

I. Spark properties

1. Tổng quan

Spark cung cấp ba cấu trúc để cấu hình hệ thống:

- Spark properties: Kiểm soát hầu hết các tham số ứng dụng và có thể được đặt bằng cách sử dụng đối tượng SparkConf hoặc thông qua các thuộc tính hệ thống Java.

- Environment variables: Các biến môi trường có thể được sử dụng để đặt cài đặt cho mỗi máy, chẳng hạn như địa chỉ IP, thông qua tập lệnh conf / spark-env.sh trên mỗi nút.

- Logging: Có thể được định cấu hình thông qua *log4j.properties*.

Trong phần đầu tiên của bài viết này, ta sẽ tìm hiểu về cấu trúc cấu hình hệ thống của Spark, được gọi là Spark Properties.

Spark properties có nhiệm vụ kiểm soát hầu hết các cài đặt ứng dụng và được cấu hình riêng cho từng ứng dụng sử dụng Spark. Các thuộc tính này có thể được đặt trực tiếp trên SparkConf và được chuyển tới SparkContext. SparkConf cho phép người dùng định nghĩa cấu hình một số thuộc tính phổ biến (ví dụ: URL chính và appname - tên ứng dụng), cũng như các cặp key-values tùy ý thông qua phương thức set (). Lấy ví dụ đối với chương trình wordCount, chúng ta có thể khởi tạo một ứng dụng với đơn luồng như sau:

```
[ ]  
conf = SparkConf().setMaster('local').setAppName('wordCounting')  
sc = SparkContext.getOrCreate(conf=conf)
```

Các properties chỉ định khoảng thời gian hay kích thước byte phải được cấu hình với một đơn vị thời gian hay đơn vị kích thước trong phạm vi cho phép sử dụng của Spark. Những định dạng về đơn vị kích thước và đơn vị thời gian sau được chấp nhận:

25ms (milliseconds)	1b (bytes)
5s (seconds)	1k or 1kb (kibibytes = 1024 bytes)
10m or 10min (minutes)	1m or 1mb (mebibytes = 1024 kibibytes)
3h (hours)	1g or 1gb (gibibytes = 1024 mebibytes)
5d (days)	1t or 1tb (tebibytes = 1024 gibibytes)
1y (years)	1p or 1pb (pebibytes = 1024 tebibytes)

2. Tải động đối với các thuộc tính của Spark

Trong một số trường hợp, bạn có thể muốn tránh mã hóa cứng các cấu hình nhất định trong SparkConf. Ví dụ: Nếu bạn muốn chạy cùng một ứng dụng với các bản chính khác nhau hoặc số lượng bộ nhớ khác nhau. Spark cho phép bạn chỉ cần tạo một conf trông như sau:

```
val sc = new SparkContext(new SparkConf())
```

Khi đó bạn vẫn có thể cung cấp các giá trị cấu hình trong quá trình runtime bằng cách setting trực tiếp trên command line:

```
./bin/spark-submit --name "My app" --master local[4] --conf spark.eventLog.enabled=false
--conf "spark.executor.extraJavaOptions=-XX:+PrintGCDetails -XX:+PrintGCTimeStamps" myApp.jar
```

Spark shell và spark-submit tool hỗ trợ hai cách để tải cấu hình động. Đầu tiên là các tùy chọn dòng lệnh, chẳng hạn như --master, như hình trên. spark-submit có thể chấp nhận bất kỳ thuộc tính Spark nào sử dụng cờ --conf / -c, nhưng sử dụng cờ đặc biệt cho các thuộc tính đóng một vai trò trong việc khởi chạy ứng dụng Spark. Chạy ./bin/spark-submit --help sẽ hiển thị toàn bộ danh sách các tùy chọn này.

Trong khi đó, bin / spark-submit cũng sẽ đọc các tùy chọn cấu hình từ conf / spark-defaults.conf, trong đó mỗi dòng bao gồm một khóa và một giá trị được phân tách bằng khoảng trắng. Ví dụ:

```
spark.master          spark://5.6.7.8:7077
spark.executor.memory  4g
spark.eventLog.enabled true
spark.serializer       org.apache.spark.serializer.KryoSerializer
```

Mọi giá trị được chỉ định dưới dạng flags hoặc properties file sẽ được chuyển đến ứng dụng và được hợp nhất với những giá trị được chỉ định thông qua SparkConf. Các thuộc tính được đặt trực tiếp trên SparkConf được ưu tiên cao nhất, sau đó các flags được chuyển đến spark-submit hoặc spark-shell, sau đó sẽ là các tùy chọn trong tệp spark-defaults.conf. Một vài khóa cấu hình đã được đổi tên kể từ các phiên bản Spark trước đó; trong trường hợp đó, các tên khóa cũ hơn vẫn được chấp nhận nhưng với sự ưu tiên thấp hơn bất kỳ trường hợp nào của khóa mới hơn.

Các thuộc tính của Spark chủ yếu có thể được chia thành hai loại: một là liên quan đến triển khai, như “spark.driver.memory”, “spark.executor.instances”, loại thuộc tính này có thể không bị ảnh hưởng khi thiết lập lập trình thông qua SparkConf trong thời gian chạy, hoặc hành vi tùy thuộc vào trình quản lý cụm và chế độ triển khai bạn chọn, vì vậy bạn nên đặt thông qua tệp cấu hình hoặc tùy chọn dòng lệnh spark-submit. Một loại khác chủ yếu liên quan đến kiểm soát thời gian chạy Spark, như “spark.task.maxFailures”, loại thuộc tính này có thể được đặt theo một trong hai cách.

3. Tổng hợp và phân loại các thuộc tính trong Spark

Trang web <http://driver:4040> liệt kê các thuộc tính Spark trong tab "Environment". Đây là một nơi hữu ích để kiểm tra và đảm bảo rằng các thuộc tính của bạn đã được sử dụng chính xác. Lưu ý rằng chỉ các giá trị được chỉ định rõ ràng thông qua spark-defaults.conf, SparkConf hoặc dòng lệnh mới xuất hiện. Đối với tất cả các thuộc tính cấu hình khác, bạn có thể cho rằng giá trị mặc định đang được sử dụng.

Có rất nhiều Spark Properties, tùy vào mục đích sử dụng riêng biệt mà các các Spar properties khác nhau. Các bạn có thể truy cập đường link sau để có thể tra cứu đầy đủ và chi tiết nhất từng loại thuộc tính cũng như cách sử dụng và các giá trị mặc định : <https://spark.apache.org/docs/latest/configuration.html#compression-and-serialization>. Trong bài viết này chỉ đề cập đến một số thuộc tính phổ biến và phân loại các thuộc tính đó dựa trên mục đích sử dụng như sau:

- Application Properties: Hầu hết các thuộc tính kiểm soát cài đặt nội bộ đều có giá trị mặc định sẵn. Một số tùy chọn phổ biến nhất có thể kể đến là `spark.app.name`, `spark.driver.cores`, `spark.driver.maxResultSize`, `spark.driver.memory`.

- Runtime Environment: Các thuộc tính này thường được sử dụng trong quá trình runtime để xử lý các biến môi trường. Một số properties phổ biến như `spark.driver.extraClassPath`, `spark.driver.defaultJavaOptions`, `spark.driver.userClassPathFirst`, `spark.executor.extraLibraryPath`, `spark.python.profile`.

- Shuffle Behavior: Các properties này thường ít được sử dụng hơn, nó dùng để cung cấp các thuộc tính về mặt behavior cho chương trình như `spark.reducer.maxSizeInFlight`, `spark.shuffle.compress`, `spark.shuffle.file.buffer`, `spark.shuffle.io.maxRetries`, `spark.shuffle.io.backLog`

- Spark UI: Cung cấp các thuộc tính về giao diện người dùng như `spark.eventLog.logBlockUpdates.enabled`, `spark.eventLog.longForm.enabled`, `spark.eventLog.compress`, `spark.eventLog.dir`, `spark.eventLog.overwrite`, `spark.ui.enabled`, `spark.ui.port`

- Compression and Serialization: Các thuộc tính về nén và tuần tự hóa có thể kể đến `spark.broadcast.compress`, `spark.checkpoint.compress`, `spark.io.compression.codec`.

- Memory Management: Đây là các thuộc tính dùng để quản lý bộ nhớ như `spark.memory.fraction`, `spark.memory.storageFraction`, `spark.memory.offHeap`, `spark.cleaner`

- Execution Behavior: Loại thuộc tính này dùng để thực thi lớp hành vi của chương trình, có thể kể đến `spark.broadcast.blockSize`, `spark.broadcast.checksum`, `spark.default.parallelism`.

- Executor Metrics: Các thuộc tính này để hiển thị các chỉ số, bao gồm 3 thuộc tính sau: `spark.eventLog.logStageExecutorMetrics`, `spark.executor.processTreeMetrics.enabled` và `spark.executor.metrics.pollingInterval`

- Networking: Dùng trong các trường hợp liên quan đến mạng như `spark.rpc.message.maxSize`, `spark.blockManager.port`, `spark.driver.bindAddress`, `spark.driver.host`

- Scheduling: Xử lý các tiến trình, bao gồm các `spark.cores.max`, `spark.locality.wait`, `spark.scheduler.mode`, `spark.scheduler.revive.interval`

- Barrier Execution Mode: Gồm 3 thuộc tính day nhất: `spark.barrier.sync.timeout`, `spark.scheduler.barrier.maxConcurrentTasksCheck.interval` và `spark.scheduler.barrier.maxConcurrentTasksCheck.maxFailures`

- Dynamic Allocation: `spark.dynamicAllocation.enabled`,
`spark.dynamicAllocation.executorIdleTimeout`,
`spark.dynamicAllocation.executorAllocationRatio`

- Thread Configurations: Tùy thuộc vào công việc và cấu hình cụm, chúng ta có thể đặt số lượng luồng ở một số vị trí trong Spark để sử dụng hiệu quả các tài nguyên có sẵn nhằm đạt được hiệu suất tốt hơn. Trước Spark 3.0, các cấu hình luồng này áp dụng cho tất cả các vai trò của Spark, chẳng hạn như trình điều khiển, người thực thi, công nhân và chủ. Từ Spark 3.0, chúng ta có thể định cấu hình các luồng ở mức độ chi tiết tốt hơn bắt đầu từ trình điều khiển và trình thực thi. Lấy mô-đun RPC làm ví dụ trong bảng dưới đây. Đối với các mô-đun khác, chẳng hạn như xáo trộn, chỉ cần thay thế “rpc” bằng “xáo trộn” trong tên thuộc tính ngoại trừ `spark. {Driver | executive} .rpc.netty.dispatcher.numThreads`, chỉ dành cho mô-đun RPC.

- Security

- Spark SQL: Đối với properties này, ta có thể chia làm hai loại như sau:

- + Runtime SQL Configuration: là cấu hình Spark SQL cho mỗi phiên, có thể thay đổi. Chúng có thể được đặt với các giá trị ban đầu bằng tệp cấu hình và các tùy chọn dòng lệnh có tiền tố `--conf / -c` hoặc bằng cách đặt `SparkConf` được sử dụng để tạo `SparkSession`. Ngoài ra, chúng có thể được đặt và truy vấn bằng lệnh `SET` và đặt chúng về giá trị ban đầu bằng lệnh `RESET` hoặc bằng các phương thức setter và getter của `SparkSession.conf` trong thời gian chạy.

- + Static SQL Configuration: là các cấu hình Spark SQL xuyên phiên, bất biến. Chúng có thể được đặt với các giá trị cuối cùng bằng tệp cấu hình và các tùy chọn dòng lệnh có tiền tố `--conf / -c` hoặc bằng cách đặt `SparkConf` được sử dụng để tạo `SparkSession`. Người dùng bên ngoài có thể truy vấn các giá trị cấu hình sql tĩnh qua `SparkSession.conf` hoặc thông qua lệnh `set`, ví dụ: Đặt `spark.sql.extensions ;`, nhưng không thể đặt / bỏ thiết lập chúng.

- Spark Streaming

- SparkR

- GraphX

- Deploy

- Cluster Managers: Mỗi trình quản lý cụm trong Spark có các tùy chọn cấu hình bổ sung:

- + YARN

- + Mesos

- + Kubernetes

- + Standalone Mode

II. Spark RDD

1. Tổng quan

Ở các cấp độ cao, mọi ứng dụng của Spark bao gồm một chương trình trình điều khiển chạy chức năng chính của người dùng và thực hiện các hoạt động song song khác nhau trên một cụm. Tính trừu tượng chính mà Spark cung cấp là tập dữ liệu phân tán có khả năng phục hồi (RDD), là tập hợp các phần tử được phân vùng trên các nút của cụm có thể hoạt động song song. RDD được tạo bằng cách bắt đầu bằng một tập trong hệ thống tệp Hadoop (hoặc bất kỳ hệ thống tệp nào khác được Hadoop hỗ trợ) hoặc một bộ sưu tập Scala hiện có trong chương trình trình điều khiển và chuyển đổi nó. Người dùng cũng có thể yêu cầu Spark duy trì một RDD trong bộ nhớ, cho phép nó được sử dụng lại một cách hiệu quả trong các hoạt động song song. Cuối cùng, các RDD tự động phục hồi sau các lỗi của nút.

Sự trừu tượng thứ hai trong Spark là các biến được chia sẻ có thể được sử dụng trong các hoạt động song song. Theo mặc định, khi Spark chạy song song một hàm dưới dạng một tập hợp các tác vụ trên các nút khác nhau, nó sẽ gửi một bản sao của từng biến được sử dụng trong hàm cho mỗi tác vụ. Đôi khi, một biến cần được chia sẻ giữa các tác vụ hoặc giữa các tác vụ và chương trình điều khiển. Spark hỗ trợ hai loại biến chia sẻ: biến quảng bá, có thể được sử dụng để lưu trữ một giá trị trong bộ nhớ trên tất cả các nút và bộ tích lũy, là những biến chỉ được “thêm” vào, chẳng hạn như bộ đếm và tổng.

Resilient Distributed Datasets (RDD) là một cấu trúc dữ liệu cơ bản của Spark. Nó là một tập hợp bất biến phân tán của một đối tượng. Mỗi dataset trong RDD được chia ra thành nhiều phần vùng logical. Có thể được tính toán trên các node khác nhau của một cụm máy chủ (cluster).

RDDs có thể chứa bất kỳ kiểu dữ liệu nào của Python, Java, hoặc đối tượng Scala, bao gồm các kiểu dữ liệu do người dùng định nghĩa. RDD chỉ cho phép đọc và phân mục tập hợp của các bản ghi. RDDs có thể được tạo ra qua điều khiển xác định trên dữ liệu trong bộ nhớ hoặc RDDs, RDD là một tập hợp có khả năng chịu lỗi mỗi thành phần có thể được tính toán song song.

2. Cài đặt Spark

Spark 3.0.1 hoạt động với Python 2.7+ hoặc Python 3.4+. Nó có thể sử dụng trình thông dịch CPython tiêu chuẩn, vì vậy có thể sử dụng các thư viện C như NumPy. Nó cũng hoạt động với PyPy 2.3+.

Các ứng dụng Spark trong Python có thể được chạy bằng tập lệnh bin / spark-submit bao gồm Spark khi chạy hoặc bằng cách đưa nó vào setup.py của bạn dưới dạng:

```
install_requires=[
    'pyspark=={site.SPARK_VERSION}'
]
```

Để chạy các ứng dụng Spark bằng Python mà không cần gõ câu lệnh pip cài đặt PySpark, hãy sử dụng tập lệnh bin / spark-submit nằm trong thư mục Spark. Tập lệnh này sẽ tải các thư viện Java / Scala của Spark và cho phép bạn gửi ứng dụng đến một cụm. Bạn cũng có thể sử dụng bin / pyspark để khởi chạy một trình bao Python tương tác.

Nếu bạn muốn truy cập dữ liệu HDFS, bạn cần sử dụng một bản dựng của PySpark liên kết với phiên bản HDFS của bạn. Các gói dựng sẵn cũng có sẵn trên trang chủ Spark cho các phiên bản HDFS phổ biến. Cuối cùng, bạn cần nhập một số lớp Spark vào chương trình của mình bằng câu lệnh dưới đây:

```
from pyspark import SparkContext, SparkConf
```

3. Khởi tạo Spark

Như đã trình bày ở chương I, điều đầu tiên mà chương trình Spark phải làm là tạo một đối tượng SparkContext, đối tượng này cho Spark biết cách truy cập một cụm. Để tạo SparkContext, trước tiên bạn cần xây dựng một đối tượng SparkConf chứa thông tin về ứng dụng của bạn:

```
conf = SparkConf().setAppName(appName).setMaster(master)
sc = SparkContext(conf=conf)
```

Tham số appName là tên để ứng dụng của bạn hiển thị trên giao diện người dùng cụm. Chính là URL cụm Spark, Mesos hoặc YARN hoặc một chuỗi “cục bộ” đặc biệt để chạy ở chế độ cục bộ. Trên thực tế, khi chạy trên một cụm, bạn sẽ không

muốn hardcode master trong chương trình mà phải khởi chạy ứng dụng bằng spark-submit và nhận nó ở đó. Tuy nhiên, đối với thử nghiệm cục bộ và thử nghiệm đơn vị, bạn có thể vượt qua “cục bộ” để chạy Spark trong quá trình..

4. Resilient Distributed Datasets (RDDs)

Như đã đề cập ở phần tổng quan, Spark hoạt động dựa trên khái niệm về tập dữ liệu phân tán có khả năng phục hồi (RDD). Đây là một tập hợp các phần tử có khả năng chịu lỗi và có thể hoạt động song song. Có hai cách để tạo RDD: song song một tập hợp hiện có trong chương trình trình điều khiển của bạn hoặc tham chiếu tập dữ liệu trong hệ thống lưu trữ bên ngoài, chẳng hạn như hệ thống tệp được chia sẻ, HDFS, HBase hoặc bất kỳ nguồn dữ liệu nào cung cấp Hadoop InputFormat.

4.1 Parallelized Collections

Bộ sưu tập song song được tạo bằng cách gọi phương thức song song của SparkContext trên một bộ sưu tập hoặc bộ sưu tập có thể lặp lại hiện có trong chương trình trình điều khiển của bạn. Các phần tử của bộ sưu tập được sao chép để tạo thành một tập dữ liệu phân tán có thể hoạt động song song. Ví dụ: đây là cách tạo một tập hợp song song chứa các số từ 1 đến 5:

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

Sau khi được tạo, tập dữ liệu phân tán (distData) có thể được vận hành song song. Ví dụ, chúng ta có thể gọi distData.reduce (lambda a, b: a + b) để thêm các phần tử của danh sách. Chúng tôi mô tả các hoạt động trên tập dữ liệu phân tán sau này.

Một tham số quan trọng đối với các tập hợp song song là số lượng phân vùng để cắt tập dữ liệu vào. Spark sẽ chạy một tác vụ cho mỗi phân vùng của cụm. Thông thường, bạn muốn 2-4 phân vùng cho mỗi CPU trong cụm của mình. Thông thường, Spark cố gắng đặt số lượng phân vùng tự động dựa trên cụm của bạn. Tuy nhiên, bạn cũng có thể đặt nó theo cách thủ công bằng cách chuyển nó làm tham số thứ hai để song song hóa (ví dụ: sc.parallelize (data, 10)). Lưu ý: một số nơi trong mã sử dụng thuật ngữ lát (một từ đồng nghĩa với phân vùng) để duy trì khả năng tương thích ngược.

4.2 External Datasets

PySpark có thể tạo tập dữ liệu phân tán từ bất kỳ nguồn lưu trữ nào được Hadoop hỗ trợ, bao gồm hệ thống tệp cục bộ của bạn, HDFS, Cassandra, HBase, Amazon S3, v.v. Spark hỗ trợ tệp văn bản, SequenceFiles và bất kỳ Hadoop InputFormat nào khác.

Các RDD của tệp văn bản có thể được tạo bằng cách sử dụng phương thức `textFile` của `SparkContext`. Phương thức này lấy một URI cho tệp (đường dẫn cục bộ trên máy hoặc URI hdfs: //, s3a: //, v.v.) và đọc nó như một tập hợp các dòng. Đây là một lời gọi ví dụ:

```
>>> distFile = sc.textFile("data.txt")
```

Sau khi được tạo, `distFile` có thể được thực hiện bằng các hoạt động của tập dữ liệu. Ví dụ, chúng ta có thể cộng kích thước của tất cả các dòng bằng cách sử dụng bản đồ và các phép toán giảm như sau: `distFile.map (lambda s: len (s)). Reduce (lambda a, b: a + b)`.

Một số lưu ý khi đọc tệp với Spark:

- Nếu sử dụng một đường dẫn trên hệ thống tệp cục bộ, tệp cũng phải có thể truy cập được tại cùng một đường dẫn trên các nút công nhân. Sao chép tệp cho tất cả công nhân hoặc sử dụng hệ thống tệp chia sẻ được gắn kết trên mạng.
- Tất cả các phương thức nhập dựa trên tệp của Spark, bao gồm cả `textFile`, đều hỗ trợ chạy trên thư mục, tệp nén và cả ký tự đại diện. Ví dụ: bạn có thể sử dụng `textFile ('/ my / directory')`, `textFile ('/ my / directory / *. Txt')` và `textFile ('/ my / directory / *. Gz')`.
- Phương thức `textFile` cũng có đối số thứ hai tùy chọn để kiểm soát số lượng phân vùng của tệp. Theo mặc định, Spark tạo một phân vùng cho mỗi khối của tệp (các khối là 128MB theo mặc định trong HDFS), nhưng bạn cũng có thể yêu cầu số lượng phân vùng cao hơn bằng cách chuyển một giá trị lớn hơn. Lưu ý rằng bạn không thể có ít phân vùng hơn khối.

Ngoài các tệp văn bản, API Python của Spark cũng hỗ trợ một số định dạng dữ liệu khác:

- `SparkContext.wholeTextFiles` cho phép bạn đọc một thư mục chứa nhiều tệp văn bản nhỏ và trả về từng tệp dưới dạng cặp (tên tệp, nội dung).

Điều này trái ngược với `textFile`, nó sẽ trả về một bản ghi trên mỗi dòng trong mỗi tệp.

- `RDD.saveAsPickleFile` và `SparkContext.pickleFile` hỗ trợ lưu RDD ở định dạng đơn giản bao gồm các đối tượng Python được chọn lọc. Lô hàng được sử dụng trong tuần tự hóa đưa chưa, với kích thước lô mặc định là 10.

- Định dạng đầu vào / đầu ra của `SequenceFile` và Hadoop

PySpark `SequenceFile` hỗ trợ tải RDD của các cặp khóa-giá trị bên trong Java, chuyển đổi Writables thành các kiểu Java cơ sở và chọn các đối tượng Java kết quả bằng cách sử dụng Pyrolite. Khi lưu RDD của các cặp khóa-giá trị vào `SequenceFile`, PySpark thực hiện ngược lại. Nó giải nén các đối tượng Python thành các đối tượng Java và sau đó chuyển đổi chúng thành Writables. Các Writables sau được tự động chuyển đổi.

Mảng không được xử lý theo kiểu out-of-the-box. Người dùng cần chỉ định các kiểu phụ ArrayW ghi tùy chỉnh khi đọc hoặc ghi. Khi viết, người dùng cũng cần chỉ định bộ chuyển đổi tùy chỉnh chuyển đổi mảng thành kiểu con ArrayW ghi tùy chỉnh. Khi đọc, trình chuyển đổi mặc định sẽ chuyển đổi các kiểu phụ ArrayW ghi tùy chỉnh thành Đối tượng Java [], sau đó được chuyển thành các bộ giá trị Python. Để lấy `array.array` trong Python cho các mảng kiểu nguyên thủy, người dùng cần chỉ định bộ chuyển đổi tùy chỉnh.

Tương tự như các tệp văn bản, `SequenceFiles` có thể được lưu và tải bằng cách chỉ định đường dẫn. Các lớp khóa và giá trị có thể được chỉ định, nhưng đối với Writables tiêu chuẩn, điều này không bắt buộc.

```
>>> rdd = sc.parallelize(range(1, 4)).map(lambda x: (x, "a" * x))
>>> rdd.saveAsSequenceFile("path/to/file")
>>> sorted(sc.sequenceFile("path/to/file").collect())
[(1, u'a'), (2, u'aa'), (3, u'aaa')]
```

4.3 RDD Operations

RDD hỗ trợ hai loại hoạt động: biến đổi, tạo ra một tập dữ liệu mới từ một tập dữ liệu hiện có và các hành động, trả về một giá trị cho chương trình trình điều khiển sau khi chạy một tính toán trên tập dữ liệu. Ví dụ, bản đồ là một phép biến đổi chuyển từng phần tử tập dữ liệu qua một hàm và trả về một RDD mới đại diện cho kết quả. Mặt khác, Reduce là một hành động tổng hợp tất cả các phần tử của RDD bằng cách

sử dụng một số chức năng và trả về kết quả cuối cùng cho chương trình điều khiển (mặc dù cũng có một hàm `ReduceByKey` song song trả về một tập dữ liệu phân tán).

Tất cả các phép biến đổi trong Spark đều lười biếng, ở chỗ chúng không tính toán ngay kết quả của chúng. Thay vào đó, họ chỉ nhớ các phép biến đổi được áp dụng cho một số tập dữ liệu cơ sở (ví dụ: một tệp). Các phép biến đổi chỉ được tính khi một hành động yêu cầu kết quả được trả về chương trình điều khiển. Thiết kế này giúp Spark chạy hiệu quả hơn. Ví dụ, chúng ta có thể nhận ra rằng một tập dữ liệu được tạo thông qua bản đồ sẽ được sử dụng để giảm và chỉ trả về kết quả của việc giảm tới trình điều khiển, thay vì tập dữ liệu được ánh xạ lớn hơn.

Theo mặc định, mỗi RDD đã chuyển đổi có thể được tính toán lại mỗi khi bạn chạy một hành động trên đó. Tuy nhiên, bạn cũng có thể duy trì một RDD trong bộ nhớ bằng cách sử dụng phương thức dai dẳng (hoặc bộ nhớ cache), trong trường hợp đó Spark sẽ giữ các phần tử xung quanh trên cụm để truy cập nhanh hơn nhiều vào lần tiếp theo bạn truy vấn nó. Ngoài ra còn có hỗ trợ cho các RDD lâu dài trên đĩa hoặc được sao chép qua nhiều nút.

Để minh họa những điều cơ bản về RDD, hãy xem xét chương trình đơn giản dưới đây:

```
lines = sc.textFile("data.txt")
lineLengths = lines.map(lambda s: len(s))
totalLength = lineLengths.reduce(lambda a, b: a + b)
```

Dòng đầu tiên xác định một RDD cơ sở từ một tệp bên ngoài. Tập dữ liệu này không được tải trong bộ nhớ hoặc không được hoạt động trên: các dòng chỉ là một con trỏ đến tệp. Dòng thứ hai xác định `lineLengths` là kết quả của việc chuyển đổi bản đồ. Một lần nữa, `lineLengths` không được tính toán ngay lập tức, do sự lười biếng. Cuối cùng, chúng tôi chạy giảm, đó là một hành động. Tại thời điểm này, Spark chia nhỏ tính toán thành các tác vụ để chạy trên các máy riêng biệt và mỗi máy chạy cả phần bản đồ và phần giảm cục bộ, chỉ trả lại câu trả lời cho chương trình điều khiển.

Trong khi hầu hết các hoạt động của Spark hoạt động trên RDD có chứa bất kỳ loại đối tượng nào, một vài hoạt động đặc biệt chỉ khả dụng trên RDD của các cặp khóa-giá trị. Các thao tác phổ biến nhất là các thao tác “xáo trộn” được phân phối, chẳng hạn như nhóm hoặc tổng hợp các phần tử bằng một khóa.

Trong Python, các hoạt động này hoạt động trên RDD có chứa các bộ giá trị Python được tích hợp sẵn như (1, 2). Đơn giản chỉ cần tạo các bộ giá trị như vậy và sau đó gọi hoạt động mong muốn của bạn.

Ví dụ: đoạn mã sau sử dụng thao tác ReduceByKey trên các cặp khóa-giá trị để đếm số lần mỗi dòng văn bản xuất hiện trong một tệp:

```
lines = sc.textFile("data.txt")
pairs = lines.map(lambda s: (s, 1))
counts = pairs.reduceByKey(lambda a, b: a + b)
```

Chúng ta cũng có thể sử dụng counts.sortByKey () để sắp xếp các cặp theo thứ tự bảng chữ cái, và cuối cùng counts.collect () để đưa chúng trở lại chương trình điều khiển dưới dạng danh sách các đối tượng.

4.4 RDD Persistence

Một trong những khả năng quan trọng nhất trong Spark là duy trì (hoặc lưu vào bộ nhớ đệm) một tập dữ liệu trong bộ nhớ qua các hoạt động. Khi bạn duy trì một RDD, mỗi nút lưu trữ bất kỳ phân vùng nào của nó mà nó tính toán trong bộ nhớ và sử dụng lại chúng trong các hành động khác trên tập dữ liệu đó (hoặc các tập dữ liệu bắt nguồn từ nó). Điều này cho phép các hành động trong tương lai nhanh hơn nhiều (thường gấp hơn 10 lần). Bộ nhớ đệm là một công cụ chính cho các thuật toán lặp đi lặp lại và sử dụng tương tác nhanh.

Bạn có thể đánh dấu một RDD sẽ được duy trì bằng cách sử dụng các phương thức Kiên trì () hoặc cache () trên đó. Lần đầu tiên nó được tính toán trong một hành động, nó sẽ được lưu trong bộ nhớ trên các nút. Bộ nhớ cache của Spark có khả năng chịu lỗi - nếu bất kỳ phân vùng nào của RDD bị mất, nó sẽ tự động được tính toán lại bằng cách sử dụng các phép biến đổi đã tạo ra ban đầu.

Ngoài ra, mỗi RDD tồn tại có thể được lưu trữ bằng cách sử dụng một mức lưu trữ khác nhau, chẳng hạn như cho phép bạn duy trì tập dữ liệu trên đĩa, duy trì nó trong bộ nhớ nhưng dưới dạng các đối tượng Java được tuần tự hóa (để tiết kiệm dung lượng), sao chép nó qua các nút. Các cấp độ này được thiết lập bằng cách chuyển một đối tượng StorageLevel (Scala, Java, Python) sang Persext (). Phương thức cache () là một cách viết tắt để sử dụng mức lưu trữ mặc định, đó là StorageLevel.MEMORY_ONLY (lưu trữ các đối tượng được giải phóng trong bộ

nhớ). Tập hợp đầy đủ các mức lưu trữ bạn có thể tham khảo theo đường link: <https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-persistence>.

Lưu ý rằng trong Python, các đối tượng được lưu trữ sẽ luôn được tuần tự hóa với thư viện Pickle, vì vậy việc bạn chọn mức độ tuần tự hóa không quan trọng. Các cấp bộ nhớ khả dụng trong Python bao gồm MEMORY_ONLY, MEMORY_ONLY_2, MEMORY_AND_DISK, MEMORY_AND_DISK_2, DISK_ONLY và DISK_ONLY_2.

Spark cũng tự động lưu giữ một số dữ liệu trung gian trong các hoạt động xáo trộn (ví dụ: ReduceByKey), ngay cả khi người dùng không gọi vẫn tiếp tục. Điều này được thực hiện để tránh tính toán lại toàn bộ dữ liệu đầu vào nếu một nút bị lỗi trong quá trình trộn. Chúng tôi vẫn khuyến nghị người dùng tiếp tục gọi RDD kết quả nếu họ định sử dụng lại nó.

III. Spark Dataframes

1. Tổng quan

Trong Spark, DataFrames là tập hợp dữ liệu phân tán, được tổ chức thành các hàng và cột. Mỗi cột trong DataFrame có tên và kiểu liên kết. DataFrames tương tự như các bảng cơ sở dữ liệu truyền thống, được cấu trúc và ngắn gọn. Có thể nói rằng DataFrames là cơ sở dữ liệu quan hệ với các kỹ thuật tối ưu hóa tốt hơn.

Spark DataFrames có thể được tạo từ nhiều nguồn khác nhau, chẳng hạn như bảng Hive, bảng nhật ký, cơ sở dữ liệu bên ngoài hoặc RDD hiện có. DataFrames cho phép xử lý một lượng lớn dữ liệu.

2. Sử dụng DataFrames bổ sung cho RDD trong Spark

Khi Apache Spark 1.3 ra mắt, nó đi kèm với một API mới có tên là DataFrames giúp giải quyết các hạn chế về hiệu suất và khả năng mở rộng xảy ra trong khi sử dụng RDD.

Khi không có nhiều không gian lưu trữ trong bộ nhớ hoặc trên đĩa, các RDD sẽ không hoạt động bình thường khi chúng cạn kiệt. Bên cạnh đó, Spark RDD không có

khái niệm về lược đồ — cấu trúc của một cơ sở dữ liệu xác định các đối tượng của nó. RDD lưu trữ cả dữ liệu có cấu trúc và không có cấu trúc cùng nhau, điều này không hiệu quả lắm.

RDD không thể sửa đổi hệ thống theo cách để nó chạy hiệu quả hơn. Các RDD không cho phép chúng tôi gỡ lỗi trong thời gian chạy. Chúng lưu trữ dữ liệu dưới dạng một tập hợp các đối tượng Java.

Các RDD sử dụng kỹ thuật tuần tự hóa (chuyển đổi một đối tượng thành một dòng byte để cho phép xử lý nhanh hơn) và thu gom rác (một kỹ thuật quản lý bộ nhớ tự động phát hiện các đối tượng không sử dụng và giải phóng chúng khỏi bộ nhớ). Điều này làm tăng chi phí trên bộ nhớ của hệ thống vì chúng rất dài.

Đây là khi Spark DataFrames được giới thiệu để khắc phục những hạn chế mà Spark RDD có. Bây giờ, điều gì làm cho Spark DataFrames trở nên độc đáo? Hãy cùng xem các tính năng của Spark DataFrames khiến chúng trở nên phổ biến.

3. Tính năng chính của DataFrames

Một số tính năng độc đáo của DataFrames là:

- Sử dụng các công cụ tối ưu hóa input: DataFrames sử dụng các công cụ tối ưu hóa đầu vào, ví dụ: Trình tối ưu hóa xúc tác, để xử lý dữ liệu hiệu quả. Chúng ta có thể sử dụng cùng một công cụ cho tất cả các API Python, Java, Scala và R DataFrame.
- Xử lý dữ liệu có cấu trúc: DataFrames cung cấp một cái nhìn sơ đồ về dữ liệu. Ở đây, dữ liệu có một số ý nghĩa đối với nó khi nó đang được lưu trữ.
- Quản lý bộ nhớ tùy chỉnh: Trong RDD, dữ liệu được lưu trữ trong bộ nhớ, trong khi DataFrames lưu trữ dữ liệu ngoài đồng (bên ngoài không gian chính của Java Heap, nhưng vẫn bên trong RAM), do đó làm giảm quá tải bộ sưu tập rác.
- Tính linh hoạt: DataFrames, giống như RDD, có thể hỗ trợ nhiều định dạng dữ liệu khác nhau, chẳng hạn như CSV, Cassandra, v.v.
- Khả năng mở rộng: DataFrames có thể được tích hợp với nhiều công cụ Big Data khác và chúng cho phép xử lý megabyte đến petabyte dữ liệu cùng một lúc.

4. Khởi tạo DataFrames

Có nhiều cách để tạo DataFrames, trong đó có 3 phương pháp dưới đây là phổ biến nhất:

4.1 Khởi tạo DataFrames từ tập tin JSON

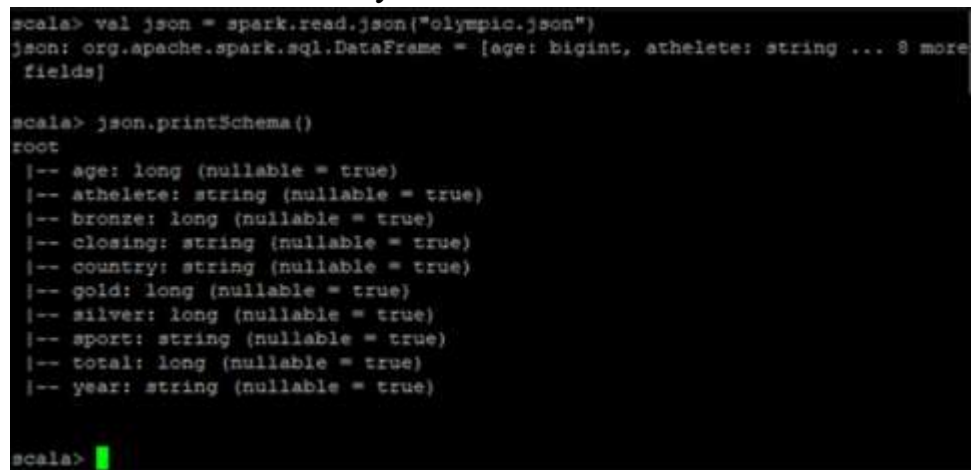
JSON là viết tắt của JavaScript Object Notation, là một loại tệp lưu trữ các đối tượng cấu trúc dữ liệu đơn giản ở định dạng .json. Nó chủ yếu được sử dụng để truyền dữ liệu giữa các máy chủ Web.

Khi nói đến Spark, các tệp .json đang được tải không phải là tệp .json điển hình. Chúng ta không thể tải tệp JSON bình thường vào DataFrame. Tệp JSON mà chúng ta tải phải ở định dạng được cung cấp bên dưới:



```
{
  "id" : "1201",
  "name" : "satish",
  "age" : "25"
}
{
  "id" : "1202",
  "name" : "krishna",
  "age" : "28"
}
```

Các tệp JSON có thể được tải lên DataFrames bằng cách sử dụng hàm read.JSON, với tên tệp mà chúng tôi muốn tải lên. Ví dụ, chúng ta đang tải bảng đếm huy chương Olympic lên DataFrame. Tổng cộng có 10 trường. Hàm printSchema () in ra lược đồ của DataFrame như dưới đây



```
scala> val json = spark.read.json("olympic.json")
json: org.apache.spark.sql.DataFrame = [age: bigint, athlete: string ... 8 more fields]

scala> json.printSchema()
root
 |-- age: long (nullable = true)
 |-- athlete: string (nullable = true)
 |-- bronze: long (nullable = true)
 |-- closing: string (nullable = true)
 |-- country: string (nullable = true)
 |-- gold: long (nullable = true)
 |-- silver: long (nullable = true)
 |-- sport: string (nullable = true)
 |-- total: long (nullable = true)
 |-- year: string (nullable = true)

scala>
```


4.2 Khởi tạo DataFrames từ RDDs đã tạo sẵn

DataFrames cũng có thể được tạo từ các RDD hiện có. Đầu tiên, chúng ta tạo một RDD và sau đó tải RDD đó vào một DataFrame bằng cách sử dụng hàm `createDataFrame (Name_of_the_rdd_file)`.

Trong hình dưới đây, trước tiên chúng ta đang tạo một RDD, chứa các số từ 1 đến 10 và các hình khối của chúng. Sau đó, chúng tôi sẽ tải RDD đó vào DataFrame.

```
scala> val rdd1 = sc.parallelize(1 to 10).map(x=>(x,x*x*x))
rdd1: org.apache.spark.rdd.RDD[(Int, Int)] = MapPartitionsRDD[7] at map at <console>:24

scala> val dataframe1 = spark.createDataFrame(rdd1).toDF("key", "cube")
dataframe1: org.apache.spark.sql.DataFrame = [key: int, cube: int]

scala> dataframe1.show()
+----+-----+
|key|cube|
+----+-----+
| 1| 1|
| 2| 8|
| 3| 27|
| 4| 64|
| 5| 125|
| 6| 216|
| 7| 343|
| 8| 512|
| 9| 729|
| 10|1000|
+----+-----+

scala>
```

4.3 Khởi tạo DataFrames từ tập tin CSV

Chúng ta cũng có thể tạo DataFrames bằng cách tải các tập tin .csv. Đây là một ví dụ về tải tập tin .csv lên DataFrame

```
scala> val emp_salary = spark.read.format("csv").option("header", "true").load("emp_salaries.csv")
emp_salary: org.apache.spark.sql.DataFrame = [Name: string, Unit: string ... 3 more fields]

scala> emp_salary.show
+-----+-----+-----+-----+-----+
| Name| Unit| Dept| Job Title| Salary|
+-----+-----+-----+-----+-----+
| Jones, Marcus D.| CTMGR| City Manager's Of...| City Manager| 318000|
| Hagemann, Robert E.| ATTOR| City Attorney| City Attorney| 248491.82|
| Lewis Jr, John M.| CATS| Executive| Transit Director| 245924.3|
| Cagle, Brent D.| AVIA| Admin - Executive| Aviation Director| 236042.01|
| Joy-Hogg, Sabrina ...| CTMGR| City Manager's Of...| Deputy City Manager| 225500|
| Campbell, Debra D.| CTMGR| City Manager's Of...| Assistant City Ma...| 207574.1|
| Stovall, Jeffrey W.| I&T| I&T Administration| Chief Information...| 203541.04|
| Putney, Kerr Y.| POL| Office of the Chi...| Police Chief| 198286|
| Eagle, Kimberly S.| CTMGR| City Manager's Of...| Assistant City Ma...| 197600|
| Jaiyeoba, Taiwo| PLAN| Planning| Planning Director| 197000|
| Harrington, Randy J.| M&FS| Administrative Ma...| Chief Financial O...| 196945.34|
| Pleasant, Danny Craig| CTMGR| City Manager's Of...| Assistant City Ma...| 195000|
| Mush, John M.| CATS| Development Admin...| Transit Dep Dir o...| 194202.06|
| Campbell, Robert D.| M&FS| Finance Administr...| Finance Director| 182965.57|
| Johnson, Carolyn D.| ATTOR| City Attorney| Senior Deputy Cit...| 182060.15|
| Christine Jr, John L.| AVIA| Admin - Executive| Dep Aviation Dire...| 180000.46|
| Mumford, Patrick T...| HR| Human Resources| Business Services...| 176491.28|
| Lee, Angela C.| CLTWT| CLT Water Admin ~...| Utilities Director| 175800|
| Smith III, Allen C...| CATS| Light Rail Admini...| Transit Rail Oper...| 175100|
| Johnson, Victoria ...| SW| SWS Department Head| Solid Waste Servi...| 174580.74|
+-----+-----+-----+-----+-----+
only showing top 20 rows

scala>
```

5. Làm việc với DataFrames

5.1 Union hai DataFrames

Để nối hai DataFrames với nhau, ta sử dụng câu lệnh sau:

```
unionDF = df1.union(df2)
display(unionDF)
```

5.2 Ghi DataFrame hợp nhất vào tệp Parquet

```
# Remove the file if it exists
dbutils.fs.rm("/tmp/databricks-df-example.parquet", True)
unionDF.write.parquet("/tmp/databricks-df-example.parquet")
```

5.3 Đọc DataFrame từ tệp Parquet

```
parquetDF = spark.read.parquet("/tmp/databricks-df-example.parquet")
display(parquetDF)
```

5.4 Explode cột trong DataFrames

Ta lấy ví dụ cần giải phóng cột employees như sau:

```
from pyspark.sql.functions import explode
```

```
explodeDF = unionDF.select(explode("employees").alias("e"))
flattenDF = explodeDF.selectExpr("e.firstName", "e.lastName", "e.email", "e.salary")

flattenDF.show()
```

Kết quả là:

firstName	lastName	email	salary
michael	armbrust	no-reply@berkeley...	100000
xiangrui	meng	no-reply@stanford...	120000
matei	null	no-reply@waterloo...	140000
null	wendell	no-reply@berkeley...	160000
michael	jackson	no-reply@neverla.nd	80000
null	wendell	no-reply@berkeley...	160000
xiangrui	meng	no-reply@stanford...	120000
matei	null	no-reply@waterloo...	140000

5.5 Sử dụng filter () để trả về các hàng khớp với một đơn vị từ

```
filterDF = flattenDF.filter(flattenDF.firstName == "xiangrui").sort(flattenDF.lastName)
display(filterDF)
```

```
from pyspark.sql.functions import col, asc

# Use `|` instead of `or`
filterDF = flattenDF.filter((col("firstName") == "xiangrui") | (col("firstName") ==
"michael")).sort(asc("lastName"))
display(filterDF)
```

Thay vì sử dụng “or”, ta có thể dùng dấu “|” trong hàm filter().

5.6 Sử dụng hàm where()

Hàm where() tương tự như hàm filter(), cụ thể như sau:

```
whereDF = flattenDF.where((col("firstName") == "xiangrui") | (col("firstName") ==
"michael")).sort(asc("lastName"))
display(whereDF)
```

5.7 Thay thế các giá trị trong DataFrames null bằng fillna()

```
nonNullDF = flattenDF.fillna("--")
display(nonNullDF)
```

5.8 Chỉ truy xuất các hàm bị thiếu giá trị

Trong ví dụ này, ta truy xuất các giá trị bị thiếu firstname và lastname

```
filterNonNullDF = flattenDF.filter(col("firstName").isNull() |  
col("lastName").isNull()).sort("email")  
display(filterNonNullDF)
```

Còn rất nhiều hàm hữu ích trong Dataframes, tùy vào mục đích sử dụng mà ta gọi các hàm tương ứng. Để có thể tìm hiểu hơn về danh sách các hàm của Dataframes, các bạn có thể truy cập link sau:

<https://spark.apache.org/docs/1.6.1/api/java/org/apache/spark/sql/DataFrame.html>.

Tham khảo

1. <https://spark.apache.org/docs/latest/sql-programming-guide.html>.
2. <https://spark.apache.org/docs/1.6.1/api/java/org/apache/spark/sql/DataFrame.html>
3. <https://docs.databricks.com/spark/latest/dataframes-datasets/introduction-to-dataframes-python.html>
4. <https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-persistence>
5. <https://spark.apache.org/docs/latest/configuration.html#viewing-spark-properties>