**Tìm hiểu về Spark RDD**

1. **Định nghĩa** :

RDD (Resilient Distributed Dataset) là một cấu trúc dữ liệu cơ bản và trừu tượng trong Apache Spark và Spark Core. RDD là tập hợp các đối tượng phân tán không thay đổi, chịu được lỗi, có nghĩa là một khi tạo RDD thì không thể thay đổi được nó. Mỗi tập dữ liệu trong RDD được chia thành các phân vùng logic, có thể được tính toán trên node khác nhau của cụm. Ngoài ra, RDD trừu tượng hóa dữ liệu của việc phân vùng và việc phân phối dữ liệu được thiết kế để chạy tính toán song song trên nhiều nút. Do đó không phải lo lắng về sự song song như Spark. Một RDD chỉ có thể có trong một SparkContext và RDD có tên và id duy nhất.

*Những ưu điểm của RDD:*

* Xử lý trong bộ nhớ
* Tính bất biến
* Khả năng chịu lỗi
* Lười tiến hóa
* Tính phân vùng
* Tính song song

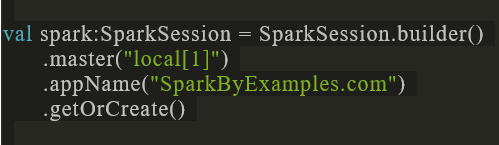
*Hạn chế:*

Spark RDD không phù hợp cho các ứng dụng thực hiện cập nhật cho kho lưu trữ trạng thái như hệ thống lưu trữ cho ứng dụng web. Đối với các ứng dụng này, sẽ hiệu quả hơn nếu sử dụng các hệ thống thực hiện ghi nhật ký truyền thống và kiểm tra dữ liệu như cơ sở dữ liệu. Mục tiêu của RDD là cung cấp một mô hình lập trình hiệu quả cho phân tích hàng loạt và loại bỏ các ứng dụng không đồng bộ này.

1. **Khởi tạo RDD**

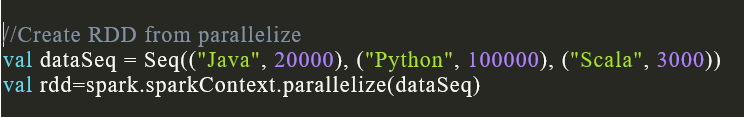
RDD được tạo chủ yếu theo hai cách, đầu tiên là song song hóa một tập hợp hiện có và thứ hai là tham chiếu tập dữ liệu trong hệ thống lưu trữ bên ngoài (HDFS, HDFS, S3, …)

Trước tiên phải khởi tạo SparkSession bằng cách sử dụng phương thức xây dựng mẫu được xác định trong lớp SparkSession. Trong khi khởi tạo, cần cung cấp master và tên ứng dụng.



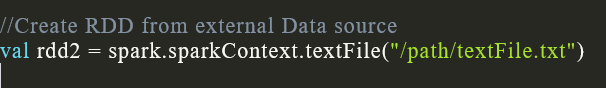
1. **Sử dụng sparkContext.parallelize()**

SparkContext.parallelize được sử dụng để song song hóa một tập hợp hiện có trong chương trình trình điều khiển. Đây là phương pháp cơ bản để tạo RDD và được sử dụng chủ yếu trong POC’s hoặc tạo mẫu và nó yêu cầu tất cả dữ liệu phải có trên chương trình trình điều khiển trước khi tạo RDD. Do đó nó không được sử dụng nhiều cho các ứng dụng sản xuất.



1. **Sử dụng sparkContext.textFile()**

Sử dụng phương thức textFile(), chúng ta có thể đọc tệp văn bản (.txt) vào RDD.



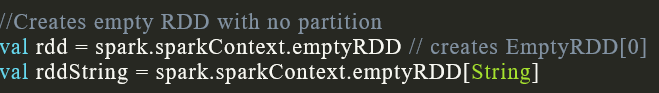
1. **Sử dụng sparkContext. wholeTextFiles()**

Hàm wholeTextFiles() trả về một PairRDD với khóa là đường dẫn tệp và giá trị là nội dung tệp

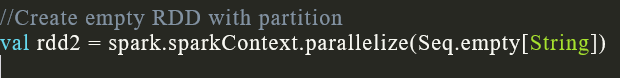
val rdd3 = spark.sparkContext.wholeTextFiles("/path/textFile.txt")

1. **Sử dụng** **sparkContext.emptyRDD()**

Sử dụng phương thức emptyRDD() trên sparkContext, chúng ta có thể tạo một RDD không có dữ liệu. Phương pháp này tạo ra một RDD trống không có phân vùng.



1. **Tạo một RDD trống với phân vùng**



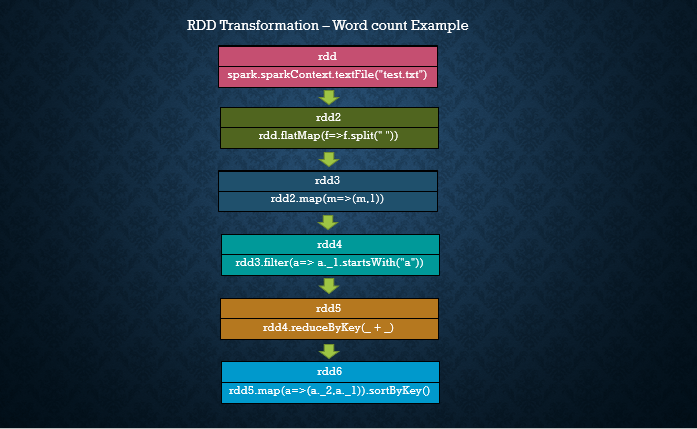
1. **RDD Operations**

**RDD transformations** - Transformations là hoạt động lười biếng, thay vì cập nhật một RDD, các phép toán này trả về một RDD khác.

**RDD actions** - các hoạt động kích hoạt tính toán và trả về giá trị RDD.

**Ví dụ** **RDD** **Transformations**

**Transformations** trên spark RDD trả về một RDD khác và các transformation là lười biếng nghĩa là chúng không thực thi cho đến khi bạn gọi một hành động trên RDD. Một số transformation trên RDD’s là flatMap, map, ReduceByKey, filter, sortByKey và trả về RDD mới thay vì cập nhật hiện tại.



Đầu tiên tạo một RDD bằng cách đọc một file text.

val rdd:RDD[String] = spark.sparkContext.textFile("src/main/scala/test.txt")

**flatMap -** Phép biến đổi flatMap() làm phẳng RDD sau khi áp dụng hàm và trả về một RDD mới. Trong ví dụ dưới đây, đầu tiên, nó chia từng bản ghi theo khoảng trắng trong RDD và cuối cùng làm phẳng nó. Kết quả RDD bao gồm một từ duy nhất trên mỗi bản ghi.

val rdd2 = rdd.flatMap(f=>f.split(" "))

**map -** Phép biến đổimap() được sử dụng để áp dụng bất kỳ hoạt động phức tạp nào như thêm một cột, cập nhật một cột, đầu ra của các phép biến đổi map sẽ luôn có cùng số lượng bản ghi như đầu vào.

**filter -** Phép biến đổi filter () được sử dụng để lọc các bản ghi trong RDD. Ví dụ lọc tất cả các từ bắt đầu bằng “a”.

val rdd4 = rdd3.filter(a=> a.\_1.startsWith("a"))

**reduceByKey -** ReduceByKey () hợp nhất các giá trị cho mỗi khóa với hàm được chỉ định. Ví dụ, nó làm giảm chuỗi từ bằng cách áp dụng hàm sum trên giá trị. Kết quả của RDD chứa các từ duy nhất và số lượng của chúng.

val rdd5 = rdd4.reduceByKey(\_ + \_)

**sortByKey -** Phép biến đổi sortByKey () được sử dụng để sắp xếp các phần tử RDD trên khóa. Ví dụ, trước tiên, chuyển đổi RDD [(String, Int]) thành RDD [(Int, String]) bằng cách sử dụng phép biến đổi map và áp dụng sortByKey mà lý tưởng là sắp xếp trên một giá trị số nguyên. Và cuối cùng, câu lệnh foreach với println trả về tất cả các từ trong RDD và số lượng của chúng là cặp khóa-giá trị.

Code:

val rdd6 = rdd5.map(a=>(a.\_2,a.\_1)).sortByKey()

//Print rdd6 result to console

rdd6.foreach(println

**Ví dụ RDD Actions**

Hành động RDD trả về các giá trị thô từ một RDD. Nói cách khác, bất kỳ hàm RDD nào trả về không phải RDD [T] đều được coi là một hành động.

**Count -** Trả về số lượng record trong RDD

//Action - count

println("Count : "+rdd6.count())

**first –** trả về record đầu tiên

//Action - first

val firstRec = rdd6.first()

println("First Record : "+firstRec.\_1 + ","+ firstRec.\_2)

**max –** trả về record lớn nhất

//Action - max

val datMax = rdd6.max()

println("Max Record : "+datMax.\_1 + ","+ datMax.\_2)

**reduce -** Giảm các bản ghi thành đơn lẻ, chúng ta có thể sử dụng điều này để đếm hoặc tính tổng.

//Action - reduce

val totalWordCount = rdd6.reduce((a,b) => (a.\_1+b.\_1,a.\_2))

println("dataReduce Record : "+totalWordCount.\_1)

**take -** Trả về bản ghi được chỉ định làm đối số

//Action - take

val data3 = rdd6.take(3)

data3.foreach(f=>{

println("data3 Key:"+ f.\_1 +", Value:"+f.\_2)

})

**Collect -** Trả về tất cả dữ liệu từ RDD dưới dạng một mảng.

//Action - collect

val data = rdd6.collect()

data.foreach(f=>{

println("Key:"+ f.\_1 +", Value:"+f.\_2)

})

**saveAsTextFile -** Sử dụng hành động saveAsTestFile, chúng ta có thể ghi RDD vào một tệp văn bản.

rdd6.saveAsTextFile("/tmp/wordCount")

**Các loại RDD**

**PairRDDFunctions hoặc PairRDD -** Pair RDDlà một cặp khóa-giá trị. Đây là loại RDD chủ yếu được sử dụng.

**ShuffledRDD**

**DoubleRDD**

**SequenceFileRDD**

**HadoopRDD**

**ParallelCollectionRDD**

**Tìm hiểu Spark DataFrame**

1. **Định nghĩa:**

Theo Databricks

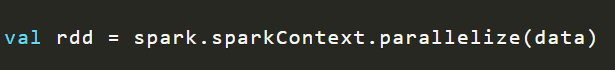
DataFrame là một tập hợp phân phối dữ liệu được tổ chức thành các cột và được đặt tên. DataFrame tương đương với một bảng trong một cơ sở dữ liệu quan hệ hoặc một khung dữ liệu trong R / Python, nhưng tối ưu và phong phú hơn. DataFrames có thể được xây dựng từ một loạt các nguồn như các tập tin dữ liệu có cấu trúc, bảng trong Hive, cơ sở dữ liệu bên ngoài, hoặc RDDs hiện có.

Dưới đây là các tính năng đặc trưng của DataFrame:

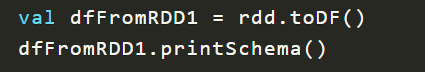
* Hỗ trợ các định dạng dữ liệu khác nhau (Avro, csv, tìm kiếm đàn hồi và Cassandra) và hệ thống lưu trữ (HDFS, bảng HIVE, mysql, v.v.).
* Tối ưu hóa hiện đại và tạo mã thông qua trình tối ưu hóa Spark SQL.
* Có thể dễ dàng tích hợp với tất cả các công cụ và khuôn khổ Big Data thông qua Spark-Core.
* Cung cấp API cho Lập trình Python, Java, Scala và R.

1. **Khởi tạo DataFrame**
2. Tạo DataFrame từ RDD

Tạo một RDD

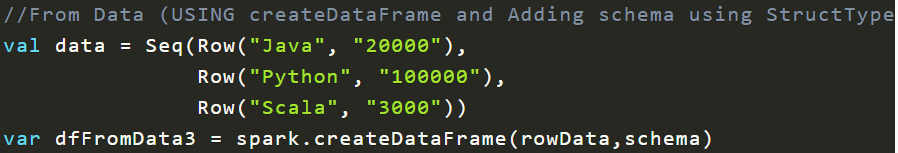


Sử dụng hàm toDF()

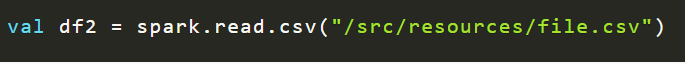


1. Tạo DataFrame từ List Collection

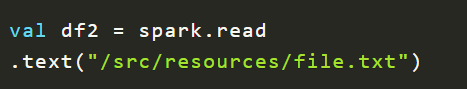
Sử dụng createDataFrame() với Row



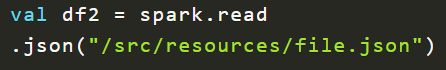
1. Tạo DataFrame từ CSV



1. Tạo DataFrame từ TXT



1. Tạo DataFrame từ JSON



1. **Các chức năng với cột trong DataFrame**
2. Tạo cột mới trong DataFrame



1. Thay đổi giá trị cột



1. Thay đổi kiểu dữ liệu



1. Đổi tên

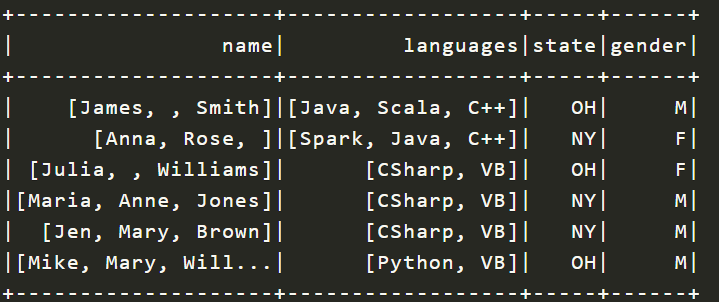


1. Xóa cột



1. **Filter**

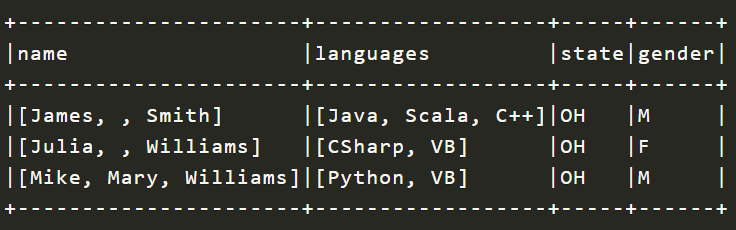
Data mẫu



Sử dụng Filter



Kết quả

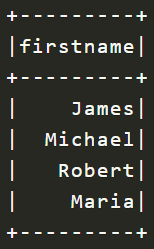


1. **Select**

Bạn có thể chọn cột một hoặc bội số của DataFrame bằng cách chuyển tên cột bạn muốn chọn đến hàm



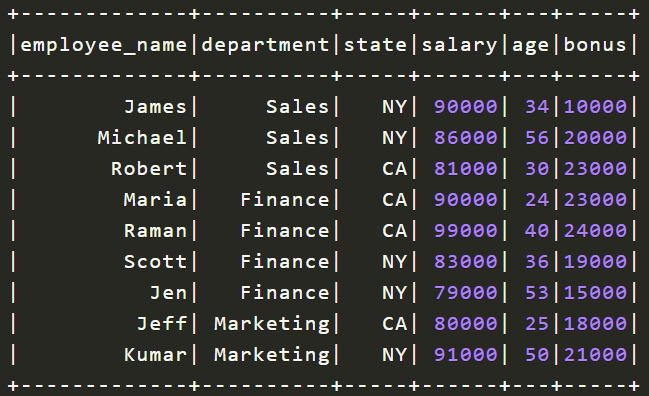
Kết quả



1. **Sort**

Để sắp xếp DataFrame bằng cách tăng dần hoặc giảm dần thứ tự dựa trên một hoặc nhiều cột.

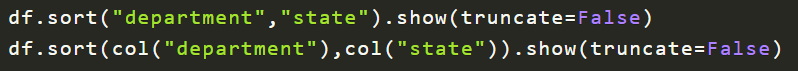
Data mẫu



Syntax



Ví dụ



Kết quả

