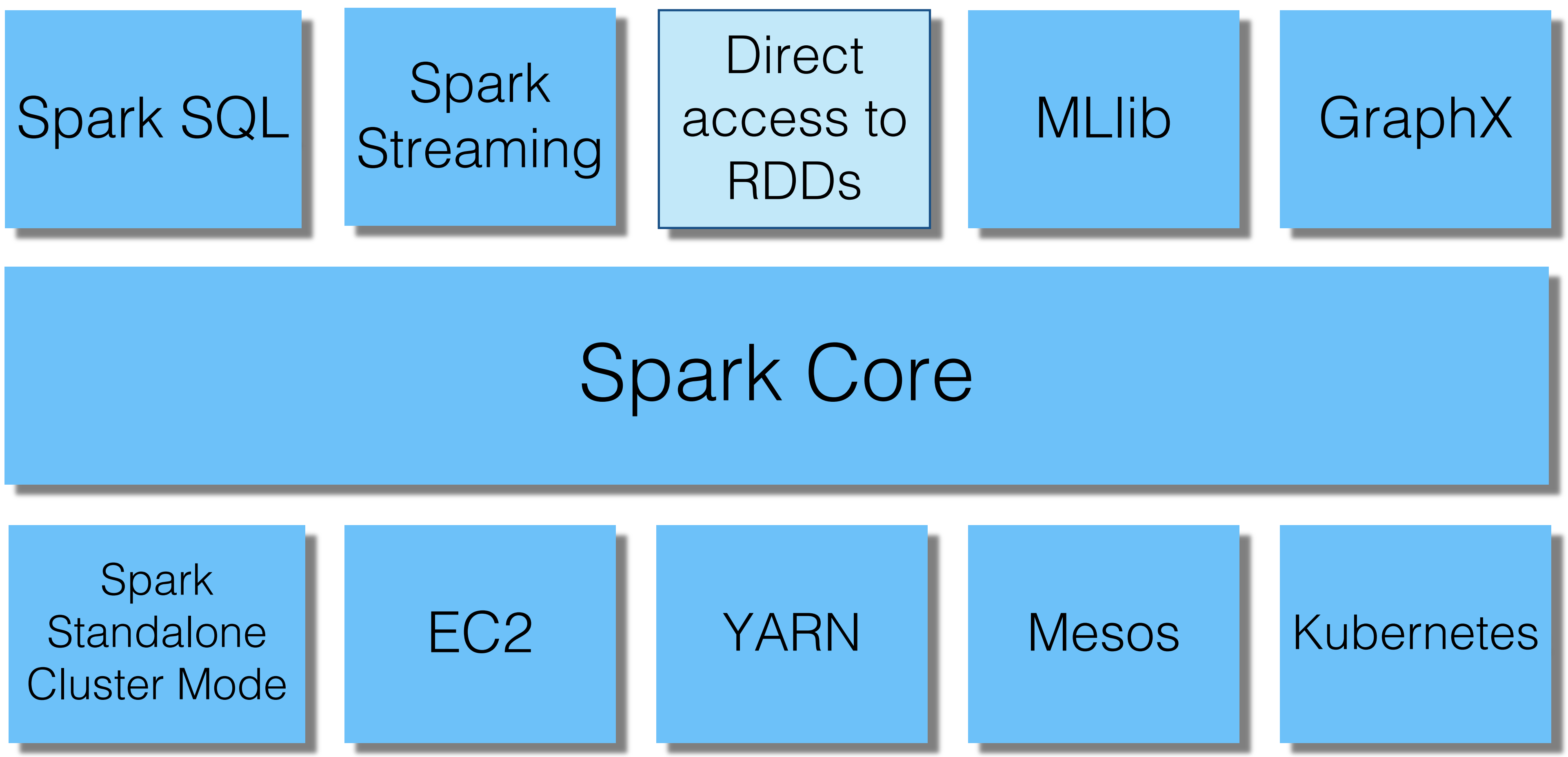
Updated: 2021-03-29

5.3
Spark Continued

Northeastern University

Spark Platform



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

Under the hood

- As mentioned previously, Spark is built in Scala
  - You could easily write your own version of Spark
    - (in fact, that's kind of what ***Majabigwaduce*** is)
    - Basically, Spark is:
      - A "resilient distributed dataset" container (RDD);
      - A DAG (directed-acyclic graph) generator;
      - An interface to a resource manager (Yarn, MESOS, etc.);
      - A higher-level API for working with SQL;
      - A few other bits and pieces.
    - But getting the details right would take a lot of work, obviously — but the point is that Spark is just Scala set up to make parallel processing (map/reduce) easy

Example of using RDD (SparkContext)

```scala
package edu.neu.csye._7200

import org.apache.spark.rdd.RDD
import org.apache.spark.{SparkConf, SparkContext}

object WordCount extends App {

  def wordCount(lines: RDD[String],separator: String) = {
    lines.flatMap(_.split(separator))
         .map((_,1))
         .reduceByKey(_ + _)
  }

  //For Spark 1.0-1.9
  val sc = new SparkContext(new SparkConf().setAppName("WordCount").setMaster("local[*]"))

  wordCount(sc.textFile("input//WordCount.txt")," ").foreach(println(_))

  sc.stop()
}
```

How to invoke Spark?

- There are lots of ways:
  - spark-shell (like we did last week)
  - spark-submit –jar ….
  - Docker…
  - Databricks notebook
  - Zeppelin
  - AWS, etc. know how to run spark.

Example of using RDD from Dataset

(using SparkSession: spark-sql)

```
package edu.neu.csye._7200
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.SparkSession
object WordCount extends App {
  def wordCount(lines: RDD[String],separator: String) = {
    lines.flatMap(_.split(separator))
         .map((_,1))
         .reduceByKey(_ + _)
  }
  val spark = SparkSession
    .builder()
    .appName("WordCount")
    .master("local[*]")
    .getOrCreate()
  wordCount(spark.read.textFile("input//WordCount.txt").rdd,"
").collect().foreach(println(_))
  spark.stop()
}
```

How does RDD work?

- As always if you want to answer a question like this: go to the source!

```
// Transformations (return a new RDD)

  /**
   * Return a new RDD by applying a function to all elements of this RDD.
   */
  def map[U: ClassTag](f: T => U): RDD[U] = withScope {
    val cleanF = sc.clean(f)
    new MapPartitionsRDD[U, T](this, (context, pid, iter) => iter.map(cleanF))
  }
```

withScope ensures that this *RDD* stays within the same hierarchy; *ClassTag* give us information about the class *U* at runtime; *sc.clean* ensures that the function *f* is serializable, etc.

- The point is that map doesn't really "do" anything: it simply creates a new *RDD* with function *f* and a reference to *this*.

Persistence

- Basically:
  - since everything is done in memory, an RDD will be garbage-collected when there are no RDDs referencing it (that’s to say until you create an ***action*** which corresponds to a ***task***).
  - If you want to avoid this: and keep an RDD around for longer, you can use ***cache*** or ***persist***. (***cache*** is just a form of ***persist*** but memory only—***persist*** allows some or all to be save to disk).

Broadcasting

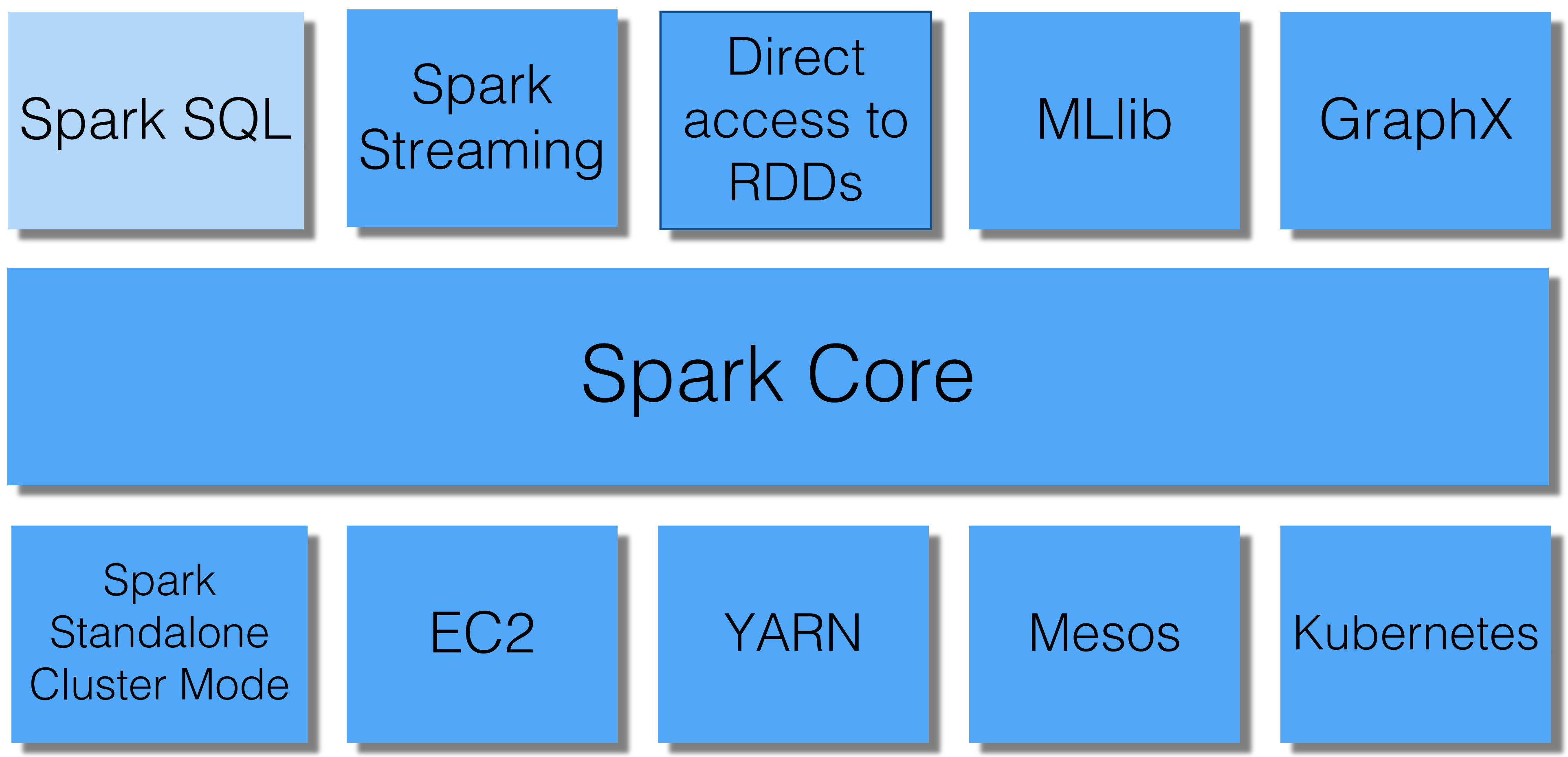
- Suppose that you have a lookup table (or something similar) that will need to be used by each of the executors?
  - The table will have to be sent over the network for every task to be run on each executor.
  - If you know that will happen ahead of time, you can broadcast the table so that it only has to be sent to each executor once.
  - *val xb = sc.broadcast(x)*
  - Now, in our code, we refer to ***xb.value*** instead of ***x***

Accumulators

- Information flows from the driver to the executors mostly in only one direction (other than the result of running a Spark task).
    - But suppose we want to keep count of the number of operations that happen on the executors, or the number of *None* values, *Failure* values, whatever…
        - It would be awkward to include this information in the return type, and that wouldn't really work if the executor threw an exception and failed.
        - The answer is to set up an accumulator: these are write-only objects that you set up in the driver and which are updated by the executors.

Spark Modules

Spark Platform



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

SparkSQL

- What exactly is SparkSQL and why would you want to use it?
  - At first, SparkSQL was fairly primitive and it was better to use RDDs (or Hive).
  - But now (especially in Spark 2.0), SparkSQL has a very good optimizer which will create an execution plan for Spark which is potentially very efficient
  - Spark 2.x (and 1.6.3?) even allows you extend the optimizer with your own rules and node types.
  - Consequently, more and more Spark work is being done not, in Scala, not in Python, Java or R: but in SQL

Datasets/Dataframes

- An alternative to using SQL is to set up a ***Dataset*** (or ***Dataframe*** in 1.6.1) and treat it similarly to an *RDD* (i.e. with Scala)
  - A ***Dataframe*** is untyped (basically a collection of tuples) but a ***Dataset*** has a type:
    - In 2.0, ***type Dataframe = Dataset[Row]***
  - ***Dataframe***/***Dataset*** do not <u>extend</u> *RDD*. But you can get the underlying *RDD* with the *rdd* method.

Spark SQL

- You can run SQL in several ways:
  - get a ***spark*** and make **SQL** queries;
  - get a ***spark*** and use the **DataFrame** or **Dataset** [API](API):
    - *DataFrame*s provide a DSL for structured data manipulation.

```
scala> val df = sqlContext.read.json("examples/src/main/resources/people.json")
df: org.apache.spark.sql.DataFrame = [age: bigint, name: string]

scala> df.show
+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+

scala> df.printSchema
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)

scala> df.select("name").show
+-------+
|   name|
+-------+
|Michael|
|   Andy|
| Justin|
+-------+
```

Joining

- SparkSQL supports various types of Join:
  - INNER, LEFT, OUTER, etc. etc.
  - There will be times when you can improve performance by using a ***broadcast*** join (aka "map" join) — same idea as broadcast variable.
  - However, Spark will do the broadcast for you if it knows the sizes of the tables in advance (e.g. you use a persistence format such as ORC).

Reading CSV as DataFrame

- The standard way to read a CSV file as a *DataFrame* is*:

```
object Diamonds extends App {
  val spark: SparkSession = SparkSession
    .builder()
    .appName("WordCount")
    .master("local[*]")
    .getOrCreate()

  val diamonds = spark.read.format("csv")
         .option("header", "true")
         .option("inferSchema", "true")
         .load("assignment-spark-
wordcount/diamonds.csv")

  diamonds.printSchema()
  diamonds.show()
}
```

* see: https://docs.databricks.com/data/data-sources/read-csv.html

Diamonds

```
root
 |-- _c0: integer (nullable = true)
 |-- carat: double (nullable = true)
 |-- cut: string (nullable = true)
 |-- color: string (nullable = true)
 |-- clarity: string (nullable = true)
 |-- depth: double (nullable = true)
 |-- table: double (nullable = true)
 |-- price: integer (nullable = true)
 |-- x: double (nullable = true)
 |-- y: double (nullable = true)
 |-- z: double (nullable = true)

+---+-----+---------+-----+-------+-----+-----+-----+----+----+----+
|_c0|carat|      cut|color|clarity|depth|table|price|   x|   y|   z|
+---+-----+---------+-----+-------+-----+-----+-----+----+----+----+
|  1| 0.23|    Ideal|    E|    SI2| 61.5| 55.0|  326|3.95|3.98|2.43|
|  2| 0.21|  Premium|    E|    SI1| 59.8| 61.0|  326|3.89|3.84|2.31|
|  3| 0.23|     Good|    E|    VS1| 56.9| 65.0|  327|4.05|4.07|2.31|
|  4| 0.29|  Premium|    I|    VS2| 62.4| 58.0|  334| 4.2|4.23|2.63|
|  5| 0.31|     Good|    J|    SI2| 63.3| 58.0|  335|4.34|4.35|2.75|
|  6| 0.24|Very Good|    J|   VVS2| 62.8| 57.0|  336|3.94|3.96|2.48|
|  7| 0.24|Very Good|    I|   VVS1| 62.3| 57.0|  336|3.95|3.98|2.47|
|  8| 0.26|Very Good|    H|    SI1| 61.9| 55.0|  337|4.07|4.11|2.53|
|  9| 0.22|     Fair|    E|    VS2| 65.1| 61.0|  337|3.87|3.78|2.49|
| 10| 0.23|Very Good|    H|    VS1| 59.4| 61.0|  338| 4.0|4.05|2.39|
| 11|  0.3|     Good|    J|    SI1| 64.0| 55.0|  339|4.25|4.28|2.73|
| 12| 0.23|    Ideal|    J|    VS1| 62.8| 56.0|  340|3.93| 3.9|2.46|
| 13| 0.22|  Premium|    F|    SI1| 60.4| 61.0|  342|3.88|3.84|2.33|
| 14| 0.31|    Ideal|    J|    SI2| 62.2| 54.0|  344|4.35|4.37|2.71|
| 15|  0.2|  Premium|    E|    SI2| 60.2| 62.0|  345|3.79|3.75|2.27|
| 16| 0.32|  Premium|    E|     I1| 60.9| 58.0|  345|4.38|4.42|2.68|
| 17|  0.3|    Ideal|    I|    SI2| 62.0| 54.0|  348|4.31|4.34|2.68|
| 18|  0.3|     Good|    J|    SI1| 63.4| 54.0|  351|4.23|4.29| 2.7|
| 19|  0.3|     Good|    J|    SI1| 63.8| 56.0|  351|4.23|4.26|2.71|
| 20|  0.3|Very Good|    J|    SI1| 62.7| 59.0|  351|4.21|4.27|2.66|
+---+-----+---------+-----+-------+-----+-----+-----+----+----+----+
only showing top 20 rows
```

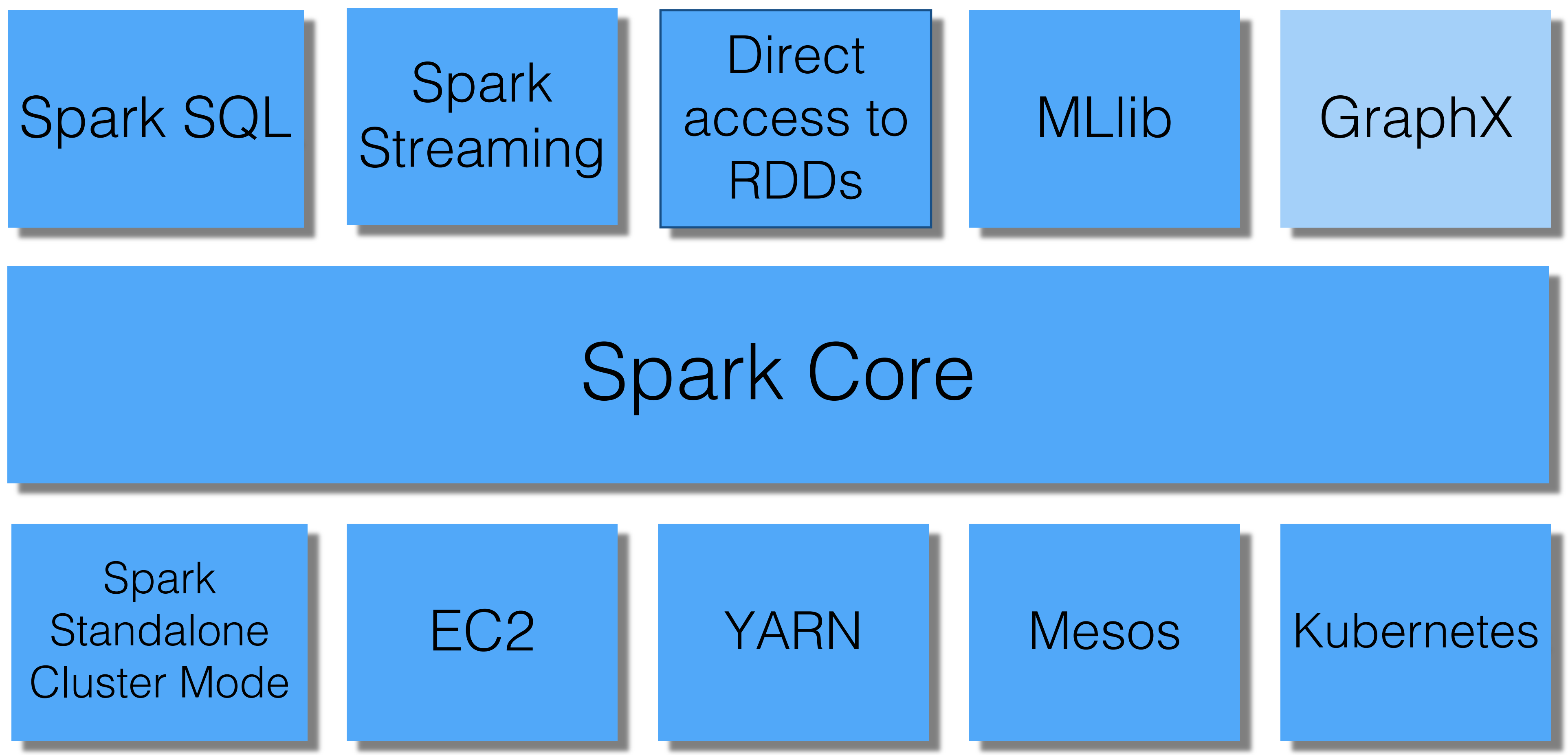
Reading CSV as DataSet

- There is no standard way to read a CSV file as a *DataSet.*
- You could use my *TableParser* library:
  - https://github.com/rchillyard/TableParser
  - https://scalaprof.blogspot.com/2019/04/new-projects.html

```
import MovieParser._
val mty: Try[Table[Movie]] = Table.parse("movies.csv")
val dy: Try[Seq[Movie]] = mty map (spark.createDataset(_.toSeq))
```

- This is the "proper" way to deal with a CSV file in Spark.

Spark Platform



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

Why is GraphX interesting?

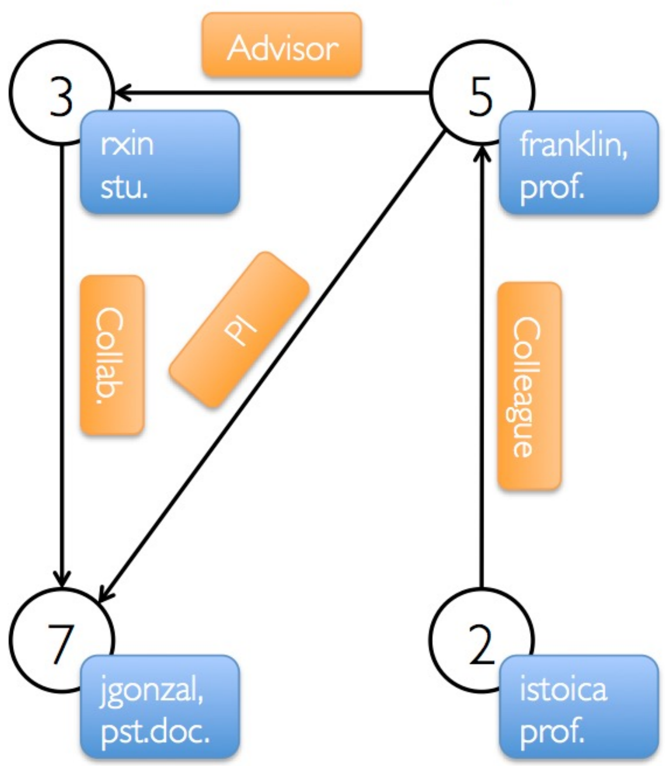
- *GraphX* is interesting because…
  - Graphs are interesting on their own.
    - Graphs represent *relationships*—and relationships are an important type of information.
    - Over the years, we have been seduced into thinking that tables are the most important way of modeling data (relational databases)
    - Actually, before relational databases, we had so-called codasyl databases which could represent graphs.
    - Graphs can store data whose structure is totally arbitrary and dynamic: trees, sparse matrices, key-value stores, tables, etc. (you might not always *want* to use a graph of course, but you could)
  - *GraphX* extends the *Spark* infrastructure (based on linear, but segmentable datasets—*RDDs*) and patterns to graph information.

Some GraphX resources

- GraphX Programming Guide
- GraphX: Graph Analytics in Spark- Ankur Dave (UC Berkeley)

An example
(from programming guide)

Property Graph



Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

An example

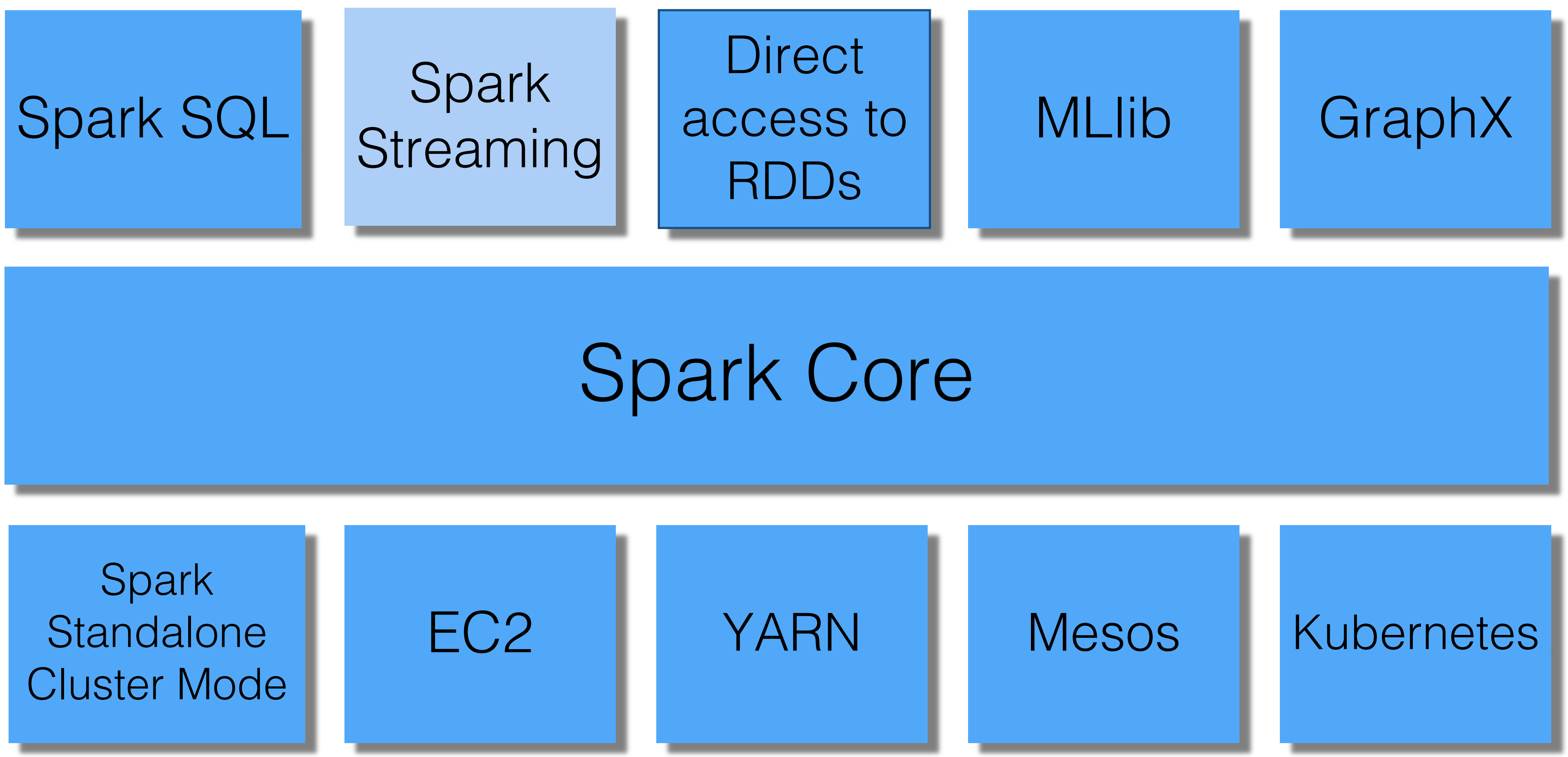
Note that *VertexId* is a type alias for *Long.*

```
case class Edge[ED](srcId: VertexId = 0, dstId: VertexId = 0,
attr: ED = null.asInstanceOf[ED]) extends Serializable with
Product
```

```
scala> import org.apache.spark._
import org.apache.spark._
scala> import org.apache.spark.graphx._
import org.apache.spark.graphx._
scala> import org.apache.spark.rdd.RDD
import org.apache.spark.rdd.RDD
scala> val users: RDD[(VertexId, (String, String))] =
     |   sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
     |                        (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))
users: org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId, (String, String))] = ParallelCollectionRDD[0] at
parallelize at <console>:29
scala> val relationships: RDD[Edge[String]] =
     |   sc.parallelize(Array(Edge(3L, 7L, "collab"),    Edge(5L, 3L, "advisor"),
     |                        Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))
relationships: org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[String]] = ParallelCollectionRDD[1] at
parallelize at <console>:29
scala> val defaultUser = ("John Doe", "Missing")
defaultUser: (String, String) = (John Doe,Missing)
scala> val graph = Graph(users, relationships, defaultUser)
graph: org.apache.spark.graphx.Graph[(String, String),String] = org.apache.spark.graphx.impl.GraphImpl@35ca1e22
scala> graph.triplets.collect
res1: Array[org.apache.spark.graphx.EdgeTriplet[(String, String),String]] =
Array(((3,(rxin,student)),(7,(jgonzal,postdoc)),collab), ((5,(franklin,prof)),(3,(rxin,student)),advisor),
((2,(istoica,prof)),(5,(franklin,prof)),colleague), ((5,(franklin,prof)),(7,(jgonzal,postdoc)),pi))
scala> graph.edges
res2: org.apache.spark.graphx.EdgeRDD[String] = EdgeRDDImpl[13] at RDD at EdgeRDD.scala:40
scala> res2.reverse
res3: org.apache.spark.graphx.EdgeRDD[String] = EdgeRDDImpl[20] at RDD at EdgeRDD.scala:40
```

Note that *EdgeRDD[X]* is an abstract class which extends *RDD[Edge[X]]* , with *reverse, mapValues* and *innerJoin*

Spark Platform



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

Spark Streaming

- Spark Streaming…
  - is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams;
  - data can be ingested from many sources like Kafka, etc.
  - finally, processed data can be pushed out to filesystems, databases, and live dashboards.
  - *in fact, you can apply Spark's machine learning and graph processing algorithms on data streams.*

Spark Streaming (contd.)

- Internally, it works as follows:
  - Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches.