Introduction to Data Management CSE 344

Lecture 27: Map Reduce,
slides on Pig Latin

Announcements

- HW8 due on Friday
  - Try to make lots of progress over weekend
- Final exam:
  - 3/20, 8:30-10:20, this room
  - Comprehensive
  - Open books, open notes, closed computers
- Review session:
  - Saturday, 3/16, 10am, 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 not discuss 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

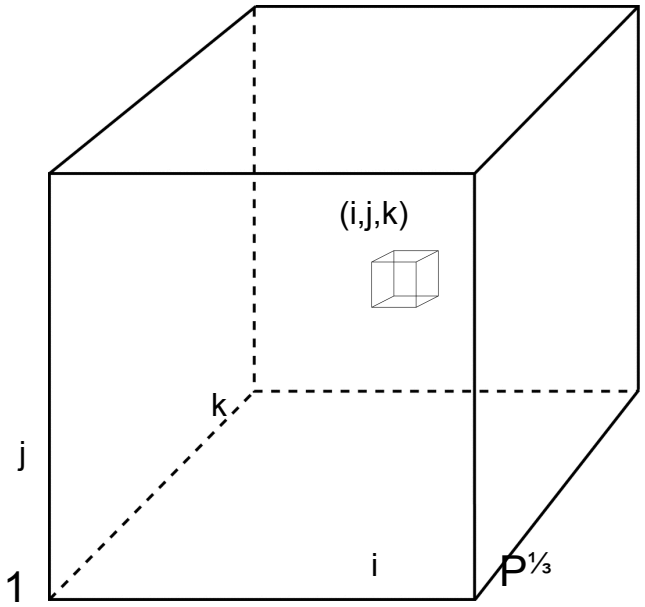
- Have P servers (say P=27 or P=1000)
- How do we compute this query?
  Q(x,y,z) = R(x,y),S(y,z),T(z,x)
- Step 1:
  - Each server sends R(x,y) to server h(y) mod P
  - Each server sends S(y,z) to server h(y) mod P
- Step 2:
  - Each server computes R⋈S locally
  - Each server sends [R(x,y),S(y,z)] to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

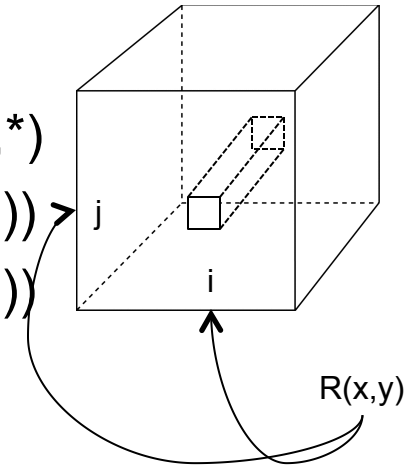
A Challenge

- 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≤$P^{1/3}$



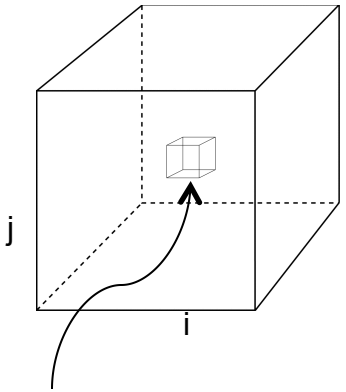
A Challenge

- 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≤$P^{1/3}$
- 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))



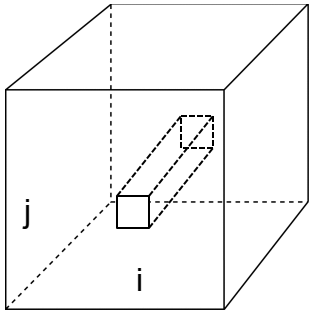
A Challenge

- 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≤$P^{1/3}$
- 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≤$P^{1/3}$
- 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)

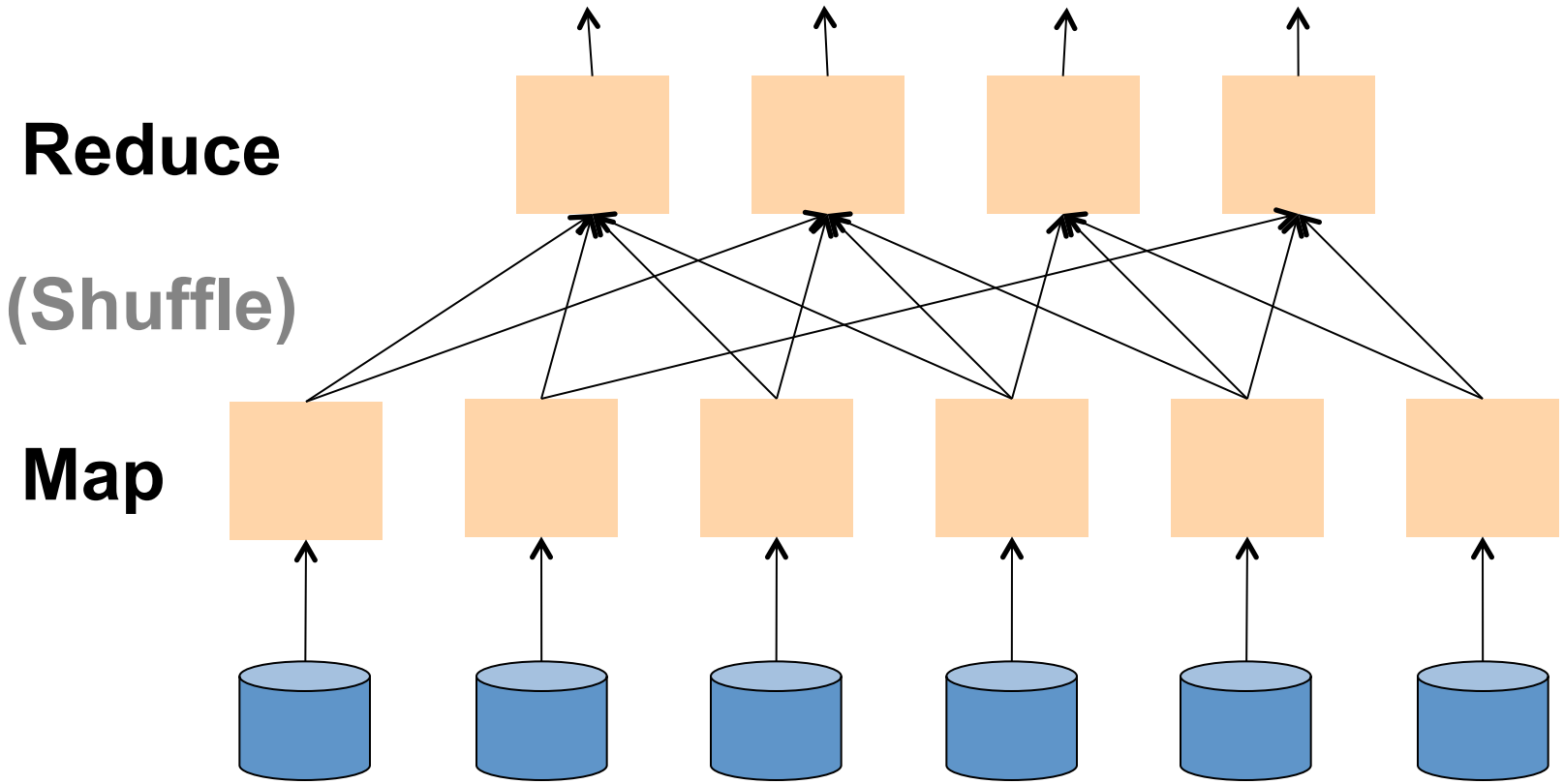
- For very large files: TBs, PBs
- Each file is partitioned into *chunks*, typically 64MB
- Each chunk is replicated several times (≥3), on different racks, for fault tolerance
- Implementations:
  - Google's DFS: GFS, proprietary
  - Hadoop's DFS: HDFS, open source

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)`**

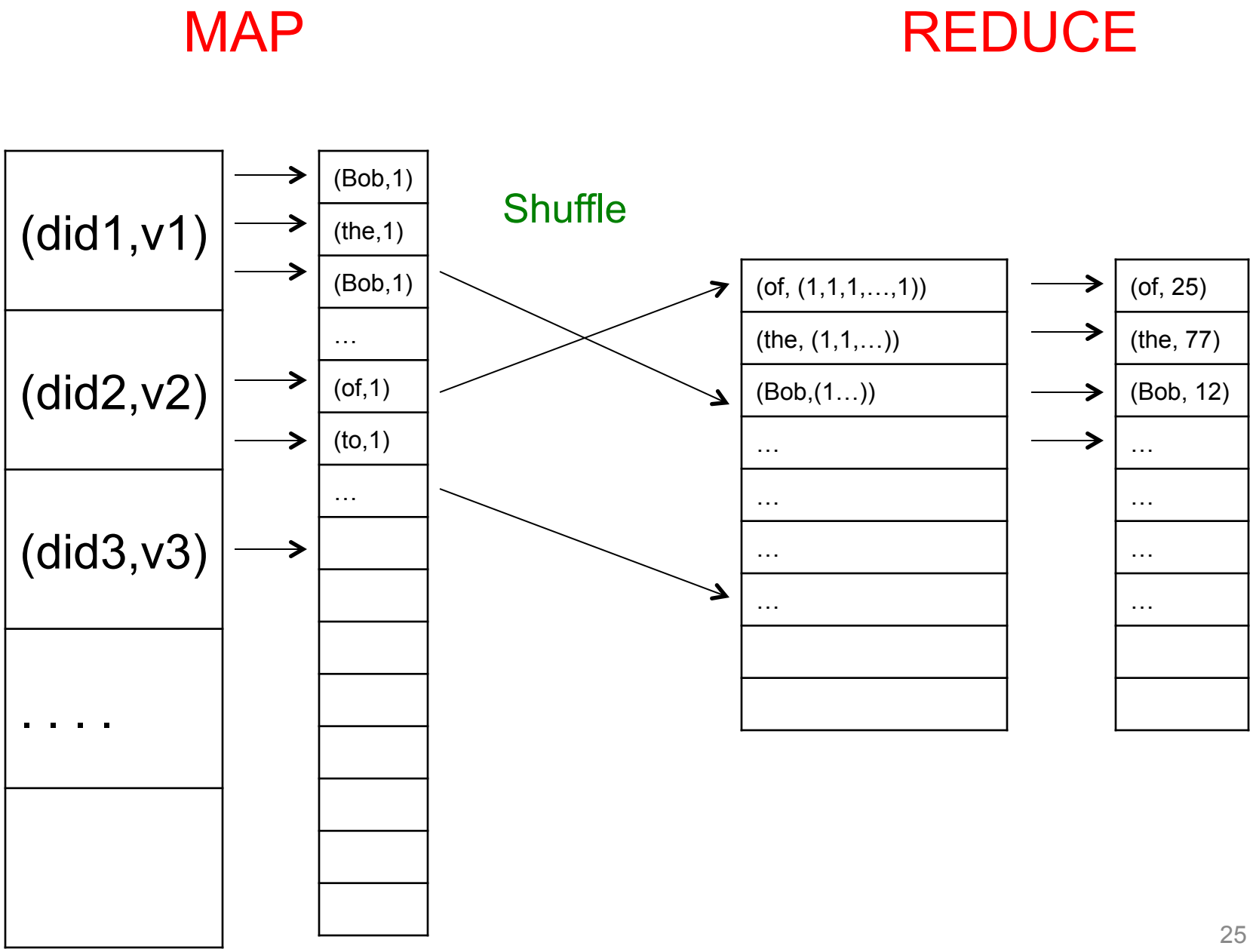
System groups all pairs with the same intermediate key, and passes the bag of values to the REDUCE function

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

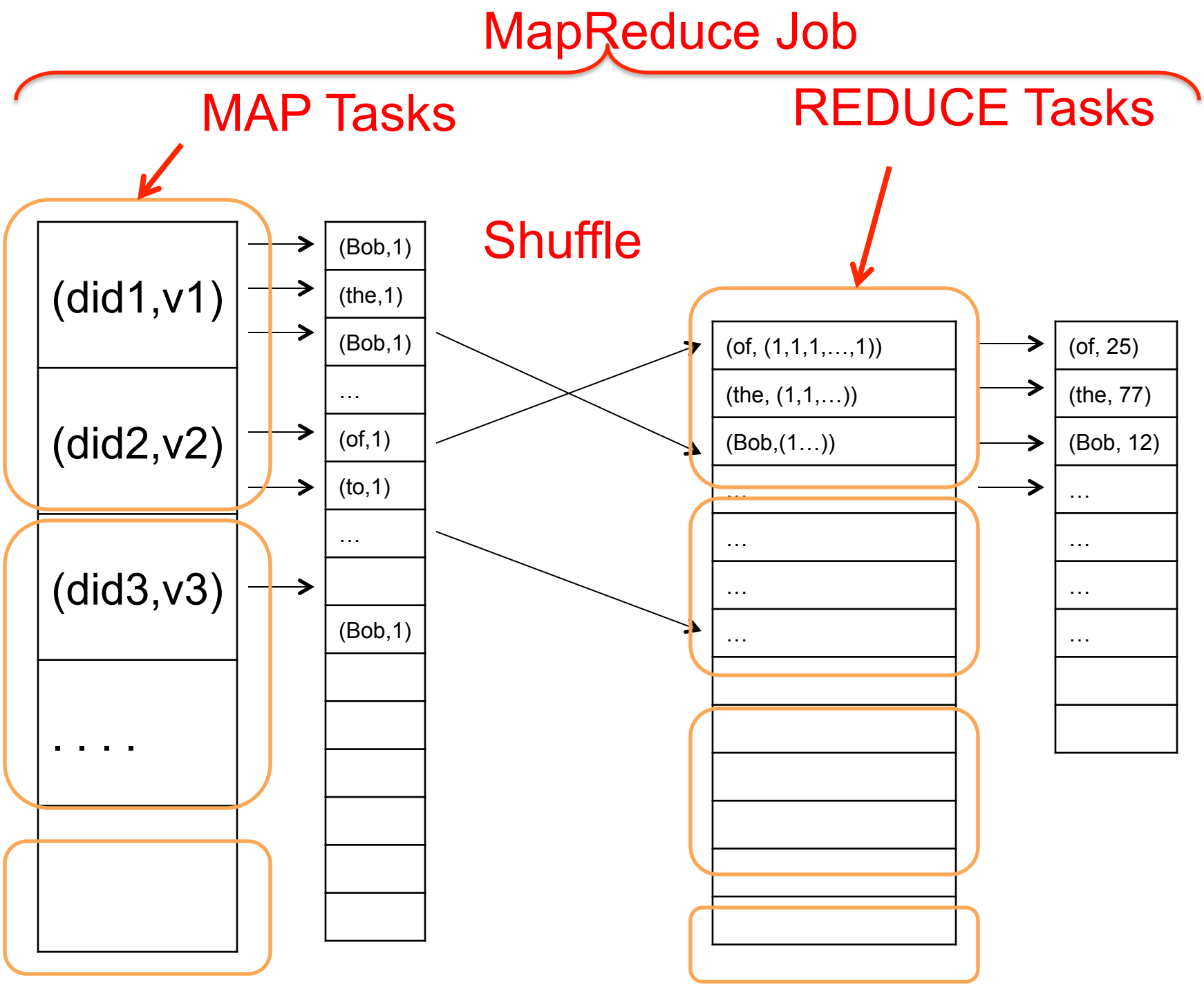
Jobs v.s. Tasks

- A MapReduce Job
  - One single "query", e.g. count the words in all docs
  - More complex queries may consists of multiple jobs

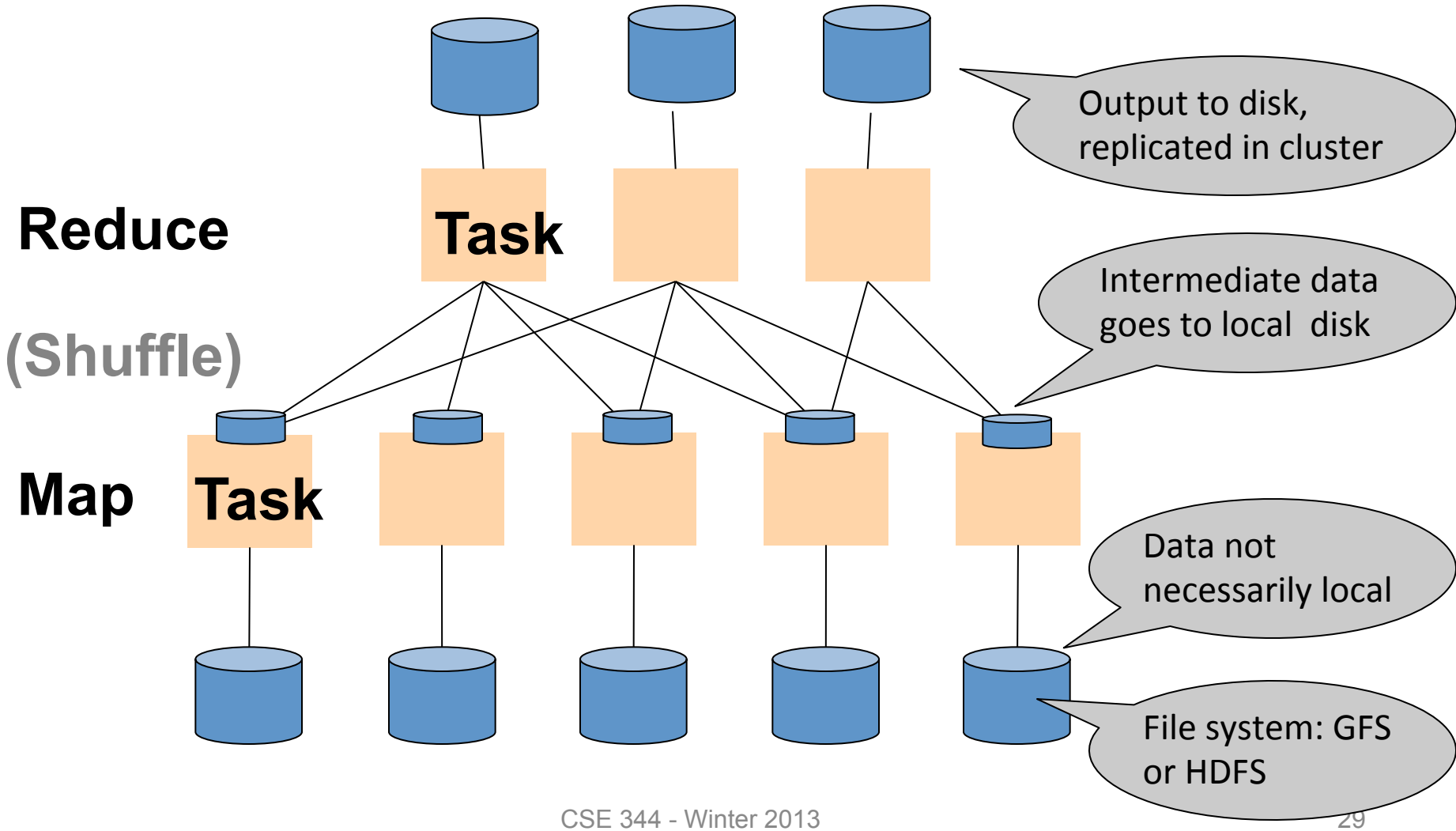
- A Map Task, or a Reduce Task
  - A group of instantiations of the map-, or reduce-function, which are scheduled on a single worker

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

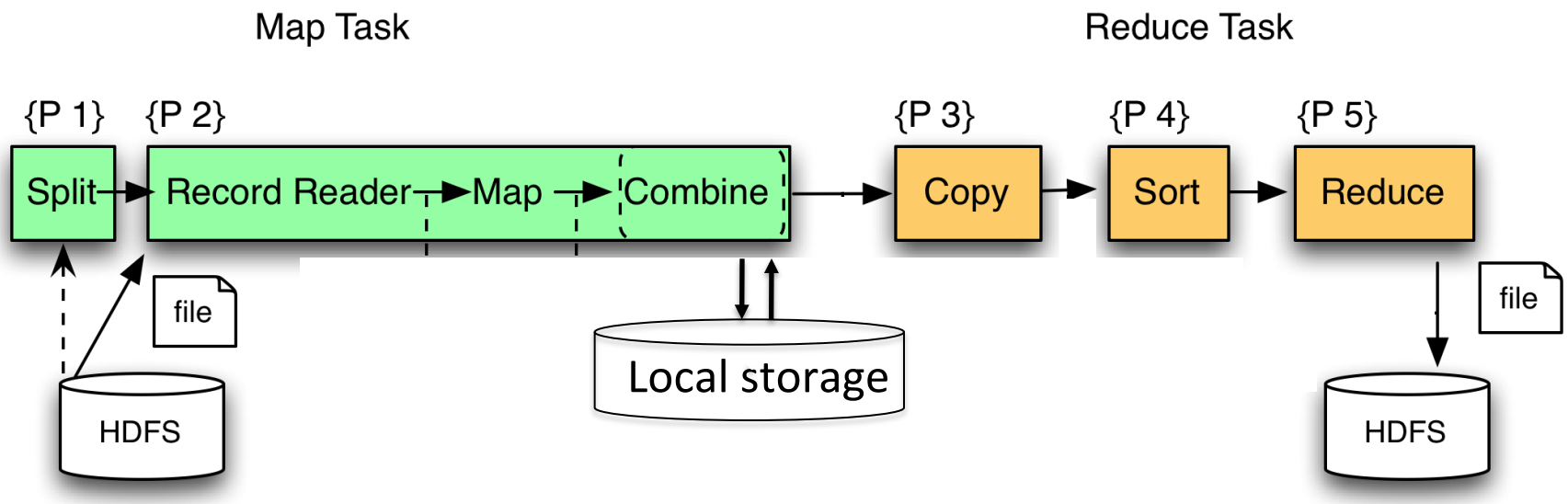

MapReduce Job
MAP Tasks
REDUCE Tasks
Shuffle
(did1,v1)
(did2,v2)
(did3,v3)
. . . .
(Bob,1)
(the,1)
(Bob,1)
…
(of,1)
(to,1)
…
(Bob,1)
(of, (1,1,1,…,1))
(the, (1,1,…))
(Bob,(1…))
…
…
…
…
(of, 25)
(the, 77)
(Bob, 12)
…
…
…
…

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

(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 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

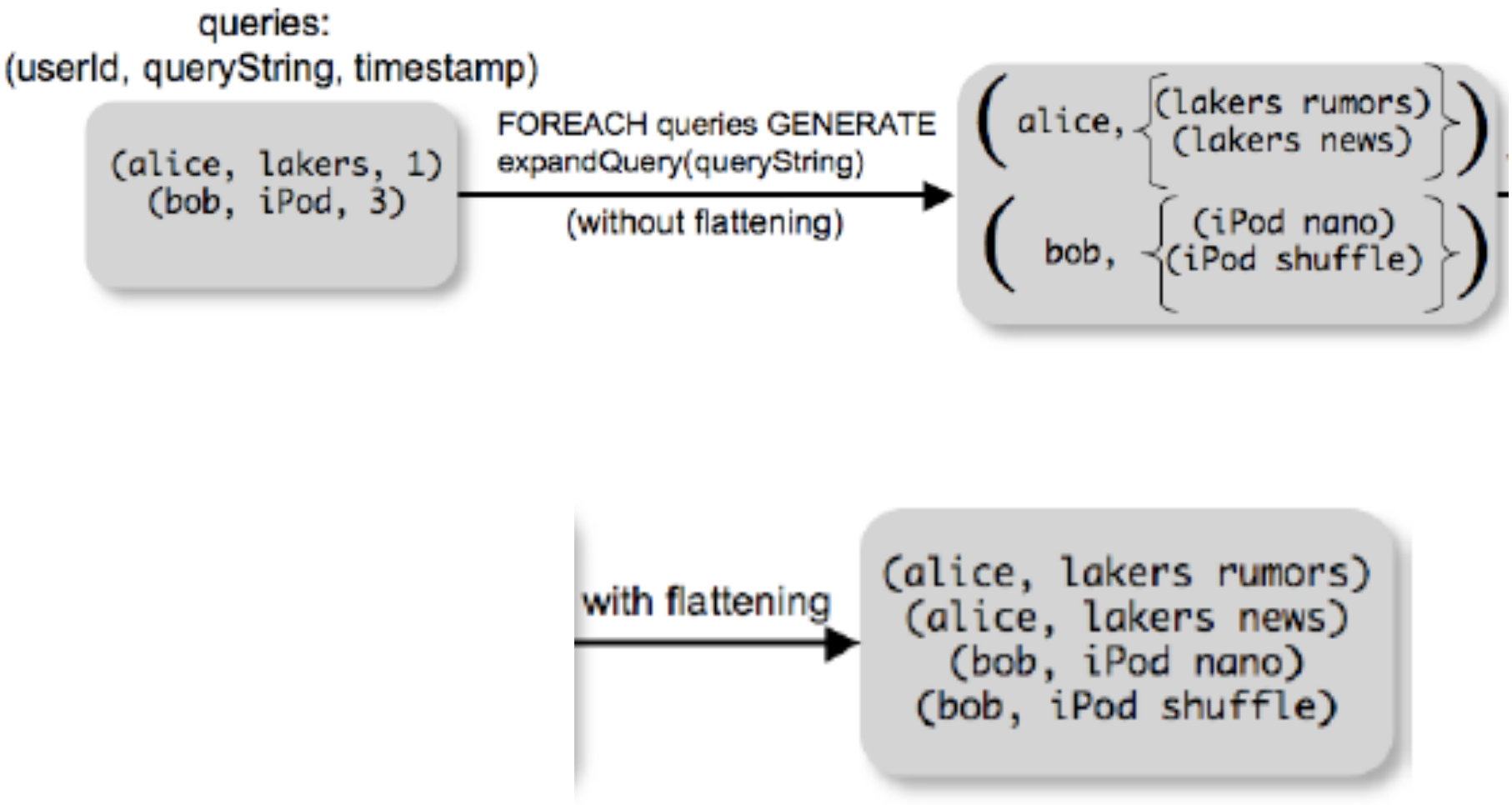
[Olston'2008]

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 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]

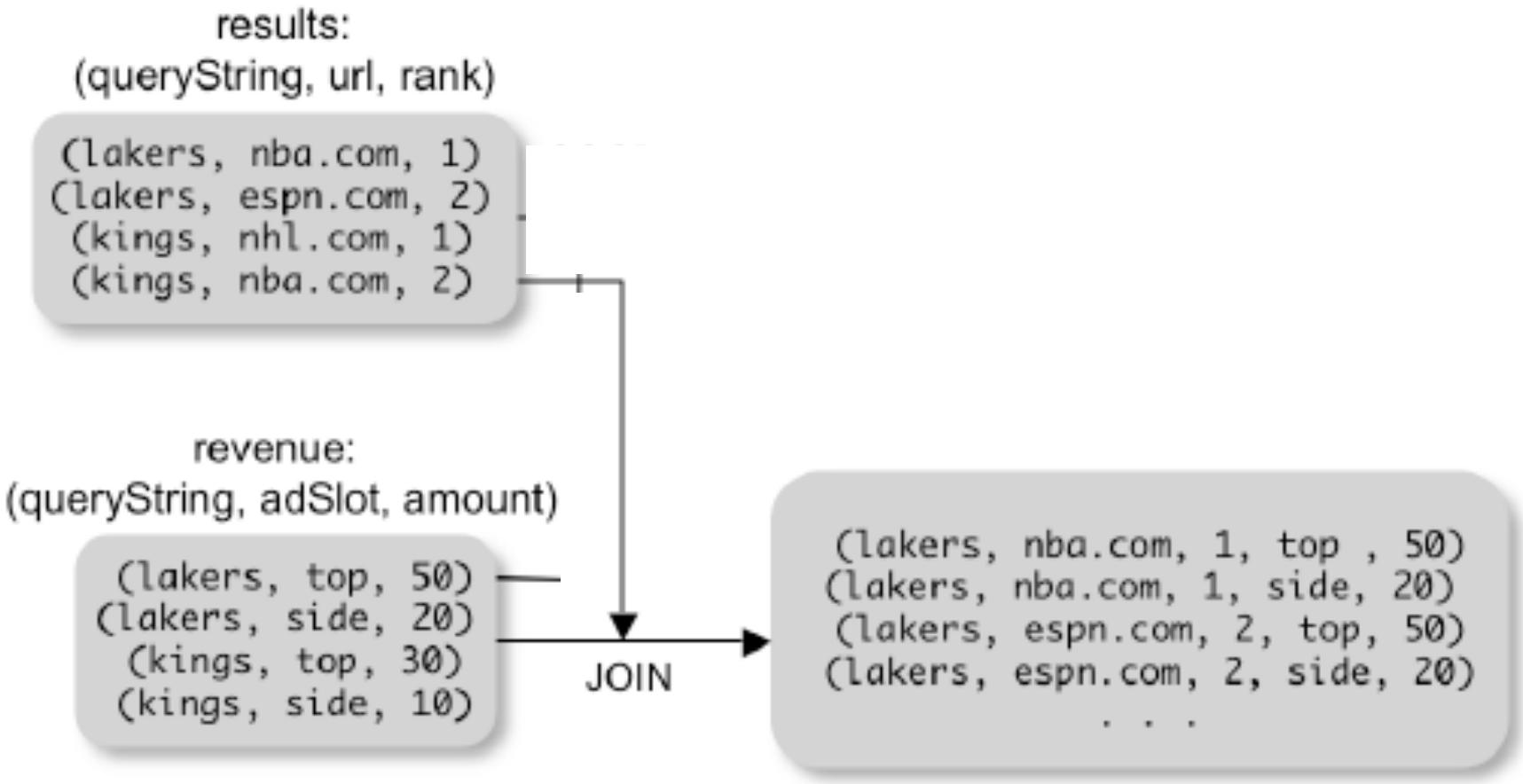
JOIN

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)}

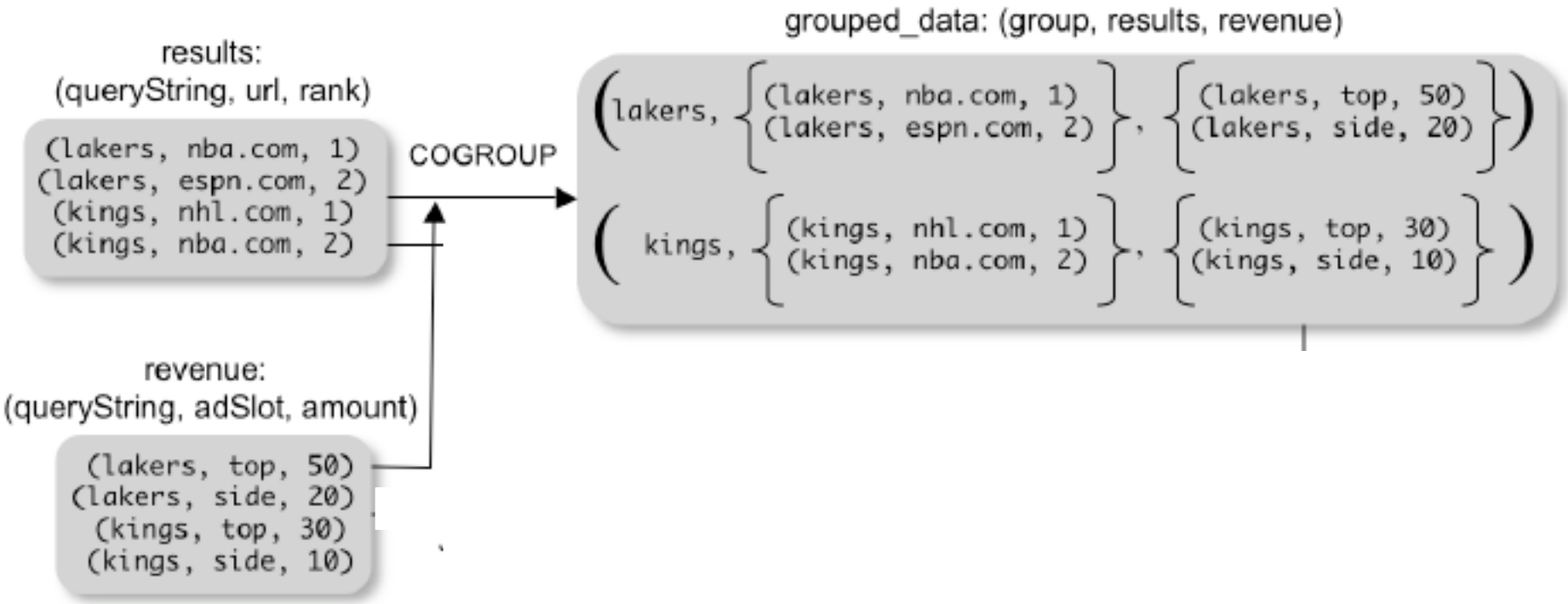
```
grouped_data =
      COGROUP results BY queryString,
              revenue BY queryString;
```

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'

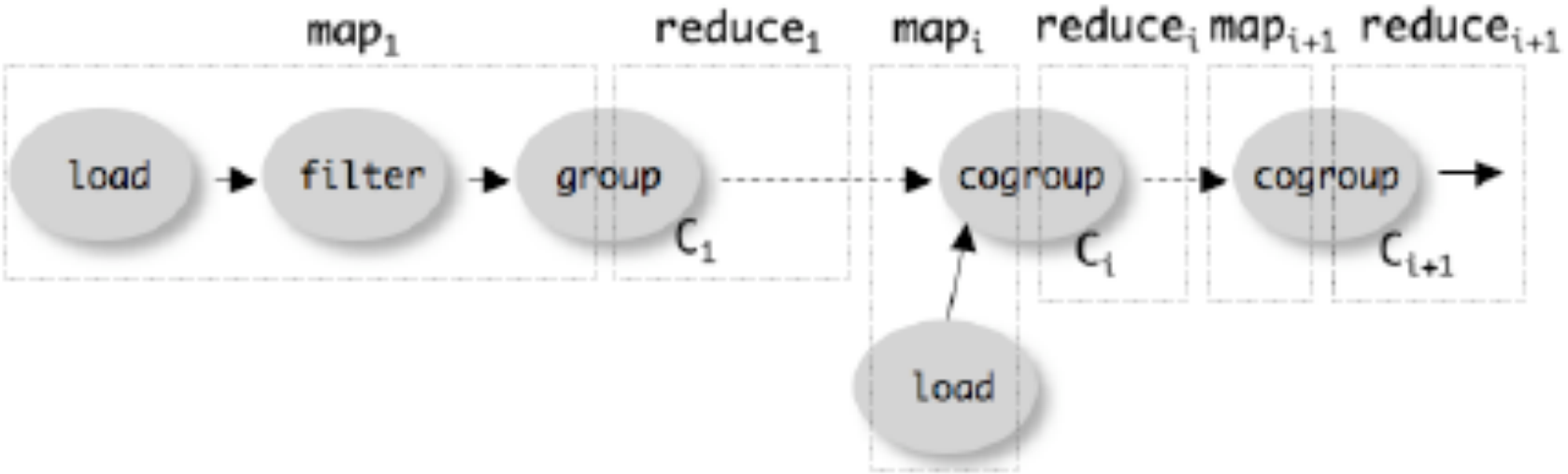
[Olston'2008]

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]

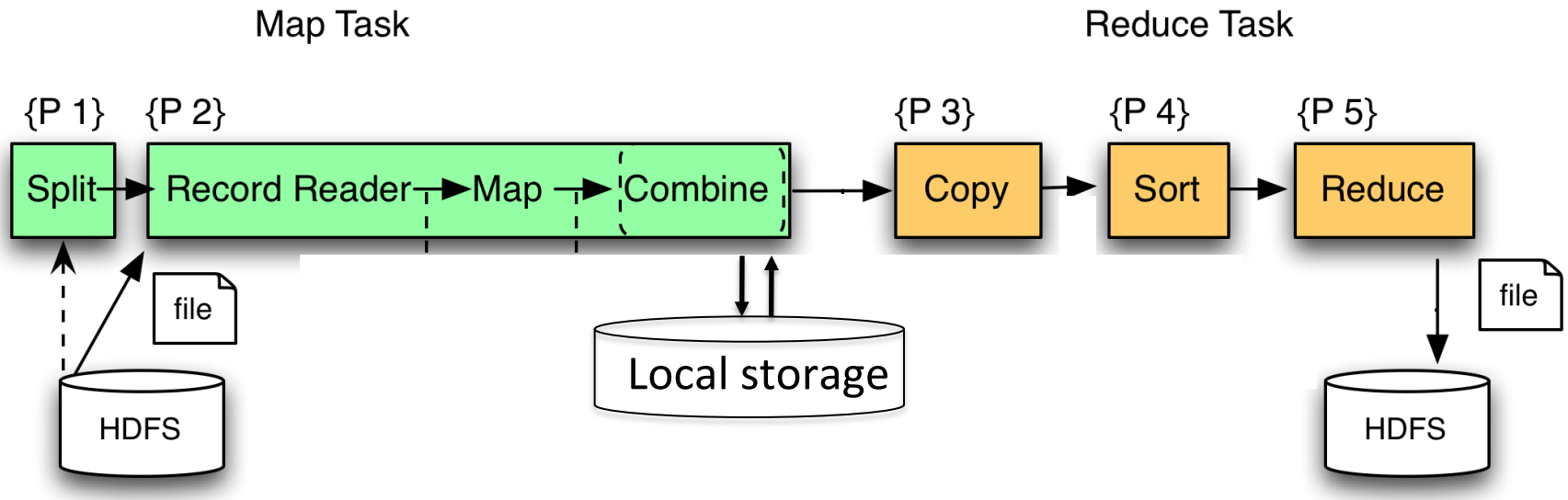
Implementation



Review: MapReduce

- Data is typically a file in the Google File System
  - HDFS for Hadoop
  - File system partitions file into chunks
  - Each chunk is replicated on k (typically 3) machines
- Each machine can run a few map and reduce tasks simultaneously
- Each map task consumes one chunk
  - Can adjust how much data goes into each map task using “splits”
  - Scheduler tries to schedule map task where its input data is located
- Map output is partitioned across reducers
- Map output is also written locally to disk
- Number of reduce tasks is configurable
- System shuffles data between map and reduce tasks
- Reducers sort-merge data before consuming it

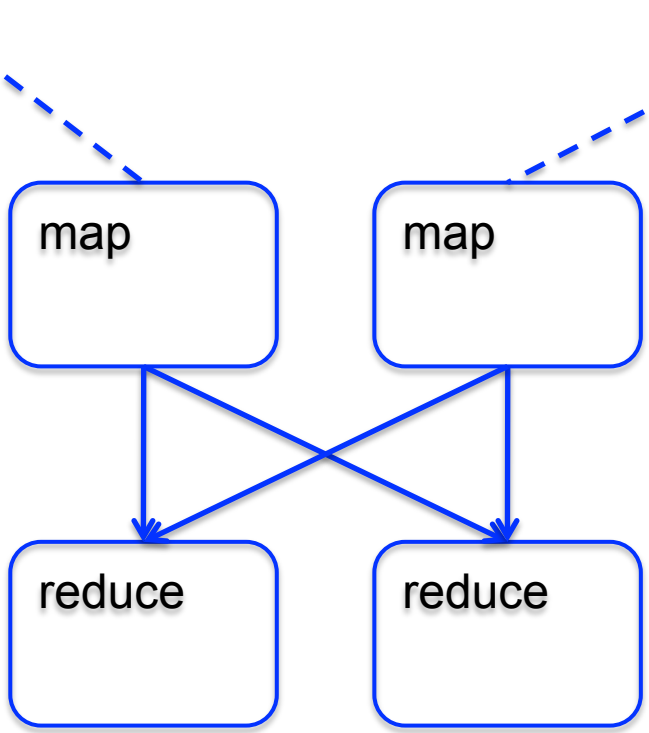
MapReduce Phases



MapReduce Illustrated

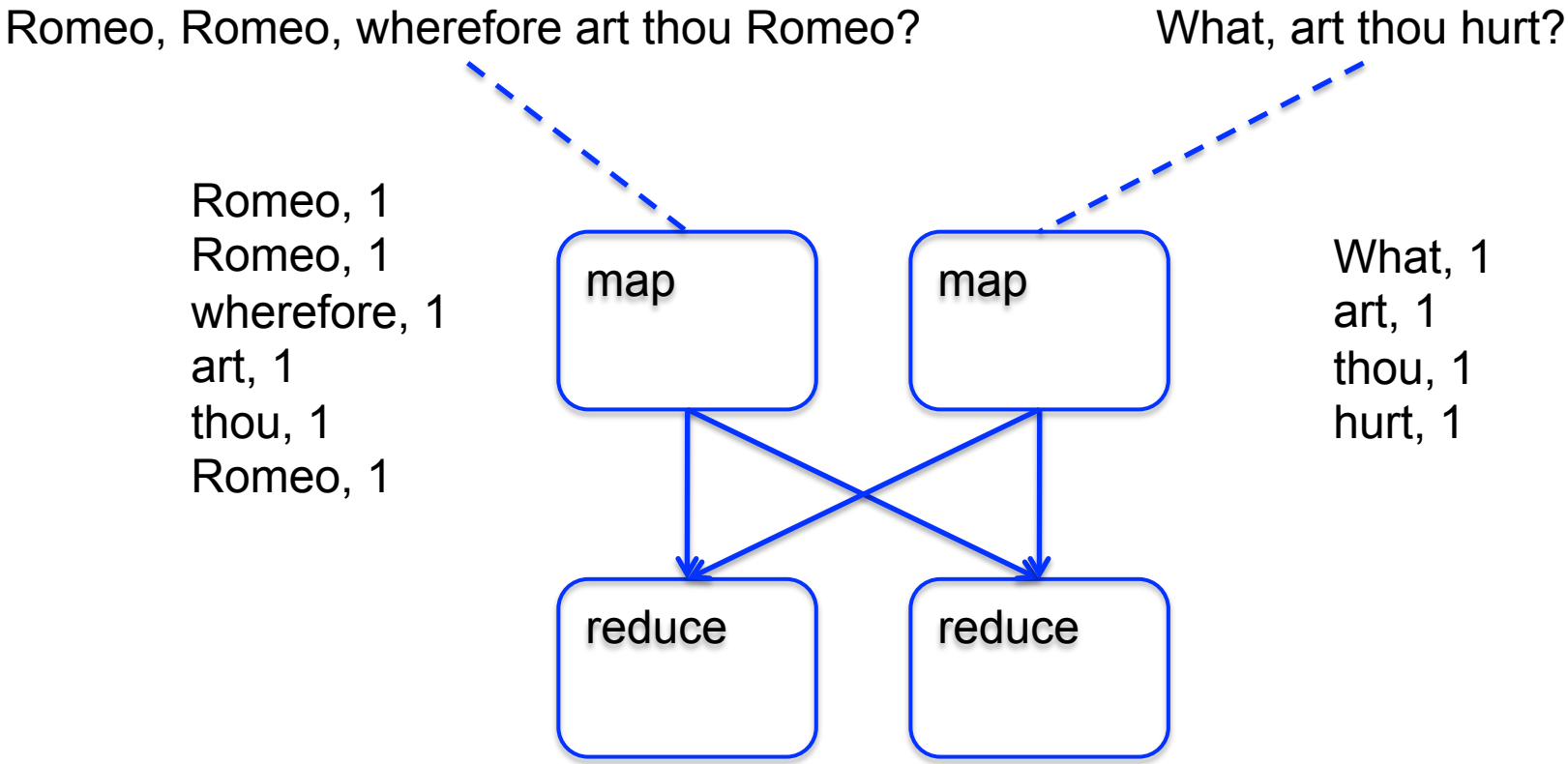
Romeo, Romeo, wherefore art thou Romeo?

What, art thou hurt?



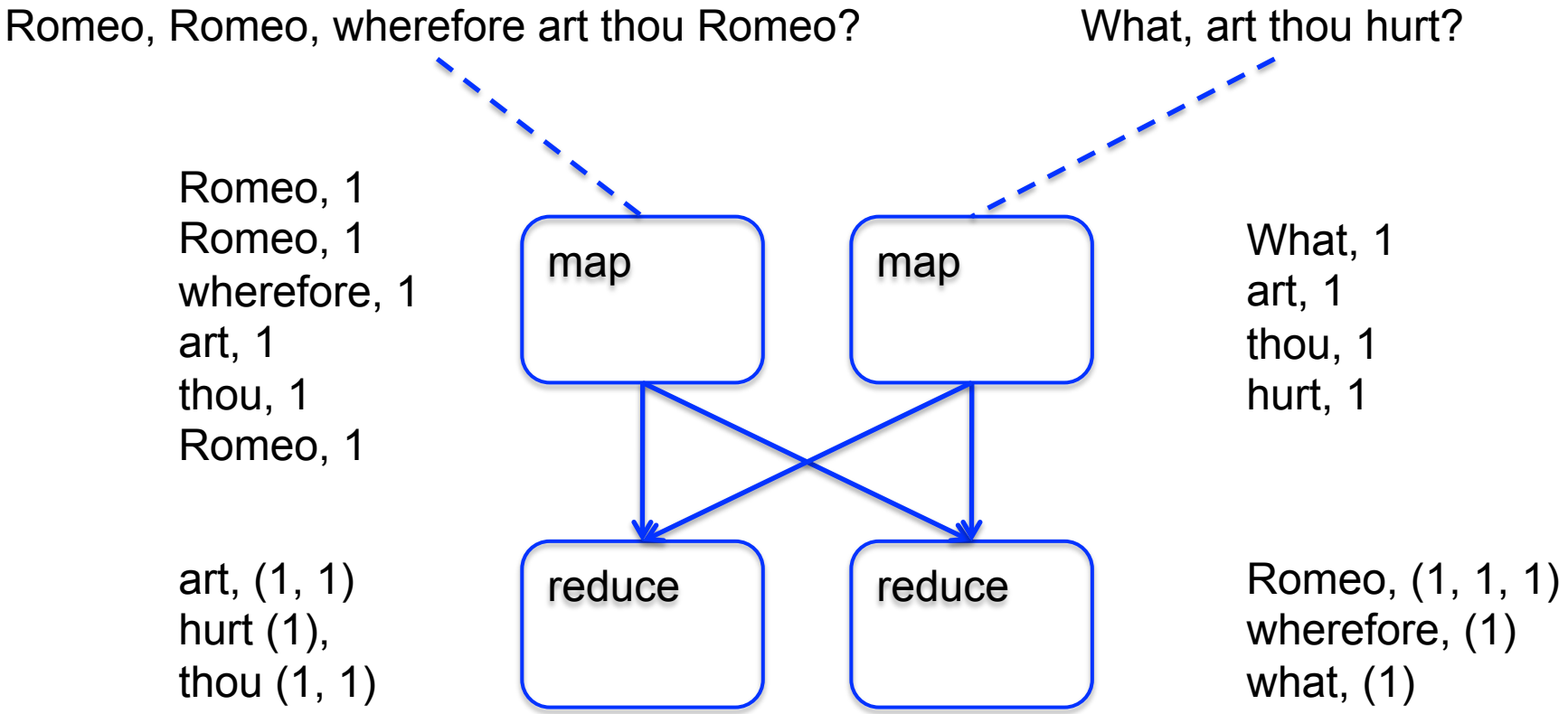
Ack: Alan Gates from Yahoo!

MapReduce Illustrated



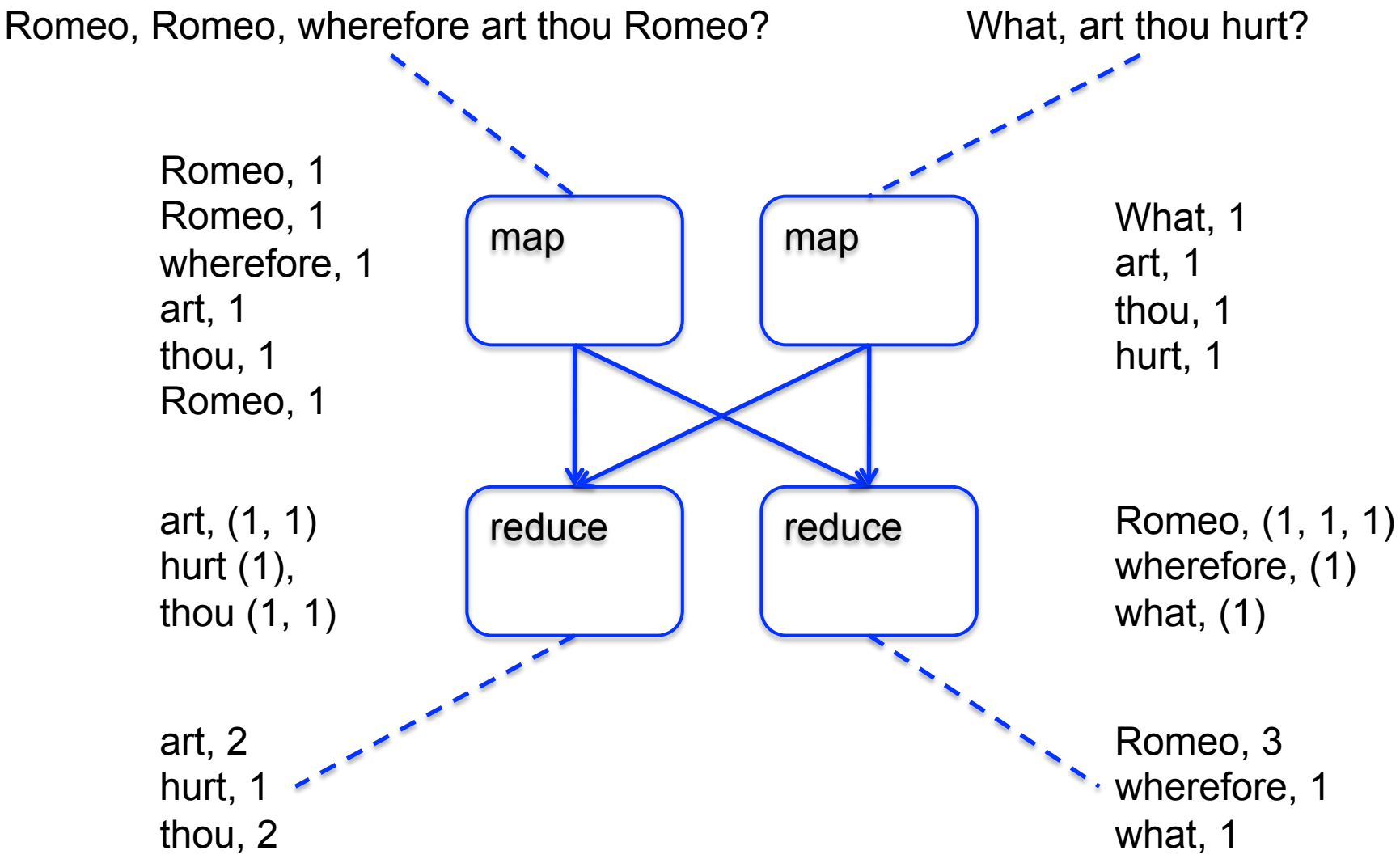
Ack: Alan Gates from Yahoo!

MapReduce Illustrated



Ack: Alan Gates from Yahoo!

MapReduce Illustrated



Ack: Alan Gates from Yahoo!

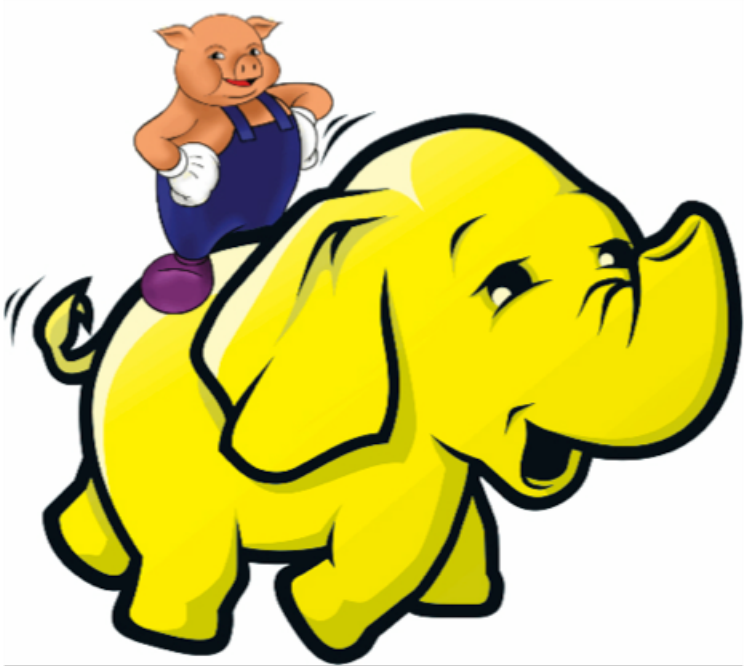
Making Parallelism Simple

- Sequential reads = good read speeds
- In large cluster failures are guaranteed; MapReduce handles retries
- Good fit for batch processing applications that need to touch all your data:
  - data mining
  - model tuning
- Bad fit for applications that need to find one particular record
- Bad fit for applications that need to communicate between processes; oriented around independent units of work

Ack: Alan Gates from Yahoo!

What is Pig?

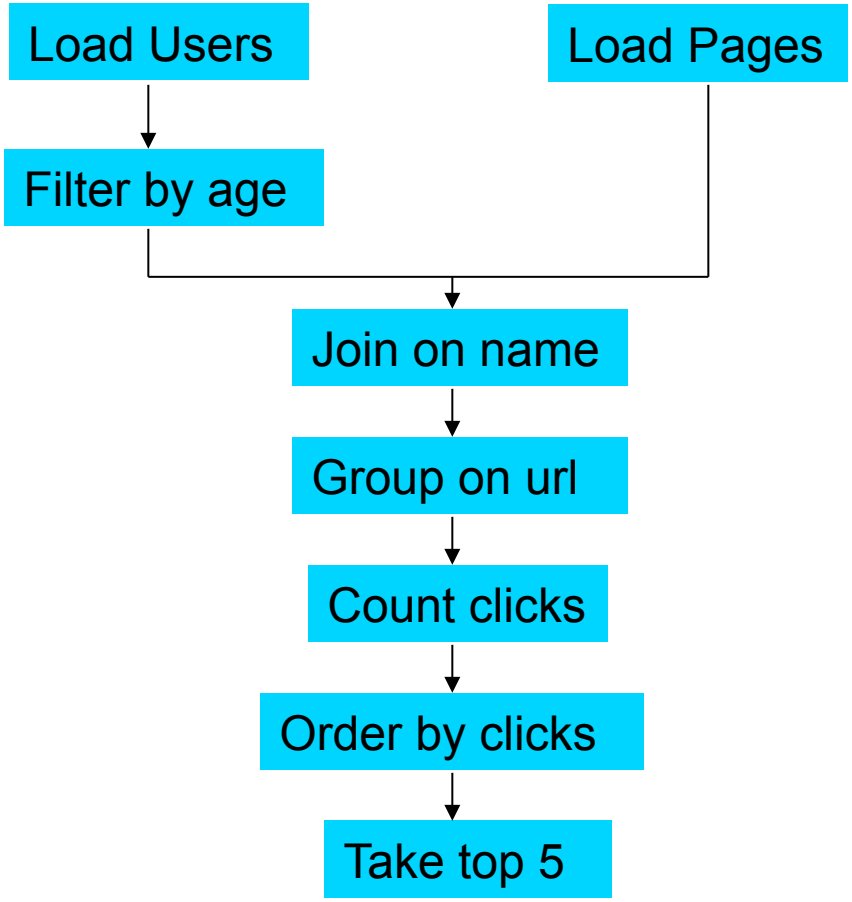
- An engine for executing programs on top of Hadoop
- It provides a language, Pig Latin, to specify these programs
- An Apache open source project
  http://hadoop.apache.org/pig/



Ack: Alan Gates from Yahoo!

Why use Pig?

Suppose you have user data in one file, website data in another, and you need to find the top 5 most visited sites by users aged 18 - 25.



In MapReduce

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.KeyValueTextInputFormat;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.RecordReader;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.SequenceFileInputFormat;
import org.apache.hadoop.mapred.SequenceFileOutputFormat;
import org.apache.hadoop.mapred.TextInputFormat;
import org.apache.hadoop.mapred.jobcontrol.Job;
import org.apache.hadoop.mapred.jobcontrol.JobControl;
import org.apache.hadoop.mapred.lib.IdentityMapper;

public class MRExample {
    public static class LoadPages extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, Text> {

        public void map(LongWritable k, Text val,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // Pull the key out
            String line = val.toString();
            int firstComma = line.indexOf(',');
            String key = line.substring(0, firstComma);
            String value = line.substring(firstComma + 1);
            Text outKey = new Text(key);
            // Prepend an index to the value so we know which file
            // it came from.
            Text outVal = new Text("1" + value);
            oc.collect(outKey, outVal);
        }
    }
    public static class LoadAndFilterUsers extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, Text> {

        public void map(LongWritable k, Text val,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // Pull the key out
            String line = val.toString();
            int firstComma = line.indexOf(',');
            String value = line.substring(firstComma + 1);
            int age = Integer.parseInt(value);
            if (age < 18 || age > 25) return;
            String key = line.substring(0, firstComma);
            Text outKey = new Text(key);
            // Prepend an index to the value so we know which file
            // it came from.
            Text outVal = new Text("2" + value);
            oc.collect(outKey, outVal);
        }
    }
    public static class Join extends MapReduceBase
        implements Reducer<Text, Text, Text, Text> {

        public void reduce(Text key,
                Iterator<Text> iter,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // For each value, figure out which file it's from and
store it
            // accordingly.
            List<String> first = new ArrayList<String>();
            List<String> second = new ArrayList<String>();

            while (iter.hasNext()) {
                Text t = iter.next();
                String value = t.toString();
                if (value.charAt(0) == '1')
first.add(value.substring(1));
                else second.add(value.substring(1));
                reporter.setStatus("OK");
            }

            // Do the cross product and collect the values
            for (String s1 : first) {
                for (String s2 : second) {
                    String outval = key + "," + s1 + "," + s2;
                    oc.collect(null, new Text(outval));
                    reporter.setStatus("OK");
                }
            }
        }
    }
    public static class LoadJoined extends MapReduceBase
        implements Mapper<Text, Text, Text, LongWritable> {

        public void map(
                Text k,
                Text val,
                OutputCollector<Text, LongWritable> oc,
                Reporter reporter) throws IOException {
            // Find the url
            String line = val.toString();
            int firstComma = line.indexOf(',');
            int secondComma = line.indexOf(',', firstComma);
            String key = line.substring(firstComma, secondComma);
            // drop the rest of the record, I don't need it anymore,
            // just pass a 1 for the combiner/reducer to sum instead.
            Text outKey = new Text(key);
            oc.collect(outKey, new LongWritable(1L));
        }
    }
    public static class ReduceUrls extends MapReduceBase
        implements Reducer<Text, LongWritable, WritableComparable,
Writable> {

        public void reduce(
                Text key,
                Iterator<LongWritable> iter,
                OutputCollector<WritableComparable, Writable> oc,
                Reporter reporter) throws IOException {
            // Add up all the values we see

            long sum = 0;
            while (iter.hasNext()) {
                sum += iter.next().get();
                reporter.setStatus("OK");
            }

            oc.collect(key, new LongWritable(sum));
        }
    }
    public static class LoadClicks extends MapReduceBase
        implements Mapper<WritableComparable, Writable, LongWritable,
Text> {

        public void map(
                WritableComparable key,
                Writable val,
                OutputCollector<LongWritable, Text> oc,
                Reporter reporter) throws IOException {
            oc.collect((LongWritable)val, (Text)key);
        }
    }
    public static class LimitClicks extends MapReduceBase
        implements Reducer<LongWritable, Text, LongWritable, Text> {

        int count = 0;
        public void reduce(
            LongWritable key,
            Iterator<Text> iter,
            OutputCollector<LongWritable, Text> oc,
            Reporter reporter) throws IOException {

            // Only output the first 100 records
            while (count < 100 && iter.hasNext()) {
                oc.collect(key, iter.next());
                count++;
            }
        }
    }
    public static void main(String[] args) throws IOException {
        JobConf lp = new JobConf(MRExample.class);
        lp.setJobName("Load Pages");
        lp.setInputFormat(TextInputFormat.class);
        lp.setOutputKeyClass(Text.class);
        lp.setOutputValueClass(Text.class);
        lp.setMapperClass(LoadPages.class);
        FileInputFormat.addInputPath(lp, new
Path("/user/gates/pages"));
        FileOutputFormat.setOutputPath(lp,
                new Path("/user/gates/tmp/indexed_pages"));
        lp.setNumReduceTasks(0);
        Job loadPages = new Job(lp);

        JobConf lfu = new JobConf(MRExample.class);
        lfu.setJobName("Load and Filter Users");
        lfu.setInputFormat(TextInputFormat.class);
        lfu.setOutputKeyClass(Text.class);
        lfu.setOutputValueClass(Text.class);
        lfu.setMapperClass(LoadAndFilterUsers.class);
        FileInputFormat.addInputPath(lfu, new
Path("/user/gates/users"));
        FileOutputFormat.setOutputPath(lfu,
                new Path("/user/gates/tmp/filtered_users"));
        lfu.setNumReduceTasks(0);
        Job loadUsers = new Job(lfu);

        JobConf join = new JobConf(MRExample.class);
        join.setJobName("Join Users and Pages");
        join.setInputFormat(KeyValueTextInputFormat.class);
        join.setOutputKeyClass(Text.class);
        join.setOutputValueClass(Text.class);
        join.setMapperClass(IdentityMapper.class);
        join.setReducerClass(Join.class);
        FileInputFormat.addInputPath(join, new
Path("/user/gates/tmp/indexed_pages"));
        FileInputFormat.addInputPath(join, new
Path("/user/gates/tmp/filtered_users"));
        FileOutputFormat.setOutputPath(join, new
Path("/user/gates/tmp/joined"));
        join.setNumReduceTasks(50);
        Job joinJob = new Job(join);
        joinJob.addDependingJob(loadPages);
        joinJob.addDependingJob(loadUsers);

        JobConf group = new JobConf(MRExample.class);
        group.setJobName("Group URLs");
        group.setInputFormat(KeyValueTextInputFormat.class);
        group.setOutputKeyClass(Text.class);
        group.setOutputValueClass(LongWritable.class);
        group.setOutputFormat(SequenceFileOutputFormat.class);
        group.setMapperClass(LoadJoined.class);
        group.setCombinerClass(ReduceUrls.class);
        group.setReducerClass(ReduceUrls.class);
        FileInputFormat.addInputPath(group, new
Path("/user/gates/tmp/joined"));
        FileOutputFormat.setOutputPath(group, new
Path("/user/gates/tmp/grouped"));
        group.setNumReduceTasks(50);
        Job groupJob = new Job(group);
        groupJob.addDependingJob(joinJob);

        JobConf top100 = new JobConf(MRExample.class);
        top100.setJobName("Top 100 sites");
        top100.setInputFormat(SequenceFileInputFormat.class);
        top100.setOutputKeyClass(LongWritable.class);
        top100.setOutputValueClass(Text.class);
        top100.setOutputFormat(SequenceFileOutputFormat.class);
        top100.setMapperClass(LoadClicks.class);
        top100.setCombinerClass(LimitClicks.class);
        top100.setReducerClass(LimitClicks.class);
        FileInputFormat.addInputPath(top100, new
Path("/user/gates/tmp/grouped"));
        FileOutputFormat.setOutputPath(top100, new
Path("/user/gates/top100sitesforusers18to25"));
        top100.setNumReduceTasks(1);
        Job limit = new Job(top100);
        limit.addDependingJob(groupJob);

       JobControl jc = new JobControl("Find top 100 sites for users
18 to 25");
        jc.addJob(loadPages);
        jc.addJob(loadUsers);
        jc.addJob(joinJob);
        jc.addJob(groupJob);
        jc.addJob(limit);
        jc.run();
    }
}
```

170 lines of code, 4 hours to write

Ack: Alan Gates from Yahoo!

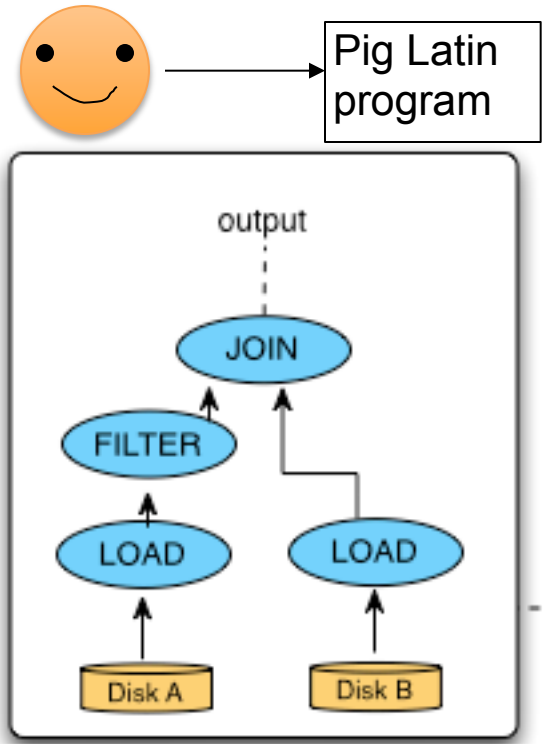
In Pig Latin

```
Users = load 'users' as (name, age);
Fltrd = filter Users by
        age >= 18 and age <= 25;
Pages = load 'pages' as (user, url);
Jnd = join Fltrd by name, Pages by user;
Grpd = group Jnd by url;
Smmd = foreach Grpd generate group,
       COUNT(Jnd) as clicks;
Srtd = order Smmd by clicks desc;
Top5 = limit Srtd 5;
store Top5 into 'top5sites';
```

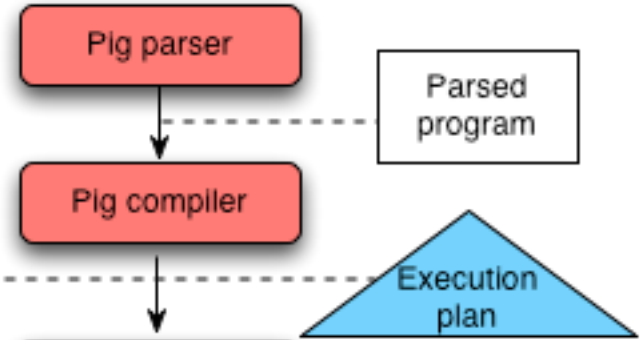
9 lines of code, 15 minutes to write

Ack: Alan Gates from Yahoo!

Pig System Overview




```
A = LOAD 'file1' AS (sid,pid,mass,px:double);
B = LOAD 'file2' AS (sid,pid,mass,px:double);
C = FILTER A BY px < 1.0;
D = JOIN C BY sid,
         B BY sid;
   STORE g INTO 'output.txt';
```



But can it fly?