whoami

Consulting

- Human-in-the-loop machine learning + MLOps

YouTuber — https://www.youtube.com/jonathandinu

- Data science and deep learning

Getting the Materials

https://github.com/jonathandinu/spark-livetraining

A Brief History

> *We have another 10 to 20 years before we reach a fundamental limit. By then they'll be able to make bigger chips and have transistor budgets in the billions.*

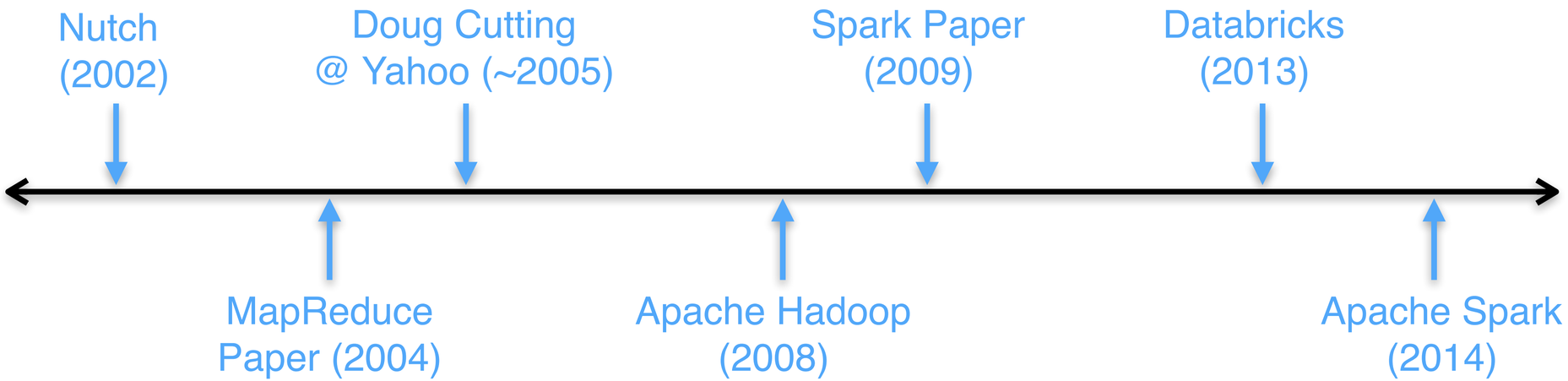
- Gordon Moore (2006)

The Recent Past

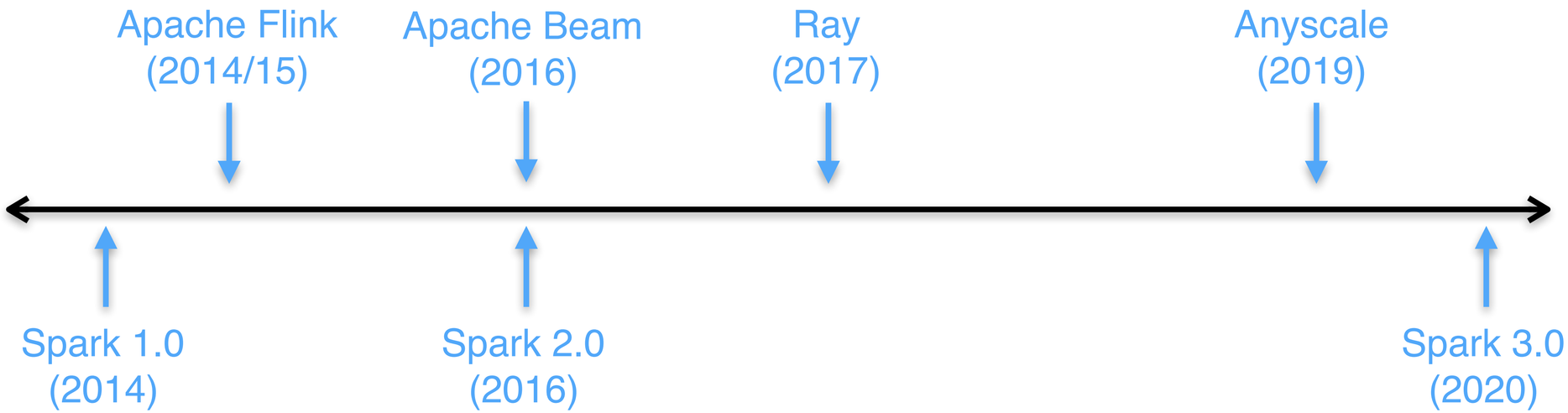
- Found by AMPLab, UC Berkeley (circa 2009)
  - Created by Matei Zaharia (PhD Thesis)
  - Maintained by Apache Software Foundation
  - Commercial Support by Databricks

The Recent Past

- Found by AMPLab, UC Berkeley (circa 2009)
  - Created by Matei Zaharia (PhD Thesis)
  - Maintained by Apache Software Foundation
  - Commercial Support by Databricks



The Recent Future: ML Systems



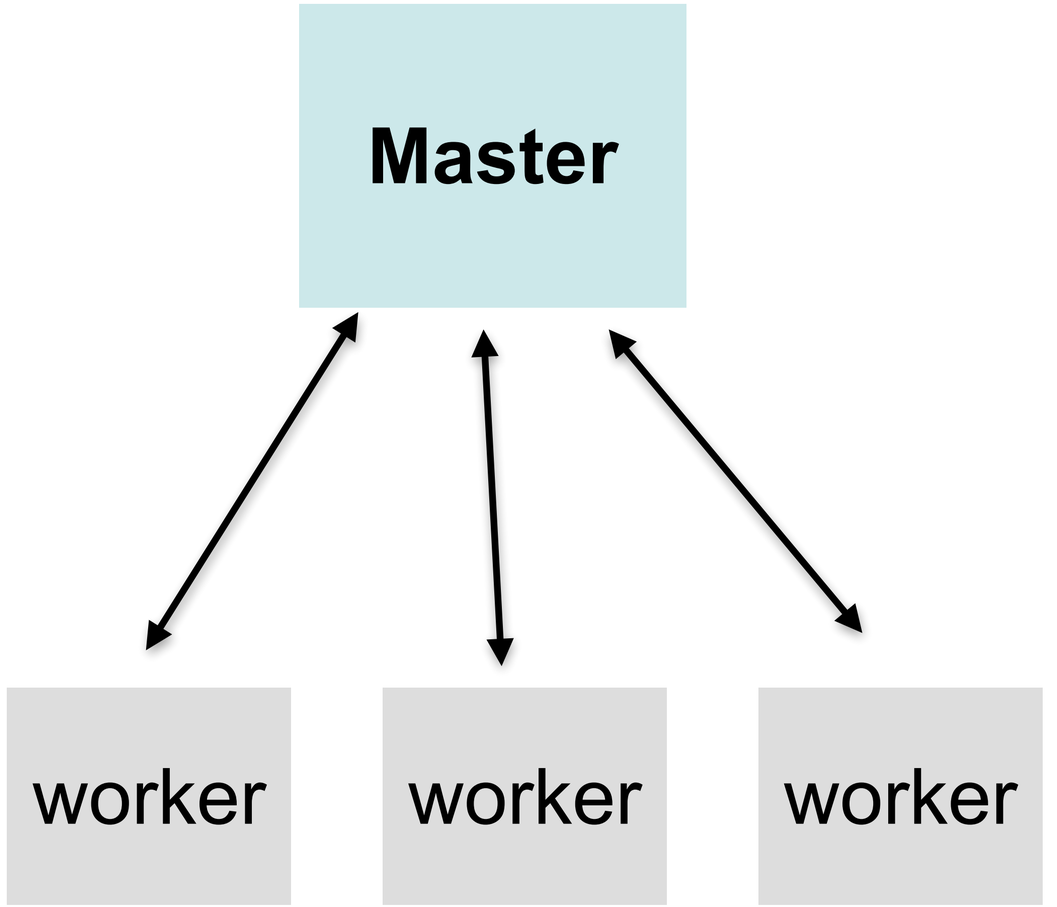
> *Distributed programming is the art of solving the same problem that you can solve on a single computer using multiple computers.*

- Distributed Systems for Fun and Profit, Mikito Takada

Local

Single Machine

Distributed



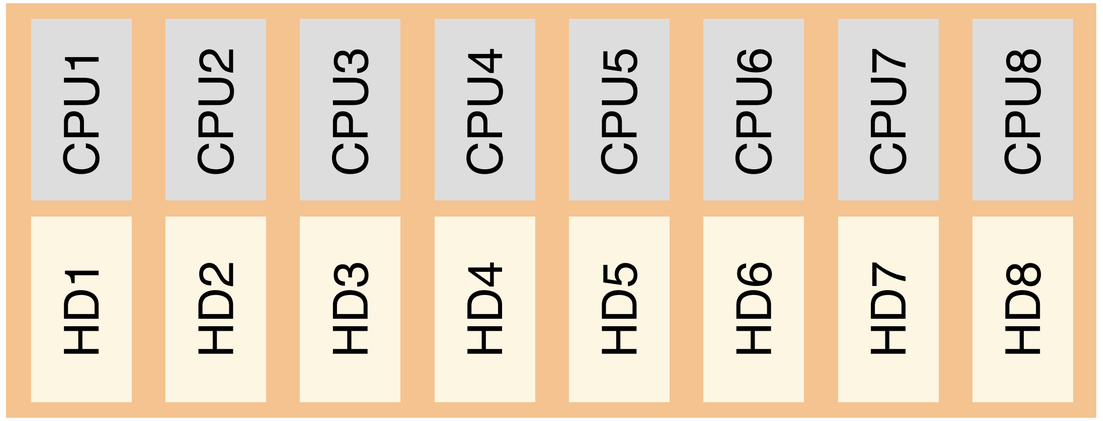
Why Distribute?

- Problem no longer fits on a single machine (storage)
- Problem need to be solved faster (computation)
- Often a combination of the two.

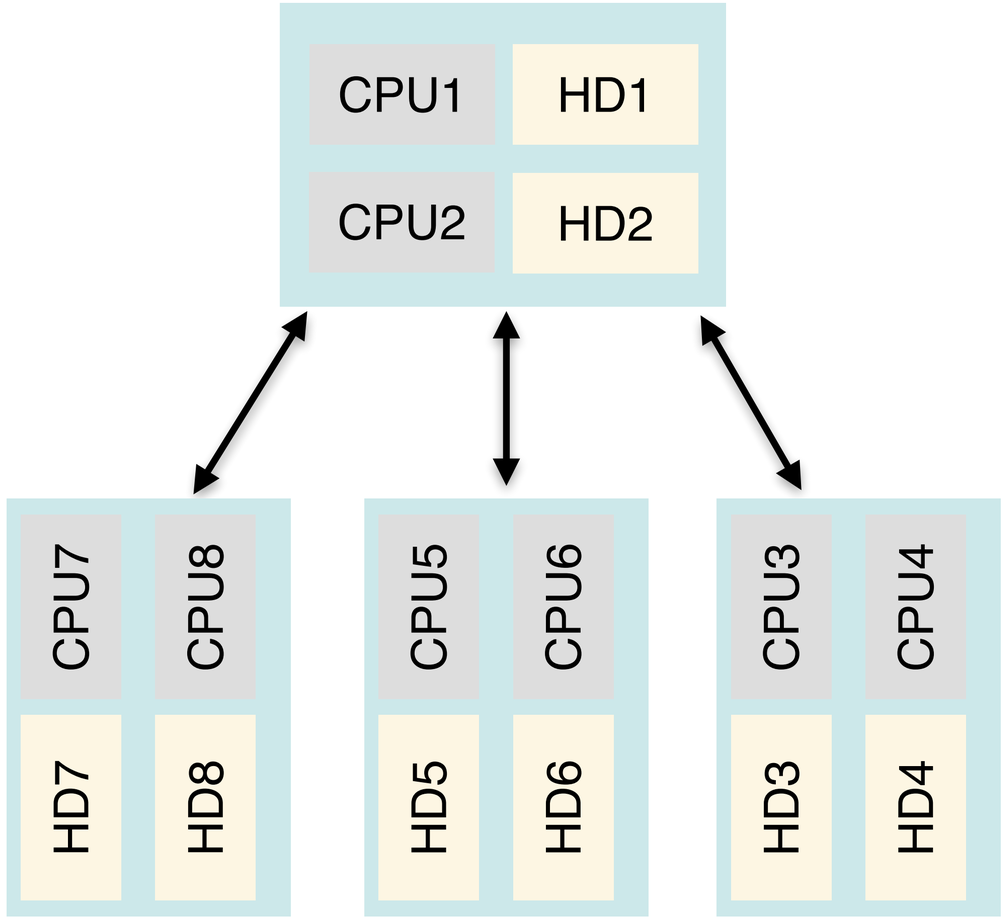
Availability is also often sought after and achieved by distributing a system but in the context of Spark we will concern ourselves with storage and computation since its programing model can be thought to be always available.

How to Distribute

Local



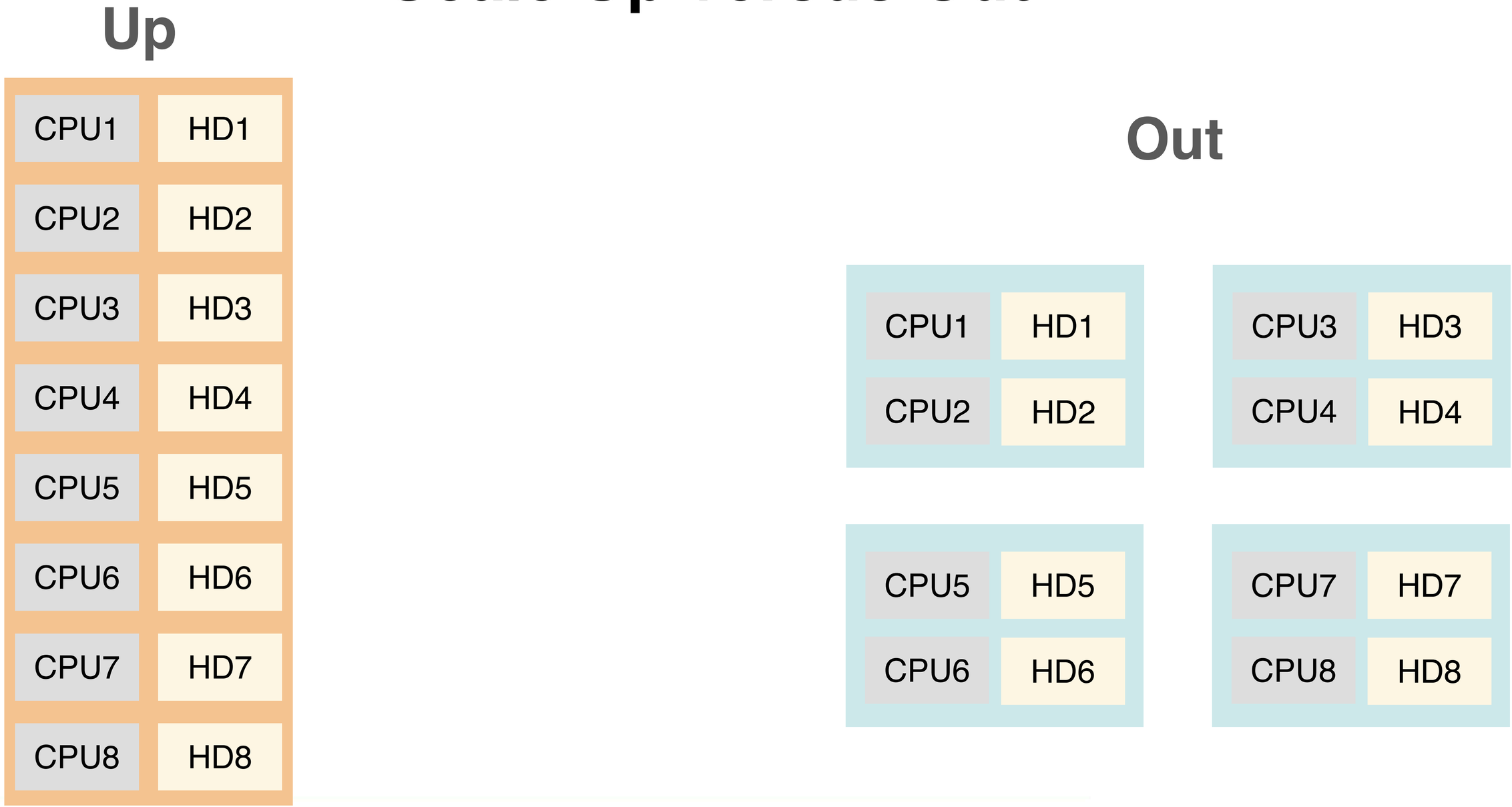
Distributed



Storage: Hard drives

Computation: CPUs

Scale Up versus Out



So why do I need to know this?

- **Logic:** if you understand the system you are programming, you can reason about your code much more effectively.
- **Debugging:** if you understand the system you are programming, you can fix your code much faster.
- **Performance:** if you understand the system you are programming, you can write smarter (and more optimal) code.

Problem Solved?

- **Coordination:** what happens when and to whom? And how do you synchronize these events
- **Communication:** how do the different nodes in your system talk to one another? And how do you talk to your nodes?
- **Fault Tolerance:** is your system robust to network and machine failures (because they will happen)?

Review

- Different distributed architectures have different tradeoffs
- Scaling up is logically easier for programmers, and easier to reason about.
- Scaling out is theoretically unbounded, but coordinating and reasoning in a distributed system is hard
- You have to make compromises to ensure fault tolerance in a distributed system, though there are ways to mitigate these.

Scaling Data

Scaling Compute

TensorFlow

RAPIDS

DASK

PyTorch

> *The Data Engineer lives in the liminal space between distributed systems (CS theory), engineering (operations/infrastructure), and statistics. While it is not crucial for them to understand each of these domains wholly, being able to synthesize concepts from each is essential for success…*

- Jonathan Dinu

> *And Spark lives exactly at this nexus! Making it one of the most powerful frameworks for data scientists and engineers alike!*

- Jonathan Dinu

What is Spark

Apache Spark™ is a fast and general engine for large-scale data processing.

What is Spark

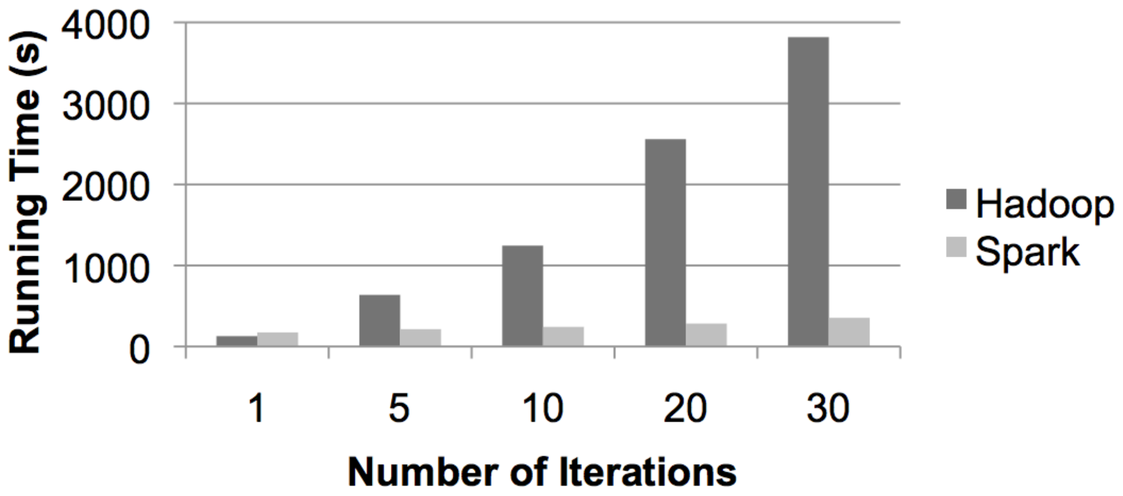
- Framework for distributed processing
- In-memory, fault tolerant data structures
- Flexible APIs in Scala, Java, Python, R, and SQL!
- Open Source

Why Spark?

- Handles Petabytes of data (and more!)
- Significantly faster than Hadoop Map-Reduce (for most jobs)
- Simple and intuitive APIs

- General Framework
  - Runs Anywhere
  - Handles (most) any I/O
  - Interoperable libraries for specific use-cases

Performance



Logistic Regression

- Very fast at iterative algorithms
- DAG scheduler supports cyclic flows (and graph computation)
- Intermediate results kept in memory when possible
- Bring computation to the data (data locality)

Rich API


```
map()          reduce()
filter()       sortBy()
join()         groupByKey()
first()        count()
```

… and more …

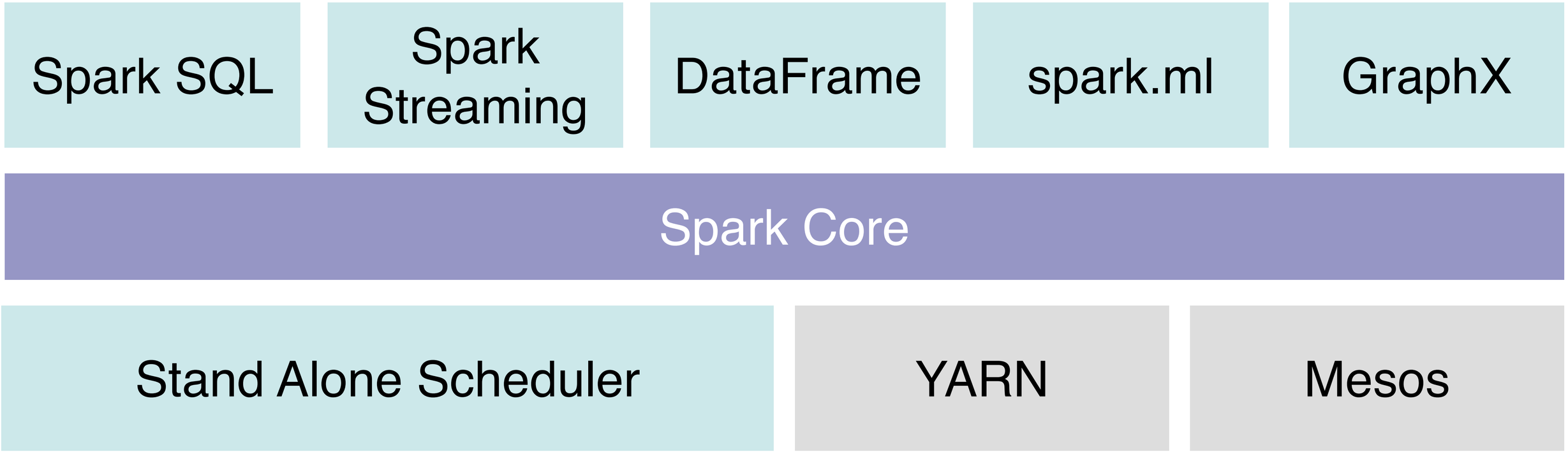

```
map()
reduce()
```

Rich API

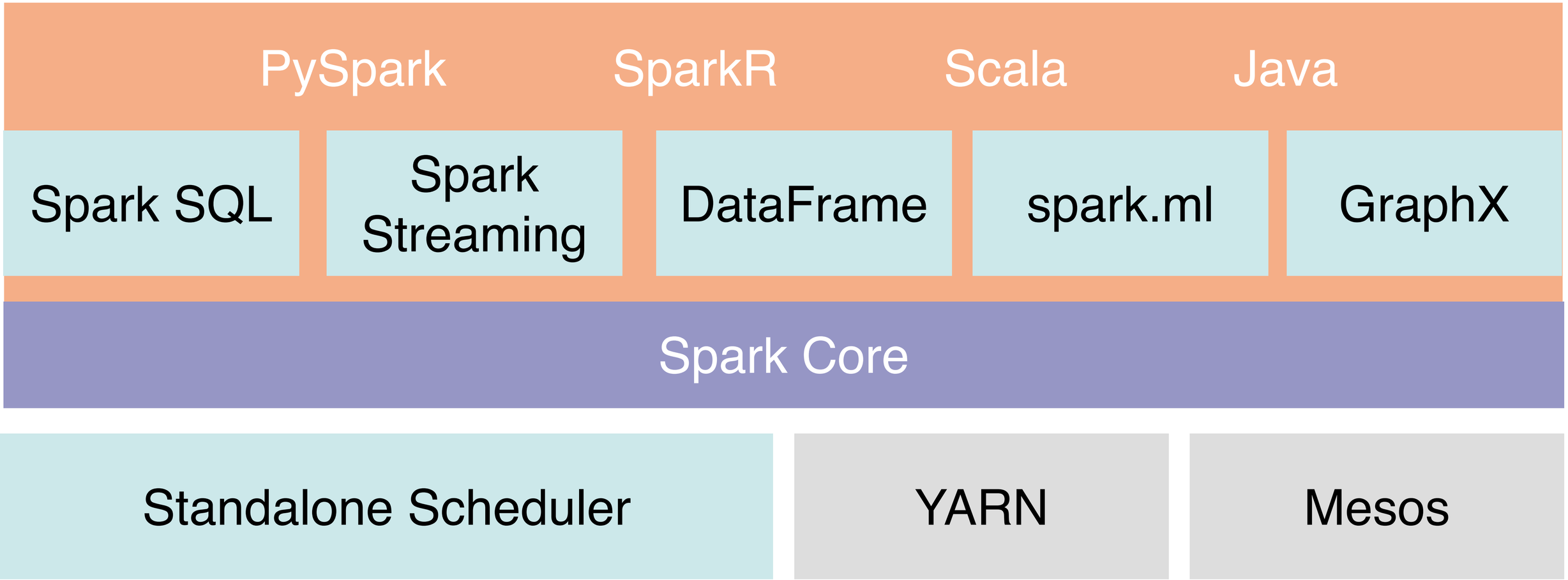

```
text_file = spark.textFile("hdfs://...")

text_file.flatMap(lambda line: line.split())
     .map(lambda word: (word, 1))
     .reduceByKey(lambda a, b: a+b)
```

```
package org.myorg;

import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class WordCount {

 public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
    }
 }

 public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
    }
 }

 public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);
 }

}
```

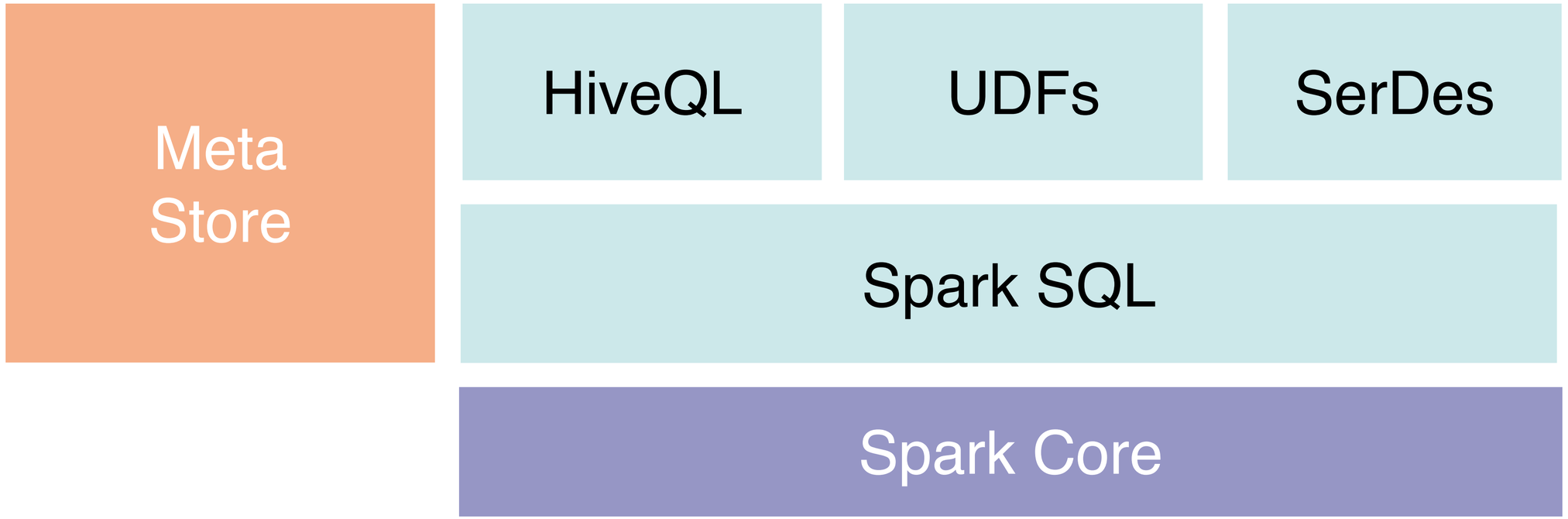
Unified Platform



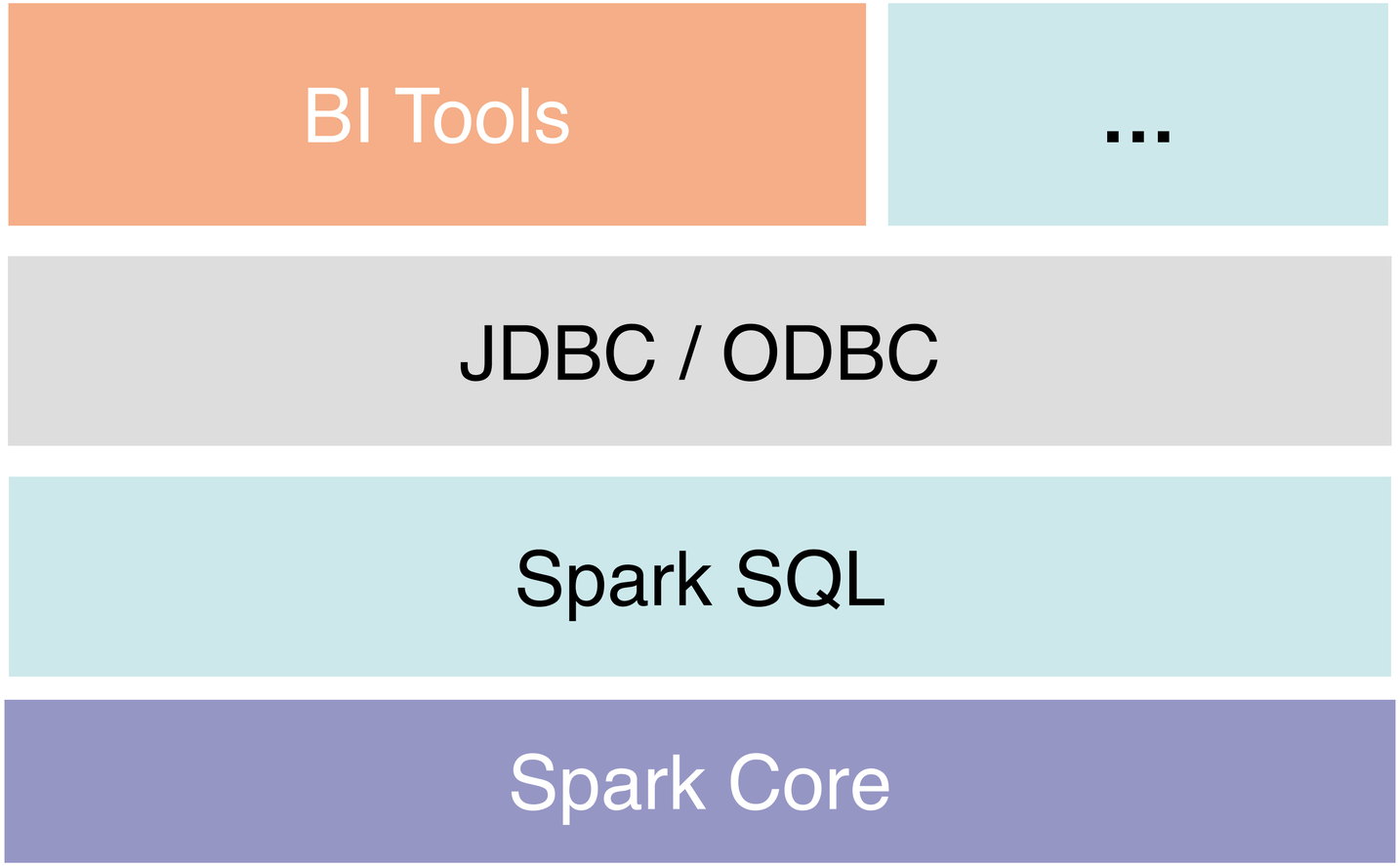
Unified Platform



Hive Integration



Standard RDBMS Integration



Review

- Framework for distributed processing
- In-memory, fault tolerant data structures
- Flexible APIs in Scala, Java, Python, SQL… and now R!
- Open Source

APIs

Spark

- Coarse grained
- SQL like abstractions (tables)
- Synchronous
- Functional

Ray

- Fine grained
- Actor based abstractions (agents)
- Asynchronous
- Object Oriented

Use Cases

Spark

- Data processing and transformation
- Data heavy workloads
- Iterative MapReduce
- Ex: Pagerank

Ray

- Distributed asynchronous algorithms
- Compute heavy workloads
- Ex: Reinforcement learning

Live Coding

```
import pyspark as ps
import random

# initialize SparkContext
sc = ps.SparkContext()

flips = 1000000

# lazy eval
coins = xrange(flips)

# lazy eval, nothing executed
heads = sc.parallelize(coins) \
          .map(lambda i: random.random()) \
          .filter(lambda r: r < 0.51) \
          .count()

print "{0} heads flipped!".format(heads)

# shutdown SparkContext
sc.stop()
```

What is an RDD?

- **Resilient:** if the data in memory (or on a node) is lost, it can be recreated
- **Distributed:** data is chunked into partitions and stored in memory across the cluster
- **Dataset:** initial data can come from a file or be created programmatically

Note: RDDs are read-only and immutable, we will come back to this later…

Functions Deconstructed

```
import random
flips = 1000000

# lazy eval
coins = range(flips)

# lazy eval, nothing executed
heads = sc.parallelize(coins) \
          .map(lambda i: random.random()) \
          .filter(lambda r: r < 0.51) \
          .count()
```

Python Generator

Create RDD

Transformations

Action (materialize result)

Functions Deconstructed

```
import random
flips = 1000000

# lazy eval
coins = range(flips)

# lazy eval, nothing executed
heads = sc.parallelize(coins) \
        .map(lambda i: random.random()) \
        .filter(lambda r: r < 0.51) \
        .count()


 # create a closure with the lambda function
 # apply function to data
```

Closures

Functions Deconstructed

```
import random
flips = 1000000

# local sequence
coins = range(flips)

# distributed sequence
coin_rdd = sc.parallelize(coins)
flips_rdd = coin_rdd.map(lambda i: random.random())
heads_rdd = flips_rdd.filter(lambda r: r < 0.51)

# local value
head_count = heads_rdd.count()
```

Functional Programming Primer

- Functions are `applied` to data (RDDs)
- RDDs are Immutable: `f(RDD) -> RDD2`
- Function application necessitates creation of new data

Spark Functions

Transformations	Actions
Lazy Evaluation (does not immediately evaluate)	Materialize Data (evaluates RDD lineage)
Returns new RDD	Returns final value (on driver)

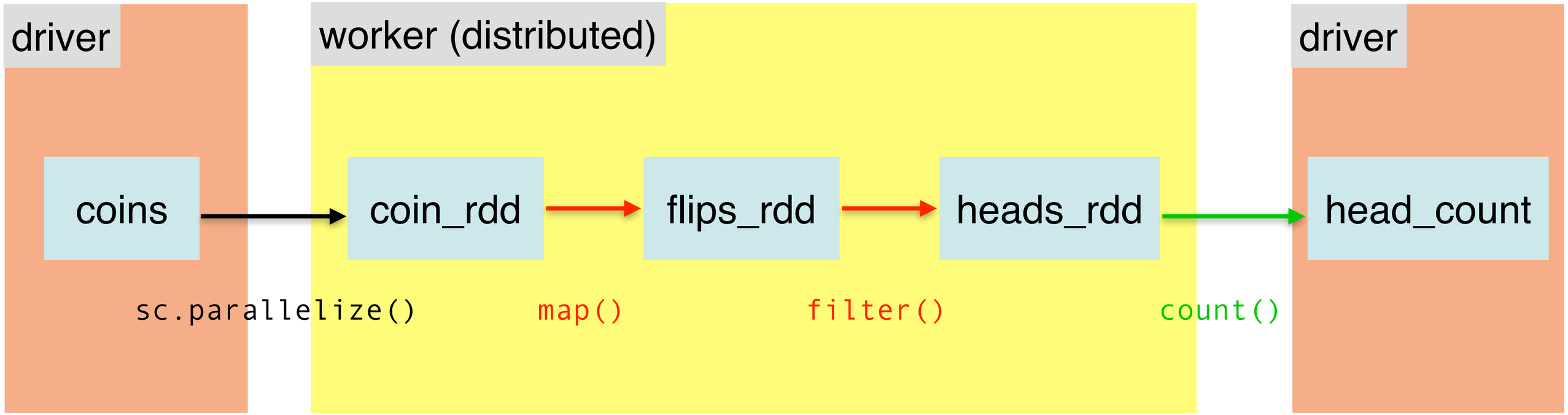
Transformations

```
# Every Spark application requires a Spark Context
# Spark shell provides a preconfigured Spark Context called `sc`
nums = sc.parallelize([1,2,3])

# Pass each element through a function
squared = nums.map(lambda x: x*x) # => {1, 4, 9}

# Keep elements passing a predicate
even = squared.filter(lambda x: x % 2 == 0) # => [4]

# Map each element to zero or more others
nums.flatMap(lambda x: range(x)) # => {0, 0, 1, 0, 1, 2}
```

Actions

```
nums = sc.parallelize([1, 2, 3])

# Retrieves RDD contents as a local collection
nums.collect() # => [1, 2, 3]

# Returms first K elements
nums.take(2) # => [1, 2]

# Count number of elements
nums.count() # => 3

# Merge elements with an associative function
nums.reduce(lambda: x, y: x + y) # => 6

# Write elements to a text file
nums.saveAsTextFile("hdfs://file.txt")
```

RDD Lineage


driver
worker (distributed)
driver
coins
coin_rdd
flips_rdd
heads_rdd
head_count
sc.parallelize()
map()
filter()
count()

Functions Deconstructed

```
import random
flips = 1000000

# lazy eval
coins = xrange(flips)
```

nothing runs here

```
# lazy eval, nothing executed
heads_rdd = sc.parallelize(coins) \
          .map(lambda i: random.random()) \
          .filter(lambda r: r < 0.51)
```

```
head_count = heads_rdd.count()
```

Everything runs here

What is an RDD?

An *Abstraction!*

What is an RDD?

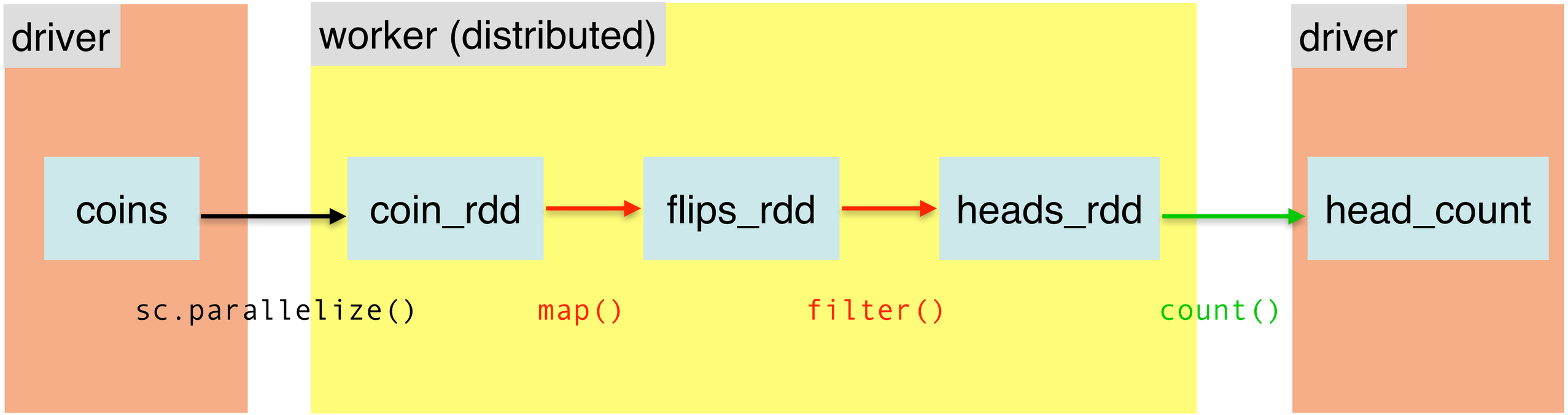
Lineage (required)

1. Set of partitions for current RDD (data)
2. List of dependencies
3. Function to compute partitions (functional paradigm)

4. Partitioner to optimize execution
5. Potential preferred location for partitions

Optimization (optional)

RDD Lineage


driver
worker (distributed)
driver
coins
coin_rdd
flips_rdd
heads_rdd
head_count
sc.parallelize()
map()
filter()
count()

What is an RDD?

Lineage (required)

1. Set of partitions for current RDD (data)
2. List of dependencies
3. Function to compute partitions (functional paradigm)

4. Partitioner to optimize execution
5. Potential preferred location for partitions

Optimization (optional)

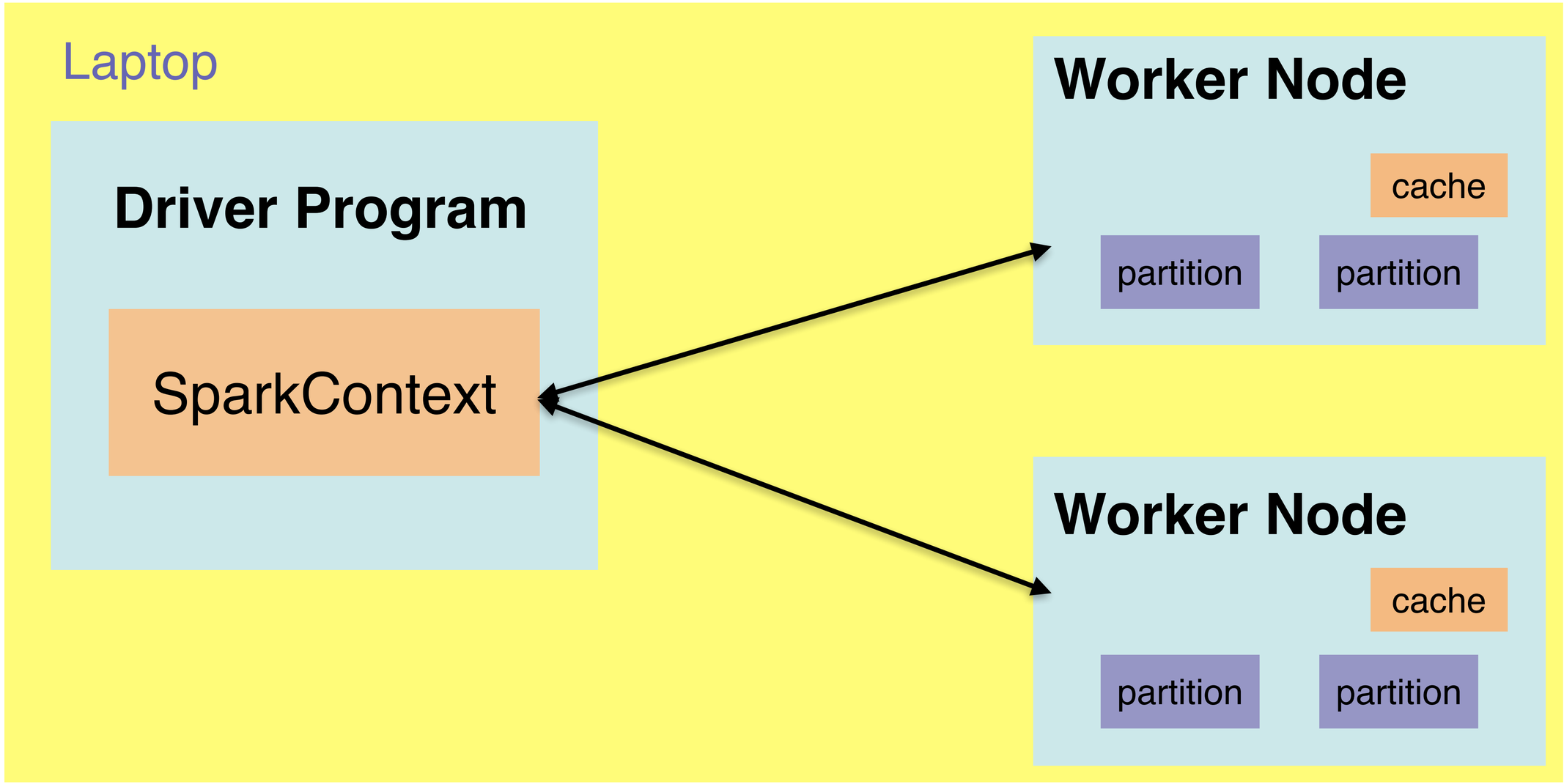
RDD as an interface

Operation	*Meaning*
`partitions()`	Return a list of `Partition` objects
`dependencies()`	Return a list of dependencies
`compute(p, parent)`	Compute the elements of `Partition` ***p*** given its ***parent*** `Partitions`
`partitioner()`	Return metadata specifying whether this RDD is hash/range partitioned
`preferredLocations(p)`	List nodes where `Partition` ***p*** can be accessed quicker due to data locality

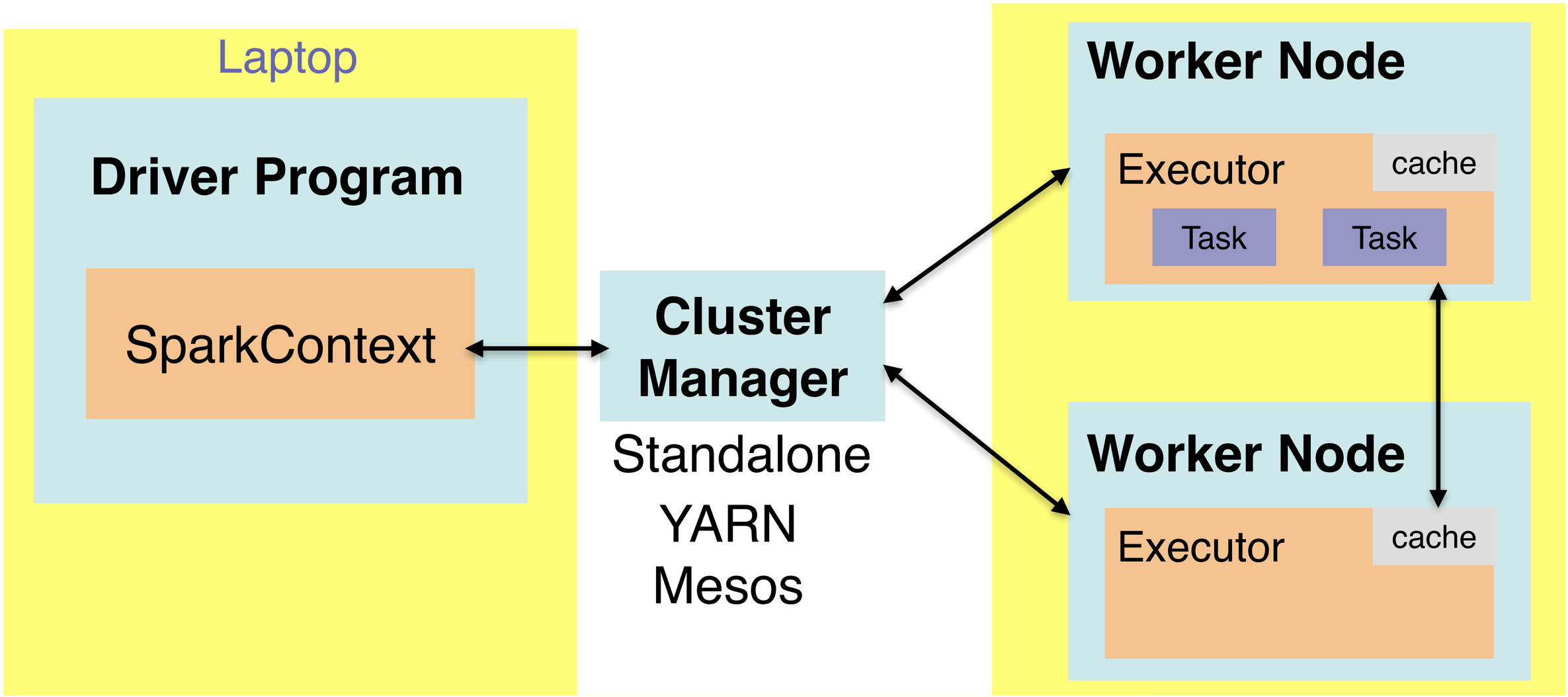
Key-Value Operations

```
pets = sc.parallelize([("cat", 1), ("dog", 1), ("cat", 2)])

pets.reduceByKey(lambda x, y: x + y) # => {(cat, 3), (dog, 1)}

pets.groupByKey() # => {(cat, [1, 2]), (dog, [1])}

pets.sortByKey() # => {(cat, 1), (cat, 2), (dog, 1)}
```

Local Spark Execution Context



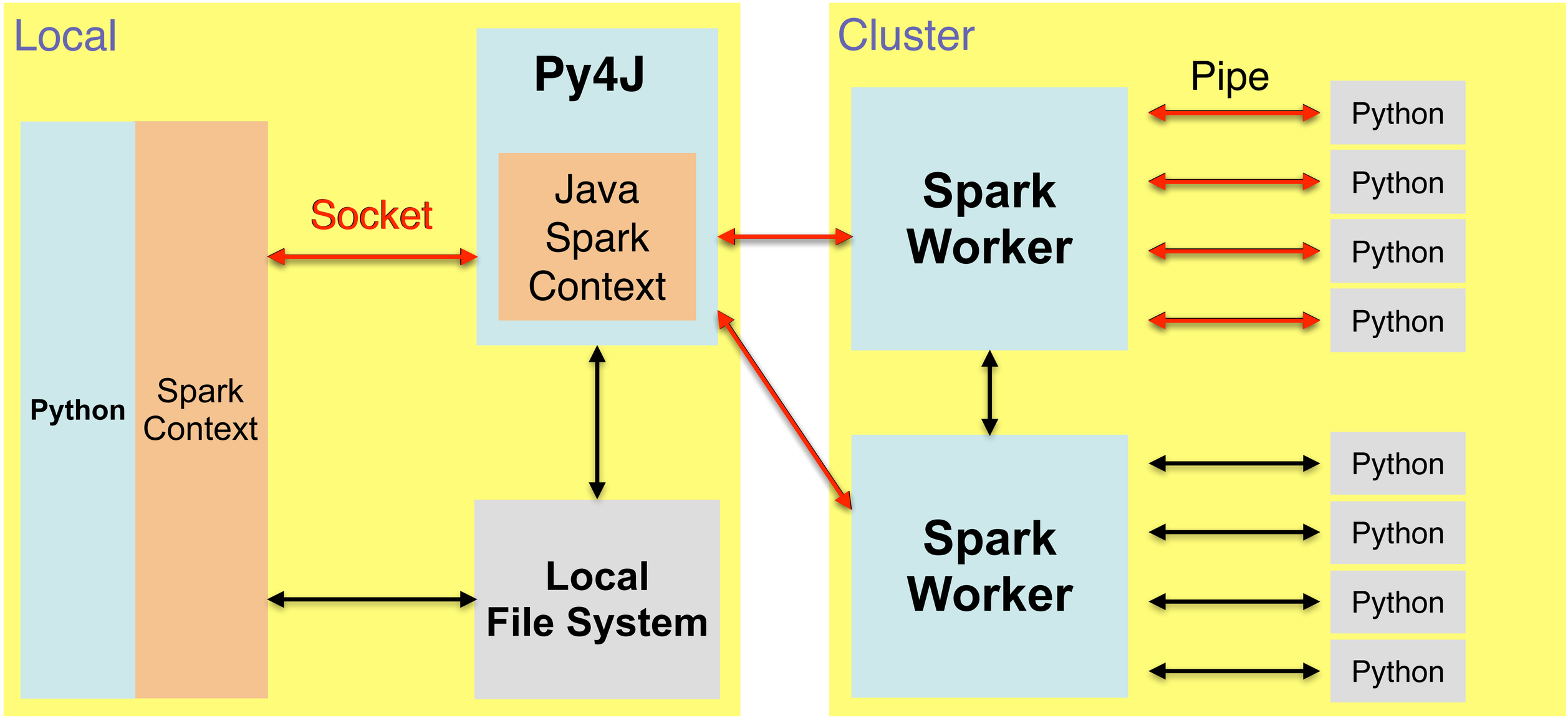
Spark on a Cluster



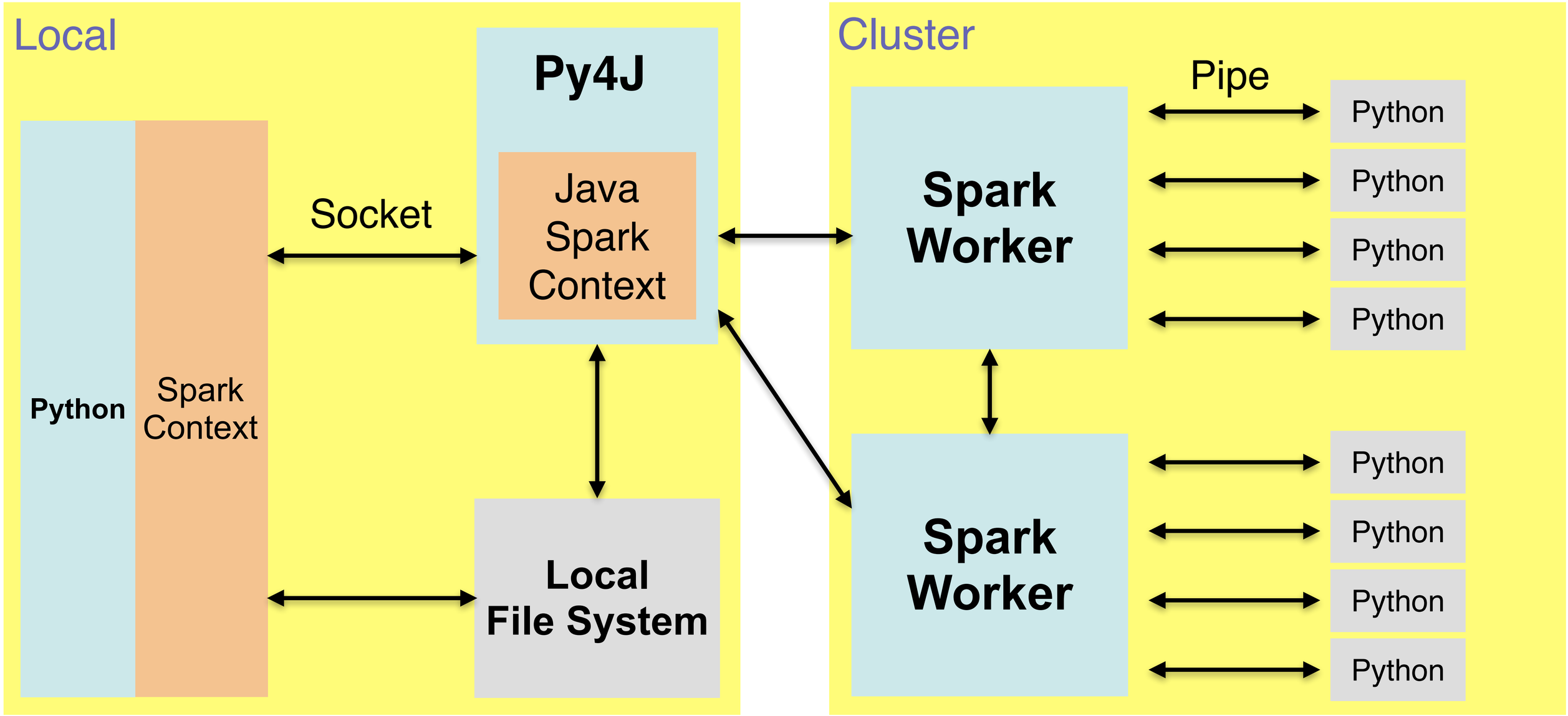
Terminology

Term	*Meaning*
Driver	*Process that contains the SparkContext*
Executor	*Process that executes one or more Spark tasks*
Master	*Process that manages applications across the cluster*
Worker	*Process that manages executors on a particular node*

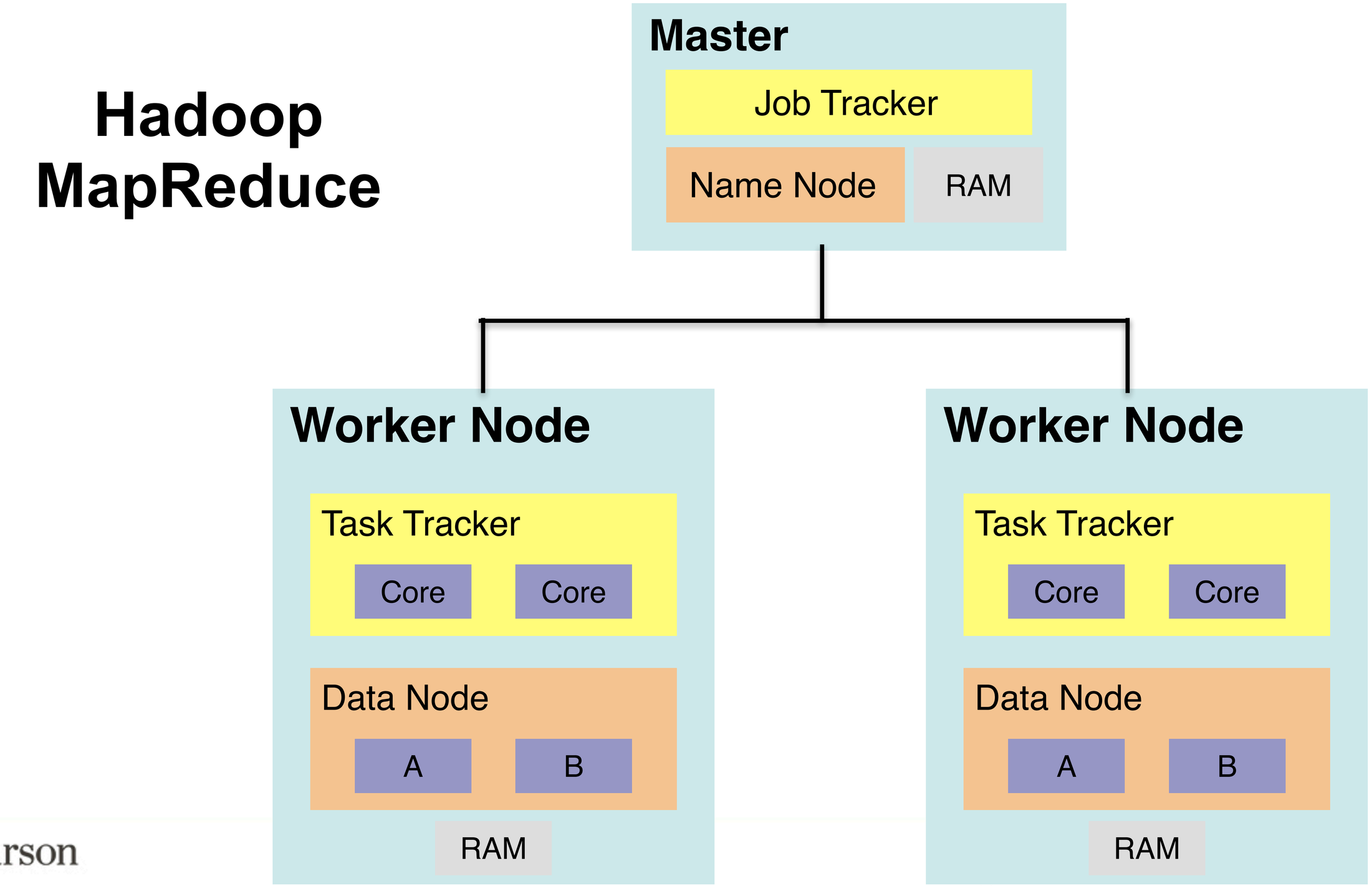
PySpark


Local
Cluster
Py4J
Pipe
Python
Spark Context
Socket
Java Spark Context
Spark Worker
Python
Python
Python
Python
Spark Worker
Python
Python
Python
Python
Local File System

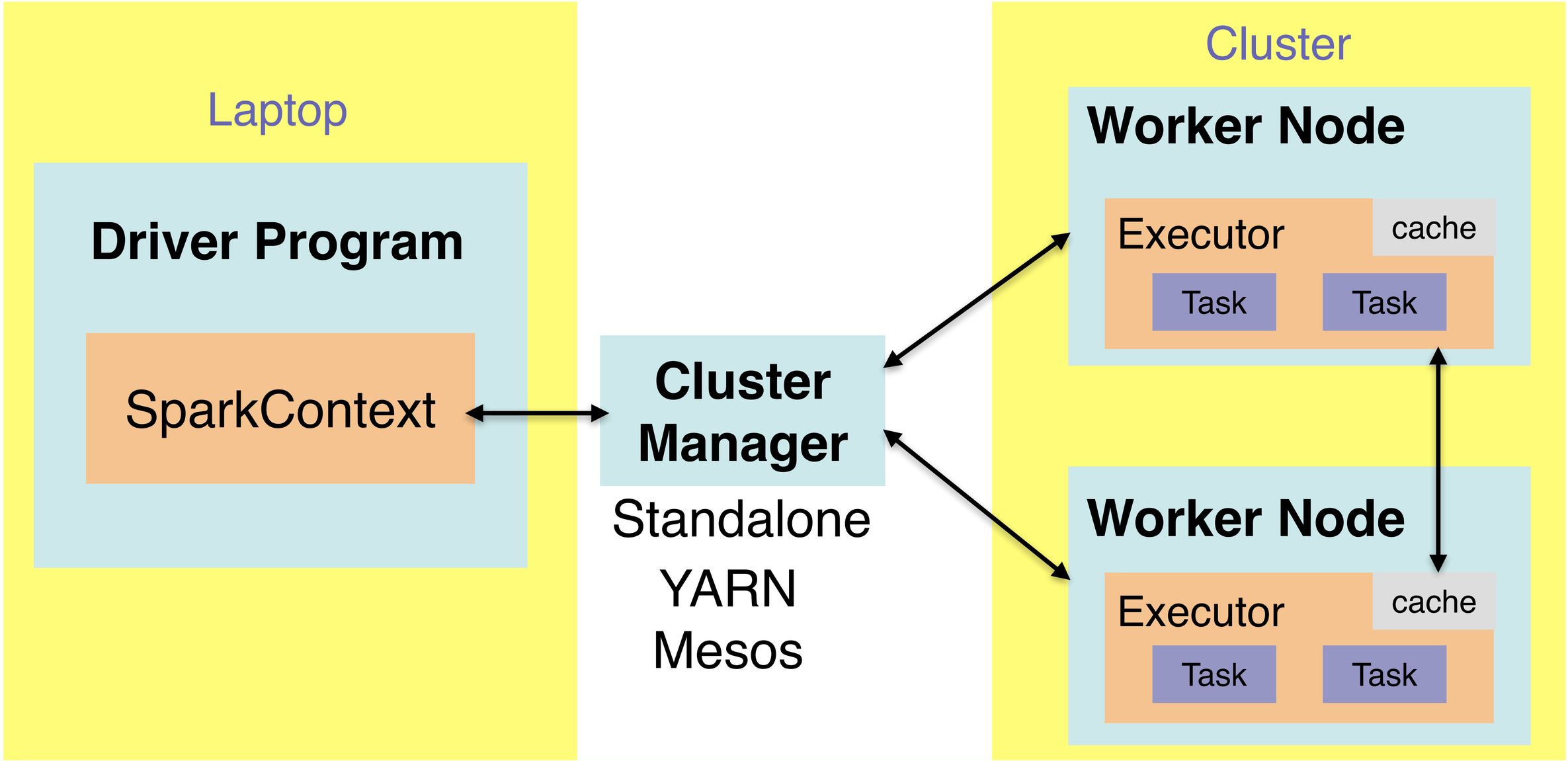
PySpark


Local
Cluster
Py4J
Pipe
Python
Spark Context
Socket
Java Spark Context
Spark Worker
Python
Python
Python
Python
Spark Worker
Python
Python
Python
Python
Local File System

Hadoop MapReduce



Spark Execution Context



Spark vs. Hadoop

- Spark only replaces MapReduce (computation)
- Still need a data store: HDFS, HBase, Hive, etc.
- Spark has a more flexible/general programming model
- Spark often faster for iterative computation

A Day in the Life of a Spark Application

1. Determine RDD lineage: construct DAG

2. Create execution plan for DAG: determine stages

3. Schedule and execute tasks

What's in a Task?

```
sc.textFile('file:///dummy.txt')
```

```
map(line => line.split(" "))
```

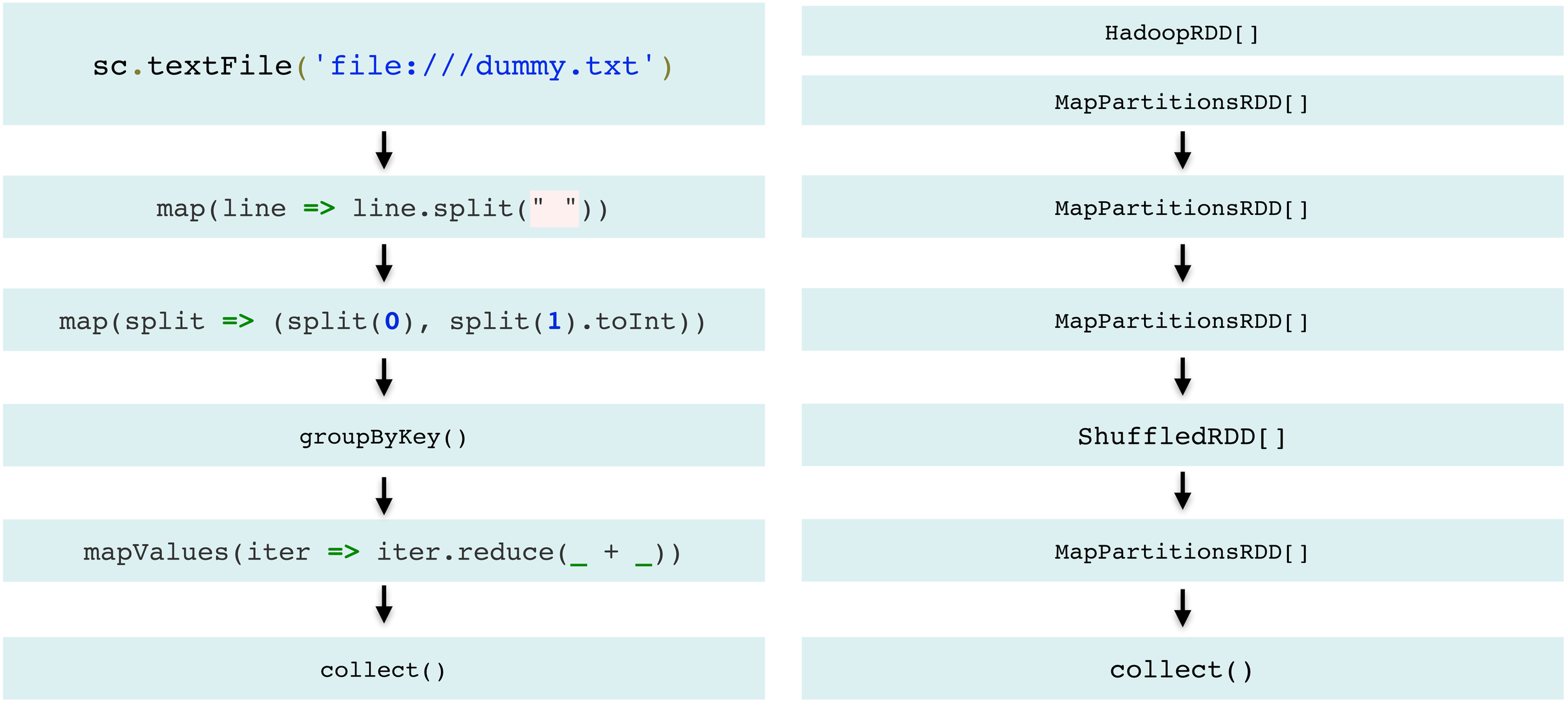
```
map(split => (split(0), split(1).toInt))
```

```
groupByKey()
```

```
mapValues(iter => iter.reduce(_ + _))
```

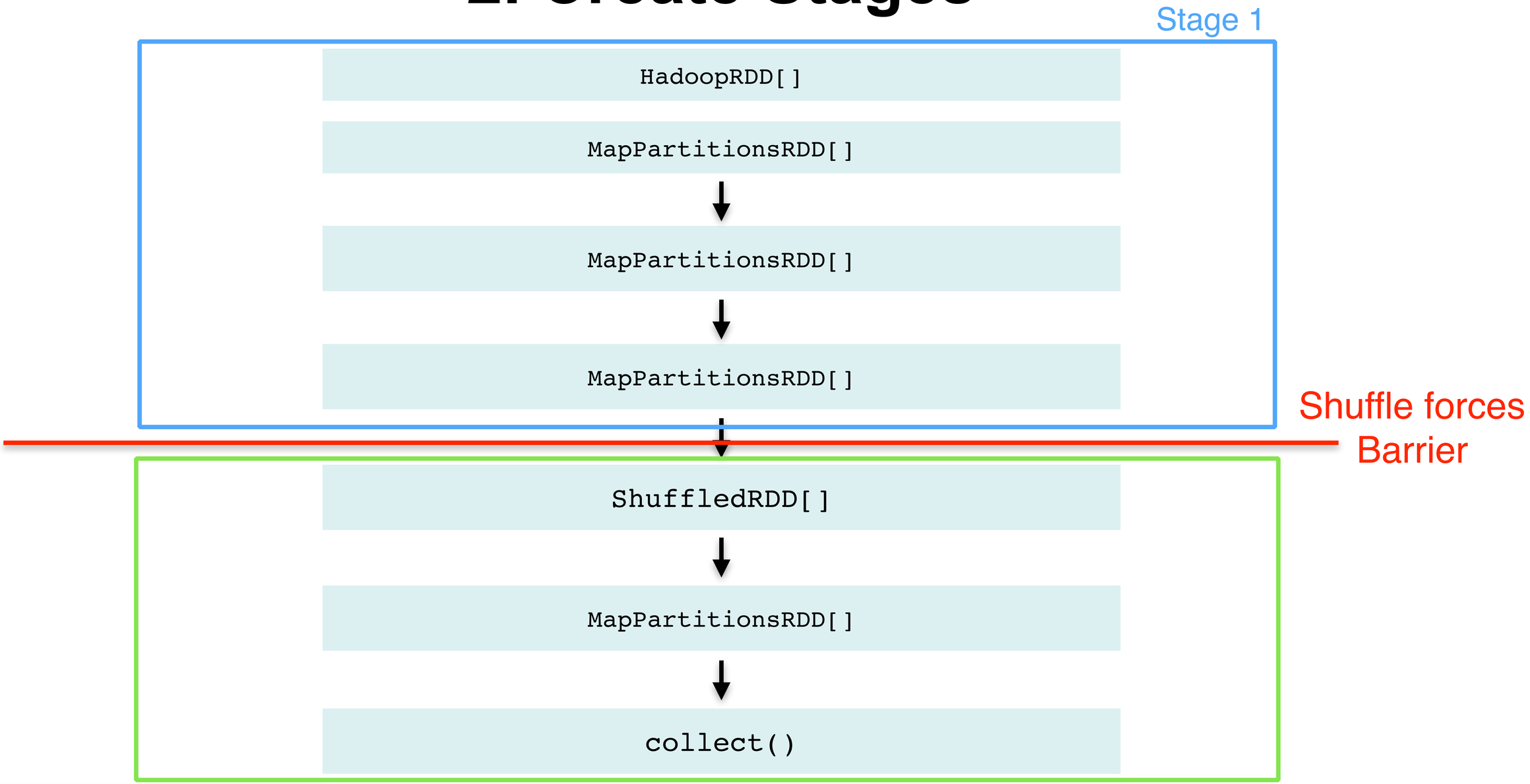
```
collect()
```

1. Create RDDs



```
scala> file_rdd.partitions
res51: Array[org.apache.spark.Partition] = Array(org.apache.spark.rdd.HadoopPartition@a17, org.apache.spark.rdd.HadoopPartition@a18)

scala> file_rdd.partitions.size
res52: Int = 2

scala> file_rdd.map(line => line.split(" ")).toDebugString
res53: String =
(2) MapPartitionsRDD[65] at map at <console>:24 []
 |  MapPartitionsRDD[23] at textFile at <console>:21 []
 |  file:///Users/jonathandinu/spark-ds-applications/dummy.txt HadoopRDD[22] at textFile at <console>:21 []

scala> file_rdd.map(line => line.split(" ")).map(split => (split(0), split(1).toInt)).toDebugString
res54: String =
(2) MapPartitionsRDD[67] at map at <console>:24 []
 |  MapPartitionsRDD[66] at map at <console>:24 []
 |  MapPartitionsRDD[23] at textFile at <console>:21 []
 |  file:///Users/jonathandinu/spark-ds-applications/dummy.txt HadoopRDD[22] at textFile at <console>:21 []

scala> file_rdd.map(line => line.split(" ")).map(split => (split(0), split(1).toInt)).groupByKey().toDebugString
res55: String =
(2) ShuffledRDD[70] at groupByKey at <console>:24 []
 +-(2) MapPartitionsRDD[69] at map at <console>:24 []
    |  MapPartitionsRDD[68] at map at <console>:24 []
    |  MapPartitionsRDD[23] at textFile at <console>:21 []
    |  file:///Users/jonathandinu/spark-ds-applications/dummy.txt HadoopRDD[22] at textFile at <console>:21 []

scala> file_rdd.map(line => line.split(" ")).map(split => (split(0), split(1).toInt)).groupByKey().mapValues(iter => iter.reduce(_ + _)).toDebugString
res56: String =
(2) MapPartitionsRDD[74] at mapValues at <console>:26 []
 |  ShuffledRDD[73] at groupByKey at <console>:26 []
 +-(2) MapPartitionsRDD[72] at map at <console>:26 []
    |  MapPartitionsRDD[71] at map at <console>:26 []
    |  MapPartitionsRDD[23] at textFile at <console>:21 []
    |  file:///Users/jonathandinu/spark-ds-applications/dummy.txt HadoopRDD[22] at textFile at <console>:21 []

scala> file_rdd.map(line => line.split(" ")).map(split => (split(0), split(1).toInt)).groupByKey().mapValues(iter => iter.reduce(_ + _)).collect()
res57: Array[(String, Int)] = Array((anna,1), (jesse,3), (jon,3), (mary,8))
```

2. Create Stages



3. Schedule Tasks

1. Split each stage into tasks to execute

2. Task is simply a partition (data) and computation (`lambda`)

3. Execute all tasks in a stage before continuing

3. Schedule Tasks

Stage 1

```
sc.textFile('file:///dummy.txt')
```

↓

```
map(line => line.split(" "))
```

↓

```
map(split => (split(0), split(1).toInt))
```

Task 0

Task 1

dummy.txt

```
jon 2
mary 3
anna 1
jon 1
```

```
jesse 1
mary 5
```

Task 0

```
partition 0
```

↓

```
HadoopRDD
```

↓

```
map(line => line.split(" "))
```

↓

```
map(split => (split(0), split(1).toInt))
```

Task 1

```
partition 1
```

↓

```
HadoopRDD
```

↓

```
map(line => line.split(" "))
```

↓

```
map(split => (split(0), split(1).toInt))
```

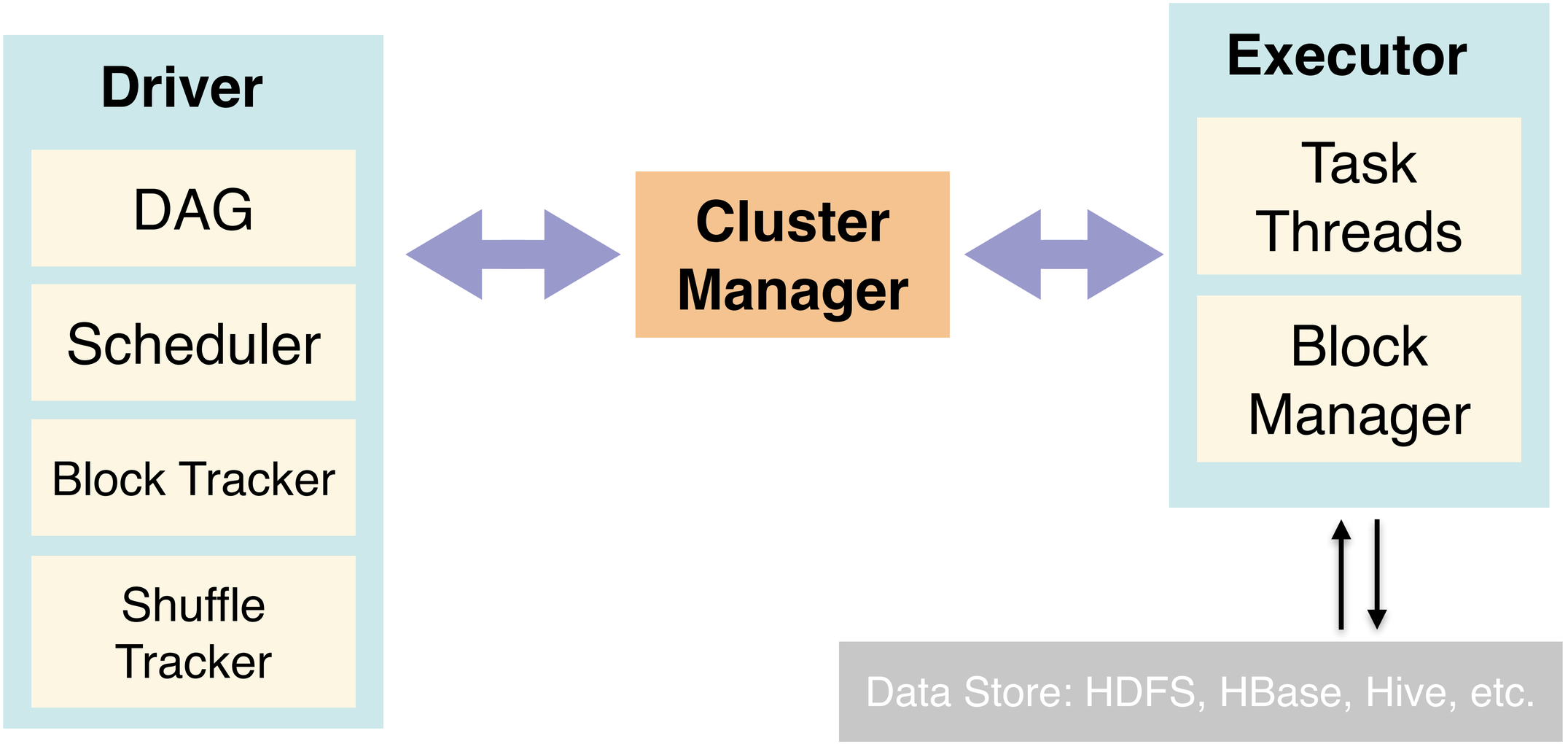
Scheduler

- RDD and partition DAG as input
- Tasks (within stages) to execute as output

Roles

- Builds stages to execute
- Submit stages to Cluster Manager
- Resubmit failed stages when output is lost

A Day in the Life of a Spark Application



What Is Exploratory Data Analysis?

> *But as much as EDA is a set of tools, it's also a mindset. And that mindset is about your relationship with the data… EDA happens between you and the data and isn't about proving anything to anyone else yet.*

- Cathy O'Neil (Doing Data Science)

What Is Exploratory Data Analysis?

- Developed at Bell Labs in the 1960's by John Tukey
- Techniques used to visualize and summarize data
  - Five-number summary: `describe()`
  - Distributions: box plots, stem and leaf, histogram, scatterplot

Goals of Exploratory Data Analysis

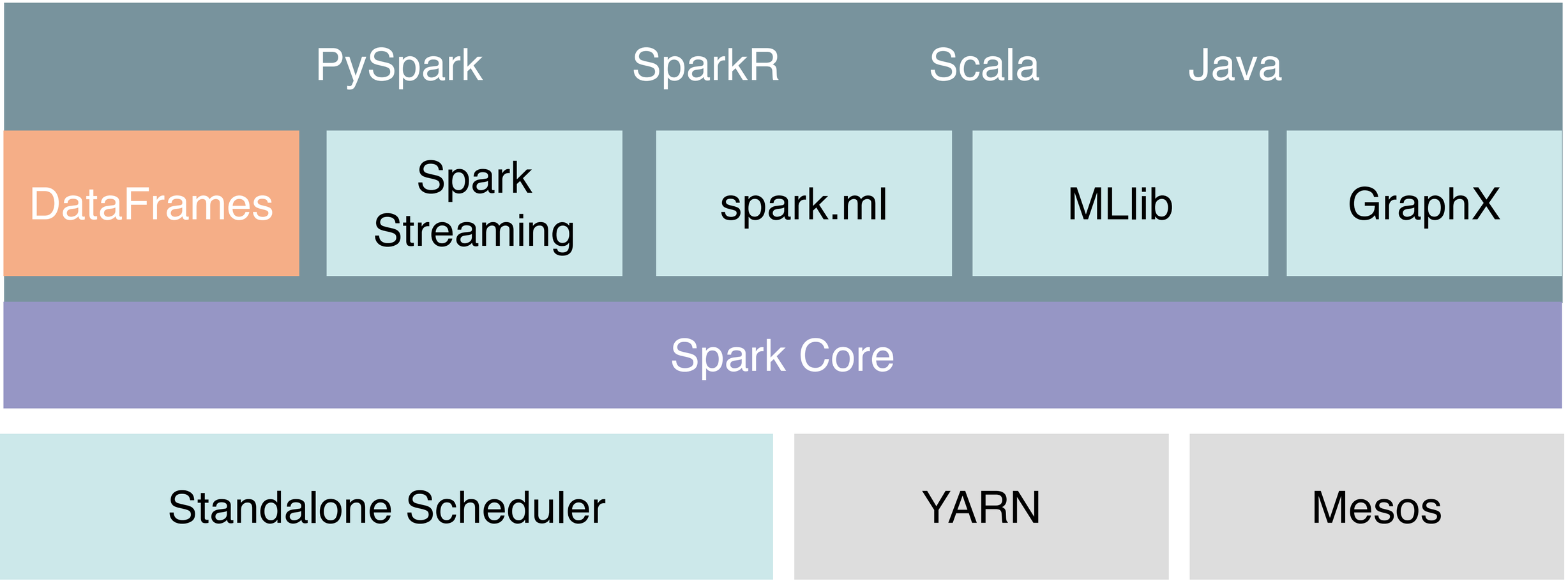
- Gain greater intuition
- Validate our data (consistency and completeness)
- Make comparisons between distributions
- Find outliers
- Treat missing data
- Summarize data (a statistic -> one number that represents many #'s)

How Can Spark Help?

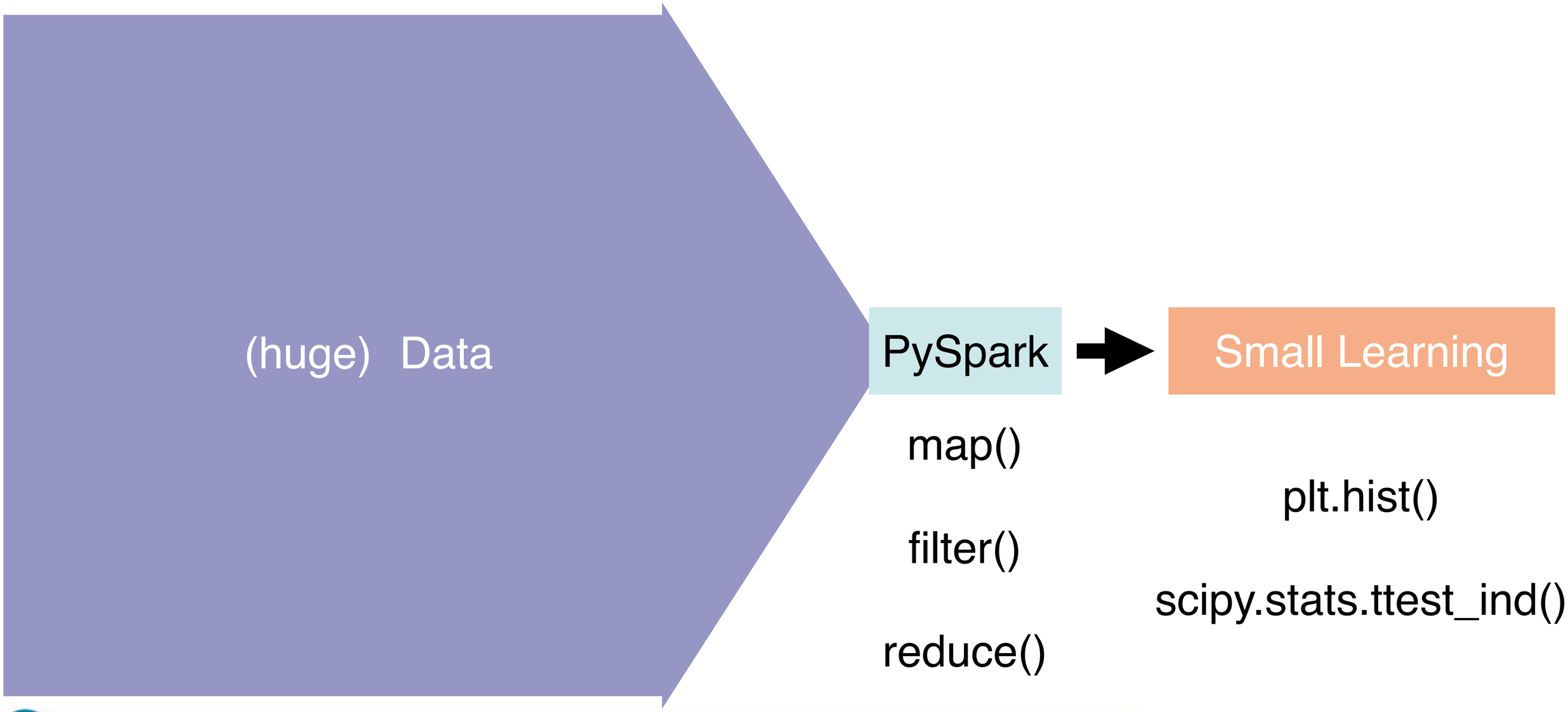
- Interactive REPL
- Rapid computation (especially aggregates) on large amounts of data
- High level abstractions for querying data
- “Condense” data for easier local exploration and visualization

Common Questions

- How many records in total are there?
- How many uniques values does each column contain?
- How many missing (or null) values are there?
- For numeric columns, what are its summary statistics
- What are values appear most often in each column?
- How are the values of each column distributed?


PySpark
SparkR
Scala
Java
DataFrames
Spark Streaming
spark.ml
MLlib
GraphX
Spark Core
Standalone Scheduler
YARN
Mesos

Big Data… Small Learning



Unique Values

- `rdd.distinct()`
- `rdd.countApproxDistinct(relative_accuracy)`

```
# HyperLogLog
rdd_dict.map(lambda row: row['_schoolid']).countApproxDistinct()
```

```
62989L
```

```
rdd_dict.map(lambda row: row['_schoolid']).distinct().count()
```

```
61402
```

http://dx.doi.org/10.1145/2452376.2452456

http://content.research.neustar.biz/blog/hll.html

Missing Values

- `column.isNull()`
- `dataframe.dropna(column_name)`
- `dataframe.fillna()`
- `dataframe.replace(to_replace, value)`
- `pyspark.accumulators`

https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrameNaFunctions

Missing Values

- `column.isNull()`
- `dataframe.fillna()`

```
df.filter(df_dates['students_reached'].isNull()).select('students_reached', 'funding_status').collect()
```

```
[Row(students_reached=None, funding_status=u'completed'),
 Row(students_reached=None, funding_status=u'completed'),
 Row(students_reached=None, funding_status=u'expired'),
 Row(students_reached=None, funding_status=u'completed'),
                            ...
```

```
df_no_null = df.fillna(0, ['students_reached'])
```

Frequently Occurring Values

- `dataframe.freqItems(columns, support)`

Note: this is an approximate algorithm that **always** returns all the frequent items, but may contain false positives.

required minimum proportion of rows

```
freq_items = df.freqItems(['school_city', 'primary_focus_area', \
                           'grade_level', 'poverty_level','resource'], 0.7).collect()
```

```
freq_items[0]
```

```
Row(school_city_freqItems=[u'Los Angeles'], primary_focus_area_freqItems=[u'Literacy & Language'], grade_level_freqIt
ems=[u'Grades PreK-2'], poverty_level_freqItems=[u'highest poverty'], resource_freqItems=[u'Supplies'])
```

http://dl.acm.org/citation.cfm?doid=762471.762473

Summary Statistics

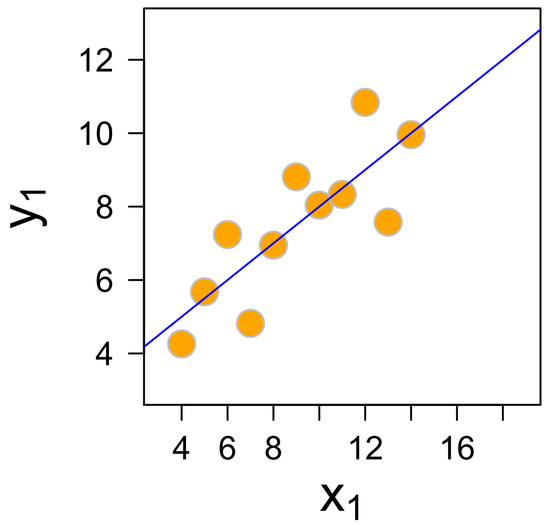
- `dataframe.describe(column_name)`

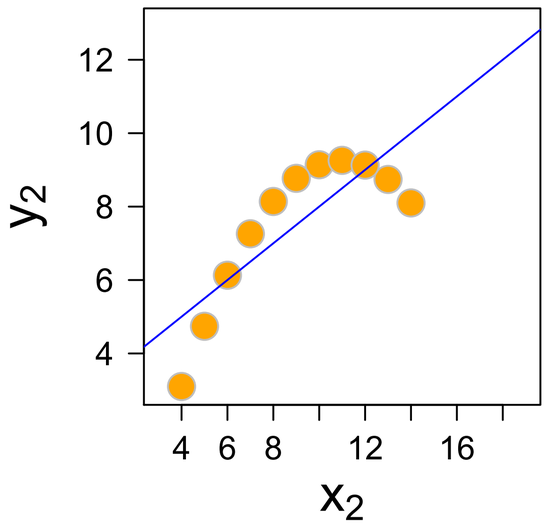
```
df.select('total_donations', 'num_donors', 'students_reached', \
                df_dates['total_price_excluding_optional_support'].alias('p_exclude'), \
                df_dates['total_price_including_optional_support'].alias('p_include')) \
  .describe().show()
```

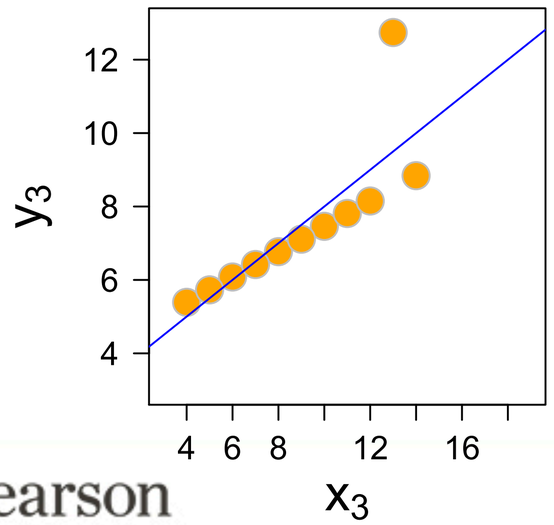
```
+-------+-----------------+-----------------+-----------------+------------------+-----------------+
|summary|  total_donations|       num_donors| students_reached|         p_exclude|         p_include|
+-------+-----------------+-----------------+-----------------+------------------+-----------------+
|  count|           771929|           771929|           771779|            771929|           771929|
|   mean|370.85023398481707|4.264279486843997|96.71620114048193| 569.6223687446723| 676.180708551764|
| stddev| 733.4647726421459|6.132976060232441|2118.592960253374|11763.955807309705|14344.347534777195|
|    min|              0.0|              0.0|              0.0|               0.0|              0.0|
|    max|         244778.0|            521.0|         999999.0|       1.0250017E7|       1.2500021E7|
+-------+-----------------+-----------------+-----------------+------------------+-----------------+
```

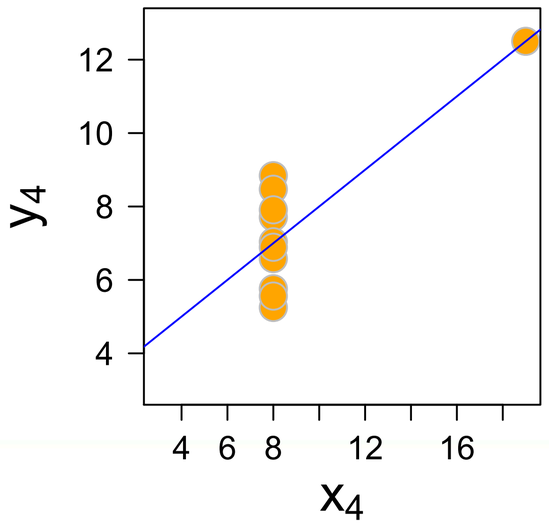
Interlude: Sometimes numbers aren’t enough!

Anscombe's Quartet









Mean (x)	9
Sample Variance (x)	11
Mean (y)	7.50
Sample Variance (y)	4.127
Correlation	0.816
Linear Regression	y = 3.00 + 0.500x

Distributions

RDD

- `rdd.histogram()`
- `rdd.stats()`

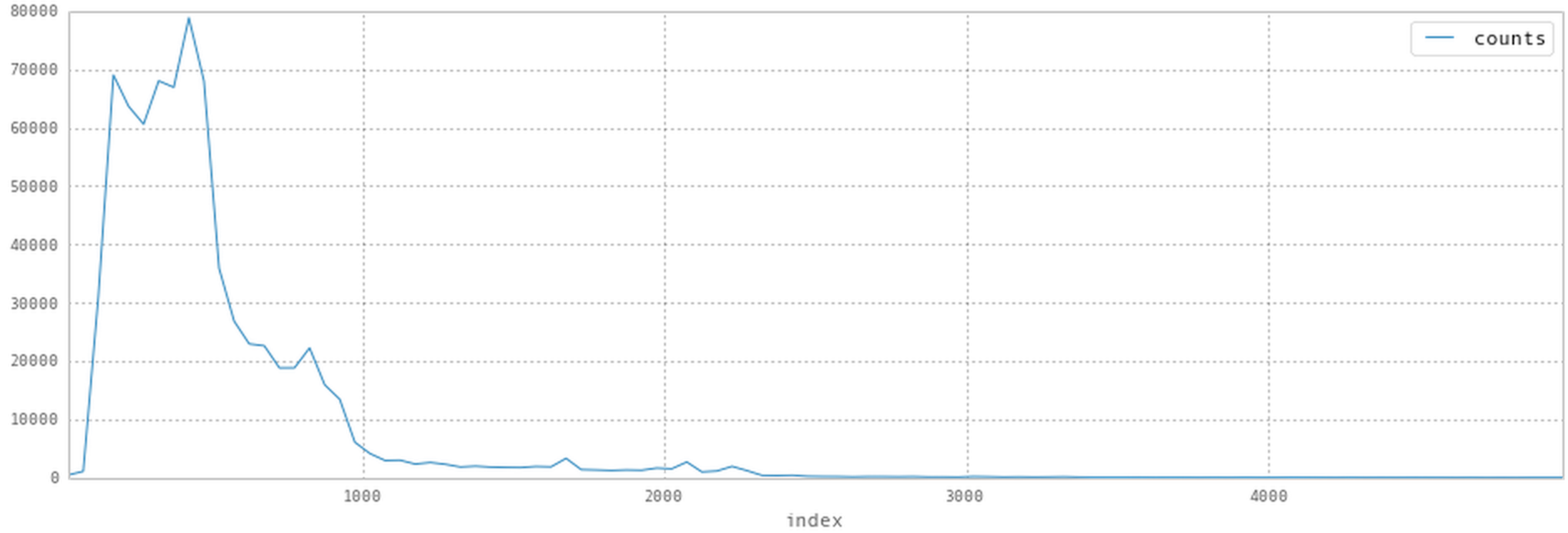
DataFrame

- `dataframe.groupby('column_name').count()`
- `dataframe.describe('column_name')`

```
price_rdd = df_no_null.select('total_price_excluding_optional_support').rdd.map(lambda r: r.asDict().values()[0])
```

```
def plot_rdd_hist(hist):
    idx = []

    for i in range(len(hist[0]) - 1):
        idx.append((hist[0][i] + hist[0][i+1])/ 2)

    pd.DataFrame({'counts': hist[1], 'index': idx}).set_index('index').plot(figsize=(16,5))
```

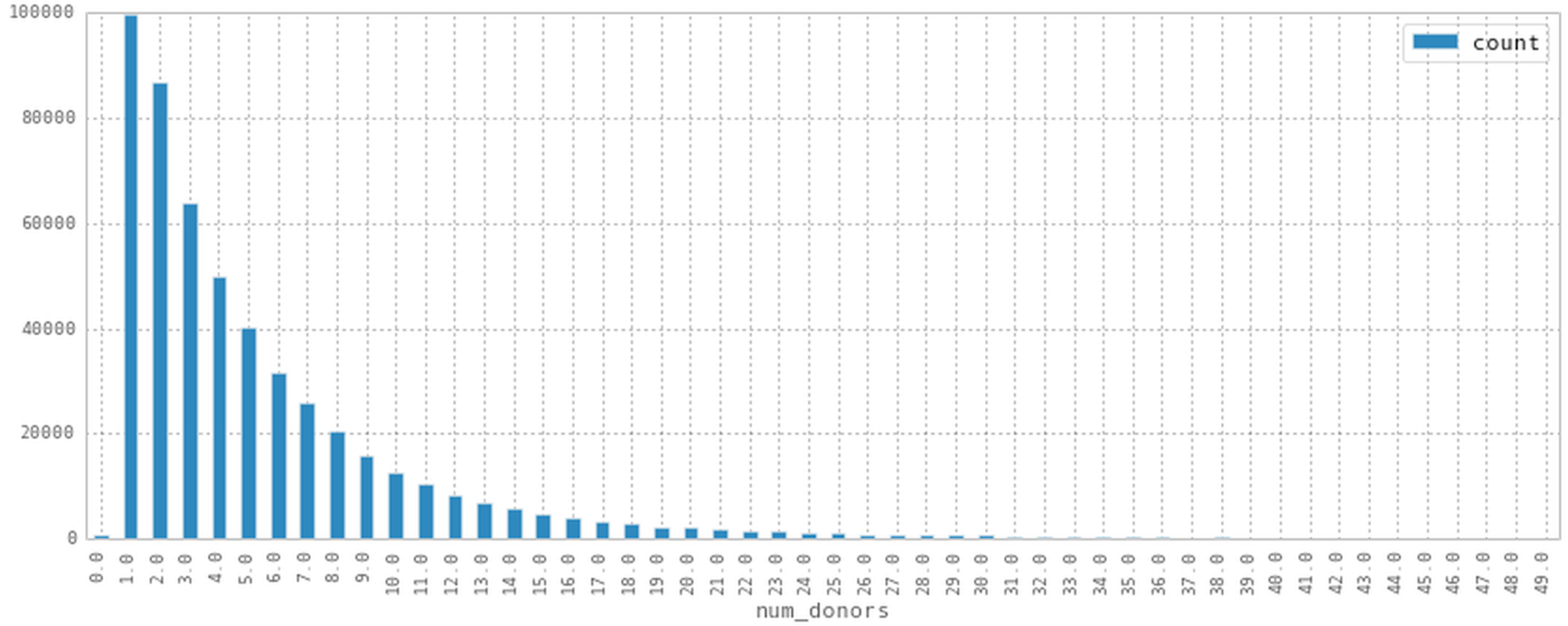
```
plot_rdd_hist(price_rdd.filter(lambda x: x < 5000).histogram(100))
```



```
def spark_histogram(df, column):
    donor_counts = df.groupby(column).count()
    donor_df = donor_counts.toPandas()
    donor_df[column] = donor_df.num_donors.astype(float)
    return donor_df.sort(column).set_index(column).iloc[:50,:].plot(kind='bar', figsize=(14,5))
```

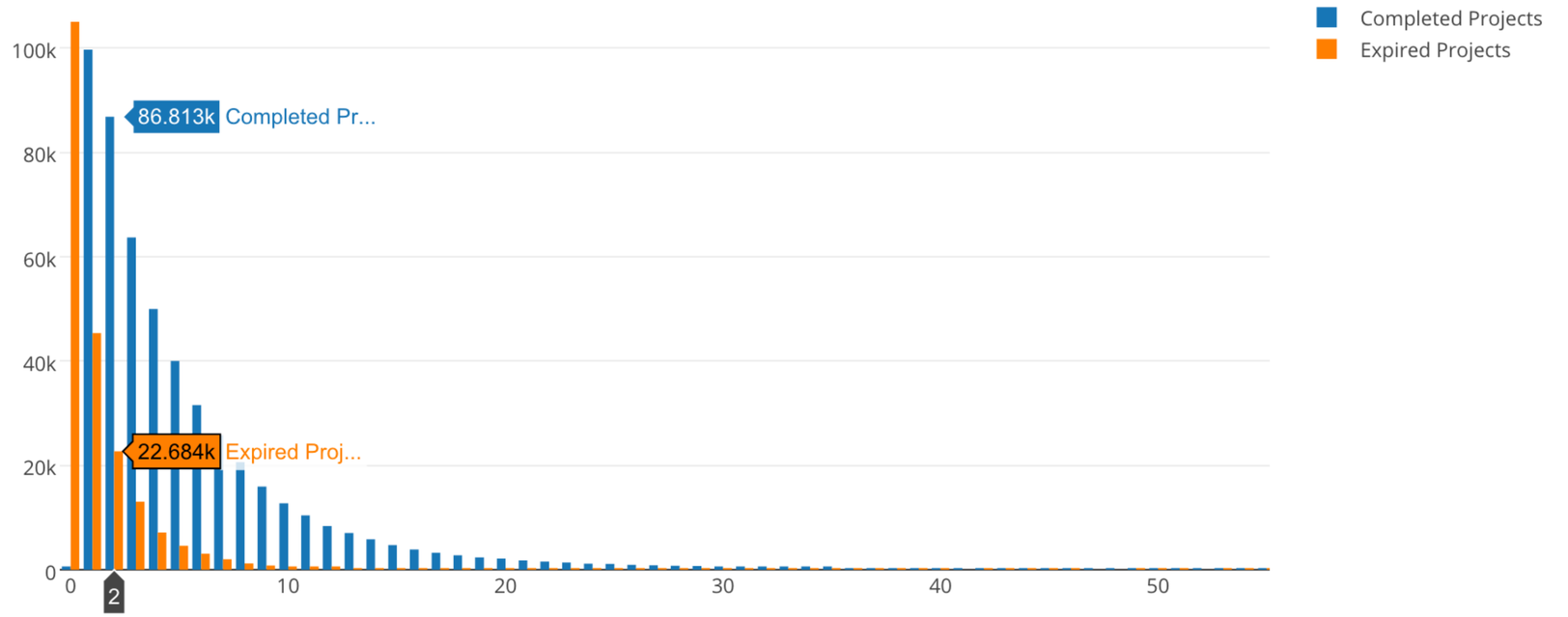
```
spark_histogram(df_complete, 'num_donors')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x113b2be50>
```



```
complete = df_complete.groupby('num_donors').count().toPandas()
expired = df_expired.groupby('num_donors').count().toPandas()
```

```
# And if we want an interactive plot, we can use a tool like plotly
# pip install plotly
import plotly.plotly as py
import plotly.tools as tls
from plotly.graph_objs import *

data = Data([
        Bar(x=complete['num_donors'], y= complete['count'], name="Completed Projects"),
        Bar(x=expired['num_donors'], y = expired['count'] , name="Expired Projects")
    ])

py.iplot(data)
```



Interactions

- dataframe.crosstab()
- dataframe.corr()

```
df_no_null.stat.corr('total_price_excluding_optional_support', 'num_donors')
```

```
0.007004254706419042
```

```
df_no_null.stat.corr('total_price_excluding_optional_support', 'students_reached')
```

```
0.0006159991686679948
```

```
df_no_null.stat.corr('total_price_excluding_optional_support', 'total_price_including_optional_support')
```

```
0.9999972199123168
```

```
df_dates.crosstab('resource_type', 'funding_status').show()
df_dates.crosstab('primary_focus_area', 'resource_type').show()
```

```
+----------------------------+-----+---------+-----------+-------+
|resource_type_funding_status| live|completed|reallocated|expired|
+----------------------------+-----+---------+-----------+-------+
|                        null|    2|       28|          0|     18|
|                       Other| 4542|    54610|        747|  22550|
|                       Books| 5982|   118810|       1527|  34554|
|                    Visitors|  102|      806|          6|    341|
|                    Supplies|11939|   185870|       2602|  63406|
|                       Trips|  347|     4381|         62|   1474|
|                  Technology|18957|   150500|       2256|  85510|
+----------------------------+-----+---------+-----------+-------+
```

```
+------------------------------+-----+--------+-----+----------+------+--------+----+
|primary_focus_area_resource_type|Trips|Visitors|Other|Technology| Books|Supplies|null|
+------------------------------+-----+--------+-----+----------+------+--------+----+
|              Literacy & Language|  630|     228|32795|    109605|127282|   75924|   4|
|                          null|    0|       0|    0|         0|     1|       0|  41|
|             Applied Learning| 1197|     104| 9429|     17869|  4863|   22596|   0|
|               Math & Science| 1902|     323|16353|     75189| 11746|   89101|   3|
|             Music & The Arts|  947|     441| 8305|     19289|  2883|   37804|   0|
|              Health & Sports|  159|      54| 4633|      3054|   432|   12970|   0|
|                 Special Needs|  241|      32| 7636|     19359|  4112|   17151|   0|
|             History & Civics| 1188|      73| 3298|     12858|  9554|    8271|   0|
+------------------------------+-----+--------+-----+----------+------+--------+----+
```

Review

- Interactive REPL
- Rapid computation (especially aggregates) on large amounts of data
- High level abstractions for efficient querying of data
- “Condense” data for easier local exploration and visualization