Cái hay của Spark là nó có khả năng tích hợp rất mạnh mẽ trong Hadoop Ecosystem, có thể xử lý dữ liệu dạng Batch/ Near realtime, có sẵn các API phục vụ Machine Learning/ Graph Processing chạy phân tán với độ ổn định cực cao. **1.Apache Spark History**

Apache Spark bắt đầu tại University of California, Berkeley vào năm 2009 với tên là “Spark research project”, được giới thiệu public lần đầu tiên sau một năm (2010) trong một bài báo có tựa đề “Spark: Cluster Computing with Working Sets” viết bởi 5 nhà nghiên cứu của AMPlab — UC Berkeley. Vào thời điểm đó, Hadoop MapReduce là công cụ lập trình song song mạnh mẽ và cũng là dự án open-source đầu tiên để xử lý xử lý song song dữ liệu trên các cụm servers với hàng ngàn node. AMPlab đã làm việc với nhiều người dùng MapReduce sớm để hiểu những lợi ích và nhược điểm của mô hình lập trình mới này và do đó có thể tổng hợp danh sách các vấn đề qua một số use cases và bắt đầu thiết kế general-computing platforms. Ngoài ra, nhóm nghiên cứu cũng đã làm việc với Hadoop users tại UC Berkeley để hiểu nhu cầu của họ đối với nền tảng, cụ thể là các nhóm đang học máy quy mô lớn bằng thuật toán lặp cần thực hiện multi passes dữ liệu.

Trong suốt quá trình tìm hiểu, nhóm nghiên cứu nhận ra **2 vấn đề**:Cluster Computing có tiềm năng phát triển mạnh mẽ bởi các ứng dụng mới hoàn toàn có thể xây dựng và sử dụng hệ thống, dữ liệu hiện có.Công cụ MapReduce khiến việc xây dựng các ứng dụng lớn trở nên khó khăn và không hiệu quả.Từ đó nhóm quyết định phát triển Spark với mục tiêu cốt lõi là giúp việc xây dựng ứng dụng có thể phát triển dễ dàng cùng với khả năng scale-up linh hoạt để xử lý tập dữ liệu lớn và rất lớn, nhóm đã thiết kế một API dựa trên Functional Programming có thể viết ứng dụng một cách ngắn gọn. Sau đó triển khai API này qua một công cụ mới có thể thực hiện chia sẻ dữ liệu trong bộ nhớ hiệu quả, cho phép tái sử dụng qua các bước tính toán. Spark cung cấp tập hợp các Computing Engine và Libraries cho việc xử lý dữ liệu song song trên hệ thống lên tới hàng ngàn Servers, ngoài ra nó còn hỗ trợ nhiều ngôn ngữ lập trình được sử dụng rộng rãi (Python, Java, Scala và R), bao gồm các thư viện cho các tác vụ khác nhau, từ SQL đến Streaming, Machine Learning và graph-parallel computation. Năm 2013, dự án Spark được trao tặng cho Apache Software Foundation, open-source project này sau đó nhận được sự công nhận và đóng góp mạnh mẽ cộng đồng. Cuối cùng, 26/5/2014, Apache release Spark ver1.0 sẵn sàng apply trong môi trường production.

**2. The Spark Components:**

**Spark Core**: là engine thực thi chung làm nền tảng cho Spark. Tất cả các chức năng khác được xây dựng dựa trên base là Spark Core. Nó cung cấp khả năng tính toán trên bộ nhớ RAM và cả bộ dữ liệu tham chiếu trong các hệ thống external storage.

**Spark SQL**: là một thành phần nằm trên Spark Core, giới thiệu một khái niệm trừu tượng hóa dữ liệu mới gọi là SchemaRDD, cung cấp hỗ trợ cho dữ liệu có cấu trúc và bán cấu trúc.

**Spark Streaming**: tận dụng khả năng lập lịch memory-base của Spark Core để thực hiện streaming analytics. Nó lấy dữ liệu theo mini-batches và thực hiện các phép biến đổi RDD (Bộ dữ liệu phân tán có khả năng phục hồi) trên các mini-batches dữ liệu đó.

**MLlib (Machine Learning Library):** là một framework machine learning phân tán trên Spark tận dụng khả năng tính toán tốc độ cao nhờ distributed memory-based của kiến ​​trúc Spark.

**GraphX**: ​​là một framework xử lý đồ thị phân tán. Nó cung cấp một API để thực hiện tính toán biểu đồ có thể mô hình hóa các biểu đồ do người dùng xác định bằng cách sử dụng API đã được tối ưu sẵn.

**3. Spark vs Hadoop MapReduce**

Về cơ chế hoạt động của MapReduce (MR):

Input data được đọc từ HDFS (component phụ trách việc lưu trữ trong Hadoop) → xử lý bằng các thao tác chỉ định → output được ghi vào HDFS → data tiếp tục được load → thao tác tiếp theo được thực hiện → output tiếp tục ghi vào HDFS …

chuỗi các step [read-process-write] đó được lặp cho đến khi hoàn thành công việc. Vì input được chia thành các block độc lập với nhau, các task map-reduce được thực hiện song song, nên về cơ bản nó hữu ích để xử lí những bộ dữ liệu lớn. Tuy nhiên, MR vẫn còn những tồn tại là quá trình xử lý không thực sự hiệu quả trong trường hợp phải lặp lại nhiều step, vì mỗi step cần thiết phải ghi output vào HDFS trước khi step tiếp theo được thực hiện → việc này tạo ra các vấn đề trong việc lưu trữ và replicate, tăng độ trễ xử lý do phần lớn thực hiện trên Disk vốn có hiệu suất I/O không cao. Bên cạnh đó là việc develop, debug với MR có phần khó khăn vì code dài dòng. **Cơ chế hoạt động của Spark:** khắc phục những tồn tại của Hadoop MapReduce, Spark đưa ra một khái niệm mới RDD — Resilient Distributed Dataset đóng vai trò như 1 cấu trúc dữ liệu cơ bản trong Spark, RDD được định nghĩa là trừu tượng cho một tập hợp các phần tử bất biến (bản chất là được lưu trên các ô nhớ read-only), được phân vùng có thể được chia sẻ, tác động song song. Qua đó, input data từ storage system chỉ cần load 1 lần duy nhất, các step thực hiện biến đổi, xử lý input data được lên kế hoạch, optimize và thực hiện một cách liên tục cho đến khi output được trả khi kết thúc công việc. Toàn bộ quá trình đó được diễn ra trên bộ nhớ RAM (khi hết RAM sẽ được chuyển sang xử lý trên Disk) tận dụng được hiệu suất I/O cao từ đó có thể giảm thời gian thực thi nhỏ hơn 10–100 lần Hadoop MapReduce. Với Apache Spark, các dữ liệu trung gian được lưu trữ in-memory, trong các object RDD

**4. Apache Spark Pros & Cons**

Tính năng và cũng là **ưu điểm** của Spark có thể kể ra :

**Advanced Analytics:** Spark không chỉ hỗ trợ “Map” và “Reduce “, nó còn hỗ trợ Spark truy vấn SQL, Streaming data, Machine learning (ML) và các thuật toán xử lý đồ thị đóng vai trò như một bộ công cụ phân tích dữ liệu cực kì mạnh mẽ.

**Speed:** Spark giúp chạy một ứng dụng với tốc độ rất nhanh. So với Hadoop cluster, Spark Application nến chạy trên bộ nhớ nhanh hơn tới 100 lần và nhanh hơn 10 lần khi chạy trên đĩa. Điều này có được nhờ giảm số lượng các hoạt động đọc / ghi vào ổ đĩa.

**Supports multiple languages:** Spark cung cấp built-in APIs phổ biến từ Java, Scala đến Python, R. Do đó, có thể code Spark applications với nhiều lựa chọn về ngôn ngữ lập trình. Bên cạnh đó Spark còn cung cấp rất nhiều high-level operators cho việc truy vấn dữ liệu…

**Nhược điểm:**

Spark không có hệ thống Filesystem riêng, do đó, nó phụ thuộc vào một số nền tảng khác như Hadoop hoặc một nền tảng dựa trên đám mây (S3, Google Cloud Storage,…).

Apache Spark đòi hỏi rất nhiều RAM để chạy trong bộ nhớ, do đó chi phí của Spark khá cao.

Spark Streaming không thực sự real-time.

Việc tối ưu hóa, tinh chỉnh để phù hợp với các bộ dữ liệu cụ thể cần có kinh nghiệm và vẫn cần thực hiện thủ công.

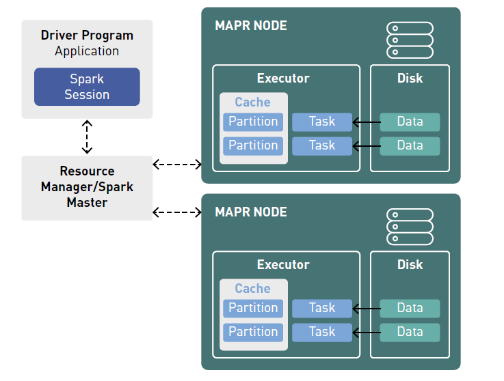
**Real-World Use cases**

Ebay: eBay sử dụng Apache Spark để cung cấp các ưu đãi được nhắm mục tiêu, nâng cao trải nghiệm của khách hàng và để tối ưu hóa hiệu suất tổng thể. Apache Spark được tận dụng tại eBay thông qua Hadoop YARN.Alibaba: Alibaba một trong những nền tảng thương mại điện tử lớn nhất thế giới, sử Apache Spark để phân tích hàng trăm petabyte dữ liệu trên nền tảng thương mại điện tử của mình. Một số công việc Spark thực hiện trích xuất tính năng trên dữ liệu hình ảnh, chạy trong vài tuần. Hàng triệu thương nhân và người dùng tương tác với nền tảng thương mại điện tử Alibaba Taobao. Mỗi tương tác này được biểu diễn dưới dạng một biểu đồ lớn phức tạp và Spark được sử dụng để xử lý nhanh quá trình bằng các thuật toán ML tinh vi trên dữ liệu này.Những use case tiêu biểu khác có thể liệt kê như các công ty công nghệ như Uber và Netflix sử dụng các công cụ Spark Streaming và MLlib, đến các tổ chức như NASA, CERN và Broad Institute of MIT và Harvard áp dụng Spark vào phân tích dữ liệu khoa học.

**Spark Application**

Một chương trình Spark sẽ gồm 2 thành phần chính:

**Driver Program:** Là 1 JVM Process, chứa hàm main() như bất kì 1 chương trình JVM nào khác, nó đóng vai trò điều phối code/ logic xử lý trên Driver. Driver program chứa **SparkSession**

**Executors:** Là các worker, chịu trách nhiệm thực hiện các tính toán các logic nhận từ Driver. Dữ liệu cần xử lý có thể được load trực tiếp vào memory của Executor.

Spark session: Đại diện cho khả năng tương tác với executors trong 1 chương trình. Spark session chính là entry point của mọi chương trình Spark. Từ SparkSession, có thể tạo RDD/ DataFrame/ DataSet, thực thi SQL… từ đó thực thi tính toán phân tán.

Khi chạy, từ logic của chương trình (chính là code xử lý thông qua việc gọi các API), Driver sẽ sinh ra các task tương ứng và lên lịch chạy các task, sau đó gửi xuống Executor để thực thi. Dữ liệu được lưu trên memory của Executor nên việc thực thi tính toán sẽ nhanh hơn rất nhiều.

**RDD**

Rõ ràng là dữ liệu phân tán rời rạc trong mạng, vậy làm sao để có thể tác động lên tập dữ liệu này? Spark giới thiệu 1 khái niệm là RDD (Resilient Distributed Dataset), dịch thô ra tiếng việt là: Tập dữ liệu phân tán linh hoạt. Trong 1 chương trình Spark, RDD là đại diện cho tập dữ liệu phân tán.

Một ví dụ cho các bạn mới dễ hiểu nhé.

Ta có 1 object colors = Array {red, blue, black, white, yellow}. Trong chương trình thông thường, phần dữ liệu (red, blue, black, white, yellow) nằm trên 1 máy tính duy nhất và thông qua biến colors, bạn có thể truy cập đến dữ liệu. Tuy nhiên trong hệ phân tán, nếu phần dữ liệu red, blue nằm trên máy tính A còn black, white, yellow lại nằm trên máy tính B thì sao? RDD sẽ giúp bạn truy cập 2 phần dữ liệu rời rạc này như 1 đối tượng thông thường. Đó là lý do tại sao mình nói RDD là đại diện cho tập dữ liệu phân tán. Điều này có nghĩa là, nó lưu trữ trạng thái bộ nhớ như một đối tượng trên các công việc và đối tượng có thể chia sẻ giữa các công việc đó. Chia sẻ dữ liệu trong bộ nhớ nhanh hơn mạng và Đĩa từ 10 đến 100 lần. Tất cả công việc trong Spark được thể hiện dưới dạng tạo RDD mới, chuyển đổi RDD hiện có hoặc gọi các hành động trên RDD để tính toán kết quả. Spark tự động phân phối dữ liệu có trong RDD trên toàn bộ cụm của bạn và song song hóa các thao tác bạn thực hiện trên chúng

Đặc điểm quan trọng của 1 RDD là số partitions. Một RDD bao gồm nhiều partition nhỏ, mỗi partition này đại diện cho 1 phần dữ liệu phân tán. Khái niệm partition là logical, tức là 1 node xử lý có thể chứa nhiều hơn 1 RDD partition. Theo mặc định, dữ liệu các partitions sẽ lưu trên memory. Thử tưởng tượng bạn cần xử lý 1TB dữ liệu, nếu lưu hết trên mem tính ra thì cung khá tốn kém nhỉ. Tất nhiên nếu bạn có 1TB ram để xử lý thì tốt quá nhưng điều đó không cần thiết. Với việc chia nhỏ dữ liệu thành các partition và cơ chế lazy evaluation của Spark bạn có thể chỉ cần vài chục GB ram và 1 chương trình được thiết kế tốt để xử lý 1TB dữ liệu, chỉ là sẽ chậm hơn có nhiều RAM thôi. Mỗi Executor có thể chứa dữ liệu của 1 hoặc 1 vài partition của 1 RDD.

**Tạo RDD**:

Lấy các tệp dữ liệu: #Creating a RDD from a file

data\_file = "./kddcup.data\_10\_percent.gz"

raw\_data = sc.textFile(data\_file)

#Creating RDD using parallelize from an already existing list.

a = range(100)

data = sc.parallelize(a)

**Cơ bản về RDD**: **# The filter transformation**

normal\_raw\_data = raw\_data.filter(lambda x: 'normal.' in x)

normal\_count = normal\_raw\_data.count() **# The map transformation**

csv\_data = raw\_data.map(lambda x: x.split(","))

head\_rows = csv\_data.take(5) **# Using map and predefined functions**

def parse\_interaction(line):

elems = line.split(",")

tag = elems[41]

return (tag, elems)

key\_csv\_data = raw\_data.map(parse\_interaction)

head\_rows = key\_csv\_data.take(5)

**# The collect action**

all\_raw\_data = raw\_data.collect()

**Thu thập tất cả các normal tương tác dưới dạng các cặp khóa-giá trị**: # get data from file

data\_file = "./kddcup.data\_10\_percent.gz"

raw\_data = sc.textFile(data\_file)# parse into key-value pairs

key\_csv\_data = raw\_data.map(parse\_interaction)# filter normal key interactions

normal\_key\_interactions = key\_csv\_data.filter(lambda x: x[0] == "normal.")# collect all

all\_normal = normal\_key\_interactions.collect()

normal\_count = len(all\_normal)

print("There are {} 'normal' interactions".format(normal\_count))

**#The sample transformation**

raw\_data\_sample = raw\_data.sample(False, 0.1, 1234)

sample\_size = raw\_data\_sample.count()

total\_size = raw\_data.count()

print("Sample size is {} of {}".format(sample\_size, total\_size)) **Output :**

Sample size is 489957 of 4898431 **#The takeSample action**

raw\_data\_sample = raw\_data.takeSample(False, 400000, 1234)

normal\_data\_sample = [x.split(",") for x in raw\_data\_sample if "normal." in x]

normal\_sample\_size = len(normal\_data\_sample)

normal\_ratio = normal\_sample\_size / 400000.0

print("The ratio of 'normal' interactions is {}".format(normal\_ratio)) The ratio is 0.1988025

**Sử dụng reduce: # parse data**

csv\_data = raw\_data.map(lambda x: x.split(",")) **# separate into different RDDs**

normal\_csv\_data = csv\_data.filter(lambda x: x[41]=="normal.")

attack\_csv\_data = csv\_data.filter(lambda x: x[41]!="normal.")

normal\_duration\_data = normal\_csv\_data.map(lambda x: int(x[0]))

attack\_duration\_data = attack\_csv\_data.map(lambda x: int(x[0]))

total\_normal\_duration = normal\_duration\_data.reduce(lambda x, y: x + y)

total\_attack\_duration = attack\_duration\_data.reduce(lambda x, y: x + y)

print("Total duration for 'normal' interactions is {}".format(total\_normal\_duration))

print("Total duration for 'attack' interactions is {}".format(total\_attack\_duration))

normal\_count = normal\_duration\_data.count()

attack\_count = attack\_duration\_data.count()

print("Meanduration\_for\_'normal'\_interactions\_is{}".format(round(total\_normal\_duration/float(normal\_count),3)))print("Mean\_duration\_for\_'attack'\_interactions is{}"… (attack\_count),3)))

**Sử dụng aggregate :** (0,0), # the initial value

(lambda acc, value: (acc[0] + value, acc[1] + 1)), # combine value with acc

(lambda acc1, acc2: (acc1[0] + acc2[0], acc1[1] + acc2[1])) # combine accumulators

)

attack\_sum\_count = attack\_duration\_data.aggregate(

(0,0), # the initial value

(lambda acc, value: (acc[0] + value, acc[1] + 1)), # combine value with acc

(lambda acc1, acc2: (acc1[0] + acc2[0], acc1[1] + acc2[1])) # combine accumulators

) / print(…)

**Làm việc với RDD của cặp khóa / giá trị: Tạo một cặp RDD:**

csv\_data = raw\_data.map(lambda x: x.split(","))

key\_value\_data = csv\_data.map(lambda x: (x[41], x)) # x[41] contains the network interaction tag

**Tổng hợp dữ liệu với RDD của cặp khóa / giá trị: Sử dụng reduceByKey :**

key\_value\_duration = csv\_data.map(lambda x: (x[41], float(x[0])))

durations\_by\_key = key\_value\_duration.reduceByKey(lambda x, y: x + y)

durations\_by\_key.collect()

counts\_by\_key = key\_value\_data.countByKey()

counts\_by\_key **Output :**

defaultdict(<type 'int'>, {u'guess\_passwd.': 53, u'nmap.': 231, u'warezmaster.': 20, u'rootkit.': 10, ..)

**Sử dụng combineByKey :** Chúng ta có thể coi nó là aggregate tương đương vì nó cho phép người dùng trả về các giá trị không cùng loại với dữ liệu đầu vào của chúng ta.

sum\_counts = key\_value\_duration.combineByKey(

(lambda x: (x, 1)), # the initial value, with value x and count 1

(lambda acc, value: (acc[0]+value, acc[1]+1)), # how to combine a pair value with the accumulator: sum value, and increment count

(lambda acc1, acc2: (acc1[0]+acc2[0], acc1[1]+acc2[1])) # combine accumulators

)sum\_counts.collectAsMap()

duration\_means\_by\_type = sum\_counts.map(lambda (key,value): (key, round(value[0]/value[1],3))).collectAsMap() **# Print them sorted**

for tag in sorted(duration\_means\_by\_type, key=duration\_means\_by\_type.get, reverse=True):

print(tag, duration\_means\_by\_type[tag]) **Output**: portsweep. 1915.299…

**Spark SQL: xử lý có cấu trúc để phân tích dữ liệu**

Về cơ bản, mọi thứ đều xoay quanh khái niệm Khung dữ liệu và sử dụng ngôn ngữ SQL để truy vấn chúng. Chúng ta sẽ thấy cách trừu tượng hóa khung dữ liệu, rất phổ biến trong các hệ sinh thái phân tích dữ liệu khác (ví dụ như R và Python / Pandas), nó rất mạnh khi thực hiện phân tích dữ liệu khám phá. Trên thực tế, rất dễ dàng để diễn đạt các truy vấn dữ liệu khi được sử dụng cùng với ngôn ngữ SQL. Hơn nữa, Spark phân phối cấu trúc dữ liệu dựa trên cột này một cách minh bạch, để làm cho quá trình truy vấn hiệu quả nhất có thể. **Lấy khung dữ liệu:**

Spark DataFramelà một tập hợp dữ liệu phân tán được tổ chức thành các cột được đặt tên. Về mặt khái niệm, nó tương đương với một bảng trong cơ sở dữ liệu quan hệ hoặc một khung dữ liệu trong R hoặc Pandas. Chúng có thể được xây dựng từ nhiều nguồn như RDD hiện có trong trường hợp của chúng tôi.

Điểm vào tất cả các chức năng SQL trong Spark là SQLContextlớp. Để tạo một thể hiện cơ bản, tất cả những gì chúng ta cần là một SparkContexttham chiếu. Vì chúng tôi đang chạy Spark ở chế độ shell (sử dụng pySpark), chúng tôi có thể sử dụng đối tượng ngữ cảnh toàn cục sccho mục đích này.

Spark SQL có thể chuyển đổi RDD của Rowcác đối tượng thành a DataFrame. Các hàng được xây dựng bằng cách chuyển một danh sách các cặp khóa / giá trị dưới dạng kwargs cho Rowlớp. Các phím xác định tên cột và các loại được suy ra bằng cách nhìn vào hàng đầu tiên. Do đó, điều quan trọng là không có dữ liệu bị thiếu trong hàng đầu tiên của RDD để có thể suy ra lược đồ một cách chính xác.

Trong trường hợp của chúng ta, trước tiên chúng ta cần tách dữ liệu được phân tách bằng dấu phẩy, rồi sử dụng thông tin trong mô tả nhiệm vụ 1999 của KDD để lấy tên cột .

from pyspark.sql import SQLContext, Row

sqlContext = SQLContext(sc)

#Inferring the schema

csv\_data = raw\_data.map(lambda l: l.split(","))

row\_data = csv\_data.map(lambda p: Row(duration=int(p[0]), protocol\_type=p[1],

service=p[2], flag=p[3],src\_bytes=int(p[4]),dst\_bytes=int(p[5]) ) )

#Once we have our RDD of Row we can infer and register the schema.

interactions\_df = sqlContext.createDataFrame(row\_data)

interactions\_df.registerTempTable("interactions"). **Truy vấn SQL qua khung dữ liệu dưới dạng bảng.**

# Select tcp network interactions with more than 1 second duration and no transfer from destination

tcp\_interactions = sqlContext.sql(""" SELECT duration, dst\_bytes FROM interactions WHERE protocol\_type = 'tcp' AND duration > 1000 AND dst\_bytes = 0 """)

tcp\_interactions.show() **Output:** duration dst\_bytes 5057 0 …

Kết quả của các truy vấn SQL là RDD và hỗ trợ tất cả các hoạt động RDD bình thường**. # Output duration together with dst\_bytes:** tcp\_interactions\_out = tcp\_interactions.map(lambda p: "Duration: {}, Dest. bytes: {}".format(p.duration, p.dst\_bytes))

for ti\_out in tcp\_interactions\_out.collect():

print ti\_out

**Truy vấn dưới dạng DataFramehoạt động:** Spark DataFrame cung cấp một ngôn ngữ dành riêng cho miền để thao tác dữ liệu có cấu trúc. Ngôn ngữ này bao gồm các phương thức mà chúng ta có thể ghép nối để thực hiện việc lựa chọn, lọc, nhóm, v.v. Ví dụ, giả sử chúng ta muốn đếm có bao nhiêu tương tác cho mỗi loại giao thức. Chúng ta có thể tiến hành như sau. Giả sử chúng ta muốn đếm có bao nhiêu tương tác cho mỗi loại giao thức. Chúng ta có thể tiến hành như sau.

interactions\_df.select("protocol\_type", "duration", "dst\_bytes").groupBy("protocol\_type").count().show()

print("Query performed in {} seconds".format(round(tt,3))) **Output:** protocol\_type count: udp 20354 tcp 190065 icmp 283602

**4. Lazy Evalution: Transformation**

Việc xử lý dữ liệu, nhìn rộng ra, chính là việc biến đổi từ tập dữ liệu này sang tập dữ liêu khác, hay với Spark, là biến đổi từ RDD này sang RDD khác.

VD bạn có 1 tập web log rất lớn gồm cả log call từ app và web, cần tìm xem số lượng log sử dụng trên app là bao nhiêu, từ tập dữ liệu ban đầu sẽ lọc ra tập dữ liệu nhỏ hơn chỉ chứa log call trên di động, rồi count trên tập dữ liệu đó. Đó chính là minh họa cho việc biến đổi các tập dữ liệu.

Làm việc với RDD, Spark có 2 loại operations là Transformation và Actions. Phép biến đổi từ RDD này sang RDD khác là 1 transformation, như việc biến đổi tập web log ban đầu sang tập web log chỉ chứa log gọi qua app là 1 transformation.

**Action:** Sau tất cả các phép biến đổi, khi muốn tương tác với kết quả cuối cùng (VD xem kết quả, collect kết quả, ghi kết quả…) ta gọi 1 action.

**Lazy evaluation** Khi thực thi, việc gọi các transformations, Spark sẽ không ngay lập tức thực thi các tính toán mà sẽ lưu lại thành 1 lineage, tức là tập hợp các biến đổi từ RDD này thành RDD khác qua mỗi transformation. Khi có 1 action được gọi, Spark lúc này mới thực sự thực hiện các biến đổi để trả ra kết quả.

Nếu gọi đến đâu chạy đến đấy thì sau bước đầu tiên, ứng dụng cần load sẵn 1TB ram vào Mem, sẵn sàng thực hiện bước tiếp theo. Tuy nhiên, nếu lazy, Driver sẽ có “cái nhìn” toản cảnh từ đầu đến cuối về output đầu ra, lúc này Driver sẽ sinh các task đọc từng phần nhỏ của 1TB, thực hiện lọc trên tập dữ liệu nhỏ và giữ lại kết quả count bản ghi là log app, load dần dần cho tới khi hết 1TB thì sum tổng các phần nhỏ lại sẽ ra được phần lớn.

Như vậy ta chỉ cần lượng ram nhỏ nhưng vẫn có thể xử lý được lượng dữ liệu lớn gấp nhiều lần.

**SparkSQL** Apache Spark là framework xử lý xư liệu phân tán, tập hợp nhiều thư viện cho các mục đích khác nhau và build on-top Spark Core. Điều này giúp Spark có tính nhất quán lớn, thay vì mỗi mục đích bạn phải tìm 1 thư viện riêng thì với việc có sẵn như Spark, trong 1 project bạn vừa có thể sử dụng Spark Core kết hợp với Spark SQL hay Mllib một cách linh hoạt, tối ưu các công việc theo đúng mục đích của chúng.

Khi thực thi, Spark SQL vẫn sẽ gọi xuống lớp Core bên dưới, sử dụng RDD để tính toán.

Có thể tóm gọn một số đặc điểm quan trọng của Spark SQL như sau:

Được xây dựng phía trên tầng Spark Core, thừa hưởng tất cả các tính năng mà RDD có.

Làm việc với tập dữ liệu là DataSet hoặc DataFrame (tập dữ liệu phân tán, có cấu trúc)

Hiệu năng cao, khả năng mở rộng và chịu lỗi tốt

Tương tích với các thành phần khác trong tổng thể Spark Framework (như Streaming/ Mllib, GraphX)

Bao gồm 2 thành phần là DataSet API và Catalyst Optimizer.

**Spark SQL Performance** Không quá khi nói rằng, trong việc biến đổi và tổng hợp dữ liệu, Spark SQL với DataFrame luôn có hiệu năng cao hơn rất nhiều so với sử dung RDD. Còn so với các công nghệ tương tự khác như Impala hay Shark thì thời gian thực thi với Spark SQL cũng luôn rất ấn tượng.

**DataFrame** Vậy là có thể thấy Spark SQL thực sự hiệu quả trong việc xử lý dữ liệu như nào. Cốt lõi của Spark SQL chính là DataFrame và các API tương tác. Vậy DataFrame là gì?

Hiểu đơn giản DF giống như 1 bảng trong hệ CSDL quan hệ. Có các trường, được định sẵn kiểu dữ liệu, và tập hợp các bản ghi. Tuy nhiên 1 DF có thể đại diện cho lượng dữ liệu lớn hơn rất nhiều so với các bảng trong DB, bản thân mình đã xử lý các DF lên tới hàng chục tỉ row.

Các đặc điểm bao gồm:

**immutable:** tính bất biến, dữ liệu của 1 DF sau khi tạo ra sẽ không thay đổi, nếu muốn chỉnh sửa ta cần tạo ra DF mới từ DF ban đầu, thông qua DF api.

**rows:** là đối tượng đại diện cho 1 bản ghi dữ liệu. 1 DF = tập các row phân tán

**set of columns has name and an associated type:** Ý nói về việc dữ liệu của DF là có cấu trúc, gồm tên là kiểu dữ liệu.

Spark SQL hỗ trợ rất nhiều nguồn dữ liệu như file, DB…. Với DataFrameReader, bạn có thể dễ dàng đọc dữ liệu từ nhiều nguồn, nhiều định dạng để rạo ra 1 DataFrame, từ đó có thể sử dụng các API của Spark SQL tương tác với chúng.

**DataFrame or DataSet** Bỏ qua về RDD do nó nằm ở tầng Spark Core, DataFrame được giới thiệu vào năm 2013 và DataSet là 2015. Sau khi tồn tại độc lập được 1 năm, tới 2016, Spark 2.0 ra mắt và gộp DF và DS lại, chỉ còn duy nhất DS.

Cả DF và DS đều là tập dữ liệu trong Spark SQL, tuy nhiên điểm khác biệt là các bản ghi trong DF được fix luôn là đối tượng Row, còn DS thì có thể tùy chỉnh được kiểu dữ liệu của bản ghi thông qua định nghĩa case class.

DataFrame = Dataset [Row] (Untyped API)

DataSet = Dataset [Type]

Một nhược điểm của DF là do kiểu dữ liệu được fix là row và truy cập dữ liệu trong DF thông qua row name nên nếu có sai sót trong việc truyền tên cột, trình biên dịch sẽ không thể phát hiện ra lỗi mà khi thực thi mới có exeption (Runtime exception). DS có thể khắc phục nhược điểm này do có sẵn định nghĩa của 1 bản ghi, dữ liệu trong DS truy cập thông qua member của case class chứ không phải truyển tên như DF.

**Spark Catalyst Optimizer** Như các bạn đã biết, Spark SQL có 2 thành phần chính:

DataFrame API, cung cấp các hàm hỗ trợ tính toán và xử lý dữ liệu có cấu trúc phân tán.

Catalyst Optimizer: Bộ tối ưu thực thi.

Trong hầu hết các bài test, Spark SQL luôn có hiệu năng rất tốt, một trong những lý do để có được điều này là do SparkSQL có module tối ưu thực thi Catalyst Optimizer.

Catalyst Optimizer sẽ như một bước bảo đảm sẽ loại bỏ bớt sự chưa tối ưu đó 1 cách chủ động, thông qua tập rule-base và cost-base được xây dựng sẵn.

Như hình trên có thể thấy, Catalyst Optimizer đóng vai trò trung gian, một bên là code của người dùng, các tương tác đọc/ ghi/ tính toán với dữ liệu thông qua DataFrame API, một bên là những gì sẽ được thực thi trên các executor.

Nói ngắn gọn, Catalyst Optimizer có thể sẽ “chỉnh sửa lại” code của bạn trước khi thực thi đảm bảo việc tính toán hiệu quả nhất mà vẫn đảm bảo kết quả như bạn mong muốn.

**Try out SparkSQL and DataFrame**

Bắt đầu nhé, trong notebook, sử dụng đoạn code sau để tạo dữ liệu sample, ở đây mình tạo 1 DF có 4 trường name, gender, weight, graduation\_year với 7 bản ghi:

//Prepare data

case class Student(name:String, gender:String, weight:Int, graduation\_year:Int)

val df = Seq(Student("John", "M", 180, 2015), Student("Mary", "F", 110, 2015), Student("Derek", "M", 200, 2015), Student("Julie", "F", 109, 2015), Student("Allison", "F", 105, 2015), Student("kirby", "F", 115, 2016), Student("Jeff", "M", 195, 2016)).toDF

**How Spark run on a Cluster** The Architecture of a Spark Application

**The Spark Driver:** Trong mô hình master — slave thì Spark Driver chính là Master, nó chứa Spark Session, thứ mà giúp các bạn tương tác với hàng chục, hàng trăm máy tính khác để giao các công việc tính toán. Những tính toán không thông qua RDD hoặc DataFrame sẽ được thực thi trên Driver.

**The Spark Executors**: Trong thực tế, 1 chương trình spark chỉ có 1 Driver nhưng lại có thể có đến hàng trăm, hàng nghìn executor, chúng làm nhiệm vụ tính toán và trả về kết quả cho Driver.

**The Cluster Manager:** nghĩ 1 chút nhé, lại là về phân tán, nếu thông thường bạn chạy 1 ứng dụng, hệ điều hành sẽ cung cấp ram, cpu, data cho tiến trình của bạn, tuy nhiên do Spark chạy phân tán trên nhiều máy tính nên ta cần 1 “ai đó” đứng ra điều phối tài nguyên giữa các máy này, đó chính là Cluster Manager. Bản thân Cluster Managercũng có mô hình master — slave như sau:

Spark đang hỗ trợ các loại cluster manager như sau:

**Hadoop YARN, Apache Mesos, Kubernetes, Spark Standalone**

Execution Modes

Spark cung cấp sẵn cho người dùng spark-submit

Dùng để lựa chọn cách mà bạn “đặt” các thành phần của chương trình Spark như Driver hay Executor ở đâu. Có 3 lựa chọn:**Cluster mode, Client mode, Local mode**

**Cluster mode:** Trong chế độ này, toàn bộ Spark Driver và Spark Executor sẽ được Cluster Manager quản lý, nói cách khác, Cluster Manager sẽ coi Driver và Executor đều trong container của chúng.

**Client mode:** Ở chế độ này, Cluster Manager chỉ quản lý Executor, Driver sẽ sử dụng tài nguyên trên máy tính mà người dùng submit job.

**Local mode:** Ngược lại vs 2 cách trên, Local mode sẽ chạy cả Spark Driver và Executor trên máy tính của bạn, đây là cách đơn giản để có môi trường học và thực hành Spark.

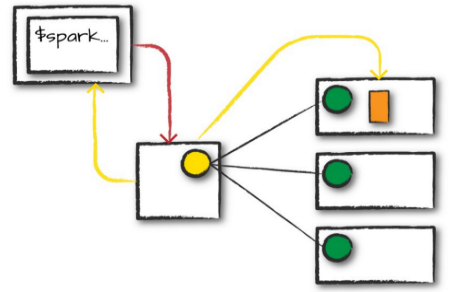
**The Life Cycle of a Spark Application**

Vòng đời 1 ứng dụng Spak có thể được chia là 4 giai đoạn như sau:

Client Request, Launch, Execution, Completion

**Client request:**

Một chương trình bắt đầu khi bạn submit, ví dụ 1 gói jar chứa code cần thực thi phân tán sử dụng thư viện Apache Spark, tùy từng mode ở trên mà sẽ có 1 chút khác biệt, tuy nhiên có thể tổng quan như hình sau:

Spark submit sẽ contact với master của Cluster Manager, yêu cầu tạo Spark Driver (Spark Session sẽ là 1 object trong Driver) sau đó, giả sử với mode cluster, Cluster Manager sẽ tạo các Container tương ứng cho Spark Driver.

Cú pháp cho Spark Submit như sau:

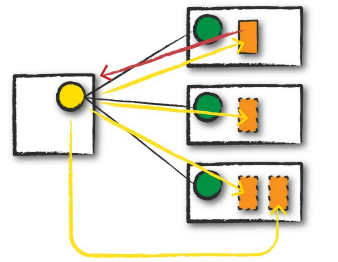
./bin/spark-submit \

--class <main-class> \

--master <master-url> \

--deploy-mode cluster \

--conf <key>=<value> \...

# other options<application-jar> \[application-arguments]

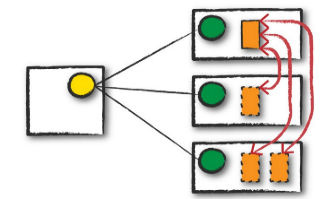
**Launch**

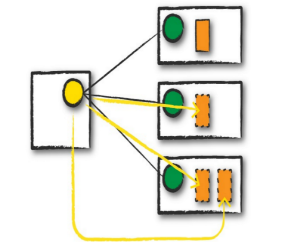
Khi có Spark Driver, ứng dụng sẽ tùy theo cấu hình mà yêu cầu cấp phát thêm tài nguyên để khởi chạy Spark Executor.

**Execution**

Đây là lúc thực thi logic trên tài nguyên đã xin cấp phát, các logic map, reduce, filter sẽ chạy ở bước này. Do đóng vai trò điều phối, Driver thường xuyên contact với Executor để giao task hoặc cập nhật trạng thái, bản thân các Executor cũng contact với nhau khi cần move dữ liệu nếu cần.

**Completion**

Là giai đoạn cuối cùng, khi các tính toán đã xong hoặc vì 1 lý do nào đó Spark Driver down, Cluster Manager sẽ thực hiện thu hồi lại các tài nguyên đã cấp cho Executor.

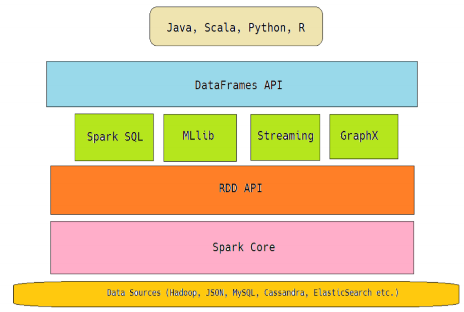
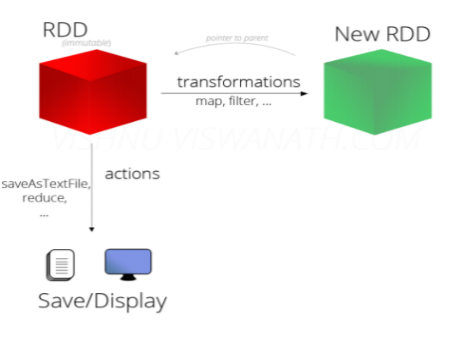


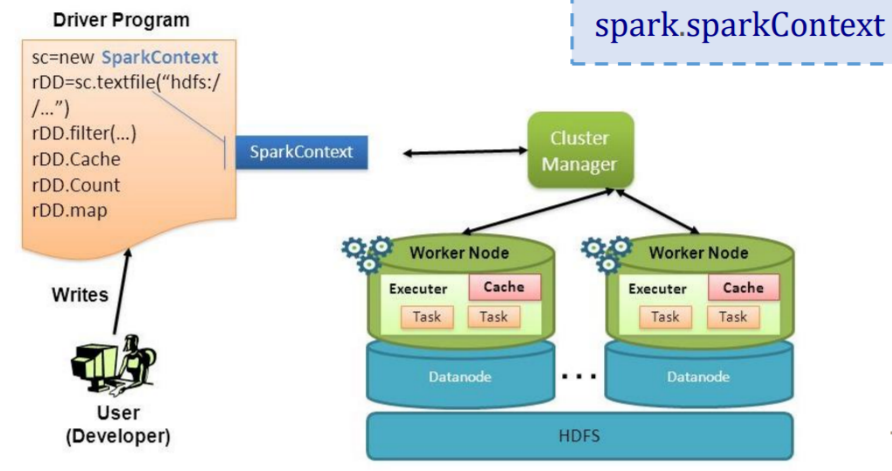
**RDD:** Low-level APIs

Khi nào sử dụng API cấp thấp

• Cần một số chức năng không có sẵn trong API cấp cao hơn Ví dụ: kiểm soát rất chặt chẽ đối với vị trí dữ liệu vật lý trên toàn bộ cụm

• Duy trì một số cơ sở mã kế thừa được viết bằng RDD

• Thực hiện một số thao tác tùy chỉnh với biến được chia sẻ

**SparkContext** là điểm đầu vào cho chức năng API cấp thấp Nó được truy cập thông qua SparkSession, đây là công cụ để tính toán trên một cụm Spark.

RDD đại diện cho một tập hợp các bản ghi được phân vùng, bất biến có thể được vận hành song song

. RDD là API chính trong sê-ri Spark 1.X • Chúng vẫn có sẵn trong 2.X, nhưng không được sử dụng phổ biến

Mạnh mẽ nhưng không phải không có vấn đề tiềm ẩn. ✓ Các đối tượng này có thể lưu trữ bất kỳ thứ gì ở bất kỳ định dạng nào.  Mọi thao tác và tương tác giữa các giá trị phải được xác định bằng tay.  Việc tối ưu hóa sẽ đòi hỏi nhiều công việc thủ công hơn. • Spark không hiểu cấu trúc bên trong của các bản ghi như với các API có cấu trúc

**Các loại RDD:** Có rất nhiều lớp con của RDD. • Hầu hết trong số đó dành cho API DataFrame để tạo kế hoạch thực thi vật lý được tối ưu hóa. • Người dùng có thể sẽ chỉ tạo hai loại RDD: Generic RDD và Key-value RDD • Các RDD khóa-giá trị có các hoạt động đặc biệt cũng như khái niệm về phân vùng tùy chỉnh theo khóa.

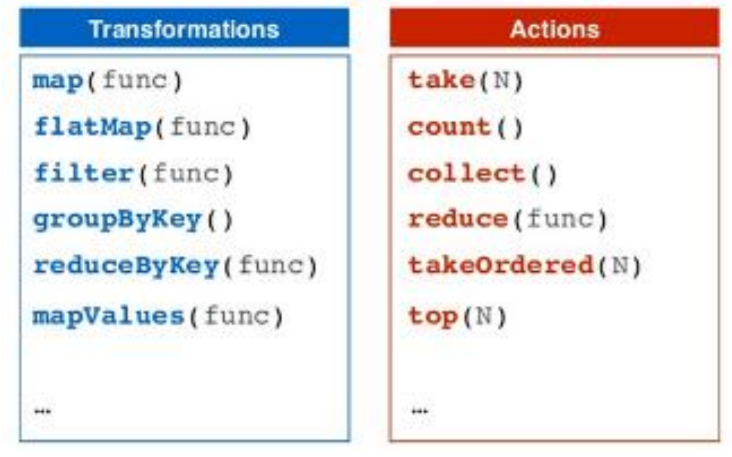
**Thuộc tính của RDDs:**

Required: Danh sách các phân vùng • Một chức năng để tính toán mỗi lần phân tách • Danh sách các phần phụ thuộc vào các RDD khác

Optional: Trình phân vùng cho RDD có khóa-giá trị Ví dụ: để nói rằng RDD được phân vùng theo hàm băm • Danh sách các vị trí ưu tiên để tính toán mỗi lần phân tách Ví dụ: chặn các vị trí cho tệp HDFS

Khả năng lập lịch và thực thi chương trình người dùng của Spark

RDD cung cấp các chuyển đổi và hành động hoạt động giống như trong DataFrames và DataSets. • Đánh giá: transformations lazily – actions eagerly



Các API RDD có sẵn bằng Python, Scala và Java • Scala và Java: hiệu suất giống nhau cho hầu hết các phần • Chi phí lớn phát sinh khi thao tác các đối tượng thô • Python: một lượng lớn hiệu suất bị mất • Chi phí tuần tự hóa giữa quy trình Python và JVM

**Tạo RDD: - Create an RDD from an existing DataFrame or Dataset**

# in Python spark.range(10).rdd

# in Python spark.range(10).toDF("id").rdd.map(lambda row: row[0])

• Scala and Java: from Dataset[T] to RDD[T] E.g., Dataset[Long] → RDD[Long]

• Python: only from DataFrames to RDDs of type Row

**- Inversely, create a DataFrame or Dataset from an RDD**

**# in Python** spark.range(10).rdd.toDF( )

**Tạo RDD từ bộ sưu tập cục bộ:**

**- Use the parallelize method on a SparkContext to turn a single node collection into a parallel collection**

**# in Python** myCollection = "Spark The Definitive Guide : Big Data Processing Made Simple".split(" ").\words = spark.sparkContext.parallelize(myCollection, 2)

**- Name the RDD to show up in the Spark UI**: # in Python

words.setName("myWords") words.name() # myWords

**Tạo RDD từ text files:**

RDD không có khái niệm “API nguồn dữ liệu” • Chúng chủ yếu xác định cấu trúc phụ thuộc và danh sách các phân vùng.

- Each record represents a line in that text file or files\

# in Python spark.sparkContext.textFile("/some/path/withTextFiles")

- The whole text file becomes a single record

• First object: filename, second string object: content

# in Python spark.sparkContext.wholeTextFile("/some/path/withTextFiles")

**Transformations:** Nhiều phiên bản transformation của RDD có thể hoạt động trên các Structured API, transformation xử lý lazily, tức là chỉ giúp dựng execution plans, dữ liệu chỉ được truy xuất thực sự khi thực hiện action

**distinct:** loại bỏ trùng lắp trong RDD

# in Python

# Spark The Definitive Guide : Big Data Processing Made Simple

words.distinct( ).count( ) # 10

**filter:** tương đương với việc sử dụng where trong SQL – tìm các record trong RDD xem những phần tử nào thỏa điều kiện. Có thể cung cấp một hàm phức tạp sử dụng để filter các record cần thiết – Như trong Python, ta có thể sử dụng hàm lambda để truyền vào filter

# in Python def startsWithS(individual): return individual.startswith("S")

# Spark The Definitive Guide : Big Data Processing Made Simple

words.filter(lambda word: startsWithS(word)).collect() # ['Spark', 'Simple']

**map:** thực hiện một công việc nào đó trên toàn bộ RDD. Trong Python sử dụng lambda với từng phần tử để truyền vào map

words2 = words.map(lambda word: (word, word[0], word.startswith("S")))

words2.filter(lambda record: record[2]).take(5) # [('Spark', 'S', True), # ('Simple', 'S', True)]

**flatMap:** cung cấp một hàm đơn giản hơn hàm map. Yêu cầu output của map phải là một structure có thể lặp và mở rộng được.

words.flatMap(lambda word: list(word)).take(5) # S, p, a, r, k

**sortBy**: mô tả một hàm để trích xuất dữ liệu từ các object của RDD và thực hiện sort được từ đó.

words.sortBy(lambda word: len(word) \* -1).take(5) # ['Definitive', 'Processing', 'Simple', 'Spark', 'Guide']

**randomSplit**: nhận một mảng trọng số và tạo một random seed, tách các RDD thành một mảng các RDD có số lượng chia theo trọng số.

fiftyFiftySplit = words.randomSplit([0.5, 0.5])

**Action** thực thi ngay các transformation đã được thiết lập để thu thập dữ liệu về driver để xử lý hoặc ghi dữ liệu xuống các công cụ lưu trữ.

**reduce**: thực hiện hàm reduce trên RDD để thu về 1 giá trị duy nhất

spark.sparkContext.parallelize(range(1, 21)).reduce(lambda x, y: x + y) # 210

def wordLengthReducer(leftWord, rightWord):

if len(leftWord) > len(rightWord):

return leftWord

else:

return rightWord

words.reduce(wordLengthReducer) # Undeterministic: 'Processing’ or Definitive'

**count**: đếm số dòng trong RDD words.count( )

**countApprox**: phiên bản đếm xấp xỉ của count, nhưng phải cung cấp timeout vì có thể không nhận được kết quả:. confidence = 0.95

timeoutMilliseconds = 400

words.countApprox(timeoutMilliseconds, confidence)

**countByValue:** đếm số giá trị của RDD words.countByValue( )

chỉ sử dụng nếu map kết quả nhỏ vì tất cả dữ liệu sẽ được load lên memory của driver để tính toán

chỉ nên sử dụng trong tình huống số dòng nhỏ và số lượng item khác nhau cũng nhỏ.

**countApproxDistinct**: đếm xấp xỉ các giá trị khác nhau

**countByValueApprox:** đếm xấp xỉ các giá trị

**first:** lấy giá trị đầu tiên của dataset

**max và min:** lần lượt lấy giá trị lớn nhất và nhỏ nhất của dataset

spark.sparkContext.parallelize(1 to 20).max()

**take và các method tương tự:** lấy một lượng giá trị từ trong RDD. take sẽ trước hết scan qua một partition và sử dụng kết quả để dự đoán số lượng partition cần phải lấy thêm để thỏa mãn số lượng lấy.

**top và takeOrdered**: top sẽ hiệu quả hơn takeOrdered vì top lấy các giá trị đầu tiên được sắp xếp ngầm trong RDD.

words.take(5) # ['Spark', 'The', 'Definitive', 'Guide', ':’]

words.takeOrdered(5) # [':', 'Big', 'Data', 'Definitive', 'Guide']

words.top(5) # ['The', 'Spark', 'Simple', 'Processing', 'Made']

t**akeSamples**: lấy một lượng giá trị ngẫu nhiên trong RDD

withReplacement = True

numberToTake = 6

randomSeed = 100

words.takeSample(withReplacement, numberToTake, randomSeed)

# ['Data', 'Definitive', 'Data', 'The', 'Definitive', 'Spark’]

# ['The', ':', 'Simple', 'Spark', 'Data', 'Big']

**Một số kỹ thuật đối với RDD**

- **Lưu trữ file:** words.saveAsTextFile("file:/tmp/bookTitle")

Thực hiện ghi vào các file plain-text

Có thể sử dụng các codec nén từ thư viện của Hadoop

codec = "org.apache.hadoop.io.compress.GzipCodec"

words.saveAsTextFile("file:/tmp/bookTitle“, codec)

Lưu trữ vào các database bên ngoài yêu cầu ta phải lặp qua tất cả partition của RDD – Công việc được thực hiện ngầm trong các high-level API

sequenceFile là một flat file chứa các cặp key-value, thường được sử dụng làm định dạng input/output của MapReduce. Spark có thể ghi các sequenceFile bằng các ghi lại các cặp key-value

words.saveAsObjectFile("/tmp/my/sequenceFilePath")

Đồng thời, Spark cũng hỗ trợ ghi nhiều định dạng file khác nhau, cho phép define các class, định dạng output, config và compression scheme của Hadoop.

**- Caching:** Tăng tốc xử lý bằng cache

Caching với RDD, Dataset hay DataFrame có nguyên lý như nhau.

Chúng ta có thể lựa chọn cache hay persist một RDD, và mặc định, chỉ xử lý dữ liệu trong bộ nhớ

words.cache( ) words.persist( )

words.getStorageLevel( ) # StorageLevel(False, False, False, False, 1)

**- Checkpointing:** Lưu trữ lại các bước xử lý để phục hồi

Checkpointing lưu RDD vào đĩa cứng để các tiến trình khác để thể sử dụng lại RDD point này làm partition trung gian thay vì tính toán lại RDD từ các nguồn dữ liệu gốc

Checkpointing cũng tương tự như cache, chỉ khác nhau là lưu trữ vào đĩa cứng và không dùng được trong API của DataFrame

Cần sử dụng nhiều để tối ưu tính toán.

spark.sparkContext.setCheckpointDir("/some/path/for/checkpointing")

words.checkpoint()

WORD COUNT:

text\_file = sc.textFile(“input.txt")

counts = text\_file.flatMap(lambda line: line.split(" ")) \

.map(lambda word: (word, 1)) \

.reduceByKey(lambda a, b: a + b)

count.collect()

from pyspark.sql import SQLContext

sqlContext = SQLContext(sc)

linesDF = sqlContext.read.text("input.txt")

linesDF.show(linesDF.count(),truncate=False)

import pyspark.sql.functions as f

wordsDF = linesDF.withColumn('word',

f.explode(f.split(f.col('value'), ' ')))\

.groupBy('word')\.count()\.sort('count', ascending=False)\.show()

**MLlib** là thư viện máy học (ML) của Spark. Mục tiêu của nó là làm cho việc học máy thực tế có thể mở rộng và dễ dàng. Ở cấp độ cao, nó cung cấp các công cụ như:

Thuật toán ML: các thuật toán học tập phổ biến như phân loại, hồi quy, phân cụm và lọc cộng tác

Lông vũ: trích xuất tính năng, biến đổi, giảm kích thước và lựa chọn

Đường ống: công cụ để xây dựng, đánh giá và điều chỉnh Đường ống ML

Tính bền bỉ: lưu và tải các thuật toán, mô hình và đường ống

Các tiện ích: đại số tuyến tính, thống kê, xử lý dữ liệu, v.v.

Trước khi chúng ta bắt đầu, vui lòng thiết lập môi trường Python và Apache Spark trên máy của bạn. Truy cập blog này tại đây để cài đặt nếu bạn chưa làm như vậy.

Chúng tôi cũng sẽ sử dụng mô-đun MLlib từ Python trong môi trường ảo của chúng tôi sau này được tích hợp sẵn theo mặc định với Spark.

Apache Spark cung cấp một API Học máy được gọi là MLlib . PySpark cũng có API học máy này bằng Python. Nó hỗ trợ các loại thuật toán khác nhau, được đề cập bên dưới:

mllib.classification - Gói spark.mllib hỗ trợ nhiều phương pháp khác nhau để phân loại nhị phân, phân loại đa lớp và phân tích hồi quy. Một số thuật toán phổ biến nhất trong phân loại là Rừng ngẫu nhiên, Vịnh Naive, Cây quyết định , v.v.

mllib.clustering - Clustering là một vấn đề học tập không có giám sát, theo đó bạn nhằm mục đích nhóm các tập con của các thực thể với nhau dựa trên một số khái niệm về sự giống nhau.

mllib.fpm - Đối sánh mẫu thường xuyên là khai thác các mục thường xuyên, tập phổ biến, chuỗi con hoặc các cấu trúc con khác thường nằm trong số các bước đầu tiên để phân tích một tập dữ liệu quy mô lớn. Đây đã là một chủ đề nghiên cứu tích cực trong việc khai thác dữ liệu trong nhiều năm.

mllib.linalg - Tiện ích MLlib cho đại số tuyến tính.

mllib.recommendation - Lọc cộng tác thường được sử dụng cho các hệ thống khuyến nghị. Các kỹ thuật này nhằm mục đích điền vào các mục còn thiếu của ma trận liên kết mục người dùng.

spark.mllib - Nó hiện hỗ trợ lọc cộng tác dựa trên mô hình, trong đó người dùng và sản phẩm được mô tả bằng một tập hợp nhỏ các yếu tố tiềm ẩn có thể được sử dụng để dự đoán các mục nhập bị thiếu. spark.mllib sử dụng thuật toán Bình phương tối thiểu xen kẽ (ALS) để tìm hiểu các yếu tố tiềm ẩn này.

mllib.regression - Hồi quy tuyến tính thuộc họ thuật toán hồi quy. Mục tiêu của hồi quy là tìm mối quan hệ và sự phụ thuộc giữa các biến. Giao diện làm việc với mô hình hồi quy tuyến tính và tóm tắt mô hình tương tự như trường hợp hồi quy logistic.

Spark MLlib được tích hợp chặt chẽ trên Spark giúp giảm bớt sự phát triển của các thuật toán học máy quy mô lớn hiệu quả như thường là lặp đi lặp lại trong tự nhiên.

Cộng đồng mã nguồn mở của Spark đã dẫn đến sự phát triển nhanh chóng và việc áp dụng Spark MLlib. Có hơn 200 cá nhân từ 75 tổ chức cung cấp khoảng hơn 2000 bản vá chỉ riêng cho MLlib.

MLlib dễ triển khai và không yêu cầu cài đặt trước, nếu Hadoop 2 cluster đã được cài đặt và đang chạy.

Khả năng mở rộng, tính đơn giản và khả năng tương thích ngôn ngữ của Spark MLlib (bạn có thể viết ứng dụng bằng Java, Scala và Python) giúp các nhà khoa học dữ liệu giải quyết các vấn đề dữ liệu lặp lại nhanh hơn. Các nhà khoa học dữ liệu có thể tập trung vào các vấn đề dữ liệu quan trọng trong khi tận dụng một cách minh bạch tốc độ, sự dễ dàng và tích hợp chặt chẽ của nền tảng thống nhất của Spark.

MLlib cung cấp hiệu suất cao nhất cho các nhà khoa học dữ liệu và nhanh hơn từ 10 đến 100 lần so với Hadoop và Apache Mahout. Các thuật toán học máy Alternating Least Squares trên Amazon Đánh giá trên tập dữ liệu gồm 660 triệu người dùng, 2,4 triệu mục và xếp hạng 3,5 B chạy trong 40 phút với 50 nút.

**Vector**

from pyspark.ml.linalg import Vectors

denseVec = Vectors.dense(1.0, 2.0, 3.0)

size = 3

idx = [1, 2] # locations of non-zero elements in vector

values = [2.0, 3.0]

sparseVec = Vectors.sparse(size, idx, values)

**Let’s read a synthetic data set and see a sample**

df = spark.read.json("/data/simple-ml")

df.orderBy("value2").show()

**MLlib in action: Application**

train, test = preparedDF.randomSplit([0.7, 0.3])

from pyspark.ml.classification import LogisticRegression

lr = LogisticRegression(labelCol="label",featuresCol="features")

print lr.explainParams()

fittedLR = lr.fit(train)

fittedLR.transform(train).select("label", "prediction").show()

**MLlib in action: Pipelining**

train, test = df.randomSplit([0.7, 0.3])

rForm = RFormula()

lr = LogisticRegression().setLabelCol("label").setFeaturesCol("features")

from pyspark.ml import Pipeline

stages = [rForm, lr]

pipeline = Pipeline().setStages(stages)

**MLlib in action: Training**

from pyspark.ml.tuning import ParamGridBuilder

params = ParamGridBuilder()\

.addGrid(rForm.formula, [

"lab ~ . + color:value1",

"lab ~ . + color:value1 + color:value2"])\

.addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0])\

.addGrid(lr.regParam, [0.1, 2.0])\

.build()

**MLlib in action: Evaluation**

from pyspark.ml.evaluation import BinaryClassificationEvaluator

evaluator = BinaryClassificationEvaluator()\

.setMetricName("areaUnderROC")\

.setRawPredictionCol("prediction")\

.setLabelCol("label")

from pyspark.ml.tuning import TrainValidationSplit

tvs = TrainValidationSplit()\

.setTrainRatio(0.75)\

.setEstimatorParamMaps(params)\

.setEstimator(pipeline)\

.setEvaluator(evaluator)

