GENERAL ASSEMBLY

INTRO TO DATA SCIENCE

LECTURE 15: MAP-REDUCE

LAST TIME:

- RELATIONAL DB’S (RDBMS)

QUESTIONS?

I. BIG DATA

II. PROGRAMMING MODEL

III. IMPLEMENTATION DETAILS

EXERCISE:

V. MAP-REDUCE USING PIG

I. BIG DATA

As you have probably heard, **big data** is a hot topic these days.

Q: What does “big data” actually refer to?

As you have probably heard, **big data** is a hot topic these days.

Q: What does “big data” actually refer to?

A: Scalability; in particular, storing & processing web-scale (multi-terabyte) datasets…

One approach would be to get a huge supercomputer.

One approach would be to get a huge supercomputer.

But this has some obvious drawbacks:
- expensive
- difficult to maintain
- scalability is bounded

Instead of one huge machine, what if we got a bunch of regular (*commodity*) machines?

Instead of one huge machine, what if we got a bunch of regular (*commodity*) machines?

This has obvious benefits!
- cheaper
- easier to maintain
- scalability is unbounded (just add more nodes to the *cluster*)

Now we can give a complete answer to our earlier question.

Q: What does “big data” actually refer to?

Now we can give a complete answer to our earlier question.

Q: What does "big data" actually refer to?

A: Scalability; in particular, storing & processing web-scale (multi-terabyte) datasets *using clusters of multiple computing nodes.*

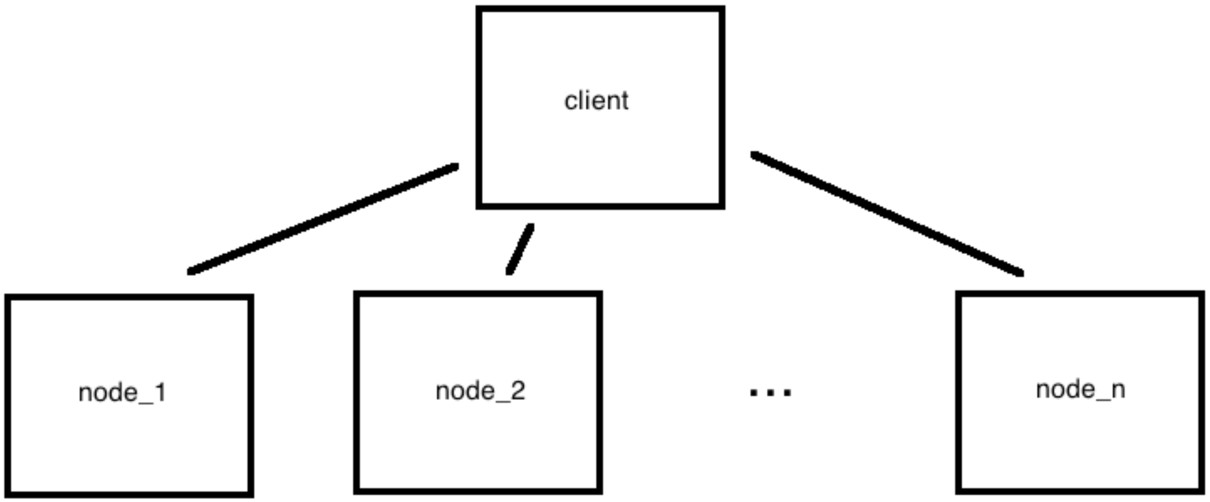
Now we can give a complete answer to our earlier question.

Q: What does "big data" actually refer to?

A: Scalability; in particular, storing & processing web-scale (multi-terabyte) datasets *using clusters of multiple computing nodes.*

"Scale out vs scale up!"

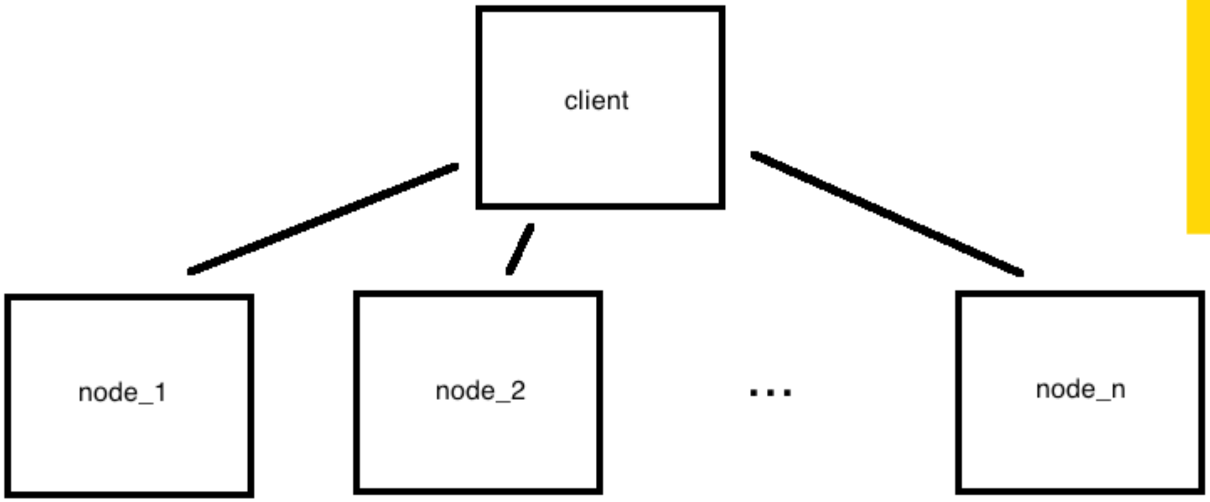
We can visualize this *horizontal* cluster architecture as a single client-multiple server relationship



We can visualize this *horizontal* cluster architecture as a single client-multiple server relationship

NOTE

A horizontally distributed system also has better *fault tolerance* than a single machine.



How do we process data in a distributed architecture?

- move code to data
  - map-reduce → less overhead (network traffic, disk I/O)

“Computing nodes are the same as storage nodes.”

Divide and conquer is a fundamental algorithmic technique for solving a given task, whose steps include:

Divide and conquer is a fundamental algorithmic technique for solving a given task, whose steps include:

1) split task into subtasks
2) solve these subtasks *independently*
3) recombine the subtask results into a final result

Divide and conquer is a fundamental algorithmic technique for solving a given task, whose steps include:

1) split task into subtasks
2) solve these subtasks *independently*
3) recombine the subtask results into a final result

Map-reduce leverages the divide and conquer approach by splitting a large dataset into several smaller datasets and performing a computation on each of these in parallel.

The defining characteristic of a problem that is suitable for the divide and conquer approach is that it can be broken down into *independent subtasks*.

The defining characteristic of a problem that is suitable for the divide and conquer approach is that it can be broken down into *independent subtasks*.

Tasks that can be *parallelized* in this way include:

- count, sum, average
- grep, sort, inverted index
- graph traversals, ***some*** ML algorithms

The defining characteristic of a problem that is suitable for the divide and conquer approach is that it can be broken down into *independent subtasks*.

Tasks that can be *parallelized* in this way include:
- count, sum, average
- grep, sort, inverted index
- graph traversals, some ML algorithms

NOTE

Parallelizing an ML algorithm can be a non-trivial exercise!

II. PROGRAMMING MODEL

As we've discussed, the map-reduce approach involves splitting a problem into subtasks and processing these subtasks in parallel.

As we’ve discussed, the map-reduce approach involves splitting a problem into subtasks and processing these subtasks in parallel.

This takes place in two phases:

As we've discussed, the map-reduce approach involves splitting a problem into subtasks and processing these subtasks in parallel.

This takes place in two phases:

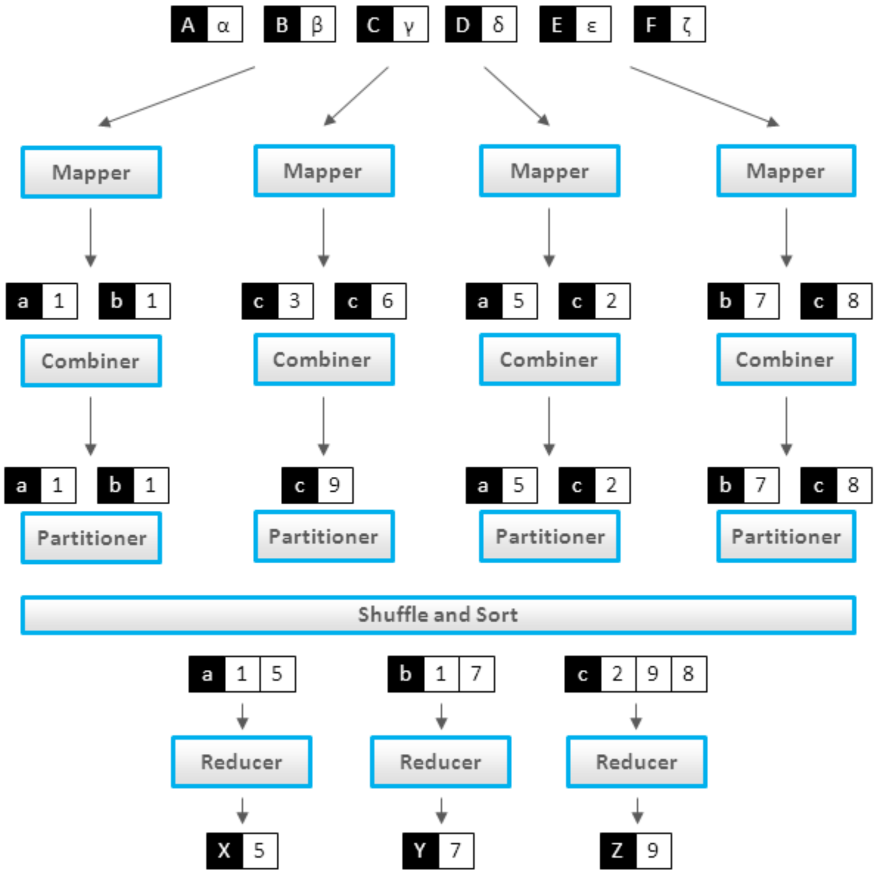
1) the **mapper** phase
2) the **reducer** phase

As we've discussed, the map-reduce approach involves splitting a problem into subtasks and processing these subtasks in parallel.

This takes place in (*approximately*) two phases:

1) the **mapper** phase
1.5) *shuffle/sort*
2) the **reducer** phase

MAP-REDUCE


A α
B β
C γ
D δ
E ε
F ζ
Mapper
Mapper
Mapper
Mapper
a 1
b 1
c 3
c 6
a 5
c 2
b 7
c 8
Combiner
Combiner
Combiner
Combiner
a 1
b 1
c 9
a 5
c 2
b 7
c 8
Partitioner
Partitioner
Partitioner
Partitioner
Shuffle and Sort
a 1 5
b 1 7
c 2 9 8
Reducer
Reducer
Reducer
X 5
Y 7
Z 9

Map-reduce uses a *functional programming* paradigm. The data processing *primitives* are mappers and reducers, as we've seen.

Map-reduce uses a *functional programming* paradigm. The data processing *primitives* are mappers and reducers, as we've seen.

mappers – filter & transform data
reducers – aggregate results

Map-reduce uses a *functional programming* paradigm. The data processing *primitives* are mappers and reducers, as we've seen.

mappers – filter & transform data
reducers – aggregate results

The functional paradigm is good at describing how to solve a problem, but not very good at describing data manipulations (eg, relational joins).

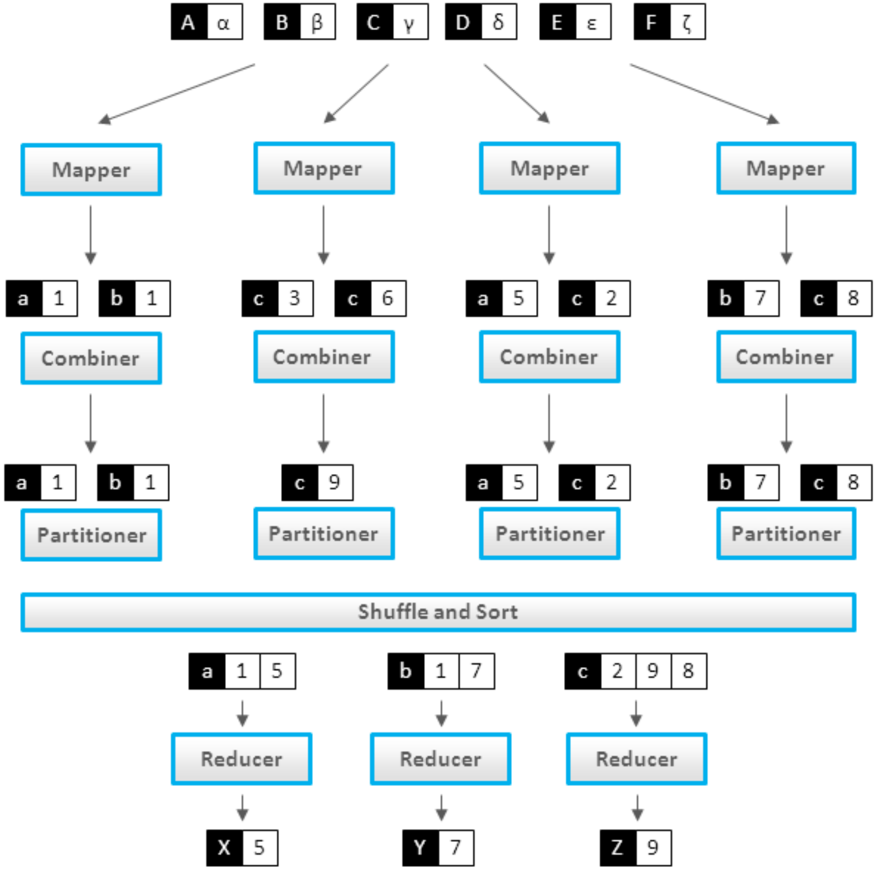
As our earlier diagram suggests, there are additional intermediate steps in a map-reduce workflow.

mappers – filter & transform data
reducers – aggregate results

As our earlier diagram suggests, there are additional intermediate steps in a map-reduce workflow.

mappers – filter & transform data
combiners – perform reducer operations on the mapper node (optional step, to reduce network traffic and disk I/O).
partitioners – shuffle/sort/redirect mapper output
reducers – aggregate results

MAP-REDUCE


A α
B β
C γ
D δ
E ε
F ζ
Mapper
Mapper
Mapper
Mapper
a 1
b 1
c 3
c 6
a 5
c 2
b 7
c 8
Combiner
Combiner
Combiner
Combiner
a 1
b 1
c 9
a 5
c 2
b 7
c 8
Partitioner
Partitioner
Partitioner
Partitioner
Shuffle and Sort
a 1 5
b 1 7
c 2 9 8
Reducer
Reducer
Reducer
X 5
Y 7
Z 9

II. IMPLEMENTATION DETAILS

The map-reduce framework handles a lot of messy details for you:

The map-reduce framework handles a lot of messy details for you:

- parallelization & distribution (eg, input splitting)
- partitioning (shuffle/sort/redirect)
- fault-tolerance (fact: tasks/nodes will fail!)
- I/O scheduling
- status and monitoring

The map-reduce framework handles a lot of messy details for you:

- parallelization & distribution (eg, input splitting)
- partitioning (shuffle/sort/redirect)
- fault-tolerance (fact: tasks/nodes will fail!)
- I/O scheduling
- status and monitoring

This (along with the functional semantics) allows you to focus on solving the problem instead of accounting & housekeeping details.

Hadoop is a popular open-source Java-based implementation of the map-reduce framework (including file storage for input/output).

Hadoop is a popular open-source Java-based implementation of the map-reduce framework (including file storage for input/output).

You can download Hadoop and configure a set of machines to operate as a map-reduce cluster, or you can run it as a *service* via Amazon's Elastic Map-Reduce.

Hadoop is a popular open-source Java-based implementation of the map-reduce framework (including file storage for input/output).

You can download Hadoop and configure a set of machines to operate as a map-reduce cluster, or you can run it as a *service* via Amazon's Elastic Map-Reduce.

Hadoop is written in Java, but the *Hadoop Streaming* utility allows client code to be supplied as executables (eg, written in any language).

Frequently when people say “map-reduce” they’re referring to Hadoop, but there are some exceptions:

Frequently when people say “map-reduce” they’re referring to Hadoop, but there are some exceptions:

- many NoSQL databases support native map-reduce queries
- commercial distributions (Cloudera, MapR, etc)
- Google’s internal implementation

Data is replicated in the (distributed) file system across several nodes.

Data is replicated in the (distributed) file system across several nodes.

This permits locality optimization (and fault tolerance) by allowing the mapper tasks to run on the same nodes where the data resides.

Data is replicated in the (distributed) file system across several nodes.

This permits locality optimization (and fault tolerance) by allowing the mapper tasks to run on the same nodes where the data resides.

So we move code to data (instead of data to code), thus avoiding a lot of network traffic and disk I/O.

Data is replicated in the (distributed) file system across se

This permits locality optimization (and fault tolerance) by a
mapper tasks to run on the same nodes where the data re

NOTE

"Compute nodes are the same as storage nodes."

So we move code to data (instead of data to code), thus avoiding a lot of network traffic and disk I/O.

The *Google File System* (GFS) was developed alongside map-reduce to serve as the native file system for this type of processing.

The *Google File System* (GFS) was developed alongside map-reduce to serve as the native file system for this type of processing.

The Hadoop platform is bundled with an open-source implementation of this file system called *HDFS*.

It’s possible to overlay the map-reduce framework with an additional *declarative syntax*.

This makes operations like select & join easier to implement and less error prone.

Popular examples include Pig and Hive.

source: http://www.slideshare.net/kevinweil/hadoop-pig-and-twitter-nosql-east-2009

A Real Pig Script

top_5.pig

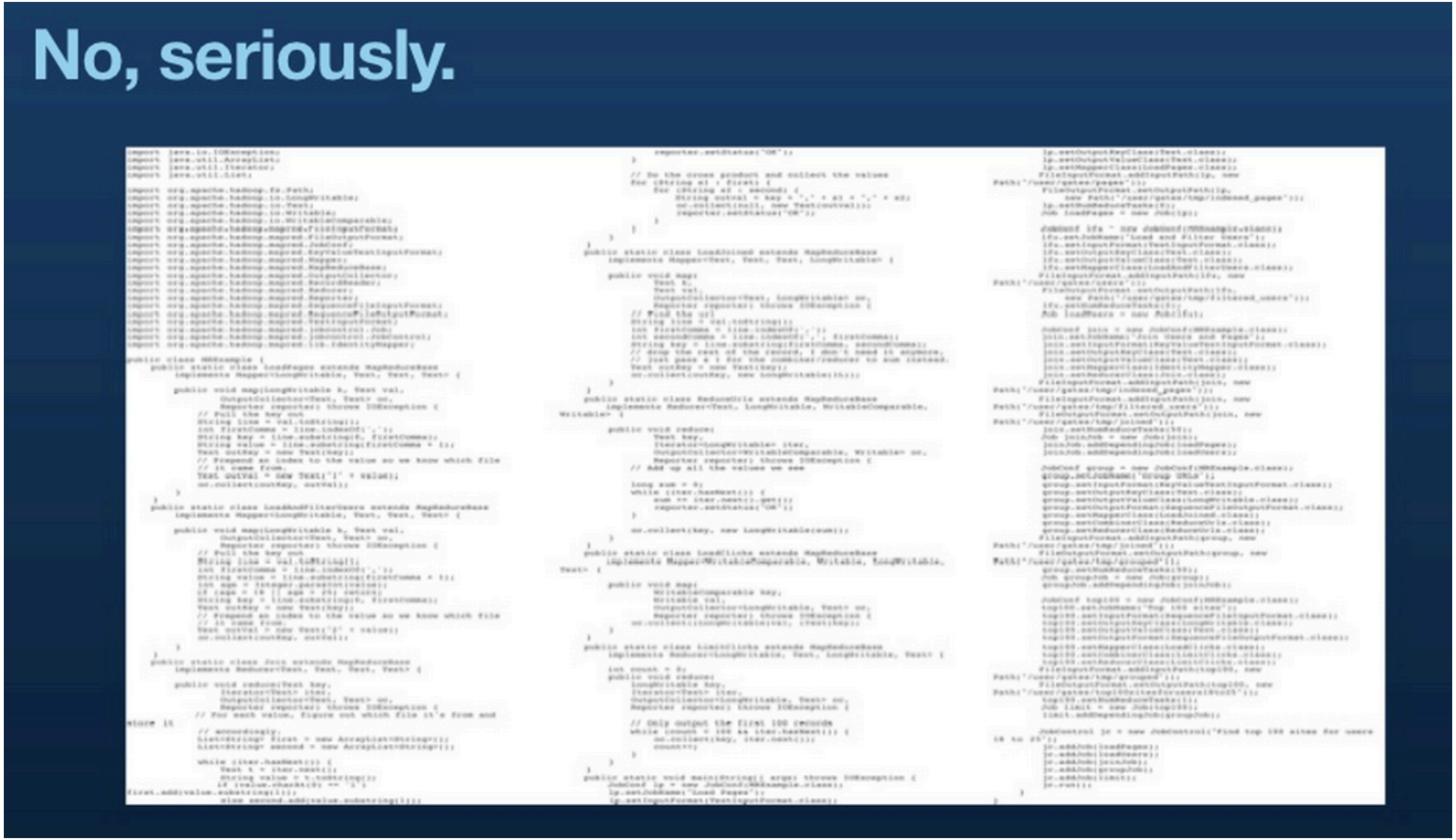
```
users = load 'users.csv' as (username: chararray, age: int);
users_1825 = filter users by age >= 18 and age <= 25;

pages = load 'pages.csv' as (username: chararray, url: chararray);

joined = join users_1825 by username, pages by username;
grouped = group joined by url;
summed = foreach grouped generate group as url, COUNT(joined) AS views;
sorted = order summed by views desc;
top_5 = limit sorted 5;

store top_5 into 'top_5_sites.csv';
```

- Now, just for fun... the same calculation in vanilla Hadoop MapReduce.


No, seriously.

III. WORD COUNT EXAMPLE

Map-reduce processes data in terms of *key-value pairs*:

```
input             <k1, v1>

mapper            <k1, v1> → <k2, v2>

(partitioner)     <k2, v2> → <k2, [all k2 values]>

reducer           <k2, [all k2 values]> → <k3, v3>
```

Using the following input, we can implement the “Hello World” of map-reduce: a *word count*.

Using the following input, we can implement the "Hello World" of map-reduce: a *word count*.

```
where
where in
where in the
where in the world
where in the world is
where in the world is carmen
where in the world is carmen sandiego
```

The first processing primitive is the mapper, which filters & transforms the input data, and *emits* transformed key-value pairs.

The first processing primitive is the mapper, which filters & transforms the input data, and *emits* transformed key-value pairs.

```
mapper(k1, v1):
// k1 = line number
// v1 = line contents (eg, space-delimited string)

    words = tokenize(v1)   // split string into words
    for word in words:
        emit (word, 1)
```

The mapper emits key-value pairs for each word encountered in the input data.

The mapper emits key-value pairs for each word encountered in the input data.

```
where 1
where 1
in    1
where 1
in    1
the   1
...
```

The partitioner is internal to the map-reduce framework, so we don't have to write this ourselves. It shuffles & sorts the mapper output, and redirects all intermediate results for a given key to a *single* reducer.

The partitioner is internal to the map-reduce framework, so we don't have to write this ourselves. It shuffles & sorts the mapper output, and redirects all intermediate results for a given key to a *single* reducer.

```
where       [1, 1, 1, 1, 1, 1, 1]
in          [1, 1, 1, 1, 1, 1]
the         [1, 1, 1, 1, 1]
world       [1, 1, 1, 1]
is          [1, 1 ,1]
carmen      [1, 1]
sandiego    [1]
```

Finally, the reducer receives all values for a given key and aggregates (in this case, sums) the results.

Finally, the reducer receives all values for a given key and aggregates (in this case, sums) the results.

```
reducer(k2, k2_vals):
// k2 = word
// k2_vals = word counts

    emit k2, sum(k2_vals)
```

Reducer output is aggregated...

```
where      7
in         6
the        5
world      4
is         3
carmen     2
sandiego   1
```

Reducer output is aggregated & sorted by key.

```
carmen      2
is          3
in          6
the         5
sandiego    1
where       7
world       4
```