第十一章 RDD编程

CONTENTS

在pyspark中运行代码

- pyspark提供了简单的方式来学习Spark API
- pyspark可以以实时、交互的方式来分析数据
- pyspark提供了Python交互式执行环境

在pyspark中运行代码

pyspark命令及其常用的参数如下：

```
pyspark --master <master-url>
```

Spark的运行模式取决于传递给SparkContext的Master URL的值。Master URL可以是以下任一种形式：

* local 使用一个Worker线程本地化运行SPARK(完全不并行)
* local[*] 使用逻辑CPU个数数量的线程来本地化运行Spark
* local[K] 使用K个Worker线程本地化运行Spark（理想情况下，K应该根据运行机器的CPU核数设定）
* spark://HOST:PORT 连接到指定的Spark standalone master。默认端口是7077
* yarn-client 以客户端模式连接YARN集群。集群的位置可以在HADOOP_CONF_DIR 环境变量中找到
* yarn-cluster 以集群模式连接YARN集群。集群的位置可以在HADOOP_CONF_DIR 环境变量中找到
* mesos://HOST:PORT 连接到指定的Mesos集群。默认接口是5050

3.2 在pyspark中运行代码

比如，要采用本地模式，在4个CPU核心上运行pyspark：

```
$ cd /usr/local/spark
$ ./bin/pyspark --master local[4]
```

或者，可以在CLASSPATH中添加code.jar，命

```
$ cd /usr/local/spark
$ ./bin/pyspark --master local[4] --jars code.jar
```

可以执行“pyspark --help”命令，获取完整的选项列表，

```
$ cd /usr/local/spark
$ ./bin/pyspark --help
```

3.2 在pyspark中运行代码

在Spark中采用本地模式启动pyspark的命令主要包含以下参数：

--master：这个参数表示当前的pyspark要连接到哪个master，如果是local[*]，就是使用本地模式启动pyspark，其中，中括号内的星号表示需要使用几个CPU核心(core)，也就是启动几个线程模拟Spark集群

--jars： 这个参数用于把相关的JAR包添加到CLASSPATH中；如果有多个jar包，可以使用逗号分隔符连接它们

在pyspark中运行代码

执行如下命令启动pyspark（默认是local模式）：

```
$ cd /usr/local/spark
$ ./bin/pyspark
```

启动pyspark成功后在输出信息的末尾可以看到“>>>”的命令提示符

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.4.0
      /_/

Using Python version 3.4.3 (default, Nov 12 2018 22:25:49)
SparkSession available as 'spark'.
>>>
```

在pyspark中运行代码

可以在里面输入scala代码进行调试：

```
>>> 8*2+5
21
```

可以使用命令“exit()”退出pyspark：

```
>>> exit()
```

安装编译打包工具

WordCount.py

```
1 from pyspark import SparkConf, SparkContext
2 conf = SparkConf().setMaster("local").setAppName("My App")
3 sc = SparkContext(conf = conf)
4 logFile = "file:///usr/local/spark/README.md"
5 logData = sc.textFile(logFile, 2).cache()
6 numAs = logData.filter(lambda line: 'a' in line).count()
7 numBs = logData.filter(lambda line: 'b' in line).count()
8 print('Lines with a: %s, Lines with b: %s' % (numAs, numBs))
```

对于这段Python代码，可以直接使用如下命令执行：

```
$ cd /usr/local/spark/mycode/python
$ python3 WordCount.py
```

执行该命令以后，可以得到如下结果：

```
Lines with a: 62, Lines with b: 30
```

开发Spark独立应用程序

1 编写程序
2 通过spark-submit运行程序

2 通过spark-submit运行程序

可以通过spark-submit提交应用程序，该命令的格式如下：

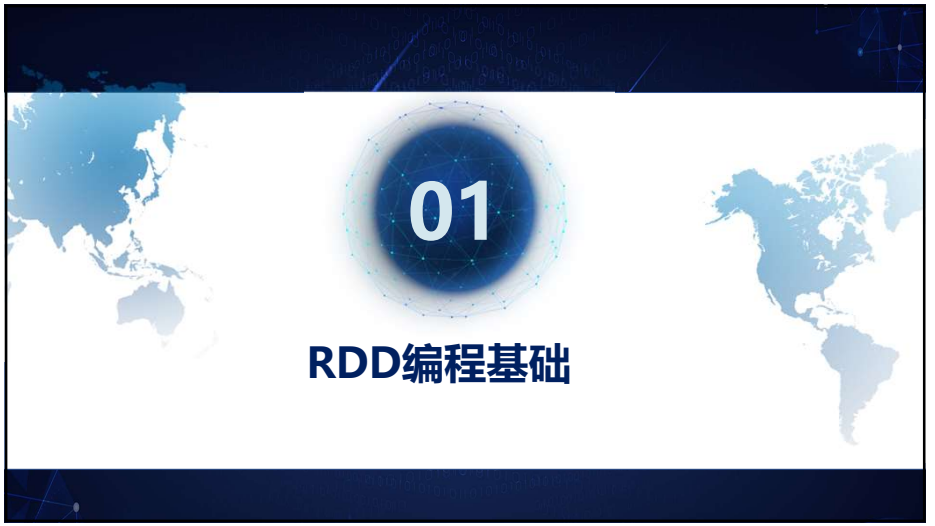
```
spark-submit
 --master <master-url>
 --deploy-mode <deploy-mode>   #部署模式
 ... #其他参数
 <application-file>  #Python代码文件
 [application-arguments] #传递给主类的主方法的参数
```

可以执行“spark-submit --help”命令，获取完整的选项列表

```
$ cd /usr/local/spark
$ ./bin/spark-submit --help
```

2 通过spark-submit运行程序

Master URL可以是以下任一种形式：
* local 使用一个Worker线程本地化运行SPARK(完全不并行)
* local[*] 使用逻辑CPU个数数量的线程来本地化运行Spark
* local[K] 使用K个Worker线程本地化运行Spark（理想情况下，K应该根据运行机器的CPU核数设定）
* spark://HOST:PORT 连接到指定的Spark standalone master。默认端口是7077.
* yarn-client 以客户端模式连接YARN集群。集群的位置可以在HADOOP_CONF_DIR 环境变量中找到。
* yarn-cluster 以集群模式连接YARN集群。集群的位置可以在HADOOP_CONF_DIR 环境变量中找到。
* mesos://HOST:PORT 连接到指定的Mesos集群。默认接口是5050。



3.3.2 通过spark-submit运行程序

以通过 spark-submit 提交到 Spark 中运行，命令如下：

```
$ /usr/local/spark/bin/spark-submit  /usr/local/spark/mycode/python/WordCount.py
```

可以在命令中间使用"\"符号，把一行完整命令"人为断开成多行"进行输入，效果如下：

```
$ /usr/local/spark/bin/spark-submit  \
> /usr/local/spark/mycode/python/WordCount.py
```

上面命令的执行结果如下：

```
Lines with a: 62, Lines with b: 30
```

为了避免其他多余信息对运行结果的干扰，可以修改log4j的日志信息显示级别：

log4j.rootCategory=INFO, console
修改为
log4j.rootCategory=ERROR, console

1.1 RDD创建

1. 从文件系统中加载数据创建RDD

- Spark采用textFile()方法来从文件系统中加载数据创建RDD
- 该方法把文件的URI作为参数，这个URI可以是：
  - 本地文件系统的地址
  - 或者是分布式文件系统HDFS的地址
  - 或者是Amazon S3的地址等等

2. 通过并行集合（列表）创建RDD

1.1 RDD创建

（1）从本地文件系统中加载数据创建RDD

```
>>> lines =
sc.textFile("file:///usr/local/spark/mycode/rdd/word.t
xt")
>>> lines.foreach(print)
Hadoop is good
Spark is fast
Spark is better
```

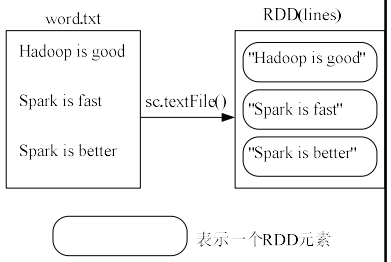


图 从文件中加载数据生成RDD

1.1 RDD创建

2. 通过并行集合（列表）创建RDD

可以调用SparkContext的parallelize方法，在Driver中一个已经存在的集合（列表）上创建。

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

或者，也可以从列表中创建：

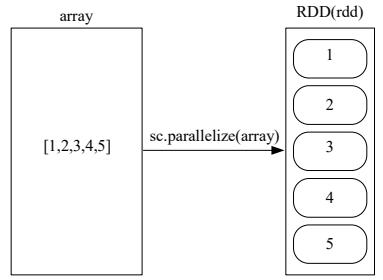


图 从数组创建RDD示意图

1.1 RDD创建

（2）从分布式文件系统HDFS中加载数据

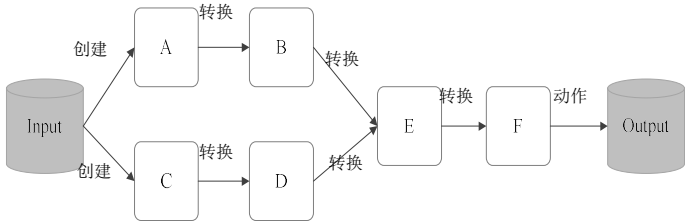
```
>>> lines =
sc.textFile("hdfs://localhost:9000/user/hadoop/word.txt")
>>> lines = sc.textFile("/user/hadoop/word.txt")
>>> lines = sc.textFile("word.txt")
```

三条语句是完全等价的，可以使用其中任意一种方式

1.2 RDD操作

1. 转换操作

- 对于RDD而言，每一次转换操作都会产生不同的RDD，供给下一个“转换”使用
- 转换得到的RDD是惰性求值的，也就是说，整个转换过程只是记录了转换的轨迹，并不会发生真正的计算，只有遇到行动操作时，才会发生真正的计算，开始从血缘关系源头开始，进行物理的转换操作



1.2 RDD操作

1. 转换操作

表 常用的RDD转换操作API

操作	含义
filter(func)	筛选出满足函数func的元素，并返回一个新的数据集
map(func)	将每个元素传递到函数func中，并将结果返回为一个新的数据集
flatMap(func)	与map()相似，但每个输入元素都可以映射到0或多个输出结果
groupByKey()	应用于(K,V)键值对的数据集时，返回一个新的(K, Iterable)形式的数据集
reduceByKey(func)	应用于(K,V)键值对的数据集时，返回一个新的(K, V)形式的数据集，其中每个值是将每个key传递到函数func中进行聚合后的结果

1.2 RDD操作

1. 转换操作

•map(func)

map(func)操作将每个元素传递到函数func中，并将结果返回为一个新的数据集

```
>>> data = [1,2,3,4,5]
>>> rdd1 = sc.parallelize(data)
>>> rdd2 = rdd1.map(lambda x:x+10)
>>> rdd2.foreach(print)
11
13
12
14
15
```

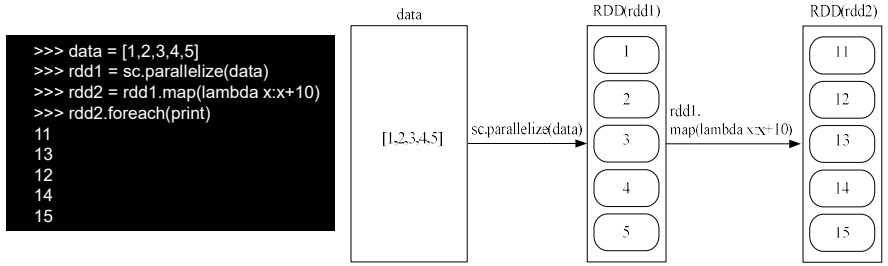


图 map()操作实例执行过程示意图

1.2 RDD操作

1. 转换操作

•filter(func)：筛选出满足函数func的元素，并返回一个新的数据集

```
>>> lines = sc.textFile("file:///usr/local/spark/mycode/rdd/word.txt")
>>> linesWithSpark = lines.filter(lambda line: "Spark" in line)
>>> linesWithSpark.foreach(print)
Spark is better
Spark is fast
```

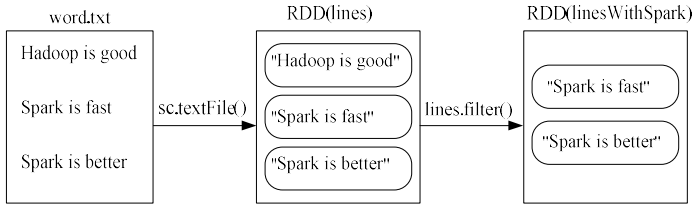


图 filter()操作实例执行过程示意图

1.2 RDD操作

1. 转换操作

•map(func)

```
>>> lines =
sc.textFile("file:///usr/local/spark/mycode/rdd/word.txt")
>>> words = lines.map(lambda line:line.split(" "))
>>> words.foreach(print)
['Hadoop', 'is', 'good']
['Spark', 'is', 'fast']
['Spark', 'is', 'better']
```

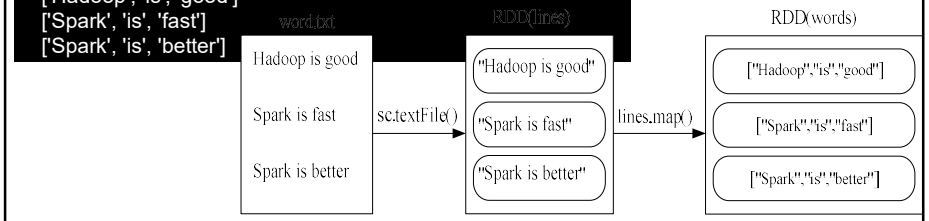


图 map()操作实例执行过程示意图

1.2 RDD操作

1. 转换操作

•flatMap(func)

```
>>> lines = sc.textFile("file:///usr/local/spark/mycode/rdd/word.txt")
>>> words = lines.flatMap(lambda line:line.split(" "))
```

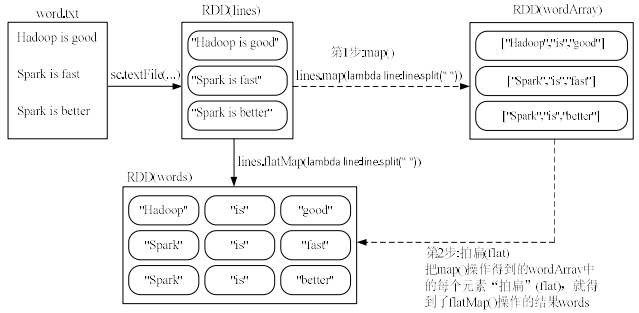


图 flatMap()操作实例执行过程示意图

1.2 RDD操作

1. 转换操作

•groupByKey()

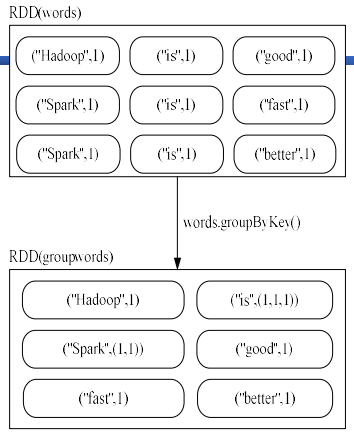


图 groupByKey()操作实例执行过程示意图

4.1.2 RDD操作

1. 转换操作

•groupByKey()　groupByKey()应用于(K,V)键值对的数据集时，返回一个新的(K, Iterable)形式的数据集

```
>>> words = sc.parallelize([("Hadoop",1),("is",1),("good",1), \
... ("Spark",1),("is",1),("fast",1),("Spark",1),("is",1),("better",1)])
>>> words1 = words.groupByKey()
>>> words1.foreach(print)
('Hadoop', <pyspark.resultiterable.ResultIterable object at 0x7fb210552c88>)
('better', <pyspark.resultiterable.ResultIterable object at 0x7fb210552e80>)
('fast', <pyspark.resultiterable.ResultIterable object at 0x7fb210552c88>)
('good', <pyspark.resultiterable.ResultIterable object at 0x7fb210552c88>)
('Spark', <pyspark.resultiterable.ResultIterable object at 0x7fb210552f98>)
('is', <pyspark.resultiterable.ResultIterable object at 0x7fb210552e10>)
```

1.2 RDD操作

1. 转换操作

•reduceByKey(func)　reduceByKey(func)应用于(K,V)键值对的数据集时，返回一个新的(K, V)形式的数据集，其中的每个值是将每个key传递到函数func中进行聚合后得到的结果

```
>>> words =
sc.parallelize([("Hadoop",1),("is",1),("good",1),("Spark",1), \
... ("is",1),("fast",1),("Spark",1),("is",1),("better",1)])
>>> words1 = words.reduceByKey(lambda a,b:a+b)
>>> words1.foreach(print)
('good', 1)
('Hadoop', 1)
('better', 1)
('Spark', 2)
('fast', 1)
('is', 3)
```

1.2 RDD操作

1. 转换操作

•reduceByKey(func)

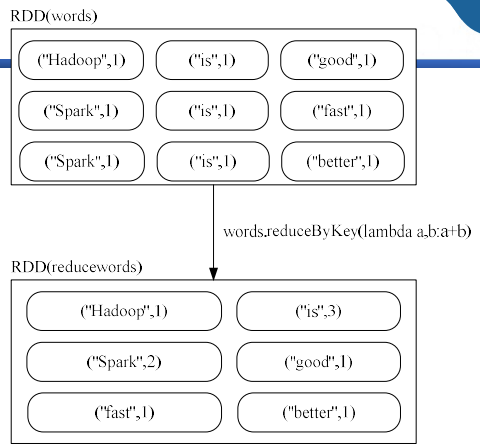


图 reduceByKey()操作实例执行过程示意图

1.2 RDD操作

2. 行动操作

行动操作是真正触发计算的地方。Spark程序执行到行动操作时，才会执行真正的计算，从文件中加载数据，完成一次又一次转换操作。最终，完成行动操作得到结果。

常用的RDD行动操作API

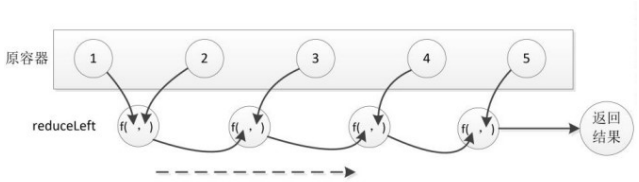
操作	含义
count()	返回数据集中的元素个数
collect()	以数组的形式返回数据集中的所有元素
first()	返回数据集中的第一个元素
take(n)	以数组的形式返回数据集中的前n个元素
reduce(func)	通过函数func（输入两个参数并返回一个值）聚合数据集中的元素
foreach(func)	将数据集中的每个元素传递到函数func中运行

1.2 RDD操作

1. 转换操作

•reduceByKey(func)

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



<“spark”,1>

<“spark”,1> → <“spark”,3>

<“spark”,1>

1.2 RDD操作

```
>>> rdd = sc.parallelize([1,2,3,4,5])
>>> rdd.count()
5
>>> rdd.first()
1
>>> rdd.take(3)
[1, 2, 3]
>>> rdd.reduce(lambda a,b:a+b)
15
>>> rdd.collect()
[1, 2, 3, 4, 5]
>>> rdd.foreach(lambda elem:print(elem))
1
2
3
4
5
```

1.2 RDD操作

惰性机制

所谓的“惰性机制”是指，整个转换过程只是记录了转换的轨迹，并不会发生真正的计算，只有遇到行动操作时，才会触发“从头到尾”的真正的计算
这里给出一段简单的语句来解释Spark的惰性机制

```
>>> lines = sc.textFile("file:///usr/local/spark/mycode/rdd/word.txt")
>>> lineLengths = lines.map(lambda s:len(s))
>>> totalLength = lineLengths.reduce(lambda a,b:a+b)
>>> print(totalLength)
```

1.3 持久化

- 可以通过持久化（缓存）机制避免这种重复计算的开销
- 可以使用persist()方法对一个RDD标记为持久化
- 之所以说“标记为持久化”，是因为出现persist()语句的地方，并不会马上计算生成RDD并把它持久化，而是要**等到遇到第一个行动操作触发真正计算以后，才会把计算结果进行持久化**
- 持久化后的RDD将会被保留在计算节点的内存中被后面的行动操作重复使用

1.3 持久化

在Spark中，RDD采用惰性求值的机制，每次遇到行动操作，都会从头开始执行计算。每次调用行动操作，都会触发一次从头开始的计算。这对于迭代计算而言，代价是很大的，迭代计算经常需要多次重复使用同一组数据

例子：

```
>>> list = ["Hadoop","Spark","Hive"]
>>> rdd = sc.parallelize(list)
>>> print(rdd.count()) //行动操作，触发一次真正从头到尾的计算
3
>>> print(','.join(rdd.collect())) //行动操作，触发一次真正从头到尾的计算
Hadoop,Spark,Hive
```

1.3 持久化

persist()的圆括号中包含的是持久化级别参数：

- persist(MEMORY_ONLY)：表示将RDD作为反序列化的对象存储于JVM中，如果内存不足，就要按照LRU原则替换缓存中的内容
- persist(MEMORY_AND_DISK)表示将RDD作为反序列化的对象存储在JVM中，如果内存不足，超出的分区将会被存放在硬盘上
- 一般而言，使用cache()方法时，会调用persist(MEMORY_ONLY)
- 可以使用unpersist()方法手动地把持久化的RDD从缓存中移除

4.1.3 持久化

针对上面的实例，增加持久化语句以后的执行过程如下：

```
>>> list = ["Hadoop","Spark","Hive"]
>>> rdd = sc.parallelize(list)
>>> rdd.cache()  #会调用persist(MEMORY_ONLY)
>>> print(rdd.count()) #第一次行动操作，触发一次真正从头到尾的计算，这时上面的rdd.cache()才会被执行，把这个rdd放到缓存中
3
>>> print(','.join(rdd.collect())) #第二次行动操作，不需要触发从头到尾的计算，只需要重复使用上面缓存中的rdd
Hadoop,Spark,Hive
```

1.4 分区

1.分区的作用
（2）减少通信开销

UserData（UserId，UserInfo）
Events（UserID，LinkInfo）
UserData 和Events 表进行连接操作，获得
（UserID，UserInfo，LinkInfo）

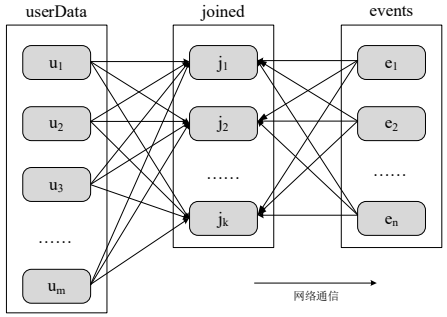


图 未分区时对UserData和Events两个表进行连接操作

1.4 分区

RDD是弹性分布式数据集，通常RDD很大，会被分成很多个分区，分别保存在不同的节点上

1.分区的作用
（1）增加并行度

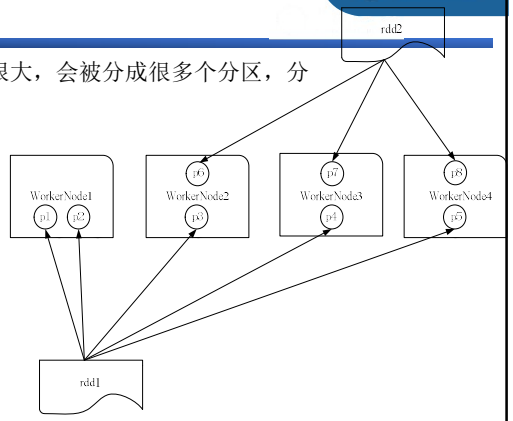


图 RDD分区被保存到不同节点上

1.4 分区

1.分区的作用
（2）减少通信开销

UserData（UserId，UserInfo）
Events（UserID，LinkInfo）
UserData 和Events 表进行连接操作，获得
（UserID，UserInfo，LinkInfo）

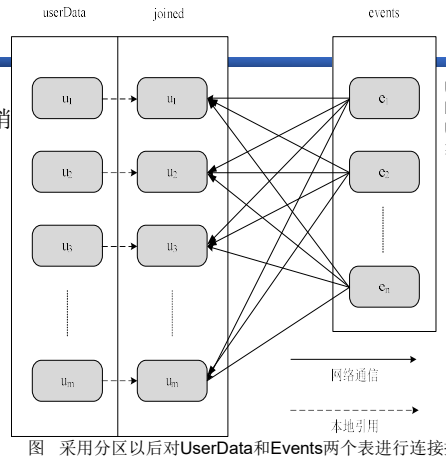


图 采用分区以后对UserData和Events两个表进行连接操作

2.RDD分区原则

RDD分区的一个原则是使得分区的个数尽量等于集群中的CPU核心（core）数目

对于不同的Spark部署模式而言（本地模式、Standalone模式、YARN模式、Mesos模式），都可以通过设置spark.default.parallelism这个参数的值，来配置默认的分区数目，一般而言：
*本地模式：默认为本地机器的CPU数目，若设置了local[N],则默认为N
*Apache Mesos：默认的分区数为8
*Standalone或YARN：在“集群中所有CPU核心数目总和”和“2”二者中取较大值作为默认值

3.设置分区的个数

（2）使用repariton方法重新设置分区个数

通过转换操作得到新 RDD 时，直接调用 repartition 方法即可。例如：

```
>>> data = sc.parallelize([1,2,3,4,5],2)
>>> len(data.glom().collect())  #显示data这个RDD的分区数量
2
>>> rdd = data.repartition(1)  #对data这个RDD进行重新分区
>>> len(rdd.glom().collect())  #显示rdd这个RDD的分区数量
1
```


3.设置分区的个数

（1）创建RDD时手动指定分区个数

在调用textFile()和parallelize()方法的时候手动指定分区个数即可，语法格式如下：
sc.textFile(path, partitionNum)
其中，path参数用于指定要加载的文件的地址，partitionNum参数用于指定分区个数。

```
>>> list = [1,2,3,4,5]
>>> rdd = sc.parallelize(list,2)  //设置两个分区
```


4.自定义分区方法

Spark提供了自带的HashPartitioner（哈希分区）与RangePartitioner（区域分区），能够满足大多数应用场景的需求。与此同时，Spark也支持自定义分区方式，即通过提供一个自定义的分区函数来控制RDD的分区方式，从而利用领域知识进一步减少通信开销

1.4 分区

实例：根据key值的最后一位数字，写到不同的文件

例如：
10写入到part-00000
11写入到part-00001
.
.
.
19写入到part-00009

1.4 分区

使用如下命令运行TestPartitioner.py：

```
$ cd /usr/local/spark/mycode/rdd
$ python3 TestPartitioner.py
```

或者，使用如下命令运行TestPartitioner.py：

```
$ cd /usr/local/spark/mycode/rdd
$ /usr/local/spark/bin/spark-submit TestPartitioner.py
```

程序运行结果会返回如下信息：

```
The main function is running
MyPartitioner is running
The key is 0
MyPartitioner is running
The key is 1
……
MyPartitioner is running
The key is 9
```

1.4 分区

TestPartitioner.py

```
from pyspark import SparkConf, SparkContext


def MyPartitioner(key):
    print("MyPartitioner is running")
    print('The key is %d' % key)
    return key%10


def main():
    print("The main function is running")
    conf = SparkConf().setMaster("local").setAppName("MyApp")
    sc = SparkContext(conf = conf)
    data = sc.parallelize(range(10),5)
    data.map(lambda x:(x,1)) \
        .partitionBy(10,MyPartitioner) \
        .map(lambda x:x[0]) \
        .saveAsTextFile("file:///usr/local/spark/mycode/rdd/partitioner")


if __name__ == '__main__':
    main()
```

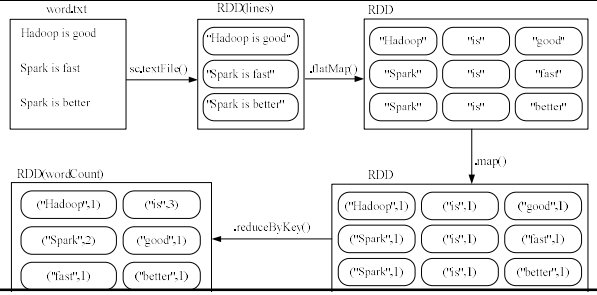
1.5 一个综合实例

假设有一个本地文件word.txt，里面包含了很多行文本，每行文本由多个单词构成，单词之间用空格分隔。可以使用如下语句进行词频统计（即统计每个单词出现的次数）：

```
>>> lines = sc. \
... textFile("file:///usr/local/spark/mycode/rdd/word.txt")
>>> wordCount = lines.flatMap(lambda line:line.split(" ")). \
... map(lambda word:(word,1)).reduceByKey(lambda a,b:a+b)
>>> print(wordCount.collect())
[('good', 1), ('Spark', 2), ('is', 3), ('better', 1), ('Hadoop', 1), ('fast', 1)]
```

4.1.5 一个综合实例

```
>>> lines = sc. \
... textFile("file:///usr/local/spark/mycode/rdd/word.txt")
>>> wordCount = lines.flatMap(lambda line:line.split(" ")). \
... map(lambda word:(word,1)).reduceByKey(lambda a,b:a+b)
>>> print(wordCount.collect())
[('good', 1), ('Spark', 2), ('is', 3), ('better', 1), ('Hadoop', 1), ('fast', 1)]
```



1.5 一个综合实例

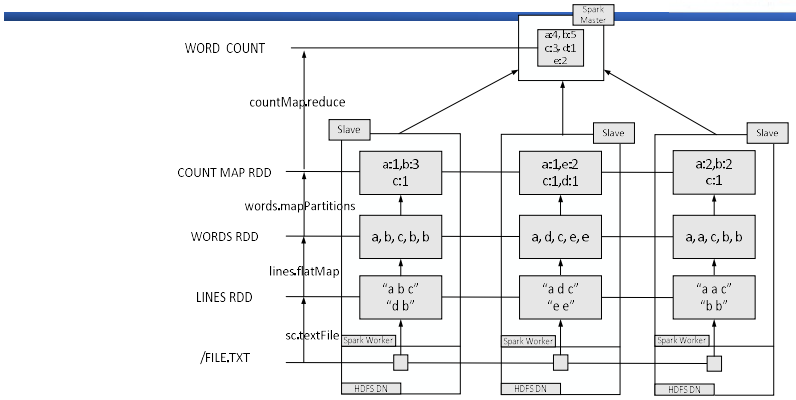


图 在集群中执行词频统计过程示意图

1.5 一个综合实例

在实际应用中，单词文件可能非常大，会被保存到分布式文件系统HDFS中，Spark和Hadoop会统一部署在一个集群上

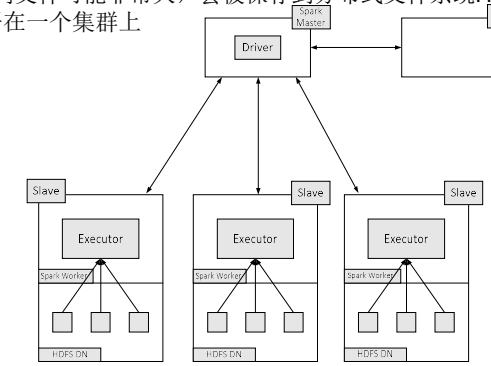


图　在一个集群中同时部署Hadoop和Spark

02

键值对RDD

4.2 键值对RDD

4.2.1 键值对RDD的创建

（1）第一种创建方式：从文件中加载

可以采用多种方式创建键值对RDD，其中一种主要方式是使用map()函数来实现

```
>>> lines =
sc.textFile("file:///usr/local/spark/mycode/pairrdd/word.txt")
>>> pairRDD = lines.flatMap(lambda line:line.split("
")).map(lambda word:(word,1))
>>> pairRDD.foreach(print)
('I', 1)
('love', 1)
('Hadoop', 1)
……
```

2.1键值对RDD的创建

（2）第二种创建方式：通过并行集合（列表）创建RDD

```
>>> list = ["Hadoop","Spark","Hive","Spark"]
>>> rdd = sc.parallelize(list)
>>> pairRDD = rdd.map(lambda word:(word,1))
>>> pairRDD.foreach(print)
(Hadoop,1)
(Spark,1)
(Hive,1)
(Spark,1)
```

2.2 常用的键值对RDD转换操作

- **reduceByKey(func)**
- **groupByKey()**
- **keys**
- **values**
- **sortByKey()**
- **mapValues(func)**
- **join**
- **combineByKey**

2.2 常用的键值对RDD转换操作

•reduceByKey(func)

reduceByKey(func)的功能是，使用func函数合并具有相同键的值

(Hadoop,1)
(Spark,1)
(Hive,1)
(Spark,1)

```
>>> pairRDD =
sc.parallelize([("Hadoop",1),("Spark",1),("Hive",1),("Spark",1)])
>>> pairRDD.reduceByKey(lambda a,b:a+b).foreach(print)
('Spark', 2)
('Hive', 1)
('Hadoop', 1)
```

2.2 常用的键值对RDD转换操作

reduceByKey和groupByKey的区别

•reduceByKey用于对每个key对应的多个value进行merge操作，最重要的是它能够在本地先进行merge操作，并且merge操作可以通过函数自定义

•groupByKey也是对每个key进行操作，但只生成一个sequence，groupByKey本身不能自定义函数，需要先用groupByKey生成RDD，然后才能对此RDD通过map进行自定义函数操作

2.2 常用的键值对RDD转换操作

•groupByKey()

groupByKey()的功能是，对具有相同键的值进行分组

比如，对四个键值对("spark",1)、("spark",2)、("hadoop",3)和("hadoop",5)，采用groupByKey()后得到的结果是：("spark",(1,2))和("hadoop",(3,5))

(Hadoop,1)
(Spark,1)
(Hive,1)
(Spark,1)

```
>>> list = [("spark",1),("spark",2),("hadoop",3),("hadoop",5)]
>>> pairRDD = sc.parallelize(list)
>>> pairRDD.groupByKey()
PythonRDD[27] at RDD at PythonRDD.scala:48
>>> pairRDD.groupByKey().foreach(print)
('hadoop', <pyspark.resultiterable.ResultIterable object at
0x7f2c1093ecf8>)
('spark', <pyspark.resultiterable.ResultIterable object at
0x7f2c1093ecf8>)
```

2.2 常用的键值对RDD转换操作

reduceByKey和groupByKey的区别

```
>>> words = ["one", "two", "two", "three", "three", "three"]
>>> wordPairsRDD = sc.parallelize(words).map(lambda word:(word, 1))
>>> wordCountsWithReduce = wordPairsRDD.reduceByKey(lambda a,b:a+b)
>>> wordCountsWithReduce.foreach(print)
('one', 1)
('two', 2)
('three', 3)
>>> wordCountsWithGroup = wordPairsRDD.groupByKey(). \
... map(lambda t:(t[0],sum(t[1])))
>>> wordCountsWithGroup.foreach(print)
('two', 2)
('three', 3)
('one', 1)
```

上面得到的wordCountsWithReduce和wordCountsWithGroup是完全一样的，但是，它们的内部运算过程是不同的

2.2 常用的键值对RDD转换操作

•keys

keys只会把Pair RDD中的key返回形成一个新的RDD

(Hadoop,1)
(Spark,1)
(Hive,1)
(Spark,1)

```
>>> list = [("Hadoop",1),("Spark",1),("Hive",1),("Spark",1)]
>>> pairRDD = sc.parallelize(list)
>>> pairRDD.keys().foreach(print)
Hadoop
Spark
Hive
Spark
```

2.2 常用的键值对RDD转换操作

•sortByKey()

sortByKey()的功能是返回一个根据键排序的RDD

(Hadoop,1)　(Spark,1)　(Hive,1)　(Spark,1)

```
>>> list = [("Hadoop",1),("Spark",1),("Hive",1),("Spark",1)]
>>> pairRDD = sc.parallelize(list)
>>> pairRDD.foreach(print)
('Hadoop', 1)
('Spark', 1)
('Hive', 1)
('Spark', 1)
>>> pairRDD.sortByKey().foreach(print)
('Hadoop', 1)
('Hive', 1)
('Spark', 1)
('Spark', 1)
```

2.2 常用的键值对RDD转换操作

•values

values只会把Pair RDD中的value返回形成一个新的RDD。

(Hadoop,1)
(Spark,1)
(Hive,1)
(Spark,1)

```
>>> list = [("Hadoop",1),("Spark",1),("Hive",1),("Spark",1)]
>>> pairRDD = sc.parallelize(list)
>>> pairRDD.values().foreach(print)
1
1
1
1
```

4.2.2 常用的键值对RDD转换操作

•sortByKey()和sortBy()

```
>>> d1 = sc.parallelize([("c",8),("b",25),("c",17),("a",42), \
... ("b",4),("d",9),("e",17),("c",2),("f",29),("g",21),("b",9)])
>>> d1.reduceByKey(lambda a,b:a+b).sortByKey(False).collect()
[('g', 21), ('f', 29), ('e', 17), ('d', 9), ('c', 27), ('b', 38), ('a', 42)]
```

```
>>> d1 = sc.parallelize([("c",8),("b",25),("c",17),("a",42), \
... ("b",4),("d",9),("e",17),("c",2),("f",29),("g",21),("b",9)])
>>> d1.reduceByKey(lambda a,b:a+b).sortBy(lambda x:x,False).collect()
[('g', 21), ('f', 29), ('e', 17), ('d', 9), ('c', 27), ('b', 38), ('a', 42)]
>>> d1.reduceByKey(lambda a,b:a+b).sortBy(lambda x:x[0],False).collect()
[('g', 21), ('f', 29), ('e', 17), ('d', 9), ('c', 27), ('b', 38), ('a', 42)]
>>> d1.reduceByKey(lambda a,b:a+b).sortBy(lambda x:x[1],False).collect()
[('a', 42), ('b', 38), ('f', 29), ('c', 27), ('g', 21), ('e', 17), ('d', 9)]
```

4.2.2 常用的键值对RDD转换操作

•**mapValues(func)**

对键值对RDD中的每个value都应用一个函数，但是，key不会发生变化

(Hadoop,1)
(Spark,1)
(Hive,1)
(Spark,1)

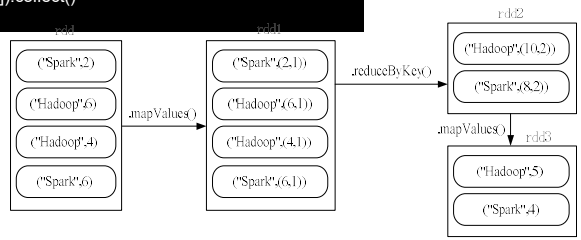
```
>>> list = [("Hadoop",1),("Spark",1),("Hive",1),("Spark",1)]
>>> pairRDD = sc.parallelize(list)
>>> pairRDD1 = pairRDD.mapValues(lambda x:x+1)
>>> pairRDD1.foreach(print)
('Hadoop', 2)
('Spark', 2)
('Hive', 2)
('Spark', 2)
```

2.3 一个综合实例

一个综合实例

题目：给定一组键值对("spark",2),("hadoop",6),("hadoop",4),("spark",6)，键值对的key表示图书名称，value表示某天图书销量，请计算每个键对应的平均值，也就是计算每种图书的每天平均销量。

```
>>> rdd = sc.parallelize([("spark",2),("hadoop",6),("hadoop",4),("spark",6)])
>>> rdd.mapValues(lambda x:(x,1)).\
... reduceByKey(lambda x,y:(x[0]+y[0],x[1]+y[1])).\
... mapValues(lambda x:x[0]/x[1]).collect()
[('hadoop', 5.0), ('spark', 4.0)]
```



2.2 常用的键值对RDD转换操作

•**join**

join就表示内连接。对于内连接，对于给定的两个输入数据集(K,V1)和(K,V2)，只有在两个数据集中都存在的key才会被输出，最终得到一个(K,(V1,V2))类型的数据集。

```
>>> pairRDD1 = sc. \
... parallelize([("spark",1),("spark",2),("hadoop",3),("hadoop",5)])
>>> pairRDD2 = sc.parallelize([("spark","fast")])
>>> pairRDD3 = pairRDD1.join(pairRDD2)
>>> pairRDD3.foreach(print)
('spark', (1, 'fast'))
('spark', (2, 'fast'))
```

03

文件数据读写

4.3.1 文件数据读写

1. 本地文件系统的数据读写
2. 分布式文件系统HDFS的数据读写

4.3.1 文件数据读写

1.本地文件系统的数据读写

（2）把RDD写入到文本文件中

```
>>> textFile = sc.\
... textFile("file:///usr/local/spark/mycode/rdd/word.txt")
>>> textFile.\
... saveAsTextFile("file:///usr/local/spark/mycode/rdd/writeback")
```

```
$ cd /usr/local/spark/mycode/wordcount/writeback/
$ ls
```

```
part-00000
_SUCCESS
```

如果想再次把数据加载在RDD中，只要使用writeback这个目录即可，如下：

```
>>> textFile = sc.\
... textFile("file:///usr/local/spark/mycode/rdd/writeback")
```

3.1 文件数据读写

1.本地文件系统的数据读写

（1）从文件中读取数据创建RDD

```
>>> textFile = sc.\
... textFile("file:///usr/local/spark/mycode/rdd/word.txt")
>>> textFile.first()
'Hadoop is good'
```

因为Spark采用了惰性机制，在执行转换操作的时候，即使输入了错误的语句，spark-shell也不会马上报错（假设word123.txt不存在）

```
>>> textFile = sc.\
...
textFile("file:///usr/local/spark/mycode/wordcount/word123.txt")
```

4.3.1 文件数据读写

2.分布式文件系统HDFS的数据读写

从分布式文件系统HDFS中读取数据，也是采用textFile()方法，可以为textFile()方法提供一个HDFS文件或目录地址，如果是一个文件地址，它会加载该文件，如果是一个目录地址，它会加载该目录下的所有文件的数据

```
>>> textFile = sc.textFile("hdfs://localhost:9000/user/hadoop/word.txt")
>>> textFile.first()
```

如下三条语句都是等价的：

```
>>> textFile = sc.textFile("hdfs://localhost:9000/user/hadoop/word.txt")
>>> textFile = sc.textFile("/user/hadoop/word.txt")
>>> textFile = sc.textFile("word.txt")
```

同样，可以使用saveAsTextFile()方法把RDD中的数据保存到HDFS文件中，命令如下：

```
>>> textFile = sc.textFile("word.txt")
>>> textFile.saveAsTextFile("writeback")
```

3.2 读写HBase数据

0. HBase简介
1. 创建一个HBase表
2. 配置Spark
3. 编写程序读取HBase数据
4. 编写程序向HBase写入数据

3.2 读写HBase数据

执行该代码文件，命令如下：

```
$ cd /usr/local/spark/mycode/rdd
$ /usr/local/spark/bin/spark-submit SparkOperateHBase.py
```

执行后得到如下结果：

```
1 {"qualifier" : "age", "timestamp" : "1545728145163", "columnFamily" : "info",
"row" : "1", "type" : "Put", "value" : "23"}
{"qualifier" : "gender", "timestamp" : "1545728114020", "columnFamily" :
"info", "row" : "1", "type" : "Put", "value" : "F"}
{"qualifier" : "name", "timestamp" : "1545728100663", "columnFamily" : "info",
"row" : "1", "type" : "Put", "value" : "Xueqian"}
2 {"qualifier" : "age", "timestamp" : "1545728184030", "columnFamily" : "info",
"row" : "2", "type" : "Put", "value" : "24"}
{"qualifier" : "gender", "timestamp" : "1545728176815", "columnFamily" :
"info", "row" : "2", "type" : "Put", "value" : "M"}
{"qualifier" : "name", "timestamp" : "1545728168727", "columnFamily" : "info",
"row" : "2", "type" : "Put", "value" : "Weiliang"}
```

3.2 读写HBase数据

编写程序读取HBase

如果要让Spark读取HBase表的内容以RDD的形式加载到内存，就可以使用newAPIHadoopRDD API将HBase表的内容以RDD的形式加载到Spark中。

SparkOperateHBase.py

```
#!/usr/bin/env python3
from pyspark import SparkConf, SparkContext
conf = SparkConf().setMaster("local").setAppName("ReadHBase")
sc = SparkContext(conf = conf)
host = 'localhost'
table = 'student'
conf = {"hbase.zookeeper.quorum": host, "hbase.mapreduce.inputtable": table}
keyConv =
"org.apache.spark.examples.pythonconverters.ImmutableBytesWritableToStringC
onverter"
valueConv =
"org.apache.spark.examples.pythonconverters.HBaseResultToStringConverter"
hbase_rdd =
sc.newAPIHadoopRDD("org.apache.hadoop.hbase.mapreduce.TableInputFormat
","org.apache.hadoop.hbase.io.ImmutableBytesWritable","org.apache.hadoop.hb
ase.client.Result",keyConverter=keyConv,valueConverter=valueConv,conf=conf)
count = hbase_rdd.count()
hbase_rdd.cache()
output = hbase_rdd.collect()
for (k, v) in output:
        print (k, v)
```

4.3.2 读写HBase数据

4. 编写程序向HBase写入数据

下面编写应用程序把表中的两个学生信息插入到HBase的student表中

表　向student表中插入的新数据

id	info		
	name	gender	age
3	Rongcheng	M	26
4	Guanhua	M	27

3.2 读写HBase数据

在SparkWriteHBase.py文件中输入下面代码：

```
#!/usr/bin/env python3

from pyspark import SparkConf, SparkContext

conf = SparkConf().setMaster("local").setAppName("ReadHBase")
sc = SparkContext(conf = conf)
host = 'localhost'
table = 'student'
keyConv = "org.apache.spark.examples.pythonconverters.StringToImmutableBytesWritableConverter"
valueConv = "org.apache.spark.examples.pythonconverters.StringListToPutConverter"
conf = {"hbase.zookeeper.quorum": host,"hbase.mapred.outputtable": table,"mapreduce.outputformat.class":
"org.apache.hadoop.hbase.mapreduce.TableOutputFormat","mapreduce.job.output.key.class":
"org.apache.hadoop.hbase.io.ImmutableBytesWritable","mapreduce.job.output.value.class":
"org.apache.hadoop.io.Writable"}
rawData =
['3,info,name,Rongcheng','3,info,gender,M','3,info,age,26','4,info,name,Guanhua','4,info,gender,M','4,info,age,27']
sc.parallelize(rawData).map(lambda x:
(x[0],x.split(','))).saveAsNewAPIHadoopDataset(conf=conf,keyConverter=keyConv,valueConverter=valueConv)
```

04

综合案例

3.2 读写HBase数据

```
$ cd /usr/local/spark/mycode/rdd
$ /usr/local/spark/bin/spark-submit SparkWriteHBase.py
```

切换到HBase Shell中，执行如下命令查看student表

```
hbase> scan 'student'
```

```
ROW                    COLUMN+CELL
 1                     column=info:age, timestamp=1479640712163, value=23
 1                     column=info:gender, timestamp=1479640704522, value=F
 1                     column=info:name, timestamp=1479640696132, value=Xueqian
 2                     column=info:age, timestamp=1479640752474, value=24
 2                     column=info:gender, timestamp=1479640745276, value=M
 2                     column=info:name, timestamp=1479640732763, value=Weiliang
 3                     column=info:age, timestamp=1479643273142,
value=\x00\x00\x00\x1A
 3                     column=info:gender, timestamp=1479643273142, value=M
 3                     column=info:name, timestamp=1479643273142,
value=Rongcheng
 4                     column=info:age, timestamp=1479643273142,
value=\x00\x00\x00\x1B
 4                     column=info:gender, timestamp=1479643273142, value=M
```

4.4 综合案例

4.4.1 案例1：求TOP值

任务描述：

orderid,userid,payment,productid

file1.txt

```
1,1768,50,155
2,1218, 600,211
3,2239,788,242
4,3101,28,599
5,4899,290,129
6,3110,54,1201
7,4436,259,877
8,2369,7890,27
```

file2.txt

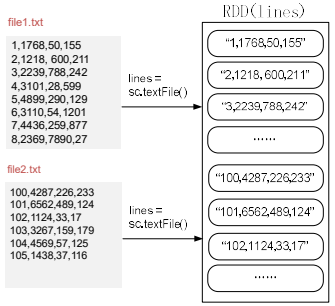
```
100,4287,226,233
101,6562,489,124
102,1124,33,17
103,3267,159,179
104,4569,57,125
105,1438,37,116
```

求Top N个payment值

4.4.1 案例1：求TOP值

```
lines = sc.textFile("file:///usr/local/spark/mycode/rdd/file")
```

该语句从文件中读取数据生成RDD（名称为lines），执行后的效果如下：



4.4.1 案例1：求TOP值

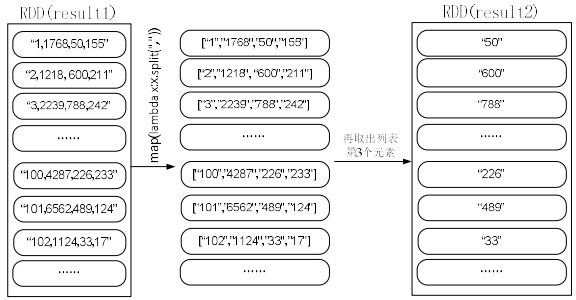
TopN.py

```
#!/usr/bin/env python3

from pyspark import SparkConf, SparkContext

conf = SparkConf().setMaster("local").setAppName("ReadHBase")
sc = SparkContext(conf = conf)
lines = sc.textFile("file:///usr/local/spark/mycode/rdd/file")
result1 = lines.filter(lambda line:(len(line.strip()) > 0) and (len(line.split(","))== 4))
result2 = result1.map(lambda x:x.split(",")[2])
result3 = result2.map(lambda x:(int(x),""))
result4 = result3.repartition(1)
result5 = result4.sortByKey(False)
result6 = result5.map(lambda x:x[0])
result7 = result6.take(5)
for a in result7:
    print(a)
```

案例1：求TOP值

```
result1 = lines.filter(lambda line:(len(line.strip()) > 0) and
(len(line.split(","))== 4))
result2 = result1.map(lambda x:x.split(",")[2])
```

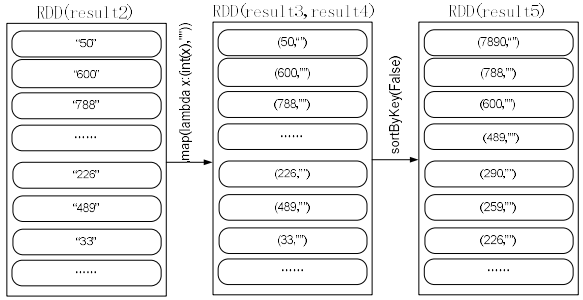
该语句执行效果如下：



4.1 案例1：求TOP值

```
result3 = result2.map(lambda x:(int(x),""))
result4 = result3.repartition(1)
result5 = result4.sortByKey(False)
```

该语句执行效果如下：



2 案例2：文件排序

任务描述：

有多个输入文件，每个文件中的每一行内容均为一个整数。要求读取所有文件中的整数，进行排序后，输出到一个新的文件中，输出的内容个数为每行两个整数，第一个整数为第二个整数的排序位次，第二个整数为原待排序的整数

输入文件

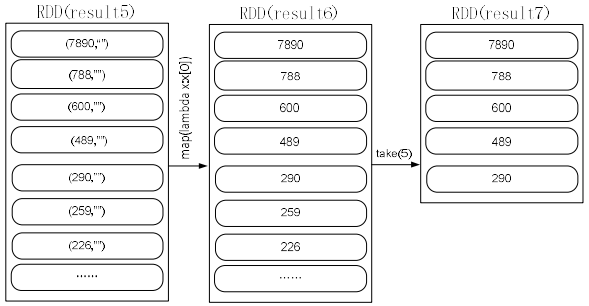
file1.txt	file2.txt	file3.txt
33	4	1
37	16	45
12	39	25
40	5	

写出输出文件

4.1 案例1：求TOP值

```
result6 = result5.map(lambda x:x[0])
result7 = result6.take(5)
```

该语句执行效果如下：



4.2 案例2：文件排序

FileSort.py

```
def getindex():
    global index
    index+=1
    return index
```

file1.txt

33
37
12
40

file2.txt

4
16
39
5

file3.txt

1
45
25

```
#!/usr/bin/env python3

from pyspark import SparkConf, SparkContext

index = 0

def main():
    conf = SparkConf().setMaster("local[1]").setAppName("FileSort")
    sc = SparkContext(conf = conf)
    lines = sc.textFile("file:///usr/local/spark/mycode/rdd/filesort/file*.txt")
    index = 0
    result1 = lines.filter(lambda line:(len(line.strip()) > 0))
    result2 = result1.map(lambda x:(int(x.strip()),""))
    result3 = result2.repartition(1)
    result4 = result3.sortByKey(True)
    result5 = result4.map(lambda x:x[0])
    result6 = result5.map(lambda x:(getindex(),x))
    result6.foreach(print)
    result6.saveAsTextFile("file:///usr/local/spark/mycode/rdd/filesort/sortresult")
if __name__ == '__main__':
    main()
```


Thank You!