AIM - 3, Homework I

Canevet Gaspard

June 11, 2021

Contents

1 Introduction

In the coming report, you will find some results regarding the first homework of AIM-3: Scalable Data Science: Systems and Methods course from TU Berlin. In the first part we used hadoop/map reduce technology to extract information from customers and orders dataset. In the second part we used Spark technology to study the customers and orders dataset but also to find the shortest path in a graph. In the third part we used Flink to find the shortest path in a graph with the same input as in second part. Finally, in fourth part we used Flink ability to compute streaming dataset to write a real-time event-based sports analytic application.

2 A. Hadoop/MapReduce

Below, we can find the architecture of the file A. It is made of each output file of task I and II, one pom.xml file for dependencies and program settings, one README that explains how to run codes and a src file with all java codes.

```
C:\Users\gaspa\Desktop\TU_Berlin\AIM-3\HW1_CANEVET\A>tree /F
Folder PATH listing for volume Windows
Volume serial number is 46C8-1B9F
C:.
│   output_taskI
│   output_taskII
│   pom.xml
│   README.txt
│
└───src
    └───main
        └───java
            │   CustomerOrdersDriver.java
            │   GenericCustomerEntity.java
            │   JoinMapper.java
            │   JoinReducerTaskI.java
            │   JoinReducerTaskII.java
            │
            └───model
                    AbstractCustomerEntity.java
                    Customer.java
                    CustomerOrdersVO.java
                    Orders.java
```

Figure 1: Folder A architecture

The code of this part is based on the gitlab repository [1]. In the model package we can find customers and orders classes. They are the two main classes of the project. It allows to initialise customers and orders object in the mapper. The abstract class AbstractCustomerEntity link customerId feature that is the same in both customers and orders object. Finally, the CustomerOrdersVO is a class made only of customers and orders feature that we want as an output for the task I: customer name, customer address and orders price average.
One good point is that, the mapper is the same for both task. It just reads the line of the file and thanks to number of features it creates a customers or orders object. Nevertheless we have to return the same object type. This is why we created the class GenericCustomerEntity.
Because we implement two different methods for each task we have to create two different Reducer, one fort the taskI: JoinReducerTaskI, and one for taskII: JoinReducerTaskII.
Finally we use the same driver class to run a hadoop map reduce task. Our program argument looks like "<input path customer> <input path orders> <output path> <taskI or taskII>".

2.1 Task I

In the first task, we have to implement a mapReduce algorithm to compute the customer's name, address and the average price of orders per customer who has acctbal more than 2000 and for orders placed after 1996-01-01. We can summary this task with an SQL query :

SELECT cust.name, cust.adress, AVG(orders.price)
FROM customer AS cust
JOIN orders ON cust.custkey = orders.custkey
WHERE cust.actball > 2000 and orders.orderdate > 1996-01-01
GROUPBY cust.name, cust.address

we decided to apply the filter in the reducer. To run the code, we used IntellIJ IDE. Nevertheless we could have compiled the code, exported it as a jar file and computed it with a terminal where Apache hadoop is running. I just find it simpler with IntellIJ. Find below a picture of the configuration.

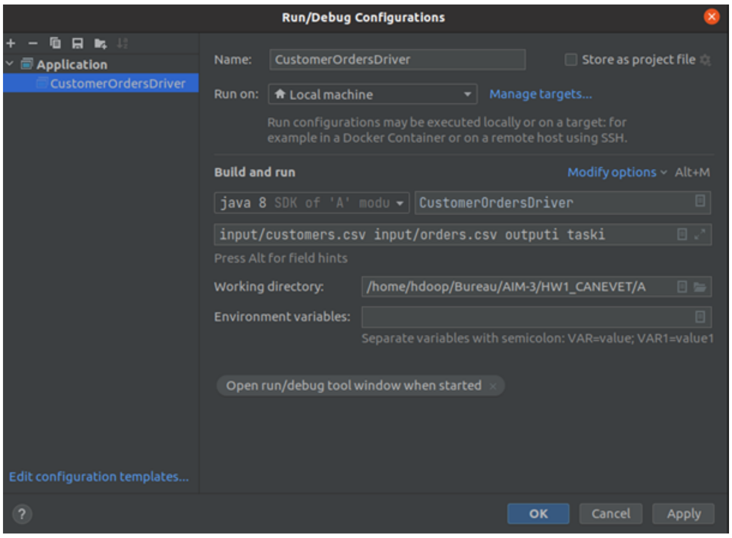
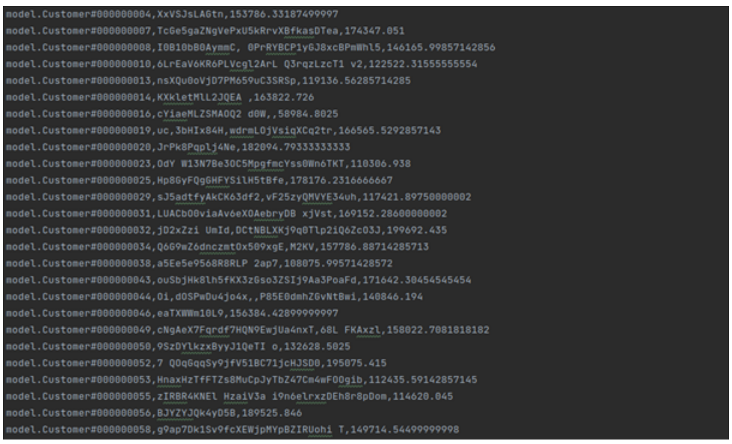


Figure 2: run configuration A task I

Then we share some of the output lines (customers.name, customer.address, AVG(orders.price)). For example the first line name : model.Customer000000004, address : XxVSJsLAGtn, average price : 153786.33187499997



```
model.Customer#000000004,XxVSJsLAGtn,153786.33187499997
model.Customer#000000007,TcGe5gaZNgVePxU5kRrvXBfkasDTea,174347.051
model.Customer#000000008,I0B10bB0AymmC, 0PrRYBCP1yGJ8xcBPmWhl5,146165.99857142856
model.Customer#000000010,6LrEaV6KR6PLVcgl2ArL Q3rqzLzcT1 v2,122522.31555555554
model.Customer#000000013,nsXQu0oVjD7PM659uC3SRSp,119136.56285714285
model.Customer#000000014,KXkletMlL2JQEA ,163822.726
model.Customer#000000016,cYiaeMLZSMAOQ2 d0W,,58984.8025
model.Customer#000000019,uc,3bHIx84H,wdrmLOjVsiqXCq2tr,166565.5292857143
model.Customer#000000020,JrPk8Pqplj4Ne,182094.79333333333
model.Customer#000000023,OdY W13N7Be3OC5MpgfmcYss0Wn6TKT,110306.938
model.Customer#000000025,Hp8GyFQgGHFYSilH5tBfe,178176.2316666667
model.Customer#000000029,sJ5adtfyAkCK63df2,vF25zyQMVYE34uh,117421.89750000002
model.Customer#000000031,LUACbO0viaAv6eXOAebryDB xjVst,169152.28600000002
model.Customer#000000032,jD2xZzi UmId,DCtNBLXKj9q0Tlp2iQ6ZcO3J,199692.435
model.Customer#000000034,Q6G9wZ6dnczmtOx509xgE,M2KV,157786.88714285713
model.Customer#000000038,a5Ee5e9568R8RLP 2ap7,108075.99571428572
model.Customer#000000043,ouSbjHk8lh5fKX3zGso3ZSIj9Aa3PoaFd,171642.30454545454
model.Customer#000000044,Oi,dOSPwDu4jo4x,,P85E0dmhZGvNtBwi,140846.194
model.Customer#000000046,eaTXWWm10L9,156384.42899999997
model.Customer#000000049,cNgAeX7Fqrdf7HQN9EwjUa4nxT,68L FKAxzl,158022.7081818182
model.Customer#000000050,9SzDYlkzxByyJ1QeTI o,132628.5025
model.Customer#000000052,7 QOqGqqSy9jfV51BC71jcHJSD0,195075.415
model.Customer#000000053,HnaxHzTfFTZs8MuCpJyTbZ47Cm4wFOOgib,112435.59142857145
model.Customer#000000055,zIRBR4KNEl HzaiV3a i9n6elrxzDEh8r8pDom,114620.045
model.Customer#000000056,BJYZYJQk4yD5B,189525.846
model.Customer#000000058,g9ap7Dk1Sv9fcXEWjpMYpBZIRUohi T,149714.54499999998
```

Figure 3: Output

Finally we share picture of the terminal in intellIJ that shows completed work.

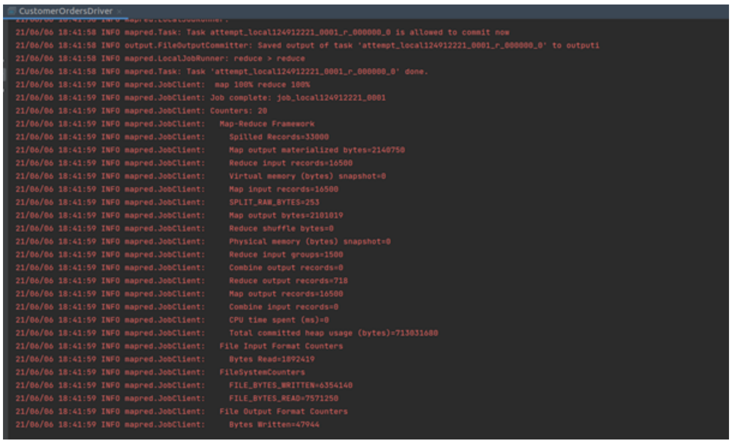


Figure 4: terminal output A task I

We can see there that the work is done (map 100% map 100%) and that lines are written in the output file (Bytes Written=47944). Finally, the output file named output_taskI is made of 718 lines.

2.2 Task II

In the second task, we have to implement a mapReduce algorithm to compute the name of all customers who did not place any order yet. We can summary this task with an SQL query :

SELECT name
FROM customers
WHERE custkey NOT IN (SELECT custkey FROM orders)

We also use intellIJ to run hadoop map reduce algorithm. Find below a picture of the configuration

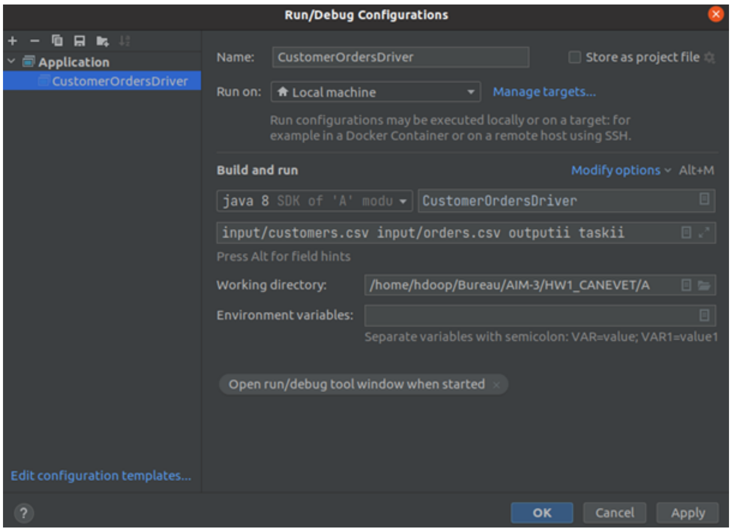


Figure 5: run configuration A task II

Then we share some of the output lines (customers.name). For example the first line name : model.Customer000000003

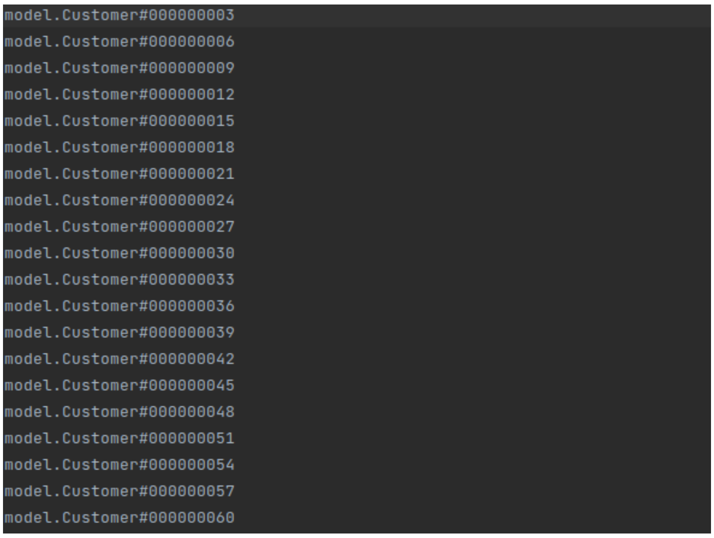


Figure 6: Output

Finally we share picture of the terminal in intellIJ that shows completed work.

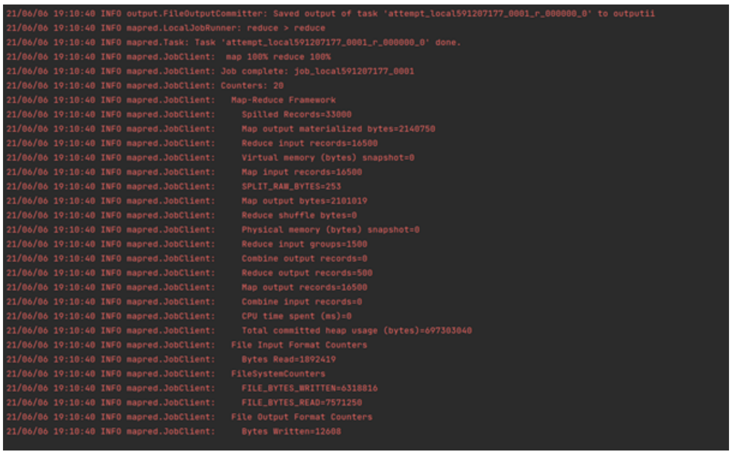


Figure 7: terminal output A task I

We can see there that the work is done (map 100% map 100%) and that lines are written in the output file (Bytes Written=12608). Finally, the output named output_taskII is made of 500 lines.

3 B. Spark

Below, we can find the architecture of the file B. It is made of each output file of task I and III, one build.sbt file for dependencies and program settings, one README that explains how to run codes and a src file with all scala codes.

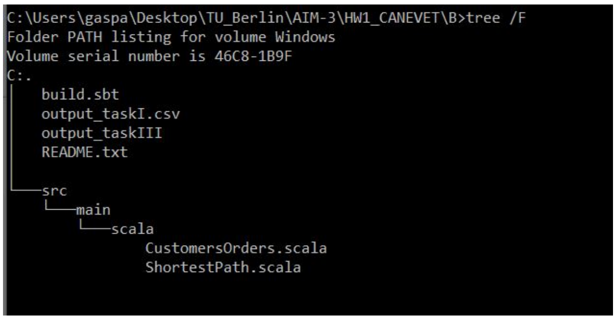


Figure 8: Folder A architecture

We decided to use Scala as the coding language in this part. It is simple to use and to understand. Here, there are two different Scala objects :

- CustomersOrders, the object that compute the task I
- ShortestPath, the object that compute the task III

3.1 Task I

as a reminder, the question of task A(I) : Implement a mapReduce algorithm to compute the customer's name, address and the average price of orders per customer who has acctbal more than 2000 and for orders placed after 1996-01-01. As we said, we can summary this task with an SQL query :

SELECT cust.name, cust.adress, AVG(orders.price)
FROM customer AS cust
JOIN orders ON cust.custkey = orders.custkey
WHERE cust.actball > 2000 and orders.orderdate > 1996-01-01
GROUPBY cust.name, cust.address

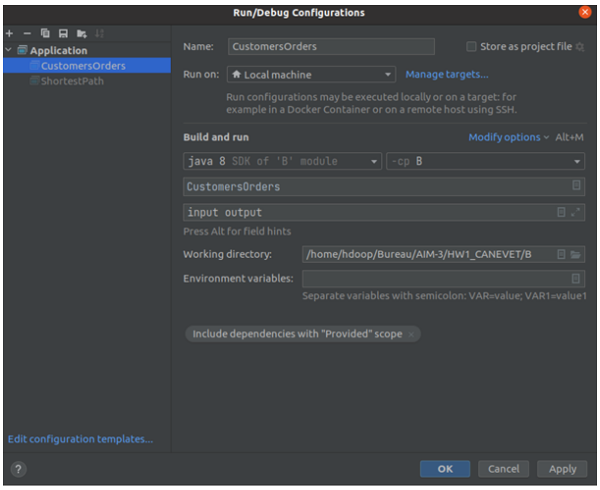


Figure 9: run configuration B task I

Here, we are going to use spark.sql to transform our input into SQL table and use the query. Finally we are going to use intellIJ to run the code. Our program argument looks like "<input path> <output path>". Find above a picture of the configuration.

Then we share some of the output lines (customers.name, customer.address, AVG(orders.price)). For example the first line name : model.Customer000000004, address : XxVSJsLAGtn, average price : 153786.331875

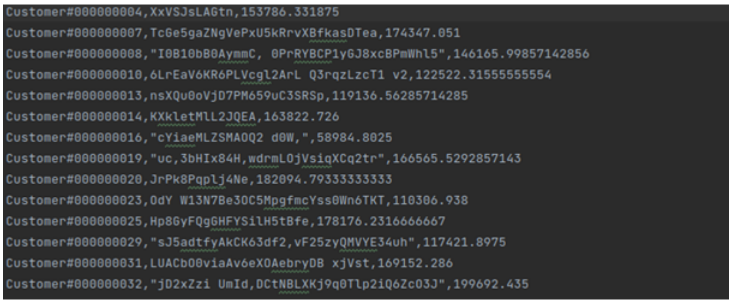


Figure 10: Output

Finally we share picture of the terminal in intellIJ that shows completed work.

Figure 11: terminal output B task I

To compare outputs from task A(I) and task B(I) we use the number of lines and the start of the document. We can see that both output have 718 lines and look quite similar (We compare the first lines and the last lines). Nevertheless, the average price from task A(I) has more significant digits.

3.2 Task II

Now, we are going to compare both implementations. First in term of expressivity. For the task A(I) we have created eight java classes with more than 50 lines per classes. For the task B(I) we have created one Scala class made of 53 lines. It is clear that Spark implementation simplify code implementation. This is the reason why Spark has been created: To simplify map reduce implementation and we can see there that is verified. Then in term of performance we ran both implementation. the first one took 7s and the second one took one minute. Thus the first one is 7 time faster. You can find below pictures of terminal outputs that expose time computing. We find two main reasons that could explain the differences in time computing. One big reason could be that we use spark.sql and note RDD implementation to compute the task. Then we used Scala for the second task and it is known that java is far more faster. Thus it is crucial to find a trade off between performance and expressivity in this type of study.

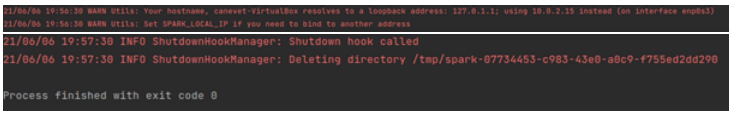


Figure 12: time computing B task I

3.3 Task III

In the task III we have to write a program that computes the shortest path from node with id 17038 to all other nodes (single source shortest path algorithm). Assuming that each edge from a source node to a destination node costs one, I used the method find on the web page [2] to implement this shortest path algorithm. It is a map reduce based method. first we have to group all source nodes and map data in order to have input like that (sourceNode, weight, state, neighbours, path).

- weight = weight of the path. If edges were differently weighted (positive or negative) it could have been a crucial point. In our case if a point were discovered, it means that we have already found the shortest path to reach that point.
- State = Discover or not
- neighbours = list of neighbours
- path = path to the node (node1 -> node2 -> ...)

Then, while our cluster is not fully explored we find reached nodes (the ones with FINITE weight and UNDISCOVER state). We change their state and we reach their neighbours. At the end we map the output and it looks like that : (initialNode -> node1 ... : path weight)

We are going to use intellIJ to run the code. Our program argument looks like "<input path> <output path>". We have to add flag -Xss512m to the JVM to perform the algorithm and prevent stackoverflow error. You can find below a picture of the configuration.

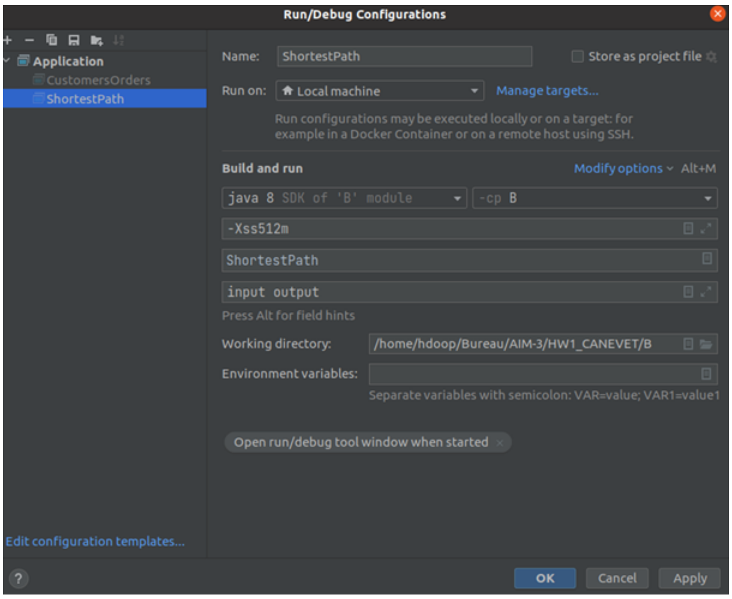


Figure 13: run configuration B task III

Then we share some of the output lines. For example the first line 17038->3466->15931->23344->10956->2350: 5

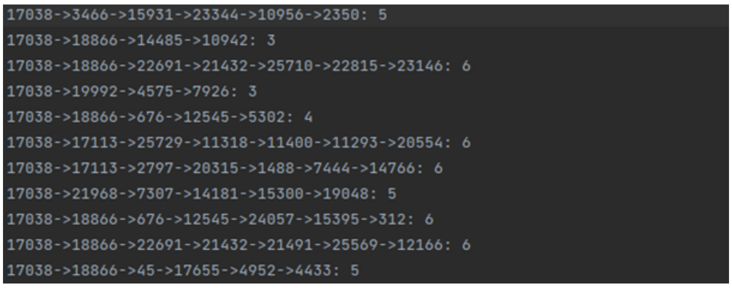


Figure 14: Output

Finally we share picture of the terminal in intellIJ that shows completed work.

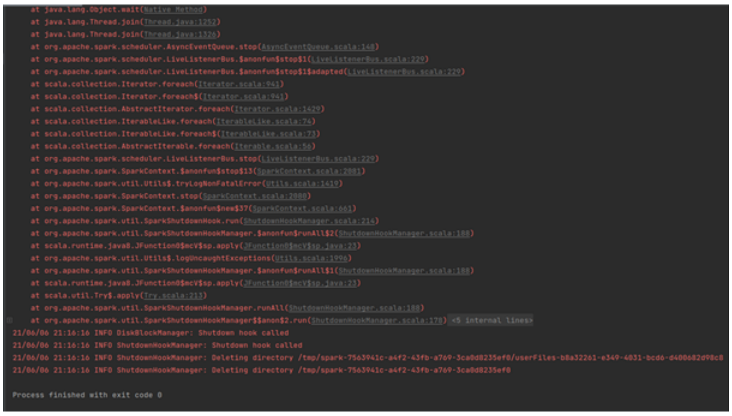
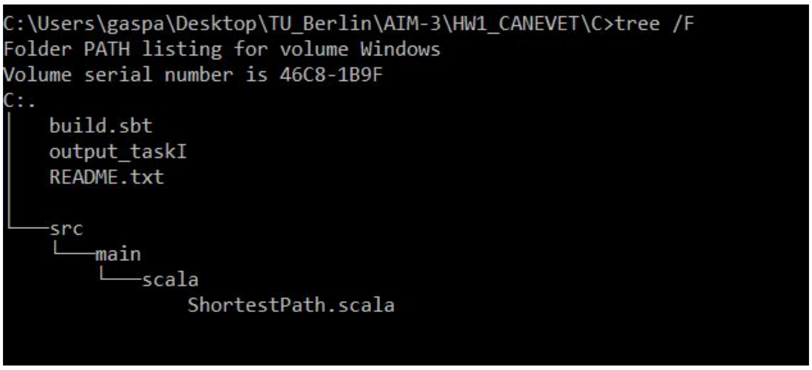


Figure 15: terminal output B task III

We can see there that jobs are done and that blocks are shutdown. To be sure that our output is proper, we can compare the number of line in our output (4158) and clusters information from the website that share the input data. It is written that the largest WCC is made of 4158 nodes. Thus, we could suppose that our output is proper.

4 C. Flink

Below, we can find the architecture of the file C. It is made of output file of task I, one build.sbt file for dependencies and program settings, one README that explains how to run codes and a src file with all scala codes.



```
C:\Users\gaspa\Desktop\TU_Berlin\AIM-3\HW1_CANEVET\C>tree /F
Folder PATH listing for volume Windows
Volume serial number is 46C8-1B9F
C:.
│   build.sbt
│   output_taskI
│   README.txt
│
└───src
    └───main
        └───scala
                ShortestPath.scala
```

Figure 16: Folder C architecture

We decided to use Flink scala api for this part. It is simple to use and it will be easier to compare two algorithms when we use same programming language. For this part we implement only one Scala object:

- ShortestPath, an object that computes the shortest path throughout a graph.

4.1 task I

We are going to use the same method as the task III from part B with the delta iteration trick from Flink api. Thus we need to defines some variables:

- a work set WS
- a solution set S
- a step function f

you can find a small schema from Flink website that explain how the iterate delta trick worked.

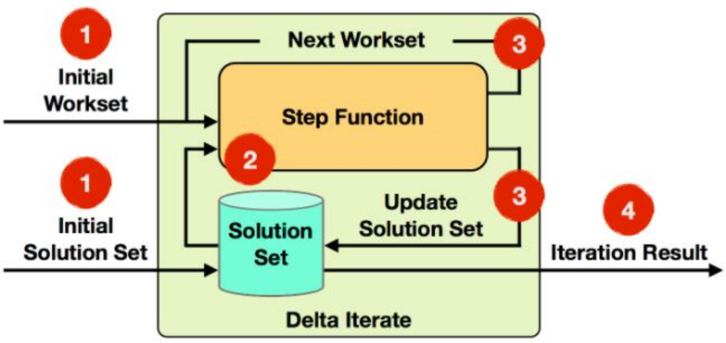


Figure 17: Delta iteration trick [3]

First, we need to define an initial work and solution set to feed the step function. Then the step function will update both set and we will iterate while the workset is not empty or while we do not reach our max iteration number.

We define our workset as the set where we put the newly reached point (the one with a FINITE weight and an UNDISCOVERED state). The initial workset looks like: (17038, 0, UNDISCOVERED, Neighbours,17038)

We define our solution set as the set where we put all the nodes with our special mapping (node,weight,state,neighbours,path). At the end of the Delta iterate we will have to filter our solution set to keep only reached nodes.

Finally our step function need to update:

- The solution set state of reached nodes (UNDISCOVERED -> DISCOVERED)
- The workset with coming reached points (UNDISCOVERED neighbours of newly DISCOVERED nodes)
- Solution set path and weight of coming reached point

Unfortunately, the implementation of deltaIteration function with Scala Flink API do not succeed. This is why I decided to use a basic while loop to perform this task. Nevertheless, I keep the algorithm settings: solution set, workset and step function.

We are going to use intellIJ to run the code. Our program argument looks like "<inputpath> <output path>". Below, a picture of the configuration.

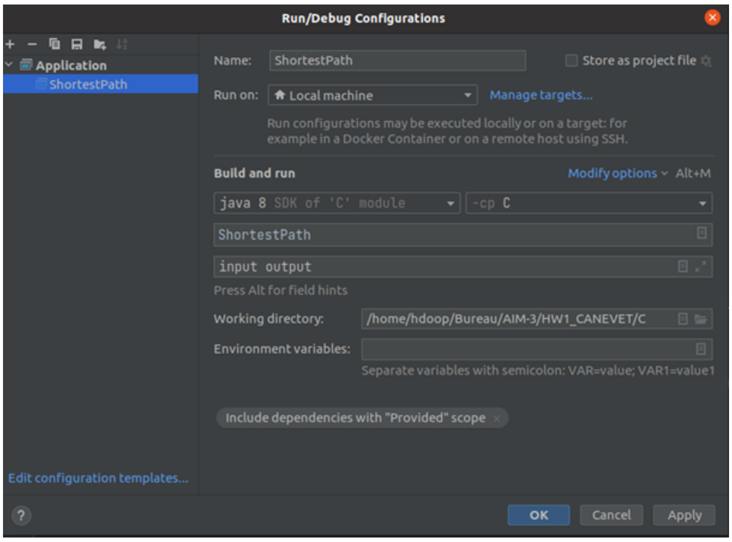
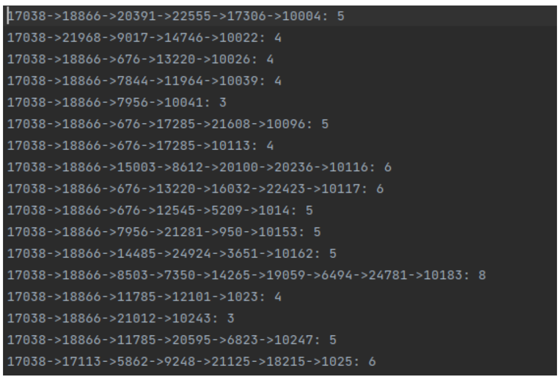


Figure 18: run configuration C task I

Then we share some of the output lines.



```
17038->18866->20391->22555->17306->10004: 5
17038->21968->9017->14746->10022: 4
17038->18866->676->13220->10026: 4
17038->18866->7844->11964->10039: 4
17038->18866->7956->10041: 3
17038->18866->676->17285->21608->10096: 5
17038->18866->676->17285->10113: 4
17038->18866->15003->8612->20100->20236->10116: 6
17038->18866->676->13220->16032->22423->10117: 6
17038->18866->676->12545->5209->1014: 5
17038->18866->7956->21281->950->10153: 5
17038->18866->14485->24924->3651->10162: 5
17038->18866->8503->7350->14265->19059->6494->24781->10183: 8
17038->18866->11785->12101->1023: 4
17038->18866->21012->10243: 3
17038->18866->11785->20595->6823->10247: 5
17038->17113->5862->9248->21125->18215->1025: 6
```

Figure 19: Output

Finally we share picture of the terminal in intellIJ that shows completed work.

```
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java ...
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.
iteration: 0
iteration: 1
iteration: 2
iteration: 3
iteration: 4
iteration: 5
iteration: 6
iteration: 7
iteration: 8
iteration: 9
iteration: 10
execution time = 28140382904ns
Process finished with exit code 0
```

Figure 20: terminal output C task I

We can see there that jobs are done and the computing time. As we did in the part B(III), we can compare the number of line in our output (4158) and clusters information from the website that share the input data. It is written that the largest WCC is made of 4158 nodes. Thus, we could suppose that our output is proper.

4.2 Task II

Lets study differences between the shortest path implementation perform in task B(III) and C(I). In term of code expressivity, it was far more easier for me to program the Flink part. After finishing the first implementation I was kind of well prepare for the second implementation and thus it was faster and easier. Nevertheless, one great advantage of flink implementation is the algorithm architecture using delta iteration. When I had to design the algorithm it was clearer to use a work set, a solution set, a step function and to loop over it. After all I think that I could have used the same architecture for Spark implementation and It would have saved me lot of time. Finally in term of code performance, it seems that Flink is really faster.

```
21/06/09 19:09:25 INFO SparkHadoopWriter: Job job_202106091909233863437654595986868_4216 committed.
execution time = 87501768352ns21/06/09 19:09:25 INFO SparkContext: Invoking stop() from shutdown hook
21/06/09 19:09:25 INFO SparkUI: Stopped Spark web UI at http://10.0.2.15:4040
```

Figure 21: time computing spark implementation

```
iteration: 9
iteration: 10
execution time = 28140382904ns
Process finished with exit code 0
```

Figure 22: time computing flink implementation

Here we can see that spark implementation take approximately 88s and flink one 28s. Flink implementation is 3 times faster. I really think that my algorithm design is really better in flink. It has surely improved the time computation.

5 D. Flink Streaming

Below, we can find the architecture of the file D. It is made of output file of task I in two parts, one pom.xml file for dependencies and program settings, one README that explains how to run codes and a src file with all java codes.

Figure 23: Folder D architecture

For this task, we used a streaming dataset made of sensor data recorded during a football match. you can find deeper information on the paper release website [5].
The total produced data rate reaches about 15,000 position events per second. The data has the following schema:

sid, ts, x, y, z, $|v|$, $|a|$, vx, vy, vz, ax, ay, az

Where sid is a sensor id, ts is a timestamp in picoseconds, x, y, z describe the position of the sensor in mm (the coordinate (0,0,0) corresponds to the center of the football field),$|v|$ (in $\mu m/s$) describes the absolute velocity and vx, vy, vz are the directed velocity vectors with a normalized size of 10,000. Similarly,$|a|$(in (in $\mu m/s^2$)), ax, ay, az describe the absolute acceleration and the directed acceleration vectors with a normalized size of 10,000.

The source code of this part is based on the gitlab repository [4]. the model DebsFeature is a Pojo in order to load streaming events. Then we will have to put our code for task I and II respectively in functions highestAverage and writeAvertedGoalEvents located in FootballStatisticsImpl class. Finally, we will have to run the main function in the class Main in order to obtain our results. Nevertheless I only have the time to perform the first task.

5.1 Task I

In the first task we have to compute the average distance that Player A1 runs in a five-minute window, where the duration between consecutive windows is one minute. We also have to report the highest average distance among all windows for Player A1. Some hints were given for this task.
First we have to filter incoming events in order to have only sensor from A1 player during the match. Thus we have to keep sensors 16 and 47, timestamps between 10 753 295 594 424 116 and 12 557 295 594 424 116 for the first half and timestamps between 13 086 639 146 403 495 and 14 879 639 146 403 495 for the second half.
Then we used sliding window method with five minutes duration each 1 minutes to compute a

moving average. The output of my moving average is a tuple (startWindowTimestamp, EndWindowTimestamp,WindowDistanceAverage, emptyStringKey). The last element of the tuple will allow us to key windows and display only the maximum average throughout all windows.
Finally to compute distance we used euclidian distance with input x and y like below:

$$d = \sqrt{x^2 + y^2} \tag{1}$$

We are going to use intellIJ to run the code. Our program argument looks like ”<inputpath>. Below, a picture of the configuration.

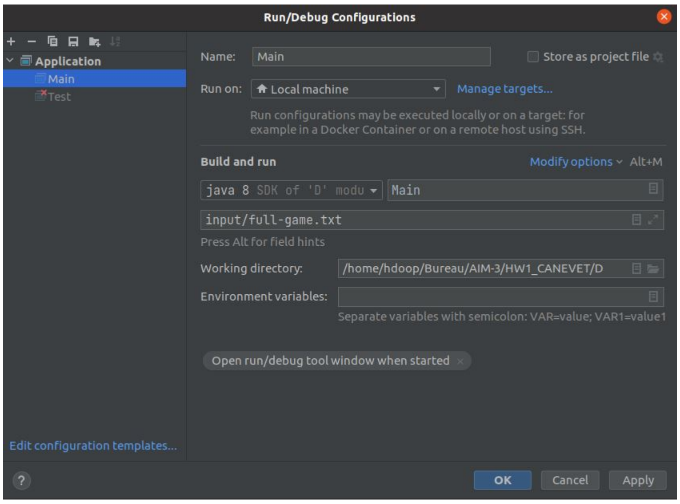
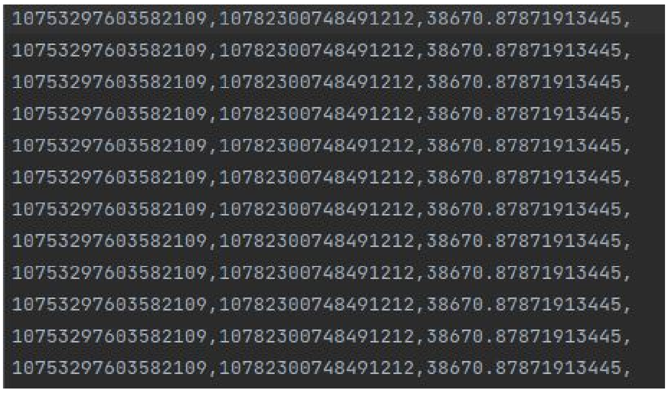


Figure 24: run configuration D task I

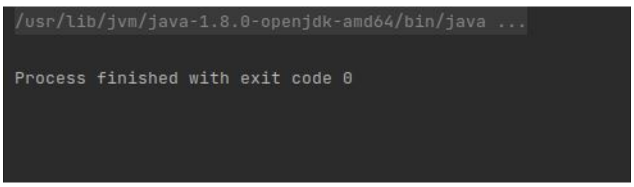
Then we share some of the output lines. (statWindowTime,endWindowTime,HighestAVG)



```
10753297603582109,10782300748491212,38670.87871913445,
10753297603582109,10782300748491212,38670.87871913445,
10753297603582109,10782300748491212,38670.87871913445,
10753297603582109,10782300748491212,38670.87871913445,
10753297603582109,10782300748491212,38670.87871913445,
10753297603582109,10782300748491212,38670.87871913445,
10753297603582109,10782300748491212,38670.87871913445,
10753297603582109,10782300748491212,38670.87871913445,
10753297603582109,10782300748491212,38670.87871913445,
10753297603582109,10782300748491212,38670.87871913445,
10753297603582109,10782300748491212,38670.87871913445,
10753297603582109,10782300748491212,38670.87871913445,
```

Figure 25: Output

Finally we share picture of the terminal in intellIJ that shows completed work. there is no crucial information, just that process is successfully finished



```
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java ...

Process finished with exit code 0
```

Figure 26: terminal output D task I

6 Conclusion

To conclude, I did not manage to do everything. I did not have the time and still the motivation to do everything. It was a really tough assignments but I learned a lot of new things. Maybe some basic example could have been shared before the homework in order to better understand the purpose of each technology. For instance some simple example on Spark and Flink could have been really useful.

7 References:

[1]$https://gitlab.tubit.tu{-}berlin.de/cschulze/AIM3{-}SQL_in_MapReduce/tree/master/src/main/java$
[2]$https://adhoop.wordpress.com/2012/03/06/shortest-path-algorithm-in-mapreduce/$
[3]$https://ci.apache.org/projects/flink/flink{-}docs{-}release{-}1.13/docs/dev/dataset/iterations/$
[4]$https://gitlab.tubit.tu-berlin.de/AIM3-SDS/ComputationalExercises$
[5]$https://dl.acm.org/doi/10.1145/2488222.2488283$