Updated: 2021-03-25

5.2
Spark Introduction

Northeastern
University

What is Spark?

- Apache *Spark* is a fast, general-purpose, cluster-computing platform.
  - It abstracts out the map/reduce model* in such a way that a programmer or shell user is simply <u>unaware</u> of it. [This is similar to the way actors abstract out the threading model of Java]
  - It does this in a way that takes more advantage of memory (as opposed to persistent storage). It's also more natural.
  - *Spark* is written in **Scala** and therefore takes full advantage of the functional programming paradigm: providing better performance than competing solutions.
  - *Spark* <u>does not require</u> *Hadoop*, although it is very compatible and is happy to use *YARN* and *HDFS* if they're available.

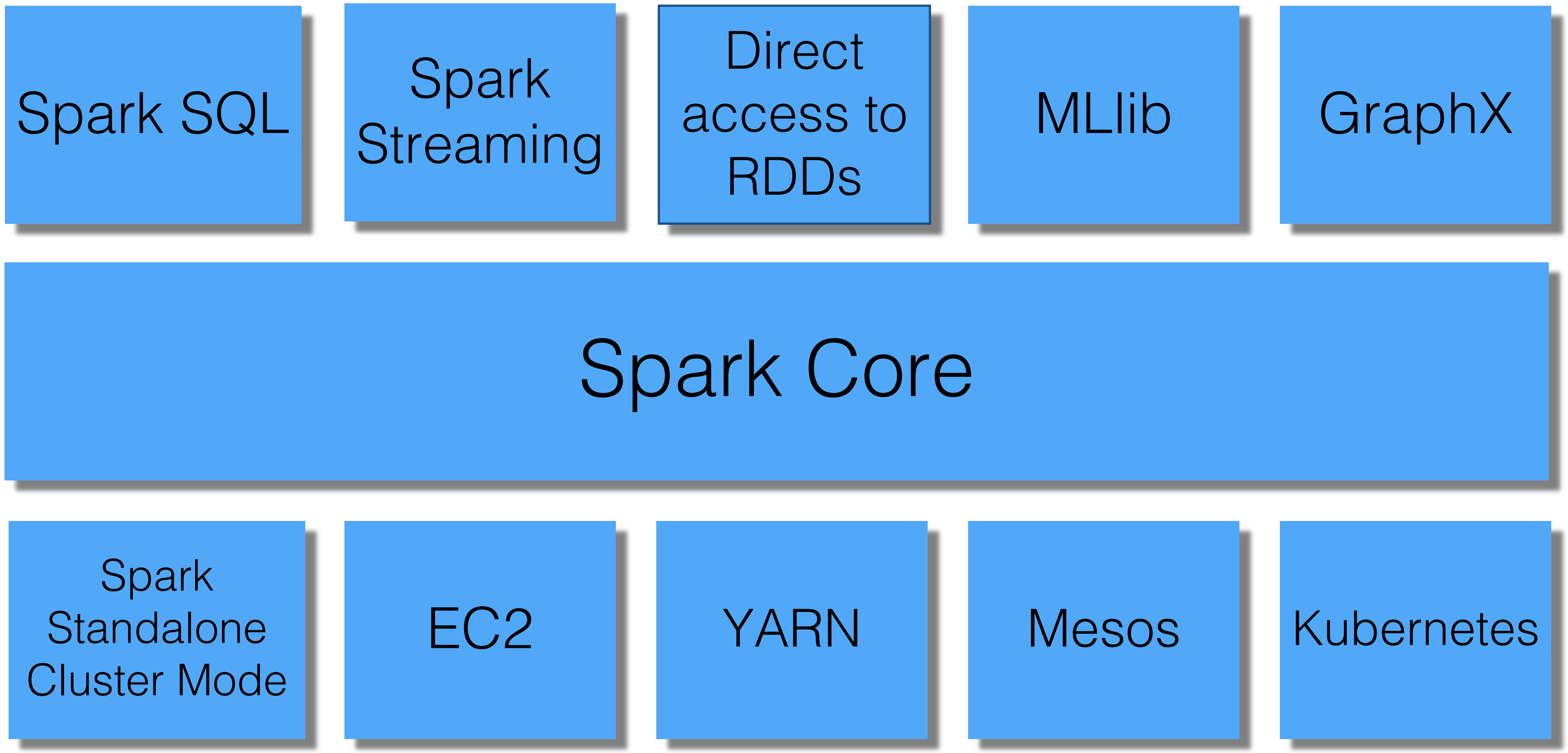
* typified by *Hadoop*

What is Spark? (2)

- *Spark* provides API bindings for *Scala*, *Python*, *R, SQL* and *Java*. So, you don't have to use *Scala*. But by now, surely you'd want to, right? I definitely wouldn't recommend the *Java* API because it can be quite messy to get things right for *Java*.
- Spark documentation can be found here: https://spark.apache.org/docs/latest/. The current latest version is 3.1.1
- **Spark 3.0.1 runs with Java 8/11* and Scala 2.12 :)**
- *Spark* provides a platform ("stack" if you prefer) which includes several components:

* Version 8u92 and onwards

Spark Platform



Books: Learning Spark, Karau et al, (O'Reilly);
Spark in Action (Manning).

Spark components

- You can work directly with Spark's RDDs **but there's really no need to**…
- **Spark-SQL**
  - Essentially a memory-intensive implementation of HIVE:
    - Or you can think of Spark-SQL as an alternative implementation language to *Scala*, *Python*, *Java*, *R*.
    - Spark-SQL in 2.0 and later is highly optimized. It may well be that you should always use Spark-SQL unless you have a good reason not to.
- **Spark Streaming**
  - This is how you would deal with a stream of events, e.g. messaging or processing log files from a web system, database, or message broker like Kafka
- **MLlib**
  - Spark's ML library, including classification, regression, clustering, etc.
- **GraphX**
  - Spark's graph-database (Pregel*-oriented)

* Name of river that flowed in Euler's time through Königsberg

RDDs

(you should still understand these).

- The *Spark* API is founded on a simple, elegant, obvious data type (everything is an RDD):
  - **Resilient Distributed Dataset** (*RDD*)

    ```
    abstract class RDD[T] extends Serializable with Logging
    ```

  - Guess what? It's a monad! So it supports ***map***, ***flatMap***, ***filter***, etc. [it's a container similar to ***Future***, ***Par***, ***Option, Try*** in that it doesn't extend *FilterMonadic*]
  - Like ***Stream***, ***RDD*** is lazy.
  - As far as the user is concerned, an ***RDD*** is just a collection that you can do stuff with. The fact that it may potentially be distributed across thousands of nodes isn't of any immediate concern to you.

Actually, there is no ***README.md*** file available but ***Spark*** doesn't worry about that yet because it's lazy!

```
scala> val lines = sc.textFile("README.md")
lines: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[1] at textFile at <console>:21

scala> val list = List(1,2,3)
list: List[Int] = List(1, 2, 3)

scala> sc.parallelize(list)
res0: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[2] at parallelize at <console>:24
```

Working with RDDs

- You already know how to do this!
  - A couple of details:

```
scala> res0 map ( _ toString )
res1: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[3] at map at <console>:26

scala> println(res1)
MapPartitionsRDD[3] at map at <console>:26

scala> res1 foreach (println)
3
2
1
```

We just pass in a function to do stuff as you would expect. Note, however, that said function must be ***serializable*** since it has to be passed to remote worker nodes. Don't pass in instance methods.

  - Hmm, that's a bit of a surprise! Not really. The method ***sc.parallelize*** by default splits into two slices which are recombined for the final ***println*** step. Specifying one slice, `sc.parallelize(list,1)`, maintains the order. But most of the time, you don't care!
  - If we want to use an *RDD* again, you can persist it:

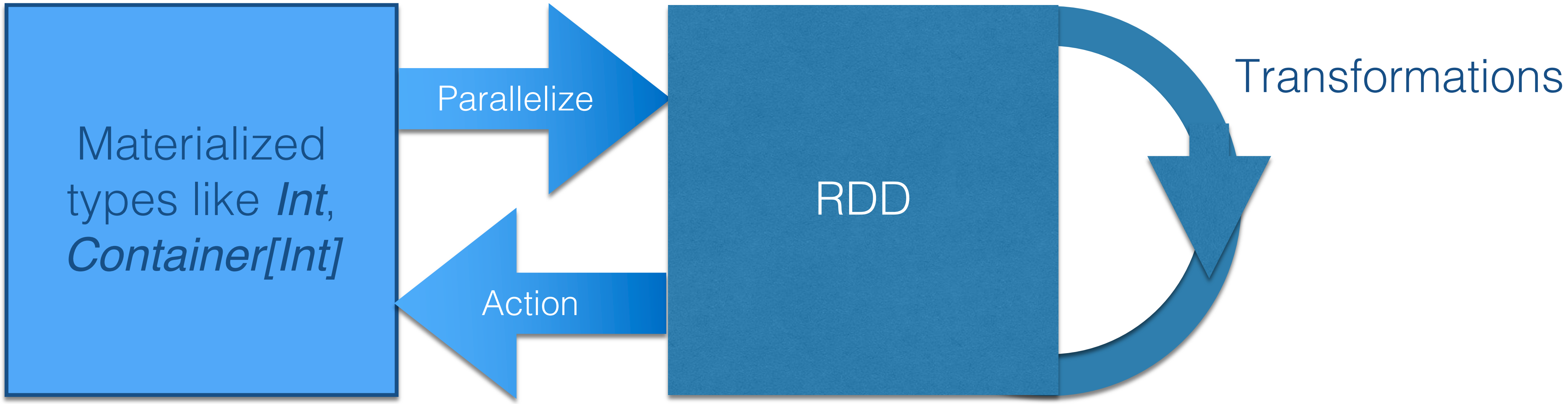
```
scala> res1.persist
res2: res1.type = MapPartitionsRDD[3] at map at <console>:26
```

    - There is also a signature of ***persist*** which allows you to specify the storage level, e.g. `res1.persist(DISK_ONLY)`
    - Note that the ***persist*** call itself doesn't force evaluation.
  - If you just want to get back a collection from an *RDD*, invoke ***collect*** (without a function parameter):

```
scala> res0.collect
res3: Array[Int] = Array(1, 2, 3)
```

Working with RDDs (2)

- Because RDD[T] is lazy, it's essentially ***opaque***



- Once we have parallelized a container as an RDD, we can apply transformations to RDDs, creating new RDDs. Since these of course are lazy, the RDDs are efficiently decorated. In order to materialize something from an RDD, we need to apply an "action"*.*

RDD methods

- What can you do with *RDD*s?
  - Actions
    - Extractions:
      - *collect* (not the same as the *Collection* method we’ve met before—which takes a partial function—think of this if you like as the opposite of *sc.parallelize*)
      - *foreach, saveAsTextFile, saveAsObjectFile,*
    - Aggregations—measure an *RDD* perhaps of an appropriate type:
      - *count, aggregate, max, min, reduce, treeReduce, fold*
  - Transformations
    - single *RDD*s:
      - *map, flatMap, filter, distinct, sample, take, drop, collect(f),*
    - single *RDD*s of an appropriate type:
      - *top, takeOrdered, takeSample, countByValue, groupBy, sortBy*
    - Combinations—two *RDD*s (of same underlying type):
      - *union, intersection, subtract, cartesian, zip*

Pair RDDs

- What are *Pair RDD*s?
  - It shouldn’t come as a big surprise that building RDDs of key/value pairs works well with the map/reduce paradigm. That’s how map/reduce works.
  - Furthermore, you can explicitly control partitioning on such pair RDDs which can give you the location-based performance boost that is a key feature of Hadoop.

```
    scala> val lines = sc.textFile("flatland.txt")
    lines: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[18] at textFile at <console>:21
    scala> val pairs = lines.map(x => (x.split(" ")(0), x))
    pairs: org.apache.spark.rdd.RDD[(String, String)] = MapPartitionsRDD[19] at map at <console>:23
    scala> pairs.groupByKey
    res24: org.apache.spark.rdd.RDD[(String, Iterable[String])] = ShuffledRDD[20] at groupByKey at <console>:26
    scala> res24.collect
    res25: Array[(String, Iterable[String])] = Array((is,CompactBuffer(is impossible that there should be anything of
what)), (luminous,CompactBuffer(luminous edges—and you will then have a pretty)), (readers,,CompactBuffer(readers, who are
privileged to live in Space.)), (their,CompactBuffer(their places, move freely about, on or in the surface,)),
(last,CompactBuffer(last when you have placed your eye exactly on the)), (one,CompactBuffer(one figure from another. Nothing
was visible, nor could be visible,)), (as,CompactBuffer(as I have described them. On the contrary, we could)),
("",CompactBuffer(, , , )), (correct,CompactBuffer(correct notion of my country and countrymen. Alas,)),
(becoming,CompactBuffer(becoming more and more oval to your view, and at)), (Imagine,CompactBuffer(Imagine a...
```

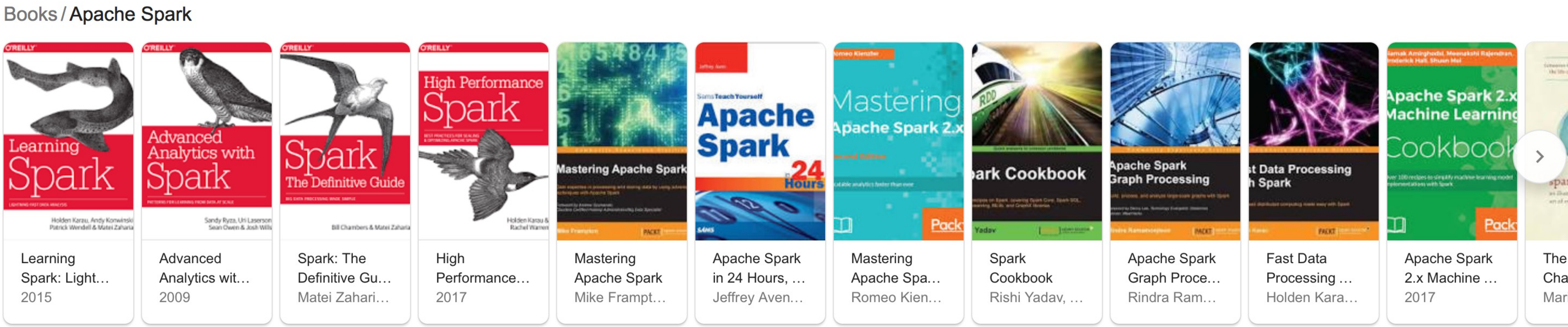
  - Here are the methods that operate on pair RDDs:
    - reduceByKey, groupByKey, combineByKey(…), mapValues(f), flatMapValues(f), keys, values, sortByKey
    - (on two pair RDDs) subtractByKey, join, rightOuterJoin, leftOuterJoin, cogroup
  - Incidentally, you will find that the methods on GraphX are very similar!

Word count by Spark

- Do you remember the code we looked at to do word count by map/reduce?
- Look how complicated it is in Spark (not!):

```
    scala> val words = lines.flatMap(x => x.split(" "))
    words: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[21] at flatMap at <console>:23
    scala> val result = words.map(x => (x,1)).reduceByKey((x, y) => x + y)
    result: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[23] at reduceByKey at
<console>:25
    scala> result.collect
    res26: Array[(String, Int)] = Array((table,1), (circle.,1), (call,3), (paper,1), (country,1),
(is,1), (penny,3), (its,1), (Flatland,,2), (fixed,1), (now,,1), (oval,2), (have,6), (upon,1),
(this,1), (countrymen.,1), (one,2), (mind,1), (with,1), (live,1), (we,3), (straight,2),
(been,1), (dare,1), (us,,1), (who,1), (correct,1), (places,,1), (over,1), (without,1), (my,4),
(rising,1), (exactly,1), (so,,1), (make,1), (instead,1), (what,1), (years,1), (becoming,1),
(are,,1), (other,2), (from,1), (now,1), (has,1), (table,,1), (leaning,1), (happy,1), (vast,1),
(world,1), (contrary,,1), (drawing,1), (demonstrate.,1), (are,1), (kind,,1), (few,1),
(luminous,1), (readers,,1), (because,1), (can,1), (their,1), (moving,1), (country,,1), (down,1),
(anything,1), (remaining,1), (last,1), (will,8), (our,1)...
```

Books



Books / Apache Spark

Learning Spark: Light…
2015

Advanced Analytics wit…
2009

Spark: The Definitive Gu…
Matei Zahari…

High Performance…
2017

Mastering Apache Spark
Mike Frampt…

Apache Spark in 24 Hours, …
Jeffrey Aven…

Mastering Apache Spa…
Romeo Kien…

Spark Cookbook
Rishi Yadav, …

Apache Spark Graph Proce…
Rindra Ram…

Fast Data Processing …
Holden Kara…

Apache Spark 2.x Machine …
2017

The Cha
Mar