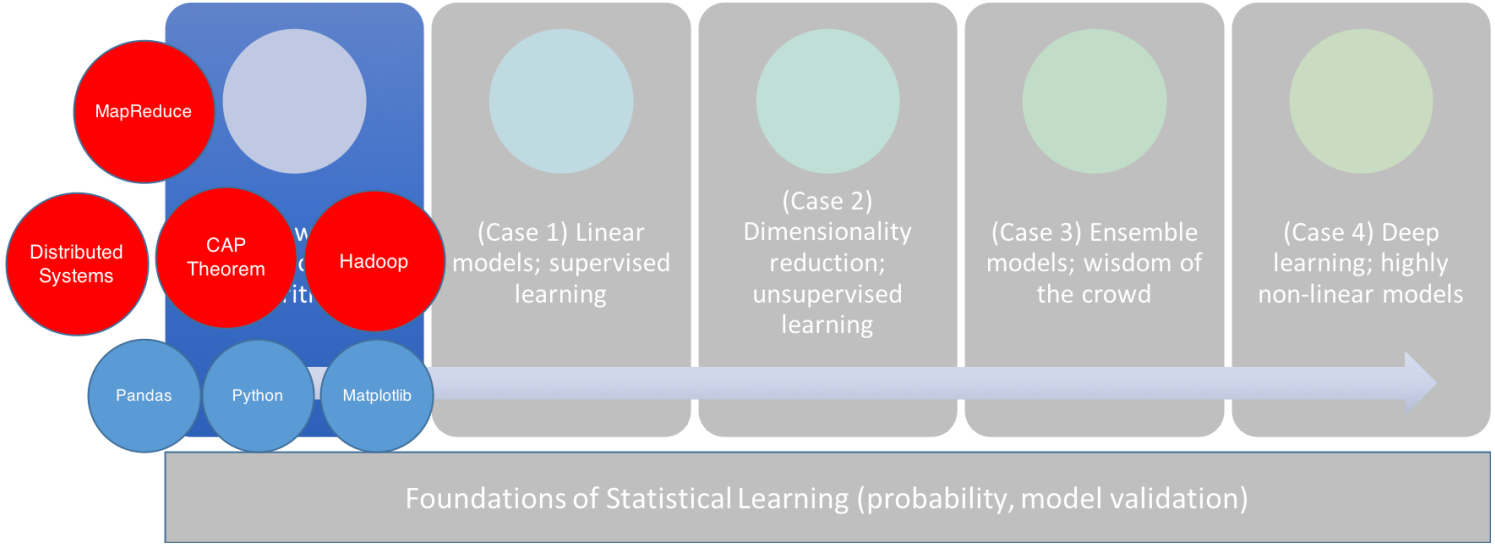
Unit 3
MapReduce, Hadoop (v1 and Yarn)

IST 718 – Big Data Analytics

Daniel E. Acuna

http://acuna.io

Course roadmap



- Small/medium data
- Low model complexity
- High interpretability
- Low computational power

- Big data
- High model complexity
- Low interpretability
- High computational power

Objectives of this unit

- Introduction to distributed systems
- The CAP theorem (**storage limitation**)
- Distributed systems may fail more (**compute limitation**)
- Hadoop and Yarn
- MapReduce

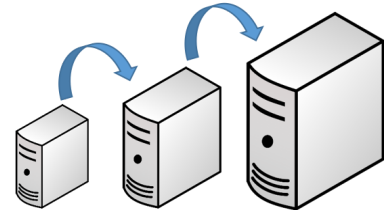
Distributed systems

- A distributed system is a collection of independent computers that appear to the users as a single coherent system
- Advantages (compare to single system):
  - It could be cheaper: building a supercomputer is expensive
  - Faster processing: parallel processing
  - Reliability: if one node fails, then we can process in a different node
  - Incremental growth: add more computers as needed
- Disadvantage:
  - Software must be customized
  - Network: often this is the bottleneck
  - More components to fail
  - Complex security

Scaling Services: How do you address growth?

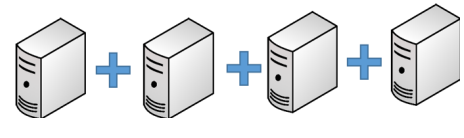
Vertical "Scale Up"

- Add more resources to an existing system running the service.
- Easier, but limited scale
- Single point of failure



Horizontal "Scale Out"

- Run the service over multiple systems, and orchestrate communication between them
- Harder, but massive scale
- Overhead to manage nodes



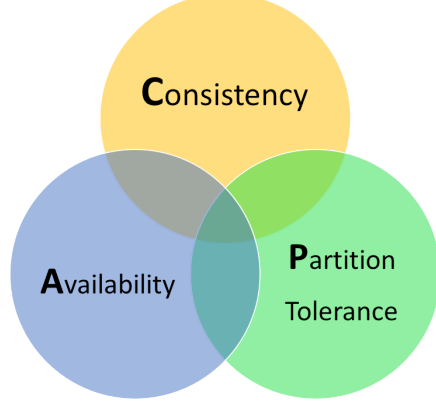
CAP Theorem of Distributed Systems (1)

- From a **storage perspective**, there are many goals for a distributed system.
- Almost all goals of distributed systems can be fit into one of these categories
  - **Data Consistency**: all nodes see the same data at the same time.
  - **Data Availability**: assurances that every request can be processed.
  - **Partition Tolerance**: network failures are tolerated, the system continues to operate

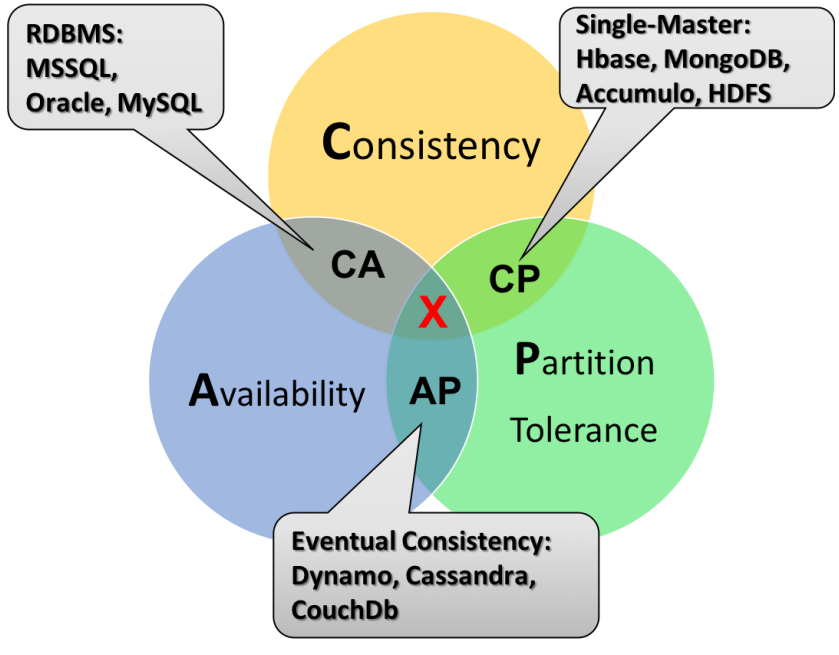
CAP Theorem of Distributed Systems (2)

However, there is a fundamental constraint: only two of the goals can be fulfilled at the same time

- **Data Consistency**: all nodes see the same data at the same time
- **Data Availability**: assurances that every request can be processed
- **Partition Tolerance**: network failures are tolerated, the system continues to operate



CAP Theorem of Distributed Systems (6)

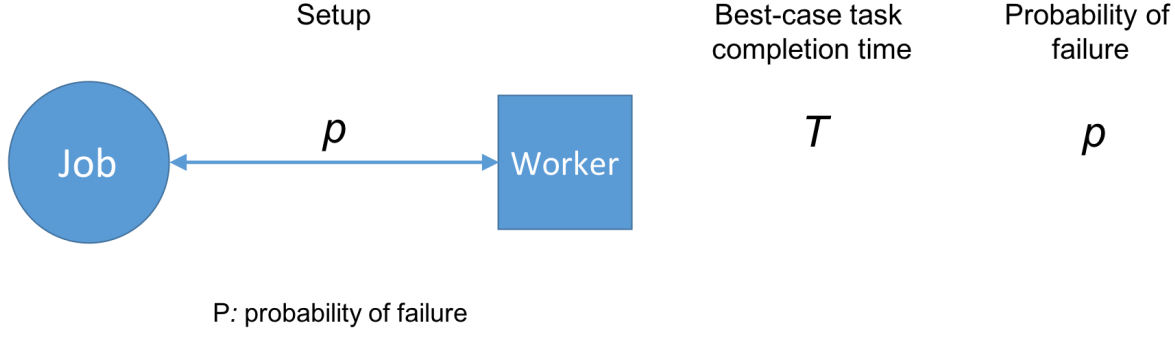


Distributed systems

How does ”More components to fail” affect us?

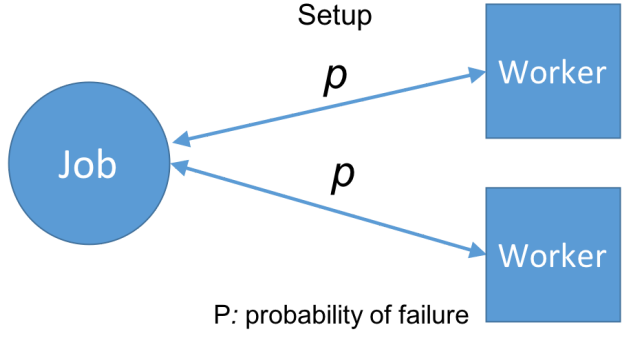
A tradeoff (1)

- We want to process massive amounts of data in parallel
- However, we make ourselves more vulnerable to failures



A tradeoff (2)

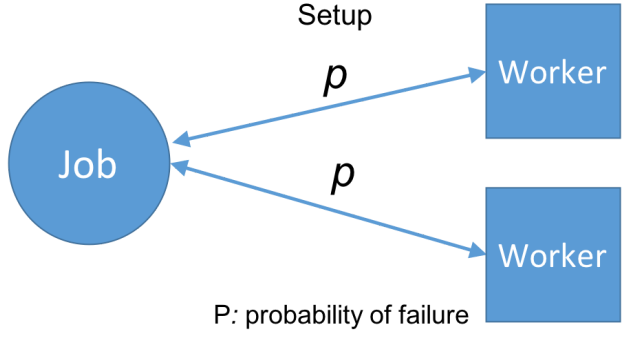
- We want to process massive amounts of data in parallel
- However, we make ourselves more vulnerable to failures



Best-case task Completion time	Probability of failure
?	?

A tradeoff (3)

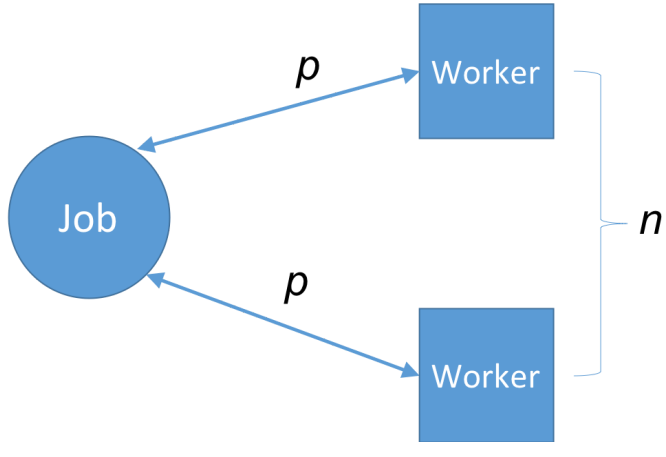
- We want to process massive amounts of data in parallel
- However, we make ourselves more vulnerable to failures



Best-case task Completion time	Probability of failure
$T/2$	$1-(1-p)^2$

A tradeoff (4)

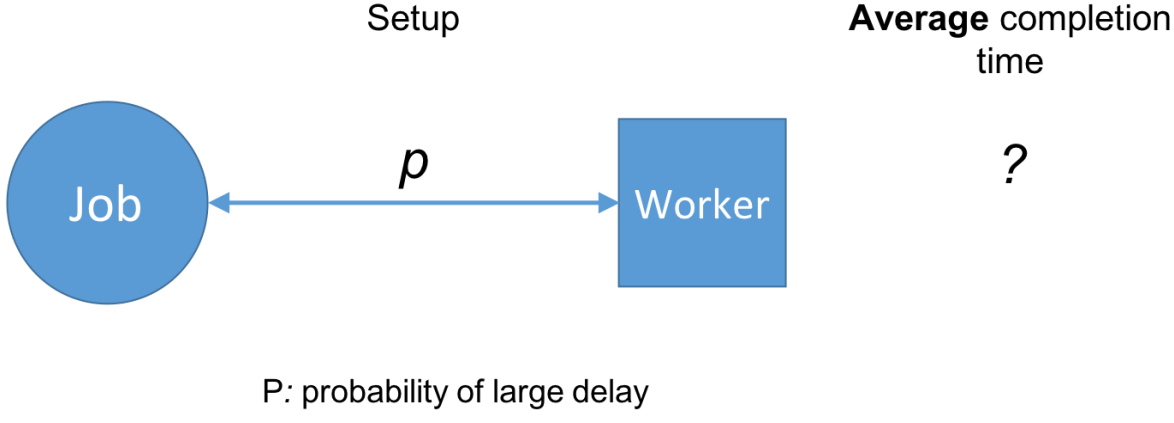
- We want to process massive amounts of data in parallel
- However, we make ourselves more vulnerable to failures



Best-case task Completion time	Probability of failure
T/n	$1-(1-p)^n$

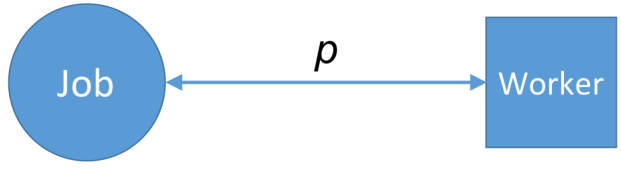
A more realistic example: long tails (1)

- We want to process massive amounts of data in parallel
- Normal responses take 10 ms, but with probability 1%, a worker takes 1 s



A more realistic example: long tails (2)

- We want to process massive amounts of data in parallel
- Normal responses take 10 ms, but with probability 1%, a worker takes 1 s



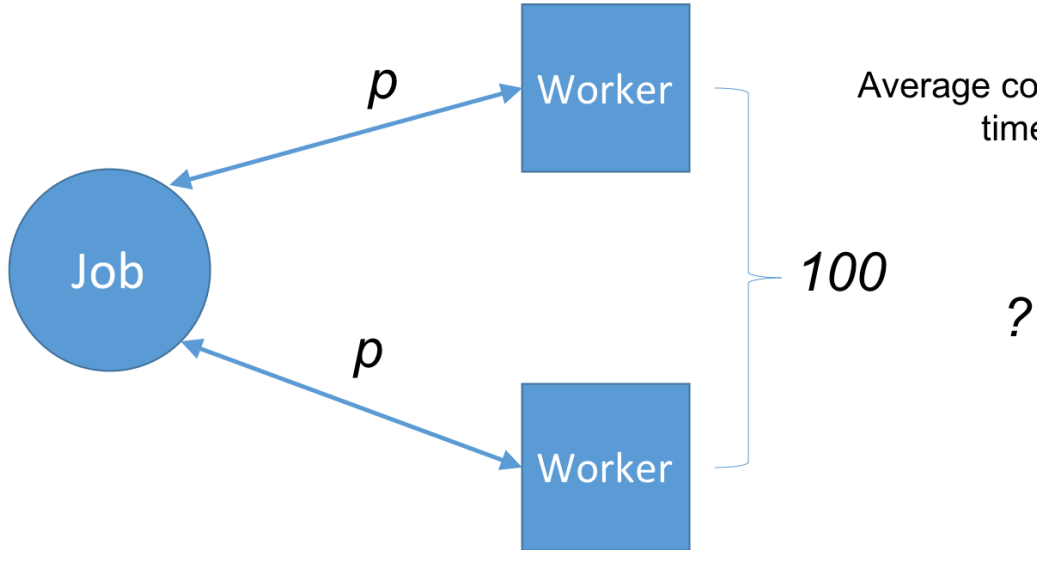
Average completion time

19.9 ms 10*0.99 + 1000*0.01

P*:* probability of large delay

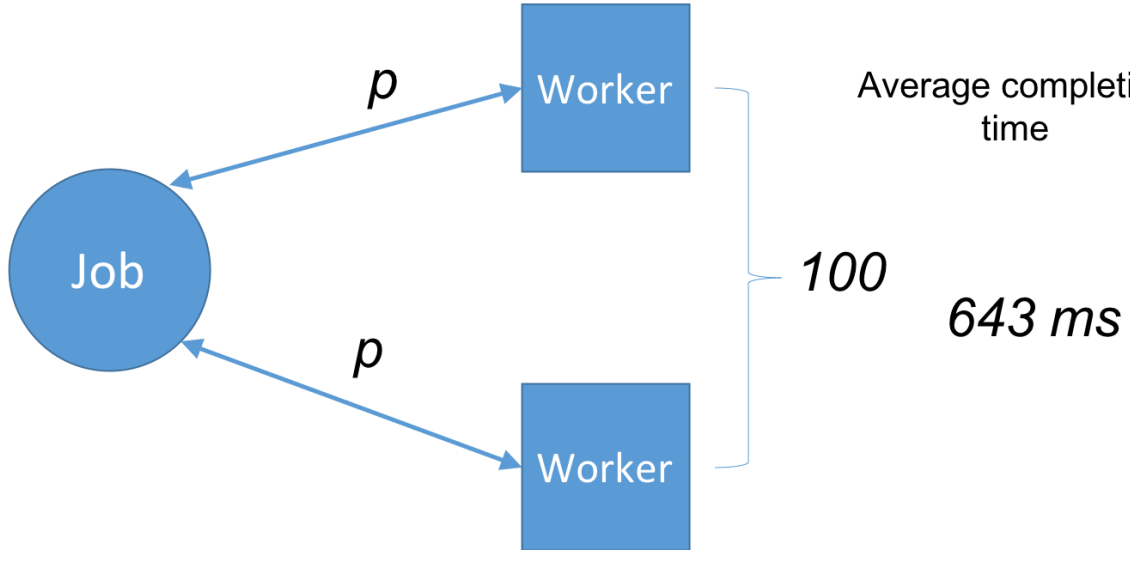
A more realistic example: long tails (3)

- We want to process massive amounts of data in parallel
- Normal responses take 10 ms, but with probability 1%, a worker takes 1 s



A more realistic example: long tails (4)

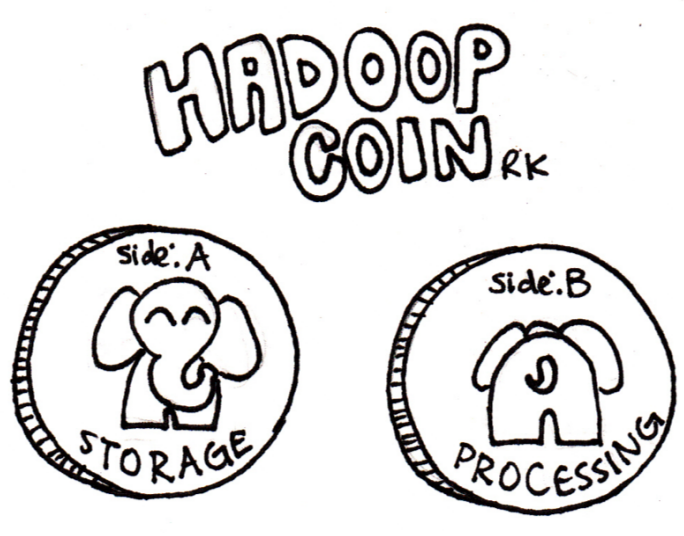
- We want to process massive amounts of data in parallel
- Normal responses take 10 ms, but with probability 1%, a worker takes 1 s



What is Hadoop?

Hadoop

- Distribute the data in a consistent manner
  HDFS
- Move processing to the data and recover from compute failures
  MapReduce / YARN



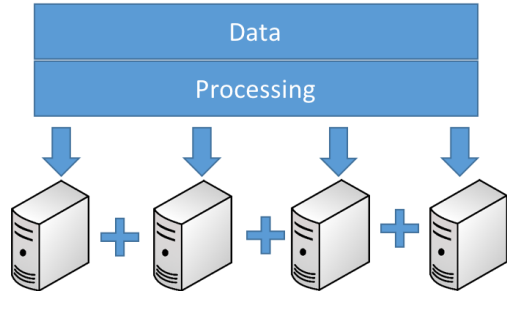
Birthplace of Hadoop

- Google, Facebook, Yahoo!
- These companies had so much data, that **enterprise DBMSs** could not meet their reporting requirements

Hadoop Handles Big Data By Scaling Out

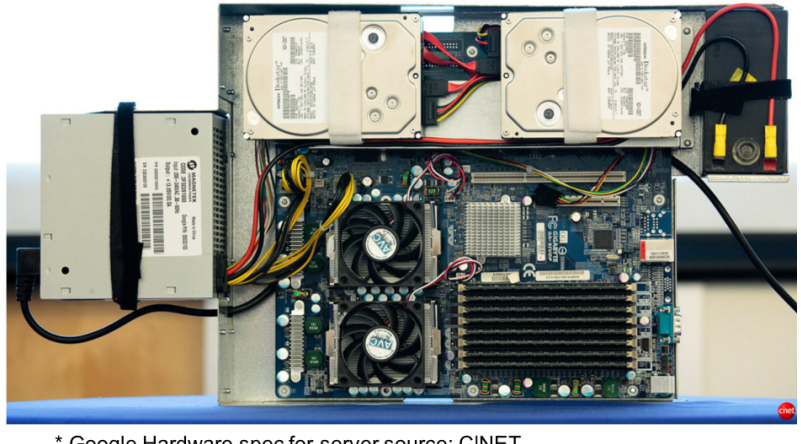
- Problem: File Too Large to fit on a single storage platform?
- Solution: Distribute the file over several computers

- Problem: Server not “fast” enough to process your data
- Solution: Distribute data processing over several computers



Hadoop is Designed to Use Commodity Hardware

- Hadoop Hardware
  - Modular
  - Easy to add and remove nodes.
  - Failure is not only acceptable, but expected.
- This is contrary to enterprise hardware
  - High-redundancy / Fault-tolerant
  - Vertical Scaling
  - Storage arrays



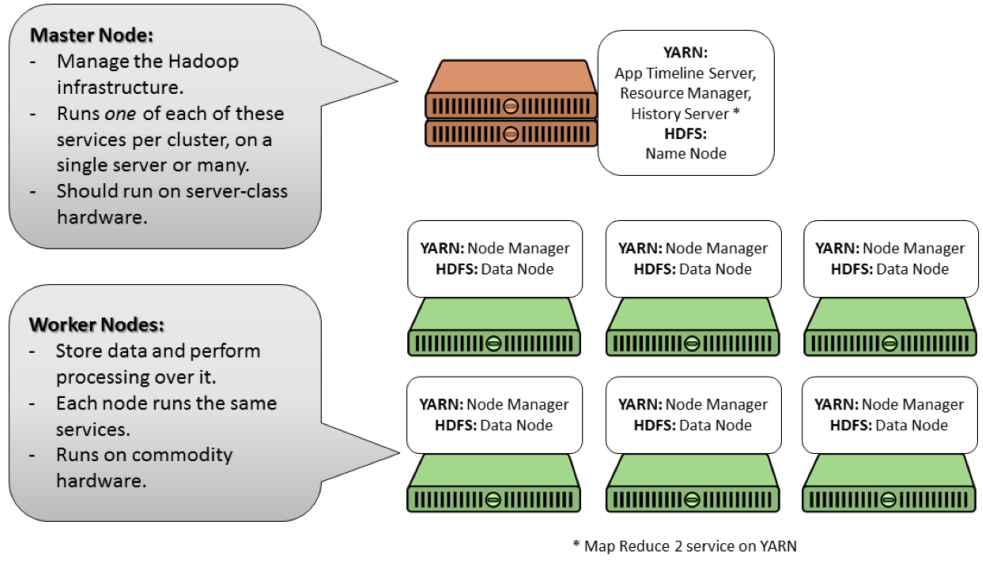
* Google Hardware spec for server source: C|NET

How Does Hadoop Store, Process and Manage Big Data?

Hadoop Clusters

Node Types:

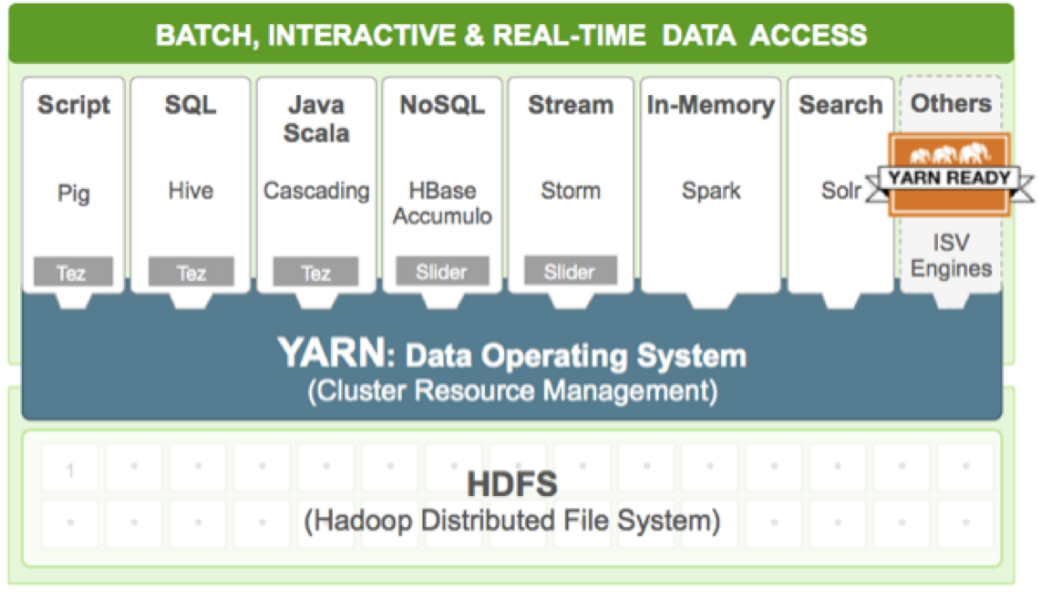
1. Master Nodes
2. Worker Nodes
3. Client Nodes



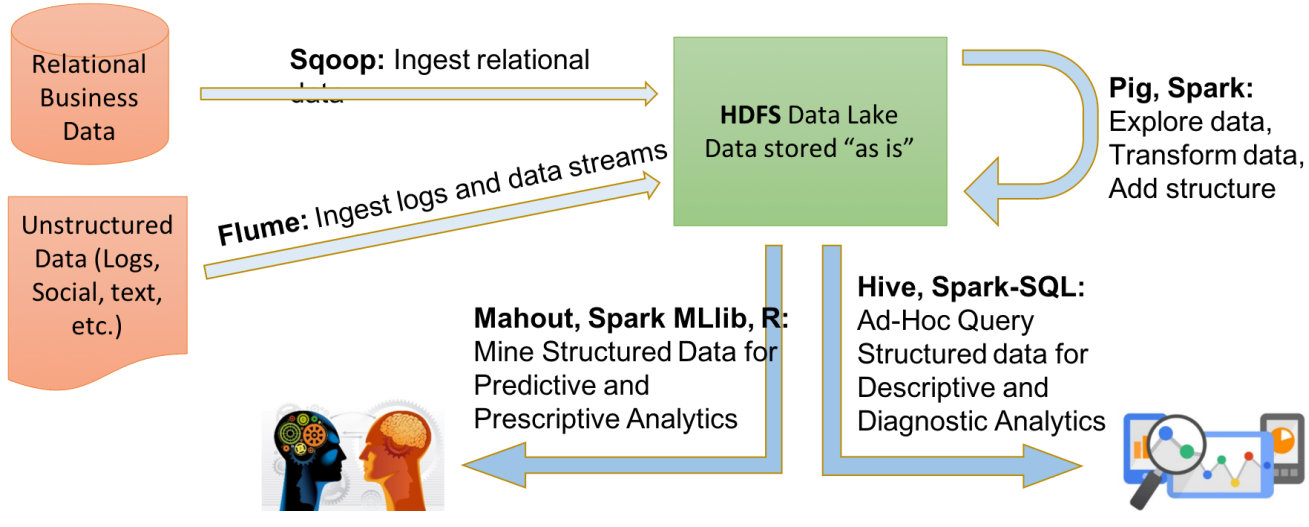
The Hadoop Ecosystem: Open Source Tools

YARN: The Data Operating System

- Hadoop 2.0 Introduces YARN (Yet Another Resource Negotiator)
- Orchestrates processing over the nodes
- Uses HDFS for storage
- Runs a variety of Applications



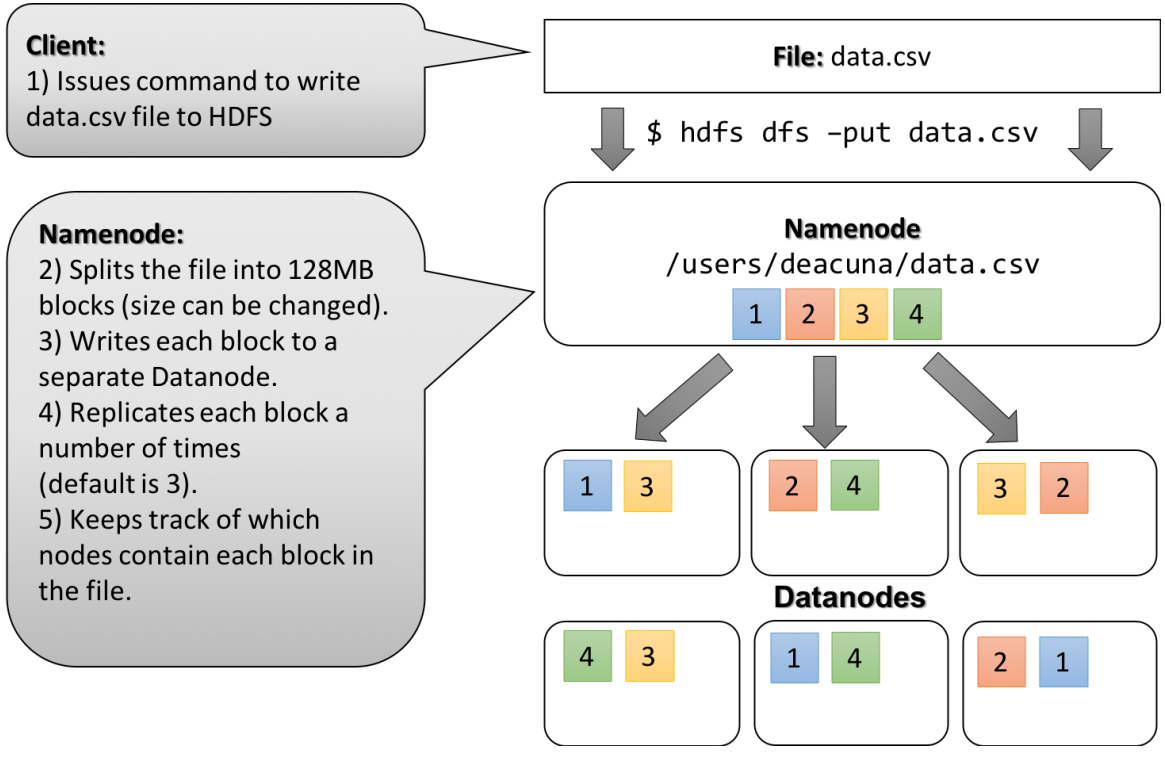
An Over-Simplified Version of the Hadoop Ecosystem in Action



HDFS: Hadoop Distributed File System

- Based on Google's GFS
- Data Distributed over Physical Nodes
- Designed for Failover
- Data Stored "as is"
- Data Split into Blocks
- Default Replication factor is 3

HDFS At Work


Client:
1) Issues command to write data.csv file to HDFS
File: data.csv
$ hdfs dfs -put data.csv
Namenode:
2) Splits the file into 128MB blocks (size can be changed).
3) Writes each block to a separate Datanode.
4) Replicates each block a number of times (default is 3).
5) Keeps track of which nodes contain each block in the file.
Namenode
/users/deacuna/data.csv
1 2 3 4
1 3
2 4
3 2
Datanodes
4 3
1 4
2 1

MapReduce

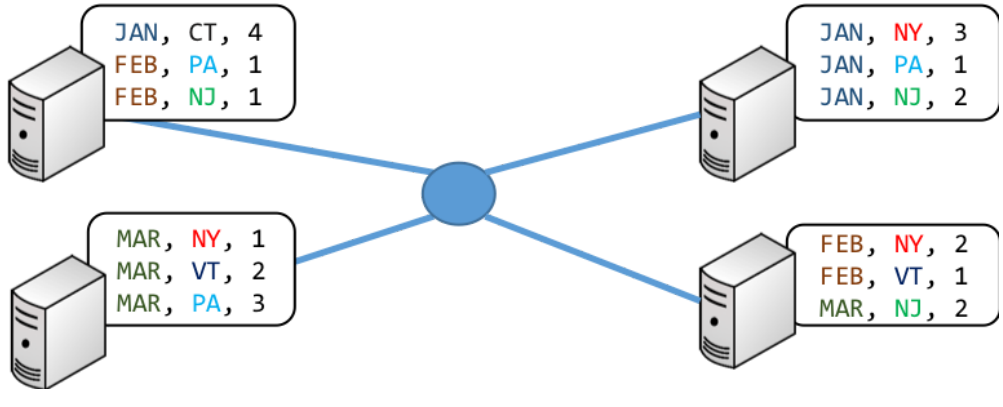
The problem

- We want to use our distributed system to process arbitrarily large amounts of data
- We can develop custom scripts for communicating among computers and solving a problem
- The problem is that this script can be different for each task
- > *"MapReduce is a programming model and an associated implementation for processing and generating large data sets"*

  J. Dean, S. Ghemawat, *"MapReduce: Simplified Data Processing on Large Clusters"*, OSDI 2004

The solution

- Create a simple programming model where there is **no communication** between tasks, enhancing fault tolerance and lowering complexity
- An example application:
  - Very large and distributed dataset of orders per month and state
  - We want to compute total number of orders per month
  - Any thoughts?

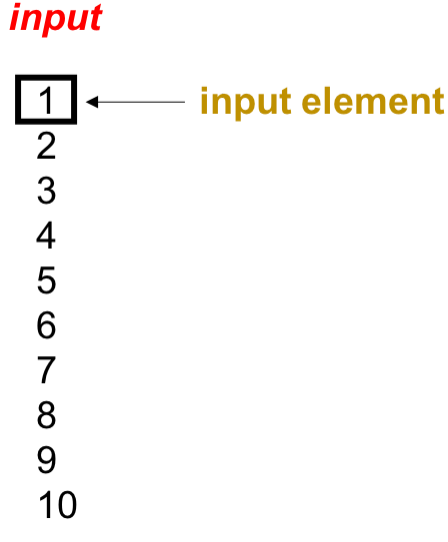


MapReduce (programming model)

- Computation takes a set of input key/value pairs and produces a set of output key/value pairs
- The user implements two functions: Map and Reduce
- **Map**: takes an input element and produces a set of intermediate key/value pairs
- **Reduce**: accepts an intermediate key and a pair of values for the same key and combines them into zero or one value.
- The program stops when no more key-value pairs can be reduced

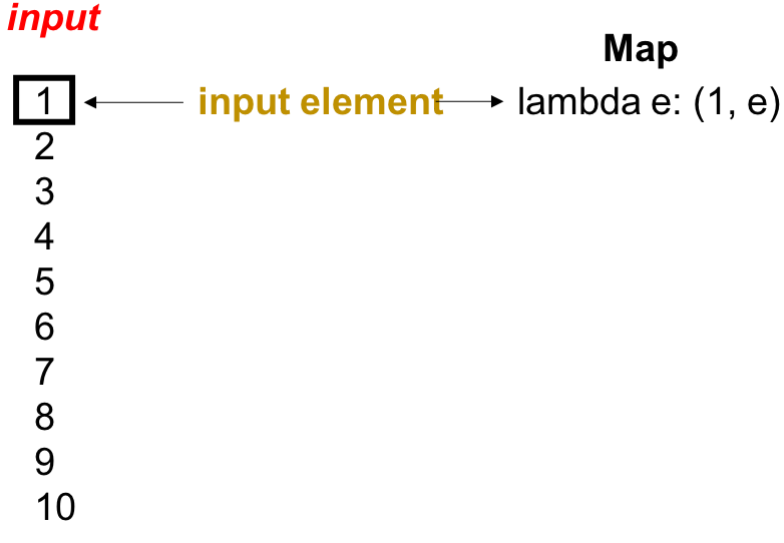
MapReduce: example

- **Map**: takes an **input element** and produces a set of ***intermediate* key/value *pairs***



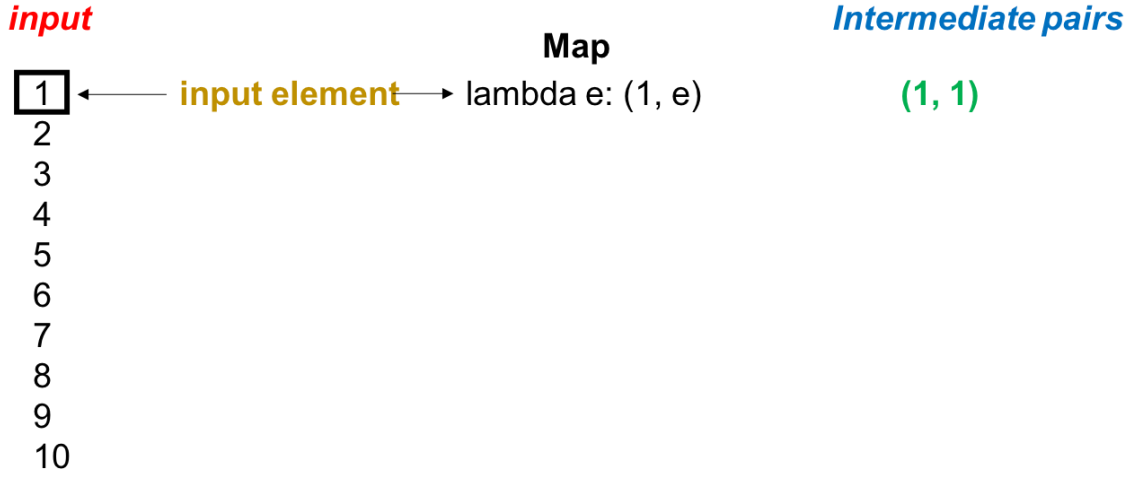
MapReduce: example

- **Map**: takes an **input element** and produces a set of ***intermediate* key/value *pairs***



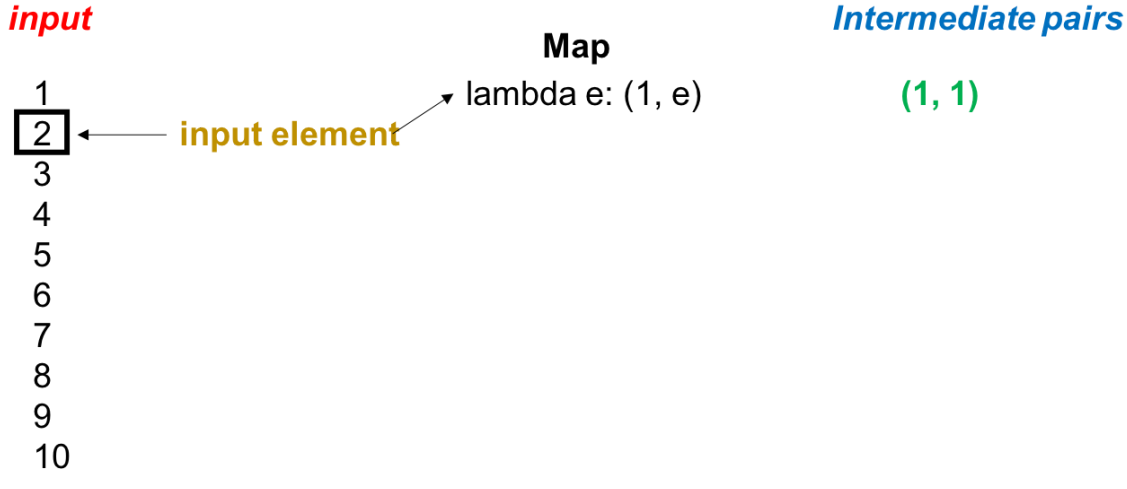
MapReduce: example

- **Map**: takes an **input element** and produces a set of ***intermediate* key/value *pairs***



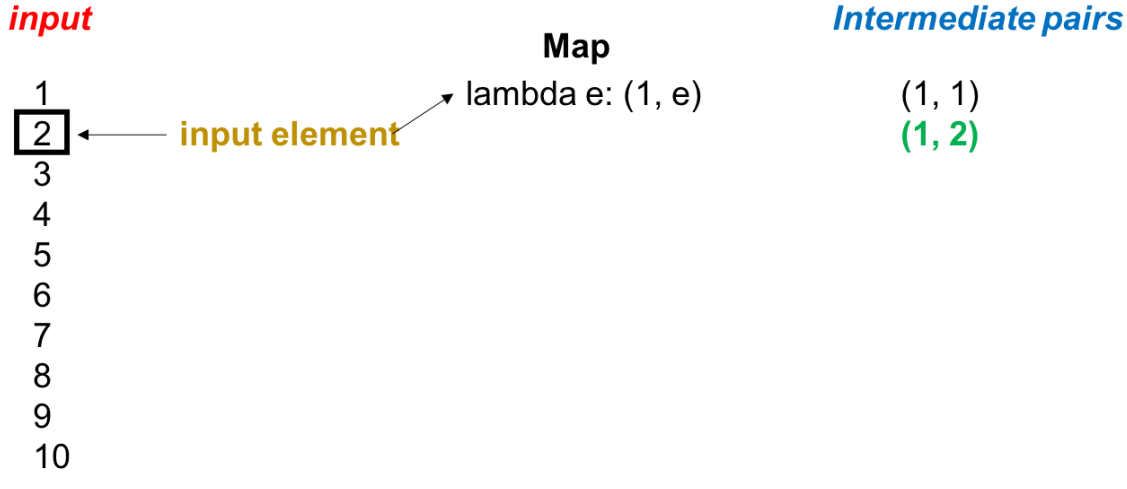
MapReduce: example

- **Map**: takes an **input element** and produces a set of ***intermediate* key/value *pairs***



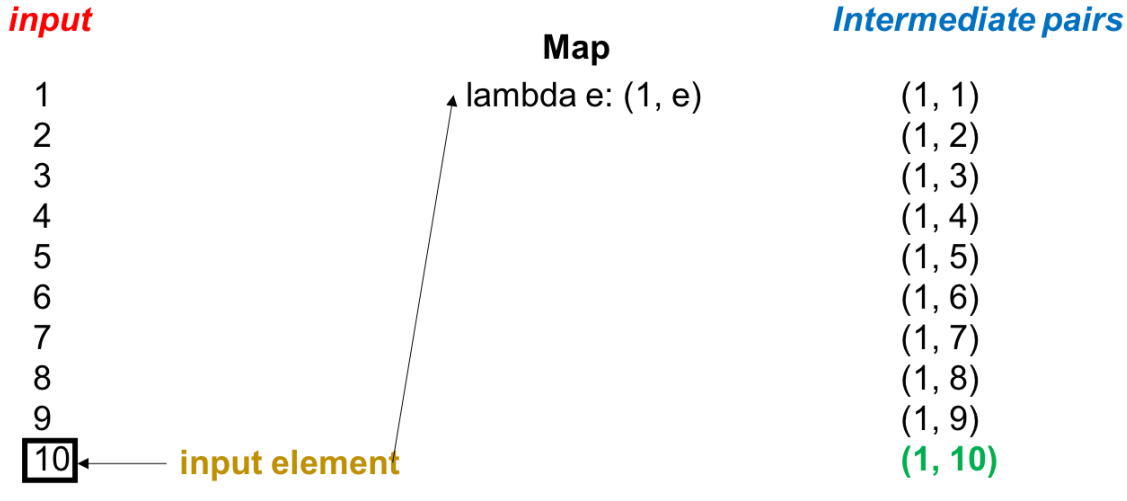
MapReduce: example

- **Map**: takes an **input element** and produces a set of ***intermediate* key/value *pairs***



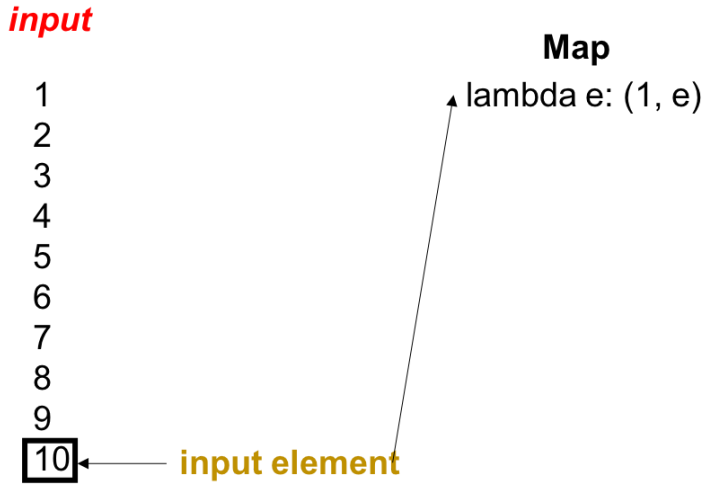
MapReduce: example

- **Map**: takes an **input element** and produces a set of ***intermediate* key/value *pairs***



MapReduce: example

- **Map**: takes an **input element** and produces a set of ***intermediate* key/value *pairs***

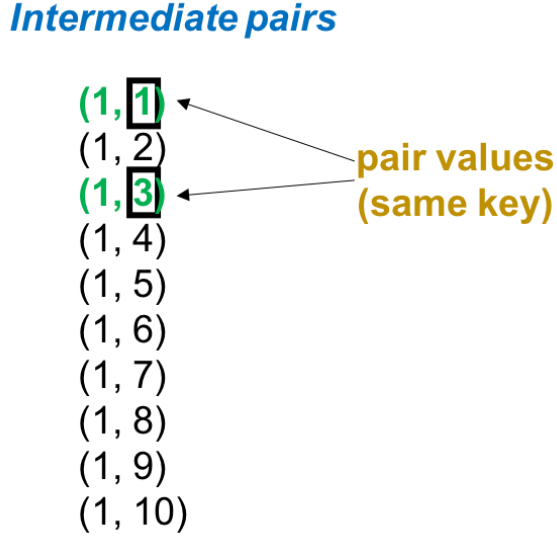


Identify

1. input
2. input element
3. map function
   - 3.1 rule for key
   - 3.2 rule for value
4. intermediate key

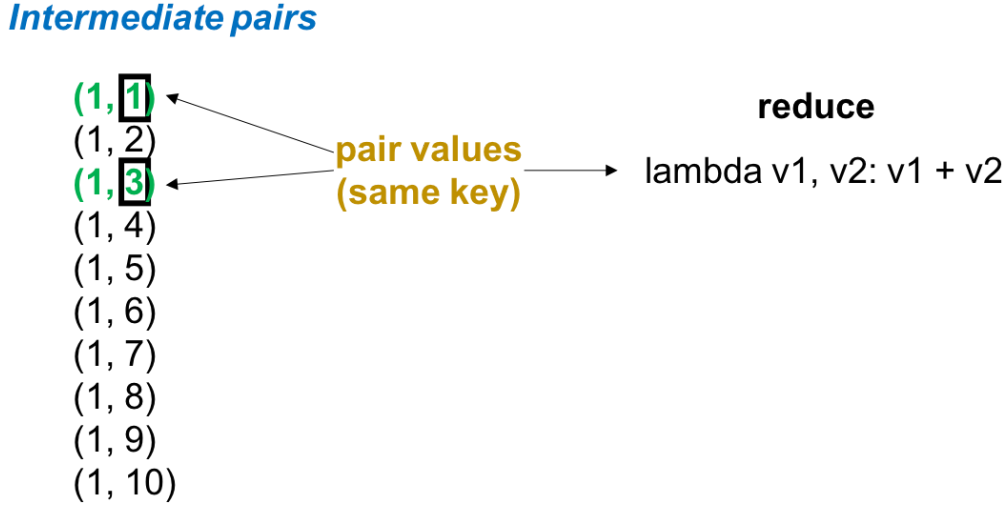
MapReduce: example

- **Reduce**: accepts an intermediate key and a pair of values for the same key and combines them into zero or one value.



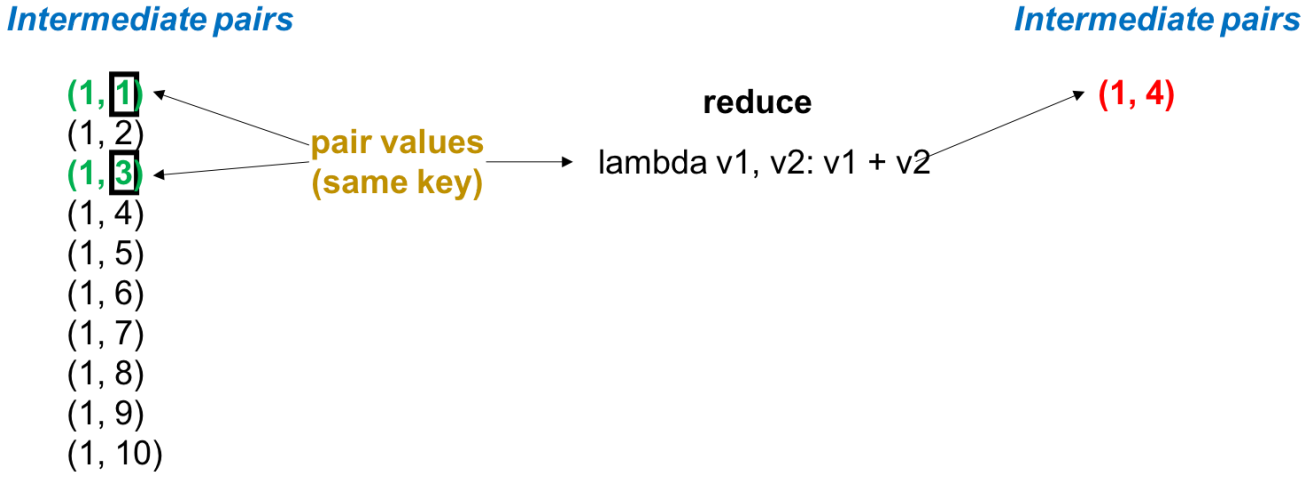
MapReduce: example

- **Reduce**: accepts an intermediate key and a pair of values for the same key and combines them into zero or one value.



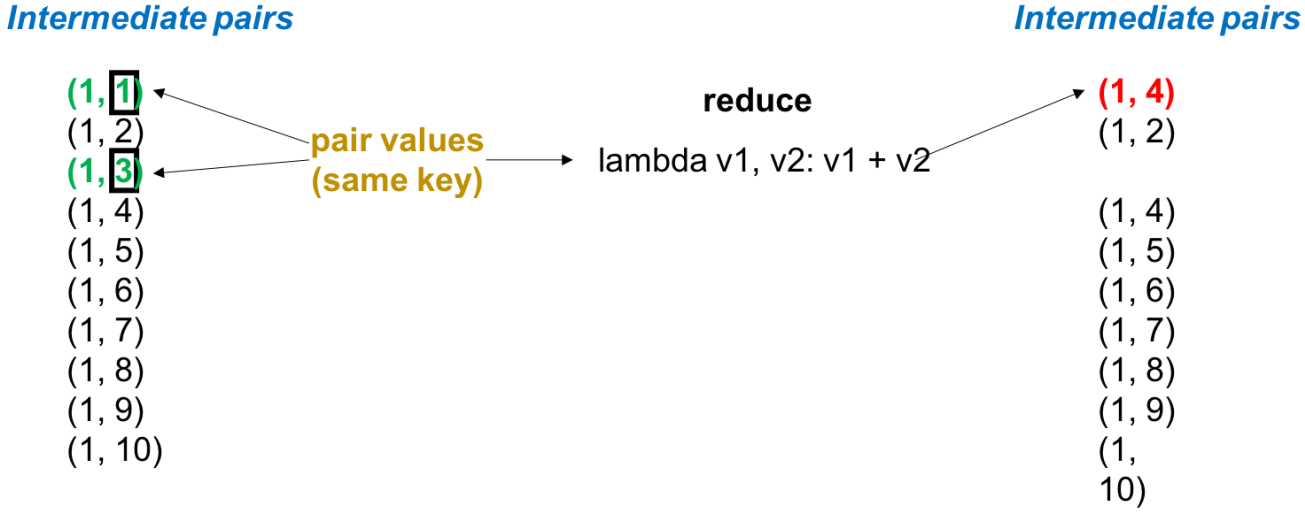
MapReduce: example

- **Reduce**: accepts an intermediate key and a pair of values for the same key and combines them into zero or one value.



MapReduce: example

- **Reduce**: accepts an intermediate key and a pair of values for the same key and combines them into zero or one value.

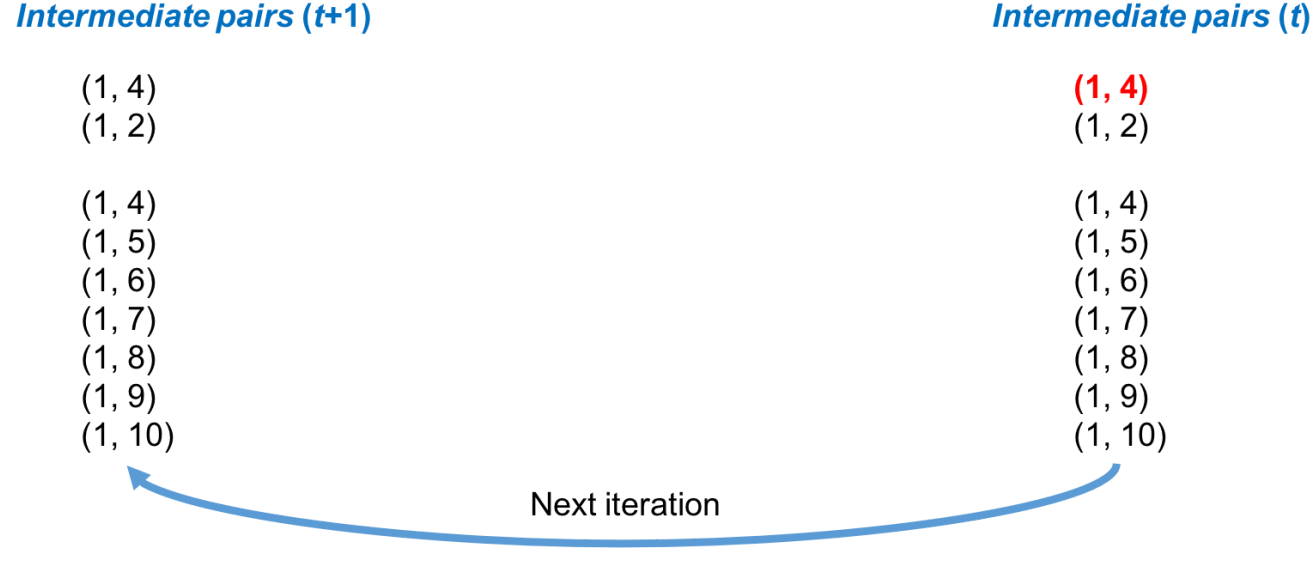


MapReduce: example

- **Reduce**: accepts an intermediate key and a pair of values for the same key and combines them into zero or one value.

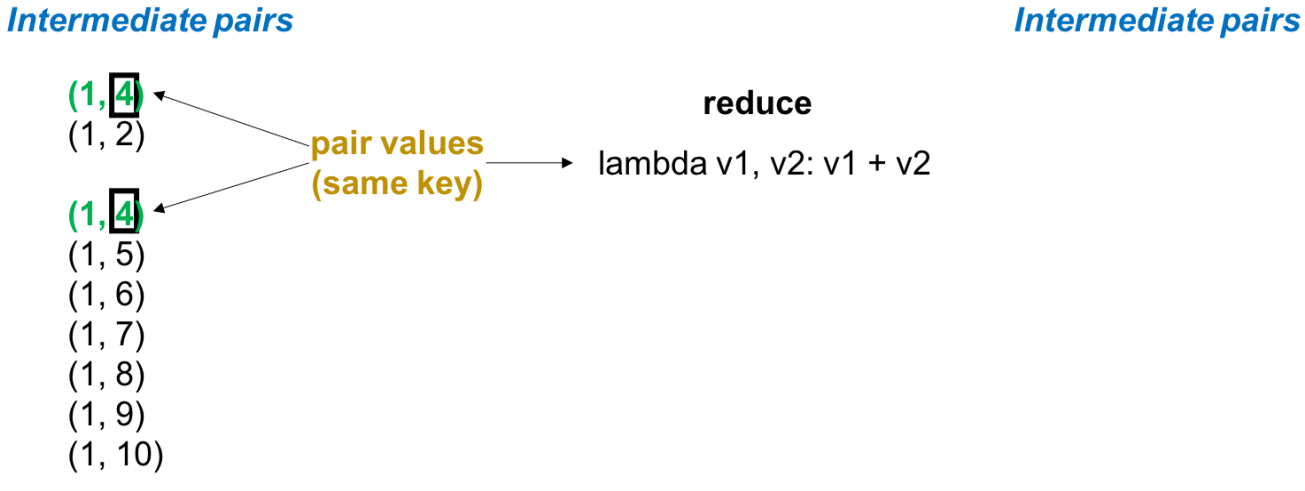
MapReduce: example

- **(Repeat) Reduce**: accepts an intermediate key and a pair of values for the same key and combines them into zero or one value.



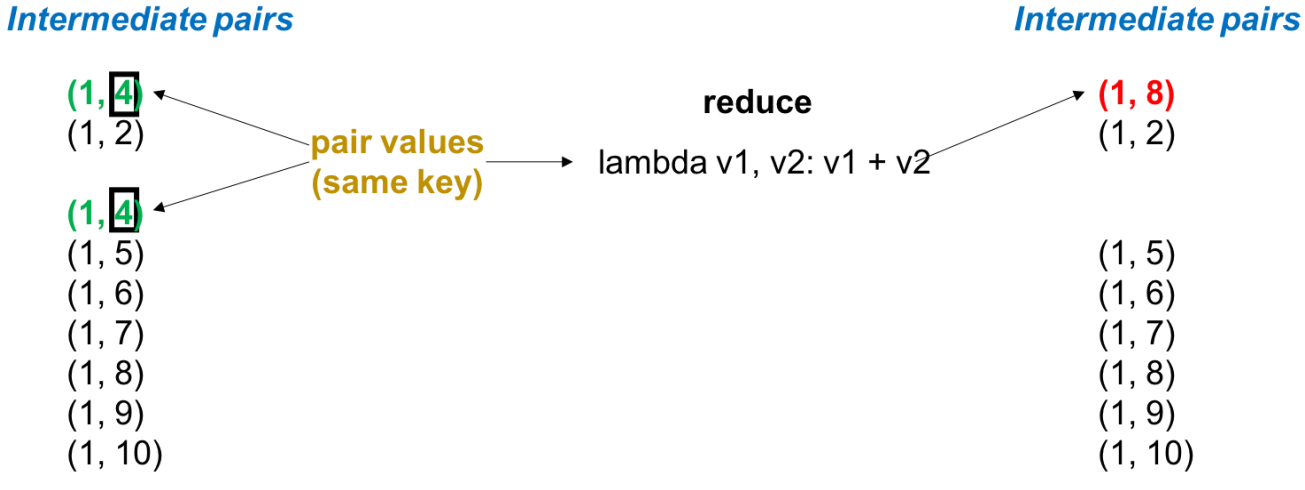
MapReduce: example

- **(Repeat) Reduce**: accepts an intermediate key and a pair of values for the same key and combines them into zero or one value.



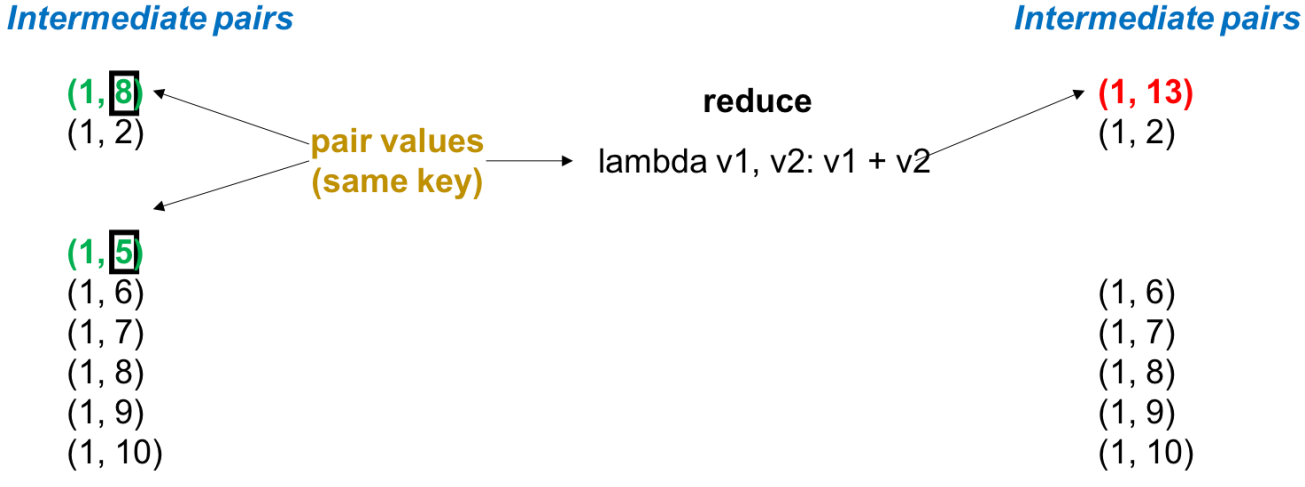
MapReduce: example

- **(Repeat) Reduce**: accepts an intermediate key and a pair of values for the same key and combines them into zero or one value.



MapReduce: example

- **(Repeat) Reduce**: accepts an intermediate key and a pair of values for the same key and combines them into zero or one value.

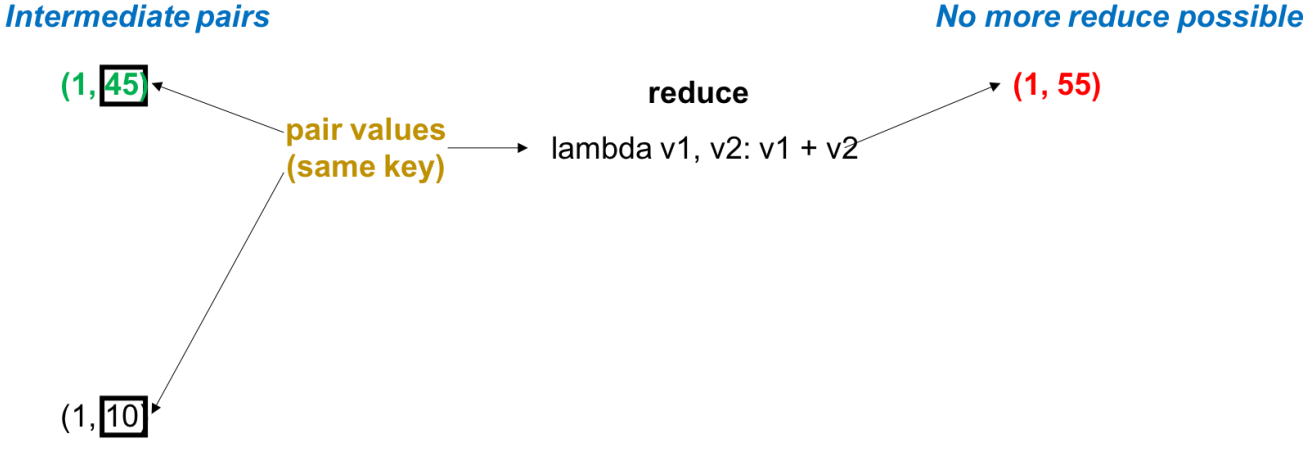


Identify

1. intermediate key
2. pair of values
3. reduce function
4. output value

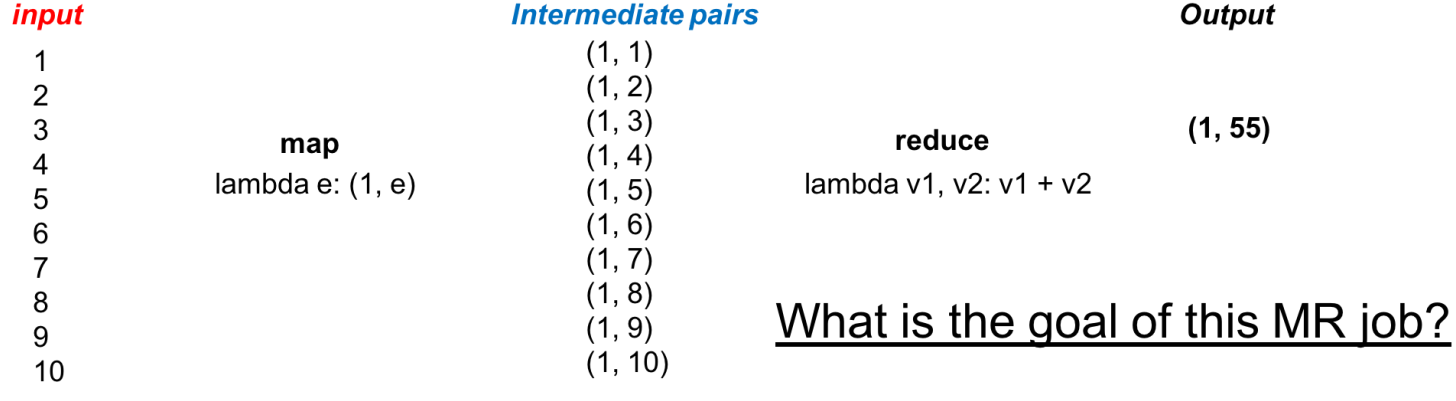
MapReduce: example

- **(Repeat) Reduce**: accepts an intermediate key and a pair of values for the same key and combines them into zero or one value.



MapReduce: example

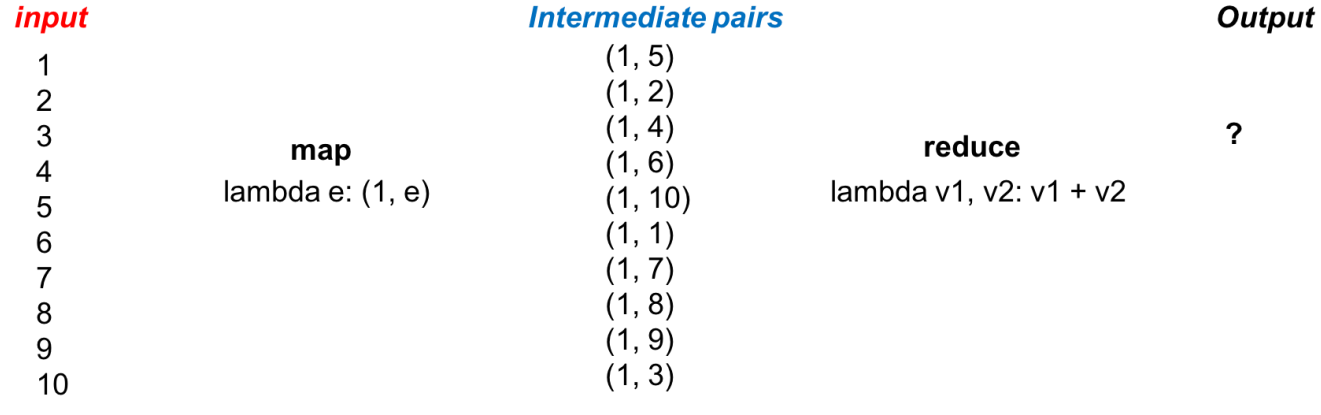
- **Map**: takes an input element and produces a set of intermediate key/value pairs
- **Reduce**: accepts an intermediate key and a **pair of values** for the same key and combines them into zero or one value.



What is the goal of this MR job?

MapReduce: example

- In MapReduce jobs the map and reduce order of application does not matter



MapReduce: example

- **Map**: takes an input element and produces a set of intermediate key/value pairs
- **Reduce**: accepts an intermediate key and a **pair of values** for the same key and combines them into zero or one value.

input

1
2
3
4
5
6
7
8
9
10

map

lambda e: (1, e)

reduce

lambda v1, v2: max(v1, v2)

What is the goal of this MR job?

MapReduce: example

- **Map**: takes an input element and produces a set of intermediate key/value pairs
- **Reduce**: accepts an intermediate key and a **pair of values** for the same key and combines them into zero or one value.

input

1
2
3
4
5
6
7
8
9
10

map

lambda e: (1, e)

reduce

lambda v1, v2: min(v1, v2)

What is the goal of this MR job?

MapReduce: example

- Inputs can be more complicated
- Some of the components can be trivial

Input (docID, text)

(1, “LOG: Time 2.4 sec”)
(2, “LOG: Time 8.4 sec”)
(3, “ERROR: Time 1.4 sec”)
(4, “LOG: Time 3.4 sec”)

map

```
def f(e):
   if “ERROR” in e[1]:
      return [e[0], None]
   return []
```

reduce

```
lambda v1, v2: None
```

What is the goal of this MR job?

MapReduce: example

- The output is **always** sorted by key

Input (docID, text)

1
10
6
3
4
6
1
4
2
3

map

```
lambda e: (e, None)
```

reduce

```
lambda v1, v2: None
```

What is the goal of this MR job?

MapReduce: example

- Map can generate several keys

Input (text)

"Document one"
"Another document"
"A document"

map

```
def f(e):
   results = []
   for word in e.split():
      results.append((word, 1))
   return results
```

reduce

```
lambda v1, v2: v1 + v2
```

What is the goal of this MR job?

MapReduce: worked out example 1

Given the following set of records of Month, State, and orders:

```
JAN, NY, 3
JAN, PA, 1
JAN, NJ, 2
JAN, CT, 4
FEB, PA, 1
FEB, NJ, 1
FEB, NY, 2
FEB, VT, 1
MAR, NJ, 2
MAR, NY, 1
MAR, VT, 2
MAR, PA, 3
```

- Compute the total number of orders per month using map reduce

MapReduce: worked out example 1 (2)

- Compute the total number of orders per month using map reduce

Data

```
JAN, NY, 3
JAN, PA, 1
JAN, NJ, 2
JAN, CT, 4
FEB, PA, 1
FEB, NJ, 1
FEB, NY, 2
FEB, VT, 1
MAR, NJ, 2
MAR, NY, 1
MAR, VT, 2
MAR, PA, 3
```

Map([JAN, NY, 3]) → [key_1, $value_1$]

Map([JAN, NJ, 2]) → [key_2, $value_2$]

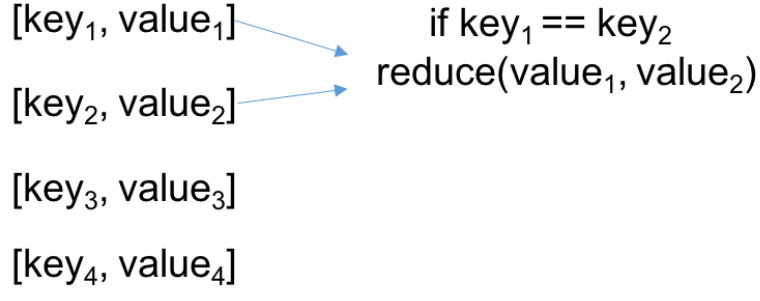
Map([FEB, PA, 1]) → [key_3, $value_3$]

MapReduce: worked out example 1 (3)

- Compute the total number of orders per month using map reduce

Data

```
JAN, NY, 3
JAN, PA, 1
JAN, NJ, 2
JAN, CT, 4
FEB, PA, 1
FEB, NJ, 1
FEB, NY, 2
FEB, VT, 1
MAR, NJ, 2
MAR, NY, 1
MAR, VT, 2
MAR, PA, 3
```

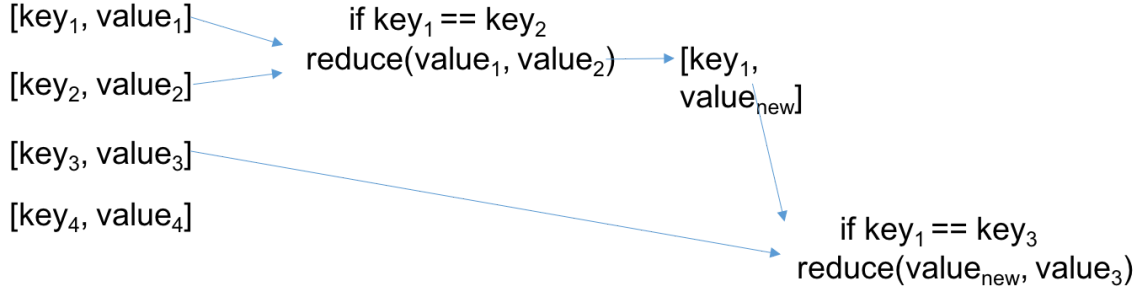


MapReduce: worked out example 1 (4)

- Compute the total number of orders per month using map reduce

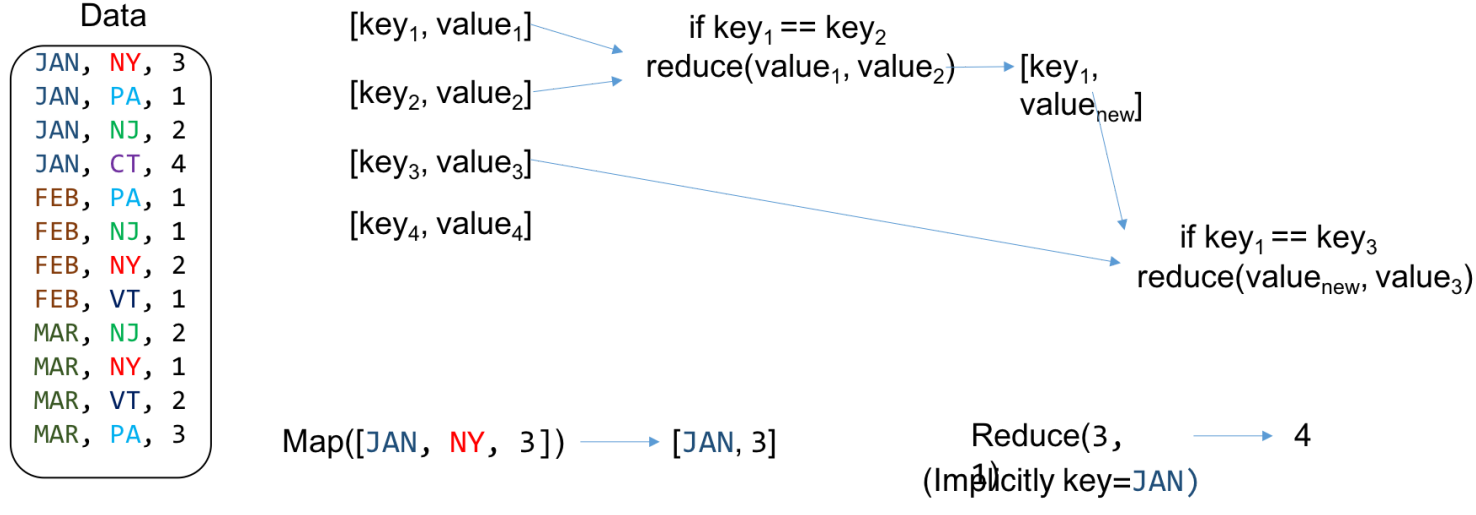
Data

```
JAN, NY, 3
JAN, PA, 1
JAN, NJ, 2
JAN, CT, 4
FEB, PA, 1
FEB, NJ, 1
FEB, NY, 2
FEB, VT, 1
MAR, NJ, 2
MAR, NY, 1
MAR, VT, 2
MAR, PA, 3
```



MapReduce: worked out example 1 (5)

- Compute the total number of orders per month using map reduce



MapReduce: worked out example 2

Given the following set of records of Month, State, and orders:

```
JAN, NY, 3
JAN, PA, 1
JAN, NJ, 2
JAN, CT, 4
FEB, PA, 1
FEB, NJ, 1
FEB, NY, 2
FEB, VT, 1
MAR, NJ, 2
MAR, NY, 1
MAR, VT, 2
MAR, PA, 3
```

- Compute the total number of orders per month using map reduce

```
def map(datapoint):
  ??
def reduce(value1, value2):
  ??
```

MapReduce: worked out example 2 (2)

Given the following set of records of Month, State, and orders:

```
JAN, NY, 3
JAN, PA, 1
JAN, NJ, 2
JAN, CT, 4
FEB, PA, 1
FEB, NJ, 1
FEB, NY, 2
FEB, VT, 1
MAR, NJ, 2
MAR, NY, 1
MAR, VT, 2
MAR, PA, 3
```

- Compute the total number of orders per month using map reduce

```
def map(datapoint):
  return [datapoint[0], datapoint[2]]
def reduce(value1, value2):
  return value1 + value2
```

MapReduce: theory

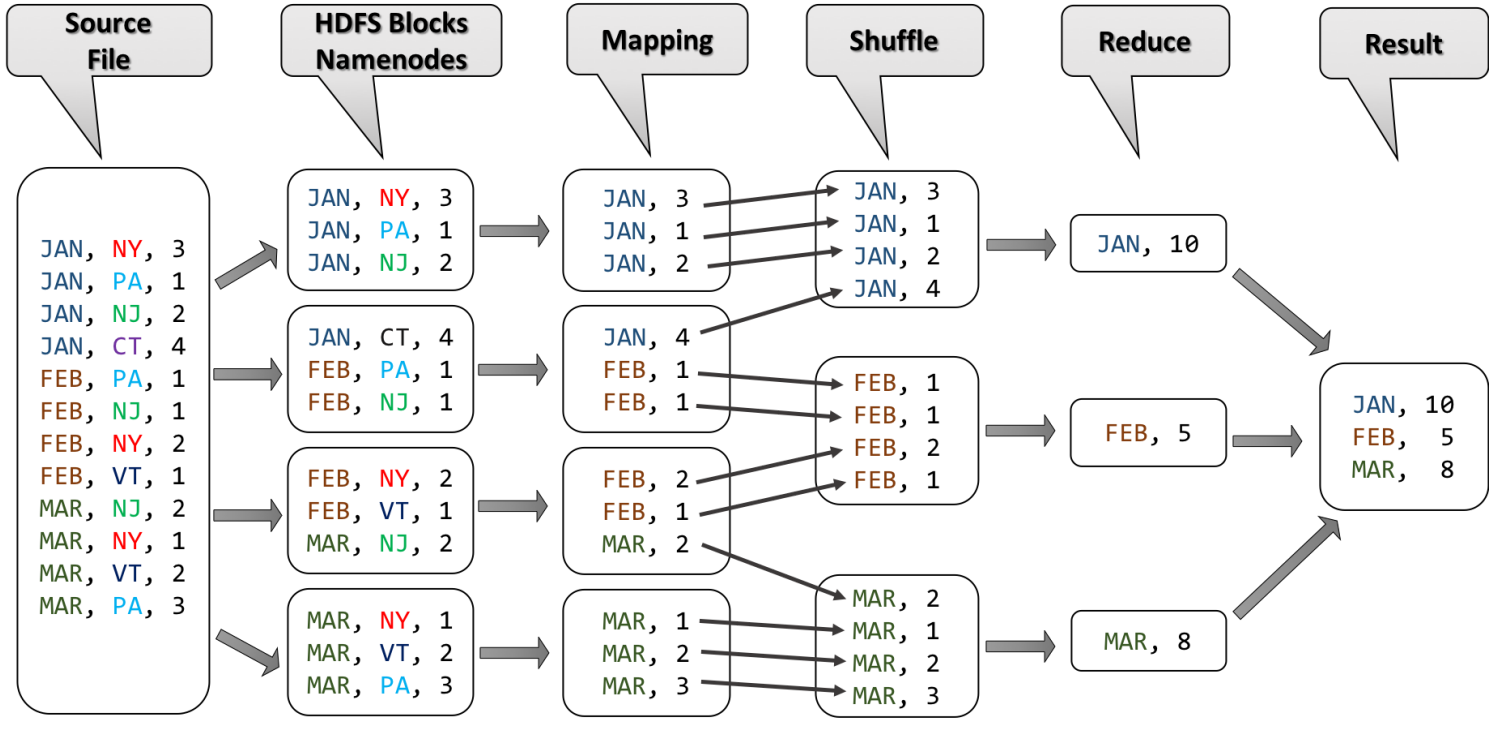
- It uses properties of functional programming
- Operations do not change data structures
- Original data always exists unmodified
- Data flows are implicitly defined by program design
- Order of operation does not matter
- This means:
  - Easy to parallelize
  - Fault-tolerant: re-execute failed operation
  - Status and monitoring
  - Easy abstraction

MapReduce (actual steps)

- The actual implementation of the process relies on four steps

- **Shuffle** and **Combine** are largely *transparent to the user*
  - Shuffle → transfer output from mapper to reducer nodes with similar keys
  - Combine → output of reducer nodes into single output.
- In Hadoop 2.0, MapReduce programs use HDFS and YARN.
- MapReduce is also implemented in Spark

MapReduce Example: Orders for each Month



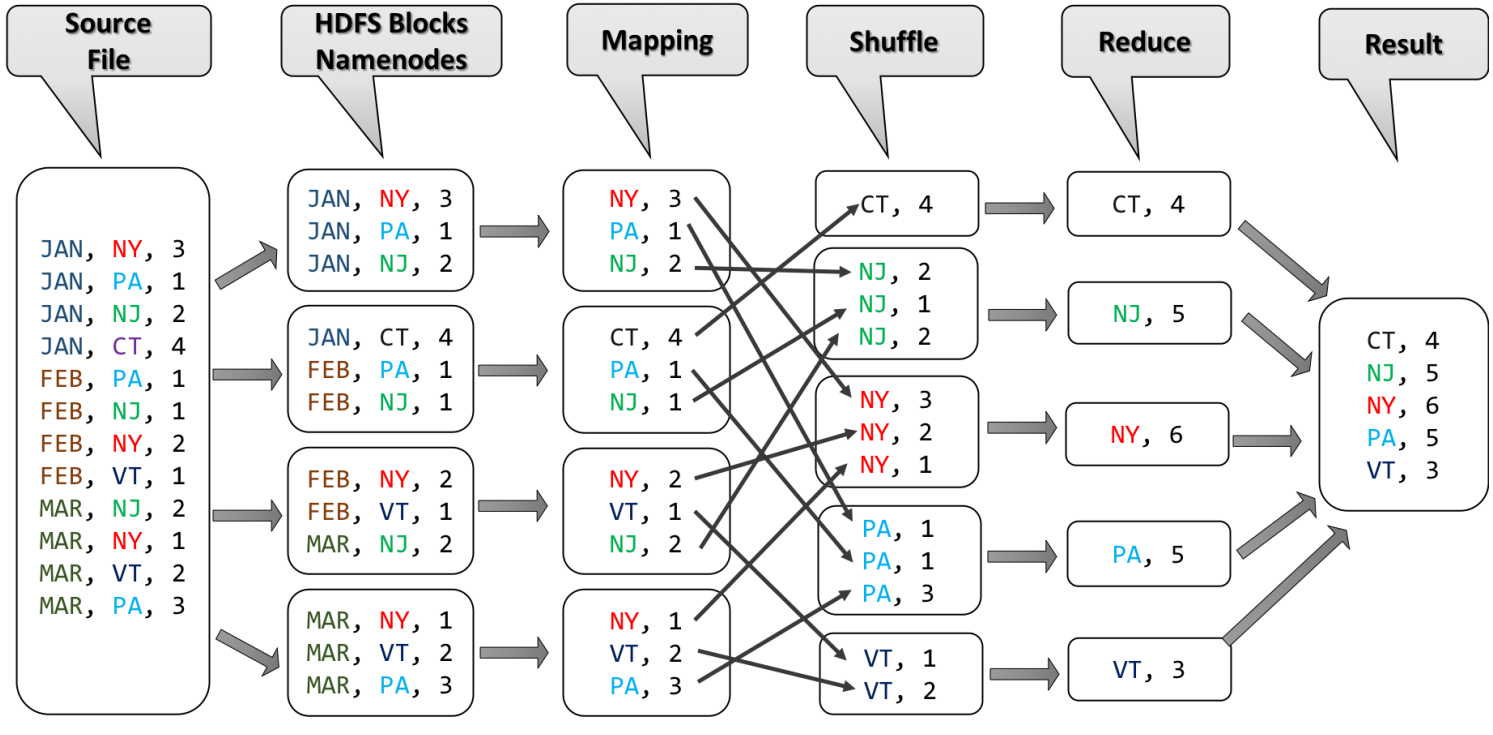
MapReduce: worked out example 2

Given the following set of records of Month, State, and orders:

```
JAN, NY, 3
JAN, PA, 1
JAN, NJ, 2
JAN, CT, 4
FEB, PA, 1
FEB, NJ, 1
FEB, NY, 2
FEB, VT, 1
MAR, NJ, 2
MAR, NY, 1
MAR, VT, 2
MAR, PA, 3
```

- Compute the total number of orders per month using map reduce

MapReduce: Total Orders by State



MapReduce: worked out example 3

Given the following set of records of Month, State, and orders

```
JAN, NY, 3
JAN, PA, 1
JAN, NJ, 2
JAN, CT, 4
FEB, PA, 1
FEB, NJ, 1
FEB, NY, 2
FEB, VT, 1
MAR, NJ, 2
MAR, NY, 1
MAR, VT, 2
MAR, PA, 3
```

- Compute the **average** number of orders per state using map reduce

The design of key, value pairs is important here

Databricks example

Create distributed input:

```
rdd = sc.paralellize(input_list)
```

Map:

```
maped_rdd = rdd.map(f)
```

Reduce:

```
maped_rdd.reduceByKey(r)
```

MapReduce exercises

- **Document search**: Each data point is the line of a document. Suppose we are searching for the lines that contain a certain word "ERROR". Final output is the list of lines that contain the word "ERROR".
- **Reserve web-link graph**: Each element data point is a key-value pair where key is a URL *A* and value is a list of URLs contained in the URL *A*. The final output should be a list of key-value pairs where key is URL *B* and value is a list of URLs that are linked to B.

Distributed system exercise

- Using simulation in Python, estimate the average response time and the average probability of failure for the case studied in class. Check that it is the same theoretical predictions.