STA323 Assignment 3 report

SID: 12110821
Name: ZHANG Chi

Solution for Q1

In this question, we need to partition the data into four parts according to the value of column `origin`. In particular, the *Spark RDD API* should be used.

To begin with, I first read the data by `sparkContext.textFile()`, and show it to see the data structure (see picture below). We can find that the column names of each column in the dataset and from left to right are `date, delay, distance, origin,` and `destination`.

```
rdd1 = spark.sparkContext.textFile("data/departuredelays.csv")
rdd1.take(5)
```

```
0.2s
```

```
['date,delay,distance,origin,destination',
 '01011245,6,602,ABE,ATL',
 '01020600,-8,369,ABE,DTW',
 '01021245,-2,602,ABE,ATL',
 '01020605,-4,602,ABE,ATL']
```

According to the requirement, we need to find the rows whose `origin` is `ATL`, which will be partitioned into one partition, and the rest will be partitioned into three others. Before achieving it, I use the `filter()` to drop the row having column names. Then my steps are as follows:

- Use `keyBy()` and `lambda` to mark the value of the `origin` column as the key of each record: if it is `ATL`, the key is `1`. Otherwise, the key is `0`
- Use `partitionBy()` to partition the data into four parts regarding to the key. The specific rule is that if the key is `1`, the record will be put into the first partition (return `0`). Otherwise, it will be put into other three partitions (return `random.randint(1, 3)`)

The result can be captured by the following code:

```
partitioned_rdd = rdd2.partitionBy(4, partition_func)

# remove the key
partitioned_rdd = partitioned_rdd.map(lambda x: x[1])

# collect the result regarding each partition
partition_list = partitioned_rdd.glom().collect()
```

The number of elements of 4 partitions and some former elements are represented in sequence below.

```
The number of elements of 4 partitons in sequence:  [91484, 434701, 432274, 433119]
Some elements in partition 1:
 ['01010640,-4,517,ATL,MIA', '01011925,-1,636,ATL,DFW', '01011245,22,636,ATL,DFW', '01011405,-3,636,ATL,DFW', '01011540,-4,636,ATL,DFW']
Some elements in partition 2:
 ['01020600,-8,369,ABE,DTW', '01021245,-2,602,ABE,ATL', '01020605,-4,602,ABE,ATL', '01041243,10,602,ABE,ATL', '01040605,28,602,ABE,ATL']
Some elements in partition 3:
 ['01050605,9,602,ABE,ATL', '01061725,69,602,ABE,ATL', '01061230,0,369,ABE,DTW', '01071725,0,602,ABE,ATL', '01071219,0,569,ABE,ORD']
Some elements in partition 4:
 ['01011245,6,602,ABE,ATL', '01031245,-4,602,ABE,ATL', '01030605,0,602,ABE,ATL', '01051245,88,602,ABE,ATL', '01061215,-6,602,ABE,ATL']
```

Solution for Q2

(1)

To set the number of partitions for streaming data in Spark, I use `spark.sql.shuffle.partitions` configuration property. This property determines the number of partitions used for shuffling data during operations like aggregations, joins, and sorting.

Since the `schema` needs to be specified when reading the streaming data, I first fetch the data schema by statistical analysis. The schema is shown in the following image after reading one randomly selected JSON file by `spark.read.json()`.

```
root
 |-- Arrival_Time: long (nullable = true)
 |-- Creation_Time: long (nullable = true)
 |-- Device: string (nullable = true)
 |-- Index: long (nullable = true)
 |-- Model: string (nullable = true)
 |-- User: string (nullable = true)
 |-- gt: string (nullable = true)
 |-- x: double (nullable = true)
 |-- y: double (nullable = true)
 |-- z: double (nullable = true)
```

Noticing that both columns regarding time are in the format `long`, which is not a typical time type. Thus, I use `to_timestamp()` to convert the `timestamp` column `Arrival_Time` and column `Creation_Time`. However, the two columns are in the format of `long,` and the unit is not `s,` which will not present a satisfying result (the picture is shown below).

> This kind of long format will cause a `long type overflow` when defining a streaming query.

```
+--------------------+---------------------+--------+-----+------+----+-----+------------+------------+------------+
|Arrival_Time        |Creation_Time        |Device  |Index|Model |User|gt   |x           |y           |z           |
+--------------------+---------------------+--------+-----+------+----+-----+------------+------------+------------+
|+47116-07-11 19:19:35|+248979-10-08 18:34:34|nexus4_1|35   |nexus4|g   |stand|0.0014038086 |5.0354E-4   |-0.0124053955|
|+47116-07-11 19:22:58|+248986-04-26 09:03:15|nexus4_1|76   |nexus4|g   |stand|-0.0039367676|0.026138306 |-0.01133728  |
|+47116-07-11 19:26:17|+248992-07-14 21:19:00|nexus4_1|115  |nexus4|g   |stand|0.003540039  |-0.034744263|-0.019882202 |
|+47116-07-11 19:29:39|-276993-05-14 17:13:32|nexus4_2|163  |nexus4|g   |stand|0.002822876  |0.005584717 |0.017318726  |
|+47116-07-11 19:33:02|-276987-10-02 08:01:38|nexus4_2|203  |nexus4|g   |stand|0.0017547607 |-0.018981934|-0.022201538 |
+--------------------+---------------------+--------+-----+------+----+-----+------------+------------+------------+
```

The solution can be found in StackOverflow, which divides the column by `1000000000` and `1000,` respectively, as their unit is `ns` and `ms`. The code is shown below.

```
df.withColumn("Creation_Time",to_timestamp(col("Creation_Time")/1000000000))\
    .withColumn("Arrival_Time ",to_timestamp(col("Arrival_Time ")/1000))\
    .show(5,truncate=False)
```

```
+-------------------+--------------------------+--------+-----+------+----+-----+------------+------------+------------+
|Arrival_Time       |Creation_Time             |Device  |Index|Model |User|gt   |x           |y           |z           |
+-------------------+--------------------------+--------+-----+------+----+-----+------------+------------+------------+
|2015-02-23 10:18:55|2015-02-23 10:18:53.176179|nexus4_1|35   |nexus4|g   |stand|0.0014038086|5.0354E-4   |-0.0124053955|
|2015-02-23 10:18:55|2015-02-23 10:18:53.382813|nexus4_1|76   |nexus4|g   |stand|-0.0039367676|0.026138306 |-0.01133728 |
|2015-02-23 10:18:55|2015-02-23 10:18:53.579072|nexus4_1|115  |nexus4|g   |stand|0.003540039 |-0.034744263|-0.019882202|
|2015-02-23 10:18:55|2015-02-23 10:49:41.834321|nexus4_2|163  |nexus4|g   |stand|0.002822876 |0.005584717 |0.017318726 |
|2015-02-23 10:18:55|2015-02-23 10:49:42.035859|nexus4_2|203  |nexus4|g   |stand|0.0017547607|-0.018981934|-0.022201538|
+-------------------+--------------------------+--------+-----+------+----+-----+------------+------------+------------+
```

Then, we can define the streaming query by following the steps.

- Read the streaming data by `spark.readStream.schema(schema).json()`. In addition, I also set the `maxFilesPerTrigger` to 10.
- Defining the data operation: After converting the time datatype, a watermark can be set by `withWatermark()` to 1 minute. Then, use `groupBy()` to count records by the `user` and designate windows of 6 minutes moving forward in every 3 (6 - 3 = 3) minutes.
- Specify the output. According to the requirement, here I use `update` mode as well as `memory` sink, and the checkpoint location can be set by `option("checkpointLocation", checkpointDir)`. Moreover, I also set the processing time to 2 seconds.

> It is puzzling that an error will be caused when rerunning the code, which asks me to delete the `offsets` directory in the `checkpoint` directory. After deleting the offsets directory by following the instructions, the error will not appear when rerunning the code. The exact reason is not apparent to me.
>
> ```
> AnalysisException: This query does not support recovering from checkpoint location. Delete checkpoint/activity-data-append-memory/offsets to start over.
> ```
>
> ```python
> import os
> import shutil
> if os.path.exists('checkpoint/activity-data/offsets'):
>  shutil.rmtree('checkpoint/activity-data/offsets')      # 强制删除文件夹
> ```

The first three query results are shown below. Each query will show ten rows, and all rows will be sorted by `window` in descending order to emphasize data streaming. Besides, the interval between two queries is 2 seconds by `time.sleep(2)`.

```
+----+------------------------------------------+-----+
|user|window                                    |count|
+----+------------------------------------------+-----+
|e   |{2015-02-24 15:21:00, 2015-02-24 15:27:00}|2271 |
|e   |{2015-02-24 15:21:00, 2015-02-24 15:27:00}|1139 |
|e   |{2015-02-24 15:21:00, 2015-02-24 15:27:00}|3405 |
|e   |{2015-02-24 15:18:00, 2015-02-24 15:24:00}|5208 |
|e   |{2015-02-24 15:18:00, 2015-02-24 15:24:00}|10397|
|e   |{2015-02-24 15:18:00, 2015-02-24 15:24:00}|15631|
|e   |{2015-02-24 15:15:00, 2015-02-24 15:21:00}|15423|
|e   |{2015-02-24 15:15:00, 2015-02-24 15:21:00}|7694 |
|e   |{2015-02-24 15:15:00, 2015-02-24 15:21:00}|23198|
|e   |{2015-02-24 15:12:00, 2015-02-24 15:18:00}|14402|
+----+------------------------------------------+-----+
```

```
+----+------------------------------------------+-----+
|user|window                                    |count|
+----+------------------------------------------+-----+
|e   |{2015-02-24 15:21:00, 2015-02-24 15:27:00}|3405 |
|e   |{2015-02-24 15:21:00, 2015-02-24 15:27:00}|2271 |
|e   |{2015-02-24 15:21:00, 2015-02-24 15:27:00}|1139 |
|e   |{2015-02-24 15:21:00, 2015-02-24 15:27:00}|4547 |
|e   |{2015-02-24 15:18:00, 2015-02-24 15:24:00}|15631|
|e   |{2015-02-24 15:18:00, 2015-02-24 15:24:00}|5208 |
|e   |{2015-02-24 15:18:00, 2015-02-24 15:24:00}|10397|
|e   |{2015-02-24 15:18:00, 2015-02-24 15:24:00}|20815|
|e   |{2015-02-24 15:15:00, 2015-02-24 15:21:00}|23198|
|e   |{2015-02-24 15:15:00, 2015-02-24 15:21:00}|15423|
+----+------------------------------------------+-----+
```

```
+----+------------------------------------------+-----+
|user|window                                    |count|
+----+------------------------------------------+-----+
|e   |{2015-02-24 15:21:00, 2015-02-24 15:27:00}|1139 |
|e   |{2015-02-24 15:21:00, 2015-02-24 15:27:00}|3405 |
|e   |{2015-02-24 15:21:00, 2015-02-24 15:27:00}|2271 |
|e   |{2015-02-24 15:21:00, 2015-02-24 15:27:00}|4547 |
|e   |{2015-02-24 15:21:00, 2015-02-24 15:27:00}|5684 |
|e   |{2015-02-24 15:18:00, 2015-02-24 15:24:00}|20815|
|e   |{2015-02-24 15:18:00, 2015-02-24 15:24:00}|10397|
|e   |{2015-02-24 15:18:00, 2015-02-24 15:24:00}|5208 |
|e   |{2015-02-24 15:18:00, 2015-02-24 15:24:00}|15631|
|e   |{2015-02-24 15:18:00, 2015-02-24 15:24:00}|25998|
+----+------------------------------------------+-----+
```

(2)

In this part, I need to define two queries simultaneously under the task in Q2.1, meaning that I only need to modify the code's last part (output and sink).

The two queries are both in `append` mode but with different output sinks. The first query is to write the result to the `memory`, while the second is to write the result to the `parquet` sink. The code is shown below. Notably, the checkpoint location, as well as instructions to remove the offsets directory, should be set for each query.

```
parquetOutputPath = "output/activity-data"
activityQuery2 = activityCounts.writeStream \
    .queryName("activity_query2")\
    .format("parquet") \
    .outputMode("append") \
    .option("checkpointLocation", "checkpoint/activity-data-append-parquet") \
    .option("path", parquetOutputPath) \
    .start()

```

```python
activityQuery3 = activityCounts.writeStream \
    .queryName("activity_query3")\
    .format("memory") \
    .outputMode("append") \
    .option("checkpointLocation", "checkpoint/activity-data-append-memory") \
    .start()
```

Solution for Q3

The data file the question gives consists of users' online shopping records. If the data is stored correctly, the `action` column should have four kinds of integer values from 1 to 4, and the `gender` column should have two kinds of integer values, 0 and 1.

(1)

Before creating a Kafka pipeline, I would like to check the data structure using `spark.read.csv()`. Unfortunately, some values are not expected in the `gender` column (see the picture below). The reason is not clear so that we can filter the data by the rule `(col("gender") != 0) & (col("gender") != 1)` tentatively.

user_id	item_id	cat_id	merchant_id	brand_id	month	day	action	age_range	gender	province
328862	323294	833	2882	2661	08	29	0	0	1	内蒙古
328862	844400	1271	2882	2661	08	29	0	1	1	山西
328862	575153	1271	2882	2661	08	29	0	2	1	山西
328862	996875	1271	2882	2661	08	29	0	1	1	内蒙古
328862	1086186	1271	1253	1049	08	29	0	0	2	浙江

Then, we can create a Kafka pipeline step by step.

First, in the shell script `ass3_q3_kafka.sh`, I start the ZooKeeper service by running the `ZooKeeper-server-start.sh` script. ZooKeeper is a distributed coordination system for managing configuration information and the status of Kafka clusters. Next, the Kafka service is run by the `Kafka-server-start.sh` script. Finally, the Python script `ass3_q3_runproducer.py` is called. This producer reads the contents of the given CSV file and sends it as a message to a specific topic `q3` in the Kafka cluster.

More specifically, I use the `pandas` library to read the whole file and then iterate over each row to send it out by a `KafkaProducer` object, which was defined before. As the question asks that messages should contain the `action` and `gender` columns, I define a dictionary `{"action": row["action"], "gender": row["gender"]}` additionally. Then, the `KafkaProducer` will serialize the dictionary by `json.dumps()` before sending it out, and the message will be sent every 0.5s because of `time.sleep(0.5)`.

To check the data in the topic `q3`, I use the `KafkaConsumer` object to subscribe to the topic `q3` and print the message in the Python script `ass3_q3_runconsumer.py`. The result is shown below.

```
The value of offset 8644 is >>> {'action': 0, 'gender': 0}.
The value of offset 8645 is >>> {'action': 0, 'gender': 2}.
The value of offset 8646 is >>> {'action': 0, 'gender': 2}.
The value of offset 8647 is >>> {'action': 2, 'gender': 0}.
The value of offset 8648 is >>> {'action': 2, 'gender': 1}.
The value of offset 8649 is >>> {'action': 2, 'gender': 2}.
The value of offset 8650 is >>> {'action': 2, 'gender': 2}.
The value of offset 8651 is >>> {'action': 2, 'gender': 1}.
The value of offset 8652 is >>> {'action': 0, 'gender': 1}.
The value of offset 8653 is >>> {'action': 0, 'gender': 0}.
```

Then, I can define the streaming query by reading the data from the topic `q3`. All messages are stored in the column `value`. The schema is shown below, from which we can find that the column `value` is binary and the column `timestamp` is integer. Hence, the datatype should be converted first.

```
root
 |-- key: binary (nullable = true)
 |-- value: binary (nullable = true)
 |-- topic: string (nullable = true)
 |-- partition: integer (nullable = true)
 |-- offset: long (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- timestampType: integer (nullable = true)
```

For the field `value`, I convert it to a string by `cast()` and then use `from_json()` to parse the string into a struct type. The schema of the struct type is defined in the `schema,` which is shown below. Then, I can extract the `action` and `gender` by `data.action` and `data.gender`, respectively. For the field `timestamp`, I convert it to a timestamp by `CAST (timestamp AS TIMESTAMP)` directly.

```
df1 = df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)", "CAST
(timestamp AS TIMESTAMP)")
schema = "gender INT, action INT"
df2 = df1.selectExpr("key", "from_json(value, '{}') AS data".format(schema),
"timestamp")
```

The `where()` is used to filter the data. Except for the value of `action`, which should be two required by the question, the value of `gender` should be 0 or 1. In order to count the number of male and female records, I use `groupBy()` to count the number of records by `gender`. Both the watermark and window are set to 10 seconds and 5 seconds, respectively. Furthermore, the output is set to the `memory` sink in `complete` mode.

```
df_filter = df2.where((col("data.action") == 2) & (col("data.gender") != 0) & (col("data.gender") != 1))
result = df_filter.withWatermark("timestamp", "10 seconds").groupBy("data.gender",window("timestamp", "5 seconds")).agg(count("*").alias("Number of transactions"))
q3_query = result.writeStream.queryName("transaction_count").format("memory").outputMode("update").trigger(processingTime="5 seconds").start()
```

The first 20 rows are displayed below.

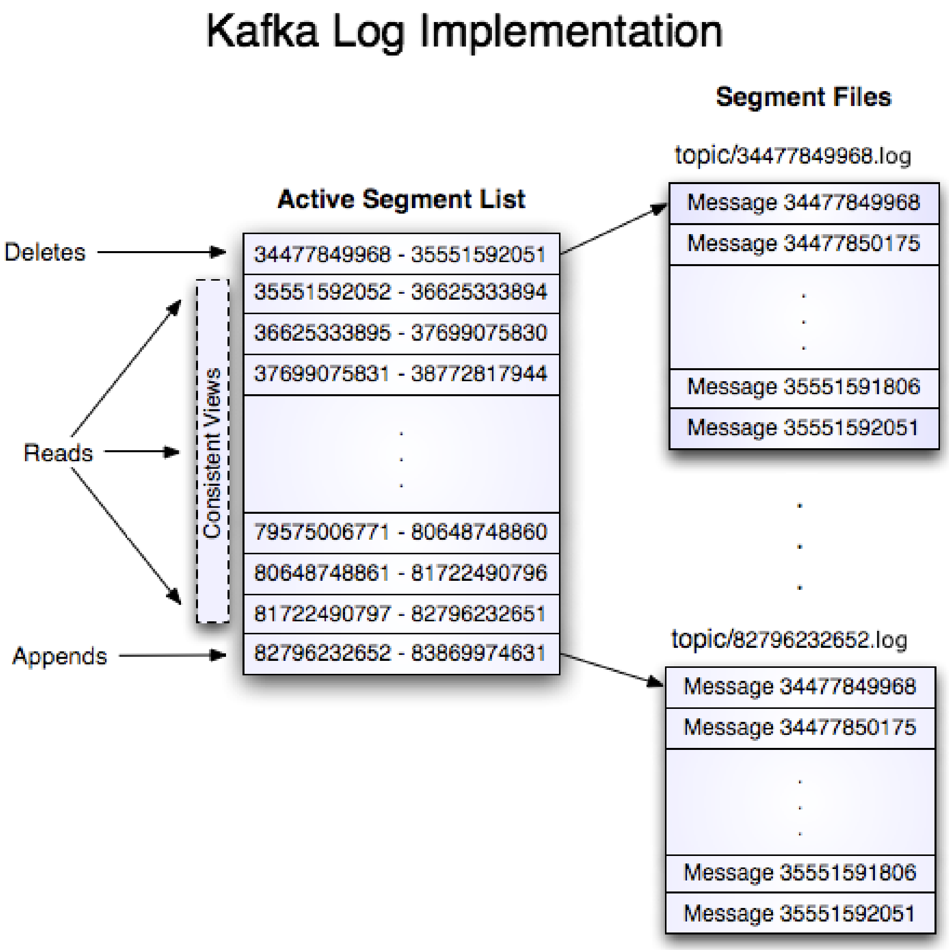
```
spark.sql("SELECT window, gender, `Number of transactions` FROM transaction_count").show(20,False)
✓ 0.0s

+------------------------------------------+------+----------------------+
|window                                    |gender|Number of transactions|
+------------------------------------------+------+----------------------+
|{2024-05-02 11:44:15, 2024-05-02 11:44:20}|2     |1                     |
|{2024-05-02 03:00:30, 2024-05-02 03:00:35}|2     |1                     |
|{2024-05-02 10:50:25, 2024-05-02 10:50:30}|2     |1                     |
|{2024-05-02 03:21:00, 2024-05-02 03:21:05}|2     |2                     |
|{2024-05-02 10:27:15, 2024-05-02 10:27:20}|2     |1                     |
|{2024-05-02 02:42:25, 2024-05-02 02:42:30}|2     |1                     |
|{2024-05-02 10:36:50, 2024-05-02 10:36:55}|2     |1                     |
|{2024-05-02 10:40:30, 2024-05-02 10:40:35}|2     |2                     |
|{2024-05-02 02:22:10, 2024-05-02 02:22:15}|2     |1                     |
|{2024-05-02 02:24:30, 2024-05-02 02:24:35}|2     |1                     |
|{2024-05-02 10:26:30, 2024-05-02 10:26:35}|2     |1                     |
|{2024-05-02 10:41:10, 2024-05-02 10:41:15}|2     |1                     |
|{2024-05-02 03:05:10, 2024-05-02 03:05:15}|2     |2                     |
|{2024-05-02 11:39:00, 2024-05-02 11:39:05}|2     |1                     |
|{2024-05-02 10:34:55, 2024-05-02 10:35:00}|2     |2                     |
|{2024-05-02 03:04:40, 2024-05-02 03:04:45}|2     |1                     |
|{2024-05-02 10:39:35, 2024-05-02 10:39:40}|2     |1                     |
|{2024-05-02 11:14:30, 2024-05-02 11:14:35}|2     |1                     |
|{2024-05-02 10:23:20, 2024-05-02 10:23:25}|2     |1                     |
|{2024-05-02 02:53:50, 2024-05-02 02:53:55}|2     |1                     |
+------------------------------------------+------+----------------------+
```

(2)

Kafka uses a log-based storage mechanism to handle messages. The log is organized into topics and partitions. Messages are stored in log files, with each file containing a sequence of log entries. Each log entry consists of a message length and the message itself.

Here is a brief overview of how Kafka handles logging. I scratched it from the Kafka documentation.



Writing to the log involves serial appends to the last file, which is rolled over when it reaches a certain size. Kafka provides durability guarantees by flushing messages to disk after a configurable number of messages or time intervals.

Reading from the log is done by providing the offset of a message, and Kafka returns the messages starting from that offset. If a message is larger than the buffer size, the read can be retried with a larger buffer.

Log segments are **deleted** based on time and size policies. The log manager deletes the oldest segments until the partition's size is within the configured limit.

Kafka ensures data integrity by verifying the validity of log entries during startup. Corruption detection handles truncation and corruption scenarios, and the log is truncated to the last valid offset if corruption is detected.

In a nutshell, the logging mechanism in Kafka provides fault tolerance, scalability, and efficient data storage and retrieval. It enables high-throughput message processing and reliable data replication across multiple brokers.