

Large Scale Data Processing with Python and Apache Spark

About

- @MLnick
- Co-founder @graphflow - big data & machine learning applied to recommendations, consumer behaviour & insights
- Apache Spark committer
- Author of “Machine Learning with Spark”
 - *Packt RAW: <http://www.packtpub.com/machine-learning-with-spark/book>*



Agenda

- BIG DATA!
- Herding Elephants with Hadoop
- Introduction to Apache Spark
- PySpark Internals
- Conclusion



TA

D

Cutting through the Hype

- Massive and growing amount of data collected
- Moore's law => cost of compute, disk, RAM decreasing rapidly
- But data still growing faster than single-node performance can handle
- Factors:
 - *ease of collection*
 - *cost of storage*
 - *web*
 - *mobile / wearables*
 - *IoT*
 - *science (CERN, SKA)*

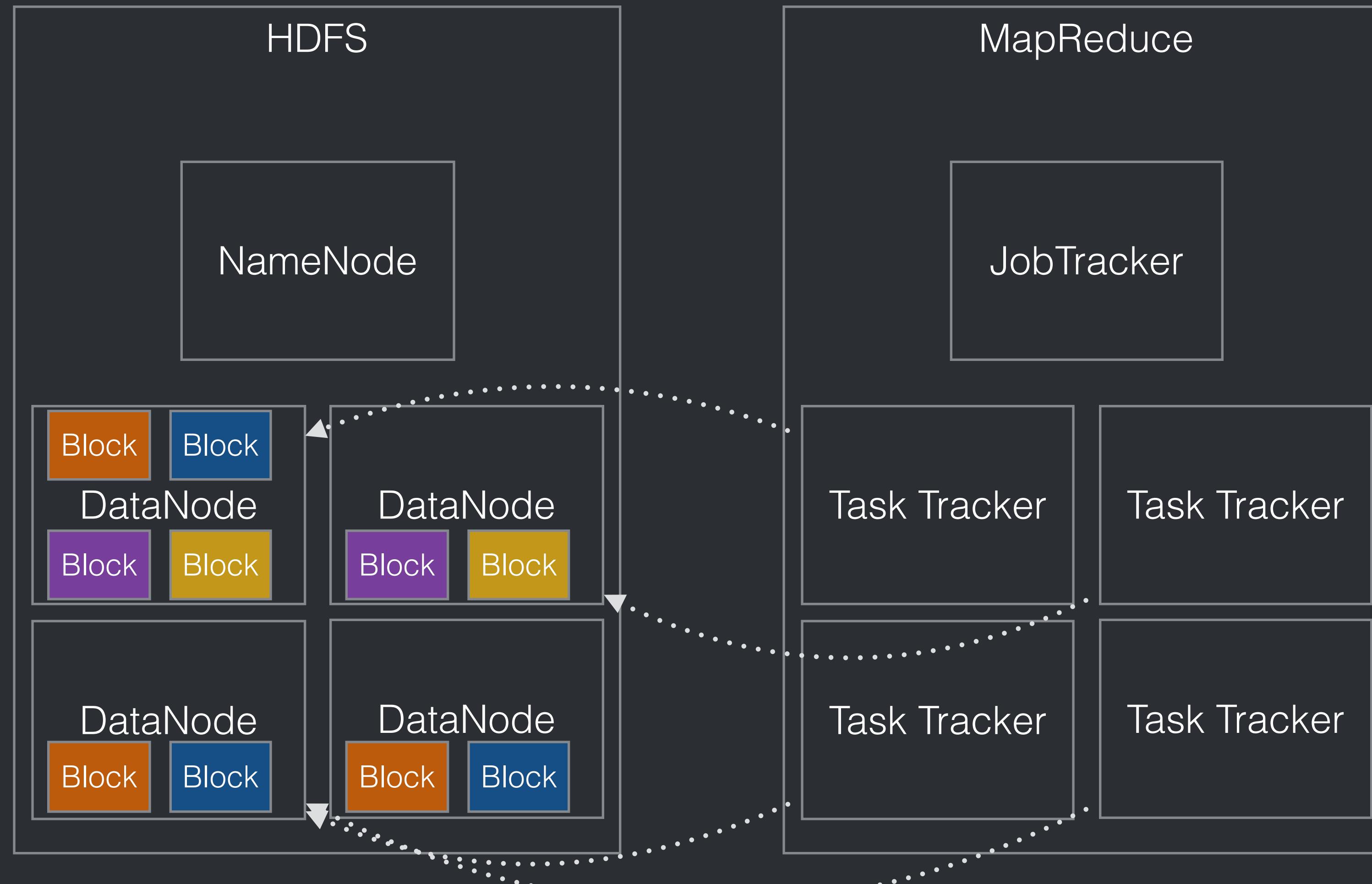
Herdin Elephants - Hadoop

- Google was doing big data before it was cool...
- Google File System Paper (2003) -> Apache Hadoop Distributed Filesystem (HDFS)
- Google MapReduce Paper (2004) -> Apache Hadoop MapReduce
- BigTable (2006) -> Apache HBase
- Dremel (2010) -> Apache Drill (and others)

Herdin Elephants - Hadoop

- Hadoop started at Yahoo
- Open sourced -> Apache Hadoop
- Spawned Cloudera, MapR, Hortonworks... and an entire big data industry
- Old scaling == vertical, big tin
- New scaling == horizontal, shared nothing, data parallel, commodity hardware, embrace failure!

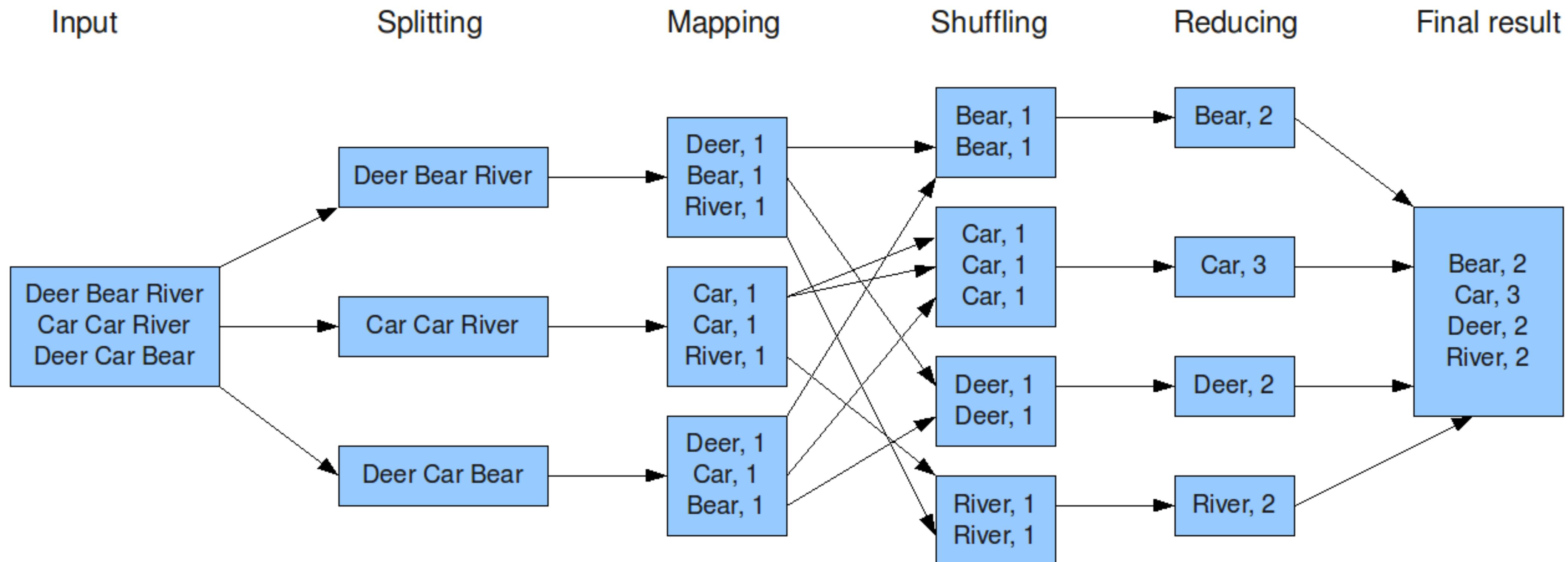
Herdin Elephants - Hadoop



- HDFS
 - *Replication*
 - *Fault tolerance*
- Map Reduce
 - *Data locality*
 - *Fault tolerance*

MapReduce

The overall MapReduce word count process



MapReduce: Counting Words

- “Hadoop is a distributed system for counting words” (Scalding GitHub)
- Map

```
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);  
            }  
        }  
    }  
}
```

MapReduce: Counting Words

- Reduce

```
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));
    }
}
```

MapReduce: Counting Words

- Job Setup

```
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);
```

MapReduce Streaming: Counting Words with Python

- Map

```
# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        #
        # tab-delimited; the trivial word count is 1
        print '%s\t%s' % (word, 1)
```

MapReduce Streaming: Counting Words with Python

- Reduce

```
# input comes from STDIN
for line in sys.stdin:
    ...
    # parse the input we got from mapper.py
    word, count = line.split('\t', 1)
    ...
    # this IF-switch only works because Hadoop sorts map output
    # by key (here: word) before it is passed to the reducer
    if current_word == word:
        current_count += count
    else:
        if current_word:
            # write result to STDOUT
            print '%s\t%s' % (current_word, current_count)
    ...

```

MapReduce Streaming: Counting Words with Python

- Job Setup

```
hadoop jar /.../streaming/hadoop-*streaming*.jar \
    -file /path/to/mapper.py \
    -mapper /path/to/mapper.py \
    -file /path/to/reducer.py \
    -reducer /path/to/reducer.py \
    -input /path/to/input/* \
    -output /path/to/output
```

With thanks to Joseph Misiti:

<https://medium.com/cs-math/a-simple-map-reduce-word-counting-example-using-hadoop-1-0-3-and-python-streaming-1a9e00c7f4b4>

Hadoop Issues

- Pros
 - Reliable in face of failure (*will* happen at scale) - disk, network, node, rack ...
 - Very scalable: ~40,000 nodes at Yahoo!
- Cons
 - Disk I/O for every job
 - Unwieldy API (hence Cascading, Scalding, Crunch, Hadoopy ...)
 - Very hard to debug - especially Streaming jobs 😱

So Why Spark?

- **In-memory** caching == fast!
- Broadcast variables and accumulator primitives built-in
- Resilient Distributed Datasets (RDD) - recomputed on the fly in case of failure
- **Hadoop compatible**
- Rich, functional API in **Scala**, **Python**, **Java** and **R**
- One platform for multiple use cases:
 - Shark / SparkSQL - **SQL** on Spark
 - Spark **Streaming** - Real time processing
 - **Machine Learning** - MLlib
 - **Graph Processing** - GraphX

Spark Word Count

- Power of functional constructs and the Python language

```
file = spark.textFile("hdfs://...")  
counts = file.flatMap(lambda line: line.split(" ")) \  
    .map(lambda word: (word, 1)) \  
    .reduceByKey(lambda a, b: a + b)  
counts.saveAsTextFile("hdfs://...")
```

- Same code locally or on a cluster

```
sparkLocal = SparkContext("local[4]", "Local 4 Threads")  
sparkOnCluster = SparkContext("spark://...:7077",  
                            "Cluster of 1000 Nodes")
```

Functional API

- Functional Python

```
lines = open("somefile").readlines()
words = [(word, 1) for line in lines for word in line.split(" ")]
counts = [(key, len(list(group))) for key, group in
          groupby(sorted(counts, key=lambda x: x[0]), lambda x: x[0])]
```

- Functional PySpark

```
counts = file.flatMap(lambda line: line.split(" ")) \
    .map(lambda word: (word, 1)) \
    .reduceByKey(lambda a, b: a + b)
```

Spark Machine Learning

- Caching data in memory allows subsequent iterations to be faster

```
points = spark.textFile(...).map(parsePoint).cache()
w = numpy.random.rand(D) # initial weight vector
for i in range(100):
    gradient = points.map(lambda p:
        (1 / (1 + exp(-p.y*(w.dot(p.x)))) - 1) * p.y * p.x
    ).reduce(lambda a, b: a + b)
    w -= gradient
```

- Python and NumPy allow concise code, closer to the actual maths!

Plugging in Python Libraries

- Such as scikit-learn

```
from sklearn import linear_model as lm

# init stochastic gradient descent
sgd = lm.SGDClassifier(loss='log')
# training
for i in range(100):
    sgd = sc.parallelize(data, numSlices=slices) \
        .mapPartitions(lambda x: train(x, sgd)) \
        .reduce(lambda x, y: merge(x, y))
sgd = avg_model(sgd, slices) # averaging weight vector
```

SparkSQL

- SparkSQL

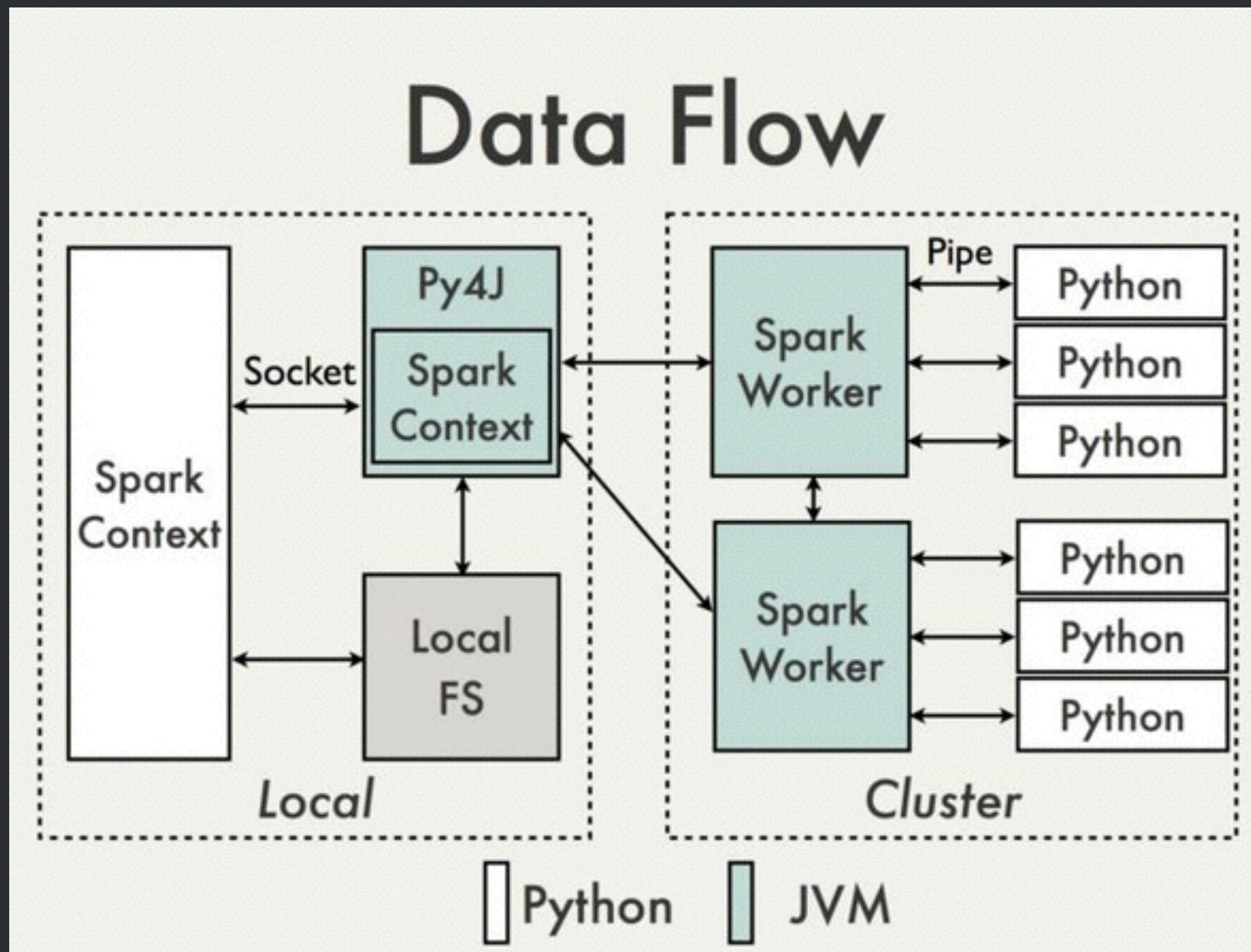
```
events = rdd.map(lambda row: Row(time = row[0], event=row[1], ...))

# complex join and filter in Spark
...

schema = sqlContext.inferSchema(events)
events.registerTempTable('events')
val aggs = sql("select
    from_unixtime(cast(time/1000.0 as bigint), 'yyyy-MM-dd HH:00:00') hour,
    event,
    count(1)
    from events ...")

# aggs is a normal PySpark RDD, with all the normal operations available
aggs.map( ... )
...
```

PySpark Internals



- Python driver
- *Py4J: Python <-> Java*
- *Large data transfer through filesystem rather than socket*
- Workers
 - *Launch Python subprocesses*
 - *Functions pickled and shipped to workers*
 - *Bulk pickling optimizations*
 - *Works in console - IPython FTW!*

PySpark Internals

- Still quite a lot slower than Scala / Java :-(
 - ... but improving all the time
- Not quite feature-parity with Scala / Java ...
 - ... but almost
 - e.g. PySpark Streaming PR: <https://github.com/apache/spark/pull/2538>
- Python 1st class citizen!

Live Demo Time

Apache Spark - The Next Generation of Big Data

- Apache Spark 1.1.0!
- All Hadoop providers announced support / partnerships
Cloudera, MapR, Hortonworks
- Databricks Cloud (<http://databricks.com/cloud/>)
- Try it out: <http://spark.apache.org/>
 - *Local mode on your laptop*
 - *Cluster in Amazon EC2 via Spark launch scripts*



Graphflow

nick@graphflow.com