6.7 PySpark和pandas的整合

6.7.1 PySpark和pandas进行整合的可行性

6.7.2 pandas数据结构

6.7.3 实例1：两种DataFrame之间的相互转化

6.7.4 实例2：使用自定义聚合函数

6.7.1 PySpark和pandas进行整合的可行性



- pandas是基于NumPy的一个开源Python库，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。

- 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和DataFrame，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的能力，并可以在集群上运行，处理大量的数据。

- pandas的DataFrame和PySpark的DataFrame都提供了类似的API，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。

6.7.1 PySpark和pandas进行整合的可行性



- 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。

6.7.2 pandas数据结构



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

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

pandas数据结构主要包括Series和DataFrame。

1.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）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。

6.7.2 pandas数据结构



下面是具体实例：

```
>>> 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
```

6.7.2 pandas数据结构



上面的代码中，我们没有为数据指定索引，因此，pandas会自动创建一个整型索引。现在，我们创建对数据点进行标记的索引，作为演示，下面继续执行如下代码：

```
>>> 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
```

6.7.2 pandas数据结构



创建好Series以后，可以利用索引的方式选取Series的单个或一组值。作为演示，下面继续执行如下代码：

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

6.7.2 pandas数据结构



2.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。

6.7.2 pandas数据结构



下面是具体实例：

```
>>> 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
```

6.7.2 pandas数据结构



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。如果指定列索引，则会按照指定顺序排列。作为演示，继续执行如下代码：

```
>>> 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
```

6.7.2 pandas数据结构



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
```

6.7.2 pandas数据结构



通过类似字典标记或属性的方式，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
```

6.7.2 pandas数据结构



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
```

6.7.3 实例1：两种DataFrame之间的相互转化



```
# coding:utf8
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("Simple Application")₩
        .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]
    }
    )
```

6.7.3 实例1：两种DataFrame之间的相互转化



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

6.7.4 实例2：使用自定义聚合函数



假设有一个文本文件people_data.txt，其数据内容如下：

```
1,zhangsan,22
2,lisi,26
3,wangwu,28
4,maliu,24
```

要求计算出所有人的年龄的总和。
首先，需要使用如下命令安装第三方模块PyArrow（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）：

```
$ pip install pyarrow
查看已安装的所有包，并筛选 pyarrow
$pip list | grep pyarrow
```

6.7.4 实例2：使用自定义聚合函数



然后，新建一个代码文件PysparkPandasFunction.py，内容如下：

```
# coding:utf8
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("Simple Application")₩
        .getOrCreate()
```

6.7.4 实例2：使用自定义聚合函数



```
    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)
# 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: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，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
```

6.7.4 实例2：使用自定义聚合函数



```
    # 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
    sumDF = peopleDF.select(sum_func("age"))
    sumDF.show()
    # 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")
sumDF2.show()
spark.stop()
```

6.8 综合实例



有一个电影评分数据集，数据格式如下：

```
1::661::3::978302109
1::914::3::978301968
1::3408::4::978300275
```

用户ID、电影ID、电影评分、时间戳

6.8 综合实例



需要根据该数据集求解如下问题：

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

6.8 综合实例



根据上述要求可以设计一个程序MovieRating.py，具体代码如下：

```
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("Simple Application")₩
        .getOrCreate()
    # 1.读取数据集
    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)
```

6.8 综合实例



```
print("1.数据集如下： ")
ratingsDF.show()
ratingsDF.createTempView("ratings")

# 2. 求每个用户的平均打分
print("2.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()

# 3. 求每部电影的平均打分
print("3.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()
```

6.8 综合实例



```
    # 4.查询大于平均打分的电影的数量
    print("4.查询大于平均打分的电影的数量")
    movieCount = ratingsDF.where(ratingsDF["rating"]>ratingsDF.select(F.avg(ratingsDF["ratin
g"])).first()["avg(rating)"]).count()
    print("大于平均打分的电影的数量是：", movieCount)

    # 5. 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
    print(" 5.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()
```

6.8 综合实例



```
# 6.查询每个用户的平均打分、最低打分和最高打分
print("6.查询每个用户的平均打分、最低打分和最高打分")
ratingsDF.groupBy("user_id").⑩
    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()

# 7. 查询打分次数超过100次的电影的平均分排行榜的TOP10
print("7.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").⑩
    agg(
        F.count("movie_id").alias("cnt"),
        F.round(F.avg("rating"),3).alias("avg_rating")
    ).where("cnt>100").⑩
    orderBy("avg_rating",ascending=False).⑩
    limit(10).⑩
    show()
```

6.9 本章小结



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。本章在开头部分介绍的数据仓库Hive，就相当于提供了一种编程语言接口，只要求用户输入SQL语句，它就可以自动把SQL语句转化为底层的MapReduce程序。Shark在设计上完全照搬了Hive，实现了SQL语句到Spark程序的转换。但是，Shark存在很多设计上的缺陷，因此，Spark SQL摈弃了Shark的设计思路，对组件进行了重新设计，获得了较好的性能。

本章介绍了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是Spark SQL中的最基本的概念，可以通过多种方式创建，例如，结构化的数据集、Hive表、外部数据库或者RDD等。创建好DataFrame以后，可以执行一些常用的DataFrame操作，包括printSchema()、select()、filter()、groupBy()和sort()等。从RDD转换得到DataFrame，有时候可以实现自动的隐式转换，但是，有时候需要通过编程的方式实现转换，主要有两种方式，即利用反射机制推断RDD模式和使用编程方式定义RDD模式。

本章最后介绍了通过JDBC连接MySQL数据库的详细过程以及PySpark和pandas的整合方法。