**Tìm hiểu về Spark RDD**

1. **RDD là gì**?

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. **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.

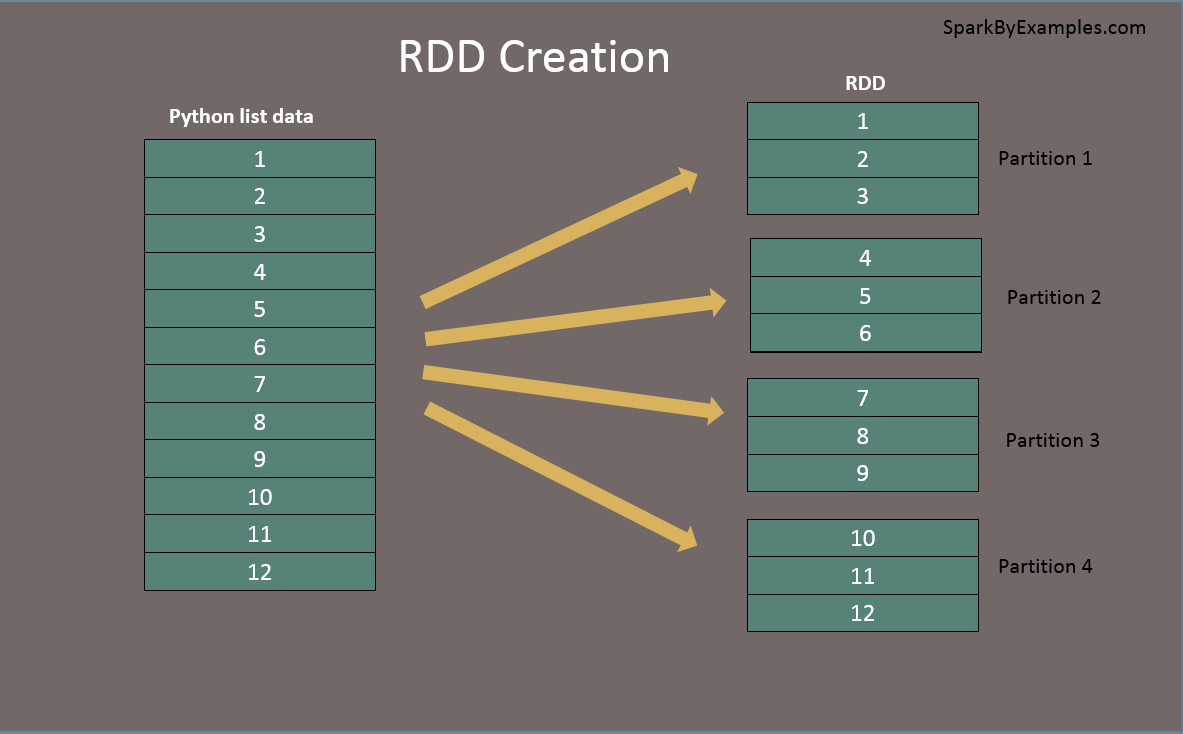
Code:

val spark:SparkSession = SparkSession.builder()

.master("local[1]")

.appName("SparkByExamples.com")

.getOrCreate()



**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.

Code:

//Create RDD from parallelize

val dataSeq = Seq(("Java", 20000), ("Python", 100000), ("Scala", 3000))

val rdd=spark.sparkContext.parallelize(dataSeq)

**Sử dụng sparkContext.textFile()**

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

Code:

//Create RDD from external Data source

val rdd2 = spark.sparkContext.textFile("/path/textFile.txt")

**Sử dụng sparkContext. wholeTextFiles()**

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

Code:

//Reads entire file into a RDD as single record.

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

**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.

Code:

//Creates empty RDD with no partition

val rdd = spark.sparkContext.emptyRDD // creates EmptyRDD[0]

val rddString = spark.sparkContext.emptyRDD[String]

**Tạo một RDD trống với phân vùng**

Code:

//Create empty RDD with partition

val rdd2 = spark.sparkContext.parallelize(Seq.empty[String])

**RDD song song hóa và phân vùng lại**

Khi chúng ta sử dụng phương thức parallelize(), textFile() hoặc wholeTextFiles() của SparkContxt để khởi tạo RDD, nó sẽ tự động chia dữ liệu thành các phân vùng dựa trên tính khả dụng của tài nguyên.

**getNumPartitions** - Trả về một số phân vùng mà tập dữ liệu được chia thành. Bất kỳ phép transformation nào được áp dụng trên RDD đều thực hiện song song. Spark sẽ chạy một tác vụ cho mỗi phân vùng của cụm

Code:

println("initial partition count:"+rdd.getNumPartitions)

//Outputs: initial partition count:2

**Set parallelize manually** – Chúng ta cũng có thể đặt một số phân vùng theo cách thủ công, tất cả những gì chúng ta cần là chuyển một số phân vùng làm tham số thứ hai cho các hàm này, ví dụ như sparkContext.parallelize (dataSeq, 10)).

**Repalallize sử dụng repartition và coalesce** - Spark cung cấp hai cách để phân vùng lại; đầu tiên sử dụng phương thức repartition() xáo trộn dữ liệu từ tất cả các nút còn được gọi là trộn đầy đủ và phương thức coalesce() thứ hai trộn dữ liệu từ các nút tối thiểu. Hàm repartition() hoặc coalesce() đều trả về một RDD mới.

Code:

val reparRdd = rdd.repartition(4)

println("re-partition count:"+reparRdd.getNumPartitions)

//Outputs: "re-partition count:4

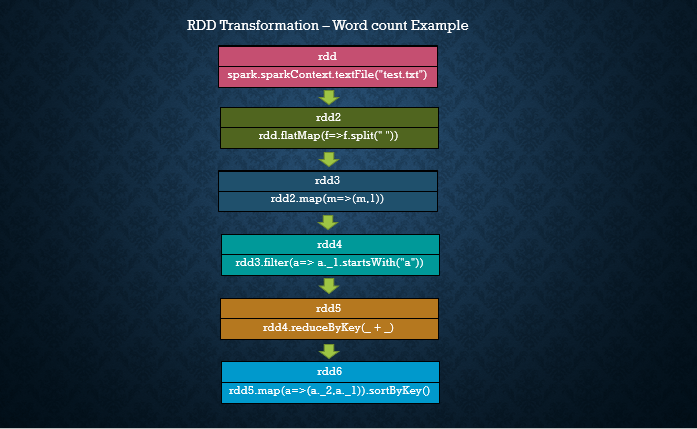
**RDD Operations**

**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. Code:

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.

Code:

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”.

Code:

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.

Code:

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

Code:

//Action - count

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

**first –** trả về record đầu tiên

Code:

//Action - first

val firstRec = rdd6.first()

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

**max –** trả về record lớn nhất

Code:

//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.

Code:

//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

Code:

//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.

Code:

//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.

Code:

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**

**Hoạt động Shuffle**

**Shuffle -** Xáo trộn là một cơ chế mà Spark sử dụng để phân phối lại dữ liệu giữa các trình thực thi khác nhau và thậm chí trên các máy. Xáo trộn kích hoạt khi thực hiện các hoạt động chuyển đổi nhất định như gropByKey (), ReduceByKey (), join () trên RDDS.

Spark Shuffle là một hoạt động tốn kém vì nó liên quan đến những điều sau:

* Disk I/O
* Liên quan đến tuần tự hóa dữ liệu và giải mã hóa
* Network I/O

Khi tạo RDD, Spark không nhất thiết phải lưu trữ dữ liệu cho tất cả các khóa trong một phân vùng vì tại thời điểm tạo, không có cách nào chúng ta có thể đặt khóa cho tập dữ liệu. Do đó, khi chạy thao tác ReduceByKey () để tổng hợp dữ liệu trên các khóa, Spark sẽ thực hiện những việc sau, trước tiên cần chạy các tác vụ để thu thập tất cả dữ liệu từ tất cả các phân vùng và Spark RDD kích hoạt xáo trộn và phân vùng lại cho một số hoạt động như repartition () và Coalesce (), groupByKey (), ReduceByKey (), cogroup () và join ().

**Kích thước phân vùng Shuffle và hiệu suất**

Dựa trên kích thước tập dữ liệu, xáo trộn một số lõi và bộ nhớ Spark có thể có lợi hoặc có hại cho công việc. Khi xử lý với lượng dữ liệu ít hơn, thông thường nên giảm các phân vùng xáo trộn, nếu không, sẽ có nhiều tệp được phân vùng với số lượng bản ghi ít hơn trong mỗi phân vùng, dẫn đến việc chạy nhiều tác vụ với ít dữ liệu hơn để xử lý. Mặt khác, khi có quá nhiều dữ liệu và có ít phân vùng hơn dẫn đến ít tác vụ chạy lâu hơn và đôi khi cũng có thể bị lỗi hết bộ nhớ.

**RDD Persistence**

Spark Cache và Persist là các kỹ thuật tối ưu hóa để cải thiện hiệu suất của các công việc RDD lặp đi lặp lại và tương tác.

Mặc dù Spark cung cấp khả năng tính toán nhanh hơn 100 lần so với các công việc Map Reduce truyền thống, nhưng nếu không thiết kế các công việc để sử dụng lại các phép tính lặp lại, hiệu suất sẽ giảm khi xử lý hàng tỷ hoặc hàng nghìn tỷ dữ liệu.

Sử dụng các phương thức cache () và persist (), Spark cung cấp cơ chế tối ưu hóa để lưu trữ tính toán trung gian của RDD để chúng có thể được sử dụng lại trong các hành động tiếp theo.

**Điểm mạnh của RDD Persistence**

Tiết kiệm chi phí - Tính toán Spark rất tốn kém do đó việc sử dụng lại các tính toán được sử dụng để tiết kiệm chi phí.

Tiết kiệm thời gian - Sử dụng lại các phép tính lặp đi lặp lại tiết kiệm rất nhiều thời gian.

Thời gian thực thi - Tiết kiệm thời gian thực hiện công việc cho phép thực hiện nhiều công việc hơn trên cùng một cụm.

**RDD Cache**

Phương thức Spark RDD cache () theo mặc định lưu tính toán RDD ở mức lưu trữ `MEMORY\_ONLY` có nghĩa là nó sẽ lưu trữ dữ liệu trong heap JVM dưới dạng các đối tượng chưa được quản lý. Phương thức Spark cache () trong lớp RDD gọi nội bộ phương thức Persist () mà lần lượt sử dụng sparkSession.sharedState.cacheManager.cacheQuery để lưu vào bộ nhớ đệm kết quả của RDD.

Code:

val rdd = sc.textFile("src/main/resources/zipcodes-noheader.csv")

val rdd2:RDD[ZipCode] = rdd.map(row=>{

val strArray = row.split(",")

ZipCode(strArray(0).toInt,strArray(1),strArray(3),strArray(4))

})

val rdd3 = rdd2.cache()

println(rdd3.count())

**RDD Persist**

Phương thức Spark Persist() được sử dụng để lưu trữ RDD vào một trong các mức lưu trữ MEMORY\_ONLY, MEMORY\_AND\_DISK, MEMORY\_ONLY\_SER, MEMORY\_AND\_DISK\_SER, DISK\_ONLY, MEMORY\_ONLY\_2, MEMORY\_AND\_DISK\_2 và hơn thế nữa. Spark Persist có hai chữ ký chữ ký đầu tiên không nhận bất kỳ đối số nào mà theo mặc định sẽ lưu nó vào mức lưu trữ MEMORY\_ONLY và chữ ký thứ hai lấy StorageLevel làm đối số để lưu trữ nó ở các mức lưu trữ khác nhau.

Code:

val dfPersist = rdd.persist(StorageLevel.MEMORY\_ONLY)

dfPersist.show(false)

**RDD Unpersist**

Spark tự động theo dõi mọi cuộc gọi Persist () và cache () bạn thực hiện và nó kiểm tra việc sử dụng trên mỗi nút và loại bỏ dữ liệu lâu dài nếu không được sử dụng hoặc bằng cách sử dụng thuật toán ít được sử dụng nhất (LRU). Bạn cũng có thể xóa thủ công bằng phương thức unpersist (), unpersist () đánh dấu RDD là không liên tục và xóa tất cả các khối cho nó khỏi bộ nhớ và đĩa.

Code:

val rddPersist2 = rddPersist.unpersist()

**Cấp độ lưu trữ Persist**

Tất cả các mức lưu trữ khác nhau mà Spark hỗ trợ đều có sẵn tại lớp org.apache.spark.storage.StorageLevel. Mức lưu trữ xác định cách thức và nơi lưu trữ RDD. MEMORY\_ONLY - Đây là hành vi mặc định của phương thức RDD cache () và lưu trữ RDD dưới dạng các đối tượng được giải hóa vào bộ nhớ JVM. Khi không có đủ bộ nhớ, nó sẽ không lưu vào RDD của một số phân vùng và chúng sẽ được tính toán lại khi cần thiết. Điều này chiếm nhiều dung lượng hơn nhưng chạy nhanh hơn vì mất vài chu kỳ CPU để đọc từ bộ nhớ.

MEMORY\_ONLY\_SER - Giống với MEMORY\_ONLY nhưng sự khác biệt là nó lưu trữ RDD dưới dạng các đối tượng được tuần tự hóa vào bộ nhớ JVM. Nó chiếm ít bộ nhớ hơn (tiết kiệm không gian) sau đó MEMORY\_ONLY vì nó lưu các đối tượng dưới dạng tuần tự hóa và mất thêm một vài chu kỳ CPU để giải không gian.

MEMORY\_ONLY\_2 - Giống như mức lưu trữ MEMORY\_ONLY nhưng sao chép từng phân vùng thành hai nút cụm

MEMORY\_ONLY\_SER\_2 - Giống như MEMORY\_ONLY\_SER mức lưu trữ nhưng sao chép từng phân vùng thành hai nút cụm.

MEMORY\_AND\_DISK - Trong Mức lưu trữ này, RDD sẽ được lưu trữ trong bộ nhớ JVM dưới dạng các đối tượng được giải hóa. Khi bộ nhớ được yêu cầu lớn hơn bộ nhớ khả dụng, nó sẽ lưu trữ một số phân vùng dư thừa vào đĩa và đọc dữ liệu từ đĩa khi cần. Nó chậm hơn vì có I / O tham gia.

MEMORY\_AND\_DISK\_SER - Điều này giống với sự khác biệt về mức lưu trữ MEMORY\_AND\_DISK là nó tuần tự hóa các đối tượng RDD trong bộ nhớ và trên đĩa khi không còn dung lượng. MEMORY\_AND\_DISK\_2 - Giống như mức lưu trữ MEMORY\_AND\_DISK nhưng sao chép từng phân vùng thành hai nút cụm.

MEMORY\_AND\_DISK\_SER\_2 - Giống như MEMORY\_AND\_DISK\_SER mức lưu trữ nhưng sao chép từng phân vùng thành hai nút cụm.

DISK\_ONLY - Trong mức lưu trữ này, RDD chỉ được lưu trữ trên đĩa và thời gian tính toán của CPU cao khi có liên quan đến I / O.

DISK\_ONLY\_2 - Tương tự như mức lưu trữ DISK\_ONLY nhưng sao chép từng phân vùng thành hai nút cụm.

**Biến chia sẻ Spark**

Khi Spark thực hiện chuyển đổi bằng cách sử dụng các hoạt động map () hoặc Reduce (), Nó thực hiện các phép biến đổi trên một nút từ xa bằng cách sử dụng các biến được vận chuyển cùng với các tác vụ và các biến này không được gửi lại cho Spark Driver do đó không có khả năng sử dụng lại và chia sẻ các biến giữa các nhiệm vụ. Các biến chia sẻ Spark giải quyết vấn đề này bằng cách sử dụng hai kỹ thuật dưới đây. Spark cung cấp hai loại biến chia sẻ:

* Broadcast variables (biến chia sẻ chỉ đọc)
* Accumulator variables (biến chia sẻ có thể cập nhật)

Broadcast variables là các biến chia sẻ chỉ đọc được lưu trong bộ nhớ đệm và có sẵn trên tất cả các nút trong một cụm để các tác vụ truy cập hoặc sử dụng. Thay vì gửi dữ liệu này cùng với mọi tác vụ, spark phân phối các Broadcast variables đến máy bằng các thuật toán Broadcast hiệu quả để giảm chi phí giao tiếp. Spark Broadcast được tạo bằng phương thức broadcast (v) của lớp SparkContext. Phương thức này nhận đối số v muốn broadcast. Code:

scala> val broadcastVar = sc.broadcast(Array(0, 1, 2, 3))

broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)

scala> broadcastVar.value

res0: Array[Int] = Array(0, 1, 2, 3)

Spark Accumulators là một loại biến chia sẻ khác chỉ được “thêm vào” thông qua một phép toán kết hợp và giao hoán và được sử dụng để thực hiện các bộ đếm (Tương tự như bộ đếm Rút gọn bản đồ) hoặc các phép tính tổng.

Accumulator variables được tạo bằng SparkContext.longAccumulator (v).

Code:

scala> val accum = sc.longAccumulator("SumAccumulator")

accum: org.apache.spark.util.LongAccumulator = LongAccumulator(id: 0, name: Some(SumAccumulator), value: 0)

scala> sc.parallelize(Array(1, 2, 3)).foreach(x => accum.add(x))

-----

-----

scala> accum.value

res2: Long = 6

Spark theo mặc định cung cấp các phương thức accumulator cho các loại long, double và các loại collection. Tất cả các phương thức này đều có trong lớp SparkContext và trả về LongAccumulator, DoubleAccumulator và CollectionAccumulator tương ứng.

**API nâng cao – DataFrame và DataSet**

**Tạo RDD từ DataFrame và ngược lại**

Mặc dù có nhiều API nâng cao hơn so với RDD, chúng ta nên chuyển đổi DataFrame thành RDD hoặc RDD thành DataFrame.

Code ví dụ:

//Converts RDD to DataFrame

val dfFromRDD1 = rdd.toDF()

//Converts RDD to DataFrame with column names

val dfFromRDD2 = rdd.toDF("col1","col2")

//using createDataFrame() - Convert DataFrame to RDD

val df = spark.createDataFrame(rdd).toDF("col1","col2")

//Convert RDD to Dataset

val ds = spark.createDataset(rdd)

//Convert DataFrame to RDD

val rdd = df.rdd

**Spark DataFrame**

1. **Định nghĩa:**

Theo Databricks

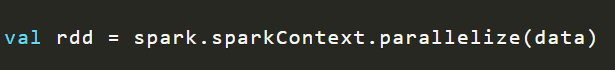
DataFrame là một tập hợp phân phối dữ liệu được tổ chức thành các cột và được đặt tên. DataFrame tương đương với một bảng trong một cơ sở dữ liệu quan hệ hoặc một khung dữ liệu trong R / Python, nhưng tối ưu và phong phú hơn. DataFrames có thể được xây dựng từ một loạt các nguồn như các tập tin dữ liệu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

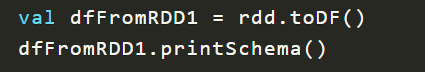
* Hỗ trợ các định dạng dữ liệu khác nhau (Avro, csv, tìm kiếm đàn hồi và Cassandra) và hệ thống lưu trữ (HDFS, bảng HIVE, mysql, v.v.).
* Tối ưu hóa hiện đại và tạo mã thông qua trình tối ưu hóa Spark SQL.
* Có thể dễ dàng tích hợp với tất cả các công cụ và khuôn khổ Big Data thông qua Spark-Core.
*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

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

Tạo một RDD

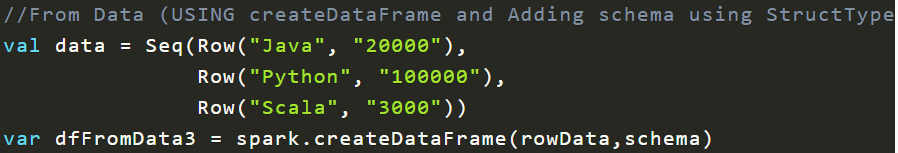


Sử dụng hàm toDF()

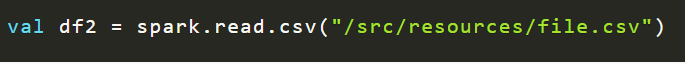


1. Tạo DataFrame từ List Collection

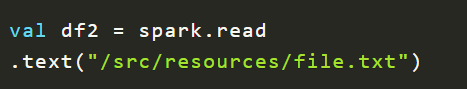
Sử dụng createDataFrame() với Row



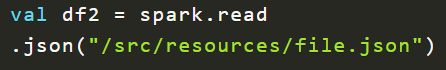
1. Tạo DataFrame từ CSV



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

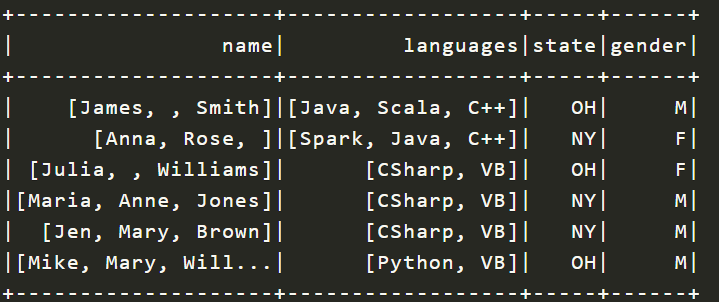


1. Xóa cột



1. **Filter**

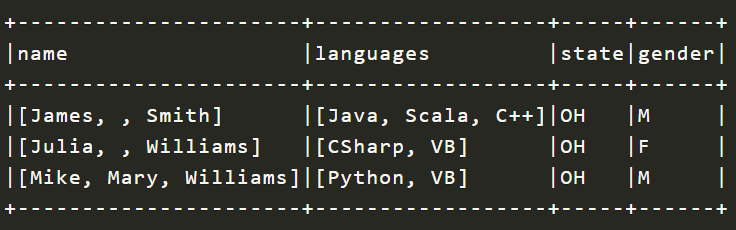
Data mẫu



Sử dụng Filter



Kết quả

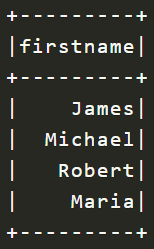


1. **Select**

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



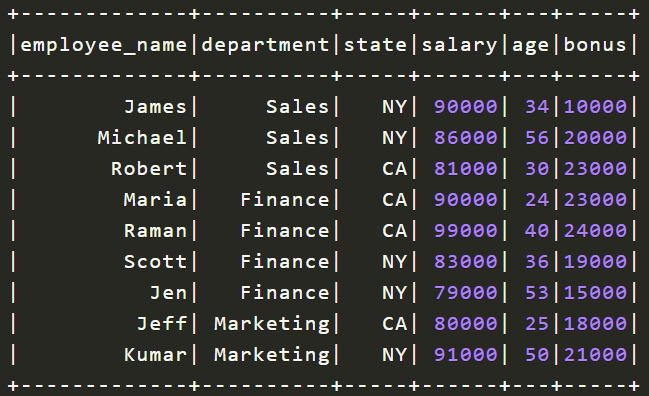
Kết quả



1. **Sort**

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

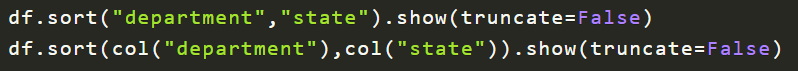
Data mẫu



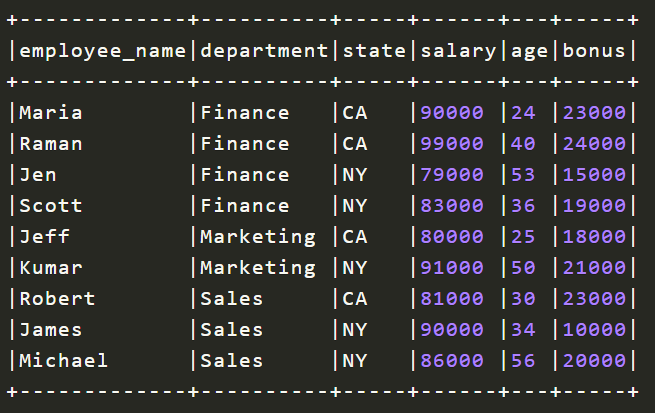
Syntax



Ví dụ



Kết quả



**TÀI LIỆU THAM KHẢO**

<https://sparkbyexamples.com>

<https://www.tutorialspoint.com/apache_spark/index.htm>