Lab 2 - Kafka Spark

Harald Ng
Chuan Su

October 5, 2019

1 Implementation

First, the Kafka Configuration had to be set up. The most essential configurations were the following:

- `zookeeper.connect`: Set to `localhost:2181` which was the port that the zookeeper had been started on our local machine.
- `boostrap.servers`: This configuration is used to specify where Kafka clients should connect to bootstrap into the Kafka cluster. It was set to `localhost:9092`
- `key.deserializer` and `value.deserializer`: Consumer configurations to set how to deserialize incoming data. As the provided generator program would generate data as `string` for the stream, we set both to `classOf[StringDeserializer]`.

Then we had to set up Spark to read the incoming data from Kafka as a stream. This was done using the `KafkaUtils.createDirectStream` function where we would specify the incoming data type and also which topic we want to subscribe to. As we earlier had produced message from Kafka using the topic name "avg", it was set to that name.

To calculate the average, the state was set to be a tuple with two integers, the sum and the number of elements. The key was of course a string. The mapping function was defined so that it would return a type of float. When a key-value pair was processed we would check if there was any value for that key before. It there wasn't, the state would be `None`, else we would update the state by adding the received value to the sum and also increment the number of elements with 1. The function would then return the average by dividing the two elements of the state tuple.

To make the incoming data from the stream work with the mapping function, each message from Kafka had to be mapped to a tuple of `(key, value)`, where `key` would be a string and `value` an integer. By reading the provided code in `generator`, we could see that each massage would have the key to be `null` and the value would be a string with the format "<key>,<value>". Hence, we would just split the string on the comma and use the key as the first element of the splitted string and the second element (casted to integer) as value.

Storing the result to Cassandra was straight forward. The `saveCassandra` function was called and the arguments would be the name of the keyspace, the name of the table and the columns.

2 Results

The results can be seen in Figure 1.

```
cqlsh:avg_space> select * from avg;

 word | count
------+----------
    z |  9.92308
    a |     13.32
    c | 11.85714
    m | 13.07921
    f | 13.21875
    o | 12.50538
    n | 12.87013
    q | 12.32911
    g | 12.38235
    p | 11.91089
    e | 12.56436
    r | 11.71277
    d |   13.125
    h |      12.9
    w | 13.49495
    l | 12.80702
    j | 13.11236
    v | 12.49057
    y | 13.26596
    u | 13.08163
    i | 12.74227
    k | 12.87805
    t | 13.76923
    x | 13.38461
    b | 12.90217
    s | 11.88073

(26 rows)
```

Figure 1: Keys are in the word column and average values are in the count column.

3 How to run

The submitted code was tested on Spark 2.4.4 and Kafka 2.3.0. Furthermore, The Spark Streaming integration for Kafka was version 0.10. To run the code, one needs to start Zookeeper, Kafka and Cassandra. Then the generator and the submitted code can be run

with sbt run in their respective directories. To check that the program is writing the results to Cassandra, we can start a cql shell and query it. The following shows to commands to run the program:

```
// start zookeeper
$KAFKA_HOME/bin/zookeeper-server-start.sh $KAFKA_HOME/config/zookeeper.properties

// start kafka
$KAFKA_HOME/bin/kafka-server-start.sh $KAFKA_HOME/config/server.properties

// start cassandra
$CASSANDRA_HOME/bin/cassandra -f

// start the program
cd /directory/of/our/program
sbt run

// start generating kafka messages
cd /directory/of/generator/program
sbt run

// check results in cql
$CASSANDRA_HOME/bin/cqlsh
...
cqlsh> use avg_space;
cqlsh:avg_space> select * from avg;
```