

# **Introduction to Big Data Analytics and Engineering**

# Schedule

- What is “Big Data”
- The Hadoop ecosystem
- Working with Spark RDDs

# Download

**Slides:** <https://goo.gl/xb2MzB>

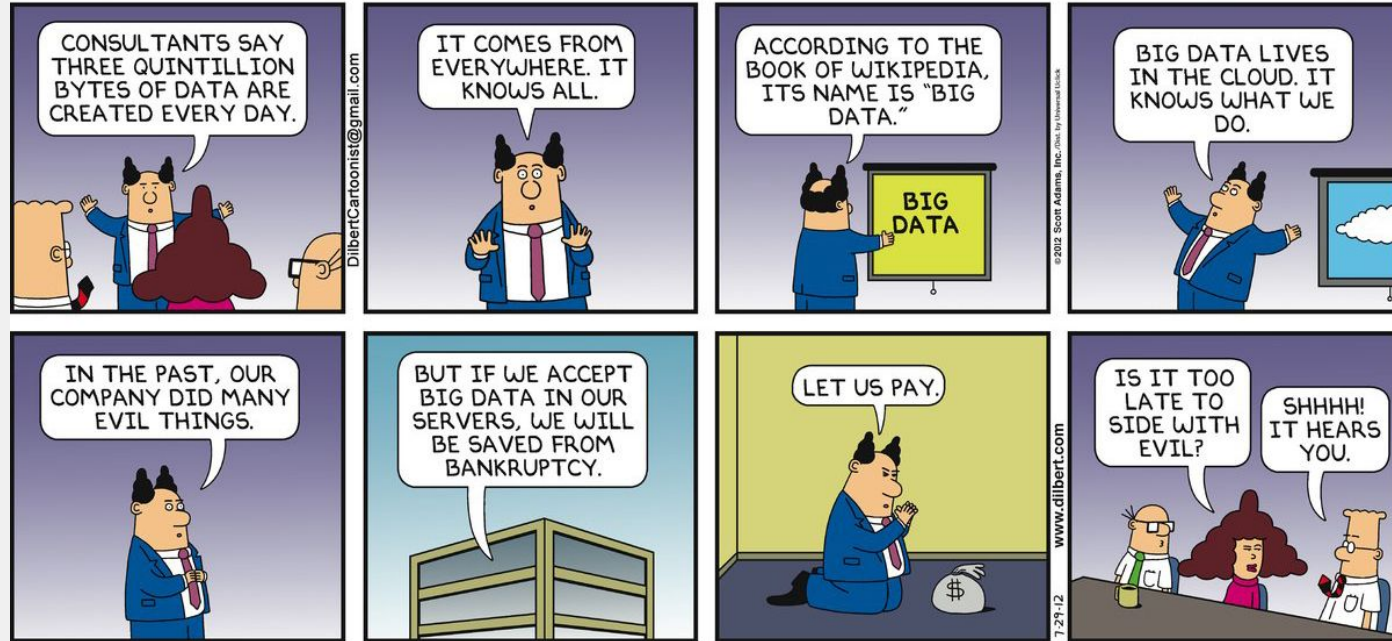
**Files:** <http://gitlab.cambridgespark.com/pub/bigdata-spark>

# Setup Instructions (see SETUP.MD)

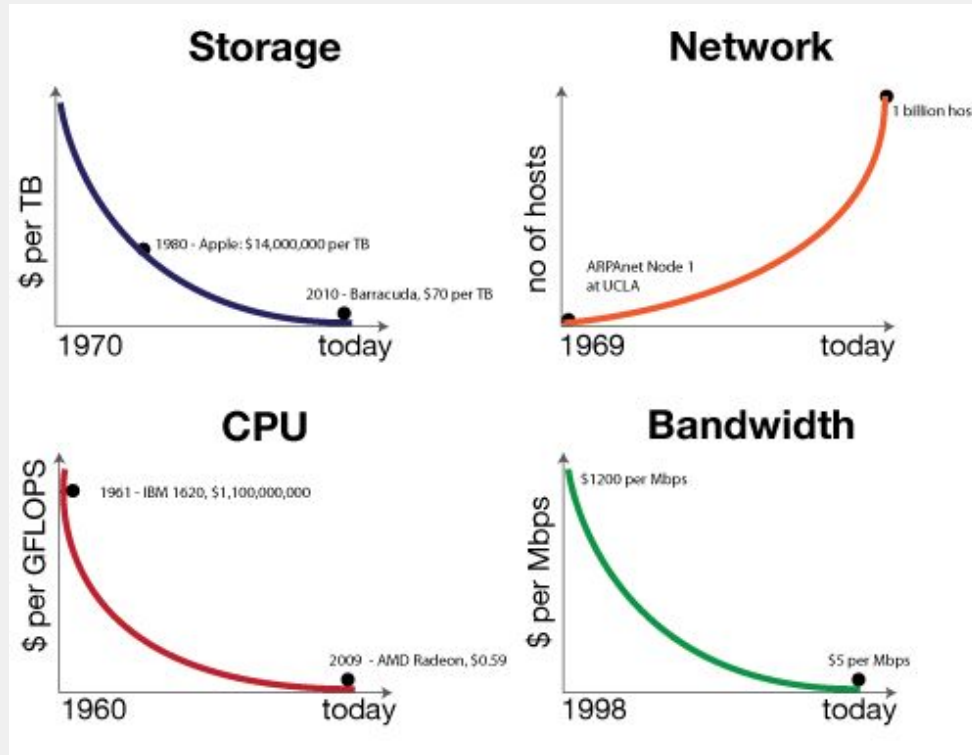
- Python 3.6 / Anaconda: <https://www.anaconda.com/download>
- JDK 8: [https://java.com/en/download/help/index\\_installing.xml](https://java.com/en/download/help/index_installing.xml)
- `pip install pyspark==2.2.0`
- `jupyter notebook Spark.ipynb`

# What is “Big Data”

# Big Data knows what we do



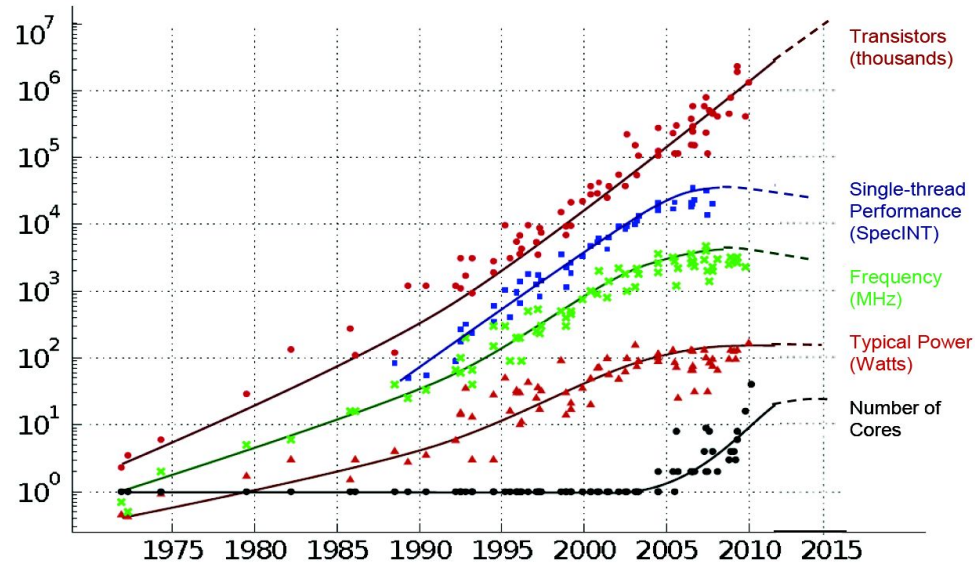
# Context



<http://radar.oreilly.com/2011/08/building-data-startups.html>

# Context

## 35 YEARS OF MICROPROCESSOR TREND DATA



Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten  
Dotted line extrapolations by C. Moore

<https://www.karlrupp.net/wp-content/uploads/2015/06/35years.png>



# Context

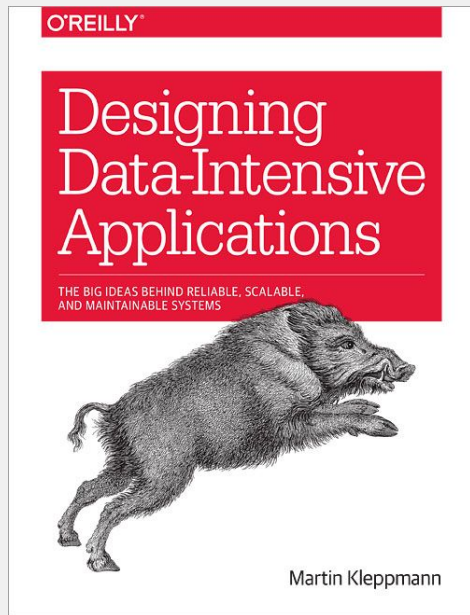
1. More Data
2. Cheaper hardware
3. Going parallel and distributed

# Defining Big Data

1. Volume
2. Velocity
3. Variety

# Defining Big Data

Many of the technologies described in this book fall within the realm of the *Big Data* buzzword. However, the term “Big Data” is so overused and underdefined that it is not useful in a serious engineering discussion. This book uses less ambiguous terms, such as single-node versus distributed systems, or online/interactive versus offline/batch processing systems.



NoSQL... Big Data... Scalability... CAP  
Theorem... Eventual Consistency...  
Sharding...

Nice buzzwords, but how does the stuff  
actually work?

# LA >>> What volume do you deal with?

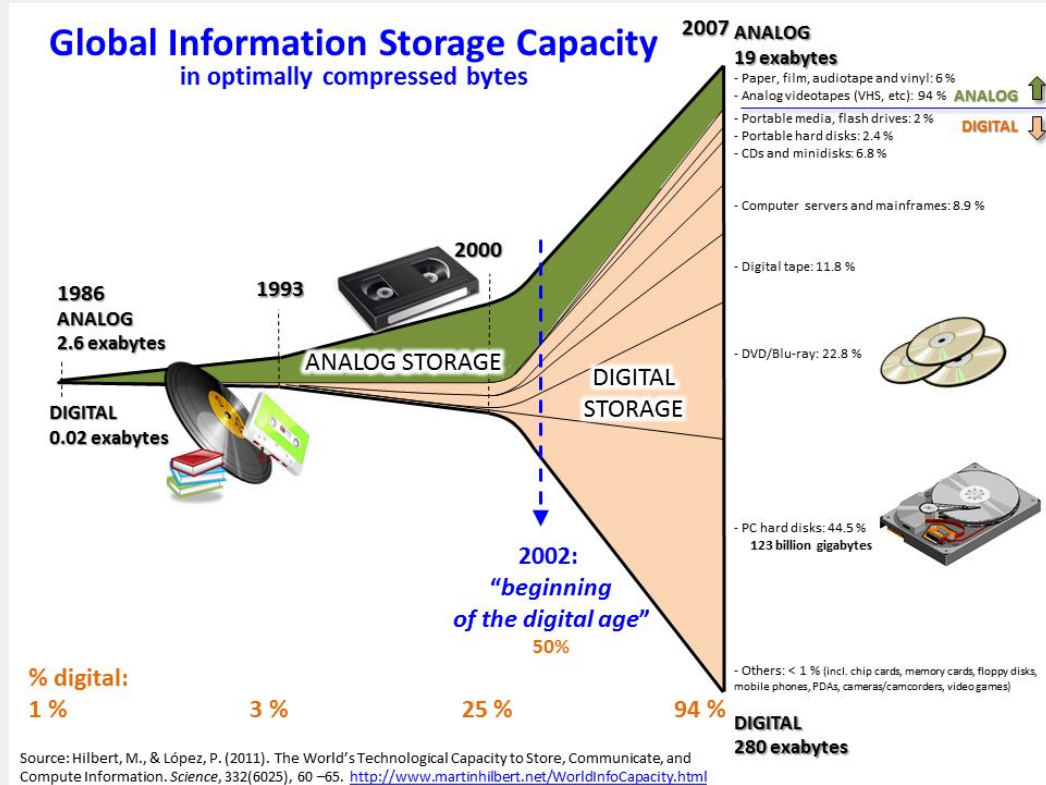
What sort of volume are you dealing with at your workspace?

- In terms of disk space?
- In terms of number of files?
- In terms of number of transactions?

# Volume example

- Facebook average daily contribution of one user takes 1MB of space
- Upwards of 1,000 million users.
- This translates into an average of **1,000TB** or **1PB** per day.

# Volume trend



# Volume challenges

- **Single machine not enough**
- How to process data across multiple machines?
- How to provide fault tolerance?

# Velocity

- CERN tracks ~600 million particle collisions per second resulting in 10 GB of data transferred from its servers every second.
- Twitter recently indicated that they handle **3,000 images every second**.
- WhatsApp is reported to process 60 billion messages a day.



# Variety

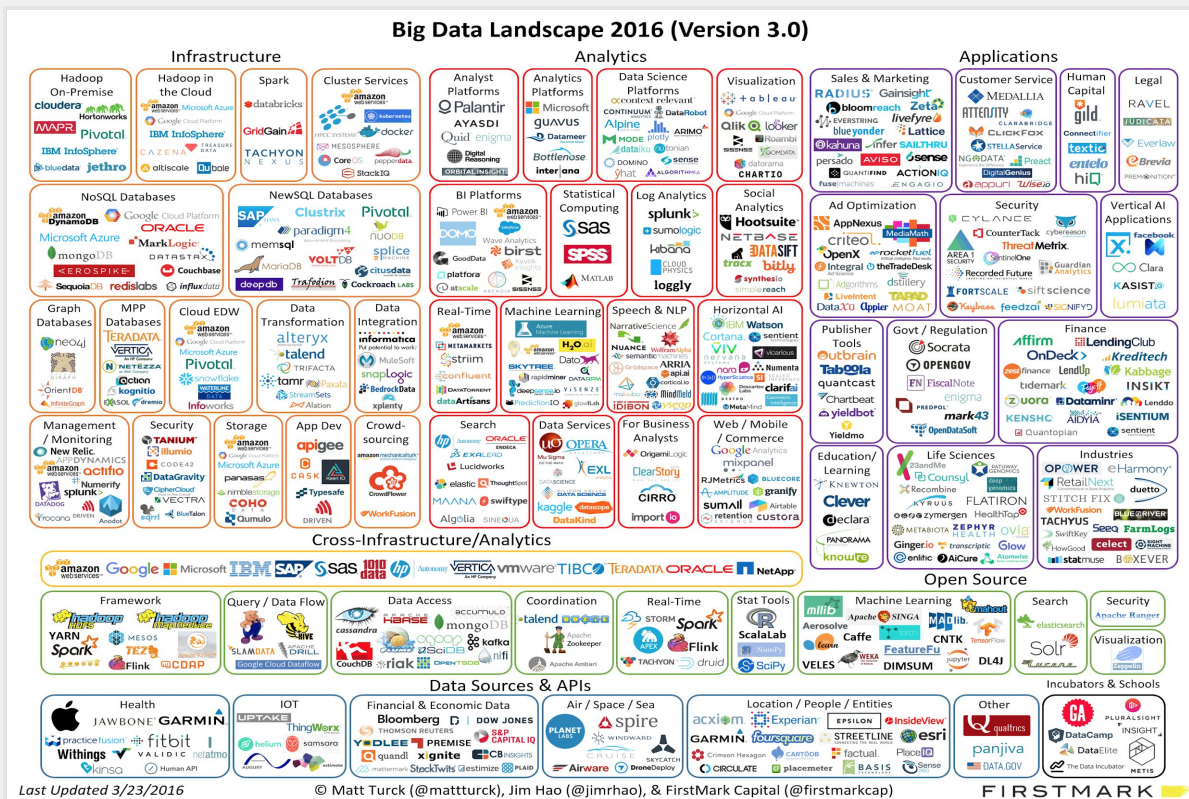


- Popular columns and rows model
- Relational databases
- Conceptually simple to related to

# Variety challenges

- Unified database schema impractical
- Graph structures
- Some data is *semi-structured* (JSON)
- Some data is *unstructured* (e.g. images and videos)

# Big Data Systems



See <https://hadoopecosystemtable.github.io>

# Batch processing: examples

- Calculate a monthly payroll summary.
- Train a machine learning model.
- Analyse a large set of statements to build regulatory compliance reports.

# Batch processing

- Data at rest
- High latency
- Concerned with volume and variety

# Stream processing: examples

- Classify an incoming email as spam or not.
- Detect whether a bank transaction is fraudulent.

# Stream processing

- Data in motion
- Low latency
- Concerned with velocity

## LA >>> Quiz

**Which of the following business problems are suited for batch or stream processing?**

1. Generate a balance sheet from all the invoices in a month.
2. Provide real-time alerts to stock market changes.
3. Build page-rank metrics for the web.
4. Detect anomalies in network traffic.



# Big Data: recap

- Big Data can be characterised by *volume*, *velocity* and *variety*.
- *Batch processing* is used to process all the data in one go.
- *Stream processing* is used to process data in real-time, as it is received.

# Hadoop ecosystem

# Dealing with big data



Too slow

# Dealing with big data



Too slow



Scaling-up

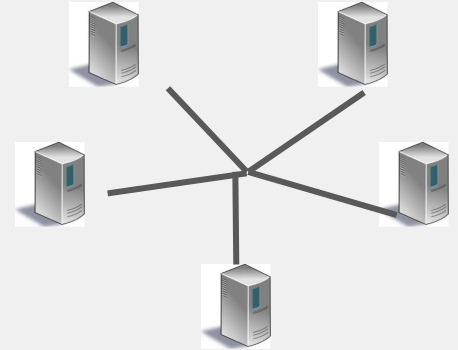
# Dealing with big data



Too slow



Scaling-up



Scaling-out

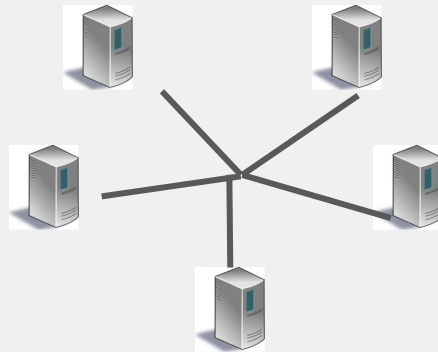
LA >>> Quiz: What are the pros and cons of each approach?



Too slow



Scaling-up



Scaling-out

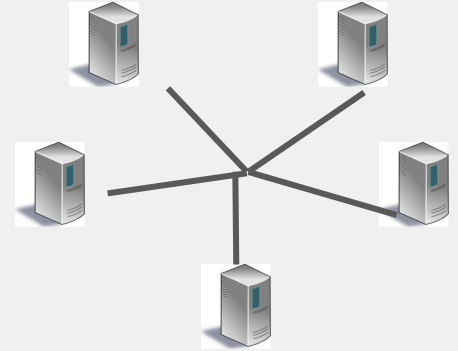
# Dealing with big data



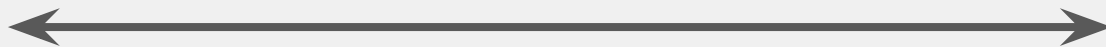
Too slow  
Usability



Scaling-up



Scaling-out



Power

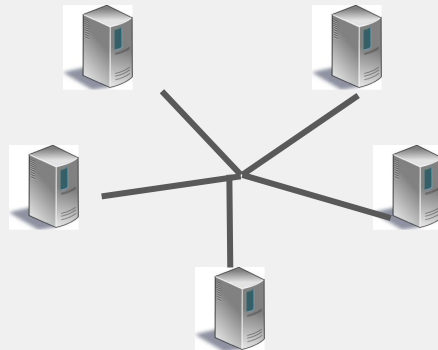
# Dealing with big data



Too slow



Scaling-up



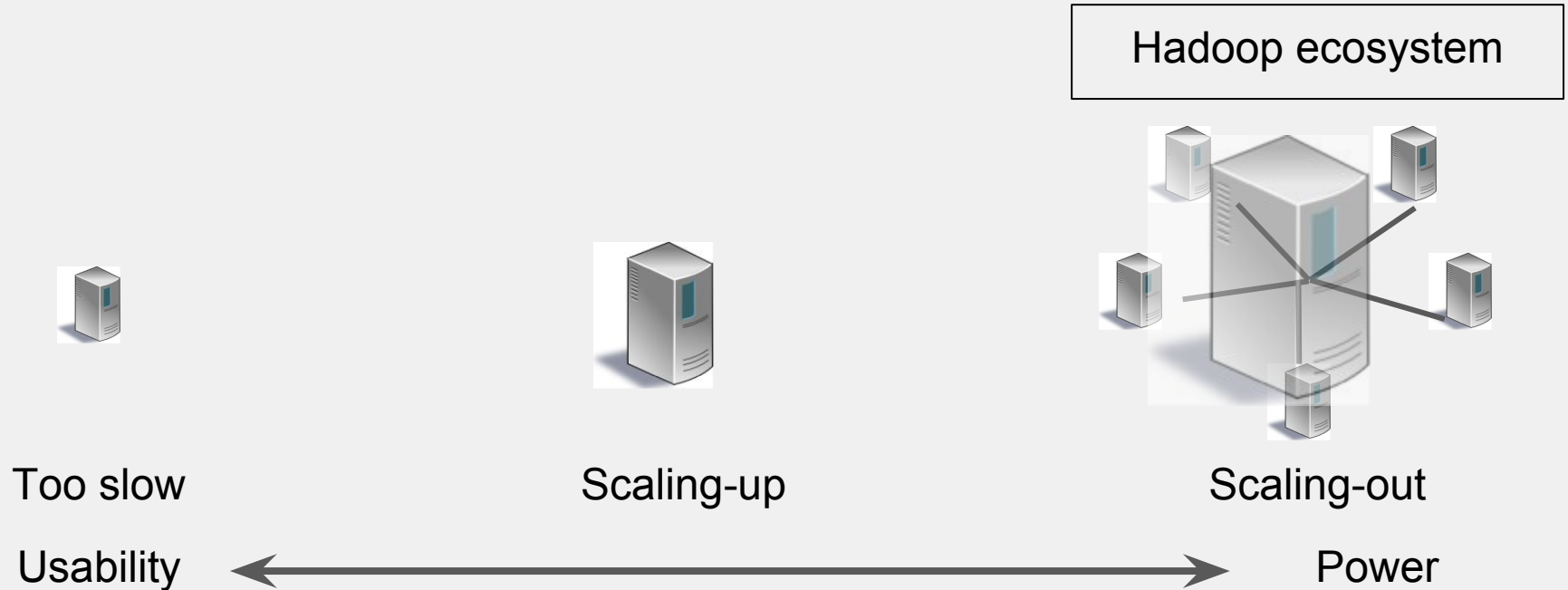
Scaling-out

- ✓ Same programming model
- ✗ High upfront cost
- ✗ Flexibility
- ✗ Single point of failure
- ✗ Technological limitations

- ✗ Difficult to use
- ✓ Cheap
- ✓ Flexible
- ✓ Fault tolerant



# Dealing with big data



# Hadoop ecosystem

Data processing:

Hadoop  
MapReduce

Apache Spark

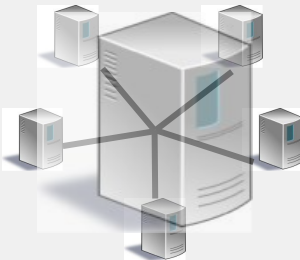
Resource management:

Hadoop YARN

File system:

Hadoop HDFS

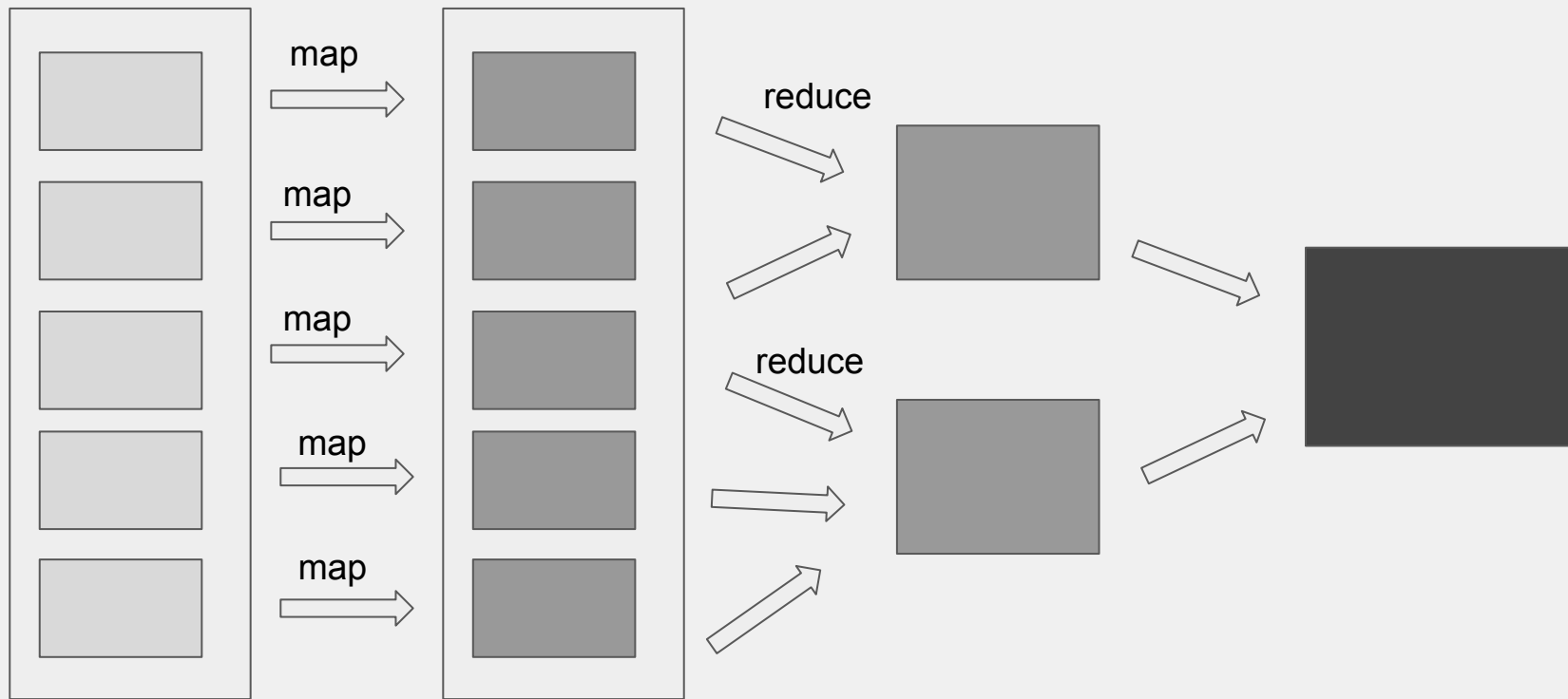
- Scalability
- Fault tolerance



# Hadoop MapReduce vs Apache Spark

	MapReduce	Spark
Initial release	2011	2014
# Operations	2	~20
Iterative computation	No	Yes
Interactive shell	No	Yes
Sorting 100TB	72 mins with 2100 machines	23 mins with 206 machines

# Map - Reduce Pattern

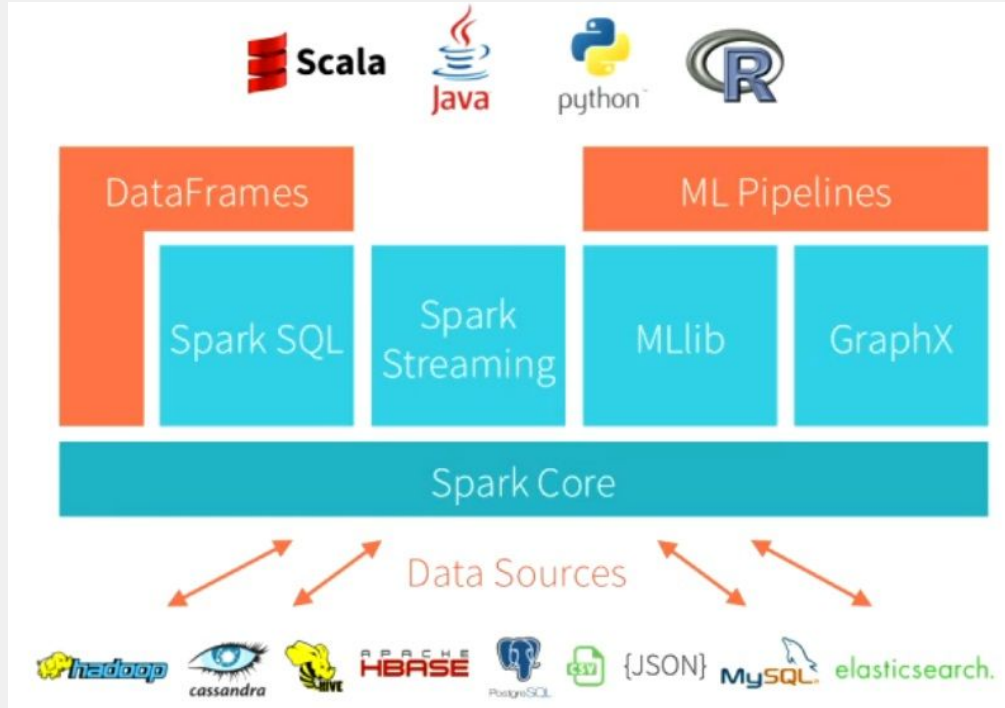


# Hadoop: recap

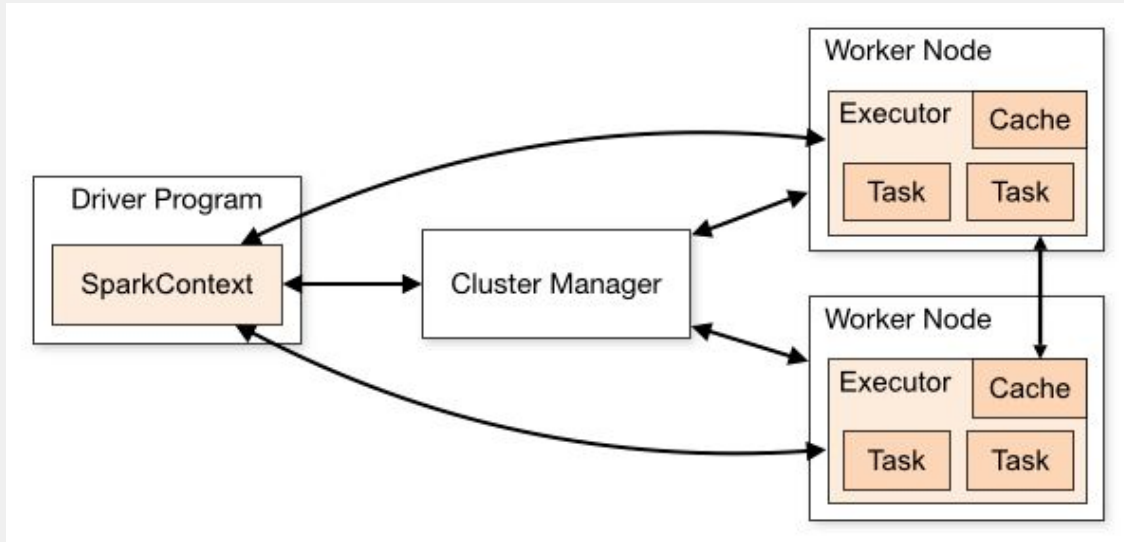
- Hadoop ecosystem makes distributed systems easy to use
- You will mainly interact with two components:
  - HDFS to store large files in a distributed way
  - Apache Spark to do data processing
- One single machine is always easier to work with than a distributed system

# Spark RDD

# Apache Spark

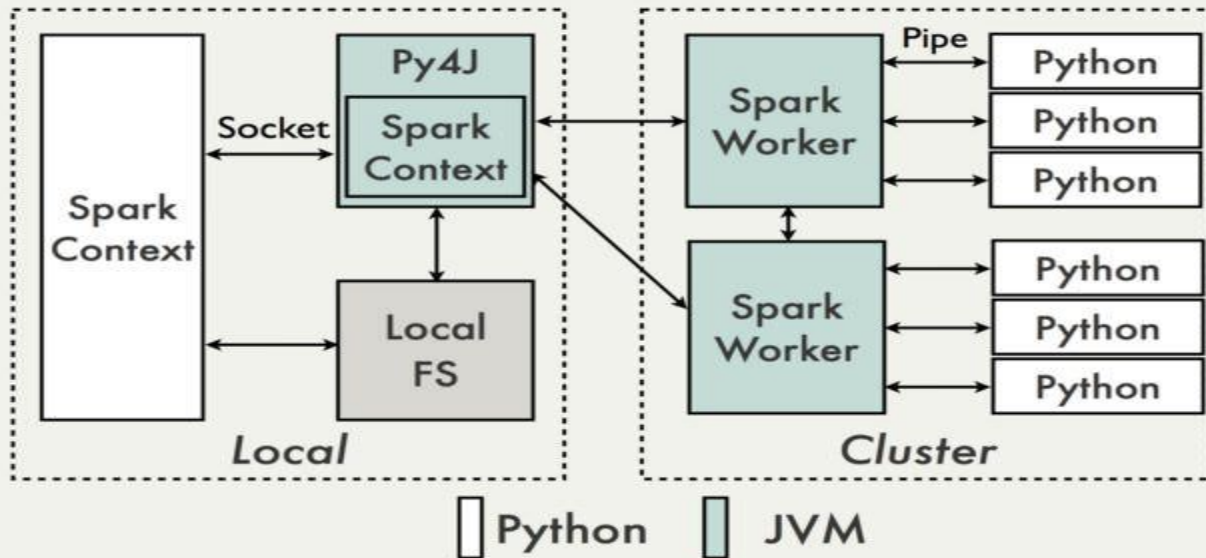


# Apache Spark Architecture





# How does Pyspark work



**Lots of overhead** required as Spark fundamentally is a JVM project. Data and Python objects need to be serialised and communicated.

# Apache Spark: Resilient Distributed Datasets (RDDs)

- Most basic abstraction
- Collection of elements (e.g. data points)
- Divided across the cluster

RDD

Amelia	Olivia	Charlie	George
Jack	Harry	Isla	James
Jessica	Lily	William	Sophie



# Creating an RDD

Python List

```
l = [1, 2, 3, 4]
```

`sc.parallelize(l)`

RDD

1, 2, 3, 4

File (local or on HDFS)

```
f = "data.txt"
```

`sc.textFile(f)`

RDD

line1, line2, ...

# Basic RDD manipulation

- `rdd.count()`: returns the number of elements in your RDD
- `rdd.take(n)`: returns the first n elements in your RDD
- `rdd.first()`: returns the first element in your RDD

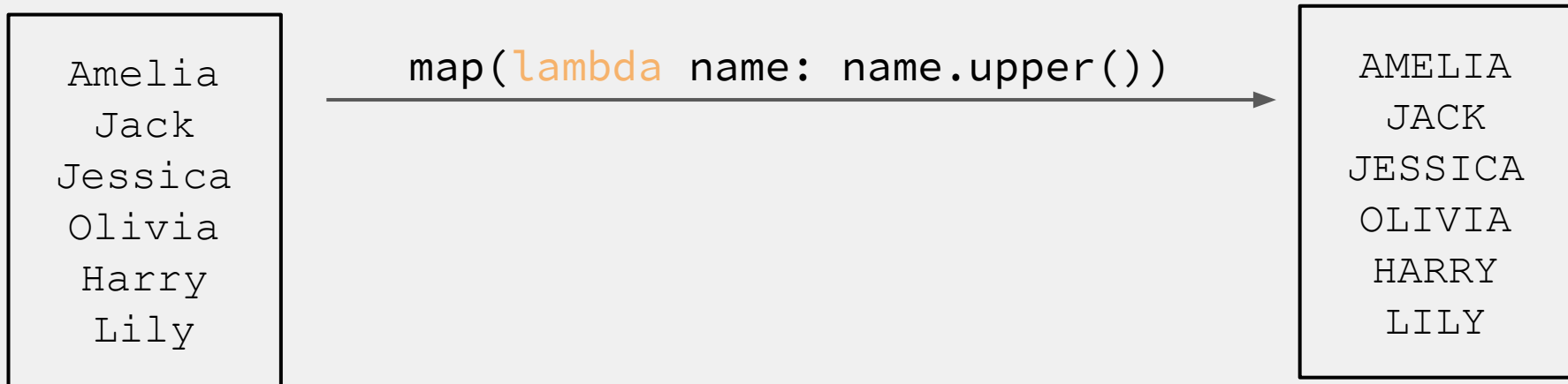
## Common transformations: `filter()`

Amelia  
Jack  
Jessica  
Olivia  
Harry  
Lily

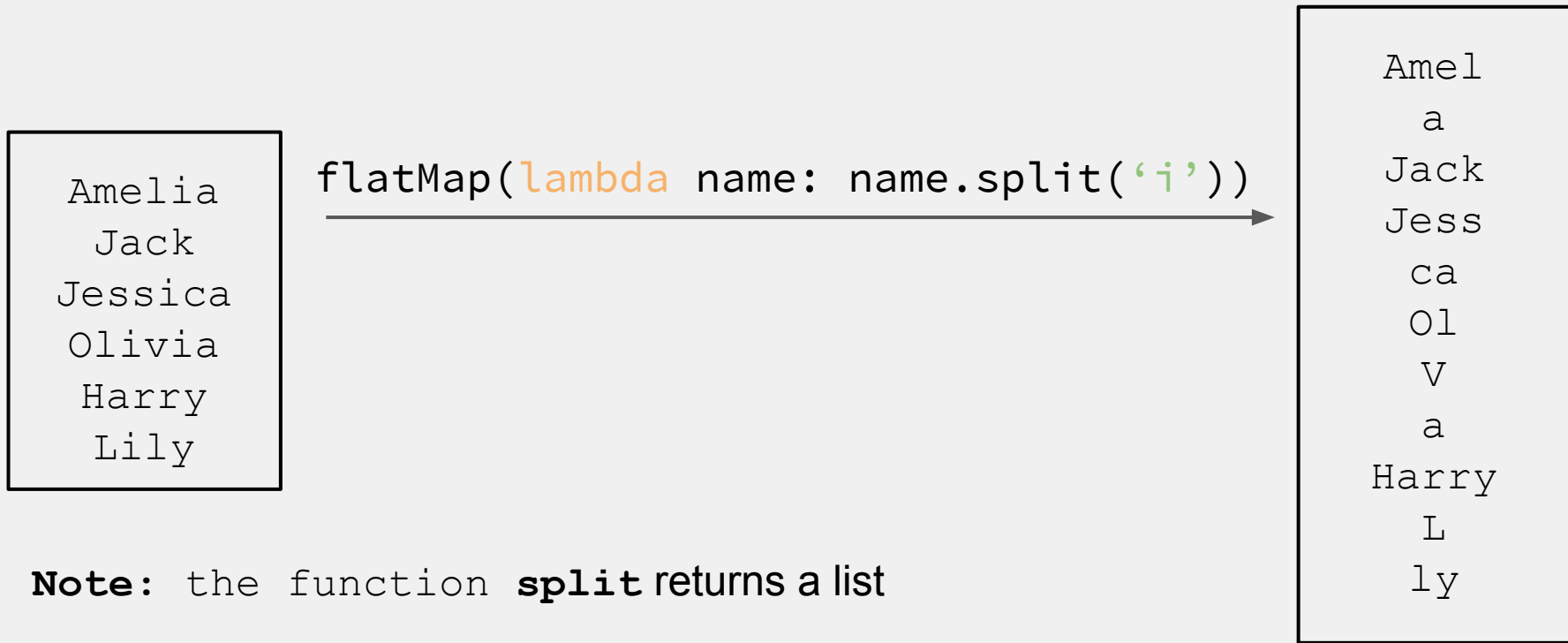
`filter(lambda name: name[0] == 'J')`

Jack  
Jessica

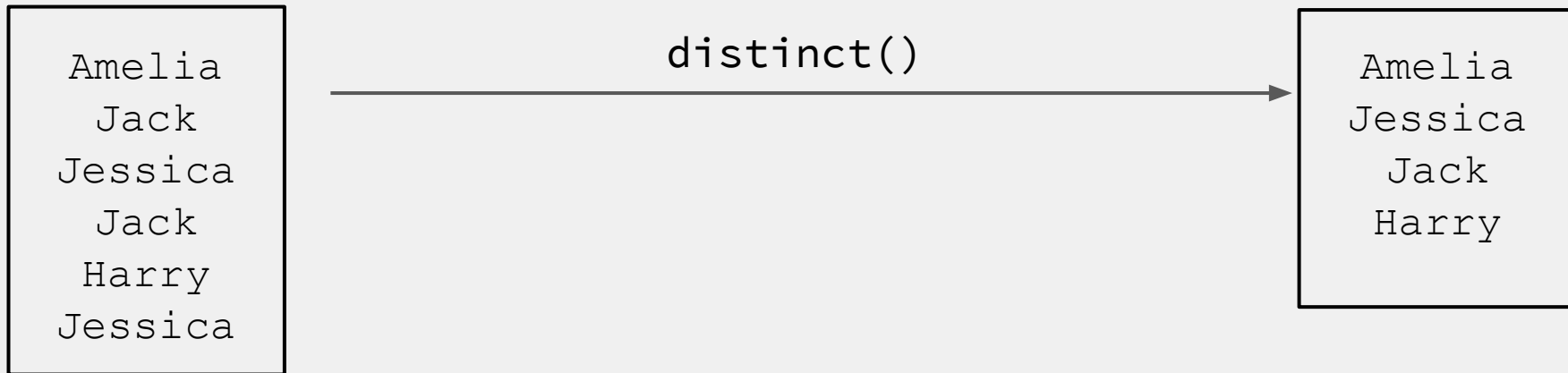
## Common transformations: map ( )



## Common transformations: flatMap()



## Common transformations: `distinct()`





# Transformations and Actions

- Transformations: operations that return a new RDD
- Actions: operations that return a value to your Python programs

## Quiz:

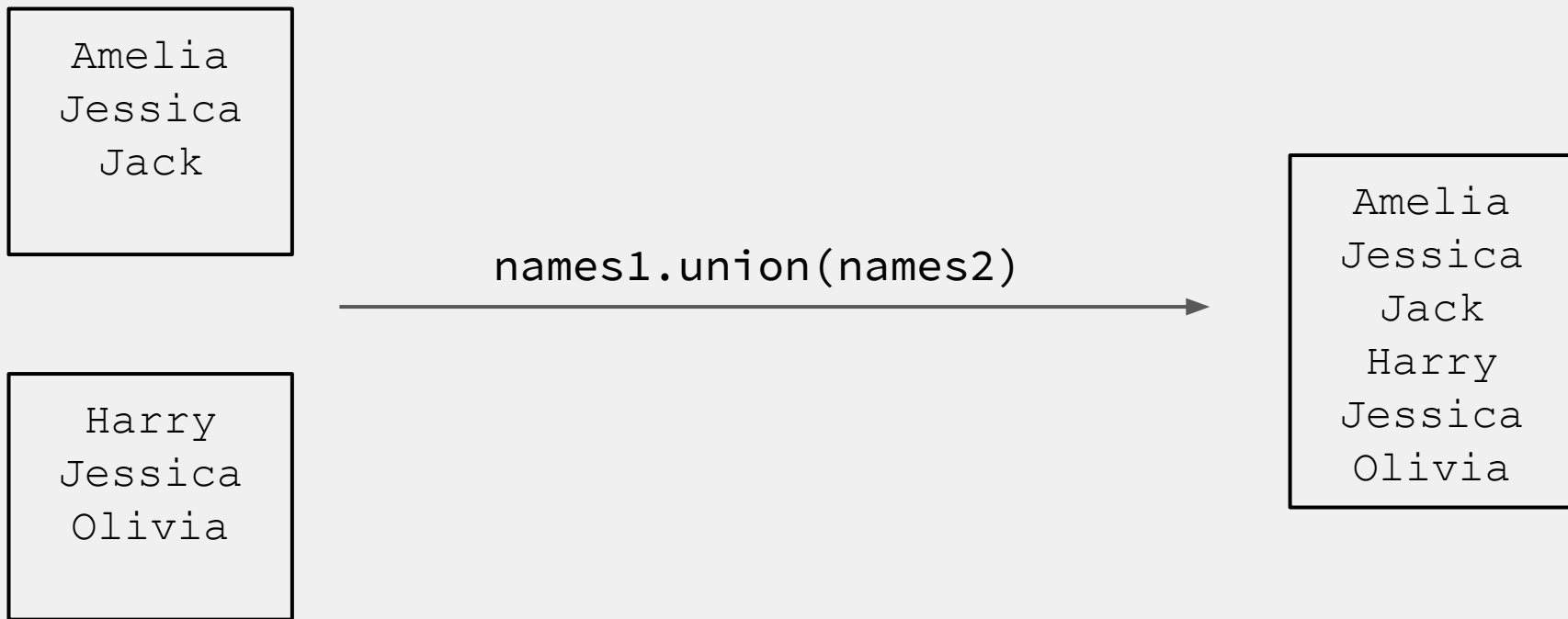
- Is `count()` an action or a transformation?
- Is `take()` an action or a transformation?
- Is `filter()` an action or a transformation?



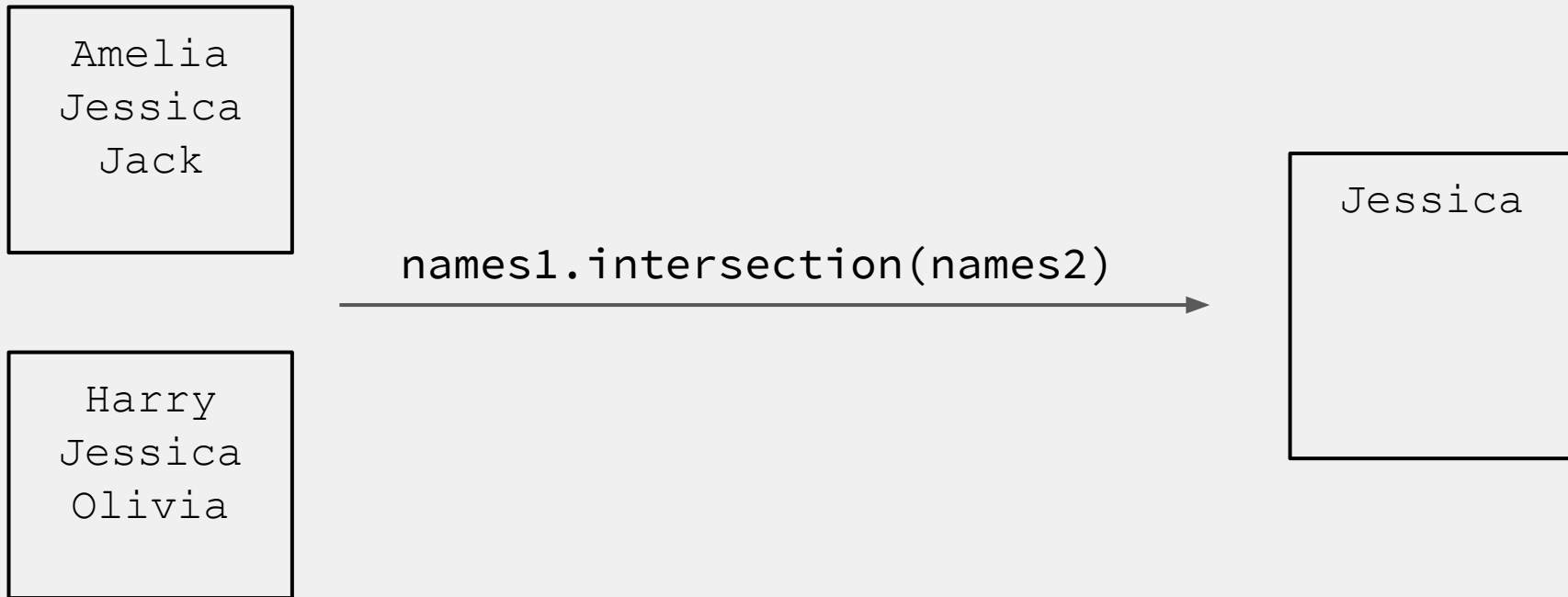
## Hands-on session

**>>> Create RDDs, `filter()`, `map()` and `distinct()`**

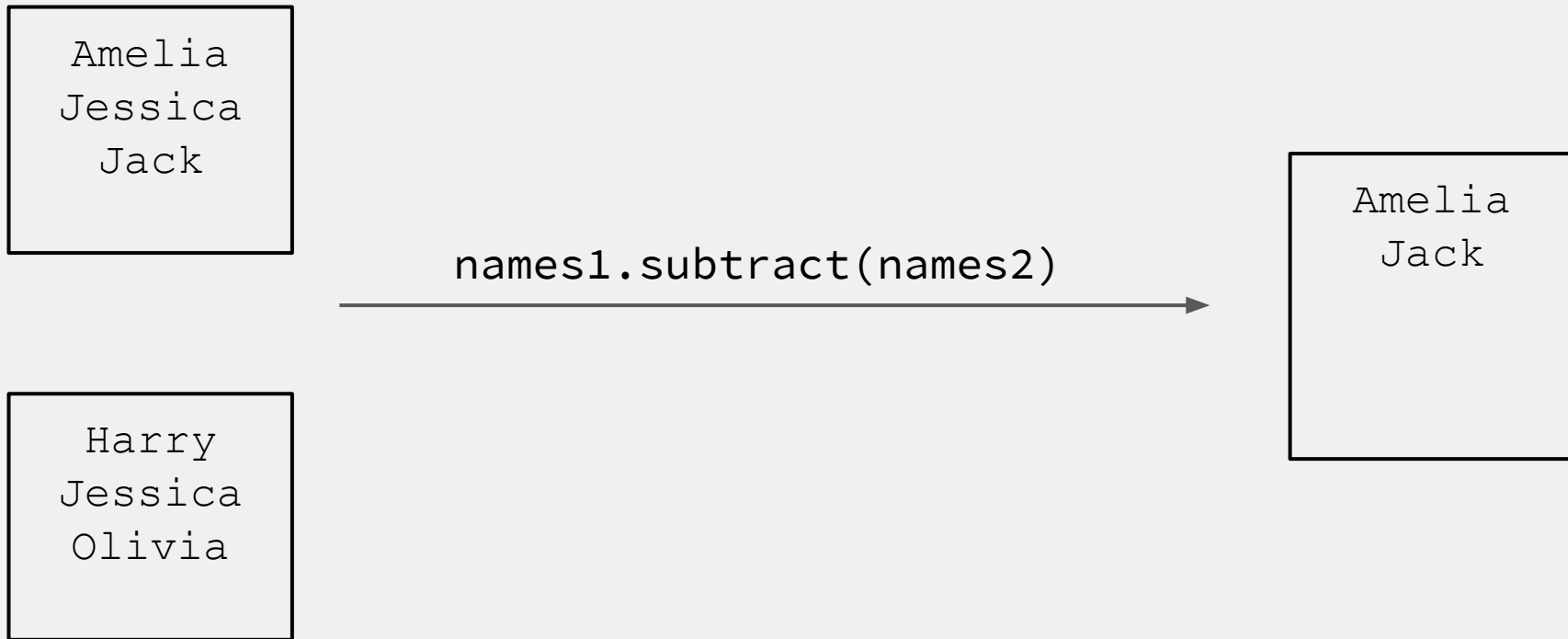
## Set-like transformations: union ( )



## Set-like transformations: intersection()

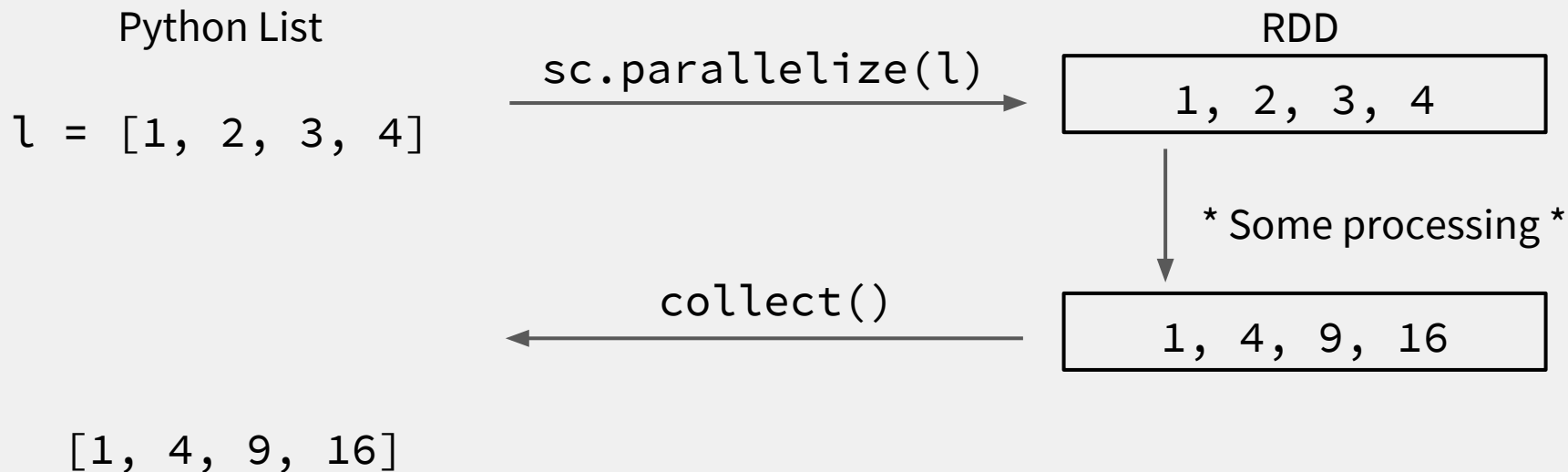


## Set-like transformations: subtract()



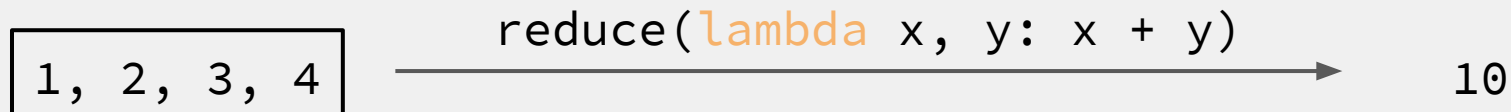
# Common actions

- `count()`
- `take()`
- `collect()`: opposite of `parallelize()`, turns an RDD into a Python list



## Common actions: reduce()

Takes as input a function that processes two elements and returns one





## Hands-on session

**>>> Set like transformations, reduce() & bonus**



# RDD Basics: recap

- RDDs are Sparks main abstraction to hold collections
- They have a rich API to process them
  - Transformations (`filter()`, `map()`, `union()`...) return processed RDDs
  - Actions (`count()`, `collect()`, `reduce()`...) return values to the Python program

# Lazy evaluation

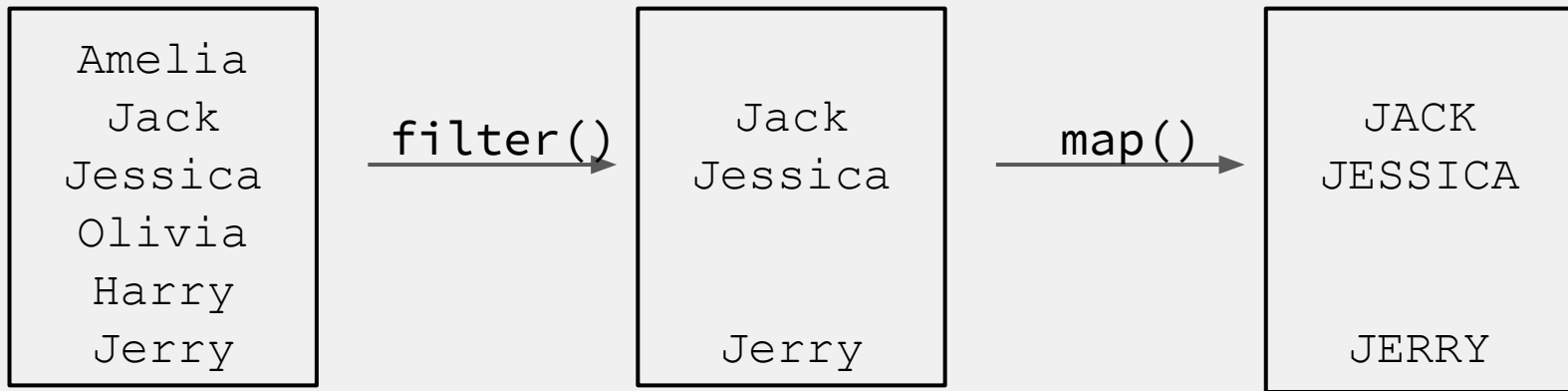
All transformations are *lazy*, they are only executed once an action gets called.

Similar in concept to a recipe:

- Transformations (e.g. `filter()`, `map()`) add instructions onto the recipe
- Actions (e.g. `count()`, `collect()`) process the entire recipe to return a result

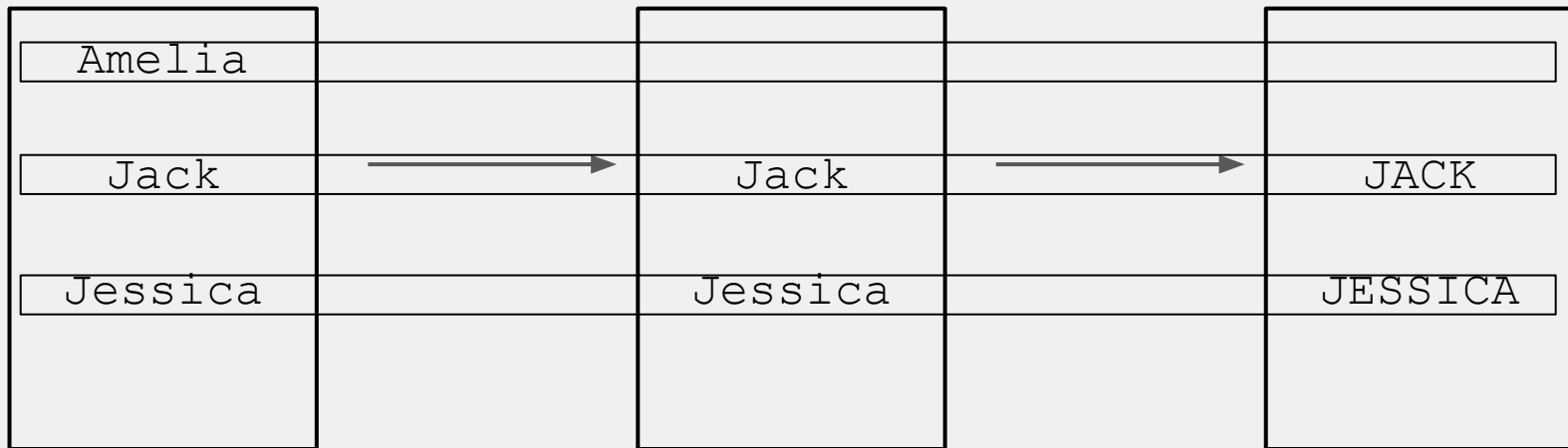
# Lazy evaluation

Allows Spark to batch together transformations:



# Lazy evaluation

Allows Spark to batch together transformations:



# Pair RDDs

Spark's way of storing key/value pairs

Normal RDD where elements are tuples

`(key, value)`

Note that in Python, the key and value of a tuple can be accessed as:

```
key = key_value[0]
```

```
value = key_value[1]
```

## Transformations on pair RDDs: groupByKey()

```
Alice: skiing  
Bob:   cats  
Bob:   coffee  
Greg:  pasta  
Alice: cars  
Alice: dogs
```

groupByKey() →

```
Alice:[skiing, dogs, cars]  
Bob:   [cats, coffee]  
Greg:  [pasta]
```

## Transformations on pair RDDs: reduceByKey()

Alice:	4
Bob:	2
Bob:	1
Greg:	4
Alice:	3
Alice:	3

`reduceByKey(lambda x, y: x+y)`

Alice:	10
Bob:	3
Greg:	4

## Some useful pair RDD transformations

- **keys ()**: an RDD of the keys
- **values ()**: an RDD of the values
- **mapValues (func)** and **flatMapValues (func)**: Apply func onto the values





## Hands-on session

**>>> Word count in Spark**

## **AWS EMR + pyspark Demo**

**<https://www.youtube.com/watch?v=UbkucQ2EOXg>**

## RDD Further: recap

- Transformations are lazy
- Pair RDDs store keys and values
- They have a few specific transformations
  - `groupByKey()` , `reduceByKey()` . . .
- Joins are useful to merge pair RDDs together

# **Spark dataset challenge: Analysing the HackerNews dataset**