National Technical University of Athens

Interdisciplinary Master's Programme in

Data Science and Machine Learning

Large Scale Data Management

SEMESTER ASSIGNMENT

Apache Spark in Databases

Professors

Konstantinou Ioannis

Tsoumakos Dimitrios

Students

Tsopelas Konstantinos 03400198

Theofili Stavroula 03400166

Table of Contents

Introduction

The purpose of the present assignment is an educational-scale introduction to Apache Spark, perhaps the most efficient (to date) open-source unified analytics engine for large-scale data processing. We will make use of the Hadoop Distributed File System (HDFS) in order to store our data. HDFS employs a NameNode and DataNode (master-slave) architecture to implement a distributed file system that provides high-performance access to data across highly scalable Hadoop clusters. The dataset we are given is from Common Crawl and contains raw web page data and metadata, all stored in Amazon S3. To be more specific, it consists of 3 files: *warc.csv, wat.csv* and *wet.csv,* which contain a subset of the Common Crawl data. Additionally, we have the 2 files: *departmentsR.csv* and *employeesR.csv*, which contain mock data for the second part. We will use the first three .csv files in order to perform some queries using two different APIs of Apache Spark; the RDD API and the Dataframe API / Spark SQL. Then, we will compare the performance of these two APIs and come to conclusions based on the results. The remaining two .csv files of the dataset will be used to implement and evaluate the broadcast join algorithm (Map side join) and the repartition join algorithm (Reduce side join). Last but not least, we will also use *departmentsR.csv, employeesR.csv* in order to execute a given query with the query optimizer Catalyst enabled at first, and then disabled. In this way, we will familiarize ourselves with this optimization method and we will evaluate the difference in the execution time with optimizer Catalyst on and off.

Part A

In this part, the first thing we have to do is create a files directory on HDFS, upload all the given .csvs to this and then convert these files to parquet file format. To continue, using both RDDs and DataFrames we will write code to answer the queries Q1- Q5, measuring the execution time of each case. Finally, we will proceed with comparing and interpreting the results as well as the execution time of all the SQL and RDD queries.

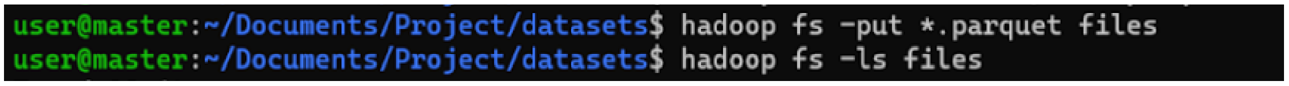
1. PARQUET FILE FORMAT

The first step of the experimental procedure consists of setting up the virtual machines provided by Okeanos (GRNET's cloud service) and installing Spark. The next step is to create a new directory named "files" in the HDFS to store all of our data. Before proceeding with the development of the code for the required queries, we need to transform the csv files to parquet. Parquet is a data file format provided by Apache Spark to increase efficiency and performance. It uses the record shredding and assembly algorithm which differs from simple record flattening. Parquet is optimized to work with complex data in bulk and features different ways for efficient data compression and encoding types. Moreover, the required space to load a .parquet file format in RAM is less than the .csv as well as the corresponding required space in the hard drive, and for that reason, it optimizes the I/O tasks by reducing the execution time.

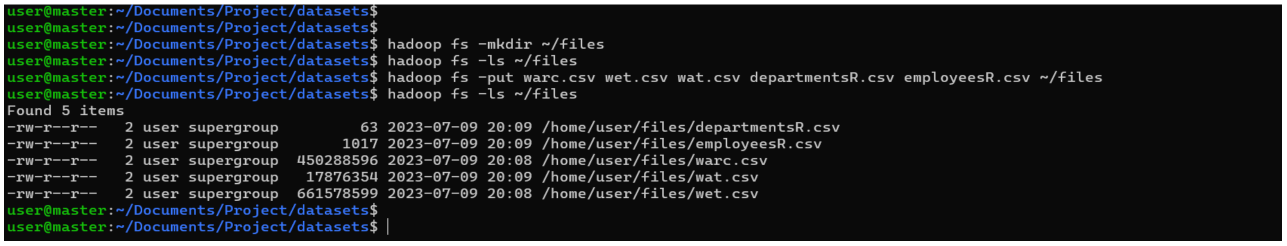
An indicative representation of the transformation-storage process is presented below:

```
import pandas as pd

# csv_names = ["warc", "wat", "wet", "departmentsR", "employeesR"]

col_names = {
    "warc": ["date", "warc_ID", "type", "content_length", "publicIP", "tar
getURL", "server", "htmldom"],
    "wat": ["warc_rec_id", "metadata_content_length", "targetURL"],
    "wet": ["warc_rec_id", "plaintext"],
    "departmentsR": ["id", "name"],
    "employeesR": ["id", "name", "dep_id"]
}

for name, cols in col_names.items():
    df = pd.read_csv("../datasets/{}.csv".format(name), names=cols, dtype=str)
    df.to_parquet("../datasets/{}.parquet".format(name))
```

Snippet 1: .parquet file transformation



```
user@master:~/Documents/Project/datasets$ hadoop fs -put *.parquet files
user@master:~/Documents/Project/datasets$ hadoop fs -ls files
```



```
user@master:~/Documents/Project/datasets$
user@master:~/Documents/Project/datasets$
user@master:~/Documents/Project/datasets$ hadoop fs -mkdir ~/files
user@master:~/Documents/Project/datasets$ hadoop fs -ls ~/files
user@master:~/Documents/Project/datasets$ hadoop fs -put warc.csv wet.csv wat.csv departmentsR.csv employeesR.csv ~/files
user@master:~/Documents/Project/datasets$ hadoop fs -ls ~/files
Found 5 items
-rw-r--r--   2 user supergroup         63 2023-07-09 20:09 /home/user/files/departmentsR.csv
-rw-r--r--   2 user supergroup       1017 2023-07-09 20:09 /home/user/files/employeesR.csv
-rw-r--r--   2 user supergroup  450288596 2023-07-09 20:08 /home/user/files/warc.csv
-rw-r--r--   2 user supergroup   17876354 2023-07-09 20:09 /home/user/files/wat.csv
-rw-r--r--   2 user supergroup  661578599 2023-07-09 20:08 /home/user/files/wet.csv
user@master:~/Documents/Project/datasets$
user@master:~/Documents/Project/datasets$ |
```

Figure 1.1: Upload to Hadoop

2. QUERIES EXECUTION AND RESULTS

In this section, we execute all 15 scripts that correspond to the 5 described queries and record the results and execution times for each implementation. The execution time is measured at the time of query evaluation in order to be as objective as possible, allowing comparison between implementations. In the following subsections, we first define the query, then record the results generated and finally present the execution times for the three implementations of each query in a bar plot. It's important to mention that in this section, only the code developed using the RDD API is presented. The code used for the execution of Spark SQL on the csv and parquet files, is shown in part A of Appendix, at the end of this report. Last but not least, the output we got using SQL on .parquet and on .csv files is the exact same.

Before developing the code for every RDD query, we need to define some helper code. First, we initialize a SparkSession and retrieve the SparkContext for further interaction with the Spark cluster. Furthermore, due to RDDs being able to process one line at a time and since the input is a string we need to split the columns based on ',' . The code developed for these tasks is shown in Snippet 2.

```python
from pyspark.sql import SparkSession
sc = SparkSession \
    .builder \
    .appName("Part 1 RDD query 1 execution") \
    .getOrCreate() \
    .sparkContext

def parse_row(row):
    row = row.split(",")
    fields = row[:7]
    final_field = ",".join(row[7:])
    fields.append(final_field)
    return fields
```

Snippet 2: Code developed for splitting

2.1. Query 1

For the time range between 2017-03-22 22:00 and 2017-03-22 23:00, find the most used servers in descending order

The Query No1 can be broken down to a series of map, filter and reduce calls. To begin with, we define a function that takes as input a row, parses the date from the first element of the row and checks if it falls within the specified range. Then, after reading the .csv file named "warc.csv" from HDFS, we perform a series of transformations on the RDD. Firstly, we map each row using the `parse_row` function, to break into the appropriate fields, and we filter out rows where the first element is an empty string. To continue, we filter rows based on the date range and we map each row to a tuple with the 7th element (server) as the key and a value of 1. The next step is to perform a reduce operation to calculate the count for each server and after that, we map each tuple to swap the positions of the server and count and we sort the tuples based on the keys (now the counts), which are in descending order. The 5 most used servers and the times they were used are shown in the table below. The code we developed in order to conduct this process is presented in Snippet 3.

Frequency	**Servers**
10431	Apache
9745	nginx
3185	Microsoft-IIS
2782	cloudflare-nginx
2086	GSE

```python
def date_in_range(row):
    date = datetime.strptime(row[0][2:], "%y-%m-%dT%H:%M:%SZ")
    mindate = datetime.strptime("17-03-22 22:00", "%y-%m-%d %H:%M")
    maxdate = datetime.strptime("17-03-22 23:00", "%y-%m-%d %H:%M")
    return date >= mindate and date <= maxdate

start_time = time.time()

result = sc.textFile("hdfs://master:9000/user/user/files/warc.csv") \
            .map(parse_row) \
            .filter(lambda row: row[0] != "") \
            .filter(date_in_range) \
            .map(lambda row: (row[6], 1)) \
            .reduceByKey(lambda x, y: x + y) \
            .map(lambda t: (t[1], t[0])) \
            .sortByKey(ascending=False)

end_time = time.time()
elapsed = end_time - start_time
```

Snippet 3: Map-Reduce code developed for the first query

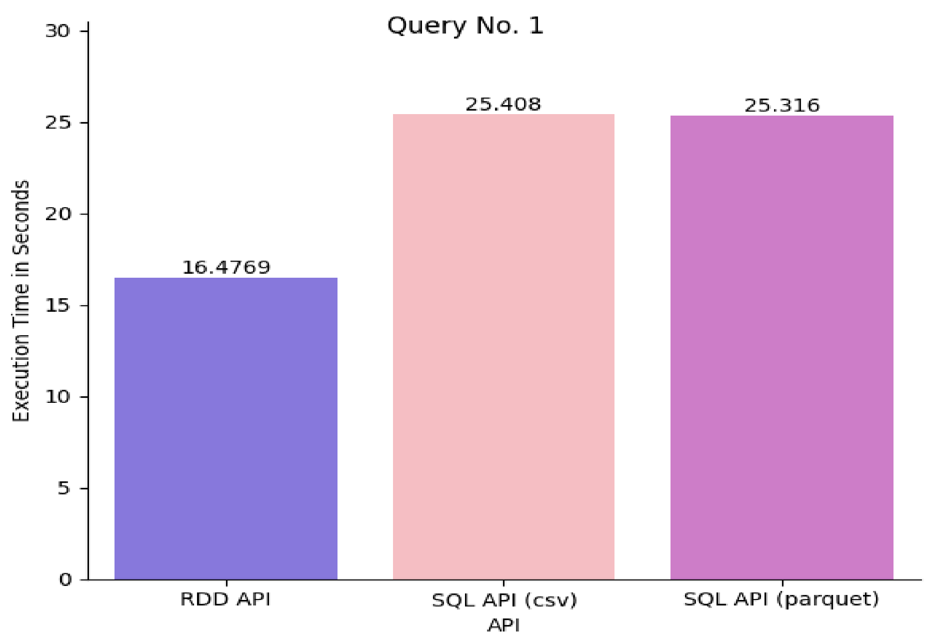


Figure 2.1: The execution times for Query No. 1

2.2. Query 2

For the warc target URL "http://1001.ru/articles/post/ai-da-tumin-443", find the content length of the metadata as well as the size of HTML dom (number of characters).

In order to answer to Query No. 2, we develop a code which reads the .csv files named "warc.csv" and "wat.csv" from HDFS, filters "warc" rows where the 6th element (target URL) is 'http://1001.ru/articles/post/ai-da-tumin-443' and then joins the two RDDs (`warc.join(wat)`) based on their keys (which we made sure were the warc IDs). Each resulting tuple is mapped to a tuple with the length of the 8th element of the first RDD value (HTML DOM) as the key, and the 1st element of the second RDD value (metadata content length) as the value. The code developed for the second query using the RDD API as well as the query's result are shown below.

Size of HTML Dom	Content Length of Metadata
205,735	1,136

```
warc = sc.textFile("hdfs://master:9000/user/user/files/warc.csv") \
            .map(parse_row) \
            .filter(lambda row: row[5] ==
"http://1001.ru/articles/post/ai-da-tumin-443" and row[2] ==
"request") \
            .map(lambda row: (row[1], row))

wat = sc.textFile("hdfs://master:9000/user/user/files/wat.csv") \
            .map(lambda row: row.split(",")) \
            .map(lambda row: (row[0], (row[1], row[2])))

result = warc.join(wat) \
        .map(lambda t: (len(t[1][0][7]), t[1][1][0]))
```

Snippet 4: Map-Reduce code developed for the second query

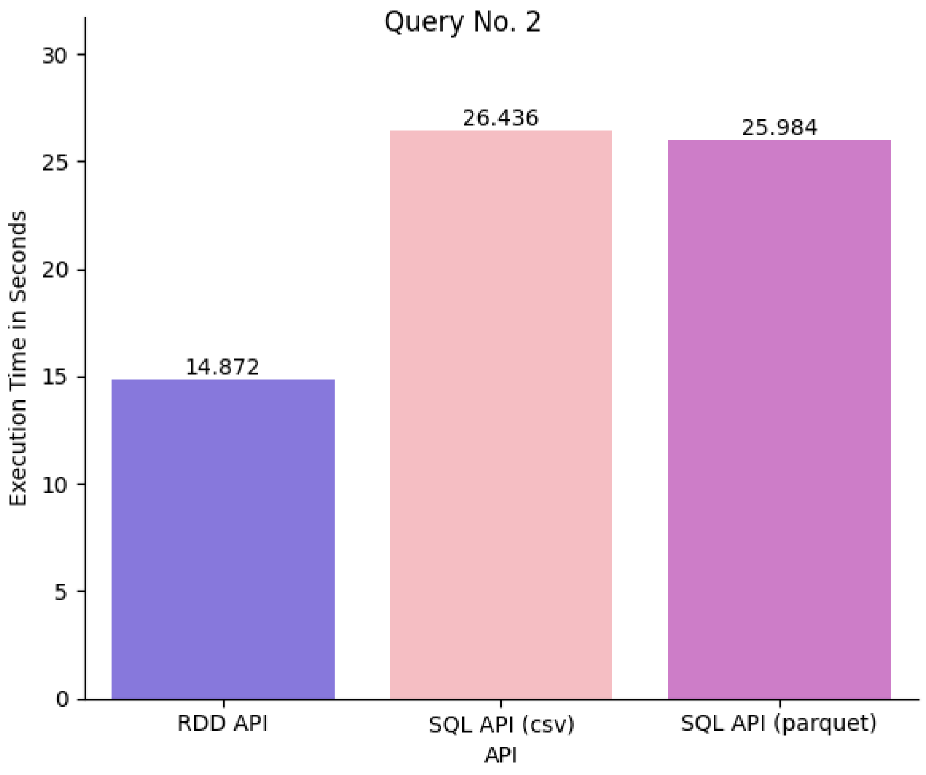


Figure 2.2: The execution times for Query No. 2

2.3. Query 3

What are the 5 warc record ids/target urls with the biggest content length that use as a server Apache in ascending order?

For Query No. 3, we develop a code which reads the .csv file named "warc.csv", maps each row using the `parse_row` function, filters rows where the 7th element (server) is "Apache" and proceeds with mapping each row to a tuple with the 4th element (content length) as an integer key, and also a tuple of the 2nd (record_id) and 6th (target_url) elements. Then, after sorting the RDD by the key in ascending order, our code retrieves the top 5 records. The query's results are presented below.

RecordID	Target URL	Content Length
<urn:uuid:acf04d8c-abe0-41db-9f5e-3644c9fe7f74>	http://k36yb-ctump.4rumer.com/t417-topic	1,253,837
<urn:uuid:5dc1b6d2-ffbf-400c-b7f0-0c31fe585094>	http://glmris.anl.gov/documents/docs/GLMRIS_Brandon_Rd_Scoping_Presentation.pdf	1,049,058
<urn:uuid:eb849dfe-a760-4e5d-b485-60112517037e>	http://de.tube8.com/amateur/adorable-brunette-masturbating-using-a-cucumber/3475551/	1,049,044
<urn:uuid:9d84df55-fa48-4979-9740-9fcb51ed0493>	http://hotwheels.wikia.com/wiki/File:Mattel-brand.svg.png	1,048,984
<urn:uuid:f4e3cb63-9268-4e94-a111-52bd65a6d198>	http://joeguide.com/summaries/	1,048,973

```
result = sc.textFile("hdfs://master:9000/user/user/files/warc.csv") \
            .map(parse_row) \
            .filter(lambda row: row[6] == "Apache") \
            .map(lambda row: (int(row[3]), (row[1], row[5]))) \
            .sortByKey(ascending=True)
```
Snippet 5: Map-Reduce code developed for the third query

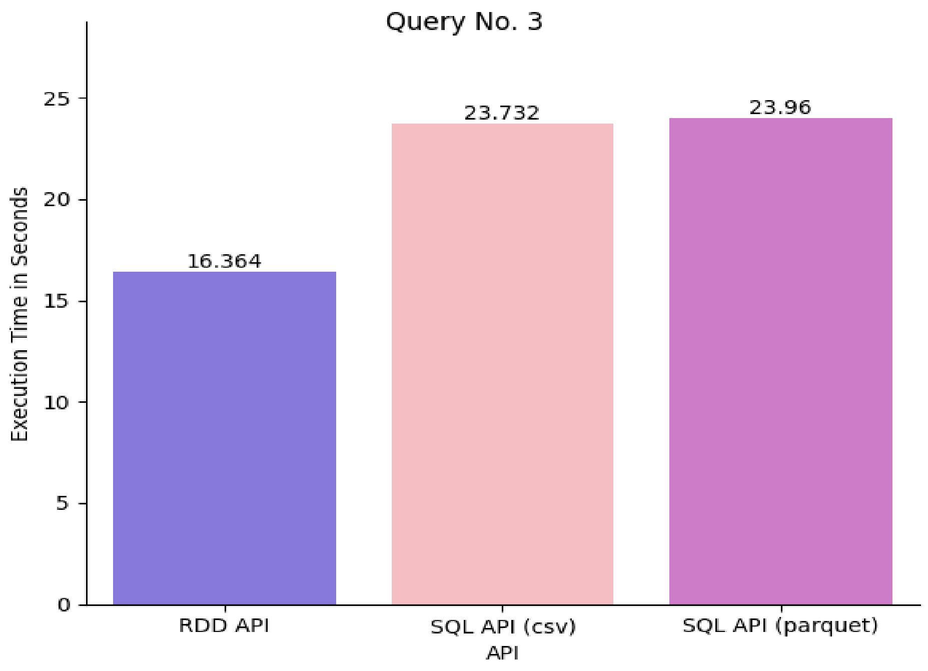


Figure 2.3: The execution times for Query No. 3

2.4. Query 4

For every server used, find the average content length of the warc records as well as the average content length of their metadata (print top 5 servers with the most content length size).

In order to conduct Query No. 4, our first step is to define a `drop_null` function that checks if any element in a given row is empty and to read both .csv files from HDFS. After proceeding "warc.csv", we map each row to a tuple with the 2nd element (record_id) as the key and the entire row as the value. We complete the same task for "wat.csv", mapping each row to a tuple with the 1st element (warc_record_id) as the key and the entire row as the value. To continue, we join the two RDDs (`warc.join(wat)`) based on their keys and we map each resulting tuple to another tuple with the 7th element (server) of the first RDD value as the key, and the 4th (content length) and 2nd (metadata_content_length) elements of the first and second RDD values as integers. Then, we group the RDD by the key and we map each group to a tuple with the key, the average of the 2nd (content length) elements in the group, and the average of the 3rd (metadata_content_length) elements in the group. Finally, after sorting the RDD by the average of the 2nd elements in descending order, we retrieve the first 5 records, which can be seen below. The main map-reduce code developed for the fourth query using the RDD API is shown in Snippet 6.

Server	Avg Cont. Length	Avg Metadata Cont. Length
Powered by Sloths	800,012	70,419
Unyil	800,12	35,190
172.17.99.36:443	527,518	1,104
bWFkaXNvbg==	404,891.333	6,395
mw1263.eqiad.wmnet	365,172.333	7,417.667

```
def drop_null(row):
    if any(not element for element in row):
        return False
    return True

warc = sc.textFile("hdfs://master:9000/user/user/files/warc.csv") \
            .map(parse_row) \
            .filter(drop_null) \
            .map(lambda row: (row[1], row))

wat = sc.textFile("hdfs://master:9000/user/user/files/wat.csv") \
            .map(lambda row: row.split(",")) \
            .map(lambda row: (row[0], row))

def avg(l):
    return sum(l) / len(l)

result = warc.join(wat) \
        .map(lambda t: (t[1][0][6], int(t[1][0][3]), int(t[1][1][1]))) \
        .groupBy(lambda t: t[0]) \
        .map(lambda t: (t[0], avg([y for x, y, z in t[1]]), avg([z for
x, y, z in t[1]]))) \
        .sortBy(lambda t: t[1], ascending=False)
```

Snippet 6: Map-Reduce code developed for the fourth query

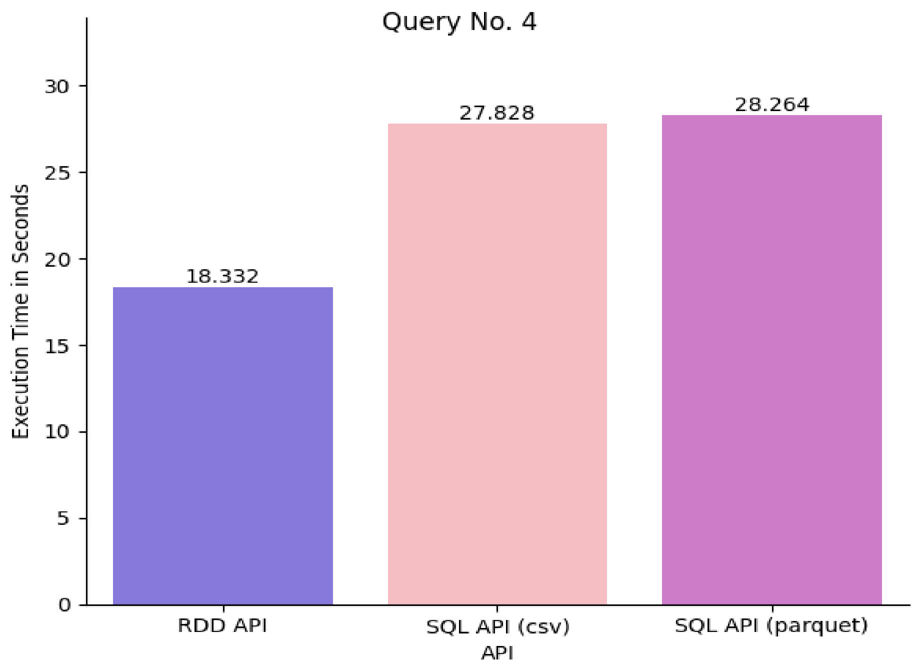


Figure 2.4: The execution times for Query No. 4

2.5. Query 5

Find the most popular target URL (i.e., the record target URL that can be found in another record's HTML DOM length size).

Finally, for Query No. 5, the first thing we have to do is define a function that checks if any field in the given record is empty. Moreover, it's necessary to define two more functions. The first one, html_to_urls, is used to extract URLs from an HTML string and the second one, main, serves as the entry point for the code. We continue by reading "warc.csv" from HDFS and filtering out records where any field is empty. Using flatMap we create pairs of (HTML, URL) by iterating over each record and applying html_to_urls function on the 8th field (html_dom). Then, with mapping each (URL, 1) pair and reducing by key, we can calculate the count of each URL and sort the RDD by this count in descending order. Our last step is to retrieve the top 10 results, knowing that at the top of our list is placed the most popular Target_URL. The code developed for the fifth query using the RDD API is shown in Snippet 7.

Target URL	Times Appeared
schema.org",\n "@type": "ImageObject",\n "contentUrl": "	1,010

```
def no_null_fields(rec):
    if any(not field for field in rec):
        return False
    return True

def html_to_urls(html):
    url_re =
re.compile(r'http[s]?://(?:[a-zA-Z]|[0-9]|[$-_@.&+]|[!*\\(\\),]|(?:%[0
-9a-fA-F][0-9a-fA-F]))+')
    return url_re.findall(html)

def main():
    warc = sc.textFile("hdfs://master:9000/user/user/files/warc.csv")
\
                .map(lambda x: x.split(",")) \
                .filter(no_null_fields)

    urls_in_html = warc.flatMap(lambda x: [(x[5], url) for url in
html_to_urls(x[7])])
    url_counts = urls_in_html.map(lambda x: (x[1],
1)).reduceByKey(lambda a, b: a + b)

    pprint(url_counts.sortBy(lambda x: -x[1]).take(10))
```

Snippet 7: Map-Reduce code developed for the fifth query

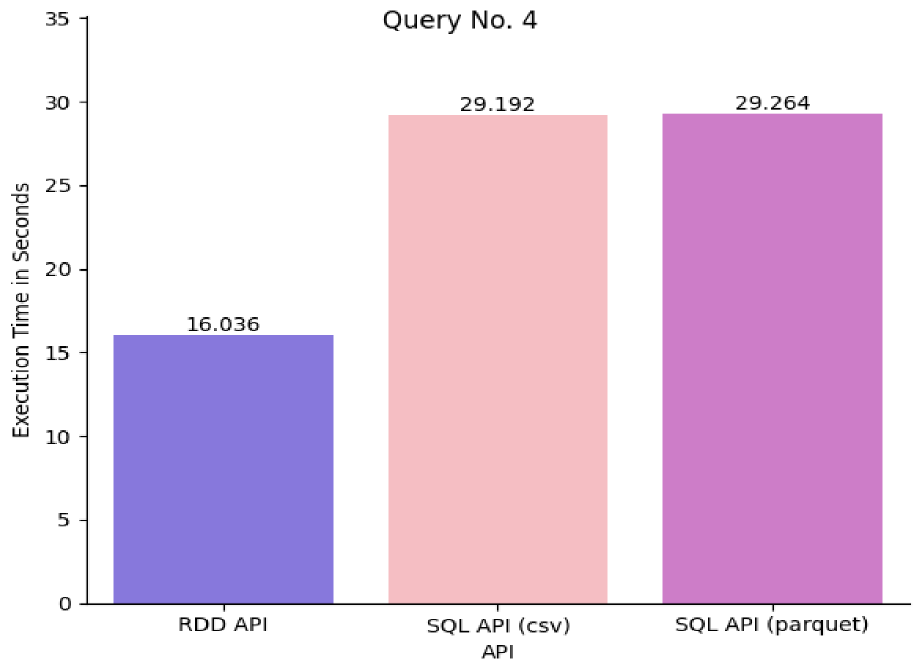


Figure 2.5: The execution times for Query No. 5

3. CONCLUSIONS

Based on all the queries' results of the previous section, it becomes evident that Spark's RDD API is by far superior, as the execution times for all queries using this API are considerably smaller compared to the ones where the DatafFrame API was used. To highlight this, we have grouped all the barplots shown in Figs. 2.1-2.5 in a single barplot shown in Fig. 3.1.

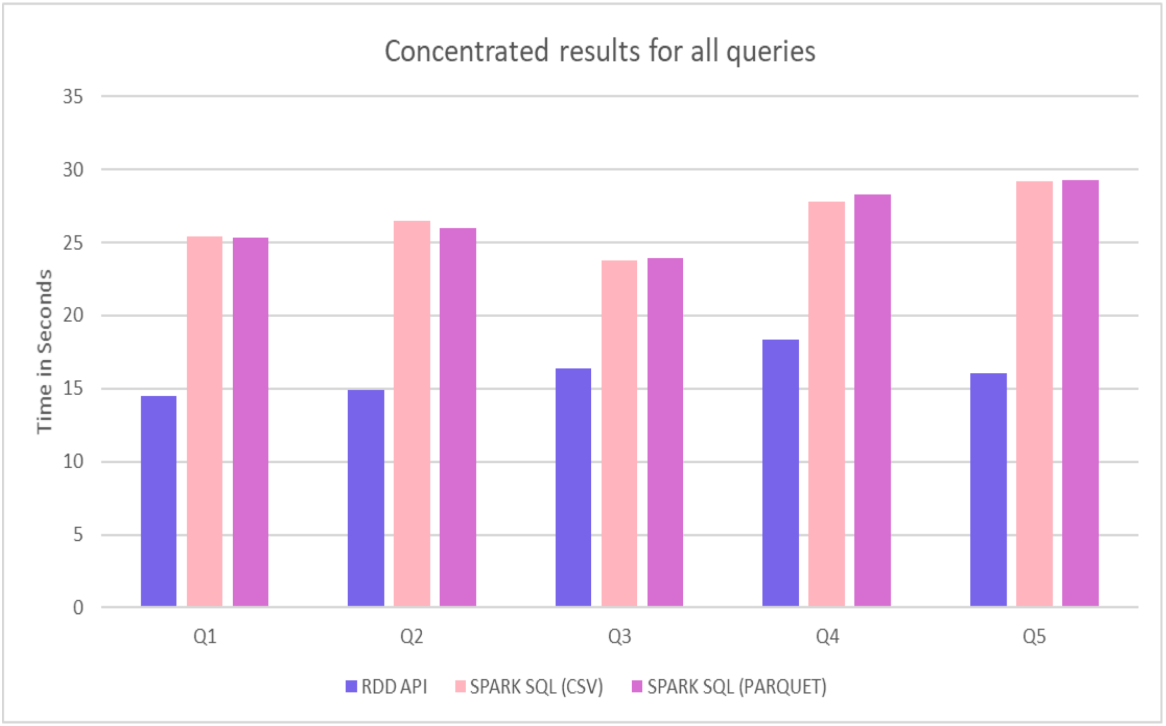


Figure 3.1: The execution times for Q1-Q5

RDD execution is not necessarily faster than DataFrame execution by default. In fact, DataFrames are designed to optimize and improve performance for many common use cases compared to RDDs. However, there can be several reasons why our RDD execution is faster. When dealing with smaller datasets or datasets with irregular or complex structures, RDDs might sometimes perform better than DataFrames. In our case, the dataset we are given is relatively small , so it probably helped the RDD achieve an optimized execution time. Moreover, in general, RDDs are more flexible and allow many custom computations, which can be beneficial in certain cases. So, it stands to reason to assume that some specific operations performed in our code may have helped RDDs to exhibit better performance characteristics compared to DataFrames.

Another significant observation based on the final barplot is that the same queries performed with the DataFrame API but on different file formats can't greatly impact the resulting execution time: by using .parquet files instead of .csv files, we didn’t manage to reduce the total run times, except from some cases where the reduction is so small that it’s not even worth mentioning. In accordance with this observation, we come to the conclusion that while CSV and Parquet formats can have differences in terms of storage, efficiency and I/O operations, Spark's DataFrame API and its default optimizations can minimize the performance gap between the two formats.

Part B

In this part, two files named *departmentsR.csv* and *employeesR.csv* will be used to implement and evaluate the broadcast join algorithm (Map side join) and the repartition join algorithm (Reduce side join). Secondly, we will execute a given query with the query optimizer Catalyst enabled at first, and then disabled. Based on the results that we will get, we will compare and comment on the performance and efficiency of this optimizer.

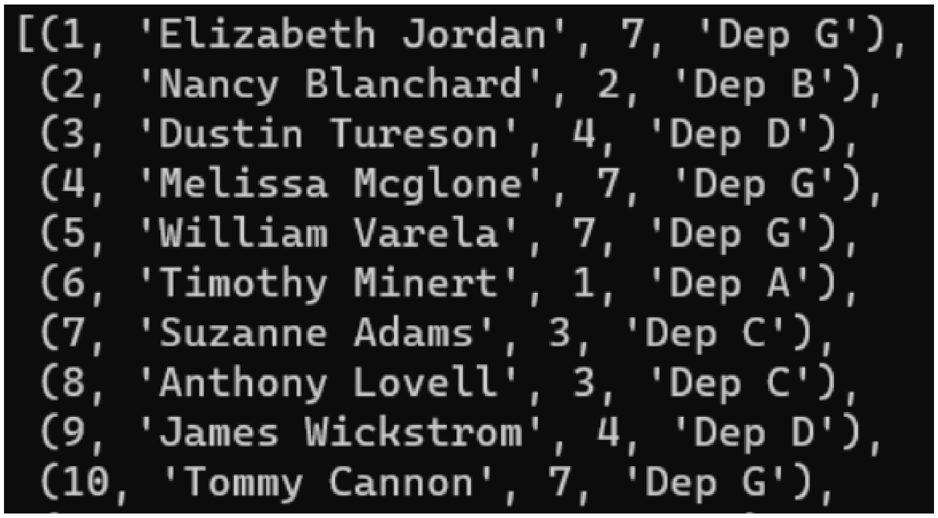
1. BROADCAST JOIN

In Spark, a broadcast join is a technique used to optimize joining operations between two RDDs by broadcasting the smaller RDD to all worker nodes. This is useful when one RDD is significantly smaller than the other and can fit in memory across all nodes. In our case, after reading the "employeesR.csv" (large RDD) and the "departmentsR.csv" (small RDD), we create a local copy of the departments RDD using the broadcast function which ensures that this copy is efficiently distributed to all worker nodes (beforehand, we have transformed the RDD of departments into a key-value mapping, where the department ID is used as the key, for more efficient access). Finally we define a function which produces all pairs of an “employee” with the whole local “department” copy, and we apply this fuse function to each “employee” tuple in the employees RDD using the map transformation. The result is an RDD of tuples with “employee” and “department” information fused together.

The code we used in order to perform this broadcast join is presented in Snippet 8. The first 10 rows of the RDD representing the joined data of employees and their respective departments can also be seen below.

```
employees = sc.textFile("files/employeesR.csv") \
        .map(lambda r: r.split(",")) \
        .map(lambda r: (int(r[0]), r[1], int(r[2])))

departments = sc.textFile("files/departmentsR.csv") \
        .map(lambda r: r.split(",")) \
        .map(lambda r: (int(r[0]), r[1]))

dep_local_copy = sc.broadcast(departments.keyBy(lambda t:
t[0]).collectAsMap())

def fuse_employee_with_its_department(emp):
    emp_id, emp_name, emp_dep_id = emp
    dep_name = dep_local_copy.value[emp_dep_id][1]
    return emp_id, emp_name, emp_dep_id, dep_name

join_result = employees.map(fuse_employee_with_its_department)
```

Snippet 8: Code with Map/Reduce implementing the broadcast join



```
[(1, 'Elizabeth Jordan', 7, 'Dep G'),
 (2, 'Nancy Blanchard', 2, 'Dep B'),
 (3, 'Dustin Tureson', 4, 'Dep D'),
 (4, 'Melissa Mcglone', 7, 'Dep G'),
 (5, 'William Varela', 7, 'Dep G'),
 (6, 'Timothy Minert', 1, 'Dep A'),
 (7, 'Suzanne Adams', 3, 'Dep C'),
 (8, 'Anthony Lovell', 3, 'Dep C'),
 (9, 'James Wickstrom', 4, 'Dep D'),
 (10, 'Tommy Cannon', 7, 'Dep G'),
```

Figure B1: RDD representation of joined data

2. REPARTITION JOIN

A repartition (or shuffle) join refers to a join operation that involves redistributing and reshuffling the data across partitions. When performing a repartition join, Spark shuffles and redistributes the data across the partitions based on the join key. This ensures that the data with the same join key resides on the same partition, making the subsequent join operation more efficient.

The process of implementing the repartition join begins with reading and processing the "employeesR.csv" and the "departmentsR.csv" files. The code we developed reads each file as an RDD of lines, splits the lines by commas and transforms the resulting list into a tuple. This transformation maps each line of each file to an RDD element representing an employee or a department. Then, after tagging and combining (union) the RDDs, we defined a function to create pairs from grouped data. This function takes a tuple `t` as input, where `t[0]` represents the department ID, and `t[1]` is a list of tagged records. It separates the records into two lists based on their tag ("emp" or "dep") and, for each combination of an employee and a department, it creates a new list of tuples and returns it. Finally, the resulting nested list is flattened into a single RDD named `res`, which contains the combined pairs of employees and departments based on the department ID.

The code we used in order to perform this repartition join is presented in Snippet 9. The RDD representation of joined data is omitted because it's the exact same as the one in Figure 8.

```
employees = sc.textFile("files/employeesR.csv") \
      .map(lambda r: r.split(",")) \
      .map(lambda r: (int(r[0]), r[1], int(r[2])))

departments = sc.textFile("files/departmentsR.csv") \
      .map(lambda r: r.split(",")) \
      .map(lambda r: (int(r[0]), r[1]))

employees_tagged = employees.map(lambda r: (r[2], ("emp", r[0], r[1])))
deps_tagged = departments.map(lambda r: (r[0], ("dep", r[1])))

together = employees_tagged.union(deps_tagged)

def make_pairs_from_list(t):
   dep_id = t[0]
   rec_list = t[1]

   emp_list = []
   dep_list = []
   for rec in rec_list:
      if rec[0] == "emp":
         emp_list.append(rec[1:])
      elif rec[0] == "dep":
         dep_list.append(rec[1:])
      else:
         print("This should not happen!")
         assert False

   ret = []
```

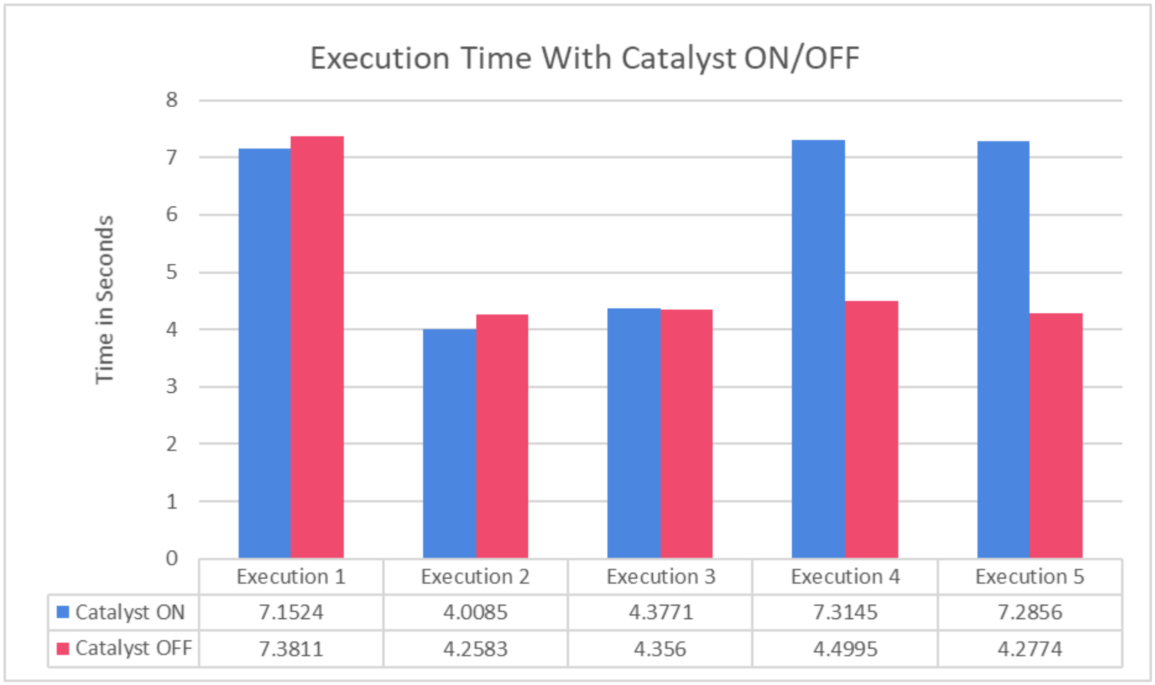
```
    for emp in emp_list:
        for dep in dep_list:
            ret.append(emp + (dep_id,) + dep)

    return ret

res = together.groupByKey() \
        .flatMap(make_pairs_from_list)
```
Snippet 9: Code with Map/Reduce implementing the repartition join

3. OPTIMIZER CATALYST

Catalyst optimizer in Spark is a very useful tool that automatically finds out the most efficient plan to execute data operations specified in the user's program. It “translates” transformations used to build the Dataset to a physical plan of execution. Since it understands the structure of used data and operations made on every dataset, the optimizer can make some decisions helping to reduce the execution time.

In order to conduct Task 3 of Part B, we will execute a given query with optimizer Catalyst enabled, using the command `spark-submit <Name_of_the_file>.py N`, and with optimizer Catalyst disabled, using the command `spark-submit <Name_of_the_file>.py Y` (the problem statement wasn’t clear). If we want to stop spark from choosing a join algorithm automatically, we can use the given script and replace `<conf>` and `<value>` with the specific configuration parameter and its corresponding value: `"spark.sql.optimizer.enabled"` and `False`.

In order to have a better oversight of the execution time with and without the Catalyst optimizer, we executed the query 5x2 times. In every case, the first execution was with the optimizer ON and the second one with the optimizer OFF. We concentrated the results in a barplot shown in Figure B2. The output we got by conducting this query is presented at the Appendix, Part B.

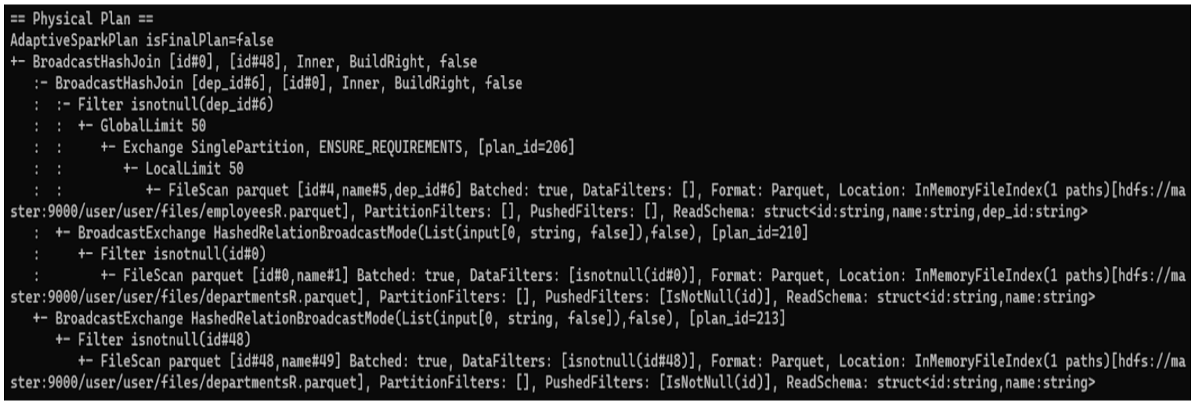


	Execution 1	Execution 2	Execution 3	Execution 4	Execution 5
🟦 Catalyst ON	7.1524	4.0085	4.3771	7.3145	7.2856
🟥 Catalyst OFF	7.3811	4.2583	4.356	4.4995	4.2774

Figure B2: Concentrated Catalyst Results

The results are quite disappointing. As we observe, the first two executions gave slightly improved time when the Catalyst optimizer was enabled, but the difference between this time and the one where the catalyst was OFF is insignificant. In the last three cases, the execution time with the optimizer disabled is clearly decreased (especially in Executions 4, 5) compared to the executions where the optimizer was ON. Based on these results, we come to the conclusion that the performance of Spark queries can depend on several factors, and the impact of the Catalyst optimizer may vary depending on the specific workload and data characteristics. Catalyst optimizer might not always result in better query execution times. In fact, it is entirely possible for a catalyst run to fail to produce useful optimizations (on top of the original execution plan), i.e. the final physical plan can be the same as without running Catalyst. As a matter of fact, this is the case here, which of course explains why there is no significant difference between the running times.

In Figure B3, the recommended physical execution plan is presented. It describes the sequence of operations that will be performed to process data in order to obtain the final result.



```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- BroadcastHashJoin [id#0], [id#48], Inner, BuildRight, false
   :- BroadcastHashJoin [dep_id#6], [id#0], Inner, BuildRight, false
   :  :- Filter isnotnull(dep_id#6)
   :  :  +- GlobalLimit 50
   :  :     +- Exchange SinglePartition, ENSURE_REQUIREMENTS, [plan_id=206]
   :  :        +- LocalLimit 50
   :  :           +- FileScan parquet [id#4,name#5,dep_id#6] Batched: true, DataFilters: [], Format: Parquet, Location: InMemoryFileIndex(1 paths)[hdfs://ma
ster:9000/user/user/files/employeesR.parquet], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<id:string,name:string,dep_id:string>
   :  +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string, false]),false), [plan_id=210]
   :     +- Filter isnotnull(id#0)
   :        +- FileScan parquet [id#0,name#1] Batched: true, DataFilters: [isnotnull(id#0)], Format: Parquet, Location: InMemoryFileIndex(1 paths)[hdfs://ma
ster:9000/user/user/files/departmentsR.parquet], PartitionFilters: [], PushedFilters: [IsNotNull(id)], ReadSchema: struct<id:string,name:string>
   +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string, false]),false), [plan_id=213]
      +- Filter isnotnull(id#48)
         +- FileScan parquet [id#48,name#49] Batched: true, DataFilters: [isnotnull(id#48)], Format: Parquet, Location: InMemoryFileIndex(1 paths)[hdfs://ma
ster:9000/user/user/files/departmentsR.parquet], PartitionFilters: [], PushedFilters: [IsNotNull(id)], ReadSchema: struct<id:string,name:string>
```

Figure B3: Recommended Physical Execution Plan

Here's a breakdown of each operation:

- AdaptiveSparkPlan: Is usually the root node of each main query or sub-query. Before the query runs or when it is running, the isFinalPlan flag of the corresponding AdaptiveSparkPlan node shows as false; after the query execution completes, the isFinalPlan flag changes to true.
- Filter isnotnull: The "Filter isnotnull" operations in the provided execution plan are used to filter out records that have a null value in a specific column.
- GlobalLimit: Limits the number of output records globally across all partitions of the data. It ensures that the final result of the query contains a maximum specified number of records.
- LocalLimit: Limits the number of output records per partition of the data. It restricts the number of records generated by each partition individually before the results are combined.
- FileScan: This operator appears in the plan when you're reading data from some external file, which is most often the case. This can be a csv or json file, but usually it will be some columnar format with metadata - probably parquet.
- BroadcastHashJoin: This operator represents a flavor of join, when one of the datasets is small enough to be broadcasted (sent to all executors) to join with a bigger dataset. In the execution plan, we can see two instances of the BroadcastHashJoin operation. Each join specifies the join condition, the join type (Inner join in this case), and the columns being joined.
- BroadcastExchange: It is used to broadcast data from a single partition to all worker nodes in a Spark cluster.
- Exchange SinglePartition: Indicates that the data is being exchanged between stages or tasks in Spark.

Sources that helped explain each operation:

i. https://www.chashnikov.dev/post/spark-understanding-physical-plans
ii. https://towardsdatascience.com/stop-using-the-limit-clause-wrong-with-spark-646e328774f5
iii. https://towardsdatascience.com/mastering-query-plans-in-spark-3-0-f4c334663aa4

Appendix

Part A

```
warc_schema = StructType([
    StructField("date", StringType()),
    StructField("id", StringType()),
    StructField("type", StringType()),
    StructField("content_length", IntegerType()),
    StructField("public_ip", StringType()),
    StructField("target_url", StringType()),
    StructField("server", StringType()),
    StructField("html_dom", StringType()),
])

warc_df = spark.read.csv("files/warc.csv",
    header=False,
    schema=warc_schema,
    quote='"',
    escape='"',
    # timestampFormat="yyyy-MM-dd'T'HH:mm:ss'Z'"
)
warc_df = warc_df.withColumn("date", to_timestamp(warc_df["date"],
"yyyy-MM-dd'T'HH:mm:ss'Z'"))

warc_df.registerTempTable("warc")

start = datetime(2017, 3, 22, 22, 0, 0)
end = datetime(2017, 3, 22, 23, 0, 0)

query = f"""
SELECT server, COUNT(server) as count
FROM warc
WHERE server IS NOT NULL AND date BETWEEN '{start}' AND '{end}'
GROUP BY server
ORDER BY count DESC;
"""
```

```
result = spark.sql(query)
result.show()
```

Snippet 10: Code Using DataFrames for query no.1

```
warc_schema = StructType([
    StructField("date", TimestampType()),
    StructField("id", StringType()),
    StructField("type", StringType()),
    StructField("content_length", IntegerType()),
    StructField("public_ip", StringType()),
    StructField("target_url", StringType()),
    StructField("server", StringType()),
    StructField("html_dom", StringType()),
])
wat_schema = StructType([
    StructField("warc_record_id", StringType()),
    StructField("metadata_content_length", IntegerType()),
    StructField("targetURL", StringType()),
])

warc_df = spark.read.csv("files/warc.csv", header=False, schema=warc_schema, quote="", escape="")
wat_df = spark.read.csv("files/wat.csv", header=False, schema=wat_schema, quote="", escape="")

warc_df.registerTempTable("warc")
wat_df.registerTempTable("wat")

query = f"""
SELECT LENGTH(warc.html_dom), wat.metadata_content_length
FROM warc
INNER JOIN wat
ON warc.id = wat.warc_record_id
WHERE warc.target_url = 'http://1001.ru/articles/post/ai-da-tumin-443' AND warc.type = 'request'
"""
result = spark.sql(query)
result.show()
```

Snippet 11: Code Using DataFrames for query no.2

```
warc_schema = StructType([
    StructField("date", TimestampType()),
    StructField("id", StringType()),
    StructField("type", StringType()),
    StructField("content_length", IntegerType()),
    StructField("public_ip", StringType()),
    StructField("target_url", StringType()),
    StructField("server", StringType()),
    StructField("html_dom", StringType()),
])

warc_df = spark.read.csv("files/warc.csv", header=False, schema=warc_schema, quote="", escape="")

warc_df.registerTempTable("warc")

query = f"""
SELECT id, target_url, content_length
FROM warc
WHERE server = "Apache"
ORDER BY content_length DESC
LIMIT 5
"""
```

Snippet 12: Code Using DataFrames for query no.3

```
warc_schema = StructType([
    StructField("date", TimestampType()),
    StructField("id", StringType()),
    StructField("type", StringType()),
    StructField("content_length", IntegerType()),
    StructField("public_ip", StringType()),
    StructField("target_url", StringType()),
    StructField("server", StringType()),
    StructField("html_dom", StringType()),
])
wat_schema = StructType([
    StructField("warc_record_id", StringType()),
    StructField("metadata_content_length", IntegerType()),
    StructField("targetURL", StringType()),
])

warc_df = spark.read.csv("files/warc.csv", header=False, schema=warc_schema, quote="", escape="")
wat_df = spark.read.csv("files/wat.csv", header=False, schema=wat_schema, quote="", escape="")

warc_df.registerTempTable("warc")
wat_df.registerTempTable("wat")

query = f"""
SELECT server, AVG(warc.content_length) as avg_cont_len, AVG(wat.metadata_content_length) as
avg_meta_cont_len
FROM warc
INNER JOIN wat
```

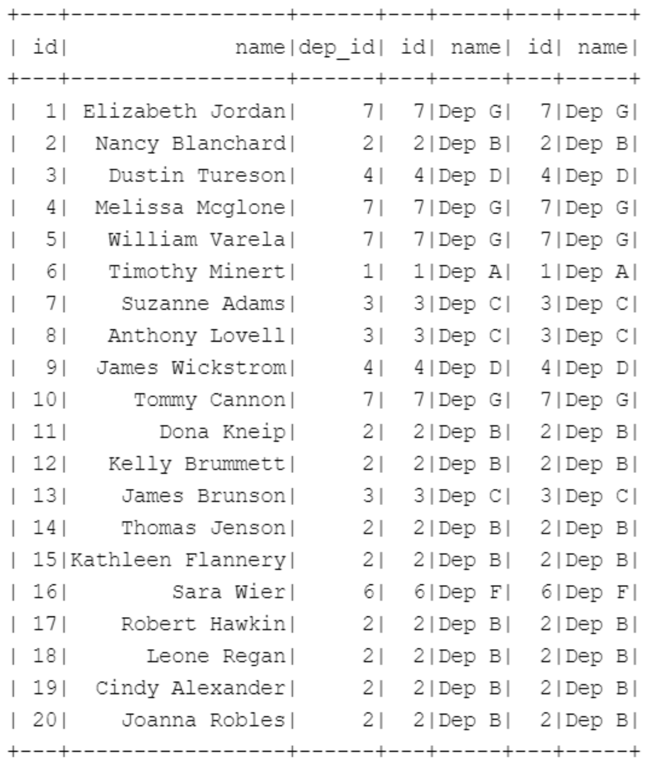
```
ON warc.id = wat.warc_record_id
GROUP BY warc.server
ORDER BY avg_cont_len DESC
LIMIT 5
"""
```

Snippet 13: Code Using DataFrames for query no.4

```
warc_schema = StructType([
    StructField("date", TimestampType()),
    StructField("id", StringType()),
    StructField("type", StringType()),
    StructField("content_length", IntegerType()),
    StructField("public_ip", StringType()),
    StructField("target_url", StringType()),
    StructField("server", StringType()),
    StructField("html_dom", StringType()),
])

warc_df = spark.read.csv("files/warc.csv", header=False, schema=warc_schema, quote="", escape="")

targetHTML = warc_df.withColumn("all_urls", explode(split(warc_df["html_dom"], "http[s]?://")))
targetHTML.registerTempTable("warc")

query = """
SELECT all_urls, COUNT(*) AS cnt
FROM warc
GROUP BY all_urls
ORDER BY cnt DESC
LIMIT 10
"""
```

Snippet 14: Code Using DataFrames for query no.5

Part B



```
+---+-----------------+------+---+-----+---+-----+
| id|             name|dep_id| id| name| id| name|
+---+-----------------+------+---+-----+---+-----+
|  1| Elizabeth Jordan|     7|  7|Dep G|  7|Dep G|
|  2|  Nancy Blanchard|     2|  2|Dep B|  2|Dep B|
|  3|   Dustin Tureson|     4|  4|Dep D|  4|Dep D|
|  4|  Melissa Mcglone|     7|  7|Dep G|  7|Dep G|
|  5|   William Varela|     7|  7|Dep G|  7|Dep G|
|  6|   Timothy Minert|     1|  1|Dep A|  1|Dep A|
|  7|    Suzanne Adams|     3|  3|Dep C|  3|Dep C|
|  8|   Anthony Lovell|     3|  3|Dep C|  3|Dep C|
|  9|  James Wickstrom|     4|  4|Dep D|  4|Dep D|
| 10|     Tommy Cannon|     7|  7|Dep G|  7|Dep G|
| 11|       Dona Kneip|     2|  2|Dep B|  2|Dep B|
| 12|   Kelly Brummett|     2|  2|Dep B|  2|Dep B|
| 13|    James Brunson|     3|  3|Dep C|  3|Dep C|
| 14|    Thomas Jenson|     2|  2|Dep B|  2|Dep B|
| 15|Kathleen Flannery|     2|  2|Dep B|  2|Dep B|
| 16|        Sara Wier|     6|  6|Dep F|  6|Dep F|
| 17|    Robert Hawkin|     2|  2|Dep B|  2|Dep B|
| 18|      Leone Regan|     2|  2|Dep B|  2|Dep B|
| 19|  Cindy Alexander|     2|  2|Dep B|  2|Dep B|
| 20|    Joanna Robles|     2|  2|Dep B|  2|Dep B|
+---+-----------------+------+---+-----+---+-----+
```

Figure A1: Output of given query in Part 2