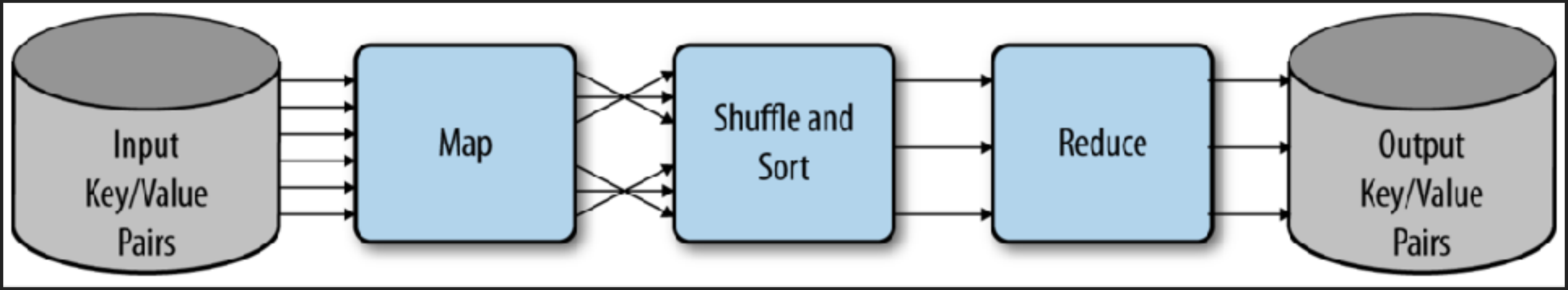
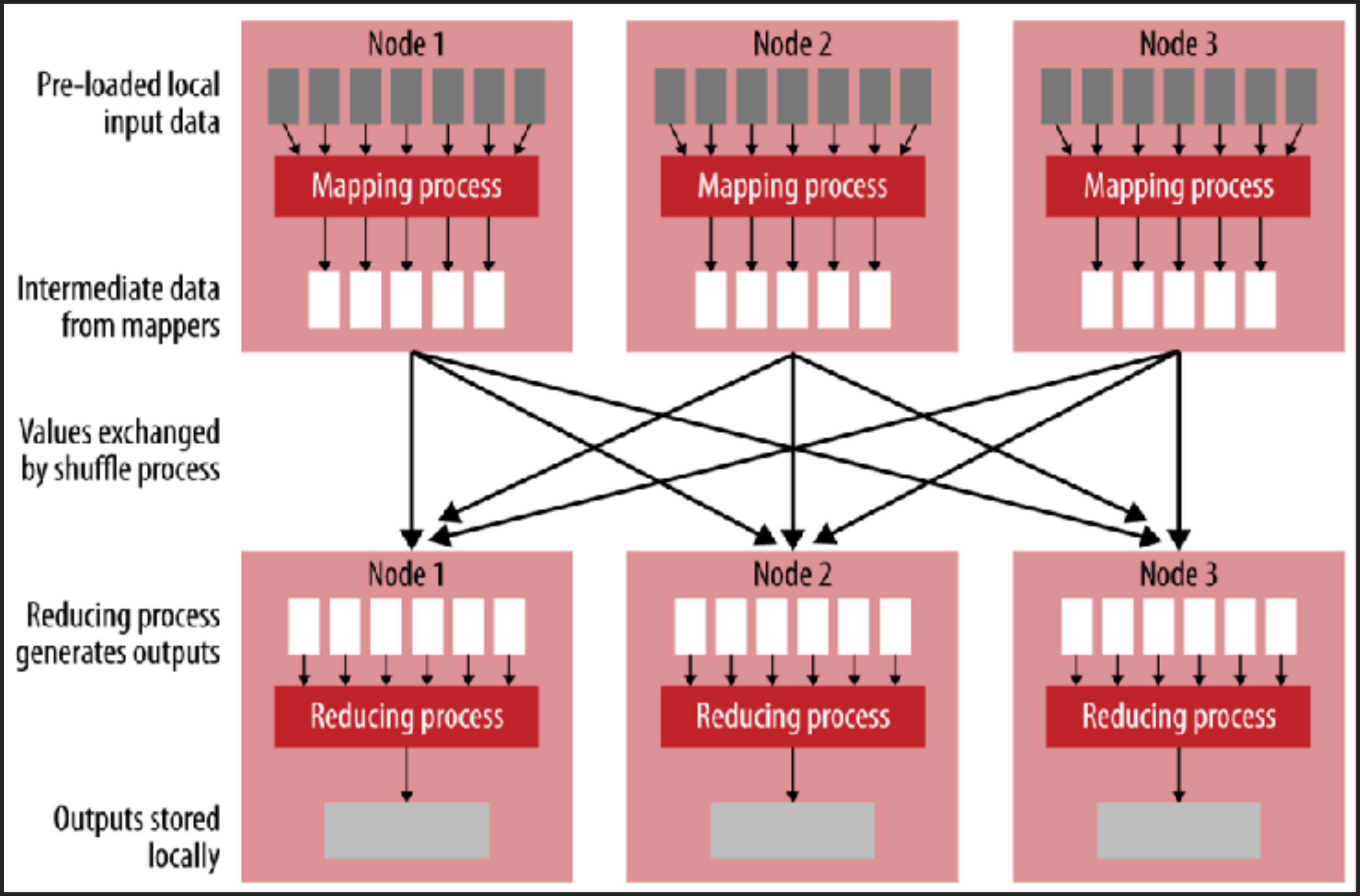
LET'S LOOK AT MAPREDUCE

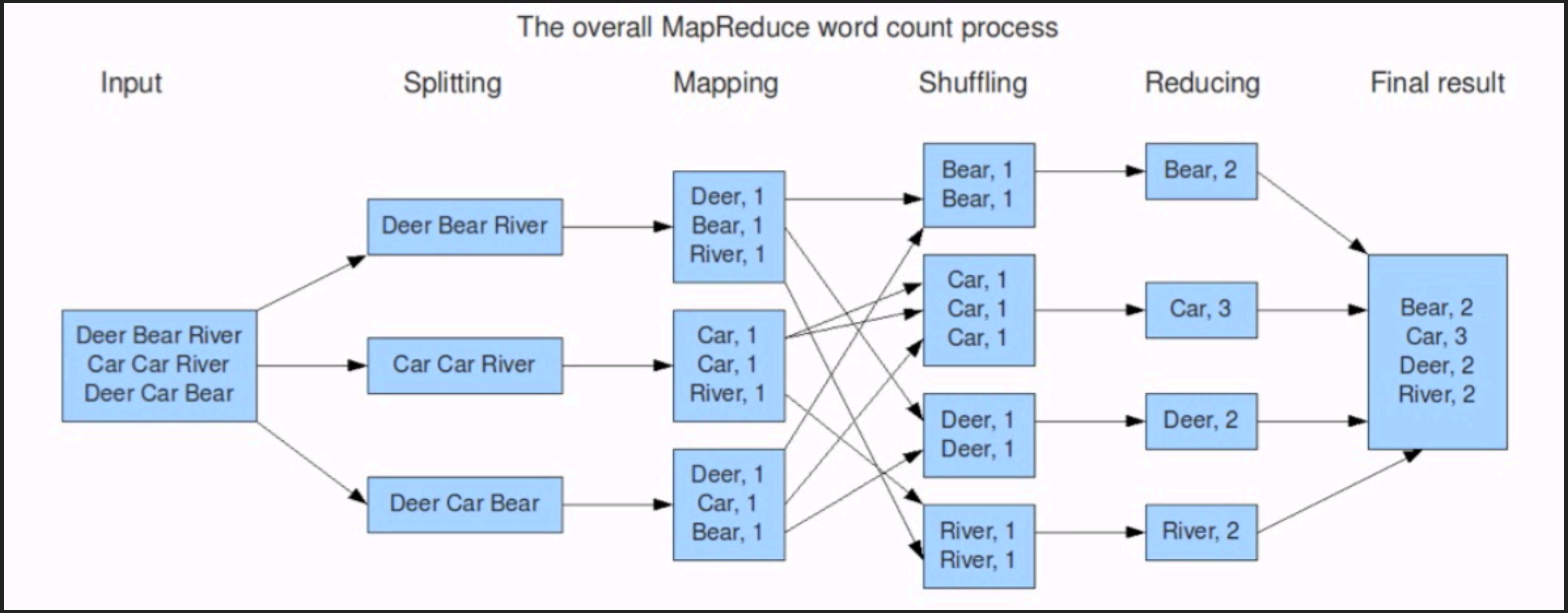


- (book_id, rating) -> count ratings -> (book_id, #ratings)

MAPREDUCE: MAP, SORT & SHUFFLE, REDUCE



MAPREDUCE: WORD-COUNT EXAMPLE



MAPREDUCE

- What are the necessary conditions to be able to use MapReduce?
  - data is row-based
  - manipulation of each row is independent of manipulation of other rows
  - manipulations can be easily combined (reduced)

KEYS

- keys are NOT unique: they are used to combine (reduce) results by combining (reduce) elements with the same key
- contrast to python dictionaries, in which keys are unique

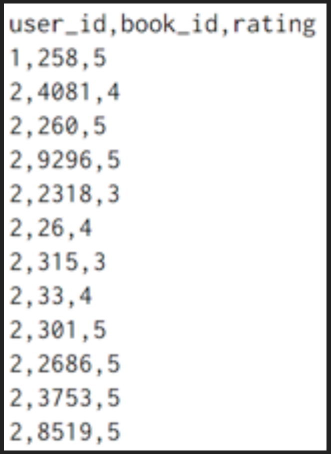
TYPICAL MAPREDUCE WORKFLOW

- read (a lot of?) lines of data
- optionally split the lines into units of analysis (e.g. words)
- map each unit of analysis to a key-value pair (usually a tuple) (e.g. (word, 1))
- reduce the tuples by some operation like addition
- emit (e.g. print) the reduced values

AND THAT LEADS US TO SPARK

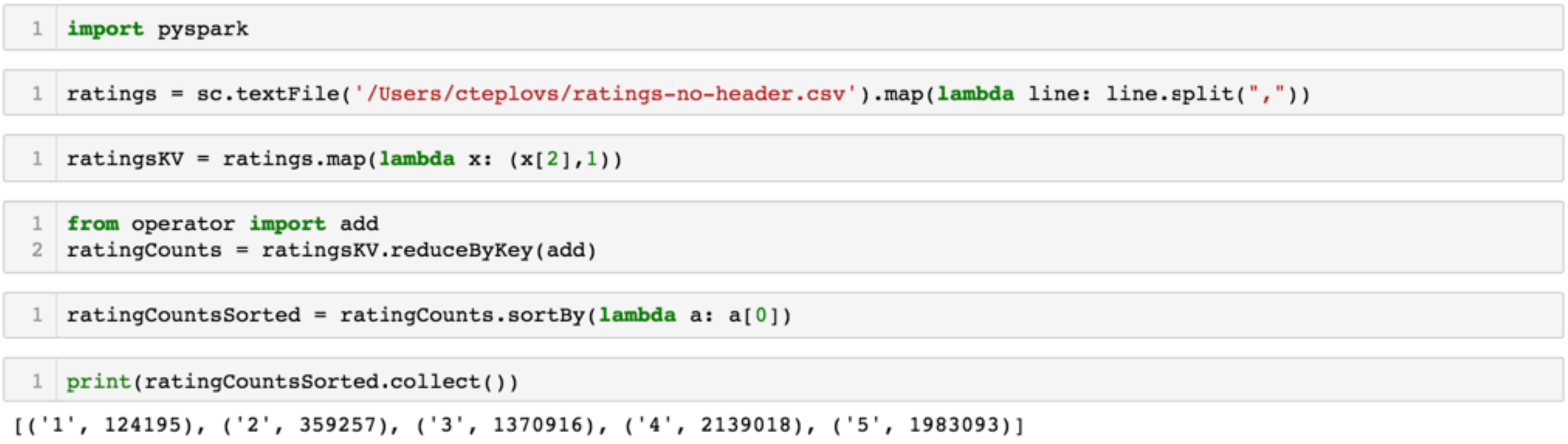
- Spark is "a general-purpose distributed computing abstraction"
- focus on computation (rather than storage)
- provides an interactive shell (PySpark)
- all about parallelization
- very lazy

PYSPARK EXAMPLE



```
user_id,book_id,rating
1,258,5
2,4081,4
2,260,5
2,9296,5
2,2318,3
2,26,4
2,315,3
2,33,4
2,301,5
2,2686,5
2,3753,5
2,8519,5
```

Given a CSV file with user_id, book_id and rating, count the number of all the different ratings (1-5)



```python
import pyspark
```

```python
ratings = sc.textFile('/Users/cteplovs/ratings-no-header.csv').map(lambda line: line.split(","))
```

```python
ratingsKV = ratings.map(lambda x: (x[2],1))
```

```python
from operator import add
ratingCounts = ratingsKV.reduceByKey(add)
```

```python
ratingCountsSorted = ratingCounts.sortBy(lambda a: a[0])
```

```python
print(ratingCountsSorted.collect())
```

```
[('1', 124195), ('2', 359257), ('3', 1370916), ('4', 2139018), ('5', 1983093)]
```

SPARK – FAST AND GENERAL ENGINE FOR LARGE-SCALE DATA PROCESSING

- Up to 100 times faster than Hadoop MapReduce
- Written in Scala, providing Scala, Java and Python APIs
- Supports both batch mode and real-time data stream processing
- Write once, run everywhere
  - Spark runs on Hadoop, Mesos, standalone, or in the cloud. It can access diverse data sources including HDFS, Cassandra, HBase, S3.
- Spark is one of the most active project in the Apache Software Foundation and among Big Data open source projects.
- Not just for Big Data!

Sources: https://spark.apache.org, http://en.wikipedia.org/wiki/Apache_Spark

THE SPARK STACK

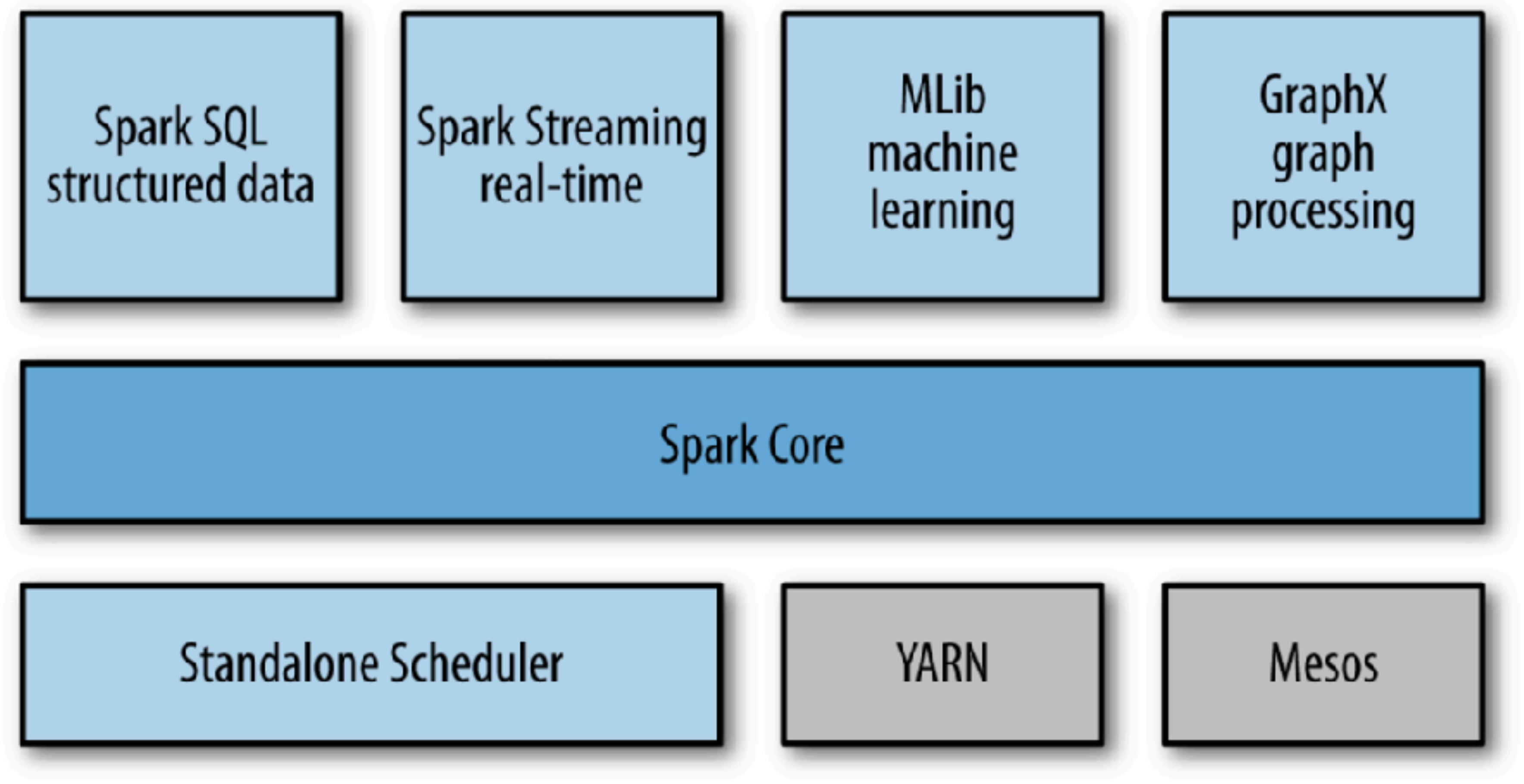


Figure 1-1. The Spark stack

From "Learning Spark"

THE SPARK STACK

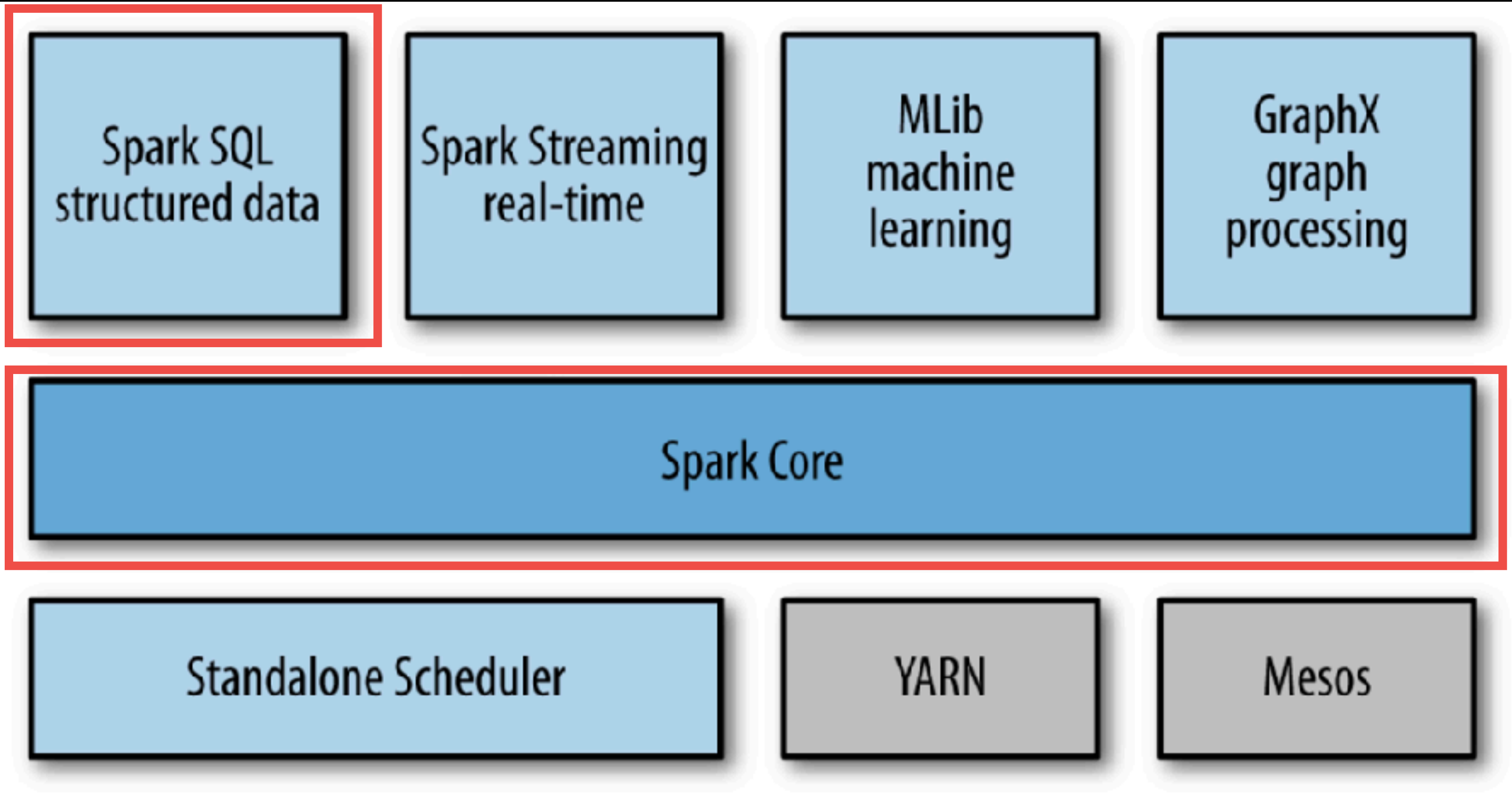


Figure 1-1. The Spark stack

From "Learning Spark"

THE SPARK STACK

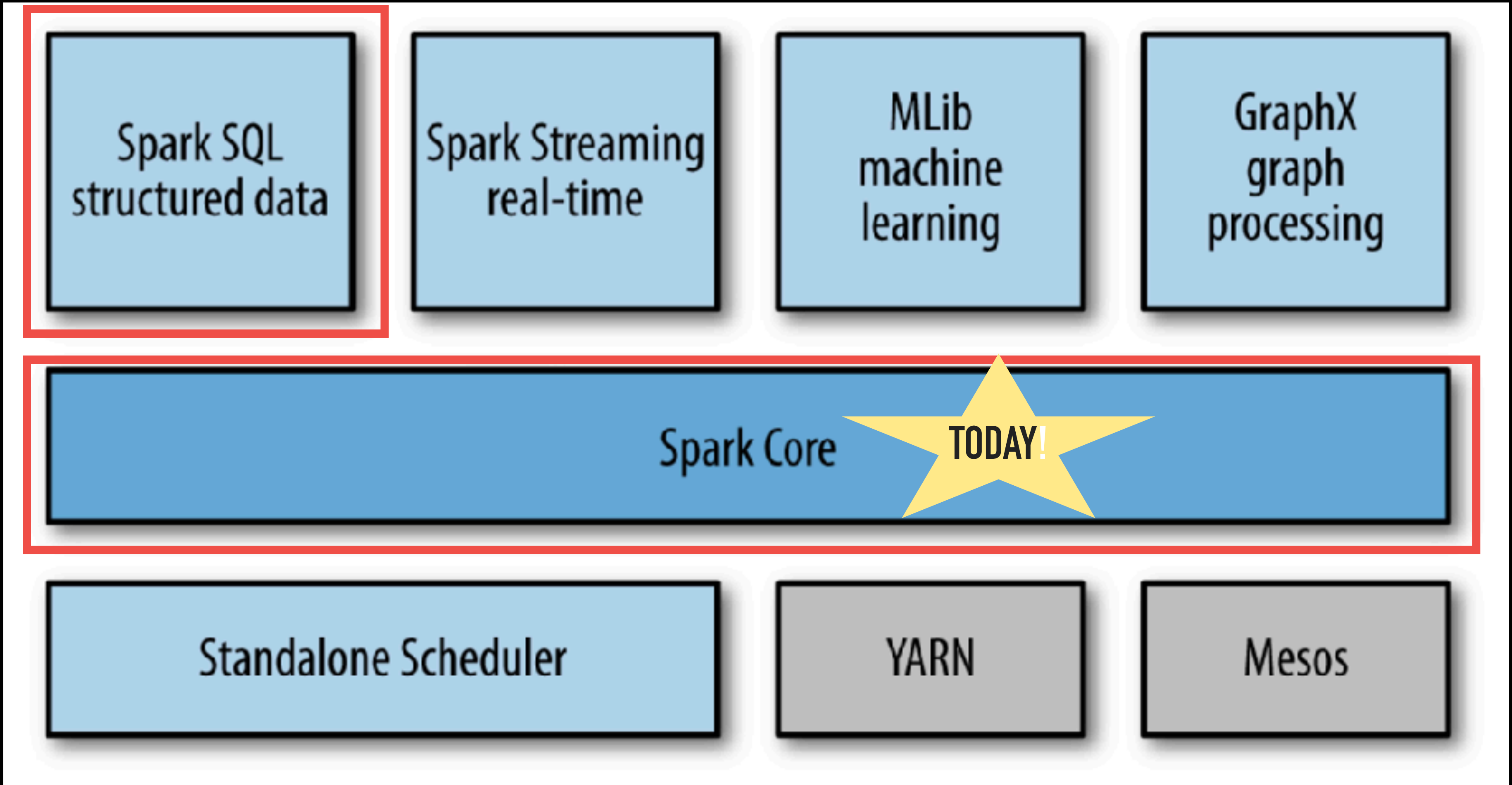


Figure 1-1. The Spark stack

From "Learning Spark"

BASIC SPARK CONCEPTS

- SparkContext (sc)
  - represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster.
- Resilient Distributed Dataset (RDD)
  - RDDs can be created from a number of different data sources: python data structures, local files, pandas DataFrames, distributed file systems
  - RDDs have actions, which return values,
  - and transformations, which return new RDDs
- Two code execution modes:
  - Interactive Python shell: PySpark and Jupyter (and Jupyter-like) notebooks
  - Running standalone applications: spark-submit
    - (we’ll cover spark-submit in a couple of weeks)

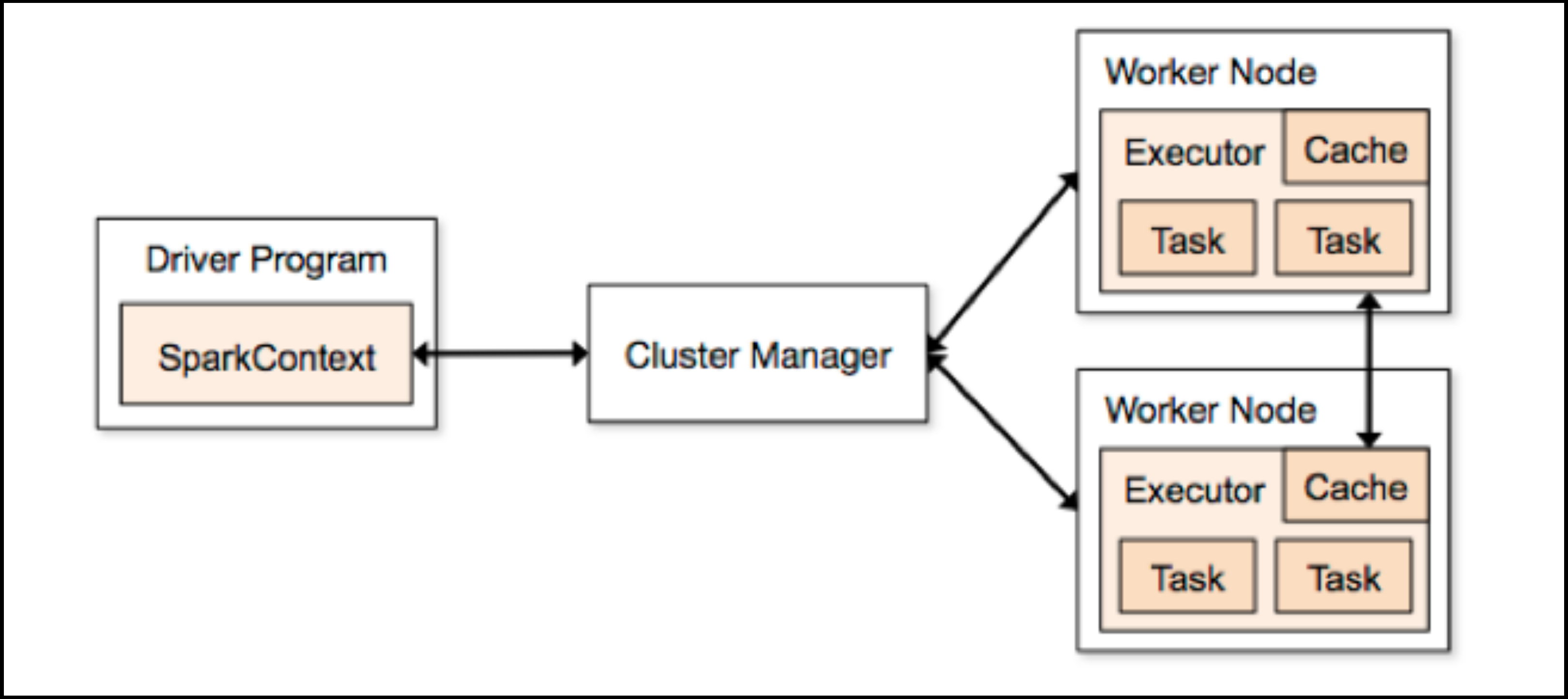
PYSPARK IN JUPYTER

- we will be using Jupyter notebooks on AWS Elastic MapReduce (EMR)

- when your notebook starts, you have access to a special instance variable:
  sc: your SparkContext

SPARKCONTEXT

- Represents a connection to a computing cluster
- In PySpark, a SparkContext is auto-created for you in a variable called `sc`



Source: https://spark.apache.org/docs/1.1.1/cluster-overview.html

RESILIENT DISTRIBUTED DATASET (RDD)

- The fundamental abstraction for distributed data computation
- An RDD is:
  - A immutable (read-only) collection of items
  - Partitioned across computing nodes
  - That can be manipulated in parallel
- Once you have a SparkContext you can create RDDs
- Example: A collection of lines from a text file

```
>>> lines = sc.textFile("input.txt") # Create an RDD called
lines

>>> lines.count() # Count the number of items in this RDD

3

>>> lines.first() # First item in this RDDu'summer school
2012 in indiana'
```

EXAMPLE SPARK API OPERATION ON AN RDD: FILTER

```
>>> lines = sc.textFile("input.txt")
>>> ponyLines = lines.filter(lambda line: "pony"
in line)
>>> ponyLines.first()
u'In general, a pony is more difficult than a
horse'
```

- The magic of Spark: operations like 'filter' are parallelized across the cluster.

IN SPARK, ALL WORK INVOLVES ONE OF THREE KINDS OF OPERATIONS ON RDDS

1. Creating new RDDs

e.g. `lines = sc.textFile("input.txt")`

Input: various. Output: new RDD

2. Transforming existing RDDs

e.g. `pigLines = lines.filter(lambda line: "pig" in line)`

Input: one or more RDDs. Output: new RDD

3. Computing actions on RDDs to get a result

e.g. `pigLines.first()`

Input: one or more RDDs. Output: various non-RDD

CREATION METHOD 1: LOAD DATA FROM EXTERNAL STORAGE

```
lines = sc.textFile('./nyt/1985.html.gz')
```

CREATION METHOD 2: PARALLELIZE AN EXISTING DATA STRUCTURE

```
lines = sc.parallelize(["pandas", "i like pandas"])
```

ELEMENT-WISE TRANSFORMATIONS: MAP AND FILTER

The map() transformation takes in a function and applies it to each element in the RDD with the result of the function being the new value of each element in the resulting RDD.

nums = sc.parallelize([1, 2, 3, 4])

```
squared = nums.map(lambda x: x * x).collect()
for num in squared:
    print(“%i " % (num))
```

The filter() transformation takes in a function and returns an RDD that only has elements that pass the filter() function.

```
def containsError(s):
    return "error" in s
word = rdd.filter(containsError)
```

ELEMENT-WISE TRANSFORMATIONS: FLATMAP

Sometimes we want to produce multiple output elements for each input element. The operation to do this is called flatMap(). As with map(), the function we provide to flatMap() is called individually for each element in our input RDD. Instead of returning a single element, we return an iterator with our return values. Rather than producing an RDD of iterators, we get back an RDD that consists of the elements from all of the iterators.

```
lines = sc.parallelize(["hello world", “hi"])
words = lines.flatMap(lambda line: line.split(" "))
print(words.collect())
```

SET-LIKE TRANSFORMATIONS

```
errorsRDD = inputRDD.filter(lambda x: "error" in x)
warningsRDD = inputRDD.filter(lambda x: "warning" in x)
badLinesRDD = errorsRDD.union(warningsRDD)

rdd1.union(rdd2)
rdd1.intersection(rdd2)
rdd1.subtract(rdd2)
rdd1.distinct()
```

ACTIONS

- `take`(*n*)
  - Take the first n elements of the RDD

```
for line in lines.take(10):
  print(line)
```

- `collect`()

- Take an RDD and turn it into a local list

PAIR RDDS: COLLECTIONS OF (KEY, VALUE) PAIRS

- **`reduceByKey(func)`**
- **`sortByKey()`**
- `groupByKey()`
- `mapValues(func)`
- `flatMapValues(func)`
- `keys()`
- `values()`

PAIR RDDS: SORTBYKEY()

Just like the Python sorted() function: sort (key, value) pairs *alphabetically*

```
word_counts_sorted = word_counts.sortByKey()
```

PAIR RDDS: SORTBY()

Just like the Python sorted() using a key= function

```
word_counts_sorted = word_count3.sortBy(lambda x: x[1],
ascending = False)
```

PAIR RDDS: REDUCEBYKEY(FUNC)

Takes a function that operates on the values of two elements with the same key and returns a new RDD.

```
sumRDD = rdd.reduceByKey(lambda x, y: x + y)
```

Note: the x and y notation is confusing

Think of ‘x’ as an accumulator and ‘y’ as the value to be combined with the accumulator. The accumulator persists across multiple iterations of lambda, the y value is unique to each iteration.

THIS IS DIFFICULT TO GET ONE’S HEAD AROUND!

PAIR RDDS: REDUCE(FUNC)

Takes a function that operates on the values of two elements and returns a new RDD.

```
sumRDD = rdd.reduce(lambda x, y: x + y)
```

Note: the x and y notation is confusing

Think of ‘x’ as an accumulator and ‘y’ as the value to be combined with the accumulator. The accumulator persists across multiple iterations of lambda, the y value is unique to each iteration.

THIS IS DIFFICULT TO GET ONE’S HEAD AROUND!

SPARK WORD COUNT

```
import re
import pyspark

WORD_RE = re.compile(r"\b[\w']+\b")
input_file = sc.textFile("totc.txt")

word_count1 = input_file.flatMap(lambda line: WORD_RE.findall(line))
word_count2 = word_count1.map(lambda word: (word, 1))
word_count3 = word_count2.reduceByKey(lambda a, b: a + b)

word_counts_sorted = word_count3.sortBy(lambda x: x[1], ascending =
False)

top100_sorted = sc.parallelize(word_counts_sorted.take(100))
print(top100_sorted.collect())
string_output = word_counts_sorted.map(lambda t : t[0] + '\t' +
str(t[1]))
print(string_output)
```

SPARK WORD COUNT (SLIGHTLY STREAMLINED)

```
import re
import pyspark

WORD_RE = re.compile(r"\b[\w']+\b")
input_file = sc.textFile("totc.txt")

word_counts_sorted = input_file.flatMap(lambda line:
WORD_RE.findall(line)) \
        .map(lambda word: (word, 1)) \
        .reduceByKey(lambda a, b: a + b) \
        .sortBy(lambda x: x[1], ascending = False)

top100_sorted = sc.parallelize(word_counts_sorted.take(100))
print(top100_sorted.collect())
string_output = word_counts_sorted.map(lambda t : t[0] + '\t' + str(t[1])
print(string_output)
```