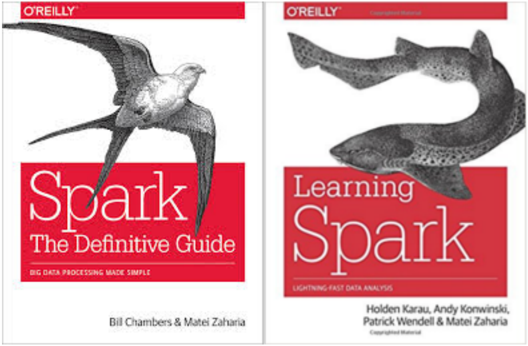
Spark

Marco Milanesio

MS Data Science 2019-2020

Overview

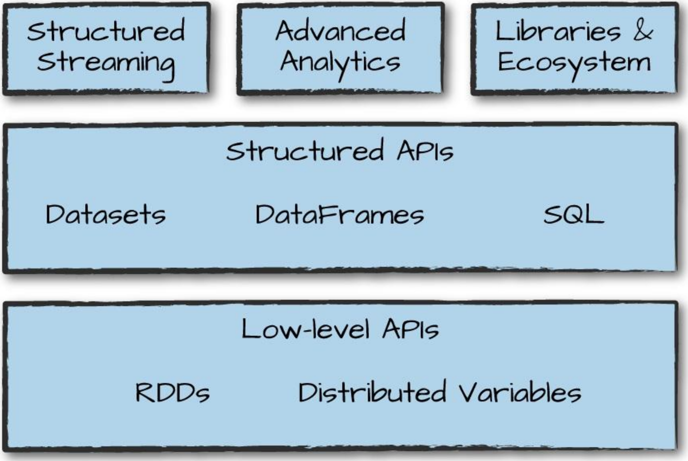
References


O'REILLY
Spark
The Definitive Guide
BIG DATA PROCESSING MADE SIMPLE
Bill Chambers & Matei Zaharia
O'REILLY
Learning
Spark
LIGHTNING-FAST DATA ANALYSIS
Holden Karau, Andy Konwinski,
Patrick Wendell & Matei Zaharia

What is Spark?

- a unified computing engine
- a set of libraries for parallel data processing on computer clusters
- support for (almost) all languages
  - Python, Java, Scala, and R
- libraries for diverse tasks
  - from SQL to streaming and machine learning
- runs everywhere
  - from a laptop to a cluster of thousands of servers

Spark's toolkit



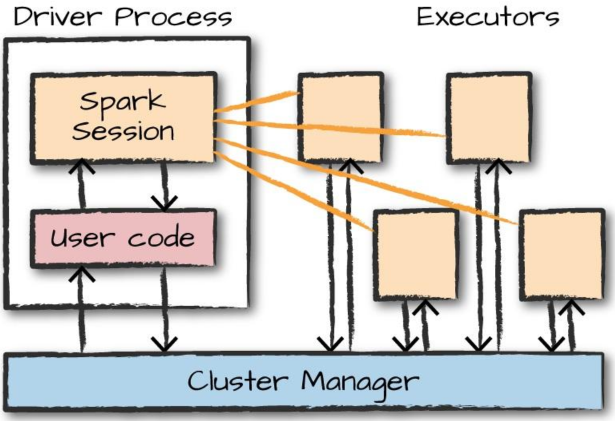
Core data structures

- RDD
  - Resilient Distributed Dataset. Like a distributed collection. Lazily evaluated. Handles faults by recompute. All data types.
- Dataframe
  - NOT Pandas Dataframe. Distributed. Limited set of operations. Columnar structured, runtime schema information only. Limited data types.
- Dataset
  - Compile time typed version of a Dataframe. Templated.

Spark application architecture

- Driver process
  - Coordinator
  - SparkSession (> 2.0)
- Executors
  - They do the job !!
- Cluster manager
  - Apache Mesos
  - Hadoop Yarn
  - Local

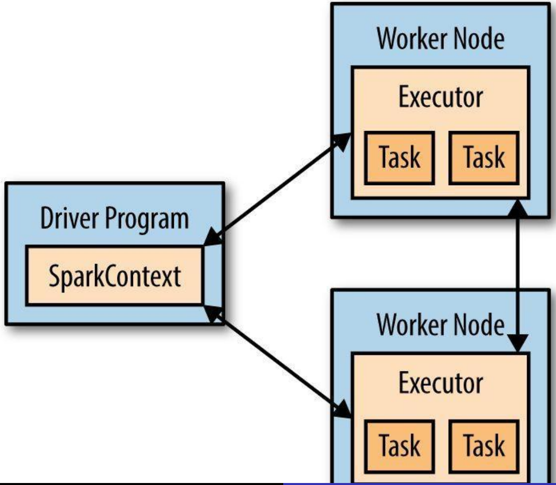
Spark application architecture



Live CODING

- Your first Spark application
  - SparkSession vs SparkContext
    - $<$ 2.0: `sc = SparkContext()`
    - $>$ 2.0: `sc = spark.sparkContext`
  - DataFrame
- Your second Spark application
  - lines count
  - RDD

What did just happened?



RDD

- Resilient Distributed Datastore
- **Immutable**
- Split into multiple partitions
- Any type / user defined objects
- Creation:
  - loading an external dataset
  - distributing a collection in the driver program

RDD Operations

- Two operations are available:
  - **Transformations**
    - construct a new RDD from a previous one
  - **Actions**
    - compute a result based on an RDD
- **Lazy evaluation**
  - Evaluate **transformations** as soon as they are used in an **action**

Why lazy evaluation?

- Allows pipelining procedures
- Less passes over the data
- Can skip materializing intermediate results
- Figuring out where the code fails becomes a little trickier.

Word Count (of course)

```
lines = sc.textFile(src)
words = lines.flatMap(lambda x: x.split(" "))

word_count =
   (words.map(lambda x: (x, 1))
         .reduceByKey(lambda x,y: x + y))

word_count.saveAsTextFile(output)
```

Lazy evaluation

```
lines = sc.textFile(src)
words = lines.flatMap(lambda x: x.split(" "))

word_count =
   (words.map(lambda x: (x, 1))
         .reduceByKey(lambda x,y: x + y))
```

No data is read or processed until this line

```
word_count.saveAsTextFile(output)
```

Common transformations and actions

transformations	actions
map	count
filter	reduce
flatMap	collect
join	take
cogroup	saveAsTextFile
reduceByKey	saveAsHadoop
	countByValue

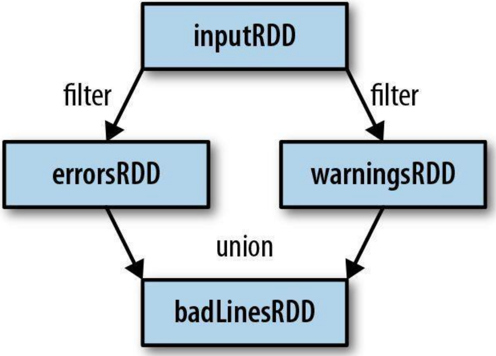
Transformations

- return a new RDD
- **lazily** evaluated

```
inputRDD = sc.textFile("log.txt")
errorsRDD = inputRDD.filter(lambda x: "error" in x)
```

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

Transformations



Actions

- return a final value to the driver
- save results to disk
- **eagerly** evaluated

```
badLinesCount = badLines.count()
for line in badLines.take(10):
    print(line)
```

Did you notice anything strange?

```
badLinesCount = badLines.count()
for line in badLines.take(10):
    print(line)
```

- we read the data twice!! **BAD!**
- `cache` and `persist` to the rescue!
- let's check in the Spark UI

Transformations

`rdd = {1,2,3,3}`

function	purpose	example	result
`map()`	Apply a function to each element of the RDD and returns an RDD of the result.	`rdd.map(lambda x: x+1)`	`{2,3,4,4}`
`flatMap()`	Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned.	`rdd.flatMap(lambda x: range(1,x))`	`{1, 1, 2, 1, 2}`
`filter()`	Return a RDD consisting of only elements that pass the condition passed to filter()	`rdd.filter(lambda x: x != 1)`	`{2,3,3}`
`distinct()`	Removes duplicates.	`rdd.distinct()`	`{1,2,3}`

Transformations

rdd = {1,2,3}; other = {3,4,5}

function	purpose	example	result
`union()`	Produce an RDD containing elements from both RDDs.	`rdd.union(other)`	`{1,2,3,3,4,5}`
`intersection()`	RDD containing only elements found in both RDDs.	`rdd.intersection(other)`	`{3}`
`subtract()`	Remove the contents of one RDD.	`rdd.subtract(other)`	`{1,2}`
`cartesian()`	Cartesian product with the other RDD.	`rdd.cartesian(other)`	`{(1,3),(2,4),(3,5)}`

Actions

rdd = {1,2,3,3}

function	purpose	example	result
`collect()`	Return all the elements of the RDD.	`rdd.collect()`	`{1,2,3,3}`
`count()`	Counts the elements of the RDD.	`rdd.count()`	`4`
`countByValue()`	Number of time each element occurs in the RDD.	`rdd.countByValue()`	`{(1,1),(2,1),(3,2)}`
`take(num)`	Return num elements.	`rdd.take(2)`	`{1,2}`
`top(num)`	Return the top num elements of the RDD.	`rdd.top(2)`	`{3,3}`
`reduce(func)`	Combine elements of the RDD in parallel.	`rdd.reduce(lambda x,y: x+y)`	`9`

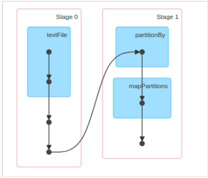
The DAG

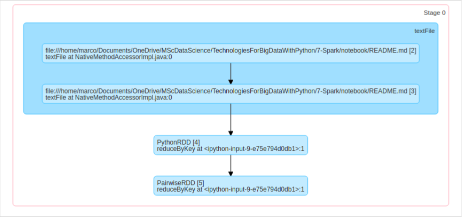
- RDDs/Dataframes: Magical Distributed Collections
- DAG/Query plan is the root of (almost) all of it
- Optimizer to combine the steps
- Resiliency: recover (not protect) from failures
- In-memory + spill-to-disk
- Functional programming to have the DAG “for free”
- Select operation without deserialization

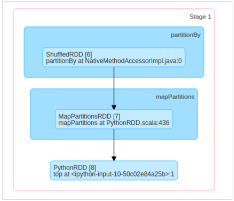
The DAG

- In Spark most of the work is done by transformation (e.g., `map()`)
- Transformation return new RDDs (or Dataframes) representing the data
- The RDD (or the Dataframe) doesn't really exist (!!!)
- They are **plans** of how to make the data show up if we force Spark's hand.
- The data doesn't exist until it **has** to!

DAG & Query plan







The DAG

- Pipelining (can put `map()`, `filter()`, `flatMap()` together
- Can do optimization by delaying work
- Used to recompute on failure
- Alas:
  - Doesn't have a whole program view (just up to the "action")
  - Combining transformations together makes it hard to know what failed
  - It can only see the pieces it understands (two maps, but can't tell what each map is doing)

Pair RDD

- key/values distributed collections
- act on each key in parallel
- regroup data across the network
- we already saw one

```
word_count = (words.map(lambda x: (x, 1))
                    .reduceByKey(lambda x,y: x + y))
```

Common transformations and actions

transformations	actions
reduceByKey(func)	countByKey()
groupByKey()	collectAsMap()
mapValues(func)	lookup(key)
flatMapValues(func)	. . .
keys()	
values()	
sortByKey()	