6.7 PySpark与pandas整合 中文讲课稿

슬라이드1: 6.7 课程目录介绍

大家好，今天我们来学习一个非常实用的内容——PySpark与pandas的整合。

那么我们先看一下今天的课程安排。

首先呢，我们要了解PySpark与pandas进行整合的可行性，也就是说为什么要把这两个技术结合起来用。

然后呢，我们要复习一下pandas的数据结构(data structure)，因为有些同学可能对pandas还不太熟悉。

接下来，我们会通过两个实例来学习具体的操作：

第一个实例是两种DataFrame(数据框)之间的相互转化，

第二个实例是使用自定义聚合函数(custom aggregation function)。

好，那我们现在开始第一部分的内容。

슬라이드2: 6.7.1 整合的可行性 - pandas介绍

首先我们来了解一下什么是pandas。

pandas呢，是基于NumPy的一个开源Python库(open source library)，它被广泛用于快速分析数据以及数据清洗和准备等工作。pandas融入了大量库和标准数据模型，能够提供高效的操作数据集所需的工具，同时提供了大量能使我们快速便捷地处理数据的函数和方法。

那么，尽管PySpark和pandas在实现上有所不同，但它们都是用于数据处理和分析的流行工具。pandas是基于NumPy的库，提供了灵活且高效的数据结构，如Series(序列)和DataFrame(数据框)，使得数据操作变得简单和直观。

另一方面，PySpark是基于Apache Spark的Python API，它适用于大规模数据处理和分析。它提供了分布式计算(distributed computing)的能力，并可以在集群(cluster)上运行，处理大量的数据。

슬라이드3: 6.7.1 API相似性与数据格式支持

大家注意啊，pandas的DataFrame和PySpark的DataFrame都提供了类似的API(应用程序编程接口)，可以用于数据的转换、过滤、聚合等操作。这使得开发人员可以在不同的环境中使用相同的操作和技术，无需重写代码。

此外，两者都支持常见的数据格式，如CSV、JSON和Parquet等，这使得数据的导入和导出变得更加方便。

那么这就为我们提供了一个很好的机会，可以在一个项目中同时使用两种技术的优势。

슬라이드4: 实际应用场景

现在我给大家举一个实际的应用场景。

假设有一个包含大量数据的CSV文件，需要对该数据进行分析和处理。首先，可以使用pandas读取CSV文件并将其转换为pandas的DataFrame。然后，可以使用pandas的API进行一些数据清洗和转换的操作。

一旦完成了初步的数据清洗和转换，就可以将pandas的DataFrame转换为PySpark的DataFrame，这可以通过使用PySpark的createDataFrame()方法来实现。

一旦数据转换完成，就可以使用PySpark的API进行进一步的分析和处理，这其中就会涉及到复杂的计算和分布式处理，使得我们可以充分利用Spark的分布式计算能力。

完成分析和处理后，可以将结果转换回pandas的DataFrame，并使用pandas的API进行数据可视化和进一步的分析。这样可以方便地使用pandas提供的丰富的图表和统计功能，展示分析结果。

슬라이드4: 6.7.2 pandas数据结构 - 安装与介绍

好，现在我们来复习一下pandas数据结构。

首先呢，pandas是一个Python的第三方模块，需要单独安装。大家可以执行如下命令进行安装：

先激活我们的pyspark环境，然后用pip安装pandas。

```
$ conda activate pyspark
$ pip install pandas
```

pandas数据结构主要包括Series和DataFrame两种。

让我们先来看Series。Series是一种类似于一维数组的对象，它由一维数组以及一组与之相关的数据标签（即索引(index)）组成，仅由一组数据即可产生最简单的Series。

Series的字符串表现形式为：索引在左边，值在右边。如果没有为数据指定索引，就会自动创建一个0到N-1(N为数据的长度)的整数型索引。可以通过Series的values和index属性获取其数组表现形式和索引对象。

슬라이드5: 6.7.2. Pandas 数据结构-Series实例演示

现在我们来看一个具体的实例。

```
>>> import numpy as np
>>> import pandas as pd
>>> from pandas import Series,DataFrame
>>> obj=Series([3,5,6,8,9,2])
>>> obj
0    3
1    5
2    6
```

```
3    8
4    9
5    2
```

大家看这个代码，我们首先导入了numpy、pandas，以及从pandas中导入Series和DataFrame。

然后我们创建了一个Series对象，传入了一个列表[3,5,6,8,9,2]。

执行结果大家可以看到，左边是索引，从0开始，右边是我们的数据值。因为我们没有指定索引，所以pandas自动为我们创建了整数型索引。

슬라이드 6: 6.7.2. Pandas 数据结构-自定义索引的 Series

现在我们来创建一个带有自定义索引的Series。

```
>>> obj2=Series([3,5,6,8,9,2],index=['a','b','c','d','e','f'])
>>> obj2
a    3
b    5
c    6
d    8
e    9
f    2
```

在这个例子中，我们在创建Series的时候，通过index参数指定了索引为['a','b','c','d','e','f']。

大家看结果，现在索引变成了字母，值还是我们原来的数据。这样我们就可以通过这些字母来访问对应的数据了。

슬라이드7: : 6.7.2. Pandas 数据结构-Series的索引操作

创建好Series以后，我们可以利用索引的方式选取Series的单个或一组值。

```
>>> obj2['a']
3
>>> obj2[['b','d','f']]
b    5
d    8
f    2
```

比如说，obj2['a']可以获取索引为'a'的值，结果是3。

如果我们想要获取多个值，可以传入一个列表，比如obj2[['b','d','f']]，这样就能获取索引为'b','d','f'的三个值。

大家看，这种索引操作非常灵活，既可以获取单个值，也可以获取多个值。

여기서 2개의 실습을 한꺼번에 수행함.

```
from pandas import Series

if __name__=='__main__':
    # 슬라이드5: 6.7.2. Pandas 数据结构-Series实例演示
    # 没有为数据指定索引
    print('没有为数据指定索引')
    obj=Series([3,5,6,8,9,2])
    print(obj)

    # 슬라이드6: 6.7.2Pandas 数据结构 - 自定义索引的Series
    # 标记的索引
    print('标记的索引')
    obj2=Series([3,5,6,8,9,2], index=['a','b','c','d','e','f'])
    print(obj2)
```

```
# 슬라이드7: : 6.7.2. Pandas 数据结构-Series的索引操作
# 可以利用索引的方式选取Series的单个或一组值
print(obj2['a'])
print(obj2[['b','d','f']])
```

슬라이드8 : : 6.7.2. Pandas 数据结构-DataFrame介绍

现在我们来看DataFrame。

DataFrame是一个表格型的数据结构，它含有一组有序的列，每列可以是不同的值类型（数值、字符串、布尔值等）。DataFrame既有行索引也有列索引，它可以被看作由Series组成的字典（共用同一个索引）。

跟其他类似的数据结构相比，DataFrame中面向行和面向列的操作基本是平衡的。其实，DataFrame中的数据是以一个或多个二维块存放的（而不是列表、字典或者别的一维数据结构）。

pandas提供了DataFrame()函数来构建DataFrame。

슬라이드 9: : 6.7.2. Pandas 数据结构-DataFrame实例演示

现在我们来看一个具体的DataFrame实例。

```
>>> import numpy as np
>>> import pandas as pd
>>> from pandas import Series, DataFrame
>>> data = {'sno':['95001', '95002', '95003', '95004'],
        'name':['Xiaoming','Zhangsan','Lisi','Wangwu'],
        'sex':['M','F','F','M'],
        'age':[22,25,24,23]}
>>> frame=DataFrame(data)
>>> frame
     sno       name   sex   age
0   95001   Xiaoming     M    22
1   95002   Zhangsan     F    25
2   95003       Lisi     F    24
3   95004     Wangwu     M    23
```

我们首先定义了一个字典data，包含了四个字段：sno(学号)、name(姓名)、sex(性别)、age(年龄)。

然后用DataFrame()函数创建了一个DataFrame对象。

大家看执行结果，这就是一个标准的表格结构，有行有列，左边是自动生成的行索引，上面是我们的列名。

슬라이드10: 6.7.2. Pandas 数据结构-指定列顺序

从执行结果可以看出，虽然没有指定行索引，但是pandas会自动添加索引。

```
>>> frame=DataFrame(data,columns=['name','sno','sex','age'])
```

```
>>> frame
      name    sno  sex  age
0  Xiaoming  95001    M   22
1  Zhangsan  95002    F   25
2      Lisi  95003    F   24
3    Wangwu  95004    M   23
```

如果指定列索引，则会按照指定顺序排列。比如我们这里指定columns=['name','sno','sex','age']，那么列的顺序就会按照我们指定的来排列。

大家看，现在name列排在了第一位，然后是sno、sex、age。

슬라이드11: 6.7.2. Pandas 数据结构-同时指定行索引和列索引

我们还可以同时指定行索引和列索引。

```
>>> frame=DataFrame(data,columns=['sno','name','sex','age','grade'],￦
... index=['a','b','c','d'])
>>> frame
     sno      name sex  age grade
a  95001  Xiaoming   M   22   NaN
b  95002  Zhangsan   F   25   NaN
c  95003      Lisi   F   24   NaN
d  95004    Wangwu   M   23   NaN
```

在这个例子中，我们指定了columns包含一个新的列'grade'，虽然我们的原始数据中没有这个列，pandas会自动用NaN(Not a Number)来填充。

同时我们指定了index=['a','b','c','d']，这样行索引就变成了字母。

大家注意grade列都是NaN，这表示缺失值。

슬라이드12: 6.7.2. Pandas 数据结构-获取列数据

通过类似字典标记或属性的方式，我们可以获取Series（列数据）。

```
>>> frame['sno']
a    95001
b    95002
c    95003
d    95004
>>> frame.name
a    Xiaoming
b    Zhangsan
c        Lisi
d     Wangwu
```

比如frame['sno']可以获取sno列的所有数据，返回的是一个Series。

或者我们可以用属性的方式，frame.name来获取name列的数据。

这两种方式都很常用，大家可以根据自己的习惯来选择。

슬라이드13：6.7.2. Pandas 数据结构-获取行数据

行也可以通过位置或名称获取。

```
>>> frame.loc['b']
sno          95002
name      Zhangsan
sex              F
age             25
grade          NaN
```

比如frame.loc['b']可以获取索引为'b'的行，返回的是一个Series，包含了这一行的所有数据。

大家看，结果显示了这一行的所有字段：sno是95002，name是Zhangsan，性别是F，年龄是25，grade是NaN。

여기서 앞의 실습을 한꺼번에 수행함.

```
from pandas import DataFrame
if __name__=='__main__':

    # 슬라이드 9: : 6.7.2. Pandas 数据结构-DataFrame实例演示
    # 给定的代码是一个 Python字典（dictionary）
    data = {'sno': ['95001', '95002', '95003', '95004'],
            'name': ['Xiaoming', 'Zhangsan', 'Lisi', 'Wangwu'],
            'gender': ['M', 'F', 'F', 'M'],
            'age': [22, 25, 24, 23]}
    frame = DataFrame(data)
    print('没有指定行索引')
    print(frame)
    print('₩n')   #反斜杠(Fǎn xié gàng)

    # 슬라이드10: 6.7.2. Pandas 数据结构-指定列顺序
    # 指定列索引
    print('  指定列索引')
    frame2=DataFrame(data,columns=['name','sno','gender','age','grade'])
    print(frame2)
    print('₩n')

    # 슬라이드11: 6.7.2. Pandas 数据结构-同时指定行索引和列索引
    #同时指定行索引和列索引
    print('同时指定行索引和列索引')
    frame3=DataFrame(data,columns=['sno','name','gender','age','grade'],
                           index=['a','b','c','d'])
    print(frame3)
    print('₩n')

    # 슬라이드12: 6.7.2. Pandas 数据结构-获取列数据
    # 通过类似字典标记或属性的方式，可以获取 Series
    print('通过类似字典标记或属性的方式，可以获取 Series')
```

```
print(frame3['sno'])
print('₩n')
print(frame3.sno)
print('₩n')

# 슬라이드13：6.7.2. Pandas 数据结构-获取行数据
# 行也可以通过位置或名称进行获取
print('行也可以通过位置或名称进行获取')
print(frame3.loc['b'])
print('₩n')
```

슬라이드14-15: 6.7.3 实例1 - DataFrame相互转换

好，现在我们进入第一个实例：两种DataFrame之间的相互转化。

这个实例很重要，因为在实际工作中，我们经常需要在pandas和PySpark之间转换数据。

首先我们导入必要的模块，包括SparkSession、数据类型，还有pandas。

然后我们创建SparkSession，这是使用PySpark的第一步。

接下来我们定义一个pandas的DataFrame，包含id、name、age三个字段。

然后我们定义PySpark DataFrame的数据结构，这叫做schema(模式)，需要明确指定每个字段的类型。

转换过程分为几个步骤：

1. 首先我们打印pandas的DataFrame
2. 然后将pandas DataFrame转换为PySpark DataFrame，使用spark.createDataFrame()方法
3. 显示PySpark DataFrame，使用show()方法
4. 再将PySpark DataFrame转化为pandas DataFrame，使用toPandas()方法
5. 最后再次显示pandas DataFrame

这样我们就完成了一个完整的转换循环，证明了两种DataFrame可以相互转换。

```
from pyspark.sql import SparkSession
from pyspark.sql.types import IntegerType, StringType, StructField, StructType
import pandas as pd

if __name__ == '__main__':
    spark = SparkSession ₩
        .builder ₩
        .master("local[*]") ₩
        .appName("case1:PySparkPandasDataFrame") ₩
        .getOrCreate()

    # 定义pandas的DataFrame
    pd_df = pd.DataFrame({
        'id': [1, 2, 3, 4, 5],
        'name': ['zhangsan ', 'lisi', 'wangwu', 'maliu', 'liuqi'],
        'age': [24, 26, 22, 28, 24]
    }
    )

    # 展示pandas中的DataFrame
    print('1.展示pandas中的DataFrame')
    print(pd_df)
    print('₩n')

    # 定义PySpark的DataFrame的数据结构
    schema = StructType([StructField("id", IntegerType(), True), ₩
                         StructField("name", StringType(), True), ₩
                         StructField("age", IntegerType(), True)])
    # 将pandas中的DataFrame转换为PySPark中的DataFrame
    spark_df = spark.createDataFrame(pd_df, schema)
    # 展示PySpark中的DataFrame
    print('2.展示PySpark中的DataFrame')
    spark_df.show()
    print('₩n')

    # 将PySpark中的DataFrame转化为pandas中的DataFrame
    pd_df2 = spark_df.toPandas()
    # 展示pandas中的DataFrame
```

```
print('3.展示pandas中的DataFrame')
print(pd_df2)
```

슬라이드16-: 6.7.3 实例2 - 自定义聚合函数准备

现在我们来看第二个实例：使用自定义聚合函数。

假设有一个文本文件people_data.txt，数据内容是用逗号分隔的：id、姓名、年龄。

我们的任务是计算出所有人的年龄总和。

首先，我们需要安装PyArrow。PyArrow是什么呢？Apache Arrow是一个高效的列式数据格式，可以在PySpark中实现JVM和Python进程之间的数据交换。这对于我们使用pandas UDF(用户定义函数)是必需的。

슬라이드17-18: 6.7.3 实例 自定义聚合函数实现

现在我们来看具体的代码实现。

首先我们读取数据文件，将文本文件转换为RDD，然后转换为DataFrame。

关键部分是自定义函数的注册，这需要两个步骤：

第一步是使用装饰器@F.pandas_udf(IntegerType())进行装饰。

这告诉PySpark这是一个pandas UDF(user defined function)，返回类型是整数。

第二步是定义我们的自定义函数myFunction。注意，这个函数接收的参数a是pd.Series类型，因为它接收的是整列数据。我们在函数内部对这一列数据进行求和操作。

슬라이드 19: 6.7.3 实例 函数注册和使用

接下来我们注册这个自定义函数：

使用spark.udf.register()方法将我们的函数注册为"sum_func"。

然后我们可以在两种语法中使用这个函数：

1. DSL风格语法：peopleDF.select(sum_func("age"))
2. SQL风格语法：先创建临时视图，然后在SQL语句中使用

两种方式都能得到相同的结果，大家可以根据自己的偏好选择(실습은 2가지 모두를 알려줌)

```
from pyspark.sql import SparkSession, functions as F
from pyspark.sql.types import IntegerType, StringType, StructField, StructType, Row
import pandas as pd

if __name__ == '__main__':

    spark = SparkSession ₩
        .builder ₩
        .master("local[*]") ₩
        .appName("Case02:PysparkPandasDataFrame Application") ₩
        .getOrCreate()

    peopleRDD = spark.sparkContext ₩
        .textFile("file:///home/hadoop/people_data.txt") ₩
        .map(lambda line: line.split(",")) ₩
        .map(lambda p: Row(id=int(p[0]), name=p[1], age=int(p[2])))
    schema = StructType([StructField("id", IntegerType(), True), ₩
                         StructField("name", StringType(), True), ₩
                         StructField("age", IntegerType(), True)])
    peopleDF = peopleRDD.toDF(schema)

    print('创建的PySpark DataFrame:')
    print(peopleDF.show())
    print('₩n')


    # 注册自定义函数，需要包括两个步骤
    # 第1步是使用装饰器@进行装饰
    @F.pandas_udf(IntegerType())
    # 第2步是定义自定义函数
    def myFunction(a: pd.Series) -> int:
        # a是接收参数，接收的是整列数据(전체 열 데이터)，所以需要指定数据类型为pd.Series
        # 对接收到的a进行计算
```

```
        result = a.sum()
        return result


# 注册自定义函数的第2步：使用PySpark的register方法进行注册
sum_func = spark.udf.register("sum_func", myFunction)
# 在DSL风格语法中使用自定义函数
print('在DSL风格语法中使用自定义函数')
sumDF = peopleDF.select(sum_func("age"))
sumDF.show()
print('₩n')
# 在SQL风格语法中使用自定义函数
peopleDF.createOrReplaceTempView("people")
sumDF2 = spark.sql("select sum_func(age) from people")
print('在SQL风格语法中使用自定义函数')
sumDF2.show()
print('₩n')
spark.stop()
```

슬라이드20 6.8: 综合实例介绍

现在我们来看一个更复杂的综合实例。

这是一个电影评分数据集，数据格式是用双冒号分隔的四个字段：用户ID、电影ID、电影评分、时间戳。

这种格式在实际的数据分析项目中很常见，特别是推荐系统的数据。

슬라이드21 6.8: 分析任务

基于这个数据集，我们需要解决以下问题：

1. 求每个用户的平均打分
2. 求每部电影的平均打分
3. 查询大于平均打分的电影的数量
4. 查询高分（大于3分）电影中打分次数最多的用户，给出此人打分的平均值
5. 查询每个用户的平均打分、最低打分和最高打分
6. 查询打分次数超过100次的电影的平均分排行榜的TOP10

这些都是实际数据分析中经常遇到的问题。

슬라이드 22 6.8: 数据读取和基础分析

我们先看数据读取部分。

由于数据是用双冒号分隔的，我们需要设置separator为"::"。同时我们定义了schema来指定数据类型。

对于第2个问题，求每个用户的平均打分，我们使用groupby操作，然后调用avg()函数。注意我们还使用了withColumnRenamed()来重命名列，使用round()函数来控制小数位数。

对于第3个问题，我们展示了两种方法：DSL风格和SQL风格。两种方法都能达到相同的效果。

슬라이드23 6.8: 复杂查询

第4个问题比较复杂，我们需要先找到大于平均打分的电影数量。这里我们使用了嵌套查询，先计算平均分，然后统计大于平均分的记录数。

第5个问题更有挑战性：我们需要找到高分电影中打分次数最多的用户。这需要多步操作：

1. 先过滤出评分大于3的记录
2. 按用户分组统计打分次数
3. 排序找出打分次数最多的用户

4. 计算这个用户的平均评分

슬라이드27: 聚合操作

第6个问题展示了复杂的聚合操作，我们使用agg()函数同时计算多个统计量：平均值、最小值、最大值。

第7个问题是一个实际的业务场景：找出评价次数足够多（超过100次）的电影中评分最高的TOP10。这需要先按电影分组，统计评分次数和平均评分，然后过滤和排序。

```
from pyspark.sql.types import IntegerType, StringType, StructField, StructType
from pyspark.sql import SparkSession
import pyspark.sql.functions as F

if __name__ == '__main__':
    spark = SparkSession ₩
        .builder ₩
        .master("local[*]") ₩
        .appName("综合实例：Movie rating Application") ₩
        .getOrCreate()

    # 0.读取数据集 (DataFrame API)
    filePath = "file:///home/hadoop/ratings.dat"
    schema = StructType([StructField("user_id", StringType(), True), ₩
                         StructField("movie_id", StringType(), True), ₩
                         StructField("rating", IntegerType(), True), ₩
                         StructField("ts", StringType(), True)])
    ratingsDF = spark.read.format("csv"). ₩
        option("sep", "::"). ₩
        option("header", False). ₩
        option("encoding", "utf-8"). ₩
        schema(schema). ₩
        load(filePath)

    print("0.数据集如下：")
    ratingsDF.show()
    ratingsDF.createTempView("ratings")
```

```
# 1. 求每个用户的平均打分（DataFrame API）
print("1.求每个用户的平均打分")
ratingsDF.groupby("user_id"). ₩
    avg("rating"). ₩
    withColumnRenamed("avg(rating)", "avg_rating"). ₩
    withColumn("avg_rating", F.round("avg_rating", 3)). ₩
    orderBy("avg_rating", ascending=False). ₩
    show()

# 2. 求每部电影的平均打分（Spark SQL）
print("2.求每部电影的平均打分")
ratingsDF.createTempView("movie_ratings")
spark.sql("""
    SELECT movie_id, ROUND(AVG(rating),3) AS avg_rating
    FROM movie_ratings
    GROUP BY movie_id
    ORDER BY avg_rating DESC
""").show()

# 3.查询大于平均打分的电影的数量（DataFrame API）
print("3.查询大于平均打分的电影的数量")
movieCount = ratingsDF.where(
    ratingsDF["rating"] > ratingsDF.select(F.avg(ratingsDF["rating"])).first()["avg(rating)"]).count()
print("大于平均打分的电影的数量是：", movieCount)

# 4. 查询高分（大于3分）电影中打分次数最多的用户，给出此人打分的平均值（DataFrame API）
print("4.查询高分（大于3分）电影中打分次数最多的用户，给出此人打分的平均值")
user_id = ratingsDF.where(ratingsDF["rating"] > 3). ₩
    groupBy("user_id"). ₩
    count(). ₩
    withColumnRenamed("count", "high_rating_count"). ₩
    orderBy("high_rating_count", ascending=False). ₩
    limit(1). ₩
    first()["user_id"]

ratingsDF.filter(ratingsDF["user_id"] == user_id). ₩
    select(F.round(F.avg("rating"), 3)).show()
```

```
# 5.查询每个用户的平均打分、最低打分和最高打分（DataFrame API）
print("5.查询每个用户的平均打分、最低打分和最高打分")
ratingsDF.groupBy("user_id"). ₩
    agg(
    F.round(F.avg("rating"), 3).alias("avg_rating"),
    F.round(F.min("rating"), 3).alias("min_rating"),
    F.round(F.max("rating"), 3).alias("max_rating")
).show()

# 6. 查询打分次数超过100次的电影的平均分排行榜的TOP10（DataFrame API）
print("6.查询打分次数超过100次的电影的平均分排行榜的TOP10")
ratingsDF.groupBy("movie_id"). ₩
    agg(
    F.count("movie_id").alias("cnt"),
    F.round(F.avg("rating"), 3).alias("avg_rating")
).where("cnt>100"). ₩
    orderBy("avg_rating", ascending=False). ₩
    limit(10). ₩
    show()

spark.stop()
```

슬라이드26: 6.9 课程总结

好，最后我们来总结一下今天学习的内容。

在大数据处理框架之上提供SQL支持，一方面可以简化开发人员的编程工作，另一方面可以用大数据技术实现对结构化数据的高效复杂分析。

我们今天学习了Spark SQL的数据模型DataFrame，它是一个由多个列组成的结构化的分布式数据集合，相当于关系数据库中的一张表。DataFrame可以通过多种方式创建，包括结构化的数据集、Hive表、外部数据库或者RDD等。

我们还学习了如何在pandas和PySpark之间进行数据转换，以及如何使用自定义聚合函数。最后通过一个电影评分数据的综合实例，展示了如何解决实际的数据分析问题。

这些技能在实际工作中非常有用，特别是当你需要处理大规模数据，但又想利用pandas丰富的分析功能时。

好，今天的课程就到这里，大家有什么问题可以提出来讨论。