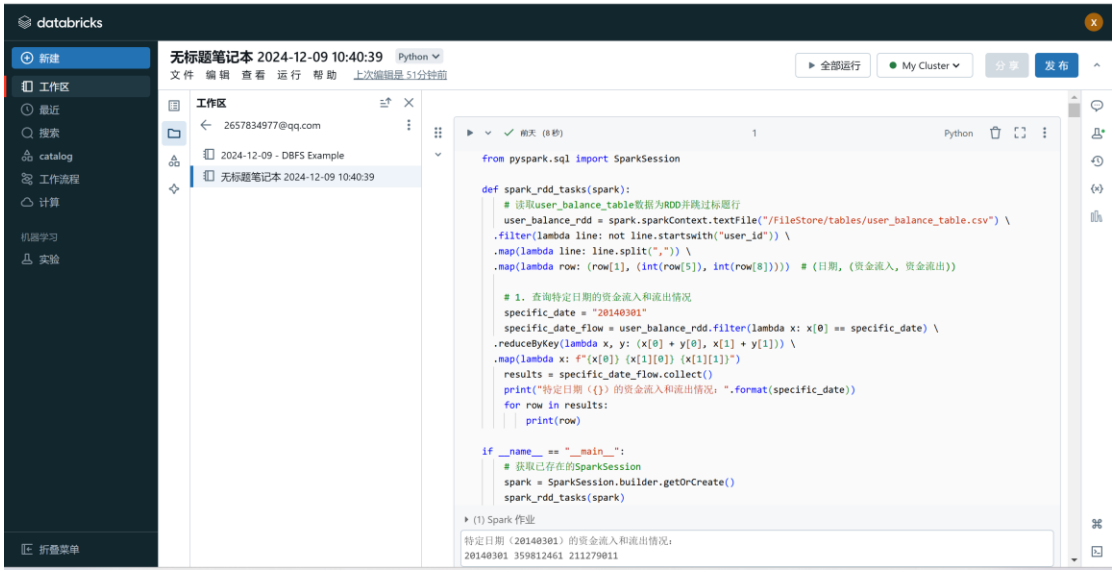
实验 4：Spark 编程

Spark 的环境配置：

使用 databricks 线上环境，注册申请后直接新建笔记本然后挂载运行资源即配置完成



然后新建表格上传本地数据。后表格会被存储在 `FileStore/tables/`里面在里面调用即可

任务 1：Spark RDD 编程

1、查询特定日期的资金流入和流出情况：

使用 user_balance_table ，计算出所有用户在每一天的总资金流入和总资金流出量。

首先读入数据并跳过标题行提取指定的列

```
# 读取user_balance_table数据为RDD并跳过标题行
user_balance_rdd = spark.sparkContext.textFile("/FileStore/tables/user_balance_table.csv") \
.filter(lambda line: not line.startswith("user_id")) \
.map(lambda line: line.split(",")) \
.map(lambda row: (row[1], (int(row[5]), int(row[8])))) # (日期, (资金流入, 资金流出))
```

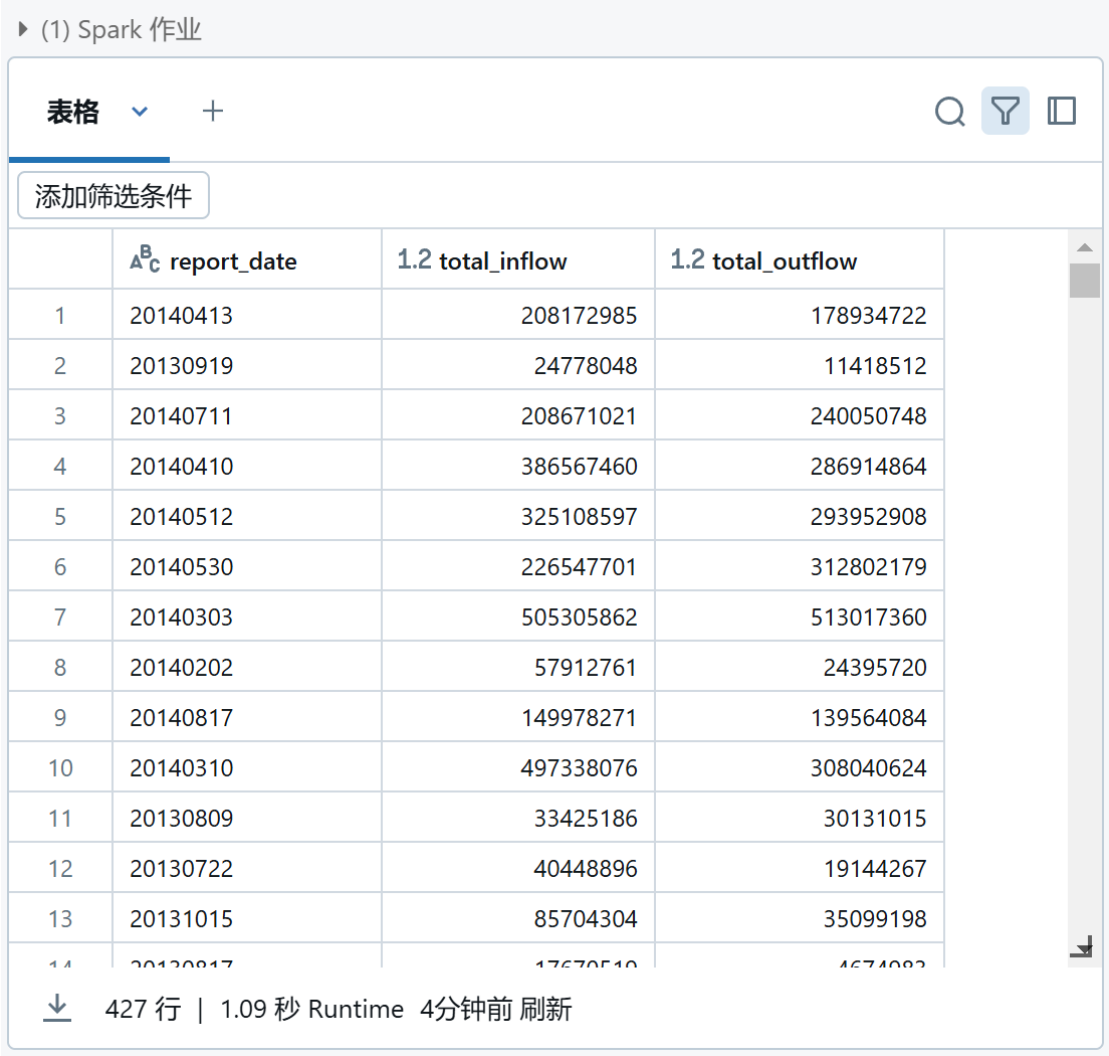
然后对列进行操作，使用reduceByKey聚合操作计算所有date值相同天的输入输出的和，并将结果存储到表中，最后用display展示出来

```
    # 查询所有天的资金流入和流出情况
    all_dates_flow = user_balance_rdd.reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1])) \
     .map(lambda x: (x[0], x[1][0], x[1][1]))

    schema = StructType([
        StructField("date", StringType(), True),
        StructField("inflow", IntegerType(), True),
        StructField("outflow", IntegerType(), True)
    ])
    df = spark.createDataFrame(all_dates_flow, schema)
    df.write.mode("overwrite").saveAsTable("fund_flow")
    df.show()
    display(df)


if __name__ == "__main__":
    # 获取已存在的SparkSession
    spark = SparkSession.builder.getOrCreate()
    spark_rdd_tasks(spark)
```

注意在这个网站中show是打印字符串，会有行数限制且无法下载，只有用display才能展示结果CSV文件并可以下载，也可以用下面这段代码读取表格

```
from pyspark.sql import SparkSession

def read_and_display_daily_flow_table(spark):
    # 表名
    table_name = "daily_flow_table"

    # 读取表格数据
    daily_flow_table_df = spark.read.table(table_name)

    # 使用display展示表格数据
    display(daily_flow_table_df)

if __name__ == "__main__":
    # 获取已存在的SparkSession
    spark = SparkSession.builder.getOrCreate()
    read_and_display_daily_flow_table(spark)
```

最后结果如下：



	report_date	total_inflow	total_outflow
1	20140413	208172985	178934722
2	20130919	24778048	11418512
3	20140711	208671021	240050748
4	20140410	386567460	286914864
5	20140512	325108597	293952908
6	20140530	226547701	312802179
7	20140303	505305862	513017360
8	20140202	57912761	24395720
9	20140817	149978271	139564084
10	20140310	497338076	308040624
11	20130809	33425186	30131015
12	20130722	40448896	19144267
13	20131015	85704304	35099198

活跃用户分析：

使用 user_balance_table ，定义活跃用户为在指定月份内有至少5天记录的用户，统计 2014 年 8 月的活跃用户总数。

和上一个任务类似，只需要先进行筛选然后按照用户分组并计算每个用户的记录天数，最后筛选出活跃用户即可，具体而言

先使用 filter 函数，只保留日期以 201408 开头的数据，也就是 2014 年 8 月的数据。

然后先将数据映射成(user_id，1)的形式，方便后续累加计数。接着通过

reduceByKey 聚合操作，统计每个用户在 8 月出现的天数。

最后用 filter 筛选出记录天数大于等于 5 的用户，定义这些用户为活跃用户 。最后通过 count 函数统计活跃用户的总数。

任务 2：Spark SQL 编程

请使用**Spark SQL**编程的方式，完成下面的任务。

按城市统计 2014 年 3 月 1 日的平均余额：

计算每个城市在2014年3月1日的用户平均余额(tBalance)，按平均余额降序排列。

```python
def calculate_city_avg_balance(spark):
    # 读取user_profile_table和user_balance_table数据为DataFrame，假设文件有标题行
    user_profile_df = spark.read.csv("/FileStore/tables/user_profile_table.csv", header=True)
    user_balance_df = spark.read.csv("/FileStore/tables/user_balance_table.csv", header=True)

    # 创建临时视图
    user_profile_df.createOrReplaceTempView("user_profile")
    user_balance_df.createOrReplaceTempView("user_balance")

    # 使用Spark SQL查询计算每个城市在2014年3月1日的平均余额并按降序排列
    query = """
        SELECT up.City, AVG(ub.tBalance) AS avg_balance
        FROM user_profile up
        JOIN user_balance ub ON up.user_id = ub.user_id
        WHERE ub.report_date = '20140301'
        GROUP BY up.City
        ORDER BY avg_balance DESC
    """
    result = spark.sql(query)

    # 显示结果
    result.show()
```

先读取两张表为dataframe型，然后只需要调用spark.sql语句即可对dataframe进行操作了

```python
def calculate_city_avg_balance(spark):
    # 读取user_profile_table和user_balance_table数据为DataFrame，假设文件有标题行
    user_profile_df = spark.read.csv("/FileStore/tables/user_profile_table.csv", header=True)
    user_balance_df = spark.read.csv("/FileStore/tables/user_balance_table.csv", header=True)

    # 创建临时视图
    user_profile_df.createOrReplaceTempView("user_profile")
    user_balance_df.createOrReplaceTempView("user_balance")

    # 使用Spark SQL查询计算每个城市在2014年3月1日的平均余额并按降序排列
    query = """
        SELECT up.City, AVG(ub.tBalance) AS avg_balance
        FROM user_profile up
        JOIN user_balance ub ON up.user_id = ub.user_id
        WHERE ub.report_date = '20140301'
        GROUP BY up.City
        ORDER BY avg_balance DESC
    """
    result = spark.sql(query)

    # 显示结果
    result.show()
```

```
    calculate_city_avg_balance(spark)
```

▸ (5) Spark 作业

```
+-------+------------------+
|   City|       avg_balance|
+-------+------------------+
|6281949| 2795923.837298216|
|6301949|2650775.0664451825|
|6081949|2643912.7566638007|
|6481949|2087617.2136986302|
|6411949|1929838.5617977527|
|6412149| 1896363.471625767|
|6581949|1526555.5551020408|
+-------+------------------+
```

2、统计每个城市总流量前 3 高的用户：

统计每个城市中每个用户在 2014 年 8 月的总流量（定义为 total_purchase_amt + total_redeem_amt ），并输出每个城市总流量排名前三的用户 ID 及其总流量。

与上文类似分三次进行，首先计算每个用户在 2014 年 8 月整月的总流量，先按用户和城市分组求和，接着使用窗口函数按城市对总流量进行排名，最后选择每个城市排名前三的用户即可

```
    query = """
        SELECT up.City, ub.user_id, SUM(ub.total_purchase_amt + ub.total_redeem_amt) AS total_flow
        FROM user_profile up
        JOIN user_balance ub ON up.user_id = ub.user_id
        WHERE ub.report_date LIKE '201408%'
        GROUP BY up.City, ub.user_id
    """
    total_flow_df = spark.sql(query)

    # 为总流量列创建临时视图
    total_flow_df.createOrReplaceTempView("total_flow")

    # 使用窗口函数按城市对总流量进行排名
    query2 = """
        SELECT City, user_id, total_flow,
               RANK() OVER (PARTITION BY City ORDER BY total_flow DESC) AS rank
        FROM total_flow
    """
    ranked_df = spark.sql(query2)

    # 选择每个城市排名前三的用户
    top_3_users = ranked_df.filter(ranked_df["rank"] <= 3)

    # 显示结果
    top_3_users.show()
```

结果如下：

```
top_3_users_by_city(spark)
```

▸ (5) Spark 作业

```
|6081949|  27235|2.2435196E7|   3|
|6281949|  15118|1.4762625E8|   1|
|6281949|  25814| 5.504067E7|   2|
|6281949|  22871|4.9574861E7|   3|
|6301949|   2429|5.4636517E7|   1|
|6301949|  10932|5.1731105E7|   2|
|6301949|   5330|4.1828956E7|   3|
|6411949|  21030|4.8883465E7|   1|
|6411949|  25025|3.0539038E7|   2|
|6411949|  25525| 2.663695E7|   3|
|6412149|  22585|6.9556206E7|   1|
|6412149|  22585|4.7453506E7|   2|
|6412149|  22585| 3.983884E7|   3|
|6481949|    670|4.9497479E7|   1|
|6481949|  21021|3.1286393E7|   2|
|6481949|  12026|2.5079319E7|   3|
|6581949|  21761|1.4885517E7|   1|
|6581949|   7107|  8743312.0|   2|
+-------+-------+-----------+----+
only showing top 20 rows
```

任务 3：Spark ML 编程

使用 **Spark MLlib** 提供的机器学习模型，预测 **2014** 年 **9** 月每天的申购与赎回总额。

首先先计算每日所有用户的总流入流出并以 201301 为基础添加行号

```
|   20130703|  2.727077E7|    5953867.0|      3|
|   20130704| 1.8321185E7|    6410729.0|      4|
|   20130705| 1.1648749E7|    2763587.0|      5|
|   20130706| 3.6751272E7|    1616635.0|      6|
|   20130707|   8962232.0|    3982735.0|      7|
|   20130708| 5.7258266E7|    8347729.0|      8|
|   20130709| 2.6798941E7|    3473059.0|      9|
|   20130710| 3.0696506E7|    2597169.0|     10|
|   20130711| 4.4075197E7|    3508800.0|     11|
|   20130712| 3.4183904E7|    8492573.0|     12|
|   20130713| 1.5164717E7|    3482829.0|     13|
|   20130714| 2.2615303E7|    2784107.0|     14|
|   20130715| 4.8128555E7|  1.3107943E7|     15|
|   20130716| 5.0622847E7|  1.1864981E7|     16|
|   20130717| 2.9015682E7|  1.0911513E7|     17|
|   20130718| 2.4234505E7|  1.1765356E7|     18|
|   20130719| 3.3680124E7|    9244769.0|     19|
|   20130720| 2.0439079E7|    4601143.0|     20|
+-----------+------------+-------------+-------+
only showing top 20 rows
```

结果如图存入表中
接着以行号为自变量，所有数据为训练集，测试集为后 30 天的数据用线性模型训练提交至天池结果如图

其中我尝试直接使用 datafram 的 date 型作为自变量来预测输入输出，但是老是报错 data_df:pyspark.sql.dataframe.DataFrame 此数据集的架构不可用 以及 result_df:pyspark.sql.dataframe.DataFrame report_date:string redemption_prediction:double 以及 test_df:pyspark.sql.dataframe.DataFrame 此数据集的架构不可用。可能是数据架构方面的一些问题，似乎是 date 这个数据类型的问题，要是将

date 拆成年月日三个 int 型数据就可以跑了，而且这个 date 似乎不能直接比大小还得转换成 Python 再来比较，不是特别好用。