



AIM-3:- Scalable Data Science WS 2020/21

Homework 1

Haftamu Hailu Tefera

ID:452835

Part I: Code Explanation

| IDE | Language | Build tool |
|----------|----------|------------|
| Intellij | Java | Maven |

Table 1: Tools and IDEs

Task A: Mapreduce

Both tasks can be treated as a mapreduce join operation between a customer and orders tables.

First read files, apply map followed by reduce operations. And ,each mapper and reducer are described below

(i) The customer name, address and the average price of orders per customer who have acctbal more than 1000 and for orders placed after 1995-01-01

Mapper

1. Convert orderdate from string into date
2. First we check whether the tuples come from order or customer file.
3. If it is from orders file, we filter order dates placed after 1995-01-01 and project over customer key and price attributes.
 - a. Write customer key and price into context and add '**orders**' text
4. If the data is from customer file, check if the account balance of a customer is more than 1000 and project over the customer key
 - a. Write customer key into context

Reducer

1. The reducer accepts iterable texts
2. If the text contains orders, we calculate total sum of prices and count the number of orders made by a customer
3. If it is from customer, we project customer name and address
4. If the number of orders is >0 && customer name is not empty

- a. Calculate average price of orders per customer by dividing sum of all prices made by customer to the total number of orders.
- b. write the data into the context

(ii) The name of all customers who have not placed any order yet

Mapper

1. If the data is from the customer file project over the customer name and customer key.
 - a. Write customer name and customer key and add 'customer' text.
2. If the data is from order ,project over the customer key.
 - a. Write the customer key into context and add '**orders**' as text.

Reducer

In the reducer, we only need to check if a customer has placed an order or not.

1. If an order has been placed, do nothing
2. If there is no order placed by the customer, we write the customer name into a context.

Note: '**customer**' and '**orders**' texts are important for the reducer phase

```
INFO:  FileSystemCounters
Jan 13, 2021 11:52:08 PM org.apache.hadoop.mapred.Counters log
INFO:  FILE_BYTES_WRITTEN=1178514
Jan 13, 2021 11:52:08 PM org.apache.hadoop.mapred.Counters log
INFO:  FILE_BYTES_READ=5806531
Jan 13, 2021 11:52:08 PM org.apache.hadoop.mapred.Counters log
INFO:  File Output Format Counters
Jan 13, 2021 11:52:08 PM org.apache.hadoop.mapred.Counters log
INFO:  Bytes Written=50866

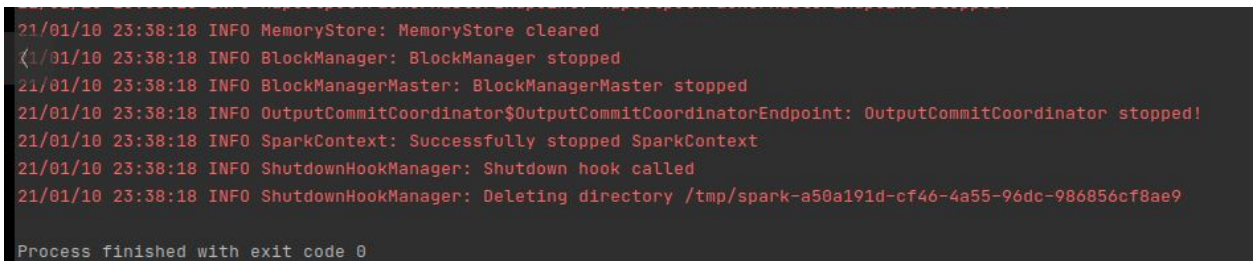
Process finished with exit code 0
```

Figure 1: Mapreduce successful compilation

Task B: Spark

(i) The customer name, address and the average price of orders per customer who have acctbal more than 1000 and for orders placed after 1995-01-01

1. Read customers and order files and store them as RDDs.
2. Apply map operations to get a subset of required attributes from orders RDD followed by a filter of orders made after '1995-01-01'.
3. Perform map operation on the customers RDD followed by a filter of account balance >1000.
4. Join the filtered RDDs using a customer key and return JavaPairRDD of Tuple4<key,
5. Calculate the average price of orders per customer by dividing the sum of all prices made by customer to the total number of orders made and write the result into a csv file.



```
21/01/10 23:38:18 INFO MemoryStore: MemoryStore cleared
21/01/10 23:38:18 INFO BlockManager: BlockManager stopped
21/01/10 23:38:18 INFO BlockManagerMaster: BlockManagerMaster stopped
21/01/10 23:38:18 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator stopped!
21/01/10 23:38:18 INFO SparkContext: Successfully stopped SparkContext
21/01/10 23:38:18 INFO ShutdownHookManager: Shutdown hook called
21/01/10 23:38:18 INFO ShutdownHookManager: Deleting directory /tmp/spark-a50a191d-cf46-4a55-96dc-986856cf8ae9
Process finished with exit code 0
```

Figure 2: Spark task 1 successful compilation

(ii) Single source shortest path

1. Read the text file and store as RDD and set the source node to 17274.
2. Apply flatmap and maptoPair operations to RDD in order to create nodes and edges respectively.
3. If it is a source node set its cost to 1, otherwise set the maximum integer value.
4. Activate the source node and loop until no more nodes are activated.
5. For each activated node, identify messages to send to their adjacent nodes by joining with the edges
6. During each iteration least cost is assigned to neighbours of activated nodes and write output into a csv file

```
21/01/10 23:45:05 INFO SparkUI: Stopped Spark web UI at http://192.168.2.101:4040
21/01/10 23:45:05 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
21/01/10 23:45:06 INFO MemoryStore: MemoryStore cleared
21/01/10 23:45:06 INFO BlockManager: BlockManager stopped
21/01/10 23:45:06 INFO BlockManagerMaster: BlockManagerMaster stopped
21/01/10 23:45:06 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator stopped!
21/01/10 23:45:06 INFO SparkContext: Successfully stopped SparkContext
21/01/10 23:45:06 INFO ShutdownHookManager: Shutdown hook called
21/01/10 23:45:06 INFO ShutdownHookManager: Deleting directory /tmp/spark-8f40c004-6726-4444-9796-ee377d1010f9

Process finished with exit code 0
```

Figure 3: Spark task 2 successful compilation

Task C: Flink Dataset

Unable to finish on time

Task D: Flink Streaming

(i). Highest average distance covered by the player in 5 minute window of 1 minute sliding window

1. After the file has read, filter operation is applied to extract Id of the player using the attached left leg sensor(47).
2. We compute the average distance run by A1 on each 5 minute window interval with 1 minute sliding one using an euclidean distance of players current and previous positions.

3. The global maximum average distance across all windows is computed by setting a highest timestamp when no more player A1 events can arrive then emit the results.
4. Store the output into a csv file

(ii): Total number of averted goal events

A filter operation is applied to get corresponding ids of teams from the sensor ids and group them by sensorid.

1. Read the file
1. A filter operation is applied to select events corresponding to balls used in the match(four different balls are used in the match)
2. Check if the ball is the ball inside the penalty box
We define a function to check if the ball is inside the penalty area of either team.
3. For every 1 minute window, if the current ball position is outside of the penalty box and previous was inside, the goal is not scored , and therefore, the corresponding teams counter will be incremented.
4. To get the total number of averted goals for the entire match, we need to set a large enough window and sum all previous counter values and emit results
5. Finally, write output into a csv file.

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Compute the total number of events over the entire game for which the team almost scored a goal
Computing maximum average distance run by player A1 in a 5 minute window, where the duration between consecutive windows is 1 minute

Process finished with exit code 0
```

Figure 4: Flink streaming successful compilation

Part II: Implementation comparison

1. Difference between Implementation A(i) and B(i)

Spark API is rich and the complexity is hidden from the user, hence, the code is readable and understandable enough compared to the spaghetti code of mapreduce implementation.

For example taking the join operation, in spark the detail is handled by the API, whereas in the case of mapreduce the task is left to the user.

Mapreduce uses persistent storage in which a job execution can introduce an additional latency for reading and writing data whereas spark uses Resilient Distributed Dataset abstraction where the data could be fit in memory. Therefore, the writing and reading overhead could be minimized with spark. Hence, overall, spark implementation is both expressive and fast.

2. Difference between implementations of B(i) and C(i)

*I did not do task C due time constraints

So, the little comparison I did is not code wise, rather it is based on what I have read.

Spark is based on non-native iteration which is implemented as regular loops(for or while) outside of the system. On the other hand, Flink API provides two dedicated iterations: operation iterate and delta iterate which are more optimized for graph processing.[1]

As the source shortest path algorithm is an iterative in nature ,therefore, in terms of efficiency, I can say that , flink would be the right choice to compute single source shortest paths in a graph.

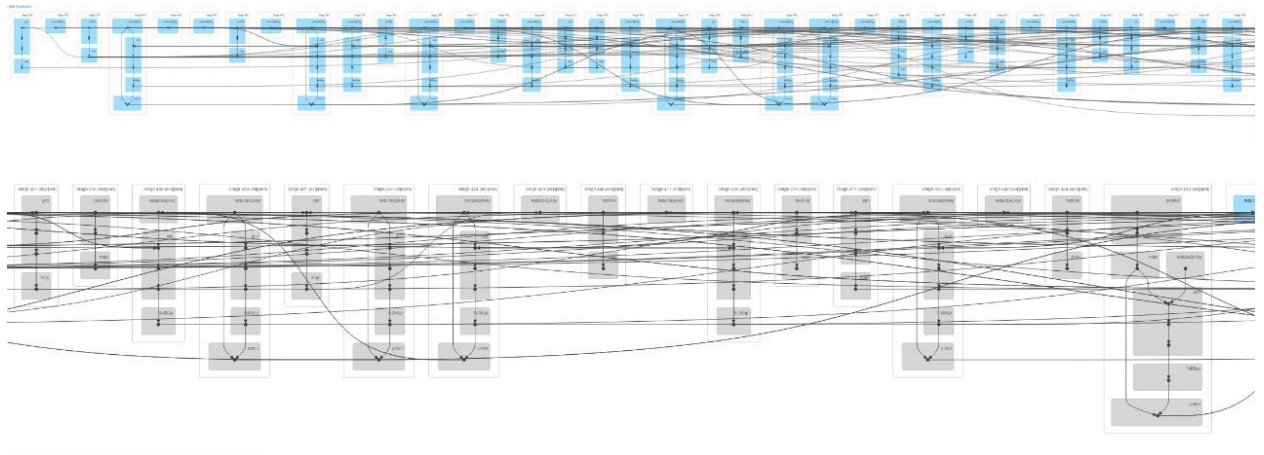


Figure 5: Spark generated execution plan

3. Shortest Path Implementation using GraphLab

Flink and spark use the bulk synchronous computation model when processing the shortest path of the different vertices. The performance of each superstep depends on the slowest node which incurs processing latency. Evaluation of condition on all vertices for every phase could be another bottleneck.

Whereas Graphlab uses asynchronous iterative computation to implement the shortest path. In each phase, It evaluates conditions only when the neighbor node changes. Additionally, each vertex can be updated independently.

Reference

1. <https://data-flair.training/blogs/comparison-apache-flink-vs-apache-spark/>