GENERAL ASSEMBLY

INTRO TO DATA SCIENCE

LECTURE 16: MAP-REDUCE

I. BIG DATA

II. HADOOP ECOSYSTEM

III. MAP-REDUCE PROGRAMMING MODEL

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.

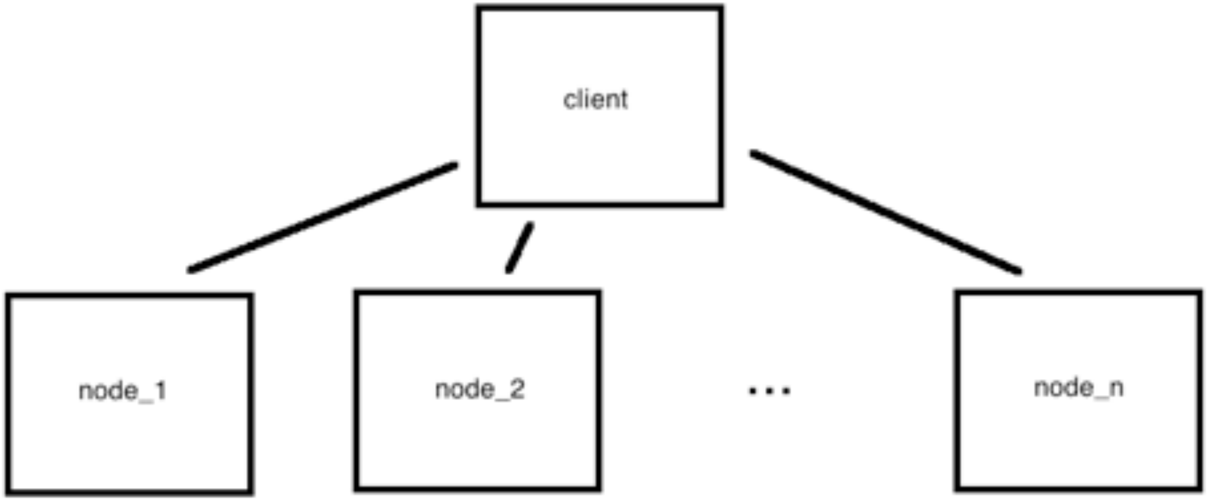
Now we can give a complete answer to our earlier question.

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!"

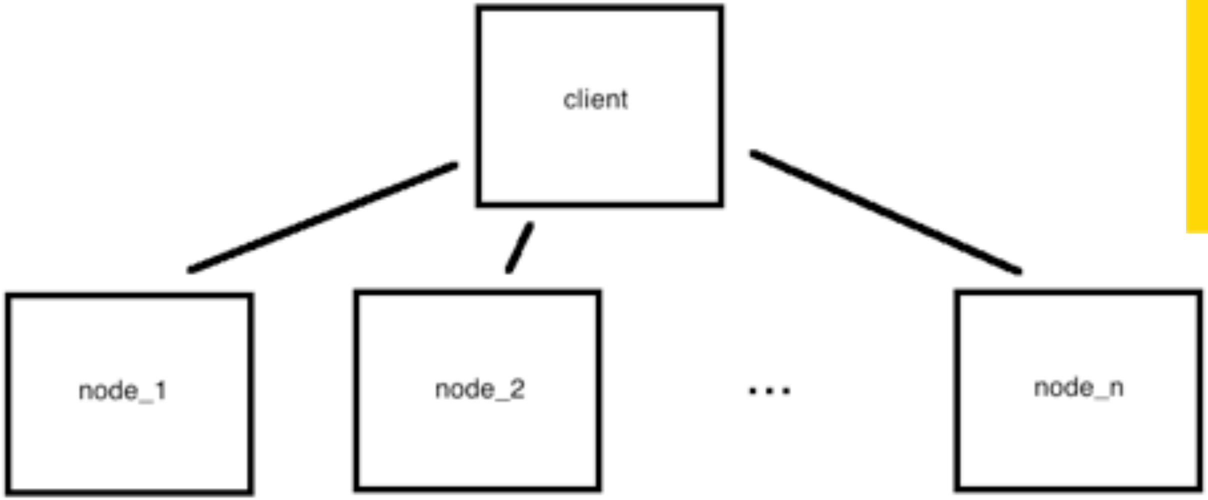
We can visualize this horizontal cluster architecture as a single client-multiple server relationship



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*

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. HADOOP ECOSYSTEM

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

Log storage and analysis

Used for charts calculation, royalty reporting, log analysis, A/B testing, dataset merging

Also used for large scale audio feature analysis over millions of tracks

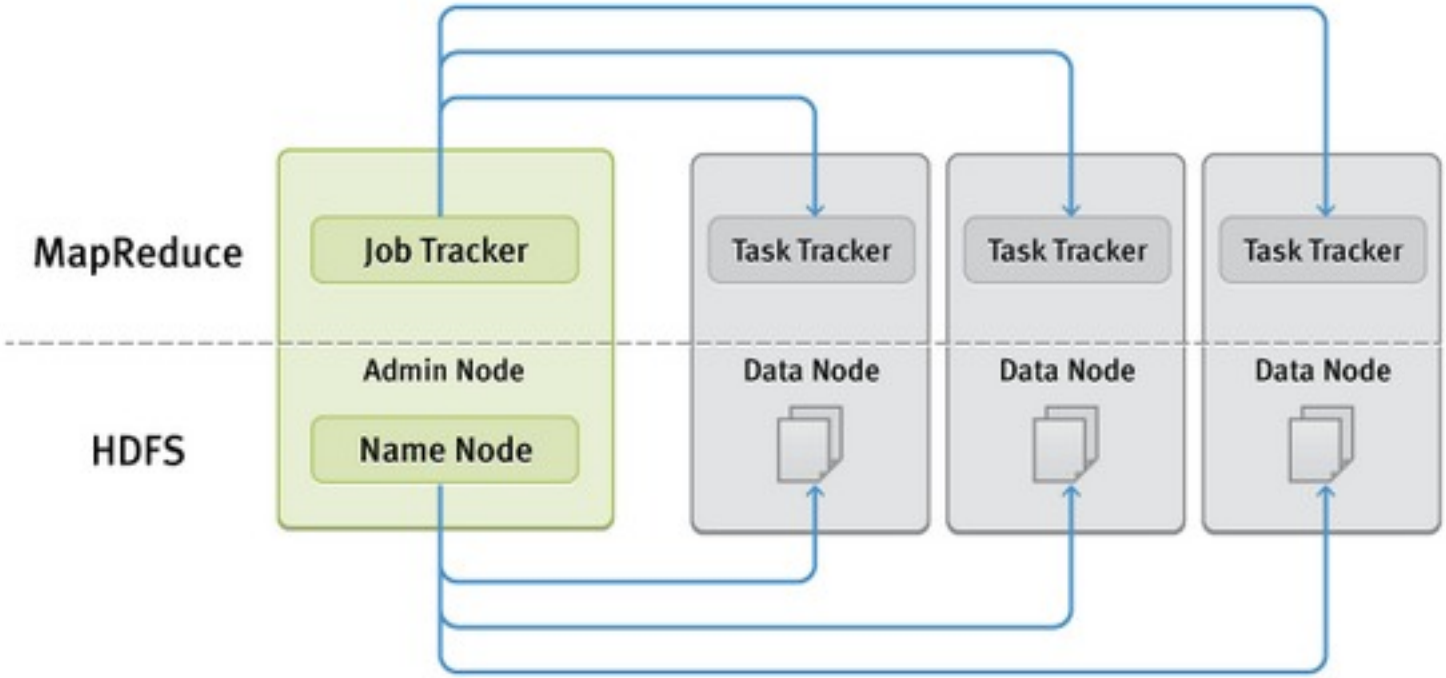
Large-scale image conversions

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).*

More often, Hadoop refers to the ecosystem of tools around distributed computing with two main components:
- distributed filesystem (HDFS)
- map-reduce job scheduler

Google	Open-source	Function
GFS	HDFS	Distributed file system
MapReduce	MapReduce	Batch distributed data processing
Bigtable	HBase	Distributed DB/key-value store
Protobuf/Stubby	Thrift or Avro	Data serialization/RPC
Pregel	Giraph	Distributed graph processing
Tenzing	Hive	Scalable SQL on MapReduce
Dremel/F1	Cloudera Impala	Scalable interactive SQL (MPP)
FlumeJava	Crunch	Abstracted data pipelines on Hadoop


MapReduce
HDFS
Job Tracker
Admin Node
Name Node
Task Tracker
Task Tracker
Task Tracker
Data Node
Data Node
Data Node

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

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.

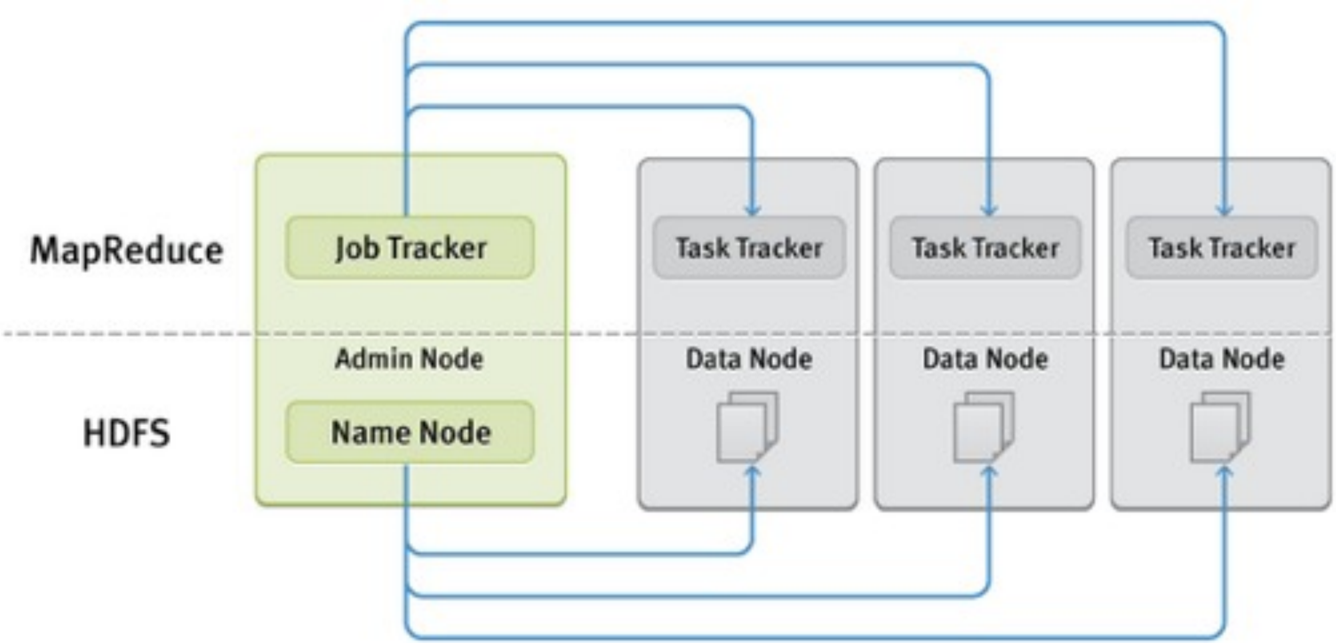
This permits locality optimization (and fault tolerance) by allowing the mapper tasks to run on the same nodes where the data resides.

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

HDFS benefits include:

- Push compute tasks to data nodes to avoid data transfer

- Data replication: data is replicated so if a single machine fails another still contains the data



Source: http://www.ndm.net/datawarehouse/Greenplum/hadoop-components

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.

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.

Google	Open-source	Function
GFS	HDFS	Distributed file system
MapReduce	MapReduce	Batch distributed data processing
Bigtable	HBase	Distributed DB/key-value store
Protobuf/Stubby	Thrift or Avro	Data serialization/RPC
Pregel	Giraph	Distributed graph processing
Tenzing	Hive	Scalable SQL on MapReduce
Dremel/F1	Cloudera Impala	Scalable interactive SQL (MPP)
FlumeJava	Crunch	Abstracted data pipelines on Hadoop

Apache Hive

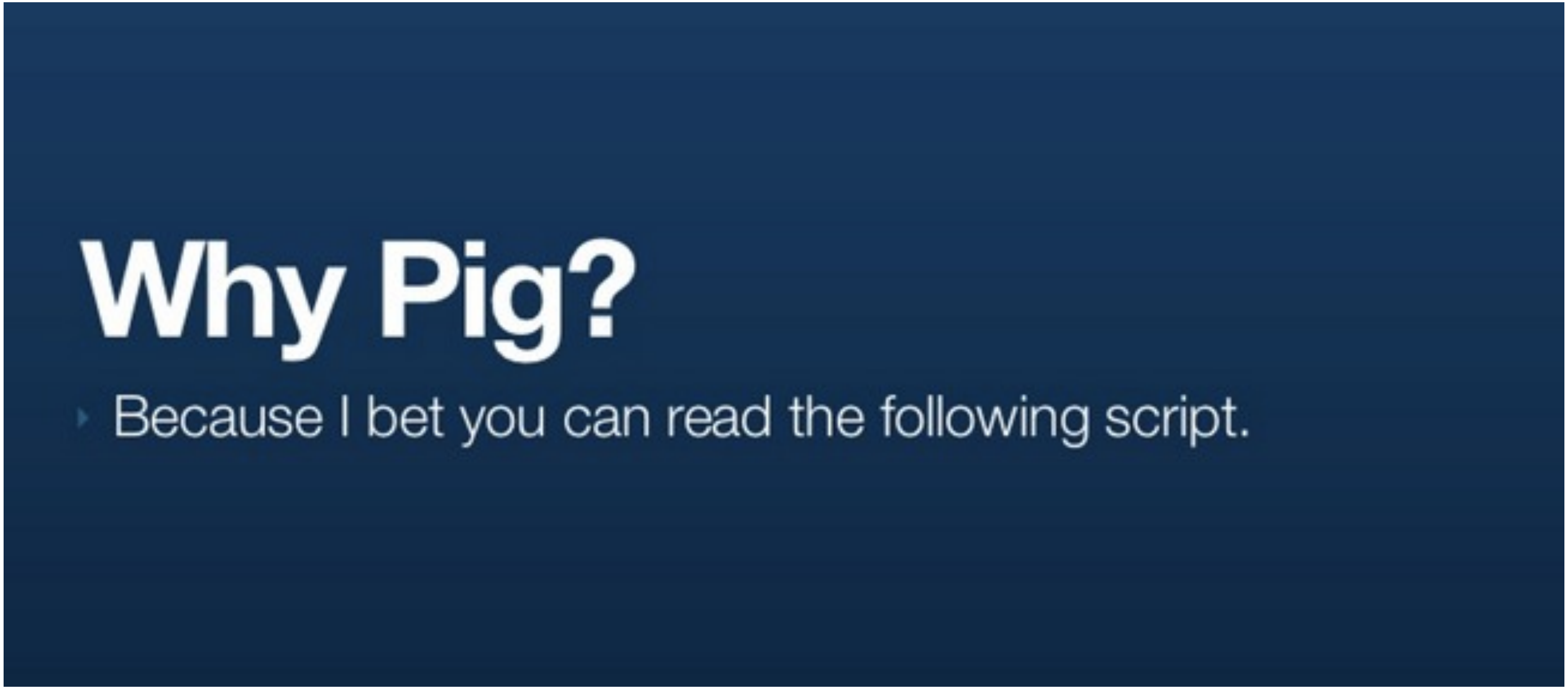
- SQL language to query data on HDFS
- Queries are translated behind the scenes into map-reduce jobs
- Data is stored on HDFS, but a metadata database contains the table schemas

Cloudera Impala

- ***ANOTHER*** *SQL language to query data on HDFS*
- *Similar interface to Hive*

BUT:

- *Impala contains its own scheduling engine, queries are not translated map-reduce jobs*
    - *Leads to faster queries, but no fault tolerance*



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.

Parquet Data Storage

- *Nested columnar data storage*
- *Based on Google Dremel paper*
- *Open-sourced by Cloudera and Twitter in July 2013*

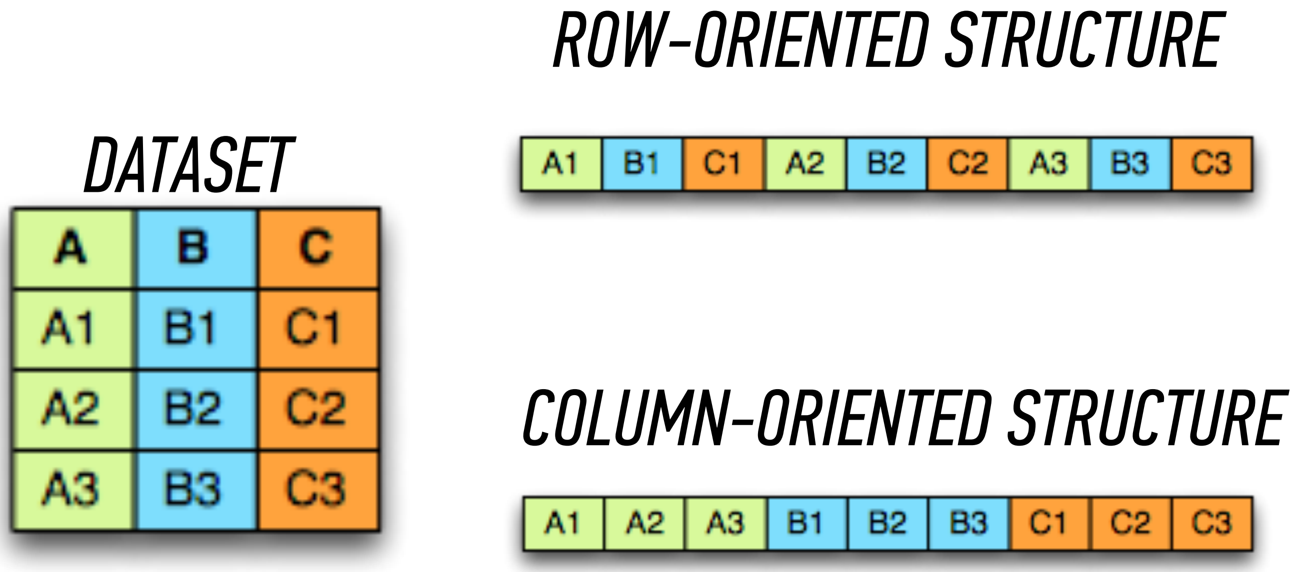
"Ideal for tables containing many columns, where most queries only refer to a small subset of the columns"

Parquet Data Storage

DATASET

A	B	C
A1	B1	C1
A2	B2	C2
A3	B3	C3

Parquet Data Storage



Parquet Data Storage

Advantages

- *Limit I/O depending on columns queried*
- *Filter rows before reading all columns*
- *Efficient compression*

III. MAP-REDUCE PROGRAMMING

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

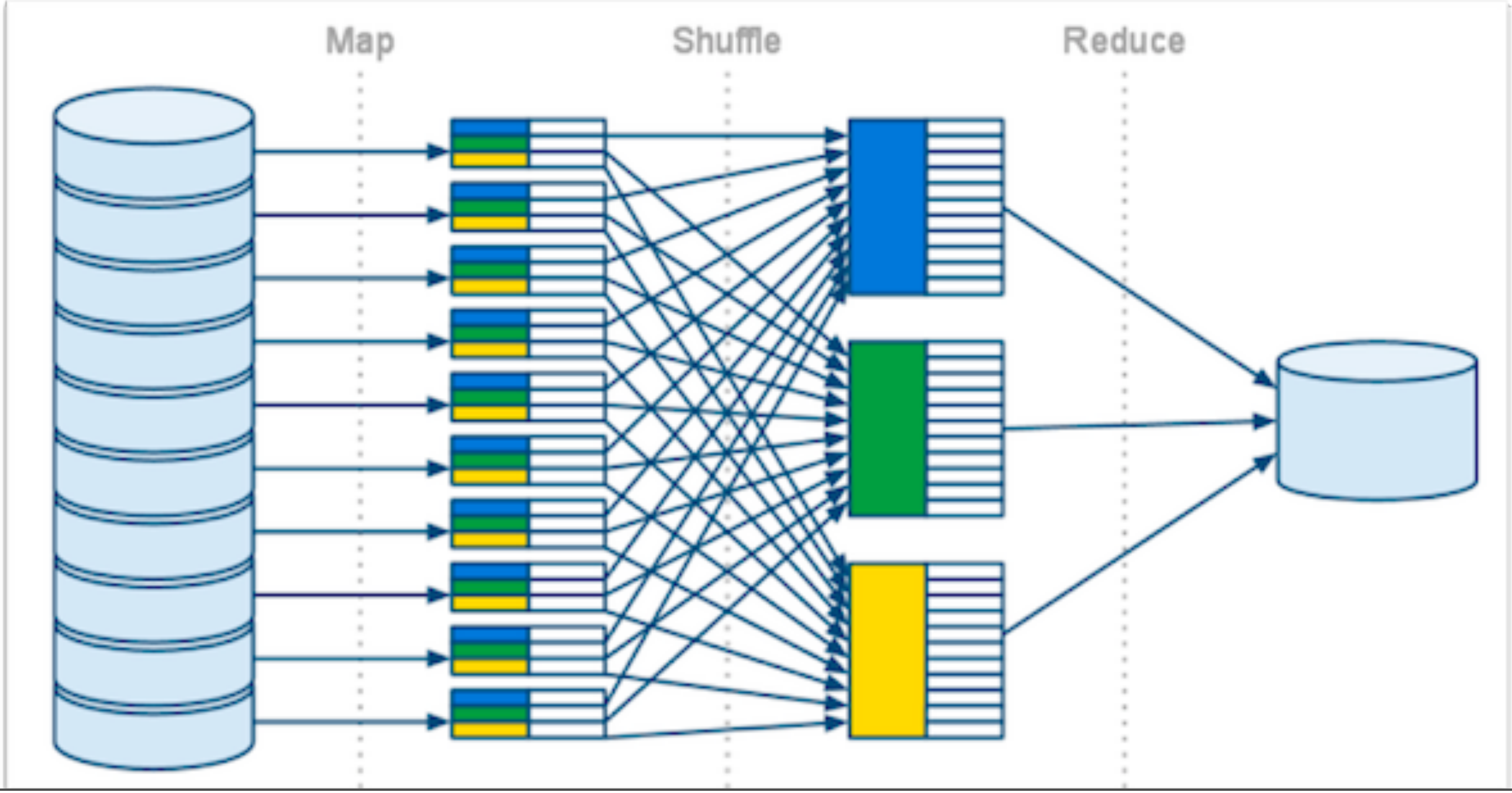
Map-reduce uses a functional programming paradigm. The data processing primitives are mappers and reducers, as we've seen.

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).


Map
Shuffle
Reduce

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

mappers – *filter & transform data*
reducers – *aggregate results*

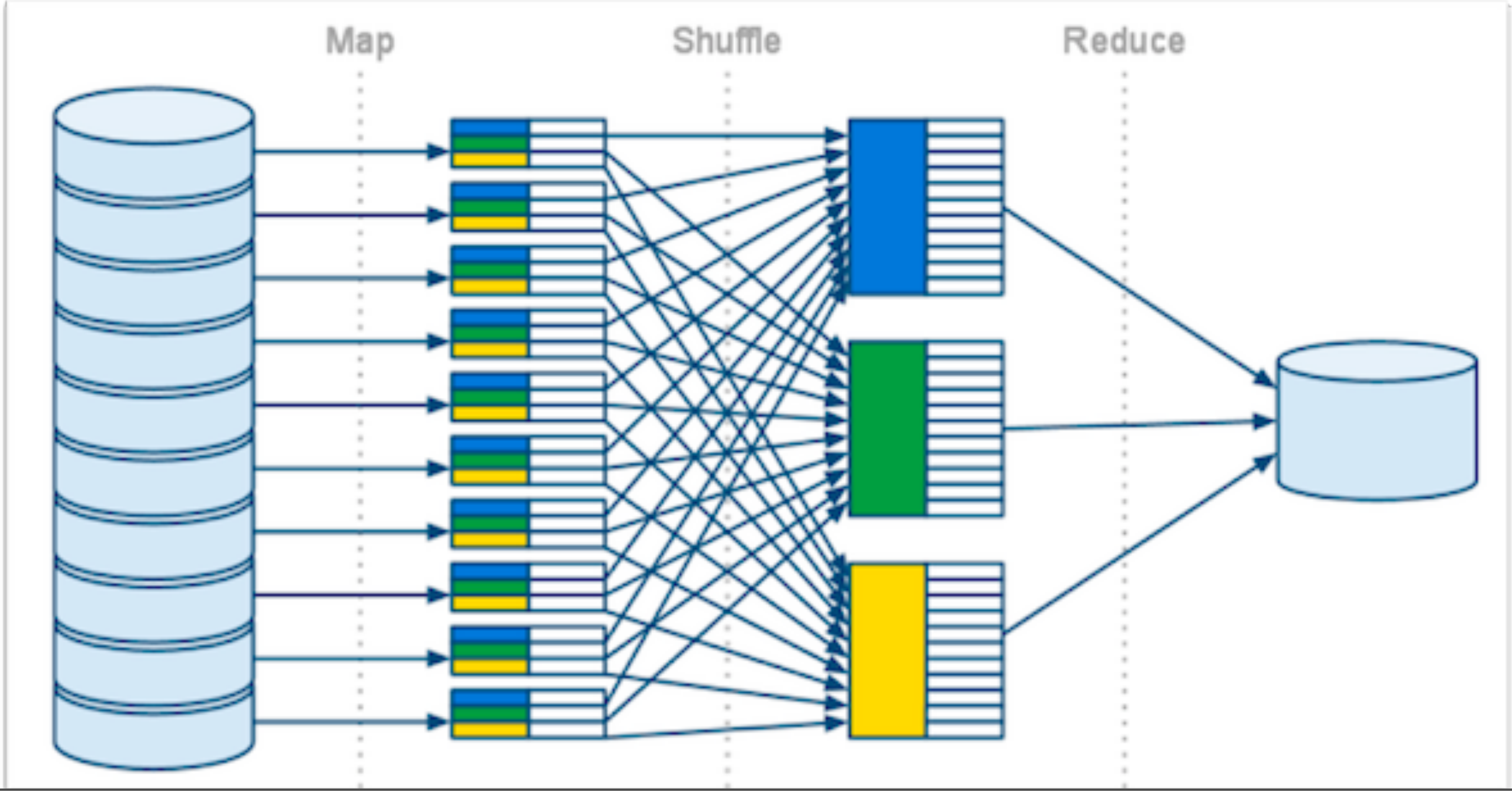
As our earlier diagram suggests, there are additional intermediate steps in a map-reduce workflow.

mappers – *filter & transform data*

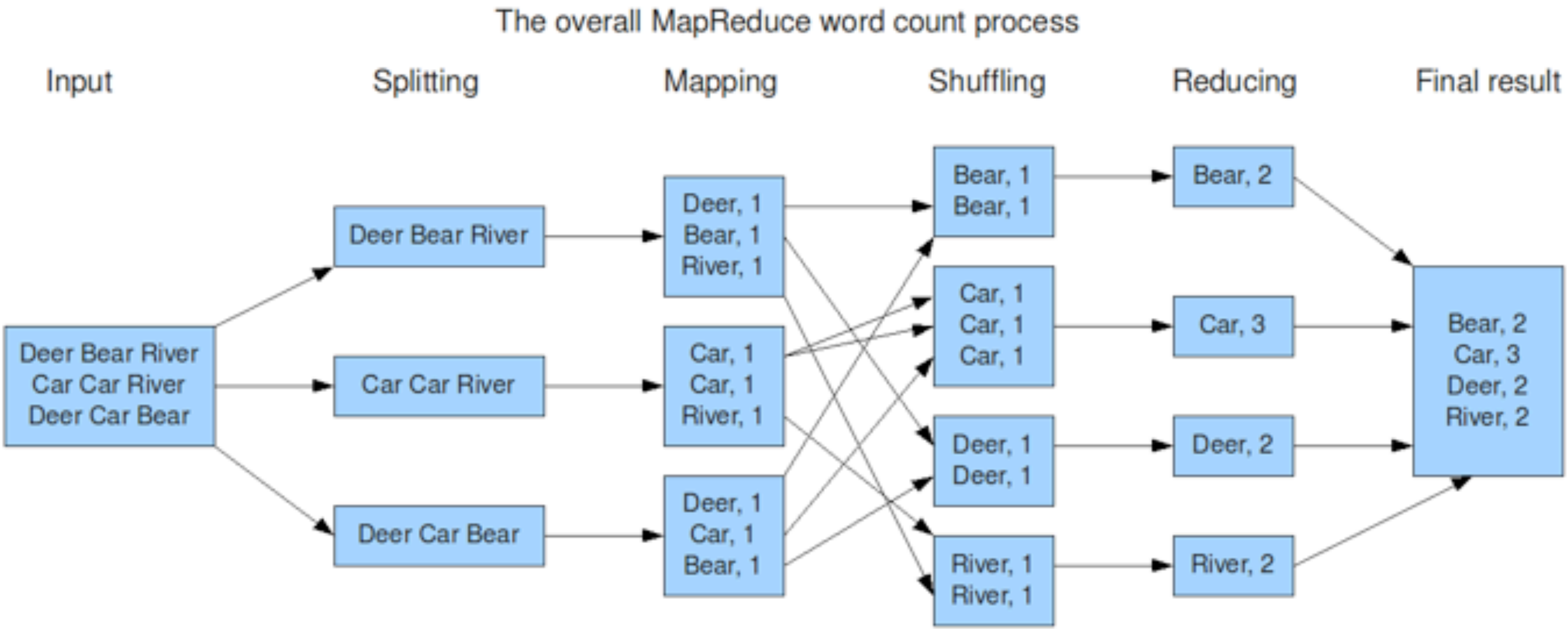
combiners – *perform reducer operations on the mapper node (optional step, to reduce network traffic and disk I/O).*

partitioners – *shuffle/sort/redirect mapper output*

reducers – *aggregate results*


Map
Shuffle
Reduce

III. WORD COUNT EXAMPLE


The overall MapReduce word count process
Input
Splitting
Mapping
Shuffling
Reducing
Final result
Deer Bear River
Car Car River
Deer Car Bear
Deer Bear River
Car Car River
Deer Car Bear
Deer, 1
Bear, 1
River, 1
Car, 1
Car, 1
River, 1
Deer, 1
Car, 1
Bear, 1
Bear, 1
Bear, 1
Car, 1
Car, 1
Car, 1
Deer, 1
Deer, 1
River, 1
River, 1
Bear, 2
Car, 3
Deer, 2
River, 2
Bear, 2
Car, 3
Deer, 2
River, 2

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.

MAP-REDUCE EXAMPLE: MAPPER INPUT

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.

MAP-REDUCE EXAMPLE: MAPPER OUTPUT

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.

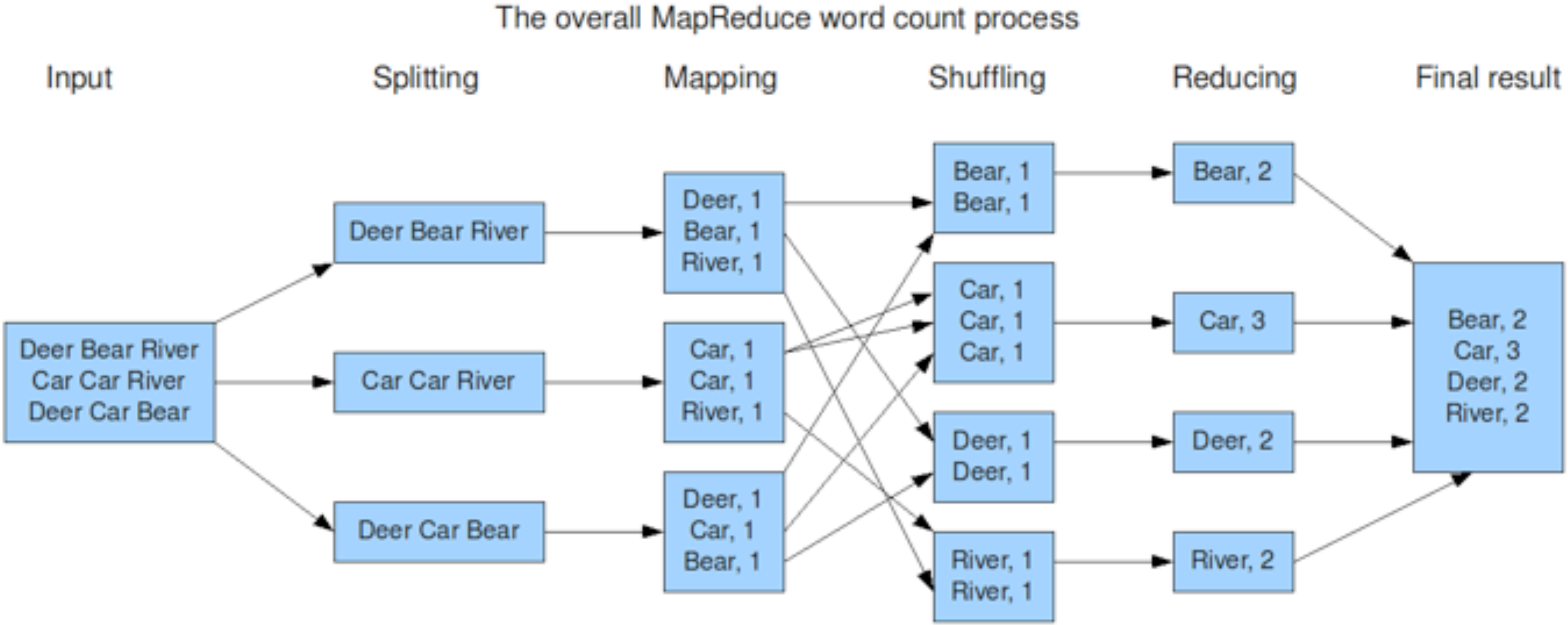
```
reducer(k2, k2_vals):
// k2 = word
// k2_vals = word counts

    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
```


The overall MapReduce word count process
Input
Splitting
Mapping
Shuffling
Reducing
Final result
Deer Bear River
Car Car River
Deer Car Bear
Deer Bear River
Car Car River
Deer Car Bear
Deer, 1
Bear, 1
River, 1
Car, 1
Car, 1
River, 1
Deer, 1
Car, 1
Bear, 1
Bear, 1
Bear, 1
Car, 1
Car, 1
Car, 1
Deer, 1
Deer, 1
River, 1
River, 1
Bear, 2
Car, 3
Deer, 2
River, 2
Bear, 2
Car, 3
Deer, 2
River, 2

I. ML REVIEW

WHAT IS MACHINE LEARNING?

from Wikipedia:

"Machine learning, a branch of artificial intelligence, is about the construction and study of systems that can *learn from data*."

source: http://en.wikipedia.org/wiki/Machine_learning

WHAT IS MACHINE LEARNING?

from Wikipedia:

“Machine learning, a branch of artificial intelligence, is about the construction and study of systems that can *learn from data*.”

“The core of machine learning deals with *representation* and *generalization*…”

source: http://en.wikipedia.org/wiki/Machine_learning

WHAT IS MACHINE LEARNING?

from Wikipedia:

"Machine learning, a branch of artificial intelligence, is about the construction and study of systems that can *learn from data*."

"The core of machine learning deals with *representation* and *generalization*…"

- *representation* – extracting structure from data
- *generalization* – making predictions from data

source: http://en.wikipedia.org/wiki/Machine_learning

Another viewpoint is that machine learning is a way to *compress data*

We *generalize* so we don't need so save every piece of data

Another viewpoint is that machine learning is a way to *compress data*

We *generalize* so we don't need so save every piece of data

We create *representations* that we can use instead of the original data

II. BLOOM FILTER

A **Bloom filter** is probabilistic data structure to check for set membership

A **Bloom filter** is probabilistic data structure to check for set membership

Traditionally, this would require storing the entire of set of object in memory to check if we have seen that object before

A **Bloom filter** is probabilistic data structure to check for set membership

Traditionally, this would require storing the entire of set of object in memory to check if we have seen that object before

How can we do so without storing *everything*?

Solution 1:

Always return the same answer, either always positive or always negative

Solution 1:

Always return the same answer, either always positive or always negative

Always positive -> high false positive rate = low precision, high recall

Solution 1:

Always return the same answer, either always positive or always negative

Always positive -> high false positive rate = low precision, high recall

Memory usage: None

Let's find some middle ground...

Solution 2:

Let's use some sort of cache, *store the last N items…*

If we have seen it in the last N items, return True

Solution 2:

Let's use some sort of cache, *store the last N items...*

If we have seen it in the last N items, return True

Memory usage: Constant (size of cache)

Error rate: ??? (more repeats -> less error)

Solution 3:

Let's use some sort of cache, *store the top N items...*

If we have seen it in the top N items, return True

Solution 3:

Let's use some sort of cache, *store the top N items...*

If we have seen it in the top N items, return True

Memory usage: Constant (size of cache)

Error rate: ??? (more skew -> less error)

A **Bloom filter** is probabilistic data structure to check for set membership

A **Bloom filter** is probabilistic data structure to check for set membership

1. Have a hash function H(x)
2. If we have seen x, store H(x) as True

We see username: arahuja, h(arahuja) = 4

0	1	2	3	4	5	6	7	8	9	10

We see username: arahuja, h(arahuja) = 4

0	1	2	3	4	5	6	7	8	9	10
				TRUE						

We see username: arahuja, h(arahuja) = 4

We see username: john, h(john) = 2

0	1	2	3	4	5	6	7	8	9	10
		TRUE		TRUE						

We see username: arahuja, h(arahuja) = 4

We see username: john, h(john) = 2

We see username: mary, h(mary) = 7

0	1	2	3	4	5	6	7	8	9	10
		TRUE		TRUE			TRUE			

We see username: arahuja, h(arahuja) = 4

We see username: john, h(john) = 2

We see username: mary, h(mary) = 7

We see username: sara, h(sara) = 2

0	1	2	3	4	5	6	7	8	9	10
		TRUE		TRUE			TRUE			

We see username: arahuja, h(arahuja) = 4

We see username: john, h(john) = 2

We see username: mary, h(mary) = 7

We see username: sara, h(sara) = 2

We check username: laura, h(laura) = 6, return False

0	1	2	3	4	5	6	7	8	9	10
		TRUE		TRUE			TRUE			

We see username: arahuja, h(arahuja) = 4

We see username: john, h(john) = 2

We see username: mary, h(mary) = 7

We see username: sara, h(sara) = 2

We check username: mary, h(mary) = 7, return True

0	1	2	3	4	5	6	7	8	9	10
		TRUE		TRUE			TRUE			

We see username: arahuja, h(arahuja) = 4

We see username: john, h(john) = 2

We see username: mary, h(mary) = 7

We see username: sara, h(sara) = 2

We check username: john, h(john) = 2, return True

0	1	2	3	4	5	6	7	8	9	10
		TRUE		TRUE			TRUE			

We see username: arahuja, h(arahuja) = 4

We see username: john, h(john) = 2

We see username: mary, h(mary) = 7

We see username: sara, h(sara) = 2

We check username: sara, h(sara) = 2, return True

0	1	2	3	4	5	6	7	8	9	10
		TRUE		TRUE			TRUE			

A **Bloom filter** is probabilistic data structure to check for set membership

Fact 1: No false negatives ... high recall

We see username: arahuja, h(arahuja) = 4

We see username: john, h(john) = 2

We see username: mary, h(mary) = 7

We see username: sara, h(sara) = 2

We check username: xfwer, h(xfwer) = 1, return False

0	1	2	3	4	5	6	7	8	9	10
		TRUE		TRUE			TRUE			

We see username: arahuja, h(arahuja) = 4

We see username: john, h(john) = 2

We see username: mary, h(mary) = 7

We see username: sara, h(sara) = 2

We check username: fdrr, h(fdrr) = 4, return True

0	1	2	3	4	5	6	7	8	9	10
		TRUE		TRUE			TRUE			

A **Bloom filter** is probabilistic data structure to check for set membership

Fact 1: No false negatives ... high recall

Fact 2: Some false positives ... **dependent on size of our hash**

We see username: arahuja, h1(arahuja) = 4

We see username: john, h1(john) = 2

We see username: mary, h(1mary) = 7

We see username: sara, h1(sara) = 2

0	1	2	3	4	5	6	7	8	9	10
		TRUE		TRUE			TRUE			

We see username: arahuja, h1(arahuja) = 4, h2(arahuja) = 9

We see username: john, h1(john) = 2, h2(john) = 3

We see username: mary, h(1mary) = 7, h2(mary) = 6

We see username: sara, h1(sara) = 2, h2(sara) = 4

0	1	2	3	4	5	6	7	8	9	10
			TRUE	TRUE		TRUE			TRUE	

0	1	2	3	4	5	6	7	8	9	10
		TRUE		TRUE			TRUE			

We check username: sara, h1(sara) = 2, h2(sara) = 4 return True

0	1	2	3	4	5	6	7	8	9	10
			TRUE	TRUE		TRUE			TRUE	

0	1	2	3	4	5	6	7	8	9	10
		TRUE		TRUE			TRUE			

We check username: sara, h1(sara) = 2, h2(sara) = 4 return True

We check username: sara, h1(xfwer) = 1, h2(xfwer) = 2 return False

0	1	2	3	4	5	6	7	8	9	10
			TRUE	TRUE		TRUE			TRUE	

0	1	2	3	4	5	6	7	8	9	10
		TRUE		TRUE			TRUE			

We check username: sara, h1(sara) = 2, h2(sara) = 4 return True

We check username: sara, h1(xfwer) = 1, h2(xfwer) = 2 return False

We check username: sara, h1(fdrr) = 2, h2(fdrr) = 7 return False

0	1	2	3	4	5	6	7	8	9	10
			TRUE	TRUE		TRUE			TRUE	

0	1	2	3	4	5	6	7	8	9	10
		TRUE		TRUE			TRUE			

A **Bloom filter** is probabilistic data structure to check for set membership

Fact 1: No false negatives ... perfect recall

Fact 2: Some false positives ... **dependent on size of our hash**

Fact 3: More hashes -> higher precision

III. COUNT MIN SKETCH

A **Bloom filter** is probabilistic data structure to check for set membership

How we do transform to this idea to check for count of an element as opposed to just membership?

We see username: arahuja, h(arahuja) = 4

0	1	2	3	4	5	6	7	8	9	10

We see username: arahuja, h(arahuja) = 4

0	1	2	3	4	5	6	7	8	9	10
				1						

We see username: arahuja, h(arahuja) = 4

We see username: john, h(john) = 2

0	1	2	3	4	5	6	7	8	9	10
		1		1						

We see username: arahuja, h(arahuja) = 4

We see username: john, h(john) = 2

We see username: mary, h(mary) = 7

0	1	2	3	4	5	6	7	8	9	10
		1		1			1			

We see username: arahuja, h(arahuja) = 4

We see username: john, h(john) = 2

We see username: mary, h(mary) = 7

We see username: sara, h(sara) = 2

0	1	2	3	4	5	6	7	8	9	10
		2		1			1			

We see username: arahuja, h(arahuja) = 4

We see username: john, h(john) = 2

We see username: mary, h(mary) = 7

We see username: sara, h(sara) = 2

We see username: arahuja, h(arahuja) = 4

0	1	2	3	4	5	6	7	8	9	10
		2		2			1			

We see username: arahuja, h(arahuja) = 4

We see username: john, h(john) = 2

We see username: mary, h(mary) = 7

We see username: sara, h(sara) = 2

We see username: arahuja, h(arahuja) = 4

We check username: laura, h(laura) = 6, return 0

0	1	2	3	4	5	6	7	8	9	10
		2		2			1			

We see username: arahuja, h(arahuja) = 4

We see username: john, h(john) = 2

We see username: mary, h(mary) = 7

We see username: sara, h(sara) = 2

We see username: arahuja, h(arahuja) = 4

We check username: mary, h(mary) = 7, return 1

0	1	2	3	4	5	6	7	8	9	10
		1		2			1			

We see username: arahuja, h(arahuja) = 4

We see username: john, h(john) = 2

We see username: mary, h(mary) = 7

We see username: sara, h(sara) = 2

We see username: arahuja, h(arahuja) = 4

We check username: sara, h(sara) = 2, return 2

0	1	2	3	4	5	6	7	8	9	10
		1		2			1			

Fact 1: No false 0's ... we always overestimate

Fact 1: No false 0's ... we always overestimate

How do fix this?

Fact 1: No false 0's ... we always overestimate

How do fix this? **MORE HASHES!**

We see username: arahuja, h(arahuja) = 4, 7

0	1	2	3	4	5	6	7	8	9	10
							1			

0	1	2	3	4	5	6	7	8	9	10
				1						

We see username: arahuja, h(arahuja) = 4, 7

We see username: john, h(john) = 2, 5

0	1	2	3	4	5	6	7	8	9	10
					1		1			

0	1	2	3	4	5	6	7	8	9	10
		1		1						

We see username: arahuja, h(arahuja) = 4, 7

We see username: john, h(john) = 2, 5

We see username: mary, h(mary) = 7, 3

0	1	2	3	4	5	6	7	8	9	10
			1		1		1			

0	1	2	3	4	5	6	7	8	9	10
		1		1			1			

We see username: arahuja, h(arahuja) = 4, 7

We see username: john, h(john) = 2, 5

We see username: mary, h(mary) = 7, 3

We see username: sara, h(sara) = 2, 4

0	1	2	3	4	5	6	7	8	9	10
			1	1	1		1			

0	1	2	3	4	5	6	7	8	9	10
		2		1			1			

We see username: arahuja, h(arahuja) = 4, 7

We see username: john, h(john) = 2, 5

We see username: mary, h(mary) = 7, 3

We see username: sara, h(sara) = 2, 4

We see username: arahuja, h(arahuja) = 4, 7

0	1	2	3	4	5	6	7	8	9	10
			1	1	1		2			

0	1	2	3	4	5	6	7	8	9	10
		2		2			1			

We check username: mary, h(mary) = 7, 3 return ??

0	1	2	3	4	5	6	7	8	9	10
			1	1	1		2			

0	1	2	3	4	5	6	7	8	9	10
		2		2			1			

We check username: laura, h(laura) = 6, 2 return ??

0	1	2	3	4	5	6	7	8	9	10
			1	1	1		2			

0	1	2	3	4	5	6	7	8	9	10
		2		2			1			

We check username: sara, h(sara) = 2,4 return ??

0	1	2	3	4	5	6	7	8	9	10
			1	1	1		2			

0	1	2	3	4	5	6	7	8	9	10
		2		2			1			

We check username: sdfsd, h(sdfsd) = 1, 2 return ??

0	1	2	3	4	5	6	7	8	9	10
			1	1	1		2			

0	1	2	3	4	5	6	7	8	9	10
		2		2			1			

Fact 1: No false 0’s … we always overestimate

Fact 2: Use multiple hashes, return MIN (Fact 1 still holds)

This is known as **Count Min Sketch**

Fact 1: No false 0's ... we always overestimate

Fact 2: Use multiple hashes, return MIN (Fact 1 still holds)

This is known as **Count Min Sketch**

How can we improve this?

How can we use the sketching algorithms from last class to

in real-time, return the number **unique** visitors to our website.