Advanced Topics in Database

Semester project

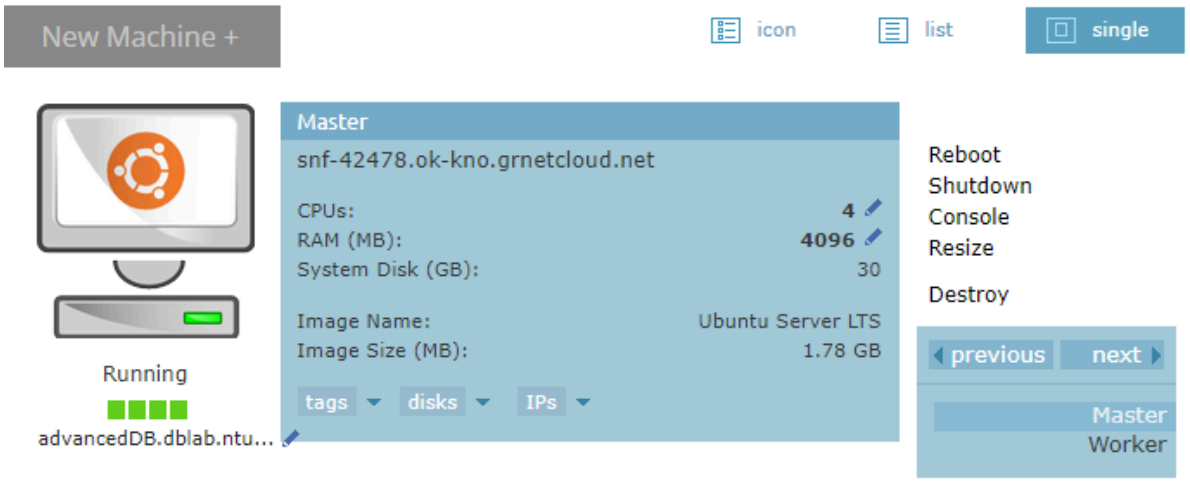
Rafał Nazarko (el23509)
Mariusz Dąbrowski (el23510)

Contents

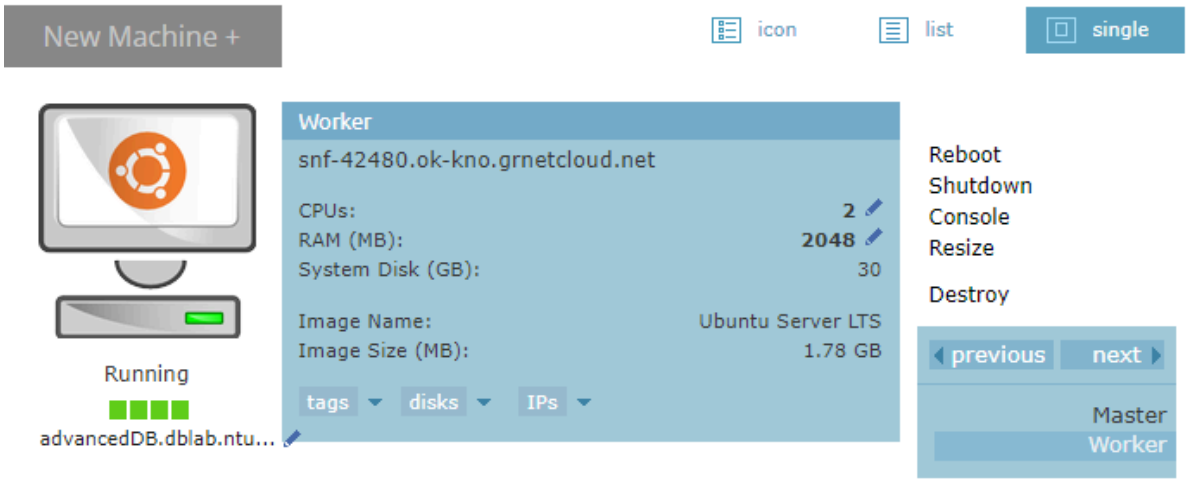
1. Okeanoss-knossos VM configuration

In the first step we have created two virtual machines: master and worker. We did everything following the instructions provided at the course description at Helios. Only thing we changed were the resources because with params given in instructions creation failed every time we tried. We found the way to create these VM with lower resources:

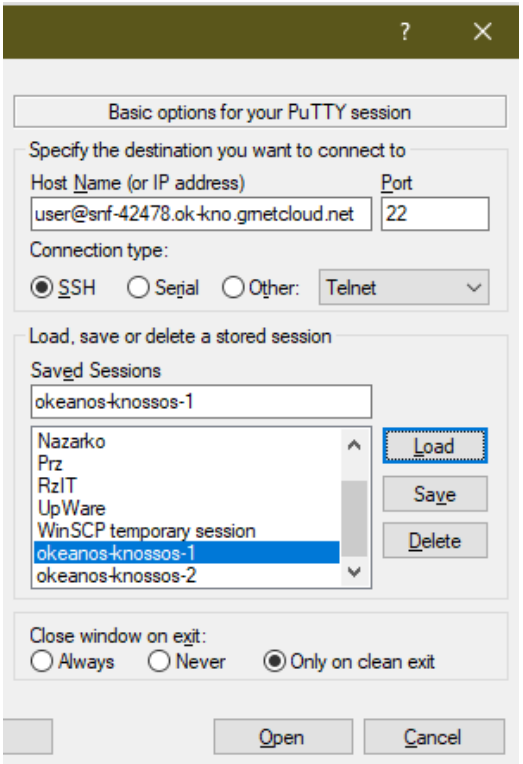
Master



Worker

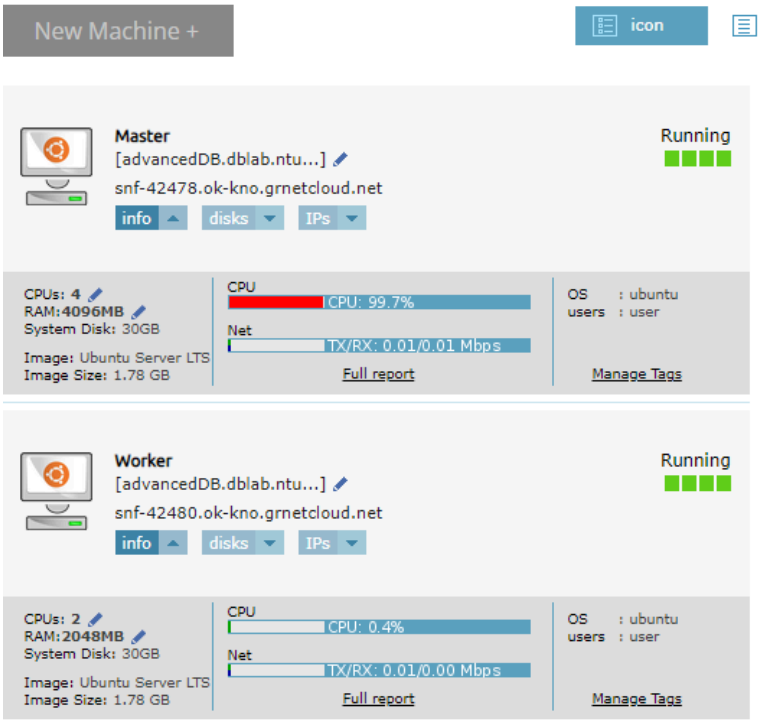


We created pair of key to login via Putty to the VMs:

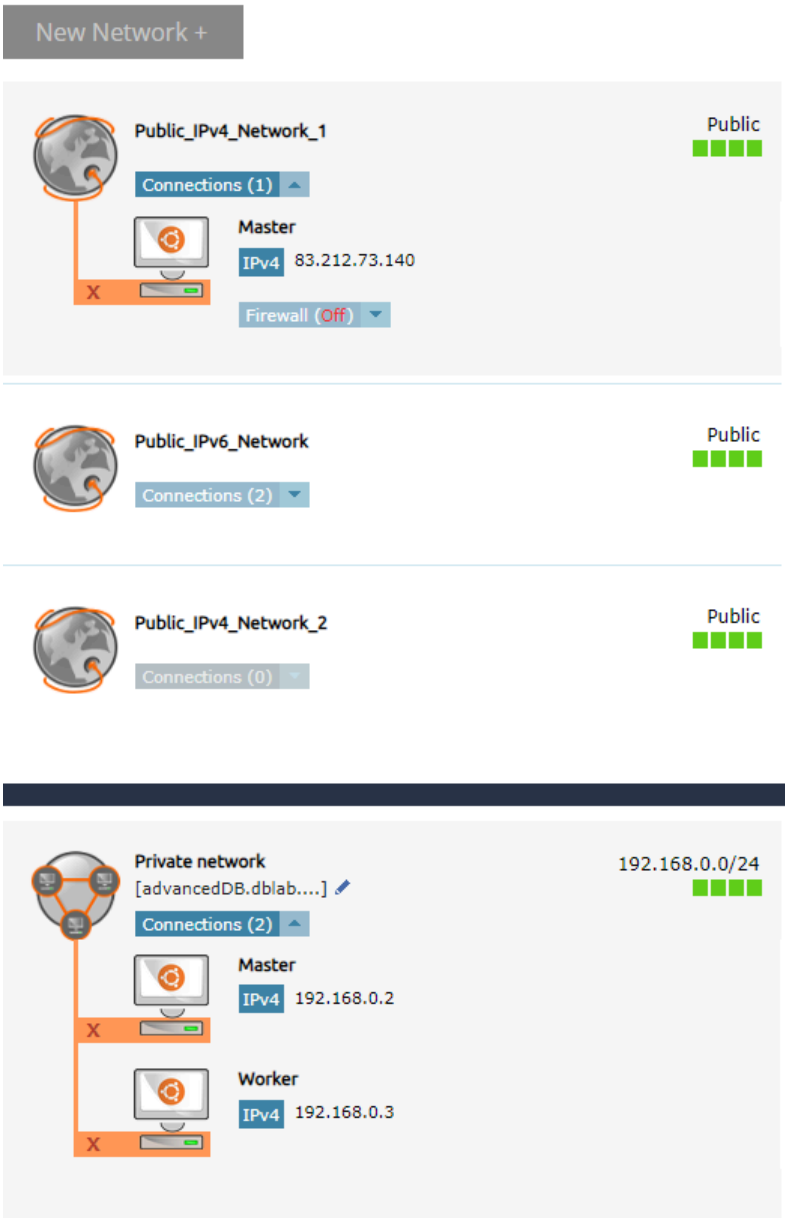


Also we omit updates of the operating system. Rest of configuration remains the same as in instruction. There is whole configuration of the VMs:

Machines



Networks



IP Addresses

At first few attempts we can't get connected to the Master VM by the public IP (even ping method). After that we established connection with the cluster and see the configurations/data on sites:

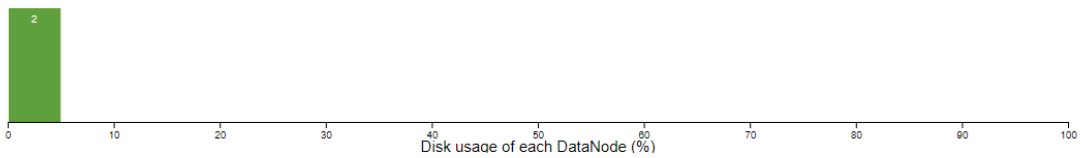
Hadoop overview

http://83.212.73.140:9870/dfshealth.html#tab-overview

Datanode Information

In service | Down | Decommissioning | Decommissioned | Decommissioned & dead
Entering Maintenance | In Maintenance | In Maintenance & dead

Datanode usage histogram



In operation

DataNode State All | Show 25 entries | Search:

Node	Http Address	Last contact	Last Block Report	Used	Non DFS Used		Capacity	Blocks	Block pool used	Version
/default-rack/okeanos-master:9866 (192.168.0.2:9866)	http://okeanos-master:9864	0s	72m	424 KB	7.07 GB	29.4 GB		1	424 KB (0%)	3.3.6
/default-rack/okeanos-worker:9866 (192.168.0.3:9866)	http://okeanos-worker:9864	2s	139m	28 KB	6.71 GB	29.4 GB		0	28 KB (0%)	3.3.6

Hadoop applications

http://83.212.73.140:8088/cluster/apps

hadoop

All Applicatic

Cluster
About
Nodes
Node Labels
Applications
NEW
NEW_SAVING
SUBMITTED
ACCEPTED
RUNNING
FINISHED
FAILED
KILLED
Scheduler
Tools

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Used Resources
12	0	0	12	0	<memory:0 B, vCores:0>

Cluster Nodes Metrics

Active Nodes	Decommissioning Nodes	Decommissioned Nodes	
2	0	0	0

Scheduler Metrics

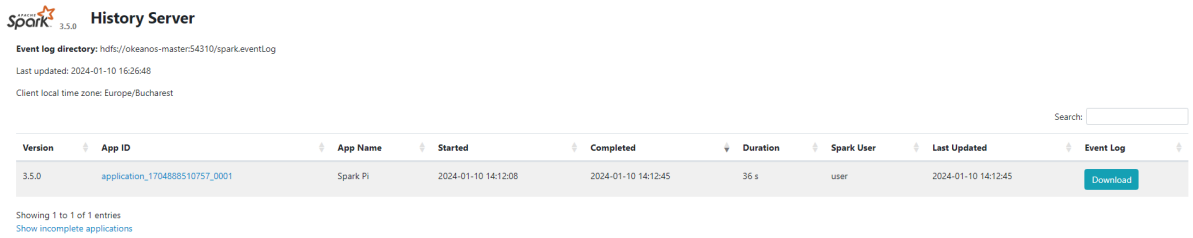
Scheduler Type	Scheduling Resource Type	Minimum Allocation	
Capacity Scheduler	[memory-mb (unit=Mi), vcores]	<memory:128, vCores:1>	<memory:6144, vC

Show 20 entries

ID	User	Name	Application Type	Application Tags	Queue	Application Priority	StartTime	LaunchTime	FinishTime	State	FinalStatus
application_1704888510757_0001	user	Spark Pi	SPARK		default	0	Wed Jan 10 14:12:23 +0200 2024	Wed Jan 10 14:12:24 +0200 2024	Wed Jan 10 14:12:45 +0200 2024	FINISHED	SUCCEEDED
application_1704888510757_0019	dr.who	get-shell	YARN		default	-1	Wed Jan 10 15:55:59 +0200 2024	Wed Jan 10 15:56:00 +0200 2024	Wed Jan 10 15:56:10 +0200 2024	FAILED	FAILED
application_1704888510757_0018	user	spark1.py	SPARK		default	0	Wed Jan 10 15:38:51 +0200 2024	Wed Jan 10 15:38:51 +0200 2024	Wed Jan 10 15:39:13 +0200 2024	FAILED	FAILED
application_1704888510757_0017	user	spark1.py	SPARK		default	0	Wed Jan 10 15:34:42 +0200 2024	Wed Jan 10 15:34:42 +0200 2024	Wed Jan 10 15:35:04 +0200 2024	FAILED	FAILED
application_1704888510757_0016	user	spark1.py	SPARK		default	0	Wed Jan 10 15:25:30 +0200 2024	Wed Jan 10 15:25:30 +0200 2024	Wed Jan 10 15:25:55 +0200 2024	FAILED	FAILED

Spark history server

http://83.212.73.140:18080/



History Server

Event log directory: hdfs://okeanos-master:54310/spark.eventLog

Last updated: 2024-01-10 16:26:48

Client local time zone: Europe/Bucharest

Search:

Version	App ID	App Name	Started	Completed	Duration	Spark User	Last Updated	Event Log
3.5.0	application_1704888510757_0001	Spark Pi	2024-01-10 14:12:08	2024-01-10 14:12:45	36 s	user	2024-01-10 14:12:45	Download

Showing 1 to 1 of 1 entries

Show incomplete applications

Apart from good configuration we cannot run our PySpark scripts on VMs. We didn't find out the solution so every script we run ended with FAILED status:

ID	User	Name	Application Type	Application Tags	Queue	Application Priority	StartTime	LaunchTime	FinishTime	State	FinalStatus
application_1704888510757_0001	user	Spark Pi	SPARK		default	0	Wed Jan 10 14:12:23 +0200 2024	Wed Jan 10 14:12:24 +0200 2024	Wed Jan 10 14:12:45 +0200 2024	FINISHED	SUCCEEDED
application_1704888510757_0019	dr.who	get-shell	YARN		default	-1	Wed Jan 10 15:55:59 +0200 2024	Wed Jan 10 15:56:00 +0200 2024	Wed Jan 10 15:56:10 +0200 2024	FAILED	FAILED
application_1704888510757_0018	user	spark1.py	SPARK		default	0	Wed Jan 10 15:38:51 +0200 2024	Wed Jan 10 15:38:51 +0200 2024	Wed Jan 10 15:39:13 +0200 2024	FAILED	FAILED
application_1704888510757_0017	user	spark1.py	SPARK		default	0	Wed Jan 10 15:34:42 +0200 2024	Wed Jan 10 15:34:42 +0200 2024	Wed Jan 10 15:35:04 +0200 2024	FAILED	FAILED
application_1704888510757_0016	user	spark1.py	SPARK		default	0	Wed Jan 10 15:25:30 +0200 2024	Wed Jan 10 15:25:30 +0200 2024	Wed Jan 10 15:25:55 +0200 2024	FAILED	FAILED
application_1704888510757_0015	user	spark1.py	SPARK		default	0	Wed Jan 10 15:24:04 +0200 2024	Wed Jan 10 15:24:05 +0200 2024	Wed Jan 10 15:24:37 +0200 2024	FAILED	FAILED
application_1704888510757_0014	user	spark1.py	SPARK		default	0	Wed Jan 10 15:17:42 +0200 2024	Wed Jan 10 15:17:42 +0200 2024	Wed Jan 10 15:18:04 +0200 2024	FAILED	FAILED
application_1704888510757_0012	user	spark1.py	SPARK		default	0	Wed Jan 10 15:13:08 +0200 2024	Wed Jan 10 15:13:09 +0200 2024	Wed Jan 10 15:13:30 +0200 2024	FAILED	FAILED
application_1704888510757_0011	user	spark1.py	SPARK		default	0	Wed Jan 10 15:07:53 +0200 2024	Wed Jan 10 15:07:53 +0200 2024	Wed Jan 10 15:08:16 +0200 2024	FAILED	FAILED
application_1704888510757_0010	user	spark1.py	SPARK		default	0	Wed Jan 10 15:03:26 +0200 2024	Wed Jan 10 15:03:27 +0200 2024	Wed Jan 10 15:03:50 +0200 2024	FAILED	FAILED
application_1704888510757_0004	dr.who	get-shell	YARN		default	-1	Wed Jan 10 14:55:59 +0200 2024	Wed Jan 10 14:56:00 +0200 2024	Wed Jan 10 14:56:08 +0200 2024	FAILED	FAILED
application_1704888510757_0002	user	spark1.py	SPARK		default	0	Wed Jan 10 14:40:54 +0200 2024	Wed Jan 10 14:40:54 +0200 2024	Wed Jan 10 14:41:15 +0200 2024	FAILED	FAILED

Every operation ends with this informations:

```
Application application_1704888510757_0018 failed 2 times due to AM Container for
appattempt_1704888510757_0018_000002 exited with exitCode: 13
Failing this attempt.Diagnostics: [2024-01-10 15:39:13.368]Exception from container-launch.
Container id: container_1704888510757_0018_02_000001
Exit code: 13
[2024-01-10 15:39:13.370]Container exited with a non-zero exit code 13. Error file: prelaunch.err.
Last 4096 bytes of prelaunch.err :
Last 4096 bytes of stderr :
Using Spark's default log4j profile: org/apache/spark/log4j2-defaults.properties
24/01/10 15:39:10 INFO SignalUtils: Registering signal handler for TERM
24/01/10 15:39:10 INFO SignalUtils: Registering signal handler for HUP
```

```
24/01/10 15:39:10 INFO SignalUtils: Registering signal handler for INT
24/01/10 15:39:11 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using
builtin-java classes where applicable
24/01/10 15:39:11 INFO ApplicationMaster: ApplicationAttemptId: appattempt_1704888510757_0018_000002
24/01/10 15:39:12 INFO ApplicationMaster: Starting the user application in a separate Thread
24/01/10 15:39:12 INFO ApplicationMaster: Waiting for spark context initialization...
24/01/10 15:39:13 ERROR ApplicationMaster: User application exited with status 1
24/01/10 15:39:13 INFO ApplicationMaster: Final app status: FAILED, exitCode: 13, (reason: User application
exited with status 1)
24/01/10 15:39:13 ERROR ApplicationMaster: Uncaught exception:
org.apache.spark.SparkException: Exception thrown in awaitResult:
at org.apache.spark.util.SparkThreadUtils$.awaitResult(SparkThreadUtils.scala:56)
at org.apache.spark.util.ThreadUtils$.awaitResult(ThreadUtils.scala:310)
at org.apache.spark.deploy.yarn.ApplicationMaster.runDriver(ApplicationMaster.scala:506)
at org.apache.spark.deploy.yarn.ApplicationMaster.run(ApplicationMaster.scala:265)
at org.apache.spark.deploy.yarn.ApplicationMaster$$anon$3.run(ApplicationMaster.scala:934)
at org.apache.spark.deploy.yarn.ApplicationMaster$$anon$3.run(ApplicationMaster.scala:933)
at java.security.AccessController.doPrivileged(Native Method)
at javax.security.auth.Subject.doAs(Subject.java:422)
at org.apache.hadoop.security.UserGroupInformation.doAs(UserGroupInformation.java:1878)
at org.apache.spark.deploy.yarn.ApplicationMaster$.main(ApplicationMaster.scala:933)
at org.apache.spark.deploy.yarn.ApplicationMaster.main(ApplicationMaster.scala)
Caused by: org.apache.spark.SparkUserAppException: User application exited with 1
at org.apache.spark.deploy.PythonRunner$.main(PythonRunner.scala:104)
at org.apache.spark.deploy.PythonRunner.main(PythonRunner.scala)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
at java.lang.reflect.Method.invoke(Method.java:498)
at org.apache.spark.deploy.yarn.ApplicationMaster$$anon$2.run(ApplicationMaster.scala:738)
24/01/10 15:39:13 INFO ApplicationMaster: Deleting staging directory
hdfs://okeanos-master:54310/user/user/.sparkStaging/application_1704888510757_0018
24/01/10 15:39:13 INFO ShutdownHookManager: Shutdown hook called
[2024-01-10 15:39:13.371]Container exited with a non-zero exit code 13. Error file: prelaunch.err.
Last 4096 bytes of prelaunch.err :
Last 4096 bytes of stderr :
Using Spark's default log4j profile: org/apache/spark/log4j2-defaults.properties
24/01/10 15:39:10 INFO SignalUtils: Registering signal handler for TERM
24/01/10 15:39:10 INFO SignalUtils: Registering signal handler for HUP
24/01/10 15:39:10 INFO SignalUtils: Registering signal handler for INT
24/01/10 15:39:11 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using
builtin-java classes where applicable
24/01/10 15:39:11 INFO ApplicationMaster: ApplicationAttemptId: appattempt_1704888510757_0018_000002
24/01/10 15:39:12 INFO ApplicationMaster: Starting the user application in a separate Thread
24/01/10 15:39:12 INFO ApplicationMaster: Waiting for spark context initialization...
24/01/10 15:39:13 ERROR ApplicationMaster: User application exited with status 1
24/01/10 15:39:13 INFO ApplicationMaster: Final app status: FAILED, exitCode: 13, (reason: User application
exited with status 1)
24/01/10 15:39:13 ERROR ApplicationMaster: Uncaught exception:
org.apache.spark.SparkException: Exception thrown in awaitResult:
at org.apache.spark.util.SparkThreadUtils$.awaitResult(SparkThreadUtils.scala:56)
at org.apache.spark.util.ThreadUtils$.awaitResult(ThreadUtils.scala:310)
at org.apache.spark.deploy.yarn.ApplicationMaster.runDriver(ApplicationMaster.scala:506)
at org.apache.spark.deploy.yarn.ApplicationMaster.run(ApplicationMaster.scala:265)
at org.apache.spark.deploy.yarn.ApplicationMaster$$anon$3.run(ApplicationMaster.scala:934)
at org.apache.spark.deploy.yarn.ApplicationMaster$$anon$3.run(ApplicationMaster.scala:933)
at java.security.AccessController.doPrivileged(Native Method)
at javax.security.auth.Subject.doAs(Subject.java:422)
at org.apache.hadoop.security.UserGroupInformation.doAs(UserGroupInformation.java:1878)
at org.apache.spark.deploy.yarn.ApplicationMaster$.main(ApplicationMaster.scala:933)
at org.apache.spark.deploy.yarn.ApplicationMaster.main(ApplicationMaster.scala)
Caused by: org.apache.spark.SparkUserAppException: User application exited with 1
at org.apache.spark.deploy.PythonRunner$.main(PythonRunner.scala:104)
at org.apache.spark.deploy.PythonRunner.main(PythonRunner.scala)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
```

```
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
at java.lang.reflect.Method.invoke(Method.java:498)
at org.apache.spark.deploy.yarn.ApplicationMaster$$anon$2.run(ApplicationMaster.scala:738)
24/01/10 15:39:13 INFO ApplicationMaster: Deleting staging directory
hdfs://okeanos-master:54310/user/user/.sparkStaging/application_1704888510757_0018
24/01/10 15:39:13 INFO ShutdownHookManager: Shutdown hook called
For more detailed output, check the application tracking page:
http://snf-42478.ok-kno.grnetcloud.net:8088/cluster/app/application_1704888510757_0018 Then click on links
to logs of each attempt.
. Failing the application.
```

To manage this problem we decided to run every script on our local PCs. Because of that we were unable to run more than 1 spark executor.

2. Tasks

2.1. Task 1:

Imports and initial project configuration:

```
import os
os.environ['JAVA_HOME']=
"D:\Erasmus\openlogic-openjdk-jre-8u382-b05-windows-32"
os.environ['SPARK_HOME']=
"D:\Erasmus\spark-3.5.0-bin-hadoop3\spark-3.5.0-bin-hadoop3"
os.environ['PYSPARK_PYTHON'] = "python"
from pyspark.sql import SparkSession
from pyspark.sql.functions import to_date

#spark connection
spark =
SparkSession.builder.master("local[*]").appName("databases_project").
getOrCreate()
#reading csv file from the disk
dataframe = spark.read.csv("Crime_Data_from_2010_to_2019.csv",
header=True, inferSchema=True)
```

2.2. Task 2:

Create a DataFrame that contains the main data-set. Keep the original column names but change the column types as instructed below:

- Date Rptd: date
- DATE OCC: date
- Vict Age: integer
- LAT: double
- LON: double

Print the total number of rows for the entire data-set and the data type of every column.

```
#setting column types
dataframe = dataframe.withColumn("Date Rptd", to_date("Date Rptd",
"MM/dd/yyyy hh:mm:ss a"))
dataframe = dataframe.withColumn("Vict Age", dataframe["Vict
Age"].cast("int"))
dataframe = dataframe.withColumn("DATE OCC", to_date("DATE OCC",
"MM/dd/yyyy hh:mm:ss a"))
dataframe = dataframe.withColumn("LON",
dataframe["LON"].cast("float"))
dataframe = dataframe.withColumn("LAT",
dataframe["LAT"].cast("float"))
print("Row count: ",dataframe.count())
dataframe.printSchema()
```

Output:

```
Row count:  2118997
root
 |-- DR_NO: integer (nullable = true)
 |-- Date Rptd: date (nullable = true)
 |-- DATE OCC: date (nullable = true)
 |-- TIME OCC: integer (nullable = true)
 |-- AREA : integer (nullable = true)
 |-- AREA NAME: string (nullable = true)
 |-- Rpt Dist No: integer (nullable = true)
 |-- Part 1-2: integer (nullable = true)
 |-- Crm Cd: integer (nullable = true)
 |-- Crm Cd Desc: string (nullable = true)
 |-- Mocodes: string (nullable = true)
 |-- Vict Age: integer (nullable = true)
 |-- Vict Sex: string (nullable = true)
 |-- Vict Descent: string (nullable = true)
 |-- Premis Cd: integer (nullable = true)
 |-- Premis Desc: string (nullable = true)
 |-- Weapon Used Cd: integer (nullable = true)
 |-- Weapon Desc: string (nullable = true)
 |-- Status: string (nullable = true)
 |-- Status Desc: string (nullable = true)
 |-- Crm Cd 1: integer (nullable = true)
 |-- Crm Cd 2: integer (nullable = true)
 |-- Crm Cd 3: integer (nullable = true)
 |-- Crm Cd 4: integer (nullable = true)
 |-- LOCATION: string (nullable = true)
```

```
|-- Cross Street: string (nullable = true)
|-- LAT: float (nullable = true)
|-- LON: float (nullable = true)
```

2.3. Task 3:

Implement Query 1 using both the DataFrame and SQL APIs. Execute both implementations with 4 Spark executors. Do you notice differences in the execution times? Justify your answer.

DataFrame approach:

```
#group by year and month and count the number of crimes
dataframe_grouped = dataframe.groupBy(year("DATE OCC").alias("Year"),
month("DATE OCC").alias("Month")) \
                .agg(count("*").alias("crime_total"))

#using Window to group the data by Year and sort by crime total
window_spec =
Window.partitionBy("Year").orderBy(dataframe_grouped["crime_total"].desc())
dataframe_ranked = dataframe_grouped.withColumn("#",
row_number().over(window_spec))

#get top-3 months with the highest number of crimes for each year
dataframe_top3 = dataframe_ranked.filter(dataframe_ranked["#"] <= 3)

#sort the results in ascending order with respect to the year and descending
order with respect to the number of crimes
dataframe_sorted = dataframe_top3.orderBy("Year",
dataframe_top3["crime_total"].desc())

#print the month, year, number of criminal acts recorded, and ranking of the
month within the respective year
dataframe_sorted.select("Month", "Year", "crime_total", "#").show()
```

Output:

Month	Year	crime_total	#
7	2010	6037	1
10	2010	6035	2
3	2010	6032	3
3	2011	14953	1
5	2011	14896	2
4	2011	14396	3
1	2012	31423	1
8	2012	31041	2
10	2012	30921	3
1	2013	8691	1
8	2013	8008	2
12	2013	8001	3
5	2014	5296	1
6	2014	5248	2
7	2014	4830	3
3	2015	10200	1
5	2015	10018	2
7	2015	9785	3
12	2016	16670	1
10	2016	16616	2

SQL approach:

```
dataframe.createOrReplaceTempView("dataframe")
spark.sql("""
SELECT Year, Month, crime_total, Rank
FROM (
    SELECT Year, Month, crime_total, ROW_NUMBER() OVER (PARTITION BY
Year ORDER BY crime_total DESC) AS Rank
    FROM (
        SELECT YEAR(`DATE OCC`) AS Year, MONTH(`DATE OCC`) AS Month,
COUNT(*) AS crime_total
        FROM dataframe
        GROUP BY Year, Month
    ) t
) t
WHERE Rank <= 3
ORDER BY Year ASC, crime_total DESC
""").show()
```

Output:

Year	Month	crime_total	Rank
2010	7	6037	1
2010	10	6035	2
2010	3	6032	3
2011	3	14953	1
2011	5	14896	2
2011	4	14396	3
2012	1	31423	1
2012	8	31041	2
2012	10	30921	3
2013	1	8691	1
2013	8	8008	2
2013	12	8001	3
2014	5	5296	1
2014	6	5248	2
2014	7	4830	3
2015	3	10200	1
2015	5	10018	2
2015	7	9785	3
2016	12	16670	1
2016	10	16616	2

2.4. Task 4:

ImplementQuery 2 using both the DataFrame/SQL and RDD APIs. Report and compare execution times for 4 Spark executors.
DataFrame approach:

```
#dividing the "Time OCC" column into 4 parts of the day
dataframe_day_parts = dataframe.withColumn("Day_Part",
                              when((col("Time OCC") >= 500) &
(col("Time OCC") < 1200), "Morning")
                              .when((col("Time OCC") >= 1200) &
(col("Time OCC") < 1700), "Afternoon")
                              .when((col("Time OCC") >= 1700) &
(col("Time OCC") < 2100), "Evening")
```

```
                                .otherwise("Night"))

#group by the day parts and count the number of crimes
dataframe_sorted = dataframe_day_parts.filter(col("Premis Desc") ==
"STREET")\
        .groupBy(col("Day_Part"))\
        .agg(count("*").alias("crime_total"))\
        .orderBy(col("crime_total").desc())
dataframe_sorted.show()
```

Output:

```
+---------+-----------+
| Day_Part|crime_total|
+---------+-----------+
|    Night|     169568|
|  Evening|     131422|
|Afternoon|     104408|
|  Morning|      87648|
+---------+-----------+
```

RDD API approach:

```
#using rdd to filter crimes commited on the street
rdd_filtered = dataframe.rdd.filter(lambda row: row["Premis Desc"] ==
"STREET")

#using rdd to map each crime record to a day part
rdd_mapped = rdd_filtered.map(
    lambda row: (
        "Morning" if 500 <= row["TIME OCC"] < 1200 else
        "Afternoon" if 1200 <= row["TIME OCC"] < 1700 else
        "Evening" if 1700 <= row["TIME OCC"] < 2100 else
        "Night"
    )
)

#using rdd to map day parts to keys
rdd_mapped = rdd_mapped.map(lambda x: (x, 1))
#using rdd to calculate conts for each day part
rdd_reduced = rdd_mapped.reduceByKey(lambda x, y: x + y)
```

```
#printing the result
for record in rdd_reduced.collect():
    print(record)
```

Output:

```
'Night', 169568)
('Morning', 87648)
('Evening', 131422)
('Afternoon', 104408)
```

2.5. Task 5:

ImplementQuery 3 using the DataFrame/SQL API. Report and compare execution times for 2, 3 and 4 Spark executors.

```
#reading income dataframe for 2015 year
income = spark.read.csv("income/LA_income_2015.csv", header=True,
inferSchema=True)
#deleting "$" and "," from Estimated Median Income column and casting to
int
income = income.withColumn("Estimated Median Income",
regexp_replace("Estimated Median Income", "\\$", ""))
income = income.withColumn("Estimated Median Income",
regexp_replace("Estimated Median Income", ",", "").cast("int"))

#sorting dataframe by descending order of income
income = income.sort(col("Estimated Median Income").desc())
#selecting top 3 and bottom 3 rows
highest_income = income.select("Zip Code").head(3)
lowest_income = income.select("Zip Code").tail(3)
#creating lists of zipcodes
highest_income_zipcodes = [row["Zip Code"] for row in highest_income]
lowest_income_zipcodes = [row["Zip Code"] for row in lowest_income]
#joining lists
zipcodes = highest_income_zipcodes + lowest_income_zipcodes

#reading dataframe for geocoding
geo = spark.read.csv("revgecoding.csv", header=True, inferSchema=True)
#casting ZIPcode column to int
```

```
geo = geo.withColumn("ZIPcode", substring(col("ZIPcode"), 0,
5).cast("int"))
#casting lat and lon columns to float
geo = geo.withColumn("LAT", geo["LAT"].cast("float"))
geo = geo.withColumn("LON", geo["LON"].cast("float"))

#filtering dataframe for 2015 year
dataframe_2015 = dataframe.filter(year("DATE OCC") == 2015)
#picking only crimes with victim age > 0
dataframe_2015 = dataframe_2015.filter(dataframe_2015["Vict Age"] > 0)
#joining dataframe with geocoding dataframe
dataframe_2015 = dataframe_2015.join(geo, on=["LAT", "LON"], how="left")
#mapping victim descent column to required format
dataframe_2015 = dataframe_2015.withColumn("Vict Descent", \
                            when(dataframe_2015["Vict Descent"] == "W",
"White") \
                            .when(dataframe_2015["Vict Descent"] == "B",
"Black") \
                            .when(dataframe_2015["Vict Descent"] == "H",
"Hispanic/Latin/Mexican") \
                            .when(dataframe_2015["Vict Descent"] == "L",
"Hispanic/Latin/Mexican") \
                            .when(dataframe_2015["Vict Descent"] == "M",
"Hispanic/Latin/Mexican") \
                            .otherwise("Unknown")
)

#filtering dataframe for zipcodes
dataframe_2015_all_zipcodes =
dataframe_2015.filter(dataframe_2015["ZIPCode"].isin(zipcodes))
dataframe_2015_highest_income =
dataframe_2015.filter(dataframe_2015["ZIPCode"].isin(highest_income_zipcod
es))
dataframe_2015_lowest_income =
dataframe_2015.filter(dataframe_2015["ZIPCode"].isin(lowest_income_zipcode
s))

#grouping by victim descent
dataframe_2015_all_zipcodes.groupBy("Vict
Descent").agg(count("*").alias("#")).sort(col("#").desc()).show()
```

Output:

```
+--------------------+---+
|        Vict Descent|  #|
+--------------------+---+
|Hispanic/Latin/Me...|118|
|               White|102|
|               Black| 99|
|             Unknown| 33|
+--------------------+---+
```

```
dataframe_2015_highest_income.groupBy("Vict
Descent").agg(count("*").alias("#")).sort(col("#").desc()).show()
```

```
+------------+---+
|Vict Descent|  #|
+------------+---+
+------------+---+
```

```
dataframe_2015_lowest_income.groupBy("Vict
Descent").agg(count("*").alias("#")).sort(col("#").desc()).show()
```

```
+--------------------+---+
|        Vict Descent|  #|
+--------------------+---+
|Hispanic/Latin/Me...|118|
|               White|102|
|               Black| 99|
|             Unknown| 33|
+--------------------+---+
```

2.6. Task 6:

Implement Query 4 using the DataFrame/SQL API.

```
#reading dataframe for police stations
police = spark.read.csv("LAPD_Police_Stations.csv", header=True,
inferSchema=True)
#casting X and Y columns to float
police = police.withColumn("X",
police["X"].cast("float").alias("Police_X"))
```

```
police = police.withColumn("Y",
police["Y"].cast("float").alias("Police_Y"))

#defining distance function
def get_distance (lat1 , lon1 , lat2 , lon2 ) :
    return round(geopy.distance.distance((lat1, lon1), (lat2,
lon2)).km, 3)

#changing function to udf
get_distance_udf = udf(get_distance, FloatType())

#deleting rows with 0 latitude and longitude
df_victims = dataframe.where((col("LAT") != 0) & (col("LON") != 0))
#picking only crimes with weapon code between 100 and 200
df_only_guns = df_victims.where((col("Weapon Used Cd") >= 100) &
(col("Weapon Used Cd") < 200))
```

Query 4a:

```
#joining dataframes
df_police = df_only_guns.join(police, [dataframe["AREA "] ==
police["PREC"]], how="left")
#selecting useful columns
df_police = df_police.select("LAT", "LON", "Police_X", "Police_Y",
"PREC", "DATE OCC", "DIVISION")
#calculating distance between crime location and police station that
was responsible for the crime
df_police = df_police.withColumn("Distance",
get_distance_udf(df_police["LAT"], df_police["LON"],
df_police["Police_Y"], df_police["Police_X"]))
#grouping by year and calculating average distance
df_police_stats = df_police.groupBy(year("DATE
OCC").alias("Year")).agg(count("*").alias("Number of Crimes"),
avg("Distance").alias("Average Distance (km)")).orderBy("Year")
#rounding the average distance to 3 decimal places
df_police_stats = df_police_stats.withColumn("Average Distance (km)",
round_df(df_police_stats["Average Distance (km)"], 3))
#grouping by division and calculating average distance
df_police_stats2 =
df_police.groupBy("DIVISION").agg(count("*").alias("Number of
Crimes"), avg("Distance").alias("Average Distance
```

```
(km)")).orderBy(col("Number of Crimes").desc())
#rounding the average distance to 3 decimal places
df_police_stats2 = df_police_stats2.withColumn("Average Distance
(km)", round_df(df_police_stats2["Average Distance (km)"], 3))
```

Output:

```
df_police_stats.show()
```

Year	Number of Crimes	Average Distance (km)
2010	2686	2.56
2011	5855	2.75
2012	12021	2.836
2013	2271	2.731
2014	2320	2.684
2015	3500	2.653
2016	6537	2.682
2017	9890	2.764
2018	2242	2.565
2019	7129	2.739
2021	10175	2.695
2022	9884	2.709
2023	2820	2.862

```
df_police_stats2.show()
```

DIVISION	Number of Crimes	Average Distance (km)
77TH STREET	12704	2.677
SOUTHEAST	8373	2.097
NEWTON	7908	2.019
SOUTHWEST	6301	2.689
HOLLENBECK	4346	2.719
HARBOR	3881	4.077
RAMPART	3335	1.631
NORTHEAST	2967	3.876
OLYMPIC	2897	1.829
MISSION	2877	4.666
FOOTHILL	2618	3.852
HOLLYWOOD	2599	1.447
WILSHIRE	2472	2.333
NORTH HOLLYWOOD	2449	2.741
WEST VALLEY	2207	3.581
VAN NUYS	1861	2.193
PACIFIC	1821	3.742
DEVONSHIRE	1791	3.977
CENTRAL	1480	1.234
TOPANGA	1323	3.438

Query 4b:

```
#doing the same for query 4 b
police_cross_join = police.crossJoin(df_only_guns)
#calculating distance between crime location and closest police
station
police_cross_join = police_cross_join.withColumn("Distance",
get_distance_udf(police_cross_join["LAT"], police_cross_join["LON"],
police_cross_join["Police_Y"], police_cross_join["Police_X"]))
#selecting closest police station using window
window_spec =
Window.partitionBy("DR_NO").orderBy(col("Distance").asc())
closest_police_station_df = police_cross_join.withColumn("row_num",
row_number().over(window_spec)).filter("row_num = 1").drop("row_num")
df_police_stats = closest_police_station_df.groupBy(year("DATE
OCC").alias("Year")).agg(count("*").alias("Number of Crimes"),
avg("Distance").alias("Average Distance (km)")).orderBy("Year")
df_police_stats = df_police_stats.withColumn("Average Distance (km)",
round_df(df_police_stats["Average Distance (km)"], 3))
df_police_stats2 =
closest_police_station_df.groupBy("DIVISION").agg(count("*").alias("N
umber of Crimes"), avg("Distance").alias("Average Distance
(km)")).orderBy(col("Number of Crimes").desc())
df_police_stats2 = df_police_stats2.withColumn("Average Distance
(km)", round_df(df_police_stats2["Average Distance (km)"], 3))
df_police_stats.show()
```

Output:

Year	Number of Crimes	Average Distance (km)
2010	2686	2.271
2011	5579	2.409
2012	6532	2.505
2013	2271	2.415
2014	2320	2.16
2015	3500	2.46
2016	6076	2.414
2017	7786	2.392
2018	2242	2.348
2019	7129	2.429
2021	8483	2.481
2022	6963	2.404
2023	2820	2.596

```
df_police_stats2.show()
```

Output:

DIVISION	Number of Crimes	Average Distance (km)
77TH STREET	7468	1.717
SOUTHWEST	7275	2.228
SOUTHEAST	6363	2.206
HOLLENBECK	4376	2.703
WILSHIRE	4256	2.46
HARBOR	3800	3.91
NEWTON	3688	1.578
HOLLYWOOD	3550	1.97
OLYMPIC	3137	1.676
RAMPART	2913	1.419
VAN NUYS	2656	2.955
FOOTHILL	2487	3.616
NORTH HOLLYWOOD	2001	2.737
NORTHEAST	1851	3.736
CENTRAL	1588	1.05
WEST VALLEY	1541	2.805
MISSION	1505	3.811
TOPANGA	1327	3.029
PACIFIC	1222	3.755
DEVONSHIRE	747	2.988

3. Github repository of our project

Our source code is available on the Github Repository.