Trường Đại học Công nghiệp Thành phố Hồ Chí Minh

pySpark

Giảng viên: Tiến sĩ Bùi Thanh Hùng
Bộ môn Khoa học dữ liệu
Khoa Công nghệ thông tin
Đại học Công nghiệp TP HCM
Email: buithanhhung@iuh.edu.vn
Website: https://sites.google.com/site/hungthanhbui1980/

Spark

Spark là một framework được xây dựng để hỗ trợ việc xử lý dữ liệu một cách phân tán.

Lợi thế là: nhanh,

tiếp cận dễ dàng,

hỗ trợ nhiều kiểu tính toán

(Hadoop MapReduce)

pySpark

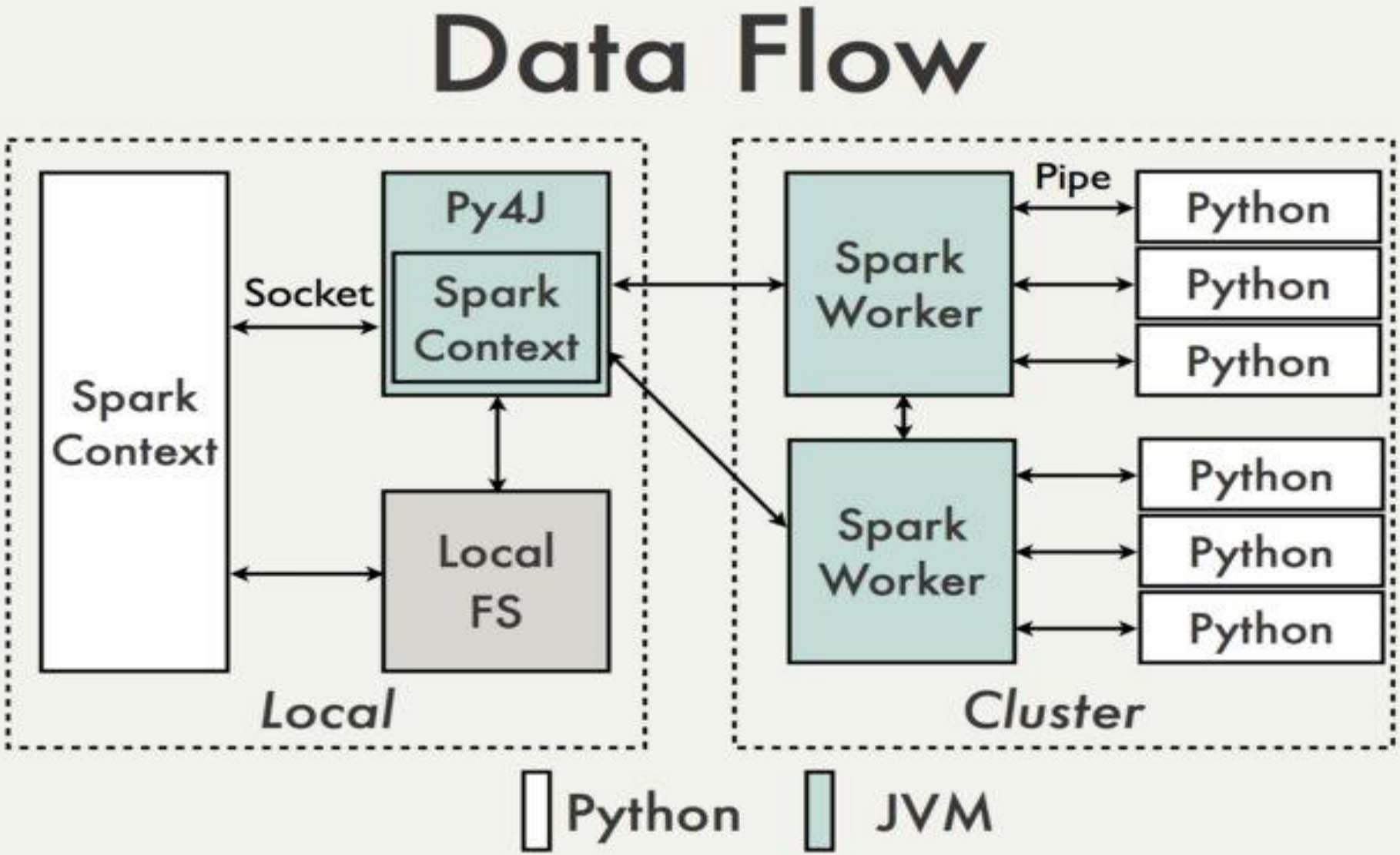
Pyspark được built trên Java API của Spark.

Bên trong **Python driver program**, SparkContext sẽ sử dụng Py4j để chạy **JVM** và khởi tạo **JavaSparkContext**.

JavaSparkContext có nhiệm vụ giao tiếp với các **Spark executors** của các cluster.

Python API khi gọi đến object **SparkContext**, những lời gọi, yêu cầu đó sẽ được **translate** bên trong **Java API calls** và được chuyển đến **JavaSparkContext**, dữ liệu được xử lý bằng Python sau đó được cached/shuffled trong JVM.

pySpark


Data Flow
Spark Context
Socket
Py4J
Spark Context
Local FS
Local
Pipe
Spark Worker
Spark Worker
Python
Python
Python
Python
Python
Python
Cluster
Python
JVM

pySpark

RDD (Resilient Distributed Datasets) được định nghĩa trong Spark Core.

RDD đại diện cho một collection các item đã được phân tán trên các cluster, và có thể xử lý phân tán.

PySpark sử dụng PySpark RDDs và nó chỉ là 1 object của Python nên khi viết code RDD transformations trên Java thực ra khi run, những transformations đó được ánh xạ lên object PythonRDD trên Java.

Spark SQL

Spark hỗ trợ một số **higher-level tool** như Spark SQL cho việc xử lý data có cấu trúc. *Mlib* cho Machine Learning, *GraphX* cho xử lý dạng biểu đồ.

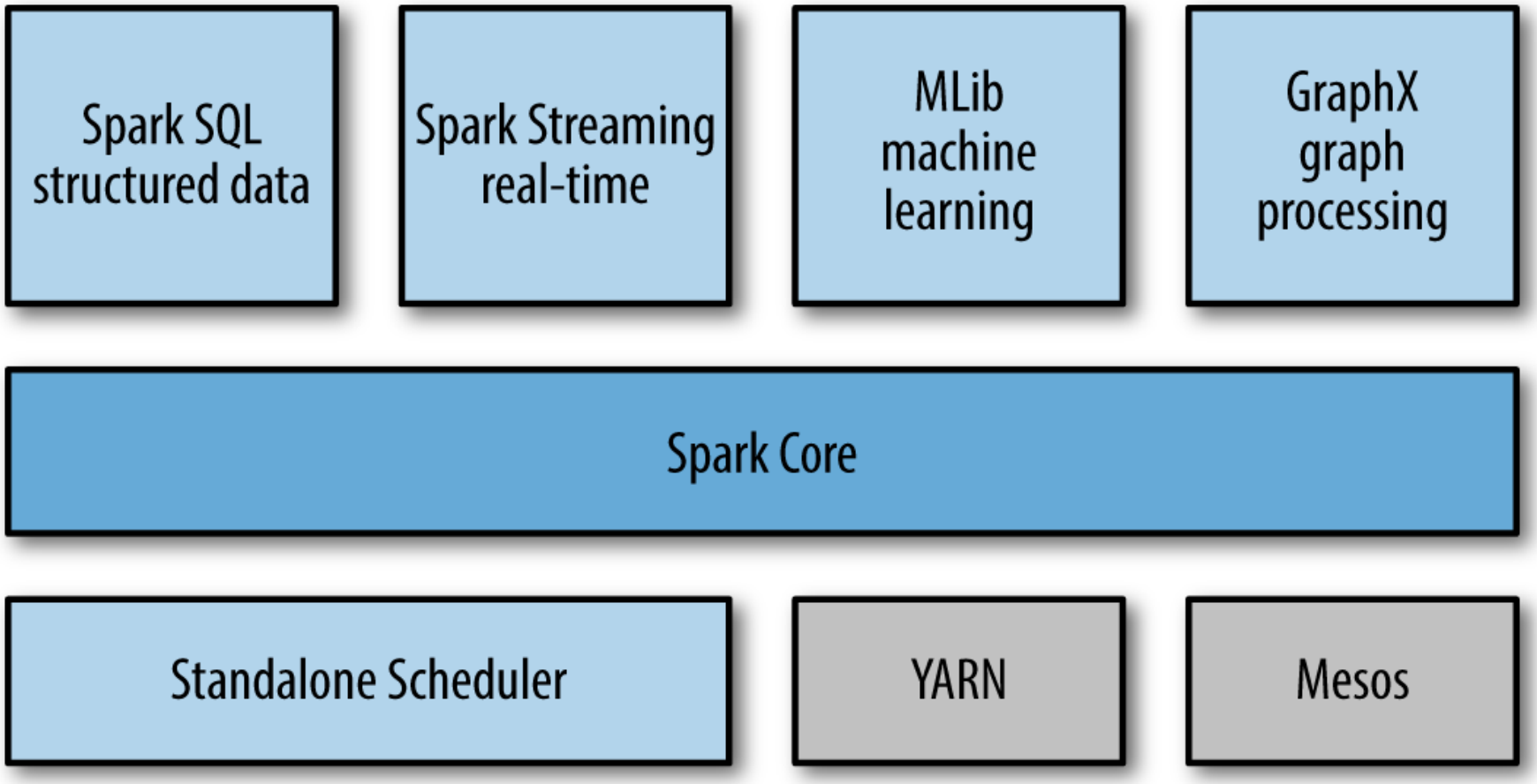
Bất cứ khi nào xử lý dữ liệu có cấu trúc điều đầu tiên mình muốn suggest là sử dụng Spark SQL, với interfaces được cung cấp bởi Spark SQL dẫn đến việc xử lý dễ dàng và hiệu quả.

Một số cách để tương tác với Spark SQL là: SQL, DataFrames API, Datasets API.

DataFrame

DataFrame trong Spark base dựa theo DataFrame trong R hay Pandas. Nó cũng khá tương tự như một bảng trong hệ quản trị CSDL quan hệ. DataFrames có thể được tạo từ file chứa dữ liệu có cấu trúc, tables trong Hive hoặc RDDs. Và nó cũng có một số điểm chung với RDD như immutable, lazy và distributed.

pySpark


Spark SQL
structured data
Spark Streaming
real-time
MLib
machine
learning
GraphX
graph
processing
Spark Core
Standalone Scheduler
YARN
Mesos

Cài đặt

```
!pip install pyspark
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
```

RDD

RDD (Resilient Distributed Datasets) được định nghĩa trong Spark Core.

RDD đại diện cho một collection các item đã được phân tán trên các cluster, và có thể xử lý phân tán.

PySpark sử dụng PySpark RDDs và nó chỉ là 1 object của Python nên khi viết code RDD transformations trên Java thực ra khi run, những transformations đó được ánh xạ lên object PythonRDD trên Java.

Resilient Distributed Datasets

Two ways to create RDD

- parallelize - from collection
- textFile - from file

Resilient Distributed Datasets

Two ways to create RDD

- parallelize - from collection
- textFile - from file

RDD creation from collection - mostly for learning & debugging purpose

#sc - handler to spark framework

```
rdd = sc.parallelize([1,2,3,4,5],2)
```

Resilient Distributed Datasets

Two ways to create RDD

- parallelize - from collection
- textFile - from file

RDD creation from collection - mostly for learning & debugging purpose

#sc - handler to spark framework

```
rdd = sc.parallelize([1,2,3,4,5],2)
```

Creating RDD using textFile from name.csv

```
names = sc.textFile('Names.csv')
```

Resilient Distributed Datasets

Two ways to create RDD

- parallelize - from collection
- textFile - from file

RDD creation from collection - mostly for learning & debugging purpose

#sc - handler to spark framework

```
rdd = sc.parallelize([1,2,3,4,5],2)
```

Creating RDD using textFile from name.csv

```
names = sc.textFile('Names.csv')
```

#PairRDD

```
l = [('awi',1000),('jack',2000),('jill',3000),('bill',500)]
rdd_pair = sc.parallelize(l)
rdd_pair.collect()
```

PairRDDFunctions

- collectAsMap()
- countByKey()*
- lookup()

Map Operations

- flatMap()
- map()
- mapPartitions()
- mapPartitionsWithIndex()

Set Operations

- cartesian()
- distinct()
- intersection()
- subtract()
- union()

Other Operations

- filter()
- groupBy()
- toDebugString

PairRDDFunctions (Single RDD)

- combineByKey()
- foldByKey()
- groupByKey()
- mapValues()
- reduceByKey()

PairRDDFunctions (Two RDD)

- cogroup()
- join()
- leftOuterJoin()
- rightOuterJoin()

Sorting

- sortByKey()
- takeOrdered()*
- top()
- Partition

Hash-Partition

- Partitioner set Operations
- Partitioner unset Operations
- range-partition

Shuffling

- coalesce()
- partitionBy()
- repartition()

Action (RDD)

- aggregate()
- collect()
- count()
- countByValue()
- first()
- fold()
- foreach()
- reduce()
- saveAsTextFile()
- take()
- takeOrdered()
- takeSample()
- top()

Broadcast variables

Khi có một lượng dữ liệu lớn muốn xử lý trên các nodes, sử dụng broadcast variables để giảm chi phí communication cost.

Nếu không lượng dữ liệu này sẽ được gửi riêng sang các nodes mỗi khi xử lý và với cơ chế chuyển mặc định được tối ưu cho các biến mang ít dữ liệu, đối với biến chứa lượng lớn dữ liệu thì sẽ chậm hơn.

Biến broadcast lưu dữ liệu dưới dạng read-only cached và deserialized trên mỗi cluster.

Broadcast variables

```
from pyspark import SparkContext
sc = SparkContext("local", "Broadcast app")
words_new = sc.broadcast(["scala", "java", "hadoop", "spark", "akka"])
data = words_new.value
print("Stored data -> %s" % (data))
elem = words_new.value[2]
print("Printing a particular element in RDD -> %s" % (elem))
```

Change to new Java version

```
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-11-openjdk-amd64"
```

Accumulator

Accumulator variables are used for aggregating the information through associative and commutative operations.

For example, you can use an accumulator for a sum operation or counters (in MapReduce).

Accumulator

```
from pyspark import SparkContext
sc = SparkContext("local", "Accumulator app")
num = sc.accumulator(10)
def f(x):
   global num
   num+=x
rdd = sc.parallelize([20,30,40,50])
rdd.foreach(f)
final = num.value
print("Accumulated value is -> %i" % (final))
```

count()

```
from pyspark import SparkContext
sc = SparkContext("local", "count app")
words = sc.parallelize (
   ["scala",
   "java",
   "hadoop",
   "spark",
   "akka",
   "spark vs hadoop",
   "pyspark",
   "pyspark and spark"]
)
counts = words.count()
print("Number of elements in RDD -> %i" % (counts))
```

collect()

```
from pyspark import SparkContext
sc = SparkContext("local", "Collect app")
words = sc.parallelize (
   ["scala",
   "java",
   "hadoop",
   "spark",
   "akka",
   "spark vs hadoop",
   "pyspark",
   "pyspark and spark"]
)
coll = words.collect()
print("Elements in RDD -> %s" % (coll))
```

foreach()

```
from pyspark import SparkContext
sc = SparkContext("local", "ForEach app")
words = sc.parallelize (
   ["scala",
   "java",
   "hadoop",
   "spark",
   "akka",
   "spark vs hadoop",
   "pyspark",
   "pyspark and spark"]
)
def f(x): print(x)
fore = words.foreach(f)
```

filter()

```
from pyspark import SparkContext
sc = SparkContext("local", "Filter app")
words = sc.parallelize (
   ["scala",
   "java",
   "hadoop",
   "spark",
   "akka",
   "spark vs hadoop",
   "pyspark",
   "pyspark and spark"]
)
words_filter = words.filter(lambda x: 'spark' in x)
filtered = words_filter.collect()
print("Fitered RDD -> %s" % (filtered))
```

map()

```
from pyspark import SparkContext
sc = SparkContext("local", "Map app")
words = sc.parallelize (
   ["scala",
   "java",
   "hadoop",
   "spark",
   "akka",
   "spark vs hadoop",
   "pyspark",
   "pyspark and spark"]
)
words_map = words.map(lambda x: (x, 1))
mapping = words_map.collect()
print("Key value pair -> %s" % (mapping))
```

reduce()

```
from pyspark import SparkContext
from operator import add
sc = SparkContext("local", "Reduce app")
nums = sc.parallelize([1, 2, 3, 4, 5])
adding = nums.reduce(add)
print("Adding all the elements -> %i" % (adding))
```

join()

```
from pyspark import SparkContext
sc = SparkContext("local", "Join app")
x = sc.parallelize([("spark", 1), ("hadoop", 4)])
y = sc.parallelize([("spark", 2), ("hadoop", 5)])
joined = x.join(y)
final = joined.collect()
print("Join RDD -> %s" % (final))
```

catch()

```
from pyspark import SparkContext
sc = SparkContext("local", "Cache app")
words = sc.parallelize (
   ["scala",
   "java",
   "hadoop",
   "spark",
   "akka",
   "spark vs hadoop",
   "pyspark",
   "pyspark and spark"]
)
words.cache()
caching = words.persist().is_cached
print("Words got chached > %s" % (caching))
```

MLlib

- Apache Spark offers a Machine Learning API called **MLlib**. PySpark has this machine learning API in Python as well. It supports different kind of algorithms, which are mentioned below −
- **mllib.classification** − The **spark.mllib** package supports various methods for binary classification, multiclass classification and regression analysis. Some of the most popular algorithms in classification are **Random Forest, Naive Bayes, Decision Tree**, etc.
- **mllib.clustering** − Clustering is an unsupervised learning problem, whereby you aim to group subsets of entities with one another based on some notion of similarity.

MLlib

- **mllib.fpm** − Frequent pattern matching is mining frequent items, itemsets, subsequences or other substructures that are usually among the first steps to analyze a large-scale dataset. This has been an active research topic in data mining for years.
- **mllib.linalg** − MLlib utilities for linear algebra.
- **mllib.recommendation** − Collaborative filtering is commonly used for recommender systems. These techniques aim to fill in the missing entries of a user item association matrix.

MLlib

- **spark.mllib** − It ¬currently supports model-based collaborative filtering, in which users and products are described by a small set of latent factors that can be used to predict missing entries. spark.mllib uses the Alternating Least Squares (ALS) algorithm to learn these latent factors.
- **mllib.regression** − Linear regression belongs to the family of regression algorithms. The goal of regression is to find relationships and dependencies between variables. The interface for working with linear regression models and model summaries is similar to the logistic regression case.

MLlib

There are other algorithms, classes and functions also as a part of the mllib package.

Refer

https://spark.apache.org/docs/3.1.1