CS328 · Distributed and Cloud Computing

Assignment 3

刘乐奇 / 12011327

May 1, 2023

1 Introduction

In this assignment, I analyzed the parking data from **parking_data_sz.csv** dataset using **Apache Spark** version 3.3.2 and **Scala** version 2.12.15 in **local mode**, and output the results as specified by the five requirements to five separate CSV files.
Each line of the data describes a car parked in a parking lot and contains five values:

Name	Description	Example
in_time	Time when the car goes into the parking lot	"2018-09-01 10:10:00"
out_time	Time when the car goes out of the parking lot	"2018-09-01 12:00:00"
berthage	Unique id of the parking lot	"201091"
section	Section to which the parking lot belongs	" 荔园路 (蛇口西段)"
admin_region	District of the parking lot	" 南山区"

Followings are requirements

1. Output the total number of parking lots in each section. The output file should have two columns, with the headers being **section** and **count**.
2. Output all unique ids (berthages), associated with their section. The output file should have two columns, with the headers being **berthage** and **section**.
3. Output for each section: the average parking time of a car in that section. The output file should have two columns, with the headers being **section** and **avg_parking_time**. The average parking time should be counted in seconds as an integer.
4. Output the average parking time for each parking lot, sorted in descending order. The output file should have two columns, with the headers being **berthage** and **avg_parking_time**. The average parking time should be counted in seconds as an integer.
5. Output for each section: the total number of parking lots in use ("in use" means there is at least one car in that parking lot) and the percentage out of the total number of parking lots in that section, in a 30-minute interval (e.g. during 09:00:00-09:29:59). The output file should have five columns, with the headers being **start_time**, **end_time**, **section**, **count** and **percentage**. The percentage value should be rounded to two decimal places. The data format of **start_time** and **end_time** should be "HH:MM:SS" , e.g. 12:00:00.

2 Environment

- Ubuntu-22.04 on WSL2
- Apache Spark 3.3.2
- Scala 2.12.15
- Python 3.9.1

3 Submit Files

- CS328_Assignment_3.pdf: The report
- Main.scala: The source code
- plot_figure.py: Code for plotting 3 figures on requirement 5
- r1.csv, r2.csv, r3.csv, r4.csv, r5.csv: Results of 5 requirement

4 Workflow

4.1 Start HDFS and Upload Dataset

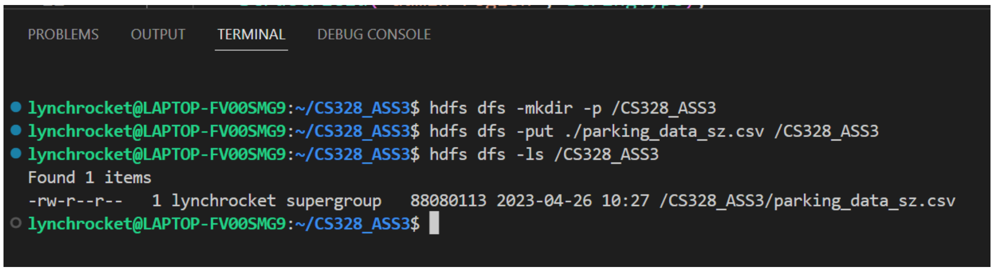
Although we can load the dataset on local file system by appointing the file path with leading **file://**, it is much more convenient and quick to load the dataset from HDFS.
First, we need to start HDFS and upload the **parking_data_sz.csv** dataset onto it. Some necessary configuration must be set before when deploy Hadoop environment. Since I had configured the environment variables, I could just input in the shell like below to start all services of HDFS.

```
$ start-all.sh
```

Then we can create a new directory and upload the dataset file on it. (The path is valid only on my machine)

```
$ hdfs dfs -mkdir -p /CS328_ASS3
$ hdfs dfs -put ./parking_data_sz.csv /CS328_ASS3
```

We can see the dataset file is on HDFS.



```
lynchrocket@LAPTOP-FV00SMG9:~/CS328_ASS3$ hdfs dfs -mkdir -p /CS328_ASS3
lynchrocket@LAPTOP-FV00SMG9:~/CS328_ASS3$ hdfs dfs -put ./parking_data_sz.csv /CS328_ASS3
lynchrocket@LAPTOP-FV00SMG9:~/CS328_ASS3$ hdfs dfs -ls /CS328_ASS3
Found 1 items
-rw-r--r--   1 lynchrocket supergroup   88080113 2023-04-26 10:27 /CS328_ASS3/parking_data_sz.csv
lynchrocket@LAPTOP-FV00SMG9:~/CS328_ASS3$
```

Figure 1: Upload Dataset

Since I had uploaded the file and have not formatted the namenode from then on, the file is stored persistently on HDFS. If you want to format the namenode (which is actually the first thing to do when first use HDFS), you can input

```
$ hdfs namenode –format
```

4.2 Writing Script and Run

The script is also submitted, named **Main.scala**, where the whole code can be obtained. Followings are only some significant manipulations.

4.2.1 dataset loading and coarse process

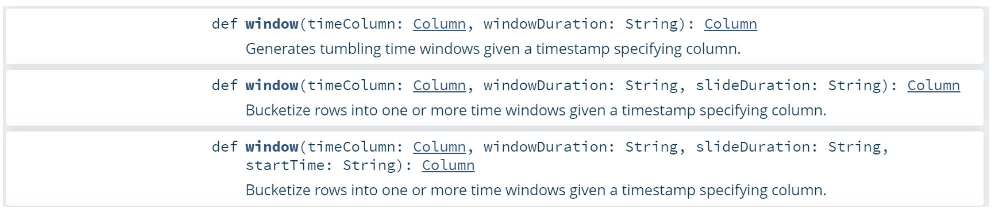
There are some invalid data in the dataset, such as null value row, downfallen out time, and so on. I chose to delete all the invalid data.
When loading dataset, we can configure the schema, i.e., the data types of each column. And we can filter the null value row by setting **option("mode", "DROPMALFORMED")**, filter downfallen out time by **filter("out_time > in_time")**. For conveniece, a new column named **duration** can be computed ahead.

```
val fileName = "hdfs://localhost:9000/CS328_ASS3/parking_data_sz.csv"
val schema = StructType(Seq(
        StructField("out_time", TimestampType),
        StructField("admin_region", StringType),
        StructField("in_time", TimestampType),
        StructField("berthage", StringType),
        StructField("section", StringType)
    ))
// read in dataset
val textFile = spark.read
                    .option("header", true)
                    .option("dateFormat", "yyyy-MM-dd HH:mm:ss")
                    .option("mode", "DROPMALFORMED") // filter null value row
                    .schema(schema) // configure data type
                    .csv(fileName)

// coarse process
val dateDF = textFile
                . filter ("out_time > in_time") // filter downfallen out time
                .withColumn("duration", col("out_time").cast("long") - col("in_time").cast("long
                    ")) // compute duration for
                    convenience
```

4.2.2 requirement 5

It is the most complex requirement. **Time slide window** in Spark is practical. There are three overloading methods, I used the one with 2 parameters. It can measure on the time stamp column with specified duration time window. Here I set window duration with 30 minutes as requested and time column with a new column that produced by **UDF** and **explode** function.



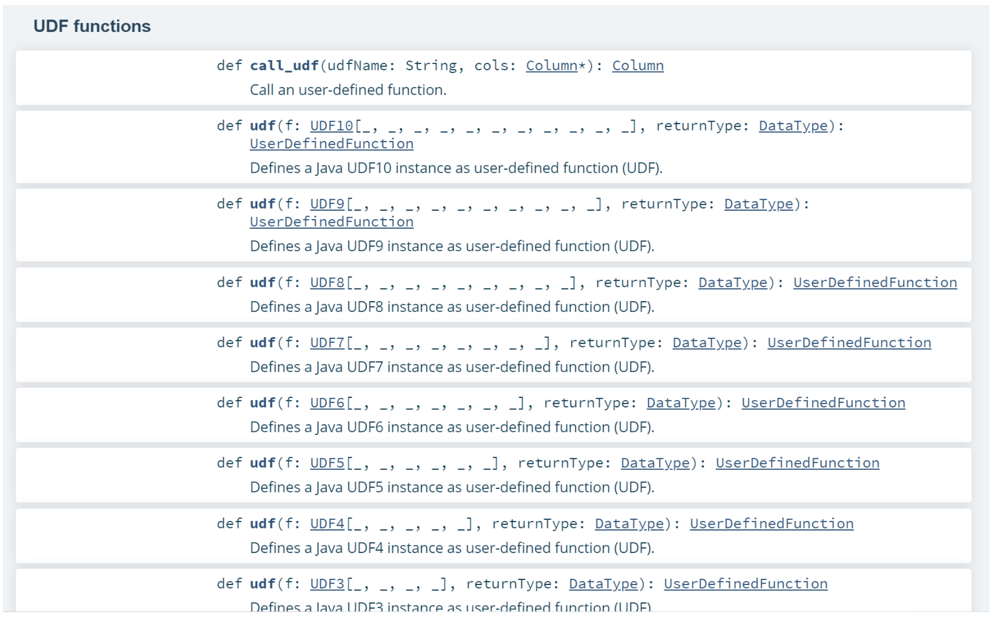
```
def window(timeColumn: Column, windowDuration: String): Column
    Generates tumbling time windows given a timestamp specifying column.

def window(timeColumn: Column, windowDuration: String, slideDuration: String): Column
    Bucketize rows into one or more time windows given a timestamp specifying column.

def window(timeColumn: Column, windowDuration: String, slideDuration: String,
    startTime: String): Column
    Bucketize rows into one or more time windows given a timestamp specifying column.
```

Figure 2: Time slide window function

Besides, I used **UDF**, i.e., user define function, to compute the time interval list that a berthage is occupied. Then **explode** function can divide a row into multiple rows depending on how many time intervals that a berthage is occupied by a record, which makes it easy to count in time window.
However, there is a drawback that after **explode**, the size of the dataframe is much larger than before like explosion by a bomb. If the memory is limited, it may be forbidden.
Due to the **time slide window** in Spark, the start of the time slide window depends on the record with the earliest column **time**, i.e., if a record with **in_time** as **2018-10-29 07:03:55**, the start of time window will be **2018-10-29 07:00:00** instead of **2018-10-29 00:00:00**.


UDF functions

```
def call_udf(udfName: String, cols: Column*): Column
    Call an user-defined function.
def udf(f: UDF10[_, _, _, _, _, _, _, _, _, _, _], returnType: DataType):
    UserDefinedFunction
    Defines a Java UDF10 instance as user-defined function (UDF).
def udf(f: UDF9[_, _, _, _, _, _, _, _, _, _], returnType: DataType):
    UserDefinedFunction
    Defines a Java UDF9 instance as user-defined function (UDF).
def udf(f: UDF8[_, _, _, _, _, _, _, _, _], returnType: DataType): UserDefinedFunction
    Defines a Java UDF8 instance as user-defined function (UDF).
def udf(f: UDF7[_, _, _, _, _, _, _, _], returnType: DataType): UserDefinedFunction
    Defines a Java UDF7 instance as user-defined function (UDF).
def udf(f: UDF6[_, _, _, _, _, _, _], returnType: DataType): UserDefinedFunction
    Defines a Java UDF6 instance as user-defined function (UDF).
def udf(f: UDF5[_, _, _, _, _, _], returnType: DataType): UserDefinedFunction
    Defines a Java UDF5 instance as user-defined function (UDF).
def udf(f: UDF4[_, _, _, _, _], returnType: DataType): UserDefinedFunction
    Defines a Java UDF4 instance as user-defined function (UDF).
def udf(f: UDF3[_, _, _, _], returnType: DataType): UserDefinedFunction
    Defines a Java UDF3 instance as user-defined function (UDF).
```

Figure 3: UDF

```
val timeList = udf((inTime: java.sql.Timestamp, duration: Int) => {
        val localDateTime = inTime.toLocalDateTime()
        val steps = duration / 1800

        var timeList = ""
        for(idx <- 0 until steps) {
            timeList += localDateTime.plus(idx * 30, ChronoUnit.MINUTES).toString + ","
        }
        timeList += localDateTime.plus(steps * 30, ChronoUnit.MINUTES).toString
        timeList
    })
val slide30MinWindows = dateDF
                    .withColumn("time_list", timeList(col("in_time"), col("duration")))
                    .withColumn("time", explode(split(col("time_list"), ",")))
                    .groupBy(
                        col("section"),
                        window(col("time"), "30 minutes")
                    ).agg(countDistinct("berthage").alias("count"))
val busyPercentage = slide30MinWindows
                    .join(berthageCountOfEachSection.withColumnRenamed("count", "
                        section_berthage_count"), "section")
                    .withColumn("percentage", round(col("count")*100/col("
                        section_berthage_count"), 2))
                    .orderBy(asc("window.start"))
                    .selectExpr("date_format(window.start, 'yyyy-MM-dd HH:mm:ss') as
                        start_time", "date_format(window.end, 'yyyy-MM-dd HH:mm:ss
                        ') as end_time", "section", "count", "percentage")
```

4.2.3 save result

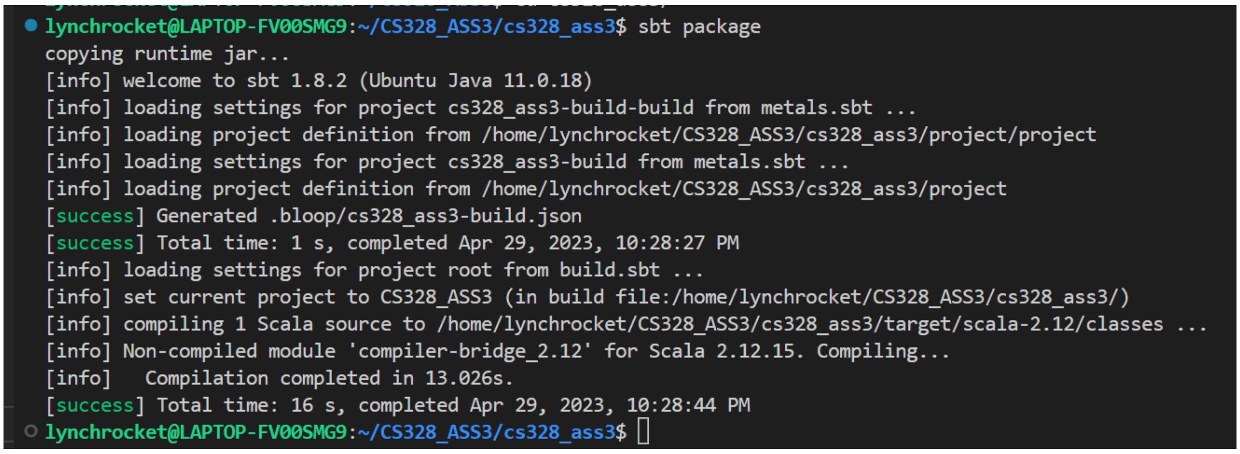
The inverse of loading dataset. Be careful that if using distributed cluster, **coalesce** function cannot be forgot. Because the data may be distributed on different clusters, **coalesce** can first gather all data to one machine then we can write data into file. For local mode, the data may be distributed on different processes, **coalesce** is also necessary.

```
val destDir = "hdfs://localhost:9000/CS328_ASS3/result/"
berthageCountOfEachSection.coalesce(1).write.option("header", true).csv(destDir + "r1")
```

4.2.4 package and submit

After finishing the script, we can use the following command, under the root of the project, to package the source code. (Be careful that you have downloaded **sbt** before)

```
$ sbt package
```



```
lynchrocket@LAPTOP-FV00SMG9:~/CS328_ASS3/cs328_ass3$ sbt package
copying runtime jar...
[info] welcome to sbt 1.8.2 (Ubuntu Java 11.0.18)
[info] loading settings for project cs328_ass3-build-build from metals.sbt ...
[info] loading project definition from /home/lynchrocket/CS328_ASS3/cs328_ass3/project/project
[info] loading settings for project cs328_ass3-build from metals.sbt ...
[info] loading project definition from /home/lynchrocket/CS328_ASS3/cs328_ass3/project
[success] Generated .bloop/cs328_ass3-build.json
[success] Total time: 1 s, completed Apr 29, 2023, 10:28:27 PM
[info] loading settings for project root from build.sbt ...
[info] set current project to CS328_ASS3 (in build file:/home/lynchrocket/CS328_ASS3/cs328_ass3/)
[info] compiling 1 Scala source to /home/lynchrocket/CS328_ASS3/cs328_ass3/target/scala-2.12/classes ...
[info] Non-compiled module 'compiler-bridge_2.12' for Scala 2.12.15. Compiling...
[info]   Compilation completed in 13.026s.
[success] Total time: 16 s, completed Apr 29, 2023, 10:28:44 PM
lynchrocket@LAPTOP-FV00SMG9:~/CS328_ASS3/cs328_ass3$
```

Figure 4: Package using sbt

The result *.jar* file can be found under the ./target directory, and we can submit it to spark

```
$ spark-submit --class parking ./cs328_ass3/target/scala-2.12/cs328_ass3_2.12-0.1.0-SNAPSHOT
    .jar
```

For convenience, we can execute the code line by line in the spark-shell so that we can watch the status through Spark Web UI. Notice that some lines of code are unnecessary to execute in the spark-shell such as the register of spark session since the spark-shell itself has registered it ahead.

4.3 Download Result

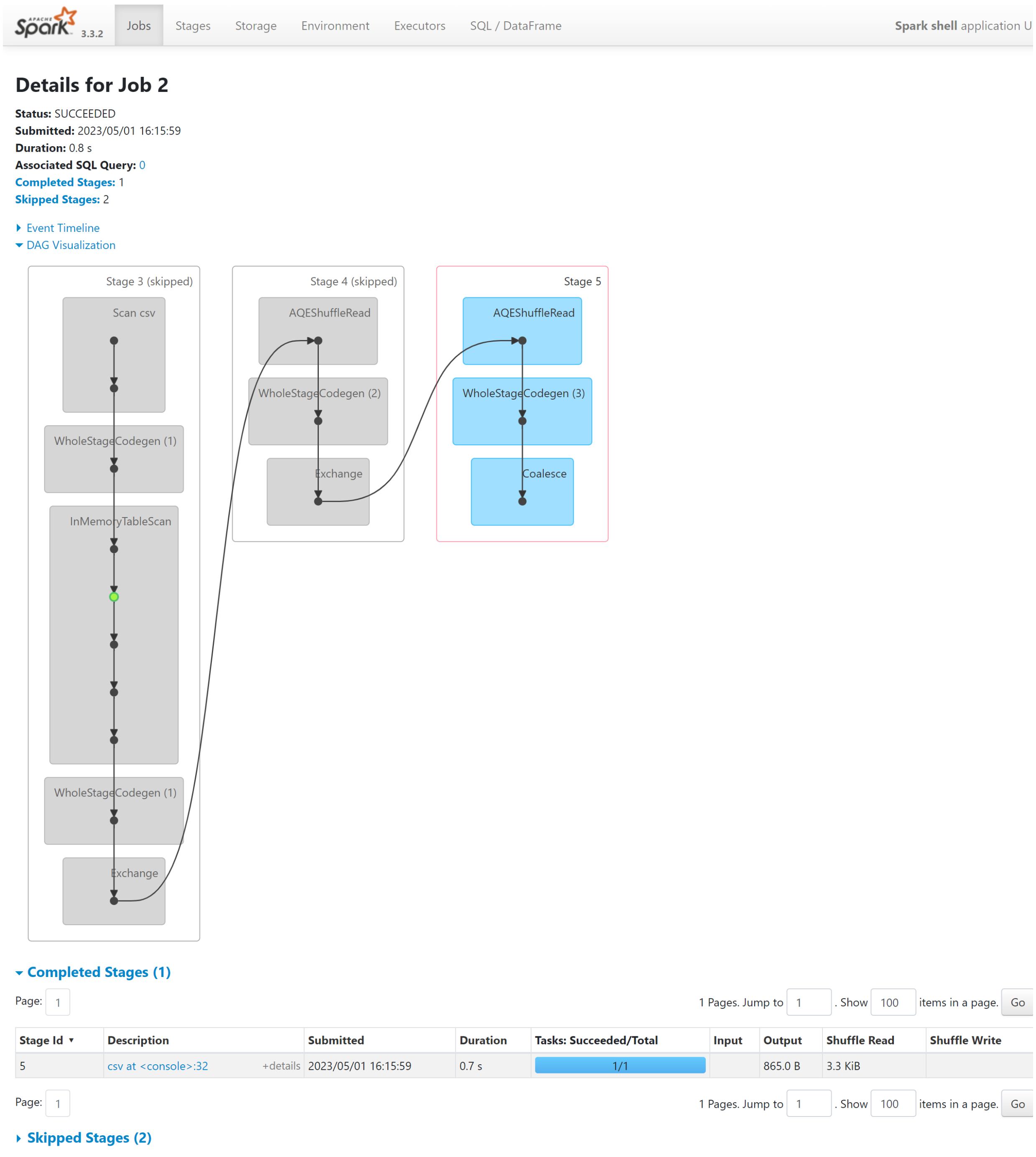
The running results are on HDFS. To obtain results from HDFS, we can execute the following command. (The path is valid only on my machine)

```
$ hdfs dfs -get /CS328_ASS3/result ~/CS328_ASS3
```

5 Results

During the execution of code in spark shell, we can access **http://localhost:4040/jobs/**, the Spark Web UI, to find some messages.

- Requirement 1



Details for Job 2

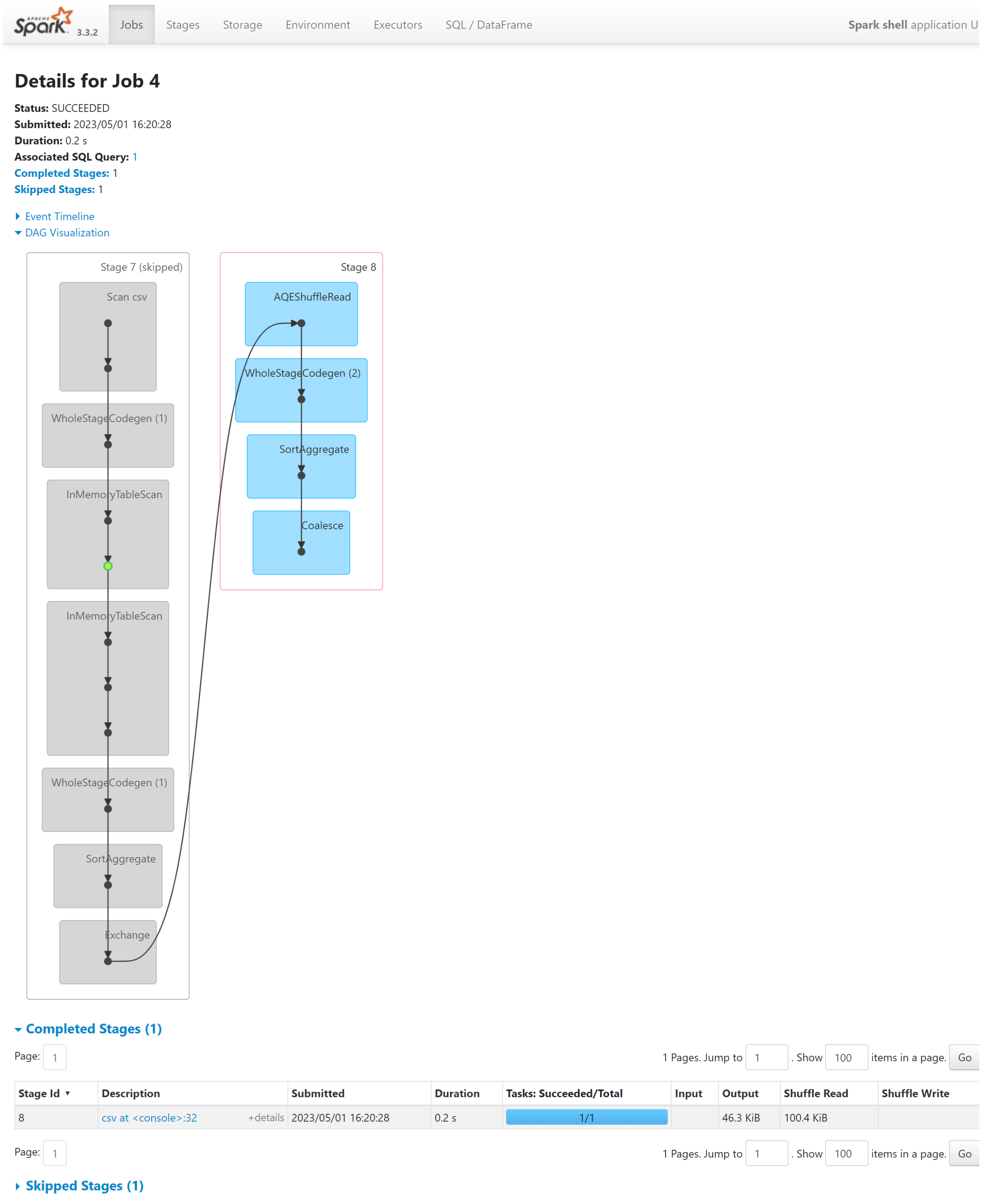
Status: SUCCEEDED
Submitted: 2023/05/01 16:15:59
Duration: 0.8 s
Associated SQL Query: 0
Completed Stages: 1
Skipped Stages: 2

Completed Stages (1)

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
5	csv at <console>:32 +details	2023/05/01 16:15:59	0.7 s	1/1		865.0 B	3.3 KiB	

Figure 5: Requirement 1

- Requirement 2



Details for Job 4

Status: SUCCEEDED
Submitted: 2023/05/01 16:20:28
Duration: 0.2 s
Associated SQL Query: 1
Completed Stages: 1
Skipped Stages: 1

Completed Stages (1)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

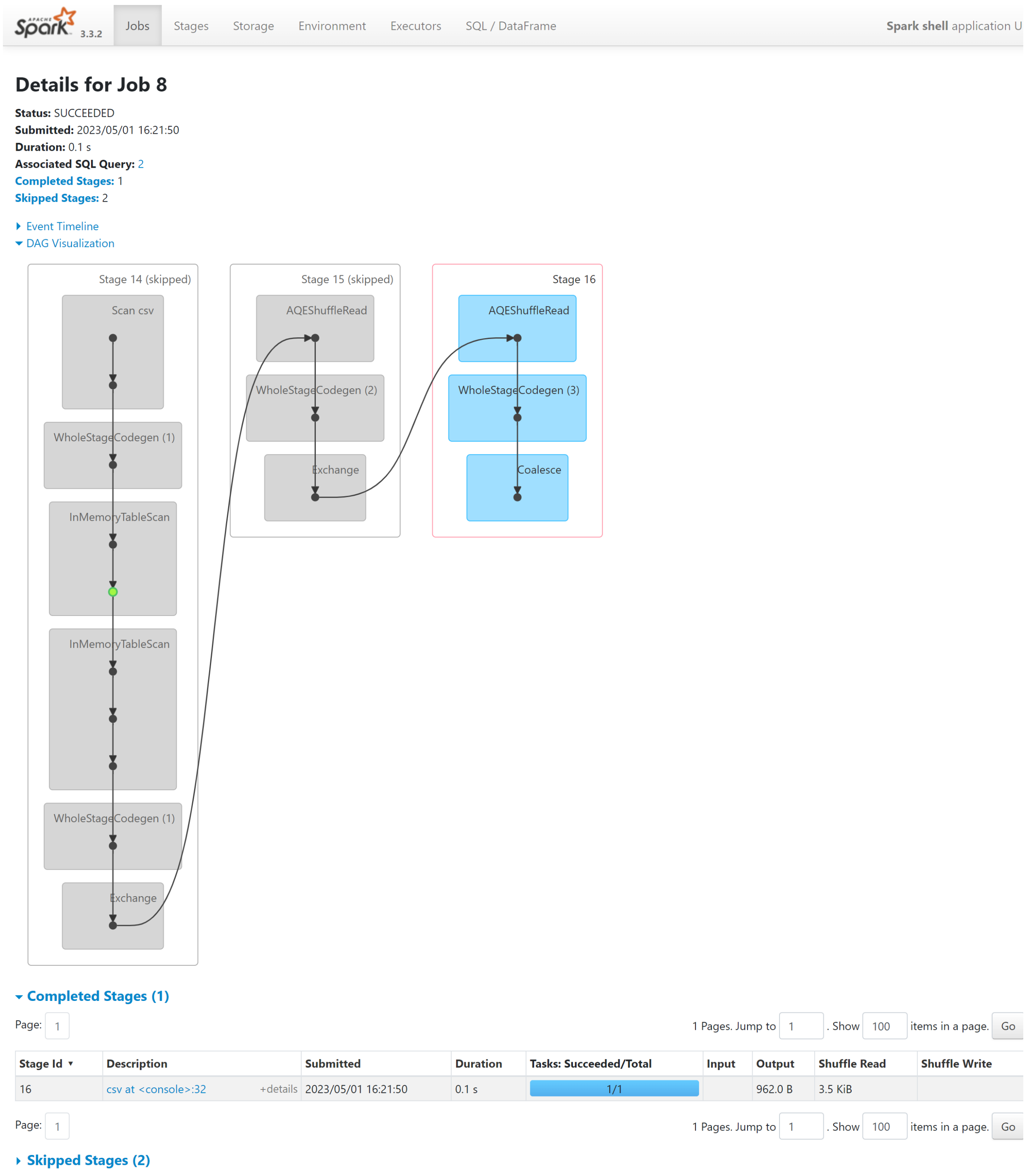
Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
8	csv at <console>:32 +details	2023/05/01 16:20:28	0.2 s	1/1		46.3 KiB	100.4 KiB	

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Skipped Stages (1)

Figure 6: Requirement 2

- Requirement 3



Details for Job 8

Status: SUCCEEDED
Submitted: 2023/05/01 16:21:50
Duration: 0.1 s
Associated SQL Query: 2
Completed Stages: 1
Skipped Stages: 2

▸ Event Timeline
▾ DAG Visualization

▾ Completed Stages (1)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

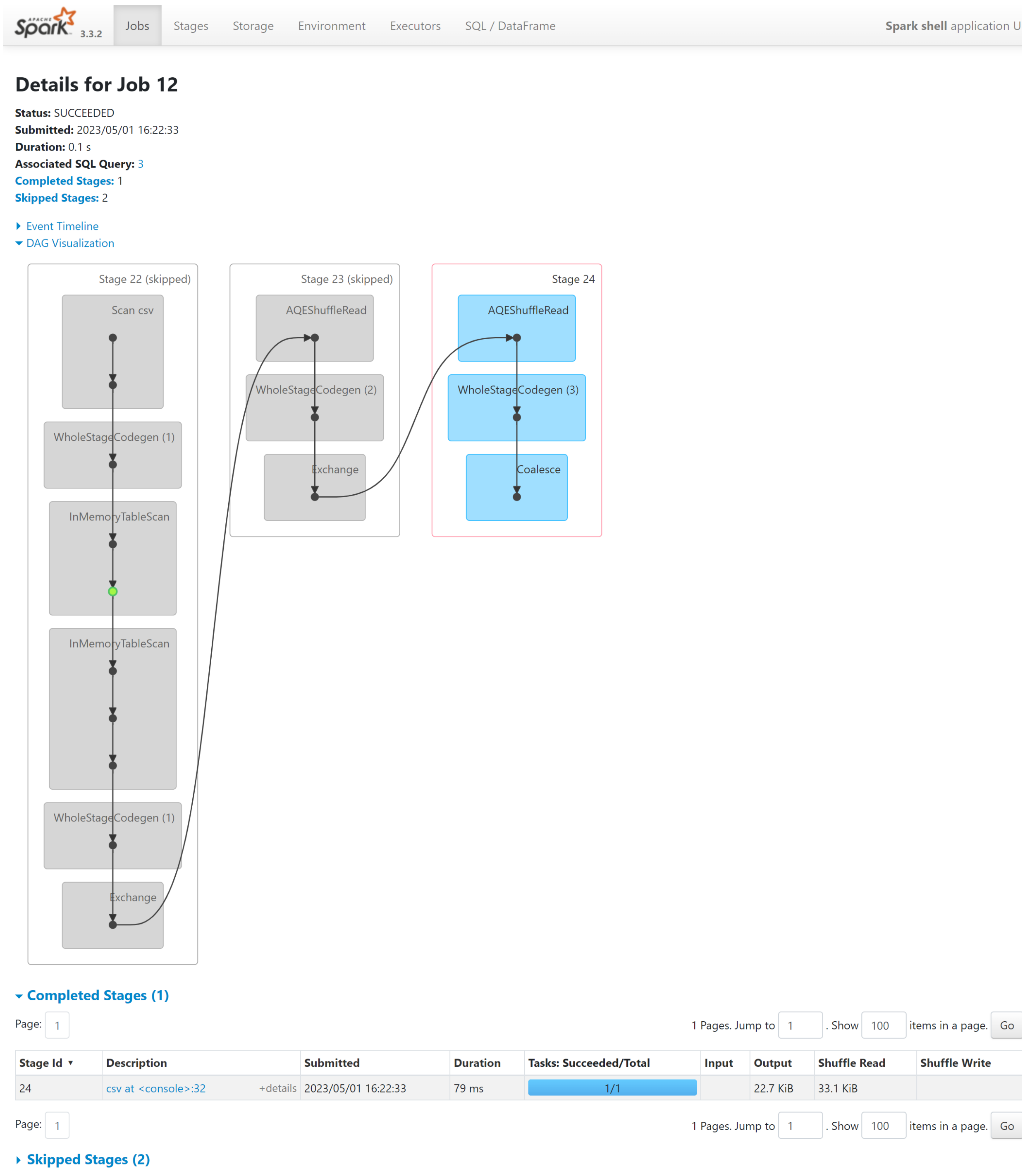
Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
16	csv at <console>:32 +details	2023/05/01 16:21:50	0.1 s	1/1		962.0 B	3.5 KiB	

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

▸ Skipped Stages (2)

Figure 7: Requirement 3

- Requirement 4



Details for Job 12

Status: SUCCEEDED
Submitted: 2023/05/01 16:22:33
Duration: 0.1 s
Associated SQL Query: 3
Completed Stages: 1
Skipped Stages: 2

Event Timeline
DAG Visualization

Completed Stages (1)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

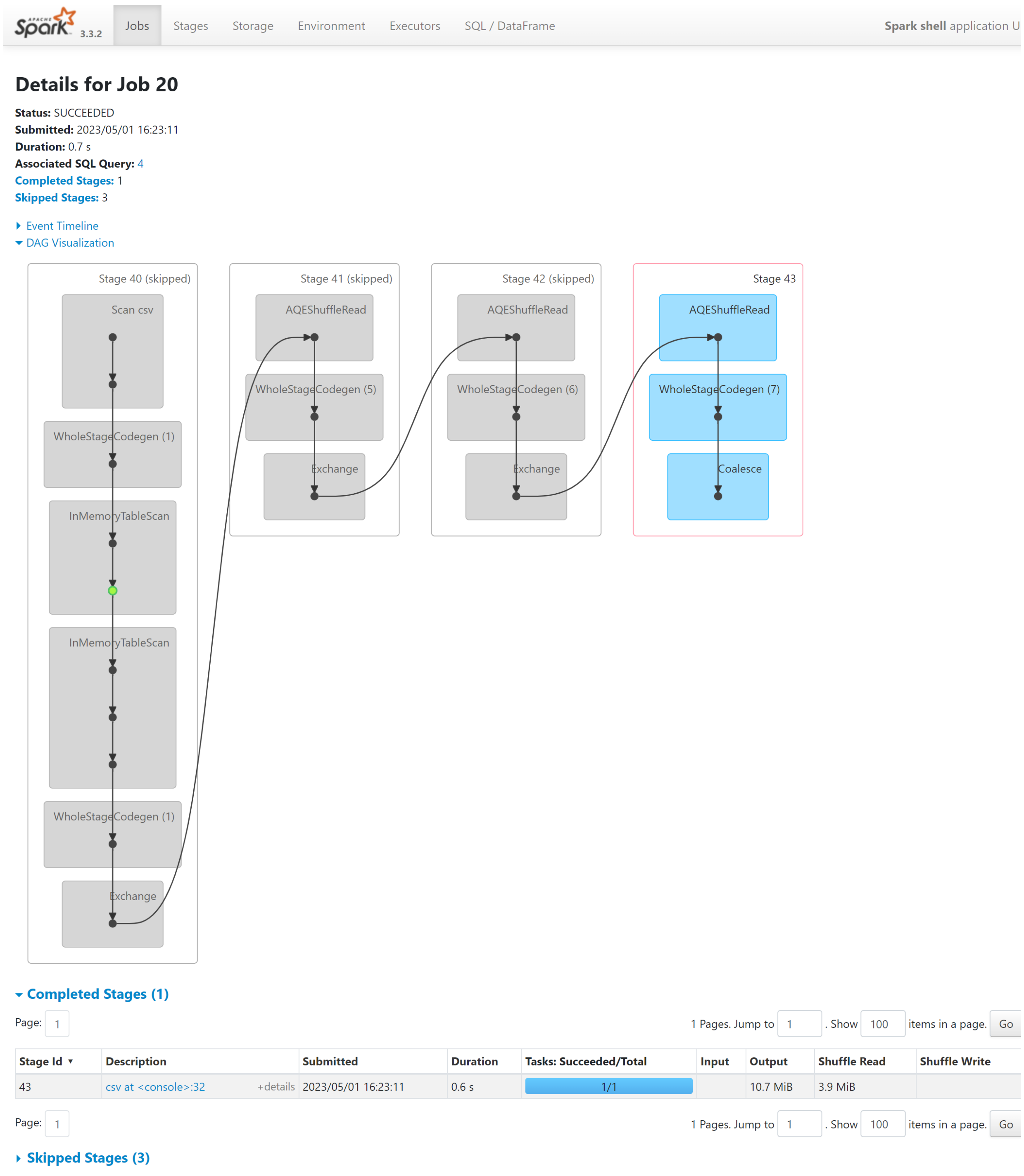
Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
24	csv at <console>:32 +details	2023/05/01 16:22:33	79 ms	1/1		22.7 KiB	33.1 KiB	

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Skipped Stages (2)

Figure 8: Requirement 4

- Requirement 5



Details for Job 20

Status: SUCCEEDED
Submitted: 2023/05/01 16:23:11
Duration: 0.7 s
Associated SQL Query: 4
Completed Stages: 1
Skipped Stages: 3

Completed Stages (1)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
43	csv at <console>:32 +details	2023/05/01 16:23:11	0.6 s	1/1		10.7 MiB	3.9 MiB	

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Skipped Stages (3)

Figure 9: Requirement 5

6 Three Figures on Requirement 5

I chose three sections named 学府路 (前海段), 工业九路, 华明路 and ploted the corresponding figures, shown as below. A bar indicates a 30 minutes time interval, i.e., 07:30:00 indicates the time interval of [07:30:00, 07:59:59].

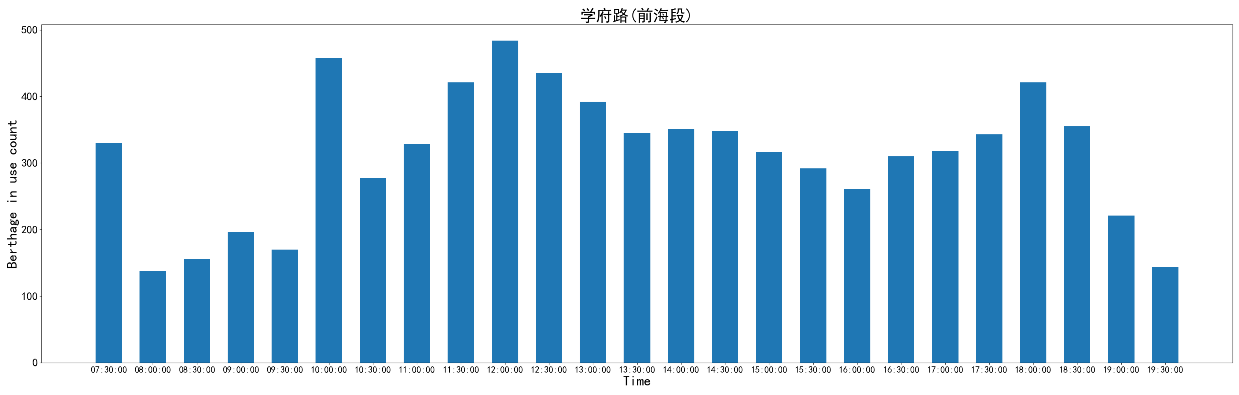


Figure 10: 学府路 (前海段)

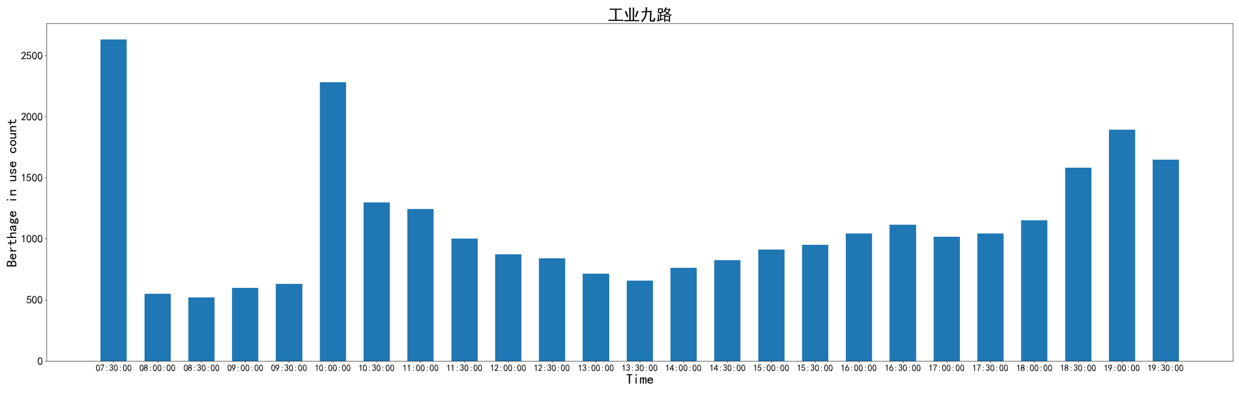


Figure 11: 工业九路

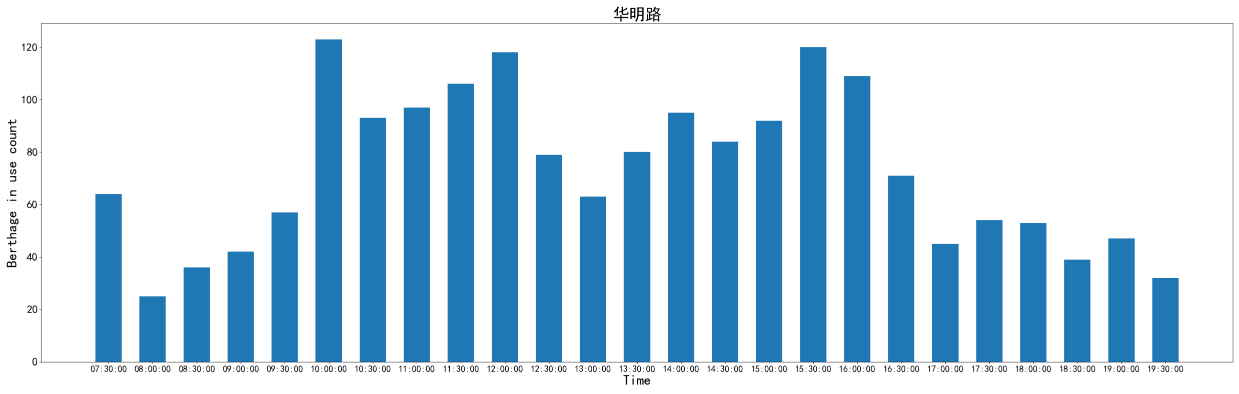
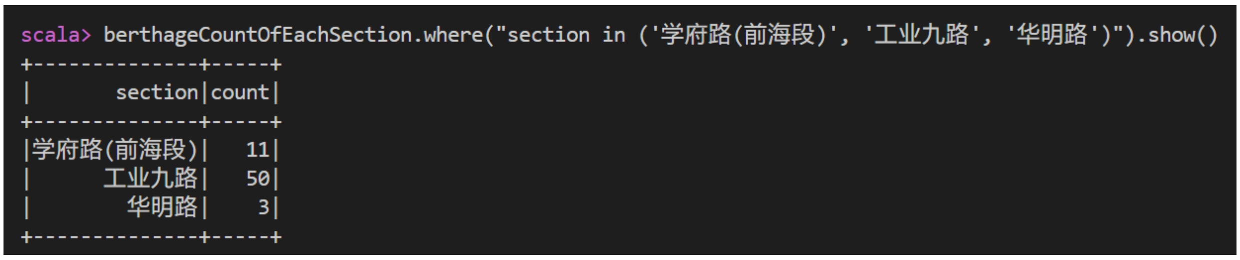


Figure 12: 华明路

There is no berthage occupied between 20:00:00 to 07:29:59 in next day. They share a similar distribution between 07:30:00 to 10:29:59 that peek at [07:30:00, 07:59:59] and [10:00:00, 10:29:59] and low plain in the middle. 工业九路 is the most bustling among the three section while 华明路 is the desolatest one. The reason may be that the number of berthages in 华明路 is much less than the others.



```
scala> berthageCountOfEachSection.where("section in ('学府路(前海段)', '工业九路', '华明路')").show()
+-------------+-----+
|      section|count|
+-------------+-----+
|学府路(前海段)|   11|
|      工业九路|   50|
|        华明路|    3|
+-------------+-----+
```

Figure 13: Berthage count of the three sections

7 Reference

[1] Hadoop Installation on Windows WSL 2 on Ubuntu 20.04 LTS (Single Node)
[2] WSL 搭建 Hadoop 与 Spark 环境
[3] Install Hadoop 3.3.0 on Linux
[4] Install Hadoop 3.3.0 on Windows 10 using WSL
[5] Apache Spark 3.0.1 Installation on Linux or WSL Guide
[6] Spark 安装和使用
[7] 基于泰坦尼克号生还数据的 Spark 数据处理分析
[8] [Spark 基础] UDAF (自定义聚合函数)
[9] spark 时间窗口操作
[10] 在 Spark DataFrame 中使用 Time Window
[11] Spark - 自定义函数（UDF、UDAF、UDTF）
[12] Spark 自定义函数 UDF UDAF
[13] Spark 3.4.0 ScalaDoc
[14] Class Timestamp
[15] Hadoop2.2.0 can't visit the web http://<ip>:8088