Introduction to Data Management CSE 344

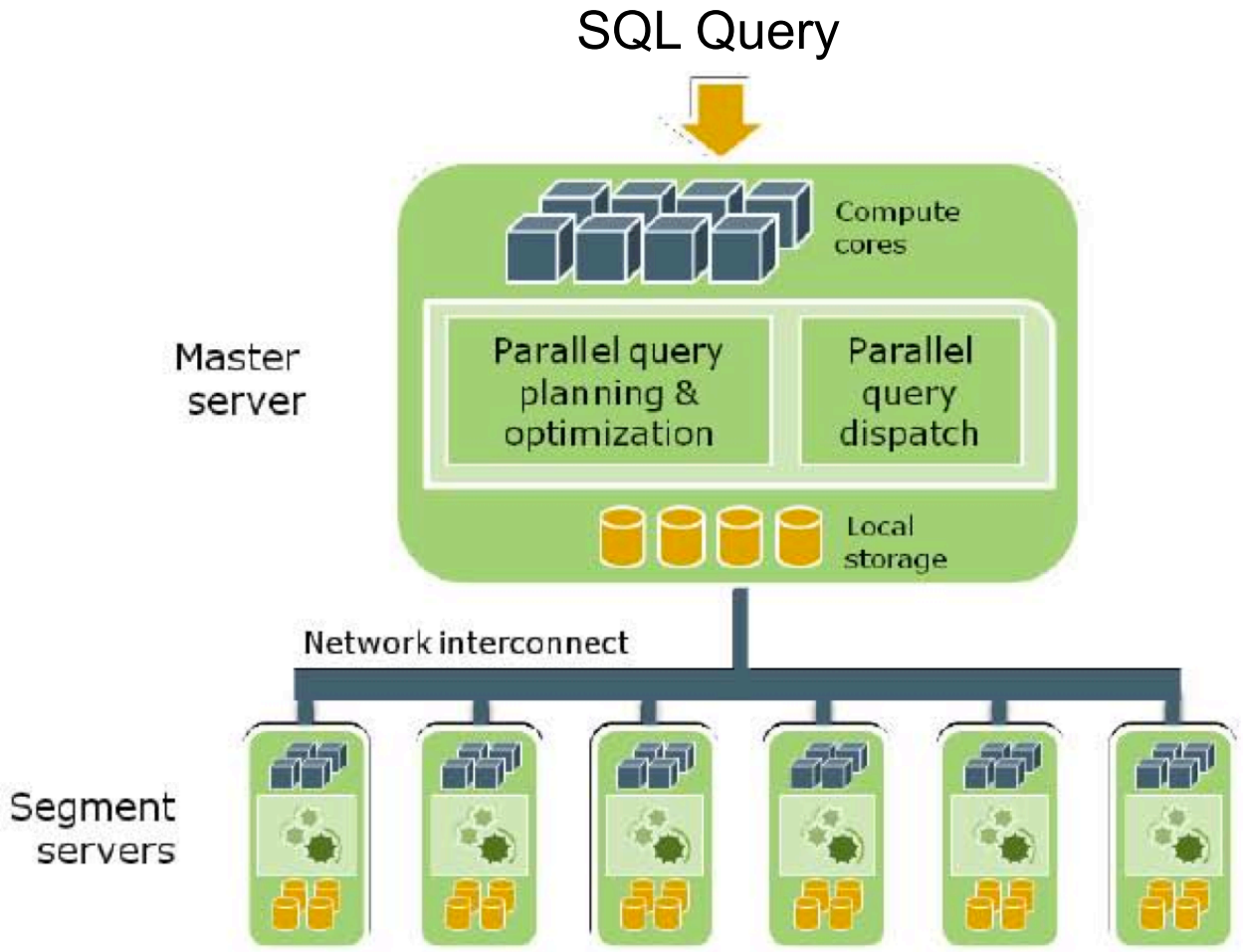
Lecture 22: MapReduce

Where We Are

- We are talking about parallel query processing
- There exist two main types of engines:
  - Parallel DBMSs (last lecture)
  - MapReduce and similar systems (this lecture)

Review: Parallel DBMS

Figure 5 - Master server performs global planning and dispatch



From: Greenplum Database Whitepaper

Parallel DBMS

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

Cluster Computing

Cluster Computing

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

READING ASSIGNMENT:
Map-reduce (Section 20.2); Chapter 2 (Sections 1,2,3 only) of Mining of Massive Datasets, by Rajaraman and Ullman
See the course Calendar Website

Cluster Computing

- 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

Map Reduce

- Google: paper published 2004
- Free variant: Hadoop

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

Data Model

Files !

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

A map-reduce 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

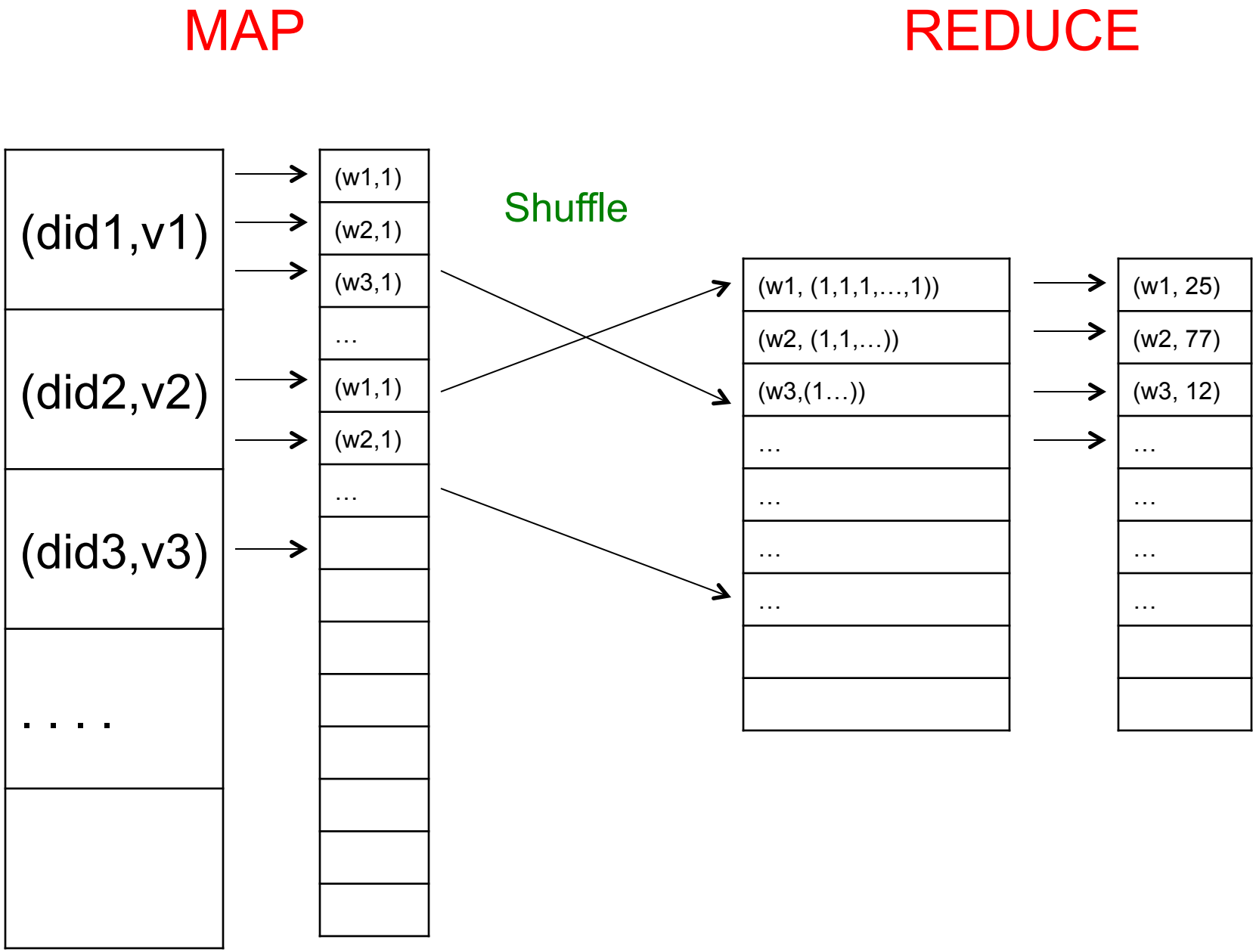
Doc(did, word)

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)
…
…
…
…

MAP = GROUP BY,
REDUCE = Aggregate

Doc(did, word)

```
SELECT word, sum(1)
FROM Doc
GROUP BY word
```

Example 2: MR word length count

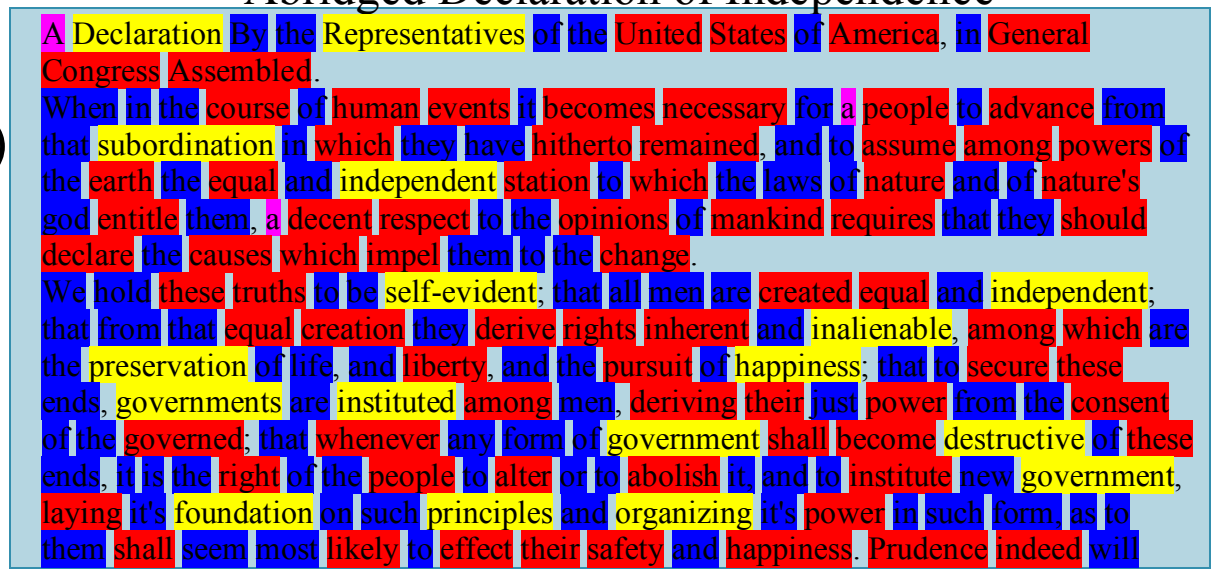
Abridged Declaration of Independence

A Declaration By the Representatives of the United States of America, in General Congress Assembled.
When in the course of human events it becomes necessary for a people to advance from that subordination in which they have hitherto remained, and to assume among powers of the earth the equal and independent station to which the laws of nature and of nature's god entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the change.
We hold these truths to be self-evident; that all men are created equal and independent; that from that equal creation they derive rights inherent and inalienable, among which are the preservation of life, and liberty, and the pursuit of happiness; that to secure these ends, governments are instituted among men, deriving their just power from the consent of the governed; that whenever any form of government shall become destructive of these ends, it is the right of the people to alter or to abolish it, and to institute new government, laying it's foundation on such principles and organizing it's power in such form, as to them shall seem most likely to effect their safety and happiness. Prudence indeed will dictate that governments long established should not be changed for light and transient causes: and accordingly all experience hath shewn that mankind are more disposed to suffer while evils are sufferable, than to right themselves by abolishing the forms to which they are accustomed. But when a long train of abuses and usurpations, begun at a distinguished period, and pursuing invariably the same object, evinces a design to reduce them to arbitrary power, it is their right, it is their duty, to throw off such government and to provide new guards for future security. Such has been the patient sufferings of the colonies; and such is now the necessity which constrains them to expunge their former systems of government. the history of his present majesty is a history of unremitting injuries and usurpations, among which no one fact stands single or solitary to contradict the uniform tenor of the rest, all of which have in direct object the establishment of an absolute tyranny over these states. To prove this, let facts be submitted to a candid world, for the truth of which we pledge a faith yet unsullied by falsehood.

Example 2: MR word length count

Abridged Declaration of Independence

Map Task 1
(204 words)



(key, value)

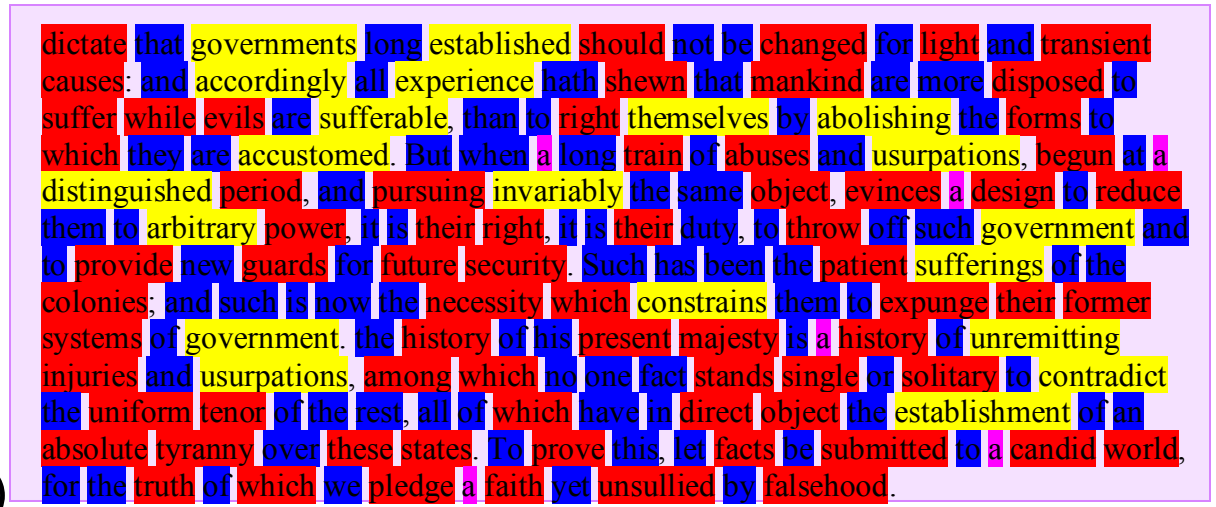
(yellow, 17)
(red, 77)
(blue, 107)
(pink, 3)

Yellow: 10+

Red: 5..9

Blue: 2..4

Pink: = 1

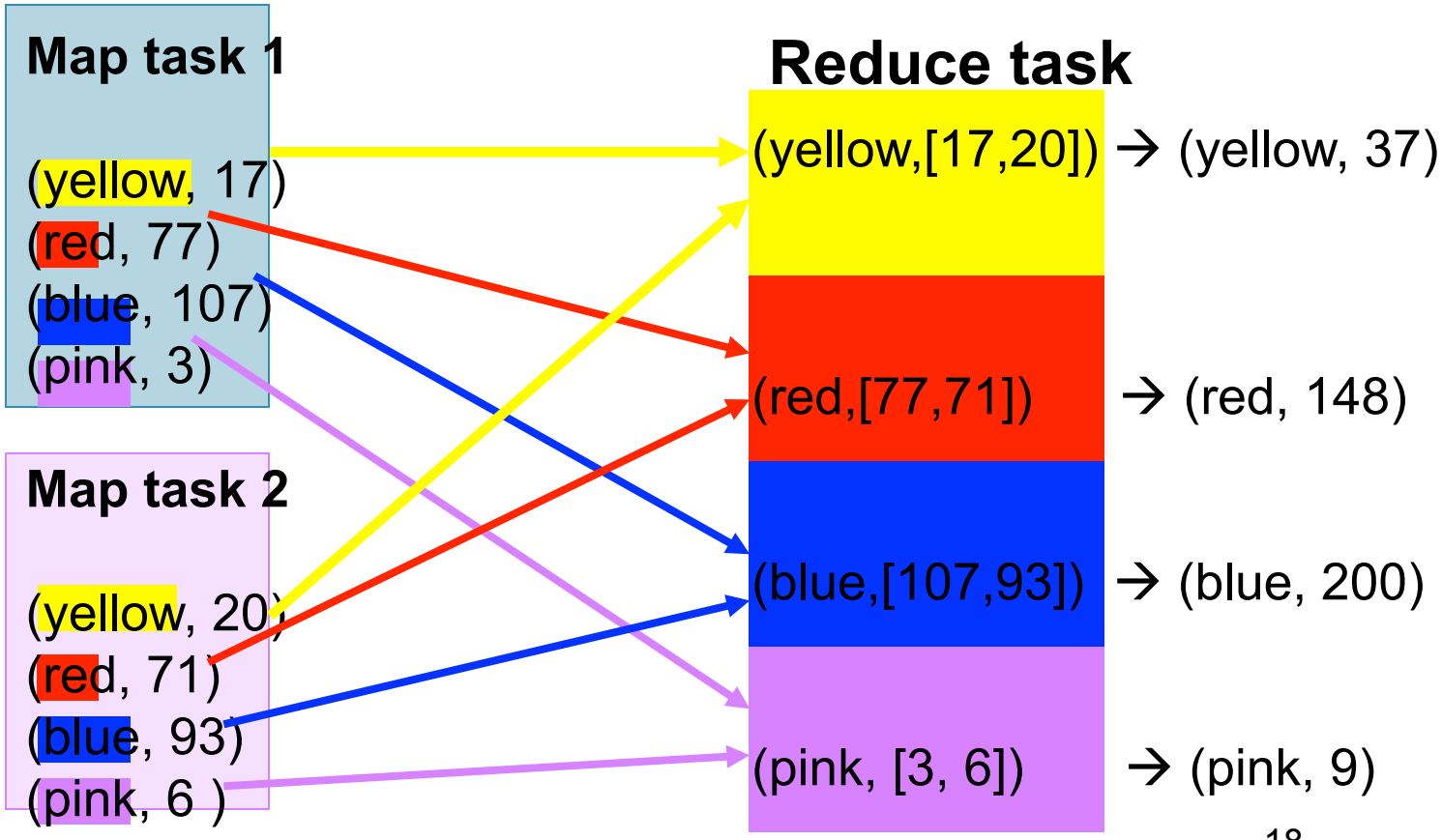


(yellow, 20)
(red, 71)
(blue, 93)
(pink, 6)

Map Task 2
(190 words)

Example 2: MR word length count

Map is a **GROUP BY** operation
Reduce is an **AGGREGATE** operation



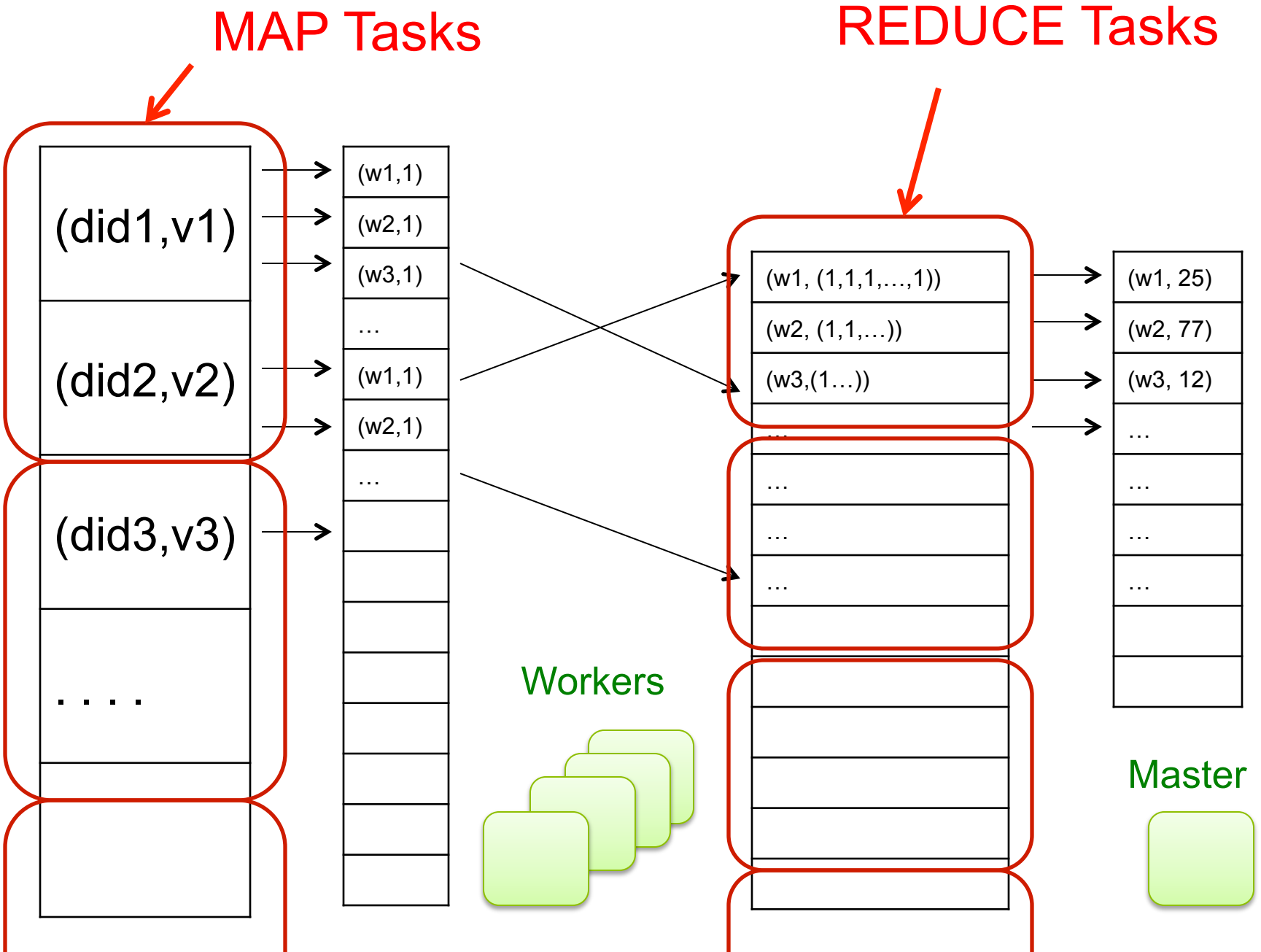
Jobs v.s. Tasks

- A Map-Reduce 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

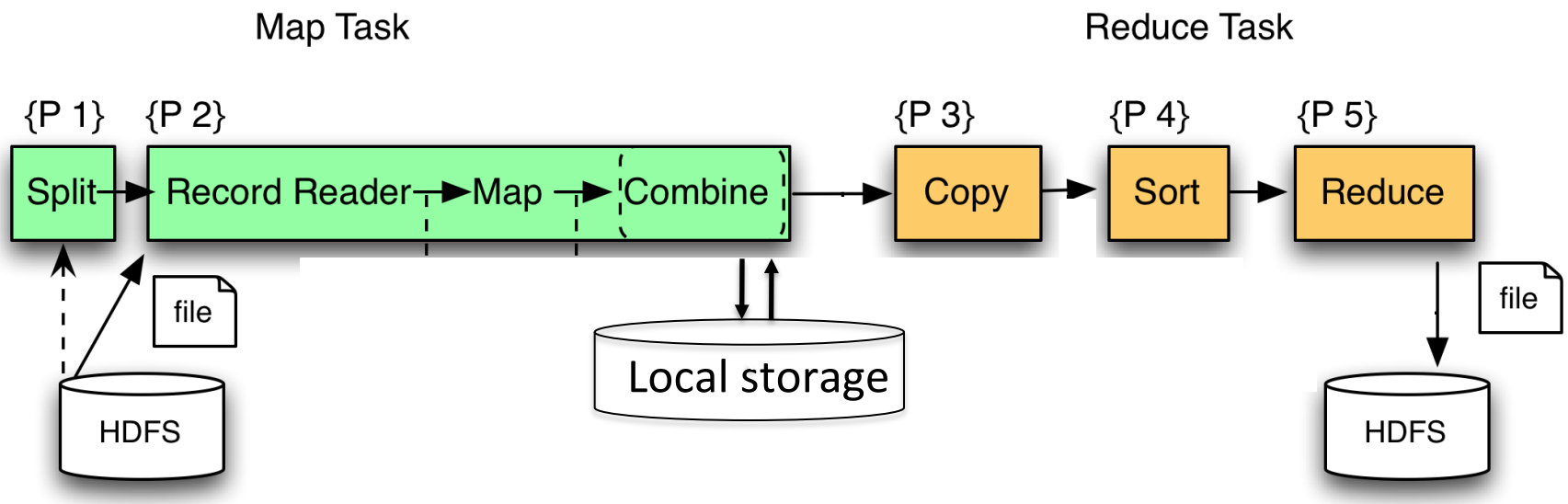
Workers

- 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
(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)
…
…
…
…
Workers
Master

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*
- 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*

Map-Reduce Summary

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

Declarative Languages on MR

- PIG Latin (Yahoo!)
  - New language, like Relational Algebra
  - Open source
- SQL / Tenzing (google)
  - SQL on MR
  - Proprietary
- Others (won't discuss):
  - Scope (MS): SQL; proprietary
  - DryadLINQ (MS): LINQ; proprietary
  - Clustera (other UW) : SQL; Not publicly available

Parallel DBMS vs MapReduce

- Parallel DBMS
  - Indexing
  - Physical tuning
  - Can stream data from one operator to the next without blocking
- MapReduce
  - 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!

HW6

- We will use MapReduce (Hadoop)
- We will use a declarative language: Pig Latin
- Cluster will run in Amazon's cloud
  - Give your credit card
  - Click, click, click… and you have a MapReduce cluster running all configured for you
- We will analyze a real 0.5TB graph

Pig Latin Mini-Tutorial

(will discuss only briefly in class;
please read in order to do homework 6)

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
  - Option 2: compile into sequence of map/reduce, run on a cluster supporting Hadoop
  - In HW6 we use only option 2

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)
```

[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

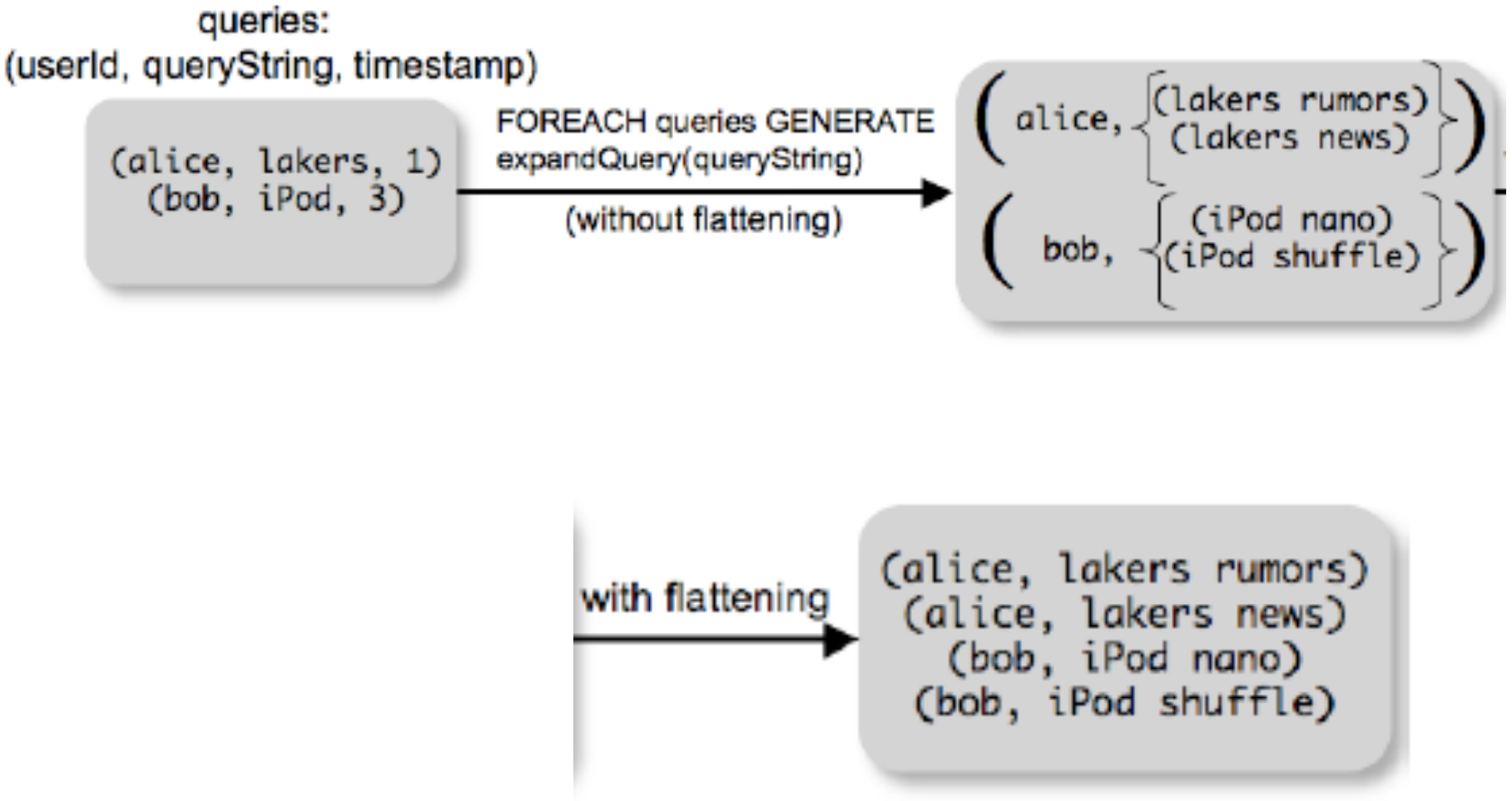
[Olston'2008]

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]

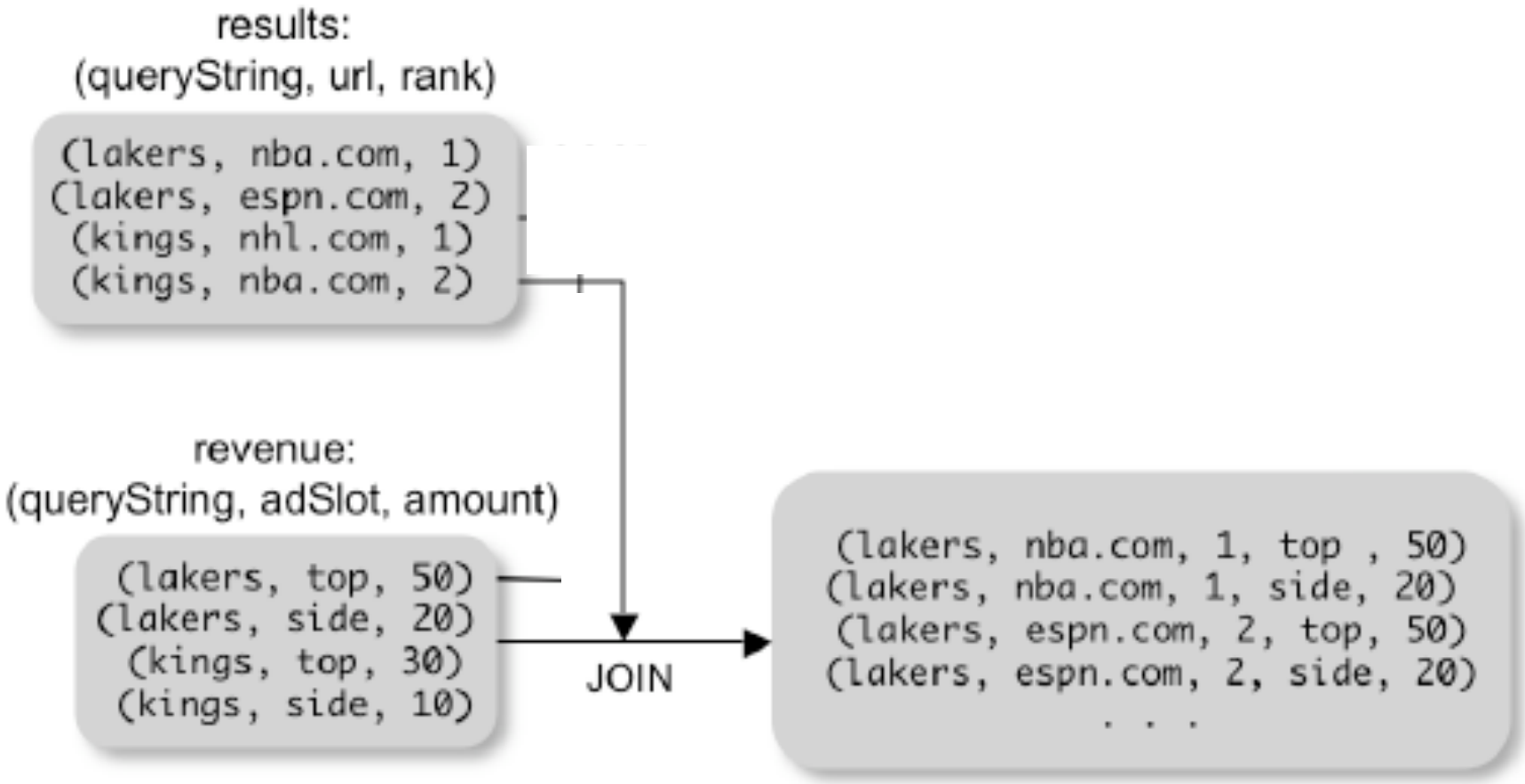
JOIN

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 Map-Reduce

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

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

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

[Olston'2008]

Co-Group

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

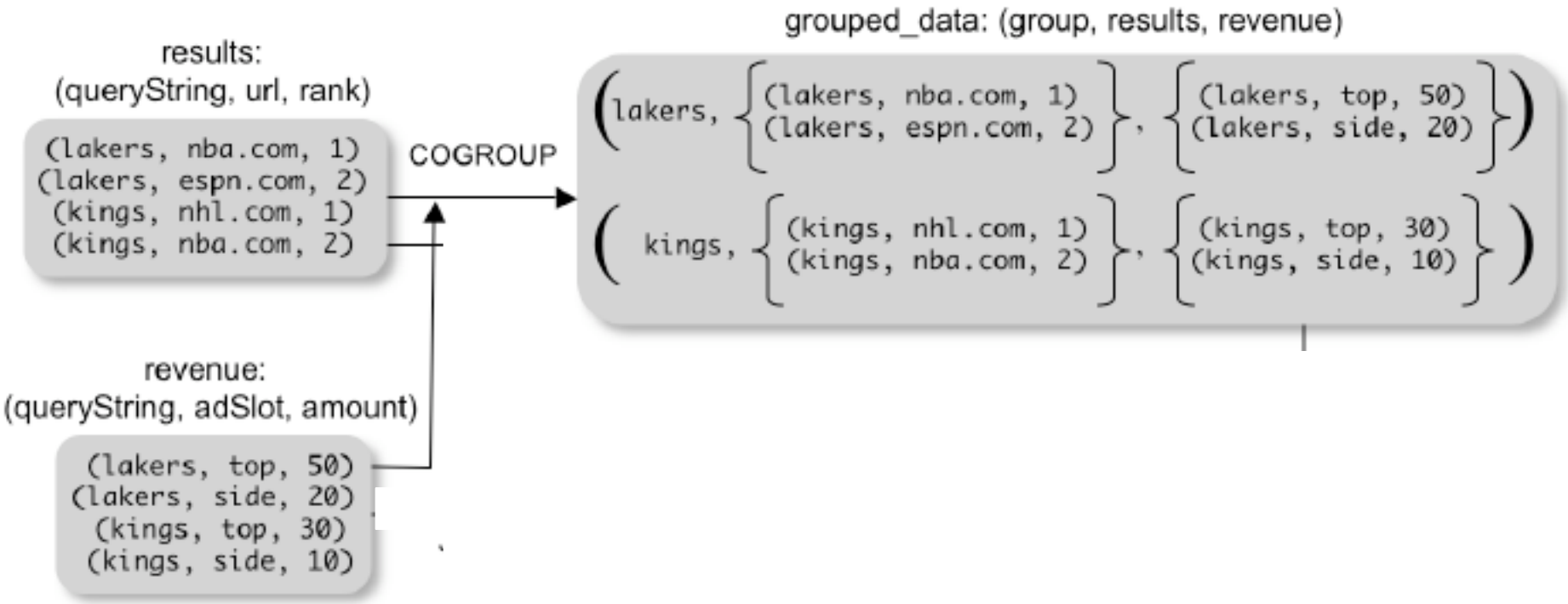
```
grouped_data =
      COGROUP results BY queryString,
              revenue BY queryString;
```

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 → sequence of Map/Reduce ops
- All statements between two (CO)GROUPs → one Map/Reduce op

[Olston'2008]

Implementation