Big Data

Lecture 3 – Introduction to Apache Spark

Gianluca Quercini

gianluca.quercini@centralesupelec.fr

Centrale DigitalLab, 2021

CENTRALE MARSEILLE

What you will learn

In this lecture you will learn:

- What **Spark** is and its main features.
- The components of the **Spark stack**.
- The high-level **Spark architecture**.
- The notion of **Resilient Distributed Dataset** (RDD).
- The main **transformations** and **actions** on RDDs.

Apache Spark

Definition (Apache Spark)

Apache Spark is a *distributed computing framework* designed to be *fast* and *general-purpose*. ▸ Source

Main features

- **Speed**. Run computations in **memory**, as opposed to Hadoop that heavily relies on disks and HDFS.
- **General-purpose**. It integrates a wide range of workloads that previously required separate distributed systems.
  - Batch applications, iterative algorithms.
  - Interactive queries, streaming applications.
- **Accessibility.** It offers APIs in Python, Scala, Java and SQL and rich built-in libraries.
- **Integration.** It integrates with other Big Data tools, such as Hadoop.

Spark stack

Spark core

- A **computational engine** responsible for **scheduling**, **distributing**, and **monitoring** applications.
  - Spark applications consist of computational **tasks** distributed in a cluster.
- Provides the API that defines **Resilient Distributed Datasets** (RDDs), Spark's main programming abstraction.

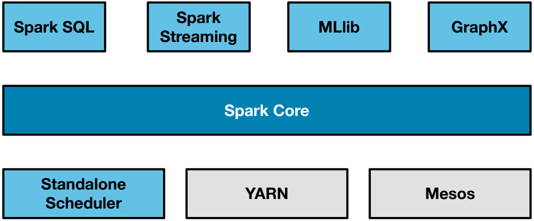


Image source

Spark stack

Spark SQL

- Spark's package for working with **(semi-)structured data**.
- Supports SQL, the Hive Query Language and many data sources:
  - Hive tables, Parquet, JSON
- Allows developers to use a combination of SQL queries and programmatic data manipulations in Python, Java or Scala.

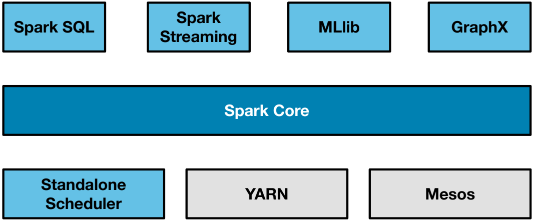


Image source

Spark stack

Spark streaming

- Spark's package for processing live **streams** of data.
- **Example.** Logfiles generated by Web servers, status updates in social network platforms . . .

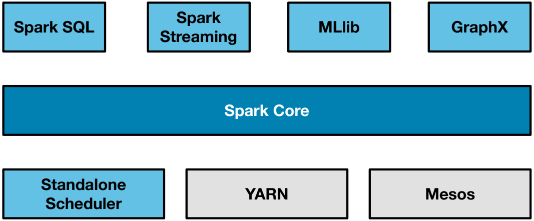


Image source

Spark stack

MLlib

- Spark's package providing numerous **machine learning** algorithms.
  - Classification, regression, clustering, model evaluation and data import.
- The ML algorithms **scale out** across a cluster.

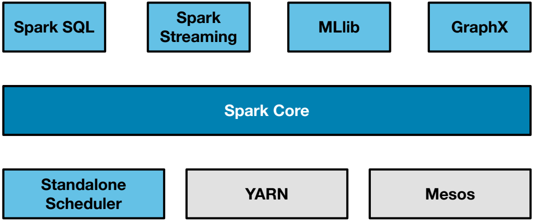


Image source

Spark stack

GraphX

- Spark's library for manipulating **graphs**.
- Extends the Spark RDD API to allow the representation of **directed graphs**.
- Provides the implementation of common graph algorithms (e.g., PageRank).

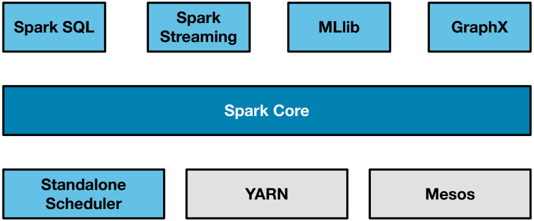


Image source

Spark stack

Cluster managers

- A **cluster manager** is a component that allocate resources across applications.
- Spark provides its own **standalone** cluster manager.
- Spark can be used on other cluster managers, such as Yarn and Mesos.

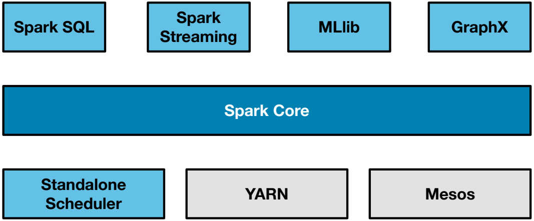


Image source

Spark stack: benefits

- **Improvements** on the bottom layers are automatically reflected on high-level libraries.
  - Optimizations in the Spark core result in better performances in Spark SQL and MLlib.
- **Remove the costs** of using **different independent systems**.
  - Deployment, maintenance, test, support of different systems (streaming, SQL, machine learning. . .).
- **Different programming models** in the same application.
  - Application that reads a stream of data.
  - Applies machine learning algorithms.
  - Uses SQL to analyze the results.

Application context of Spark

Data science tasks

- **Spark shell.** Interactive data analysis with Python or Scala.
- **Spark SQL shell.** Interactive data analysis with a SQL(-like) language.
- **Machine learning**, with the possibility of plugging into Matlab and R.
- Support for large datasets.

Data processing tasks

- **Transparent parallelization** of applications across a cluster.
- **Local tests** of applications.
- **Modular API**, allowing for the development of reusable libraries.

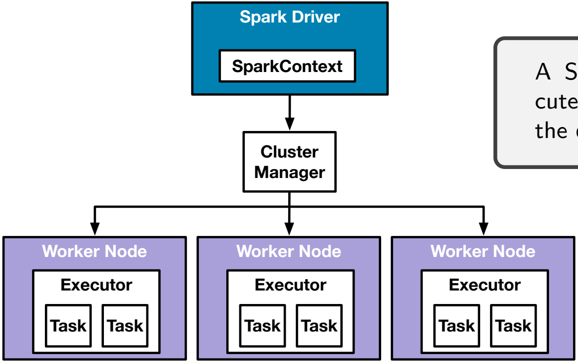
Who uses Spark

- **Amazon**.
- **eBay**. Log transaction aggregation and analytics.
- **Groupon**.
- **Stanford DAWN**. Research project aiming at democratizing AI.
- **TripAdvisor**.
- **Yahoo!**

☞ Full list available at http://spark.apache.org/powered-by.html

Spark architecture

- **Master/slave** architecture: one coordinator (the **Driver**) and many distributed workers, called **executors**.
- The driver and the executors are separate **Java processes**.
- **Spark application.** Driver + executors.



A Spark application is executed in the cluster through the **cluster manager**.

Image source

The Driver

- The **driver** is the process that runs the user program code.

Converting a user program into tasks

- From the user program code, Spark creates a **directed acyclic graph** (DAG) of operations.
- The driver converts this graph into a set of **stages**, each stage consisting of multiple **tasks**.
- Each task is sent for execution on a machine of the cluster.

Scheduling tasks on executors

- The driver coordinates the scheduling of individual tasks on executors.
- When executors are started, they register with the driver.
- Tasks are scheduled on an executor based on **data placement**.
- The drivers exposes information about the running Spark application through a Web interface.

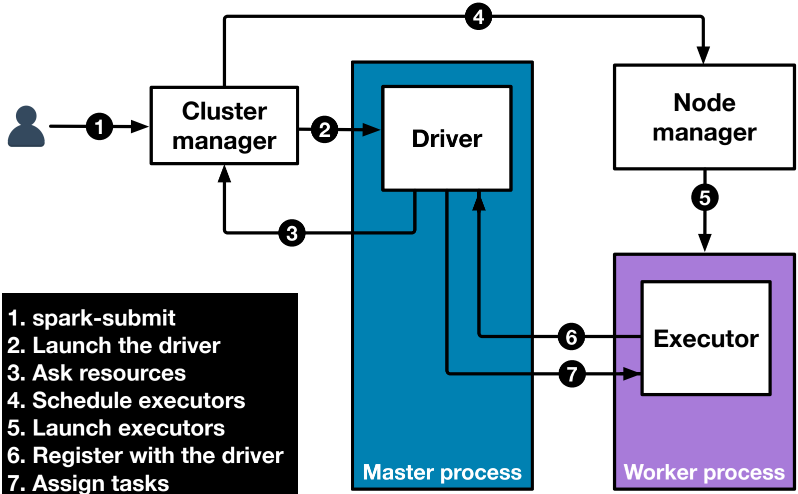
Executors

- **Executors** are processes responsible for running the individual tasks of a Spark application.
- They return the results to the driver.
- They provide **in-memory storage** for the RDDs that are cached by the user program.

☞ When Spark is executed in **local mode** (i.e., on the **local** machine), the Spark driver runs along with an executor in the same Java process.

☞ The driver and the executors are started by the cluster manager.

Launching a Spark application



1. spark-submit
2. Launch the driver
3. Ask resources
4. Schedule executors
5. Launch executors
6. Register with the driver
7. Assign tasks

Writing a Spark program

- The program accesses Spark through an object called **SparkContext**.
- **SparkContext** represents a connection to a cluster.

Initializing the SparkContext

```
from pyspark import SparkCon  f, SparkContext

conf = SparkConf().setMaster(<cluster URL>).setAppName(<app_name>)
sc = SparkContext(conf = conf)
```

- A Spark program is a sequence of operations invoked on the **SparkContext**.

- These operations manipulate a special type of data structure, called **Resilient Distributed Dataset (RDD)**.

Resilient Distributed Dataset (RDD)

Definition (Resilient Distributed Dataset)

A **Resilient Distributed Dataset**, or simply **RDD**, is an **immutable**, **distributed** collection of objects. ▸ Source

- Spark splits each RDD into multiple **partitions**.
- Partitions are distributed *transparently* across the nodes of the cluster.
- Spark **parallelizes** the operations invoked on each RDD.

☞ A Spark program is a sequence of operations invoked on RDDs. An operation can be either a **transformation** or an **action**.

Creating RDDs

Parallelize a collection

- The **SparkContext** is used to parallelize an existing collection.

```
wordList = ["This", "is", "my", "first", "RDD"]
words = sc.parallelize(wordList)
```

- It assumes that the collection be in entirely in memory.
- Used for prototyping and testing.

Load data from external storage

- The **SparkContext** offers numerous functions to load data from external sources (e.g., a text file).

```
lines = sc.textFile(<path_to_file>)
```

RDD operations

Transformations

Transformations are operations that take in one or several RDDs and return a new RDD.

Actions

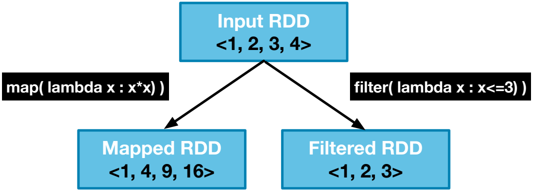
Actions are operations that take in one or several RDDs and return a result to the driver or write it to storage.

☞ A transformation **never changes** the input RDD. A **new RDD** is created instead.

Common transformations

Element-wise transformations

- `map()`. Takes in a **function** f and a RDD $< x_i \mid 0 \leq i \leq n >$; Returns a new RDD $< f(x_i) \mid 0 \leq i \leq n >$.
- `filter()`. Takes in a **predicate** p and a RDD $< x_i \mid 0 \leq i \leq n >$; Returns a new RDD $< x_i \mid 0 \leq i \leq n,\ p(x_i)\ \textit{is true} >$



Common transformations

Element-wise transformations

- `map()`. Takes in a **function** f and a RDD $< x_i \mid 0 \leq i \leq n >$; Returns a new RDD $< f(x_i) \mid 0 \leq i \leq n >$.
- `filter()`. Takes in a **predicate** p and a RDD $< x_i \mid 0 \leq i \leq n >$; Returns a new RDD $< x_i \mid 0 \leq i \leq n,\ p(x_i) \textit{ is true} >$

Map

```
nums = sc.parallelize([1, 2, 3, 4])
mapped_rdd = nums.map(lambda x: x*x)
```

Filter

```
filtered_rdd = nums.filter(lambda x: x <= 3)
```

Common transformations

Element-wise transformations

- `map()`. Takes in a **function** f and a RDD $< x_i \mid 0 \leq i \leq n >$; Returns a new RDD $< f(x_i) \mid 0 \leq i \leq n >$.
- `filter()`. Takes in a **predicate** p and a RDD $< x_i \mid 0 \leq i \leq n >$; Returns a new RDD $< x_i \mid 0 \leq i \leq n,\ p(x_i)\ \textit{is true} >$

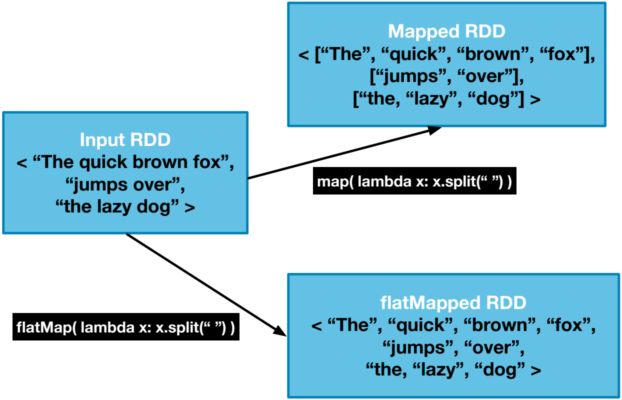
Map (alternative)

```
def power2(x):
    return x*x
nums = sc.parallelize([1, 2, 3, 4])
mapped_rdd = nums.map(power2)
```

Common transformations

`flatMap`

Similarly to `map`, takes in a function *f* and an RDD and applies the function element-wise. *f* must return a **list of values**.



Common transformations

`flatMap`

Similarly to `map`, takes in a function f and an RDD and applies the function element-wise. f must return a **list of values**.

flatMap

```
phrase=sc.parallelize(
    ["The quick brown fox", "jumps over", "the lazy dog"])
flat_mapped_rdd = phrase.flatMap( lambda x: x.split(" ") )
```

Common transformations

Pseudo set operations

- `r1.union(r2)`. Returns a **new RDD** with the elements that occur in RDD *r*1 or *r*2.
- `r1.intersect(r2)`. Returns a **new RDD** with the elements that occur in both RDDs *r*1 and *r*2.
- `r1.subtract(r2)`. Returns a **new RDD** with the elements that occur RDD *r*1, but not in *r*2.
- `r1.distinct()`. Returns a **new RDD** with the elements of the RDD *r*1 without duplicates.
- `r1.cartesian(r2)`. Returns a **new RDD** with the Cartesian product of the RDDs *r*1 and *r*2.

Common actions

Reduce

Given a RDD r, takes in a function f that takes in two elements of r and returns a new element of the **same type**.

Example. Sum the elements of a RDD

```
numbers = sc.parallelize([1, 2, 3, 4])
sum = numbers.reduce( lambda x, y: x+y )
```

Collect

- Returns the whole content the input RDD.
- The data will be **copied to the driver**.

☞ If the data doesn't fit in main memory, the driver will crash.

Common actions

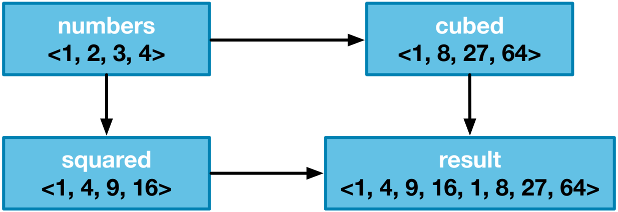
- `take(n)`. Returns *n* elements from the input RDD, while attempting to minimize the number of partitions it accesses.
  - It may represent a biased collection.
- `takeSample(withReplacement, num, seed)`. Sample data from the input RDD.
- `top(n)`. Returns the top *n* of the input RDD using the default or given ordering on data.
- `count()`. Counts the number of values in the input RDD.
- `countByValue()`. Counts the number of occurrence of each value in the input RDD.

Lineage graph

- As we derive new RDDs from each other using **transformations**, Spark keeps a **lineage graph** (dependencies between these RDDs.)

Example

```
numbers = sc.parallelize([1, 2, 3, 4])
squared = numbers.map(lambda x: x*x)
cubed = numbers.map(lambda x: x*x*x)
result = squared.union(cubed)
```



Lazy evaluation

- In Spark, transformations are **lazily evaluated**.

Definition (Lazy evaluation)

Lazy evaluation means that when a transformation is invoked, Spark **does not execute it** immediately. Transformations are only executed when Spark sees an action.

- An RDD can be thought of a set of *instructions* on how to compute the data that we build up through transformations.
- Lazy evaluation helps **reducing the number of passes** needed to load and transform the data.
  - In Hadoop, developers have to manually group operations in order to reduce the number of MapReduce iterations.
  - Spark does this optimization automatically.

Lazy evaluation

- Invoking `sc.textFile()` does not load immediately the data.
- The transformation `filter()` is not applied when it is invoked.
- Transformations are applied only when the action `count()` is invoked.
- **Only the data** that meet the constraint of the `filter` is loaded from the file.

Example

```
lines = sc.textFile("./data/logfile.txt")
exceptions = lines.filter(lambda line : "exception" in line)
nb_lines = exceptions.count()
print("Number of exception lines ", nb_lines)
```

☞ **Without lazy evaluation** we would have loaded into main memory **the whole content** of the input file.

Persisting the data

- With lazy evaluation, transformations are computed **each time** an action is invoked on a given RDD.
- In the following example, all transformations are computed when we invoke the function `count()` **and** the function `collect()`.

Example

```
lines = sc.textFile("./data/logfile.txt")
exceptions = lines.filter(lambda line : "exception" in line)
nb_lines = exceptions.count()
exceptions.collect()
```

- To avoid computing transformations multiple times, we can **persist** the data.

Persisting the data

- Persisting the data means **caching** the result of the transformations.
    - Either in main memory (default), or disk or both.
- If a node in the cluster fails, Spark **recomputes** the persisted partitions.
    - We can **replicate** persisted partitions on other nodes to recover from failures without recomputing.

```
lines = sc.textFile("./data/logfile.txt")
exceptions = lines.filter(lambda line : "exception" in line)
exceptions.persist(StorageLevel.MEMORY_AND_DISK)
nb_lines = exceptions.count()
exceptions.collect()
```

- `persist()` is called right before the first action.
- `persist()` does not force the evaluation of transformations.
- `unpersist()` can be called to evict persisted partitions.

Pair RDDs

- **Pair RDDs** are RDDs where each element is a key-value pair.

Pair RDD

```
words =  sc.parallelize(
    ['family', 'sport', 'fantasy', 'sport', 'sport', 'family'])
kvwords = words.map(lambda word : (word, 1))
```

- Pair RDDs allow all the transformations presented above.
- Pair RDDs allow all the actions presented above.
- Spark provides specific transformations and actions on Pair RDDs.

References

- Karau, Holden, et al. *Learning spark: lightning-fast big data analysis*. "O'Reilly Media, Inc.", 2015. Click here