VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

SEMESTER 241 - CO3071

LAB 2 REPORT

DISTRIBUTED SYSTEMS

Instructor: **PROF. Nguyễn Mạnh Thìn**

Tạ Gia Khang - 2152642
Nguyễn Quang Thiện - 2152994

HO CHI MINH CITY, NOVEMBER 2024

Contents

1 Homework **3**
1.1 Configuration 3
1.2 Task 1: Filtering Wrong Records 5
1.2.1 Mission 1: Log File Filtering Implementation 5
1.2.2 Mission 2: Sort by Time Request 7
1.3 Task 2: Preprocessing 8
1.3.1 Mission 1: Print out the number of records for each service group 8
1.3.2 Mission 2: Print out a list of unique IPs from the log 10
1.3.3 Mission 3: Build an RDD, which contains the map of IPs and their additional information 10
1.4 Task 3: Analysis 14
1.4.1 Mission 1: Print out the number of HIT, MISS, and HIT1 requests 14
1.4.2 Mission 2: Compute the HitRate of the system 15
1.4.3 Mission 3: Identify the ISP with the Highest Hit Rate 16

1 Homework

1.1 Configuration

Generating a docker-compose.yml file to set up a distributed system with Apache **Spark** and **Hadoop**. The system consists of Spark Master, 2 Spark Workers, a Hadoop Namenode, and a Datanode.

```yaml
version: '2'

services:
  # Configures the Spark Master with Bitnami Spark image
  spark-master:
    image: bitnami/spark:latest
    container_name: spark-master
    environment:
      - SPARK_MODE=master
      - SPARK_RPC_AUTHENTICATION_ENABLED=no
      - SPARK_RPC_ENCRYPTION_ENABLED=no
      - SPARK_LOCAL_STORAGE_ENCRYPTION_ENABLED=no
      - SPARK_SSL_ENABLED=no
    ports:
      - "8080:8080"  # Spark Master web UI
      - "7077:7077"  # Spark Master port
    networks:
      - lab2_ds_spark_network

  # Set up as Spark Workers connecting to the Spark Master
  spark-worker1:
    image: bitnami/spark:latest
    container_name: spark-worker1
    environment:
      - SPARK_MODE=worker
      - SPARK_MASTER_URL=spark://spark-master:7077
    depends_on:
      - spark-master
    networks:
      - lab2_ds_spark_network

  spark-worker2:
    image: bitnami/spark:latest
    container_name: spark-worker2
    environment:
      - SPARK_MODE=worker
      - SPARK_MASTER_URL=spark://spark-master:7077
    depends_on:
      - spark-master
    networks:
      - lab2_ds_spark_network
```

```yaml

  # Configures the Hadoop Namenode with HDFS cluster details
  namenode:
    image: bde2020/hadoop-namenode:latest
    container_name: namenode
    environment:
      - CLUSTER_NAME=hadoop_cluster
      - CORE_CONF_fs_defaultFS=hdfs://namenode:9000
    ports:
      - "9000:9000"  # HDFS namenode RPC port
      - "9870:9870"  # HDFS namenode Web UI
    networks:
      - lab2_ds_spark_network
    volumes:
      # Mounts local folder ./namenode to the container's /hadoop/dfs/name directory
      - ./namenode:/hadoop/dfs/name

  # Configures the Hadoop Datanode to connect to the Namenode
  datanode:
    image: bde2020/hadoop-datanode:latest
    container_name: datanode
    environment:
      - CORE_CONF_fs_defaultFS=hdfs://namenode:9000
    depends_on:
      - namenode
    networks:
      - lab2_ds_spark_network
    volumes:
      # Mounts local folder ./datanode to the container's /hadoop/dfs/data directory
      - ./datanode:/hadoop/dfs/data

networks:
  # Configures a custom external network
  lab2_ds_spark_network:
    external: true

volumes:
  # Defines volumes for persistent storage of Namenode and Datanode data
  namenode:
  datanode:
```

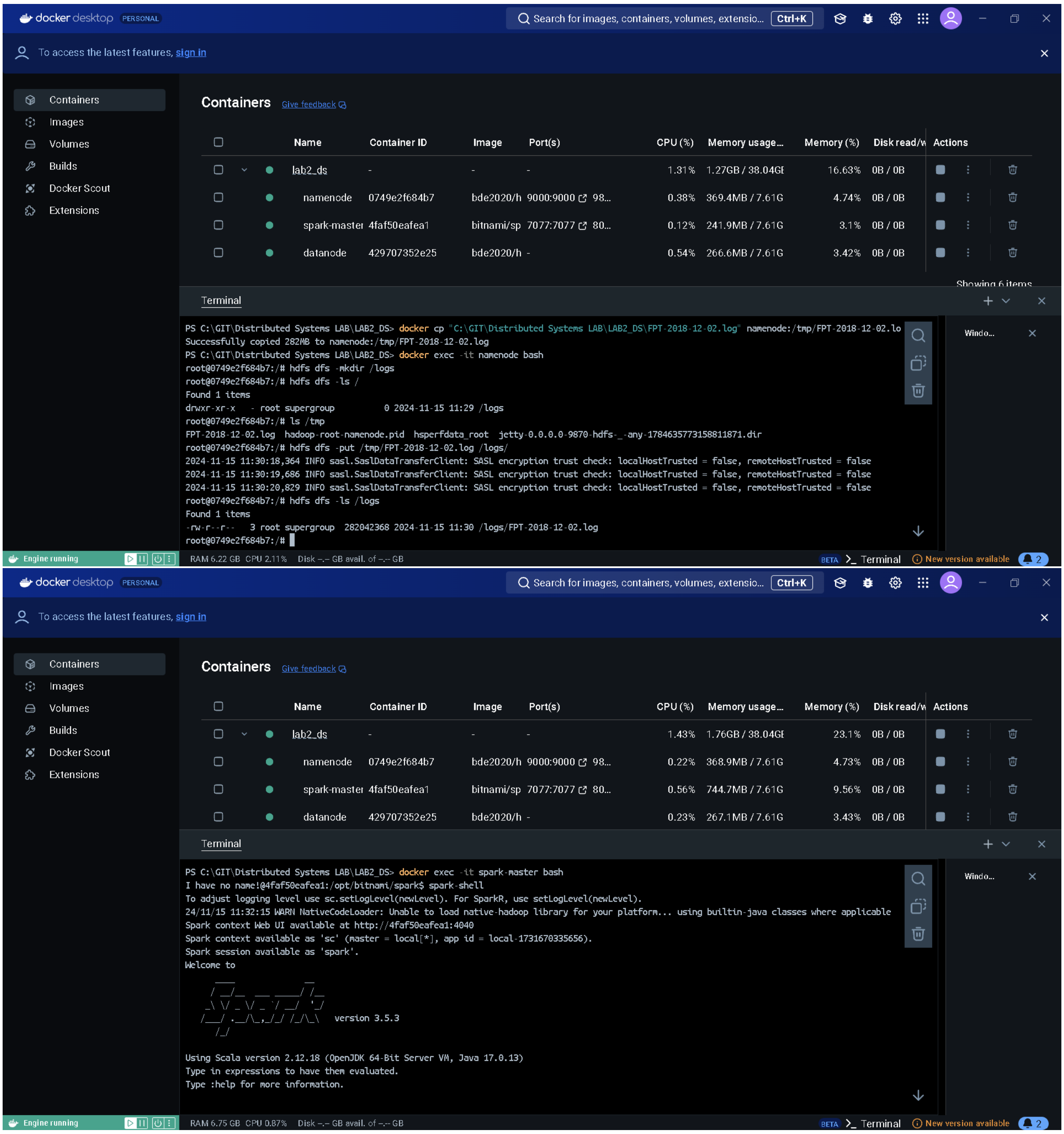


Figure 1: Setup HDFS and Spark

1.2 Task 1: Filtering Wrong Records

1.2.1 Mission 1: Log File Filtering Implementation

The log file contains records with fields such as latency, IP address, cache status, timestamp, content path, and size. The goal is to filter out incorrect records.

Step 1: Load Log Data To process log data from HDFS, the following code snippet is used to load it into an RDD:

```
val data = sc.textFile("hdfs://namenode:9000/logs/FPT-2018-12-02.log")
data.count()
```

```
scala> val data = sc.textFile("hdfs://namenode:9000/logs/FPT-2018-12-02.log")
data: org.apache.spark.rdd.RDD[String] = hdfs://namenode:9000/logs/FPT-2018-12-02.log MapPartitionsRDD[1] at textFile at <console>:23

scala> data.count()
res0: Long = 1896813
```

Figure 2: New RDD

Details:

- The log file path corresponds to data from December 2, 2018.
- The `count()` method counts the total records in the dataset.

Step 2: Validate Records Records are validated using a function that checks the number of fields, latency, content size, and cache status:

```
def record_filtered(line: String): Boolean = {
    val fields = line.split(" ")
    val criteria = fields.length == 7 &&
        fields(0).toDouble >= 0 &&
        fields(6).forall(Character.isDigit) &&
        fields(6).toInt > 0 &&
        fields(2) != "-"
  (criteria)
}
```

Validation Criteria:

1. Six fields in total.
2. Non-negative latency values.
3. Content size is a valid integer and greater than zero.
4. Cache status is not local (`fields(2) != "-"`).

Step 3: Apply Filters The filtering is applied as follows:

```
val record_data_filtered = data.filter(record_filtered)
record_data_filtered.count()

def fail_record_filtered(line: String): Boolean = !record_filtered(line)
val fail_record_data_filtered = data.filter(fail_record_filtered)
fail_record_data_filtered.count()
```

- `record_data_filtered`: RDD with valid records.

```
scala> def record_filtered(line: String):
     | Boolean =
     | {
     |   val fields = line.split(" ")
     |     // 6 fields <=> 7 spaces, Latency >= 0
     |     // Content must be a number, Content size is positive
     |     // Hit status is not '-'
     |   val criteria = fields.length == 7 &&
     |     fields(0).toDouble >= 0 &&
     |     fields(6).forall(Character.isDigit) &&
     |     fields(6).toInt > 0 &&
     |     fields(2) != "-"
     |   (criteria)
     | }
record_filtered: (line: String)Boolean

scala> val record_data_filtered = data.filter(record_filtered)
record_data_filtered: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[2] at filter at <console>:24

scala> record_data_filtered.count()
res1: Long = 1303227
```

Figure 3: Number of correct records

- `fail_record_data_filtered`: RDD with invalid records.

```
scala> def fail_record_filtered(line: String):
     | Boolean = !record_filtered(line)
fail_record_filtered: (line: String)Boolean

scala> val fail_record_data_filtered = data.filter(fail_record_filtered)
fail_record_data_filtered: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[3] at filter at <console>:24

scala> fail_record_data_filtered.count()
res2: Long = 593586
```

Figure 4: Number of incorrect records

1.2.2 Mission 2: Sort by Time Request

We used the sortBy() and pass into it a specific function for sorting by the "Time Request" value:

```scala
def convert_to_timestamp(line: String):
Long =
{
  val fields = line.split(" ")
  val inputData = fields(3) + " " + fields(4)
  val timeFormat = new SimpleDateFormat("[dd/MMM/yyyy:HH:mm:ss z]")
  timeFormat.setTimeZone(TimeZone.getTimeZone("GMT"))
  val timeStamp = timeFormat.parse(inputData).getTime()

  timeStamp
}
record_data_filtered.sortBy(convert_to_timestamp).take(10).foreach(println)
fail_record_data_filtered.sortBy(convert_to_timestamp).take(10).foreach(println)
```

```
scala> record_data_filtered.sortBy(convert_to_timestamp).take(10).foreach(println)
0.000 58.187.29.147 HIT [02/Dec/2018:00:00:00 +0700] /live/prod_kplus_ns_hd/prod_kplus_ns_hd.isml/events(1541466558)/dash/prod_kplus_ns_hd-audio_vie=56000-49397873671168.dash 28401
0.055 113.23.26.76 HIT [02/Dec/2018:00:00:00 +0700] /live/prod_kplus_1_hd/prod_kplus_1_hd.isml/events(1541466464)/dash/prod_kplus_1_hd-video=2499968-926210122800.dash 1265928
0.000 118.69.60.62 HIT [02/Dec/2018:00:00:00 +0700] /live/prod_kplus_pm_hd/prod_kplus_pm_hd.isml/stb.mpd 39902
0.000 42.118.29.197 HIT [02/Dec/2018:00:00:00 +0700] /live/prod_kplus_ns_hd/prod_kplus_ns_hd.isml/prod_kplus_ns_hd.mpd 40102
0.000 14.231.34.1 HIT [02/Dec/2018:00:00:00 +0700] /cc8c96ca2e6b3aa3465a5e2d383c8e011543687445/box/_definst_/vtv3-high.m3u8 323
0.001 42.113.129.241 HIT [02/Dec/2018:00:00:00 +0700] /d053d51fe6c3935d4c25908089b7c4961543692607/ndvr/vtv3/_definst_/20181201/vtv3-high-20181201175215-611181.ts 742224
0.002 14.244.239.143 HIT [02/Dec/2018:00:00:00 +0700] /8d7863dc41865b32054cd8478430fb521543689560/box/_definst_/vtv2-high-2063449.ts 760272
0.000 14.229.126.144 HIT [02/Dec/2018:00:00:00 +0700] /8c6bebd4edd78750291593f1756e861a1543686153/box/_definst_/vtv6-high.m3u8 323
0.000 113.22.7.224 HIT [02/Dec/2018:00:00:00 +0700] /live/prod_kplus_ns_hd/prod_kplus_ns_hd.isml/events(1541466558)/dash/prod_kplus_ns_hd-audio_vie=56000-49397873798144.dash 28840
0.000 113.179.83.143 HIT [02/Dec/2018:00:00:00 +0700] /98caef0ff9b853515fe1e3858397badf1543685819/box/_definst_/vtv6-high.m3u8 323
```

Figure 5: Top 10 of correct list

```
scala> record_data_filtered.sortBy(convert_to_timestamp).take(10).foreach(println)
0.000 58.187.29.147 HIT [02/Dec/2018:00:00:00 +0700] /live/prod_kplus_ns_hd/prod_kplus_ns_hd.isml/events(1541466558)/dash/prod_kplus_ns_hd-audio_vie=56000-49397873671168.dash 28401
0.055 113.23.26.76 HIT [02/Dec/2018:00:00:00 +0700] /live/prod_kplus_1_hd/prod_kplus_1_hd.isml/events(1541466464)/dash/prod_kplus_1_hd-video=2499968-926210122800.dash 1265928
0.000 118.69.60.62 HIT [02/Dec/2018:00:00:00 +0700] /live/prod_kplus_pm_hd/prod_kplus_pm_hd.isml/stb.mpd 39902
0.000 42.118.29.197 HIT [02/Dec/2018:00:00:00 +0700] /live/prod_kplus_ns_hd/prod_kplus_ns_hd.isml/prod_kplus_ns_hd.mpd 40102
0.000 14.231.34.1 HIT [02/Dec/2018:00:00:00 +0700] /cc8c96ca2e6b3aa3465a5e2d383c8e011543687445/box/_definst_/vtv3-high.m3u8 323
0.001 42.113.129.241 HIT [02/Dec/2018:00:00:00 +0700] /d053d51fe6c3935d4c25908089b7c4961543692607/ndvr/vtv3/_definst_/20181201/vtv3-high-20181201175215-611181.ts 742224
0.002 14.244.239.143 HIT [02/Dec/2018:00:00:00 +0700] /8d7863dc41865b32054cd8478430fb521543689560/box/_definst_/vtv2-high-2063449.ts 760272
0.000 14.229.126.144 HIT [02/Dec/2018:00:00:00 +0700] /8c6bebd4edd78750291593f1756e861a1543686153/box/_definst_/vtv6-high.m3u8 323
0.000 113.22.7.224 HIT [02/Dec/2018:00:00:00 +0700] /live/prod_kplus_ns_hd/prod_kplus_ns_hd.isml/events(1541466558)/dash/prod_kplus_ns_hd-audio_vie=56000-49397873798144.dash 28840
0.000 113.179.83.143 HIT [02/Dec/2018:00:00:00 +0700] /98caef0ff9b853515fe1e3858397badf1543685819/box/_definst_/vtv6-high.m3u8 323
```

Figure 6: Top 10 of incorrect list

1.3 Task 2: Preprocessing

After filtering out incorrect records, the next step is to analyze the log file by extracting and transforming raw data into more meaningful formats. The log file predominantly contains records from two services: web services and video streaming services. The video streaming service utilizes two protocols, HLS and MPEG-DASH. Based on the content name, records can be categorized into three groups, following these rules:

- HLS: the extension of the content name is ".mpd" or ".m3u8"
- MPEG-DASH: the extension of the content name is ".dash" or ".ts".
- Web service: They are the remaining records

1.3.1 Mission 1: Print out the number of records for each service group

```scala
def service_classified(line: String):
String =
{
  val content_name = line.split(" ")(5)
  if (content_name.endsWith(".mpd") || content_name.endsWith(".m3u8"))
    "HLS"
  else if (content_name.endsWith(".dash") || content_name.endsWith(".ts"))
    "MPEG-DASH"
  else
    "Web Service"
}
```

```
val data_classified = record_data_filtered.map(line => (service_classified(line),
1))
val serviceGroupCounts = data_classified.reduceByKey(_ + _)
serviceGroupCounts.collect().foreach {case (serviceGroup, count) =>
println(s"$serviceGroup: $count records")}
```

We define a function `service_classified` to categorize the services. The function works as follows:

- Extracts the content name from the 6th field of the log record.
- Categorizes records based on their extensions:
  - If the content name ends with ".mpd" or ".m3u8," it is classified as **HLS**.
  - If it ends with ".dash" or ".ts," it is classified as **MPEG-DASH**.
  - All other records are classified as **Web Service**.

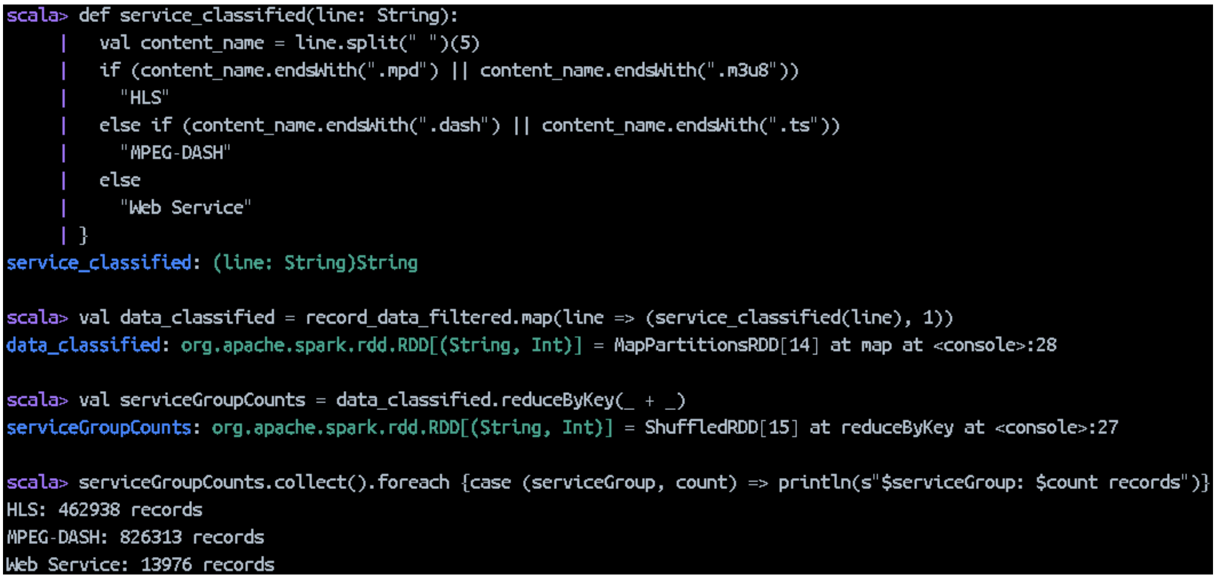
For the filtered and classified data RDD:

- The function `service_classified` is applied to each line of the filtered data.
- A new RDD is created with key-value pairs, where the key is the service group (**HLS**, **MPEG-DASH**, or **Web Service**), and the value is 1.

To summarize, the new RDD `serviceGroupCounts`:

- Uses the `reduceByKey()` transformation to group records by service group and calculate the total counts.
- Produces an RDD containing key-value pairs, where the key is the service group, and the value is the total record count.

The following image illustrates the program's output:



```
scala> def service_classified(line: String):
     |   val content_name = line.split(" ")(5)
     |   if (content_name.endsWith(".mpd") || content_name.endsWith(".m3u8"))
     |     "HLS"
     |   else if (content_name.endsWith(".dash") || content_name.endsWith(".ts"))
     |     "MPEG-DASH"
     |   else
     |     "Web Service"
     | }
service_classified: (line: String)String

scala> val data_classified = record_data_filtered.map(line => (service_classified(line), 1))
data_classified: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[14] at map at <console>:28

scala> val serviceGroupCounts = data_classified.reduceByKey(_ + _)
serviceGroupCounts: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[15] at reduceByKey at <console>:27

scala> serviceGroupCounts.collect().foreach {case (serviceGroup, count) => println(s"$serviceGroup: $count records")}
HLS: 462938 records
MPEG-DASH: 826313 records
Web Service: 13976 records
```

Figure 7: Classify Service

1.3.2 Mission 2: Print out a list of unique IPs from the log

We define a function `extractIP` to retrieve IP addresses from the log records. The function works as follows:

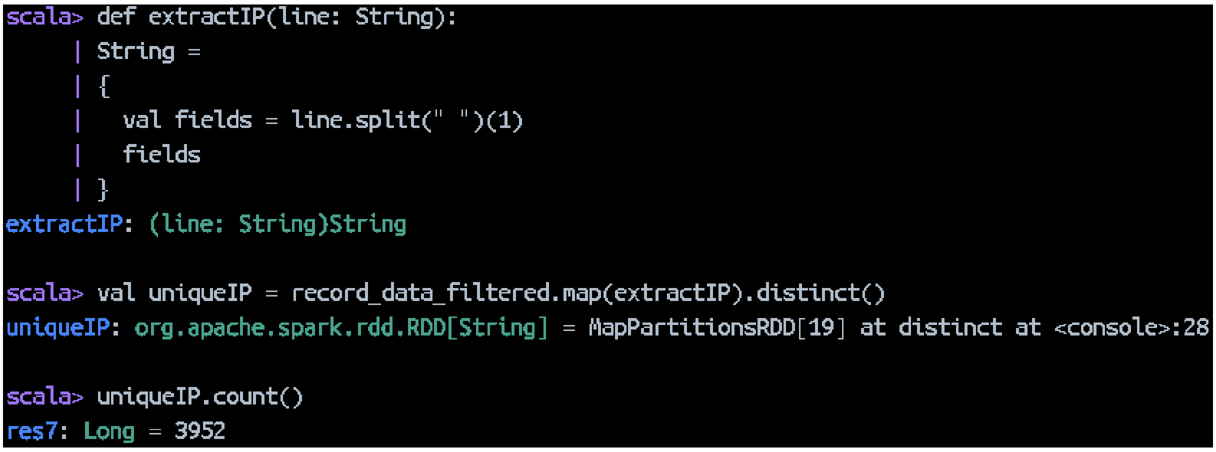
- Takes a log record as input.
- Splits the record by spaces and retrieves the second field (index 1), which is the IP address.
- Returns the extracted IP address.

```
def extractIP(line: String):
String =
{
  val fields = line.split(" ")(1)
  fields
}
val uniqueIP = record_data_filtered.map(extractIP).distinct()
uniqueIP.count()
```

Next, a new RDD `uniqueIP` is created:

- Applies the `extractIP` function to each line of the filtered data.
- Uses the `distinct()` transformation to retain only unique IP addresses.

The program output is shown below:



```
scala> def extractIP(line: String):
     | String =
     | {
     |   val fields = line.split(" ")(1)
     |   fields
     | }
extractIP: (line: String)String

scala> val uniqueIP = record_data_filtered.map(extractIP).distinct()
uniqueIP: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[19] at distinct at <console>:28

scala> uniqueIP.count()
res7: Long = 3952
```

Figure 8: Unique IPs

1.3.3 Mission 3: Build an RDD, which contains the map of IPs and their additional information

The function `log_enhanced` is defined to:

- Take a log record as input.
- Extract relevant information and enrich it with additional data from the broadcasted IP map.
- Produce an enriched record including IP, additional information, city, latency, content name, and content size.

```scala
val dataIP = sc.textFile("hdfs://namenode:9000/logs/IPDict.csv")
dataIP.count()
```

```
scala> val dataIP = sc.textFile("hdfs://namenode:9000/logs/IPDict.csv")
dataIP: org.apache.spark.rdd.RDD[String] = hdfs://namenode:9000/logs/IPDict.csv MapPartitionsRDD[21] at textFile at <console>:27

scala> dataIP.count()
res8: Long = 4849
```

Figure 9: New RDD

Now, we will first calculate the Distinct ISP. Using the following code:

```scala
val mapIP = dataIP.map(line => {
  val fields = line.split(",")
  (fields(0), (fields(1), fields(2), fields(3)))
}).collectAsMap()
val broadcastIP = sc.broadcast(mapIP)

def log_enhanced(line: String):
(String, (String, String, String), String, Double, String, Long) =
{
  val fields = line.split(" ")
  val ip = fields(1)
  val info_addition = broadcastIP.value.getOrElse(ip, ("Unknown", "Unknown",
  "Unknown"))
  val latency = fields(0).toDouble
  val city = info_addition._2
  val contentSize = fields(fields.length - 1).toLong
  (ip, info_addition, city, latency, fields(4), contentSize)
}
val log_record_enhanced = record_data_filtered.map(log_enhanced)
```

The output of this process is displayed in the following image:

```
scala> val mapIP = dataIP.map(line => {
     |   val fields = line.split(",")
     |   (fields(0), (fields(1), fields(2), fields(3)))
     | }).collectAsMap()
mapIP: scala.collection.Map[String,(String, String, String)] = Map(14.163.247.204 -> (Vietnam,Dien Bien Phu,Vietnam Posts and Telecommunications
 Group), 58.187.162.9 -> (Vietnam,Hanoi,FPT Telecom Company), 14.170.198.211 -> (Vietnam,Thanh Hoa,Vietnam Posts and Telecommunications Group),
1.55.195.66 -> (Vietnam,Ho Chi Minh City,FPT Telecom Company), 171.247.242.166 -> (Vietnam,Tra Tan,Viettel Group), 14.235.189.145 -> (Vietnam,Ha
noi,Vietnam Posts and Telecommunications Group), 123.18.105.227 -> (Vietnam,Vinh,Vietnam Posts and Telecommunications Group), 58.187.67.212 -> (
Vietnam,Vinh Yen,FPT Telecom Company), 66.249.82.103 -> (Australia,Sydney,Google LLC), 42.119.51.226 -> (Vietnam,Da Nang,FPT Telecom Company), 1
35.23.231.184 -> (Canada,Toronto,TekSavvy Solutions...

scala> val broadcastIP = sc.broadcast(mapIP)
broadcastIP: org.apache.spark.broadcast.Broadcast[scala.collection.Map[String,(String, String, String)]] = Broadcast(17)
scala> def log_enhanced(line: String):
     | (String, (String, String, String), String, Double, String, Long) =
     | {
     |   val fields = line.split(" ")
     |   val ip = fields(1)
     |   val info_addition = broadcastIP.value.getOrElse(ip, ("Unknown", "Unknown", "Unknown"))
     |   val latency = fields(0).toDouble
     |   val city = info_addition._2
     |   val contentSize = fields(fields.length - 1).toLong
     |   (ip, info_addition, city, latency, fields(4), contentSize)
     | }
log_enhanced: (line: String)(String, (String, String, String), String, Double, String, Long)

scala> val log_record_enhanced = record_data_filtered.map(log_enhanced)
log_record_enhanced: org.apache.spark.rdd.RDD[(String, (String, String, String), String, Double, String, Long)] = MapPartitionsRDD[23] at map at
 <console>:28
```

Figure 10: Initialize

Using this enriched data, we calculate the number of records from Ho Chi Minh City and other metrics like total traffic from Hanoi. Additionally, Spark MLlib is used to compute latency statistics, including mean, maximum, and minimum latencies, as well as the median. The relevant program outputs are displayed below:

```scala
val uniqueISP = log_record_enhanced.map{case (_, (_, _, isp), _, _, _, _) => isp}.distinct().collect()
println(s"Number of unique ISPs: ${uniqueISP.length}")
```

After running the above code, we will receive:

```
scala> val uniqueISP = log_record_enhanced.map{case (_, (_, _, isp), _, _, _, _) => isp}.distinct().collect()
uniqueISP: Array[String] = Array(PJSC Rostelecom, China Unicom, Chunghwa Telecom, Verizon Business, ChinaNet, Telia Company AB, Telenor Norge AS
, CD-Telematika a.s., RCS & RDS SA, Taiwan Mobile Co. Ltd., SFR SA, M1 Limited, Vodafone Telekomunikasyon A.S., Vodafone Australia Pty Limited,
Kazan Broad-band access pools, CIK Telecom INC, Taipei Taiwan, Saigon Postel Corporation, SWAN a.s., M247 Ltd, Robert Bosch GmbH, "Facebook, Tel
stra Internet, TOKAI Communications Corporation, Orange S.A., Vietnamobile Telecommunications Joint Stock Company, Vietnam Posts and Telecommuni
cations Group, Deutsche Telekom AG, TELEFiA?A1/2NICA BRASIL S.A, CTM, Vodafone NRW GmbH, Kddi Corporation, Spark New Zealand Trading Ltd, The Cl
oud Networks Limited, Tele Columbus AG, "Telia Liet...

scala> println(s"Number of unique ISPs: ${uniqueISP.length}")
Number of unique ISPs: 125
```

Figure 11: Unique ISPs

```scala
val HCM_record = log_record_enhanced.filter {case (_, (_, city, _), _, _, _, _) => city == "Ho Chi Minh City"}
println(s"Number of records in Ho Chi Minh City: ${HCM_record.count()}")
```

```
scala> val HCM_record = log_record_enhanced.filter {case (_, (_, city, _), _, _, _, _) => city == "Ho Chi Minh City"}
HCM_record: org.apache.spark.rdd.RDD[(String, (String, String, String), String, Double, String, Long)] = MapPartitionsRDD[28] at filter at <cons
ole>:27

scala> println(s"Number of records in Ho Chi Minh City: ${HCM_record.count()}")
Number of records in Ho Chi Minh City: 217212
```

Figure 12: Records of Ho Chi Minh

```scala
val HaNoi_traffic = log_record_enhanced.filter {case (_, (_, city, _), _, _, _, _) => city == "Hanoi"}
  .map {case (_, _, _, _, _, contentSize) => contentSize}
  .reduce(_ + _)
println(s"Total traffic in Hanoi: ${HaNoi_traffic}")
```

The code first filters the log records to include only those originating from Ha Noi, then extracts the content size for each record, and finally calculates the total traffic by summing up the content sizes. Here is the output of the program:

```
scala> val HaNoi_traffic = log_record_enhanced.filter {case (_, (_, city, _), _, _, _, _) => city == "Hanoi"}
HaNoi_traffic: org.apache.spark.rdd.RDD[(String, (String, String, String), String, Double, String, Long)] = MapPartitionsRDD[29] at filter at <c
onsole>:27

scala>   .map {case (_, _, _, _, _, contentSize) => contentSize}
res11: org.apache.spark.rdd.RDD[Long] = MapPartitionsRDD[30] at map at <console>:28

scala>   .reduce(_ + _)
res12: Long = 204245300091

scala> println(s"Total traffic in Hanoi: ${HaNoi_traffic}")
Total traffic in Hanoi: MapPartitionsRDD[29] at filter at <console>:27
```

Figure 13: Records of Ha Noi

The result is: **204245300091**

Finally, we will then calculate the latencies's values. Using Spark MLlib's statistical functions.

```scala
val latency_data = log_record_enhanced.map {case (_, _, _, latency, _, _) => latency}
val latency_vector = latency_data.map(latency => Vectors.dense(latency))
val latency_stats: MultivariateStatisticalSummary = Statistics.colStats(latency_vector)
println(s"Mean Latency: ${latency_stats.mean(0)}")
println(s"Maximum Latency: ${latency_stats.max(0)}")
println(s"Minimum Latency: ${latency_stats.min(0)}")
```

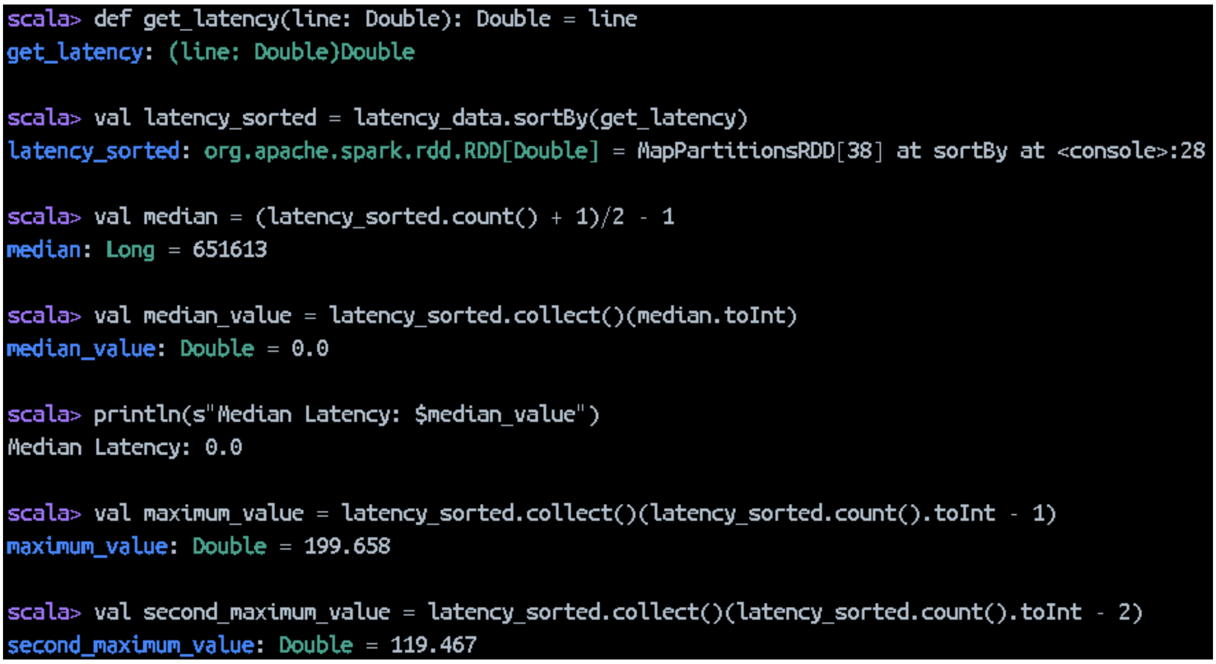
This code calculates and prints the mean, maximum, and minimum latencies from the provided data using Spark MLlib's statistical functions.

```
scala> val latency_data = log_record_enhanced.map {case (_, _, _, latency, _, _) => latency}
latency_data: org.apache.spark.rdd.RDD[Double] = MapPartitionsRDD[31] at map at <console>:27

scala> val latency_vector = latency_data.map(latency => Vectors.dense(latency))
latency_vector: org.apache.spark.rdd.RDD[org.apache.spark.mllib.linalg.Vector] = MapPartitionsRDD[32] at map at <console>:27

scala> val latency_stats: MultivariateStatisticalSummary = Statistics.colStats(latency_vector)
latency_stats: org.apache.spark.mllib.stat.MultivariateStatisticalSummary = org.apache.spark.mllib.stat.MultivariateOnlineSummarizer@7336c374

scala> println(s"Mean Latency: ${latency_stats.mean(0)}")
Mean Latency: 0.15163189835692353

scala> println(s"Maximum Latency: ${latency_stats.max(0)}")
Maximum Latency: 199.658

scala> println(s"Minimum Latency: ${latency_stats.min(0)}")
Minimum Latency: 0.0
```

Figure 14: Mean, Maximum and Minimum latencies

For the median, the RDD is sorted using `sortBy()`, and the middle element is retrieved to calculate the value. The results are as follows:

```scala
def get_latency(line: Double): Double = line
val latency_sorted = latency_data.sortBy(get_latency)
val median = (latency_sorted.count() + 1)/2 - 1
val median_value = latency_sorted.collect()(median.toInt)
println(s"Median Latency: $median_value")

val maximum_value = latency_sorted.collect()(latency_sorted.count().toInt - 1)
val second_maximum_value = latency_sorted.collect()(latency_sorted.count().toInt - 2)
```



```
scala> def get_latency(line: Double): Double = line
get_latency: (line: Double)Double

scala> val latency_sorted = latency_data.sortBy(get_latency)
latency_sorted: org.apache.spark.rdd.RDD[Double] = MapPartitionsRDD[38] at sortBy at <console>:28

scala> val median = (latency_sorted.count() + 1)/2 - 1
median: Long = 651613

scala> val median_value = latency_sorted.collect()(median.toInt)
median_value: Double = 0.0

scala> println(s"Median Latency: $median_value")
Median Latency: 0.0

scala> val maximum_value = latency_sorted.collect()(latency_sorted.count().toInt - 1)
maximum_value: Double = 199.658

scala> val second_maximum_value = latency_sorted.collect()(latency_sorted.count().toInt - 2)
second_maximum_value: Double = 119.467
```

Figure 15: Calculate median

1.4 Task 3: Analysis

1.4.1 Mission 1: Print out the number of HIT, MISS, and HIT1 requests

```scala
def HIT_classified(line: String):
String =
{
  val content_name = line.split(" ")(2)
  if (content_name.endsWith("HIT"))
  "HIT"
  else if (content_name.endsWith("HIT1"))
  "HIT1"
  else
  "MISS"
}

val data_HIT = record_data_filtered.map(line => (HIT_classified(line), 1))
val counts_HIT = data_HIT.reduceByKey(_ + _)
```

```scala

val count_HIT = counts_HIT.collect().find {case (hitStatus, _) => hitStatus == "HIT"}.map(_._2).getOrElse(0)
val count_missed = counts_HIT.collect().find {case (hitStatus, _) => hitStatus == "MISS"}.map(_._2).getOrElse(0)
val count_HIT1 = counts_HIT.collect().find {case (hitStatus, _) => hitStatus == "HIT1"}.map(_._2).getOrElse(0)
println(s"HIT: $count_HIT records")
println(s"HIT1: $count_HIT1 records")
println(s"MISS: $count_missed records")
```

First, we will create a function to classify. Here is how the classifyHIT function works: • This function takes a log line as input, splits it by spaces, and extracts the third element. • It then checks if the extracted content name ends with "HIT," "HIT1," or anything else and returns the corresponding status. Next is to map and count. hitData maps each log line to a tuple containing the classification status

("HIT," "HIT1," or "MISS") and the value 1. hitCounts then counts the occurrences of each classification status using reduceByKey.

Here is the output of the program:

```
scala> val count_HIT = counts_HIT.collect().find {case (hitStatus, _) => hitStatus == "HIT"}.map(_._2).getOrElse(0)
count_HIT: Int = 1137824

scala> val count_missed = counts_HIT.collect().find {case (hitStatus, _) => hitStatus == "MISS"}.map(_._2).getOrElse(0)
count_missed: Int = 113609

scala> val count_HIT1 = counts_HIT.collect().find {case (hitStatus, _) => hitStatus == "HIT1"}.map(_._2).getOrElse(0)
count_HIT1: Int = 51794

scala> println(s"HIT: $count_HIT records")
HIT: 1137824 records

scala> println(s"HIT1: $count_HIT1 records")
HIT1: 51794 records

scala> println(s"MISS: $count_missed records")
MISS: 113609 records
```

Figure 16: HIT count values

1.4.2 Mission 2: Compute the HitRate of the system

By having the above results, it is easy to calculate the HitRate with the give formula.

```scala
val HIT_Rate = (count_HIT + count_HIT1).toDouble / (count_HIT + count_HIT1 + count_missed).toDouble
println(s"HIT Rate: $HIT_Rate")
```

The result will be:

```
scala> val HIT_Rate = (count_HIT + count_HIT1).toDouble / (count_HIT + count_HIT1 + count_missed).toDouble
HIT_Rate: Double = 0.9128248570663438

scala> println(s"HIT Rate: $HIT_Rate")
HIT Rate: 0.9128248570663438
```

Figure 17: Calculate HIT Rate

1.4.3 Mission 3: Identify the ISP with the Highest Hit Rate

The goal of this task is to determine the Internet Service Providers (ISPs) with the highest Hit Rate. The implementation is shown below:

```
def HIT_log_enhanced(line: String):
(String, (String, String, String), String, Double, String, Long) =
{
  val fields = line.split(" ")
  val ip = fields(1)
  val info_addition = broadcastIP.value.getOrElse(ip, ("Unknown", "Unknown", "Unknown"))
  val latency = fields(0).toDouble
  val city = info_addition._2
  val contentSize = fields(fields.length - 1).toLong
  (ip, info_addition, city, latency, HIT_classified(line), contentSize)
}
val HIT_log_record_enhanced = record_data_filtered.map(HIT_log_enhanced)
```

This function is similar to the one in Task 2 but includes the "HIT status" in the return value, which is essential for this mission.

Next, the ISP-HIT Status is mapped and counted. This involves creating a mapping between ISPs and their HIT statuses and calculating the count of each status:

```
val map_ISP_HIT_status = HIT_log_record_enhanced.map {case (_, (_, _, isp), _, _, hitStatus, _) => (isp, hitStatus)}
val count_ISP_HIT_status = map_ISP_HIT_status.groupByKey().mapValues(status => (status.count(_ == "HIT"), status.count(_ == "HIT1"), status.count(_ == "MISS")))
println("ISP and HIT Status Counts:")
count_ISP_HIT_status.collect().foreach {case (isp, (count_HIT, count_HIT1, count_missed)) => println(s"$isp: HIT = $count_HIT, HIT1 = $count_HIT1, MISS = $count_missed")}
```

The following step calculates the Hit Rate for each ISP. The ‘ispHitRate‘ is derived by dividing the total "HIT" and "HIT1" counts by the total number of requests:

```
val ISP_HIT_Rate = count_ISP_HIT_status.mapValues {case (count_HIT, count_HIT1, count_missed) =>
  val total_request = count_HIT + count_HIT1 + count_missed
  val HIT_Rate = (count_HIT + count_HIT1).toDouble / total_request.toDouble.toDouble
  (HIT_Rate)
}
println("ISP HIT Rate:")
ISP_HIT_Rate.collect().foreach {case (isp, HIT_Rate) => println(s"$isp: ISP HIT Rate = $HIT_Rate")}
```

Finally, we identify the ISP(s) with the highest Hit Rate. The process involves finding the maximum Hit Rate and listing all ISPs that achieve this rate:

```
val max_HIT_Rate = ISP_HIT_Rate.values.max
val ISP_max_HIT_Rate = ISP_HIT_Rate.filter {case (_, HIT_Rate) => HIT_Rate == max_HIT_Rate}.keys
val ISP_max_HIT_RateArr = ISP_max_HIT_Rate.collect()
println(s"The ISPs with the maximum HIT Rate ($max_HIT_Rate) are: ${ISP_max_HIT_RateArr.mkString("\n")}")
```

This step outputs the ISP(s) with the highest Hit Rate, ensuring that multiple ISPs with the same maximum Hit Rate are included in the results.