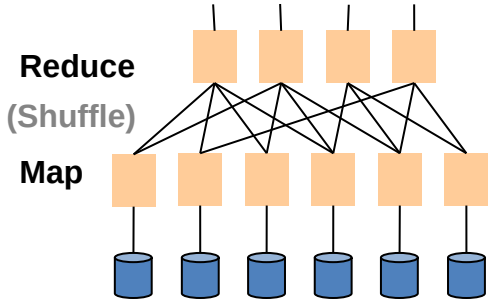


- Counting the number of occurrences of each word in a large collection of documents
- Each Document
  - The (input) key = document id (did)
  - The (input) value = set of words (word)

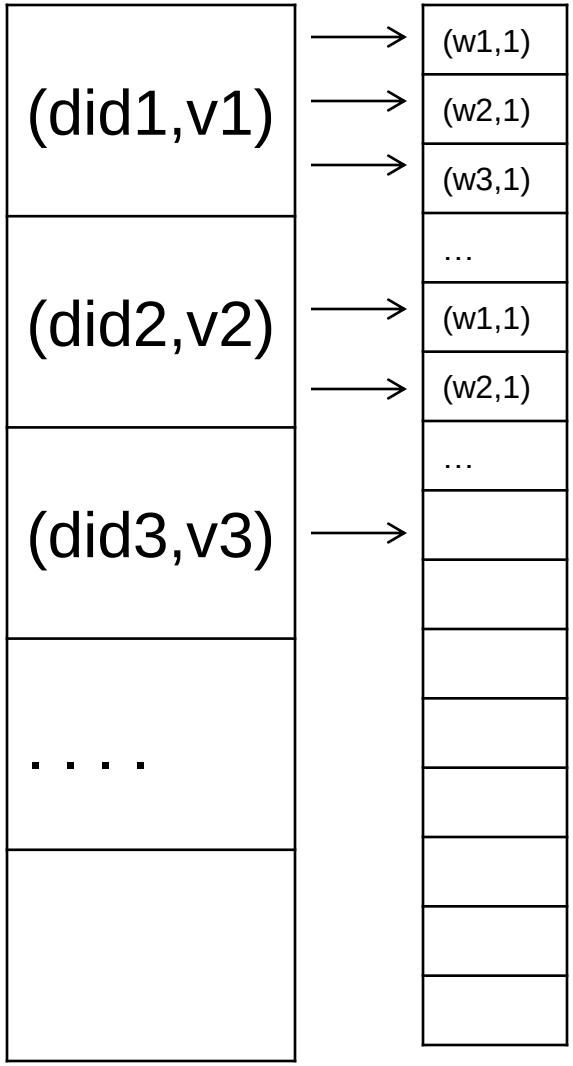
We want to have the same word in the same reducer

```
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));
```

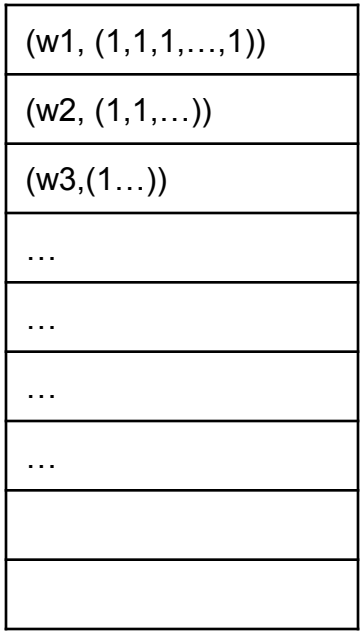


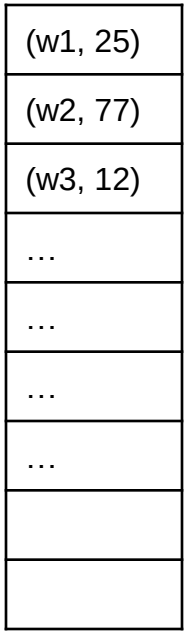
MAP



Shuffle

REDUCE





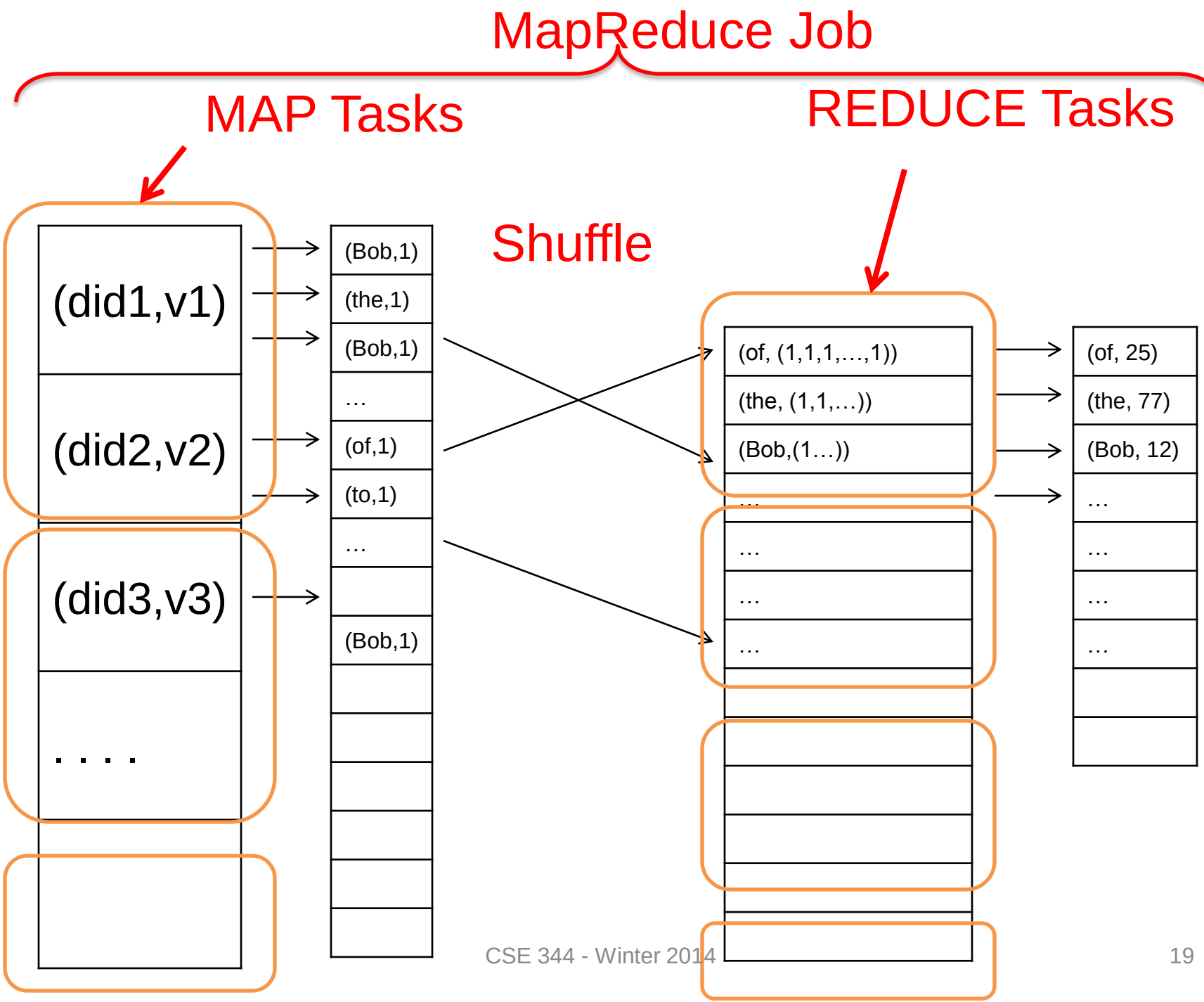
Jobs v.s. Tasks

- 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
  - “Master controller process” controls the workers

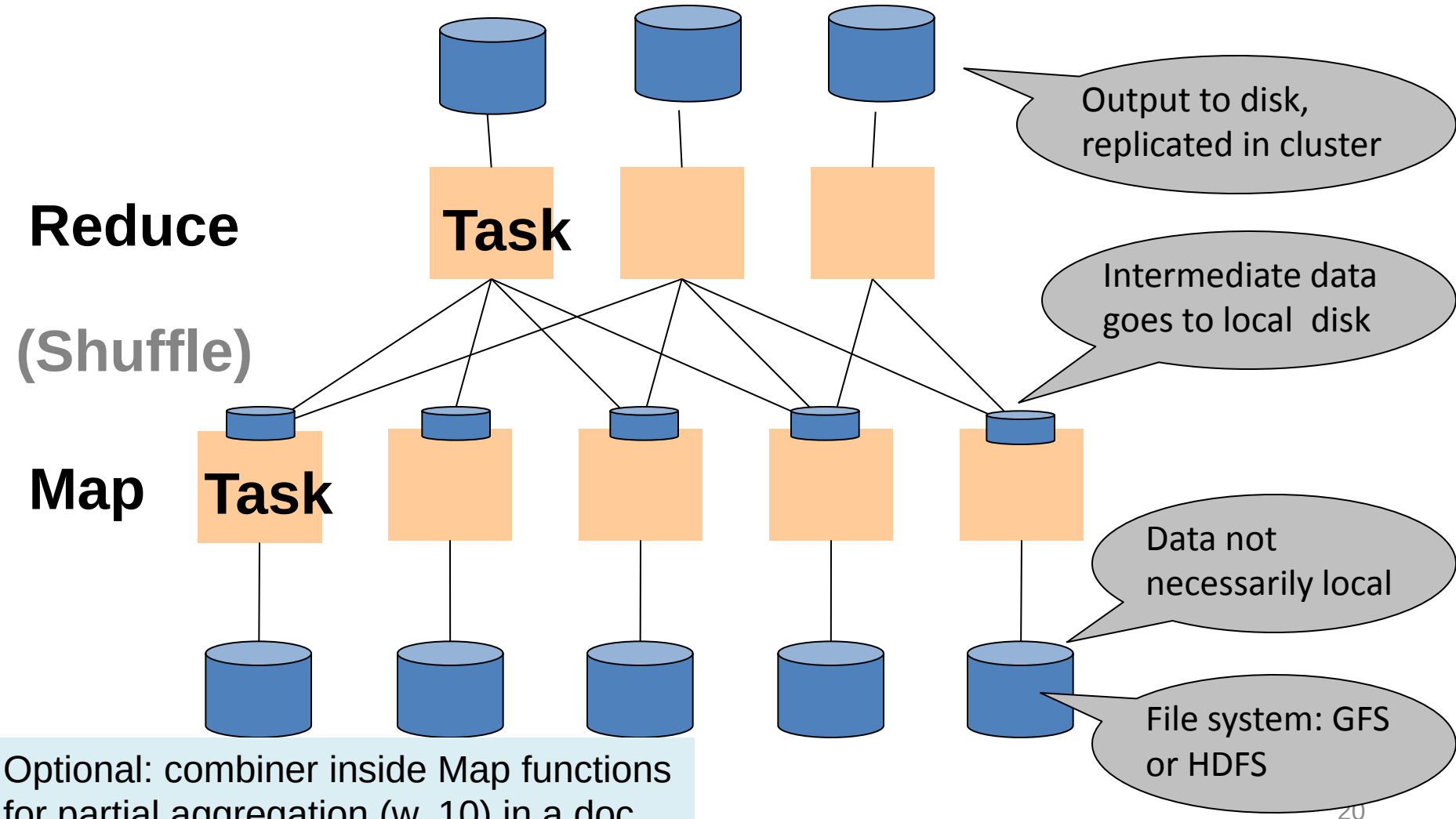
Terminology

- A worker is a process that executes one task at a time.
  - Typically 1 worker per processor, hence 4-8 workers per computing node

- A reducer : gets the same key
- A reduce task:
  - gets scheduled on the same worker
  - the same reducer always is in the same reduce task
  - but a reduce task can handle multiple reducers (multiple keys), e.g. by hashing
- Max parallelism by one reducer $\rightarrow$ one reduce task
  - But often not useful, many more keys than the number of available nodes
  - Some value lists are longer than the other (skew)
  - Needs to be handled while scheduling reducers to reduce tasks and reduce tasks to computing nodes


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)
…
…
…
…

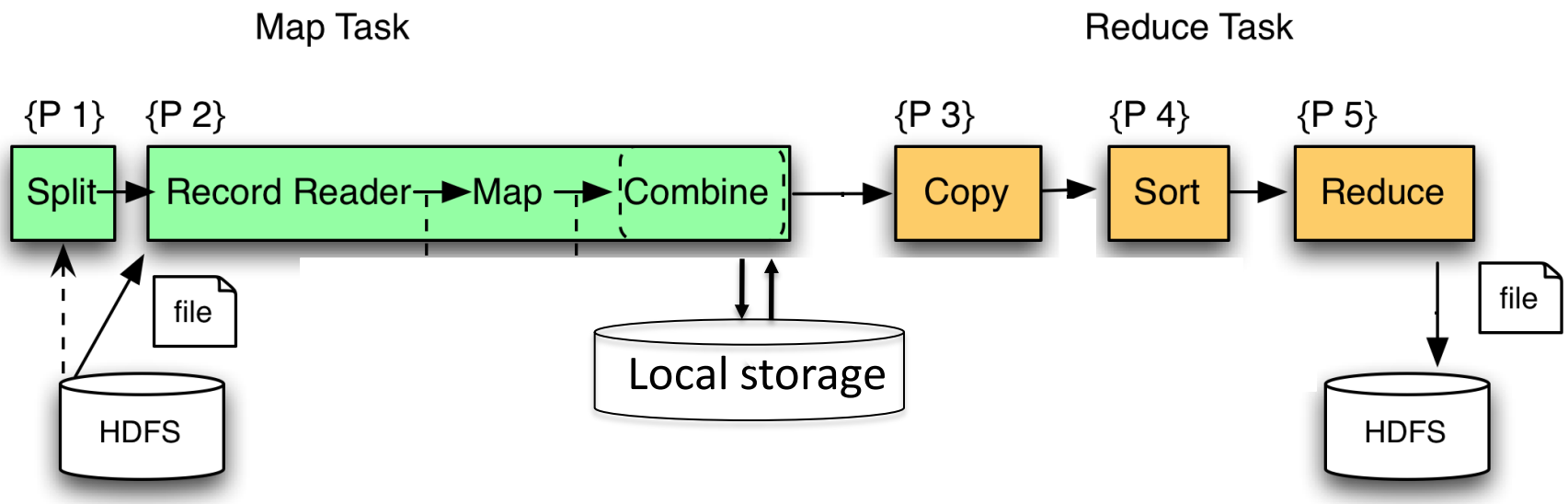
MapReduce Execution Details



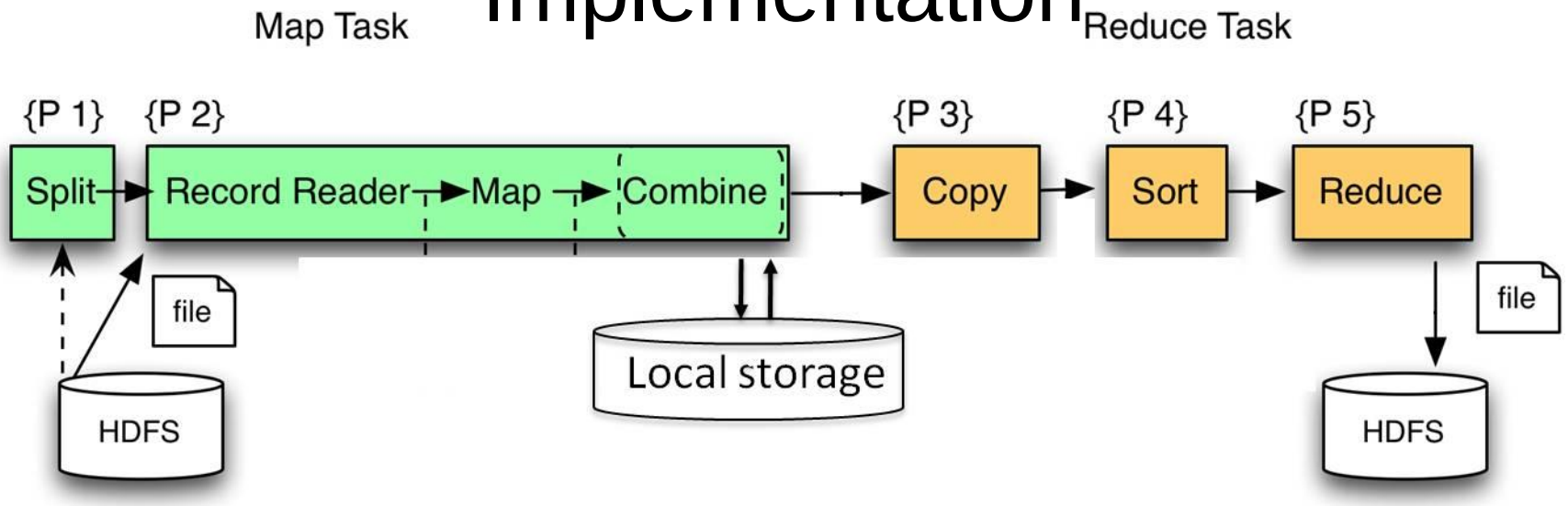
Optional: combiner inside Map functions for partial aggregation (w, 10) in a doc

MR Phases

- Each Map and Reduce task has multiple 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*
  - So M x R regions in total
- Master assigns workers to the *R reduce tasks*
- Reduce workers read regions from the map workers' local disks
  - e.g. Reduce worker#1 will read from partition#1 of each Map worker's disk
    Reduce worker#2 will …. partition#2 of ….
    and so on.

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

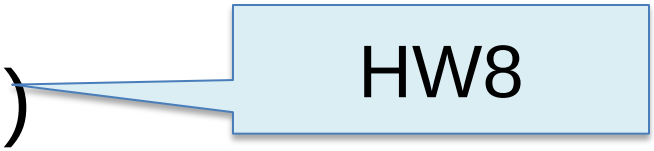
- Hides scheduling and parallelization details

- 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)
  - BigQuery – SQL in the cloud



Executing a Large MapReduce Job

Anatomy of a Query Execution

- Running problem #4
- 20 nodes = 1 master + 19 workers
- Using PARALLEL 50

March 2013

Hadoop job_201303091944_0001 on domU-12-31-39-06-75-A1

User: hadoop
Job Name: PigLatin:DefaultJobName
Job File: hdfs://10.208.122.79:9000/mnt/var/lib/hadoop/tmp/mapred/staging/hadoop/.staging/job_201303091944_0001/job.xml
Submit Host: domU-12-31-39-06-75-A1.compute-1.internal
Submit Host Address: 10.208.122.79
Job-ACLs: All users are allowed
Job Setup: Successful
Status: Succeeded
Started at: Sat Mar 09 19:49:21 UTC 2013
Finished at: Sat Mar 09 23:33:14 UTC 2013
Finished in: 3hrs, 43mins, 52sec
Job Cleanup: Successful
Black-listed TaskTrackers: 1

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	7908	0	0	7908	0	14 / 16
reduce	100.00%	50	0	0	50	0	0 / 8

	Counter	Map	Reduce	Total
Job Counters	SLOTS_MILLIS_MAPS	0	0	454,162,761
	Launched reduce tasks	0	0	58
	Total time spent by all reduces waiting after reserving slots (ms)	0	0	0
	Rack-local map tasks	0	0	7,938
	Total time spent by all maps waiting after reserving slots	0	0	0

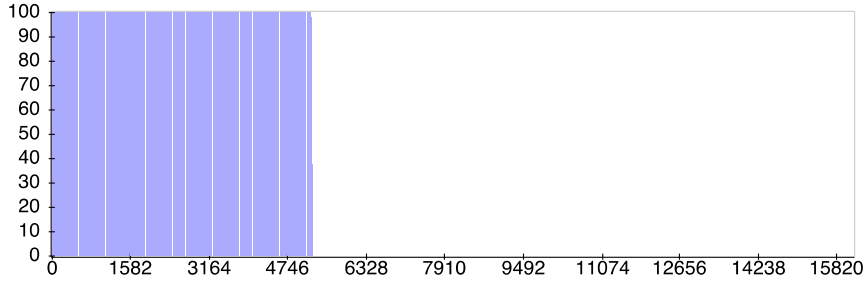
Some other time (March 2012)

- Let's see what happened…

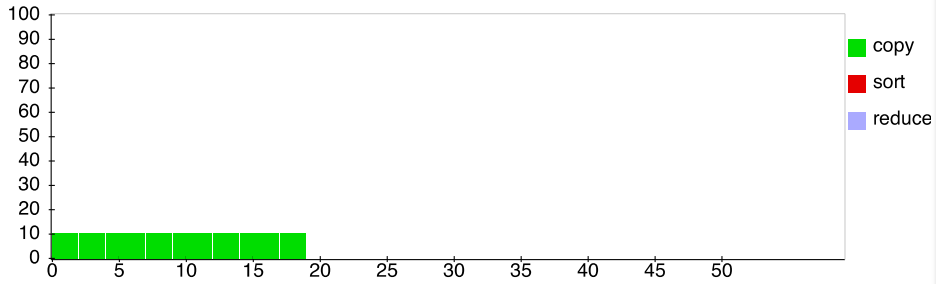
Take a look at the Hadoop tutorial:
https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html

1h 16min

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	33.17%	15816	10549	38	5229	0	0 / 0
reduce	4.17%	50	31	19	0	0	0 / 0



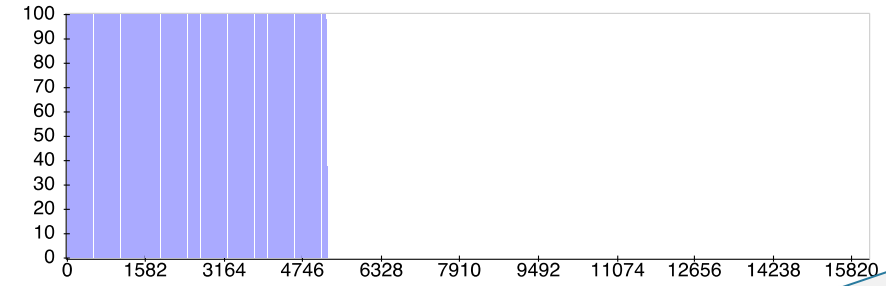
luce Completion Graph - close

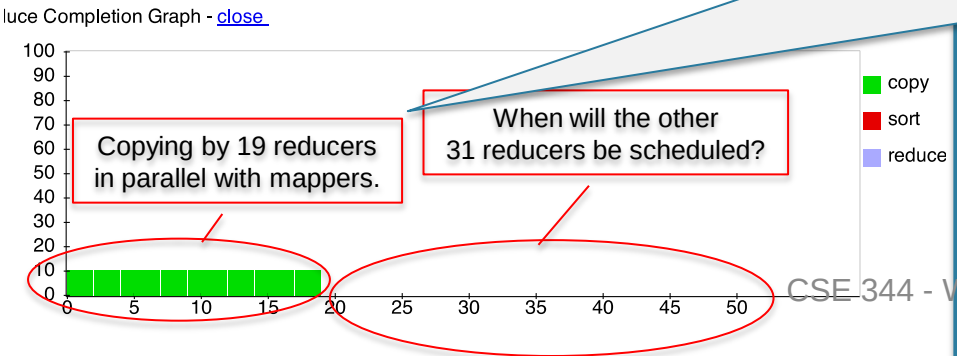


1h 16min

Only 19 reducers active, out of 50. Why?

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	33.17%	15816	10549	38	5229	0	0 / 0
reduce	4.17%	50	31	19	0	0	0 / 0



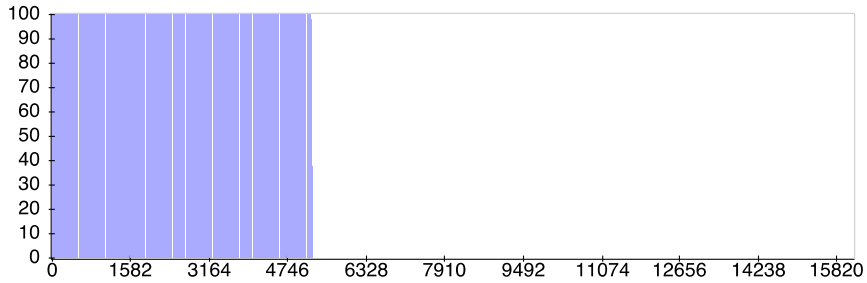


- Map workers keep writing data to local disk
- Reduce workers can start "copy"-ing in parallel
- But cannot start "reduce" functions until the map workers are done.

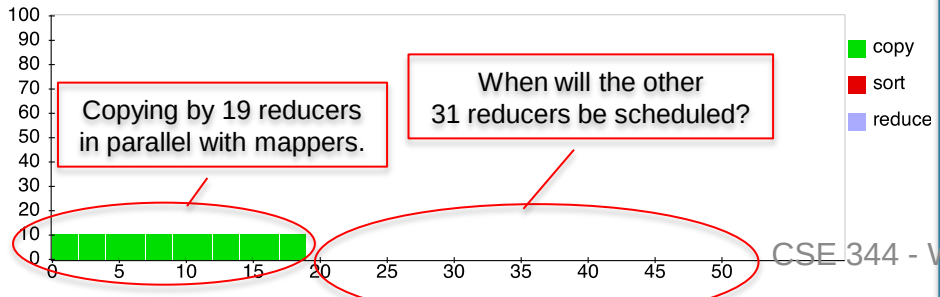
1h 16min

Only 19 reducers active, out of 50. Why?

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	33.17%	15816	10549	38	5229	0	0 / 0
reduce	4.17%	50	31	19	0	0	0 / 0

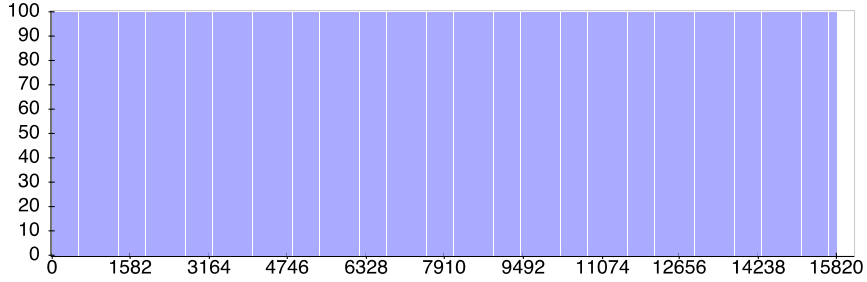


luce Completion Graph - close

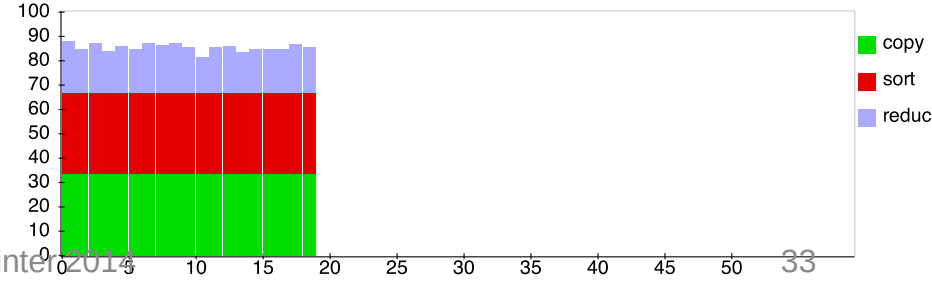


3h 50min

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	15816	0	0	15816	0	0 / 18
reduce	32.42%	50	31	19	0	0	0 / 0



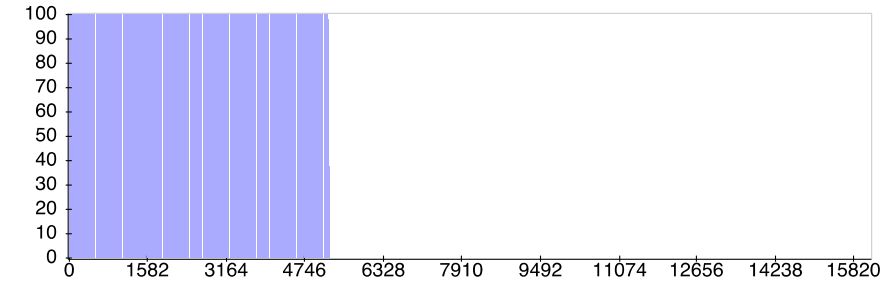
uce Completion Graph - close



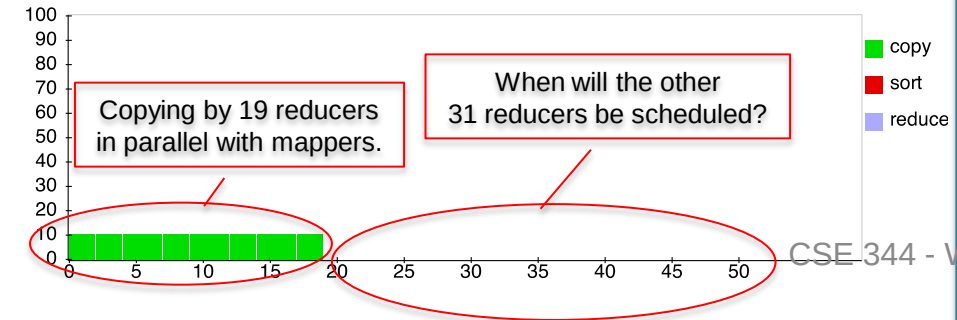
1h 16min

Only 19 reducers active, out of 50. Why?

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	33.17%	15816	10549	38	5229	0	0 / 0
reduce	4.17%	50	31	19	0	0	0 / 0



luce Completion Graph - close

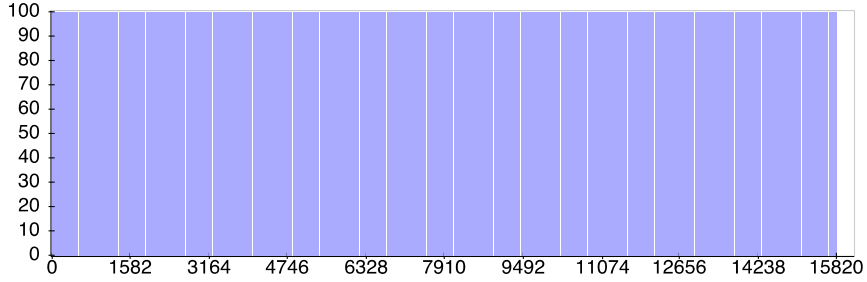


3h 50min

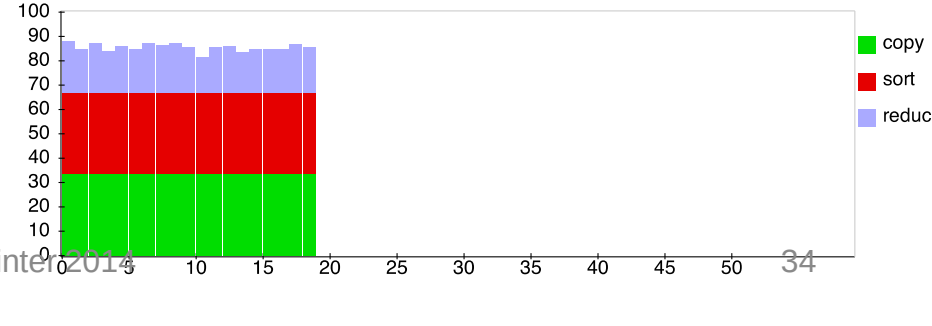
Speculative Execution

Completed. Sorting, and the rest of Reduce may proceed now

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	15816	0	0	15816	0	0 / 18
reduce	32.42%	50	31	19	0	0	0 / 0



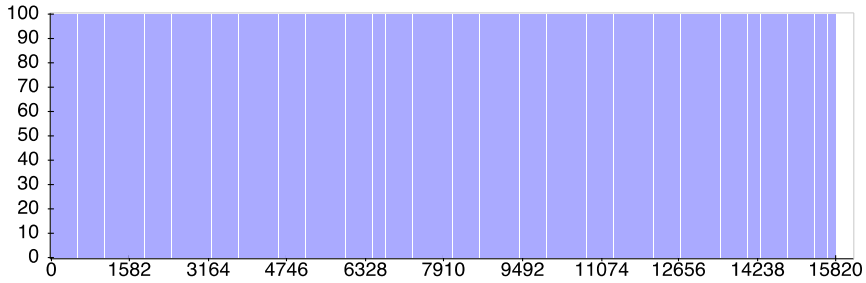
uce Completion Graph - close



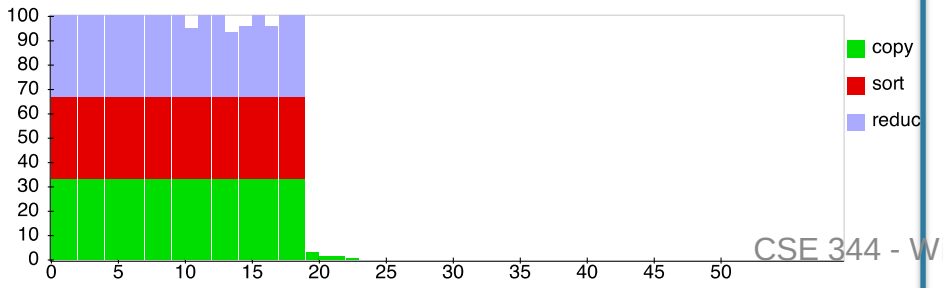
3h 51min

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	15816	0	0	15816	0	0 / 18
reduce	37.72%	50	19	22	9	0	0 / 0

ɔ Completion Graph - close



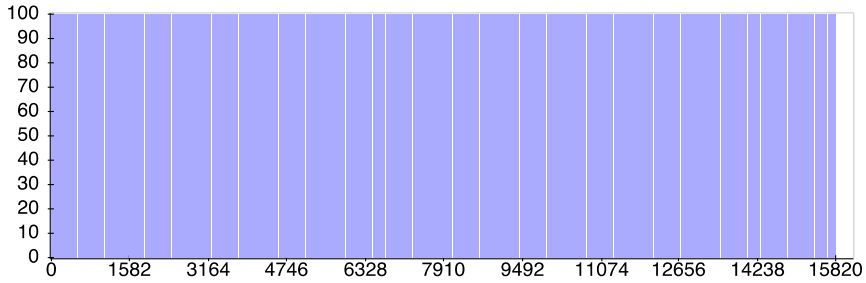
duce Completion Graph - close



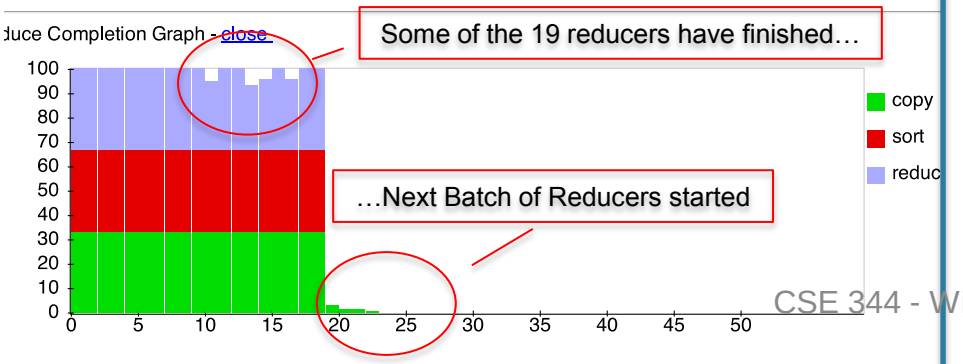
3h 51min

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	15816	0	0	15816	0	0 / 18
reduce	37.72%	50	19	22	9	0	0 / 0

ɔ Completion Graph - close



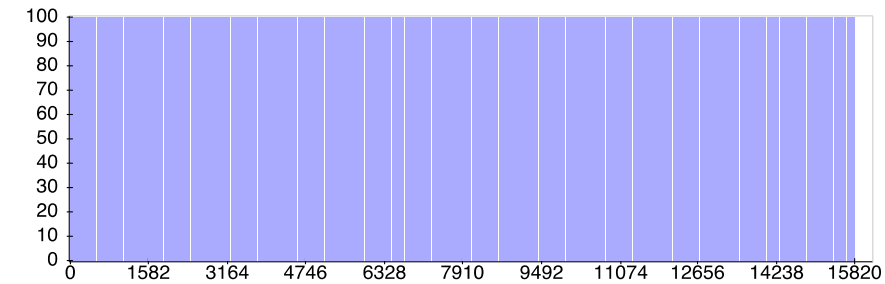
duce Completion Graph - close



3h 51min

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	15816	0	0	15816	0	0 / 18
reduce	37.72%	50	19	22	9	0	0 / 0

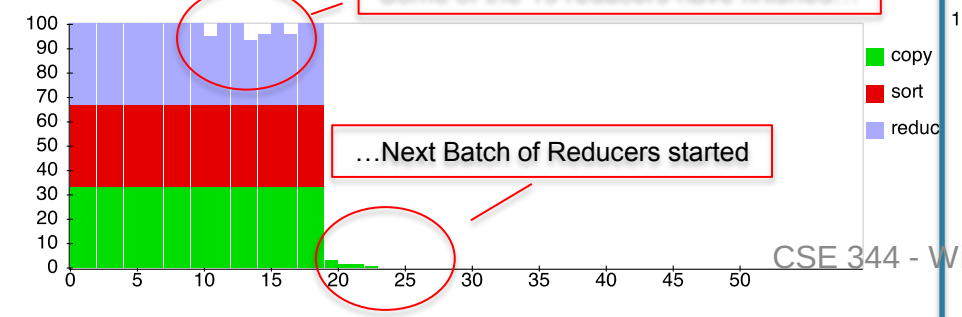
ɔ Completion Graph - close



duce Completion Graph - close

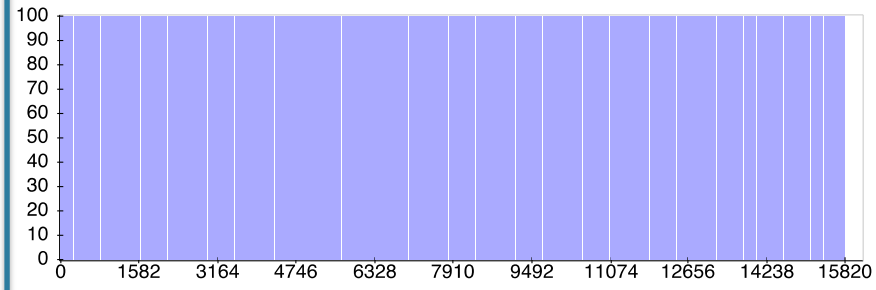
Some of the 19 reducers have finished…

…Next Batch of Reducers started



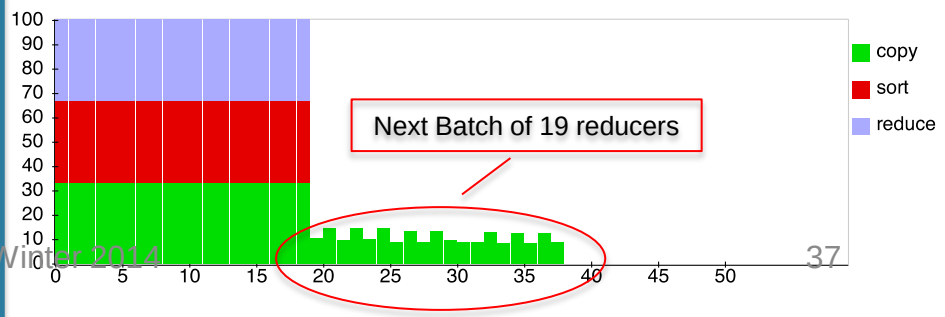
3h 52min

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	15816	0	0	15816	0	0 / 18
reduce	42.35%	50	11	20	19	0	0 / 0



ce Completion Graph - close

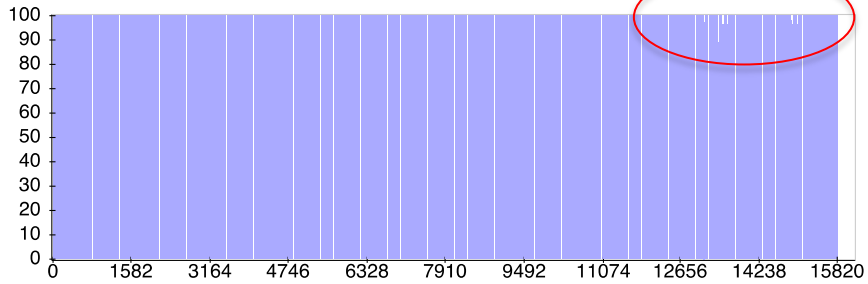
Next Batch of 19 reducers



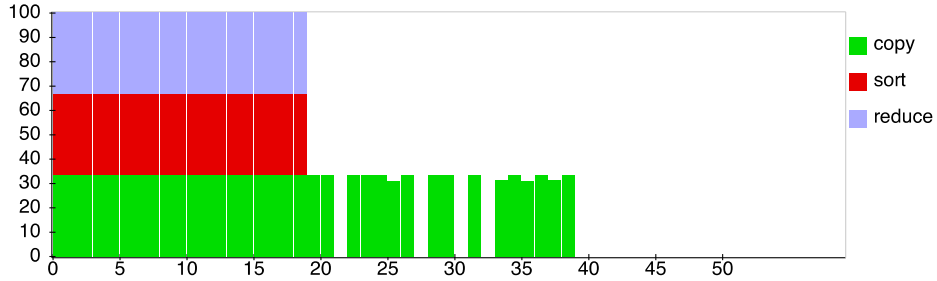
4h 18min

Several servers failed: “fetch error”. Their map tasks need to be rerun. All reducers are waiting….

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	99.88%	15816	2638	30	13148	0	15 / 3337
reduce	48.42%	50	15	16	19	0	0 / 0



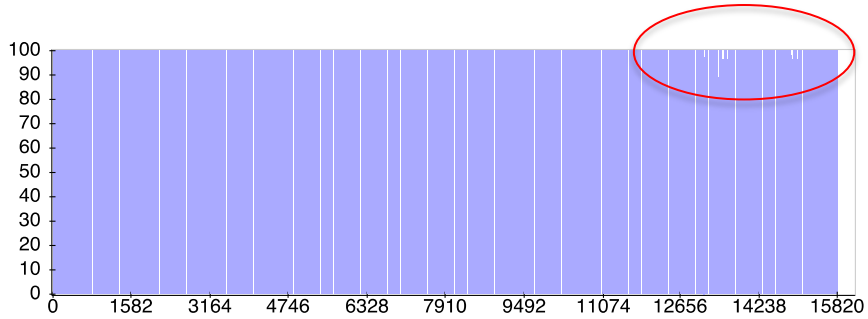
uce Completion Graph - close



4h 18min

Several servers failed: “fetch error”. Their map tasks need to be rerun. All reducers are waiting….

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	99.88%	15816	2638	30	13148	0	15 / 3337
reduce	48.42%	50	15	16	19	0	0 / 0

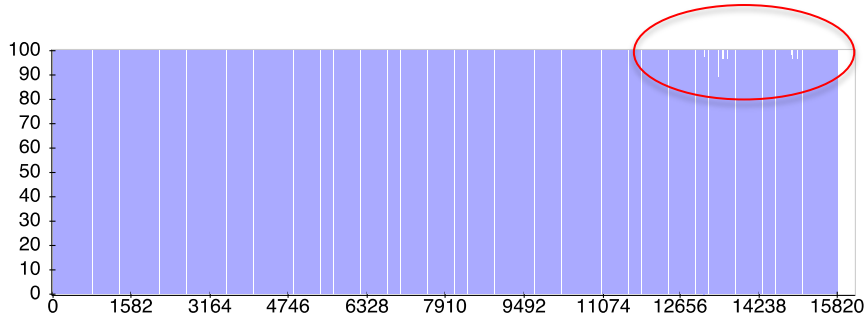


uce Completion Graph - close

4h 18min

Several servers failed: “fetch error”. Their map tasks need to be rerun. All reducers are waiting….

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	99.88%	15816	2638	30	13148	0	15 / 3337
reduce	48.42%	50	15	16	19	0	0 / 0

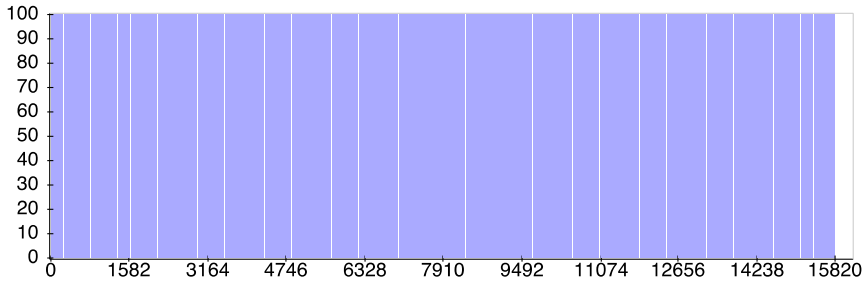


uce Completion Graph - close

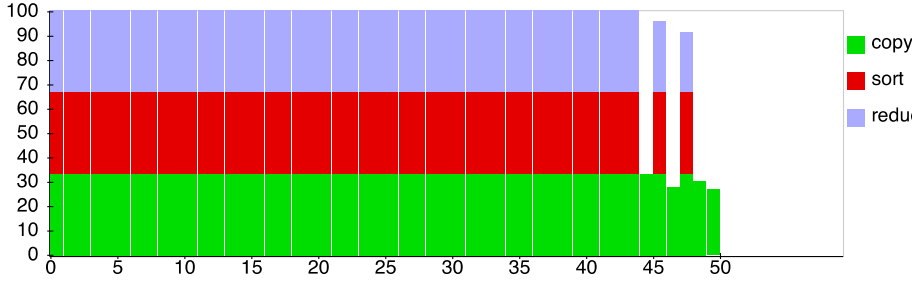
7h 10min

Mappers finished, reducers resumed.

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	15816	0	0	15816	0	26 / 5968
reduce	94.15%	50	0	6	44	0	0 / 8



uce Completion Graph - close



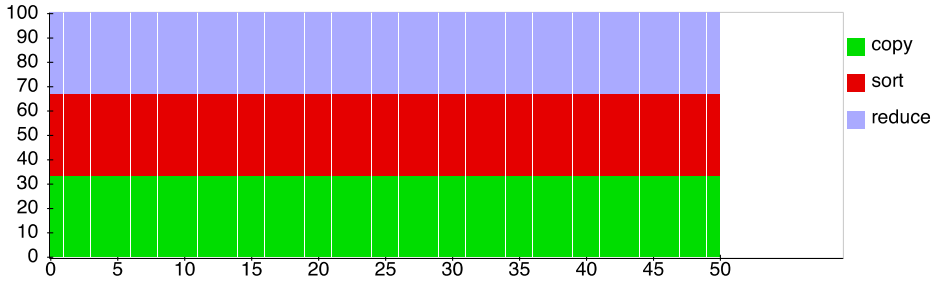
7h 20min

Success! 7hrs, 20mins.

Hadoop job_201203041905_0001 on ip-10-203-30-146

User: hadoop
Job Name: PigLatin:DefaultJobName
Job File: hdfs://10.203.30.146:9000/mnt/var/lib/hadoop/tmp/mapred/staging/hadoop/.staging/job_201203041905_0001/job.xml
Submit Host: ip-10-203-30-146.ec2.internal
Submit Host Address: 10.203.30.146
Job-ACLs: All users are allowed
Job Setup: Successful
Status: Succeeded
Started at: Sun Mar 04 19:08:29 UTC 2012
Finished at: Mon Mar 05 02:28:39 UTC 2012
Finished in: 7hrs, 20mins, 10sec
Job Cleanup: Successful
Black-listed TaskTrackers: 3

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	15816	0	0	15816	0	26 / 5968
reduce	100.00%	50	0	0	50	0	0 / 14



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

(will not discuss in class; please read 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 HW8, 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]

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

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

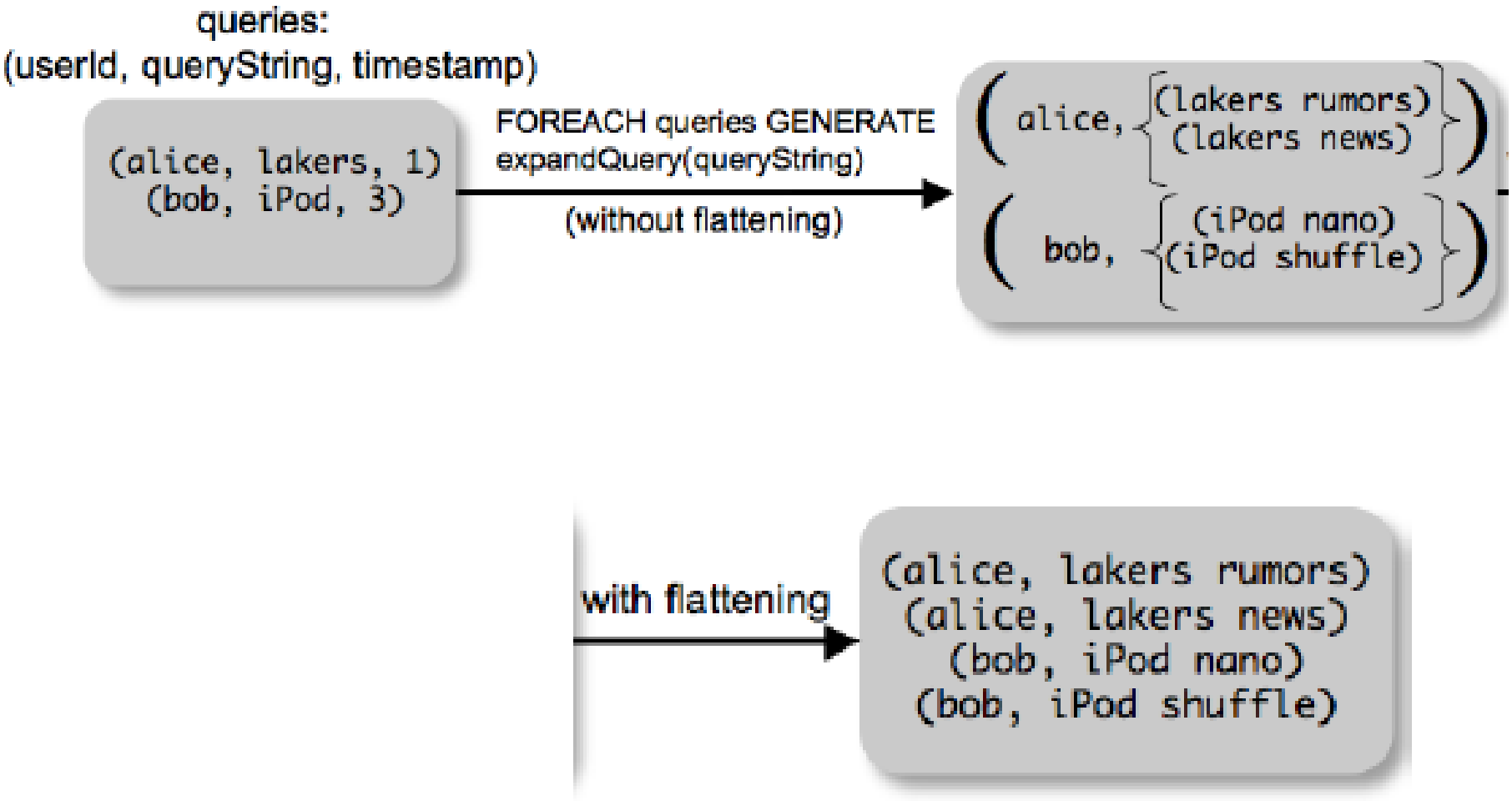
[Olston'2008]

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 I don't like 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]

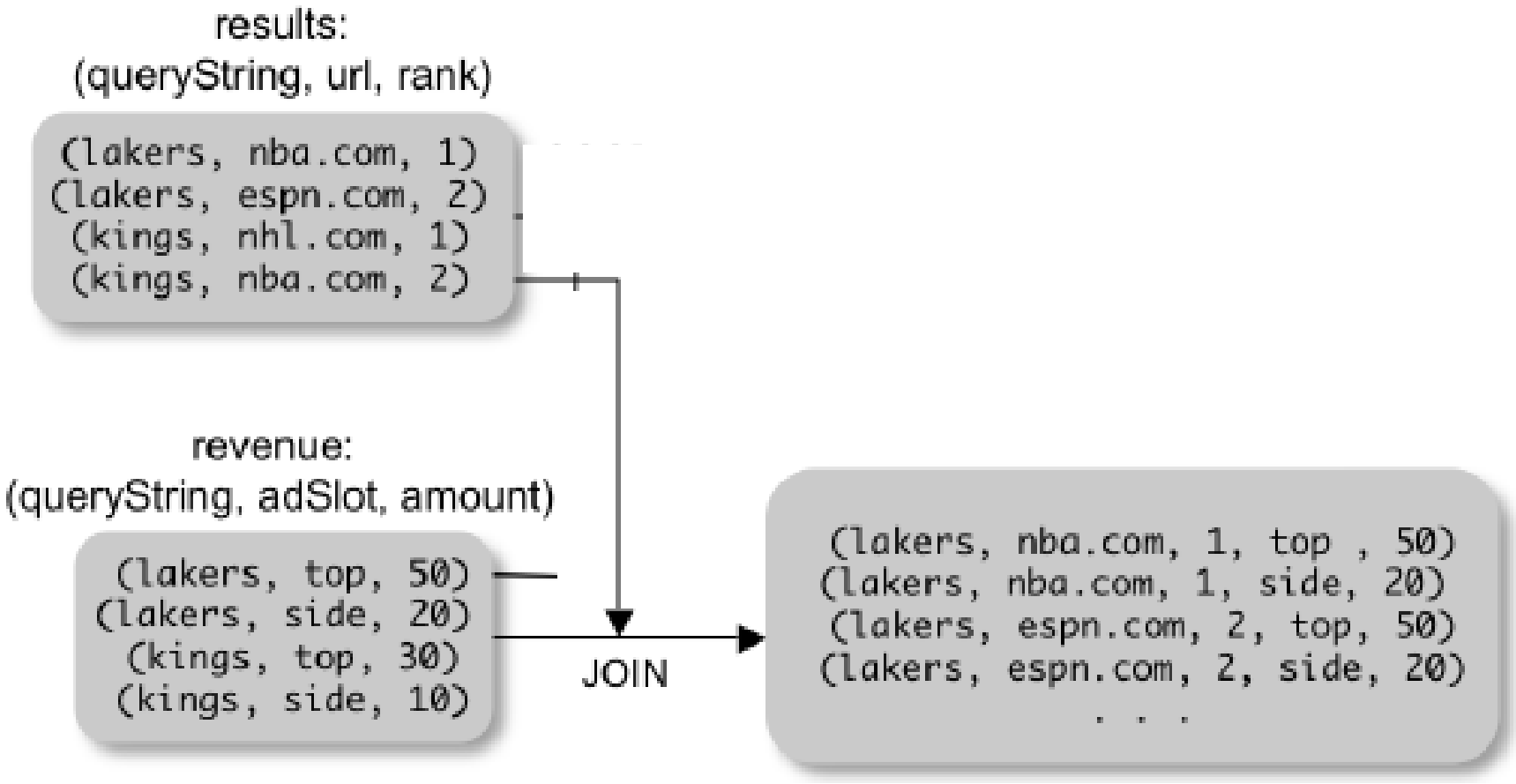
JOIN

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)}

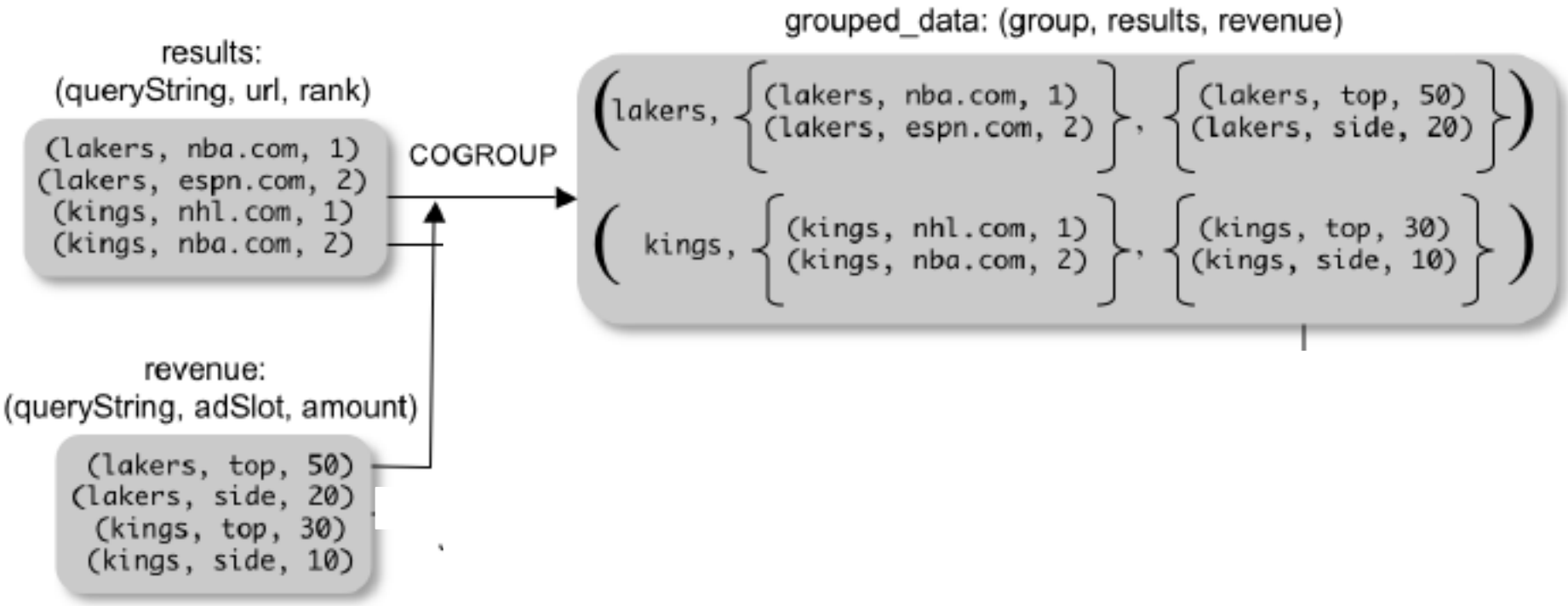
```
grouped_data =
      COGROUP results BY queryString,
              revenue BY queryString;
```

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 results 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