ICT337_ECA_B2110802_YANGXIANWEISHAWN_30102024.docx

by SHAWN YANG XIAN WEI

Submission date: 30-Oct-2024 11:21PM (UTC+0800)
Submission ID: 2502721109
File name: ICT337_ECA_B2110802_YANGXIANWEISHAWN_30102024.docx (6.13M)
Word count: 10369
Character count: 72328

ICT337
Big Data Computing in the Cloud
July 2024
ECA

Name:	YANG XIAN WEI SHAWN
T-Group	T01
Date Submitted	30 October 2024 Wednesday

(Full marks: 100)

Question 1

Question 1(a) (6 marks)

ANS:

The Spark job execution process in a cluster environment involves several steps.

1. **Spark Job Submission**

The first step involves a user submitting an application to Spark using the terminal command “spark-submit”. When this command is executed, a Spark job would be initialized, and the command would also launch a driver program that is responsible for ensuring that the main functions of the application are executed.

The driver program will define a “SparkContext” object which is stored in the driver program and its purpose is to establish a connection with the Cluster Manager to provide computing resources to the respective worker nodes. A user could configure the Cluster Manager so that it would be configured as the Spark’s standalone Cluster Manager or Mesos, YARN, or Kubernetes by specifying a terminal command.

Once a connection has been established between the “SparkContext” object and the Cluster Manager, the “SparkContext” object would be able to connect with the Worker Nodes who would be able to store and process data.

2. **SparkContext initialization**

The “SparkContext” object which is defined and is stored in the driver program, performs several tasks such as connecting with a Cluster Manager and connecting with Worker Nodes through the Cluster Manager to acquire Executors which are stored in the Worker Nodes.

The executors represent processes that perform computations and store data for active applications that are currently running.

3. **Application Code Distribution**

The driver program would send application code that is defined by the Spark JAR folder which represents a Java Archive folder which stores Spark library files or Python files that are stored in the “SparkContext” object, would be sent to the Executors in the Worker Nodes.

This action ensures that each executor is equipped with necessary source codes to perform their respective assigned tasks. The “SparkContext” object would then send tasks via the Cluster Manager to each executor to execute the tasks.

4. **DAG Creation**

When a user executes an action on the Resilient Distributed Dataset (RDD), this would trigger an execution plan that is represented by a Directed Acyclic Graph (DAG) which would be sent to a cluster. Subsequently, the DAG Scheduler would allocate tasks to different Worker Nodes that are located in the cluster.

The DAG Scheduler purpose is to monitor RDDs and create a minimum schedule to execute a job.

5. Stage Scheduling
The DAG Scheduler would divide the DAG into task stages and each stage stores tasks based on partitioning of input data. The task stages are then sent to the Task Scheduler for execution and the Task Scheduler executes the task stages by the Cluster Manager.

The Task Scheduler purpose is to execute the tasks in the executors.

6. Task Execution
The Task Scheduler get the set of tasks sent by the DAG Scheduler for each task stage and its purpose is to upload the tasks that are stored in the task stages to the cluster, execute the tasks and retry the execution of the tasks if there are any execution failures.

8. Fault Tolerance
Spark achieves fault tolerance through the adoption of RDDs which is a Spark user-facing API which is an immutable distributed datasets that are partitioned by Worker Nodes in a cluster.

The Spark RDDs maintains the lineage of transformations utilized by datasets so that in case of an executor failure, Spark would be able to perform data recovery by recomputing the failed RDD partition by using the lineage of transformations.

9. Result Retrieval
Once all tasks within a stage has completed its execution, the results would be transmitted back to the “SparkContext” object that is stored in the Driver Program and subsequently, the “SparkContext” object would transform the results by aggregation to display the final results.

Question 1(b) (4 marks)

ANS:

The Directed Acyclic Graph (DAG) plays an important role in the Spark framework because for a Spark job to be executed in a cluster, Spark would need to partition the job into small independent tasks that will be executed in parallel.

DAG purpose is to provide a Spark job a logical execution plan by partitioning the job into a sequence of stages and where each stage stores tasks that would be executed independently of other tasks and in parallel.

The DAG enables Spark to perform several types of optimizations for example, pipelining, task reordering and removing unutilized operations to enhance the job execution efficiency.

The DAG partitions jobs into stages and tasks which enables Spark to execute tasks in parallel and allocate them into a cluster of machines for more efficient execution of large processing jobs.

Question 1(c) (10 marks)

ANS:

Spark Resilient Distributed Datasets (RDDs):

Concept:
RDDs represent the original API for Apache Spark and RDDs are a immutable set of data objects that are distributed across Worker Nodes within a Spark Cluster of machines.

RDDs enables parallel processing by partitioning a job into small independent tasks and provides fault tolerance by being immutable which enables RDDs to avoid data corruption by creating a new data object from the existing failed RDD partition instead of overwriting it. RDDs also provides efficient performance of Spark jobs within a cluster of machines by executing in-built memory computations.

Structure:
RDDs represent an immutable set of data objects, and these are either Scala or Java objects that can process structured and unstructured data but do not have predefined schemas which means that users would have to define the schemas.

API:
RDDs provide an OOP-style API that can perform low level transformations, actions and persistence operations. Transformations represent operations performed on RDDs to return a new RDD which for example includes several functions such as map(), filter(), flatMap(), groupByKey(), reduceByKey and join().

Actions represent operations that will either send a Spark job results to the Driver Program or write the results to an external storage location which for example includes several functions such as collect(), count(), take() and reduce().

Persistence represents operations that enables users to persist or cache an RDD in the in-built memory which is a useful operation when you want to reuse an RDD on several occasions which for example includes functions such as persist() and cache().

Optimization:
Spark RDDs do not have built-in optimization engine which means that users would have to manually optimize code performance.

Use Cases:
Spark RDDs would be the preferable option for use cases where users need to have fine-grained control over their datasets and need to perform low-level transformation and action operations on their datasets.

Spark RDDs should also be used for other uses cases such as when users need to work with unstructured data for example media streams and streams of text as well as in uses cases where users don't need to create a schema that applies a format such as a columnar format in a Spark cluster of machines.

DataFrames:

Concept:
Spark DataFrames are immutable distributed sets of data which is similar to Spark RDDs however, Spark DataFrames format data into named columns which is similar to a table that is stored in a relational database. Spark DataFrames provide higher-level abstraction by enabling users to apply a structure to immutable distributed sets of data.

Spark DataFrames provides users with a domain specific language (DSL) API which enables users to manipulate their datasets using DataFrames and opens Spark to a large audience.

Structure:
Spark DataFrames are designed to manage structured and semi-structured data such as CSV, JSON and Parquet files because they have a schema which means that a Spark DataFrame would format data into 2-dimensional table of rows and columns where the columns that have a name and type as well as rows which represent a single record for more efficient data organization.

API:
Spark DataFrames have built-in APIs that provides schema support which enables them to automatically infer a predefined schema when processing structured data so that when the DataFrame is defined, it would have a structure that resembles a SQL table. Users would be able to use the schema to execute query operations on the DataFrame to manipulate the data.

Spark DataFrames also offer users a domain specific language (DSL) API which is used for data manipulation that enables users to perform SQL-like queries that involves several types of query operations such as joining, filtering and aggregation.

Optimization:
Spark DataFrames can use Spark's Catalyst Optimizer to enhance query execution plans to increase performance efficiency by analysing DataFrame query operations followed by producing optimized code.

The Spark's Catalyst Optimizer includes optimizations such as predicated pushdown, column pruning and advanced code generation. The Spark's Catalyst Optimizer main purpose is to allow users to add new Spark DataFrame optimization solutions and features to Spark SQL as well as extend the Catalyst Optimizer.

Use Cases:
Spark DataFrames would be ideal use cases that require a schema to specified and used for processing of structured data or semi-structured data from specific sources such as CSV, JSON and MySQL using higher level abstractions.

Spark DataFrames would also be ideal in use cases that involve data analysis where users need to perform handling of missing values and transformations operations such as groupBy(), agg() and pivot() for the purpose of data exploration and cleaning to quickly identify data distributions.

Question 2

Question 2(a) (6 marks)

ANS:

```
#Import libraries
from pyspark.sql import SparkSession

from pyspark.sql import functions as f

from pyspark.sql.functions import *

from functools import reduce

from pyspark.sql.functions import avg, desc, round

from pyspark.sql.types import IntegerType, DoubleType, DateType, NumericType

from pyspark.sql.functions import to_date

from pyspark.sql.functions import lower, trim, when

from pyspark.sql.functions import sum, col

from pyspark.sql.functions import sum as spark_sum, col

from pyspark.sql.functions import col, avg, format_number

import matplotlib.pyplot as plt

import seaborn as sns

import pandas as pd

import numpy as np
```

```
""" Set the SPARK_LOCAL_IP environment variable: Before running your script, set this environment variable: """
import os
os.environ['SPARK_LOCAL_IP'] = 'localhost'

# Start spark session

""" Set the spark.driver.bindAddress: Add the following configuration to your SparkSession builder:

Use a specific port: If the issue persists, try specifying a port explicitly: """
spark = SparkSession \
    .builder \
    .appName("ICT337 ECA July 2024 Semester Question 2") \
    .config("spark.driver.bindAddress", "localhost") \
    .config("spark.driver.port", "4043") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()



""" Question 2(a) (6 marks) """
print("\nQuestion 2(a) (6 marks)\n")

# Read csv file and store into dataframe
airbnb_data_PySpark_DF = spark\
    .read\
        .option("inferSchema", "true")\
            .option("header", "true")\
                .csv("/Users/shawnyang/Downloads/ICT337 ECA July 2024 Semester/ECA Datasets/airbnb_data.csv")


"""
Sample airbnb_data.csv data:

id,name,host_id,host_name,neighbourhood_group,neighbourhood,latitude,longitude,room_type,price,minimum_night
s,number_of_reviews,last_review,reviews_per_month,calculated_host_listings_count,availability_365
```

```
"""

# Show the type of the object to check if its a dataframe
print(f"Show the type of the Airbnb Data DataFrame:\n{type(airbnb_data_PySpark_DF)}\n")

print("Show the content of airbnb data:\n")

airbnb_data_PySpark_DF.show(20)

""" To change the data types of the columns in your airbnb_data_PySpark_DF based on their content,
you can use the following PySpark code:

This code will change the data types as follows:

Integer columns: id, host_id, minimum_nights, number_of_reviews, calculated_host_listings_count

Double (decimal) columns: latitude, longitude, price, reviews_per_month

Date column: last_review

The other columns (name, host_name, neighbourhood_group, neighbourhood, room_type) will remain as strings. """
airbnb_data_PySpark_DF = airbnb_data_PySpark_DF\
    .withColumn("id", airbnb_data_PySpark_DF["id"].cast(IntegerType())) \
    .withColumn("host_id", airbnb_data_PySpark_DF["host_id"].cast(IntegerType())) \
    .withColumn("latitude", airbnb_data_PySpark_DF["latitude"].cast(DoubleType())) \
    .withColumn("longitude", airbnb_data_PySpark_DF["longitude"].cast(DoubleType())) \
    .withColumn("price", airbnb_data_PySpark_DF["price"].cast(DoubleType())) \
    .withColumn("minimum_nights", airbnb_data_PySpark_DF["minimum_nights"].cast(IntegerType())) \
    .withColumn("number_of_reviews", airbnb_data_PySpark_DF["number_of_reviews"].cast(IntegerType())) \
    .withColumn("last_review", to_date(airbnb_data_PySpark_DF["last_review"])) \
    .withColumn("reviews_per_month", airbnb_data_PySpark_DF["reviews_per_month"].cast(DoubleType())) \
    .withColumn("calculated_host_listings_count",
airbnb_data_PySpark_DF["calculated_host_listings_count"].cast(IntegerType()))
```

```
print("Display the schema of the Airbnb Data DataFrame:\n")


""" This shows the structure of the DataFrame, including column names and their data types. """
airbnb_data_PySpark_DF.printSchema()


# Show the dimensions (number of rows and columns) of the DataFrame
""" .count() and len(airbnb_data_DataFrame.columns): These get the number of rows and columns in the DataFrame.
"""
print(f"Airbnb Data DataFrame Dimensions (number of rows and columns):\n({airbnb_data_PySpark_DF.count()}, {len(airbnb_data_PySpark_DF.columns)})\n")


# Find missing data
print("Check for missing data in each column and display the count:\n")


# Initialize an empty list to store column expressions
airbnb_data_column_Expressions = []


# Iterate through each column in the DataFrame
for airbnb_data_column in airbnb_data_PySpark_DF.columns:
    # Create an expression to count null values for the current column
    null_count_for_column_Expression = count(when(col(airbnb_data_column).isNull(), airbnb_data_column)).alias(airbnb_data_column)
    # Add the expression to the list
    airbnb_data_column_Expressions.append(null_count_for_column_Expression)


# Use select() with the list of expressions to create the missing_data DataFrame
missing_data_PySpark_DF = airbnb_data_PySpark_DF.select(airbnb_data_column_Expressions)


missing_data_PySpark_DF.show()


""" This code selects only the "room_type" column from the DataFrame.  """
distinct_room_types_PySpark_DF = airbnb_data_PySpark_DF.select("room_type").distinct().collect()


""" This code will show you all the distinct values in the room_type column and count them. """
print(f"\nTotal number of distinct room types: {len(distinct_room_types_PySpark_DF)}\n")
```

```
""" To clean up the room_type column and ensure you only have the three expected values,
you can use a combination of string manipulation and filtering. """
airbnb_data_with_distinct_room_types_PySpark_DF = airbnb_data_PySpark_DF.withColumn(
    "room_type",
    when(
        lower(trim("room_type")).isin(["private room", "shared room", "entire home/apt"]),

        lower(trim("room_type"))
    )\
        .otherwise("unknown")
)

""" This code selects only the "room_type" column from the DataFrame.  """
distinct_room_types = airbnb_data_with_distinct_room_types_PySpark_DF.select("room_type").distinct().collect()

print("Distinct room types after cleaning:")

""" It then iterates through each row in the distinct_room_types list

For each row, it prints the value of the "room_type" column """
for row in distinct_room_types:
    print(row["room_type"])

""" After printing all distinct room types, it prints the total count of distinct room types using len(distinct_room_types) """
print(f"\nTotal number of distinct room types after cleaning: {len(distinct_room_types)}\n")

# Count total rows before dropping missing values
total_rows_before_dropping_missing_values_PySpark_DF =
airbnb_data_with_distinct_room_types_PySpark_DF.count()

print(f"Show the number of rows before dropping missing values:
{total_rows_before_dropping_missing_values_PySpark_DF}\n")
```

```
# Drop rows with missing values
""" Create a new DataFrame with missing values removed. """
airbnb_data_cleaned_PySpark_DF = airbnb_data_with_distinct_room_types_PySpark_DF.dropna()


# Count total rows after dropping missing values
total_rows_after_dropping_missing_values_PySpark_DF = airbnb_data_cleaned_PySpark_DF.count()


print(f"Show the number of rows after dropping missing values: {total_rows_after_dropping_missing_values_PySpark_DF}\n")


# Calculate the number of rows dropped
""" Calculate and display the number of rows that were dropped. """
total_rows_dropped_PySpark_DF = total_rows_before_dropping_missing_values_PySpark_DF - total_rows_after_dropping_missing_values_PySpark_DF


print(f"Number of rows that were dropped: {total_rows_dropped_PySpark_DF}\n")


# Show a few rows of the cleaned DataFrame
print("Show a sample of the cleaned DataFrame:\n")


airbnb_data_cleaned_PySpark_DF.show(20)


""" Calculate basic statistics for all columns


This code will generate a summary of basic statistics for all columns in your DataFrame, including:"""
basic_stats_PySpark_DF = airbnb_data_cleaned_PySpark_DF.describe()


# Show the results
print("Basic statistics for each column:\n")


basic_stats_PySpark_DF.show()
```

Output:

Question 2(a) (6 marks)

Show the type of the Airbnb Data DataFrame:
<class 'pyspark.sql.dataframe.DataFrame'>

Show the content of airbnb data:

id	name	host_id	host_name	neighbourhood_group	neighbourhood	latitude	longitude	room_type	price	minimum_nights	number_of_reviews	last_review	reviews_per_m
2539	Clean & quiet apt...	2787	John	Brooklyn	Kensington	40.64749	-73.97237	Private room	149	1	9	2018-10-19	
2595	Skylit Midtown Ca...	2845	Jennifer	Manhattan	Midtown	40.75362	-73.98377	Entire home/apt	225	1	45	2019-05-21	
3647	THE VILLAGE OF HA...	4632	Elisabeth	Manhattan	Harlem	40.80902	-73.9419	Private room	150	3	0	NULL	
3831	Cozy Entire Floor...	4869	LisaRoxanne	Brooklyn	Clinton Hill	40.68514	-73.95976	Entire home/apt	89	1	270	2019-07-05	
5022	Entire Apt: Spaci...	7192	Laura	Manhattan	East Harlem	40.79851	-73.94399	Entire home/apt	80	10	9	2018-11-19	
5099	Large Cozy 1 BR A...	7322	Chris	Manhattan	Murray Hill	40.74767	-73.975	Entire home/apt	200	3	74	2019-06-22	
5121	BlissArtsSpace!!	7356	Garon	Brooklyn	Bedford-Stuyvesant	40.68688	-73.95596	Private room	60	45	49	2017-10-05	
5178	Large Furnished R...	8967	Shunichi	Manhattan	Hell's Kitchen	40.76489	-73.98493	Private room	79	2	430	2019-06-24	
5203	Cozy Clean Guest ...	7490	MaryEllen	Manhattan	Upper West Side	40.80178	-73.96723	Private room	79	2	118	2017-07-21	
5238	Cute & Cozy Lower...	7549	Ben	Manhattan	Chinatown	40.71344	-73.99037	Entire home/apt	150	1	160	2019-06-09	
5295	Beautiful 1br on ...	7702	Lena	Manhattan	Upper West Side	40.80316	-73.96545	Entire home/apt	135	5	53	2019-06-22	
5441	Central Manhattan...	7989	Kate	Manhattan	Hell's Kitchen	40.76076	-73.98867	Private room	85	2	188	2019-06-23	
5803	Lovely Room 1, Ga...	9744	Laurie	Brooklyn	South Slope	40.66829	-73.98779	Private room	89	4	167	2019-06-24	
6021	Wonderful Guest B...	11528	Claudio	Manhattan	Upper West Side	40.79826	-73.96113	Private room	85	2	113	2019-07-05	
6090	West Village Nest...	11975	Alina	Manhattan	West Village	40.7353	-74.00525	Entire home/apt	120	90	27	2018-10-31	
6848	Only 2 stops to M...	15991	Allen & Irina	Brooklyn	Williamsburg	40.70837	-73.95352	Entire home/apt	140	2	148	2019-06-29	
7097	Perfect for Your ...	17571	Jane	Brooklyn	Fort Greene	40.69169	-73.97185	Entire home/apt	215	2	198	2019-06-28	
7322	Chelsea Perfect	18946	Doti	Manhattan	Chelsea	40.74192	-73.99501	Private room	140	1	260	2019-07-01	
7726	Hip Historic Brow...	20950	Adam And Charity	Brooklyn	Crown Heights	40.67592	-73.94694	Entire home/apt	99	3	53	2019-06-22	
7750	Huge 2 BR Upper E...	17985	Sing	Manhattan	East Harlem	40.79685	-73.94872	Entire home/apt	190	7	0	NULL	

only showing top 20 rows

Display the schema of the Airbnb Data DataFrame:

```
root
 |-- id: integer (nullable = true)
 |-- name: string (nullable = true)
 |-- host_id: integer (nullable = true)
 |-- host_name: string (nullable = true)
 |-- neighbourhood_group: string (nullable = true)
 |-- neighbourhood: string (nullable = true)
 |-- latitude: double (nullable = true)
 |-- longitude: double (nullable = true)
 |-- room_type: string (nullable = true)
 |-- price: double (nullable = true)
 |-- minimum_nights: integer (nullable = true)
 |-- number_of_reviews: integer (nullable = true)
 |-- last_review: date (nullable = true)
 |-- reviews_per_month: double (nullable = true)
 |-- calculated_host_listings_count: integer (nullable = true)
 |-- availability_365: integer (nullable = true)
```

Airbnb Data DataFrame Dimensions (number of rows and columns):
(49079, 16)

Check for missing data in each column and display the count:

id	name	host_id	host_name	neighbourhood_group	neighbourhood	latitude	longitude	room_type	price	minimum_nights	number_of_reviews	last_review	reviews_per_month	calculated_host_listings_count	
184	32	350	206	185	185	194	343	185	192	188	341	10379	10221	188	

Total number of distinct room types: 87

Distinct room types after cleaning:
entire home/apt
private room
unknown
shared room

Total number of distinct room types after cleaning: 4

Show the number of rows before dropping missing values: 49079

Show the number of rows after dropping missing values: 38672

Number of rows that were dropped: 10407

Show a sample of the cleaned DataFrame:

id	name	host_id	host_name	neighbourhood_group	neighbourhood	latitude	longitude	room_type	price	minimum_nights	number_of_reviews	last_review	reviews_per_m
2539	Clean & quiet apt...	2787	John	Brooklyn	Kensington	40.64749	-73.97237	private room	149.0	1	9	2018-10-19	
2595	Skylit Midtown Ca...	2845	Jennifer	Manhattan	Midtown	40.75362	-73.98377	entire home/apt	225.0	1	45	2019-05-21	
3831	Cozy Entire Floor...	4869	LisaRoxanne	Brooklyn	Clinton Hill	40.68514	-73.95976	entire home/apt	89.0	1	270	2019-07-05	
5022	Entire Apt: Spaci...	7192	Laura	Manhattan	East Harlem	40.79851	-73.94399	entire home/apt	80.0	10	9	2018-11-19	
5099	Large Cozy 1 BR A...	7322	Chris	Manhattan	Murray Hill	40.74767	-73.975	entire home/apt	200.0	3	74	2019-06-22	
5121	BlissArtsSpace!	7356	Garon	Brooklyn	Bedford-Stuyvesant	40.68688	-73.95596	private room	60.0	45	49	2017-10-05	
5178	Large Furnished R...	8967	Shunichi	Manhattan	Hell's Kitchen	40.76489	-73.98493	private room	79.0	2	430	2019-06-24	
5203	Cozy Clean Guest ...	7490	MaryEllen	Manhattan	Upper West Side	40.80178	-73.96723	private room	79.0	2	118	2017-07-21	
5238	Cute & Cozy Lower...	7549	Ben	Manhattan	Chinatown	40.71344	-73.99037	entire home/apt	150.0	1	160	2019-06-09	
5295	Beautiful 1br on ...	7702	Lena	Manhattan	Upper West Side	40.80316	-73.96545	entire home/apt	135.0	5	53	2019-06-22	
5441	Central Manhattan...	7989	Kate	Manhattan	Hell's Kitchen	40.76076	-73.98867	private room	85.0	2	188	2019-06-23	
5803	Lovely Room 1, Ga...	9744	Laurie	Brooklyn	South Slope	40.66829	-73.98779	private room	89.0	4	167	2019-06-24	
6021	Wonderful Guest B...	11528	Claudio	Manhattan	Upper West Side	40.79826	-73.96113	private room	85.0	2	113	2019-07-05	
6090	West Village Nest...	11975	Alina	Manhattan	West Village	40.7353	-74.00525	entire home/apt	120.0	90	27	2018-10-31	
6848	Only 2 stops to M...	15991	Allen & Irina	Brooklyn	Williamsburg	40.70837	-73.95352	entire home/apt	140.0	2	148	2019-06-29	
7097	Perfect for Your ...	17571	Jane	Brooklyn	Fort Greene	40.69169	-73.97185	entire home/apt	215.0	2	198	2019-06-28	
7322	Chelsea Perfect	18946	Doti	Manhattan	Chelsea	40.74192	-73.99501	private room	140.0	1	260	2019-07-01	
7726	Hip Historic Brow...	20950	Adam And Charity	Brooklyn	Crown Heights	40.67592	-73.94694	entire home/apt	99.0	3	53	2019-06-22	
7801	Sweet and Spaciou...	21207	Chaya	Brooklyn	Williamsburg	40.71842	-73.95718	entire home/apt	299.0	3	9	2011-12-28	
8024	CBG CtyBGd HelpsH...	22486	Lisel	Brooklyn	Park Slope	40.68069	-73.97706	private room	130.0	2	130	2019-07-01	

only showing top 20 rows

Basic statistics for each column:

summary	id	name	host_id	host_name	neighbourhood_group	neighbourhood	latitude	longitude	room_type	
count	38672	38672	38672	38672	38672	38672	38672	38672	38672	
mean	1.807877253646049E7	29.666666666666668	6.410088258365226E7	NaN	NULL	NULL	40.72809231407731	-73.95122429018399	NULL	142.
stddev	1.0693917856207395E7	48.7886598845811	7.583538267883126E7	NaN	NULL	NULL	0.05493468313003914	0.04660635645956902	NULL	197.
min	2539	1 Bed Apt in Uto...	2438	"Destiny ""Bunmi"""	Bronx	Allerton	40.50641	-74.24442	entire home/apt	
max	36455809	ｳｨﾘｱﾑｽﾞﾊﾞｰｸﾞのかわいい...	273841667	소정	Staten Island	Woodside	40.91306	-73.71299	shared room	

Question 2(b) (6 marks)

ANS:

```
""" Question 2(b) (6 marks) """

print("\nQuestion 2(b) (6 marks)\n")


""" Group by neighbourhood_group and calculate average price

Order the results by average price in descending order


Limit the output to the top 10 results  """

neighbourhood_group_average_price_PySpark_DF =
airbnb_data_cleaned_PySpark_DF.groupBy("neighbourhood_group") \

    .agg(round(avg("price"), 2).alias("average_price")) \

    .orderBy(desc("average_price")) \

    .limit(10)


print("Show the top 10 Airbnb's neighbourhood_group sorted from highest to lowest average price:\n")
```

```
# Show the results
neighbourhood_group_average_price_PySpark_DF.show()

""" Convert the Spark DataFrame to a Pandas DataFrame using toPandas(). """
neighbourhood_group_average_price_Pandas_DF = neighbourhood_group_average_price_PySpark_DF.toPandas()

# Display the Pandas DataFrame
print(f"Display the Pandas DataFrame:\n\n{neighbourhood_group_average_price_Pandas_DF}")

""" Set up the plot size. """
plt.figure(figsize=(12, 6))

""" Set up the plot style. """
sns.set_style("whitegrid")

""" Create a bar plot using seaborn's barplot function. """
ax = sns.barplot(x='neighbourhood_group', y='average_price', data=neighbourhood_group_average_price_Pandas_DF)

""" Customize the plot with a title. """
plt.title('Top 10 Airbnb Neighbourhood Groups by Average Price ($)', fontsize=16)

""" Customize the plot with labels. """
plt.xlabel('Neighbourhood Group', fontsize=12)

plt.ylabel('Average Price ($)', fontsize=12)

""" Customize the plot with rotated x-axis labels for better readability. """
plt.xticks(rotation=45, ha='right')

""" Add value labels on top of each bar for precise price information. """
for i, v in enumerate(neighbourhood_group_average_price_Pandas_DF['average_price']):
    ax.text(i, v, f'${v:.2f}', ha='center', va='bottom')
```

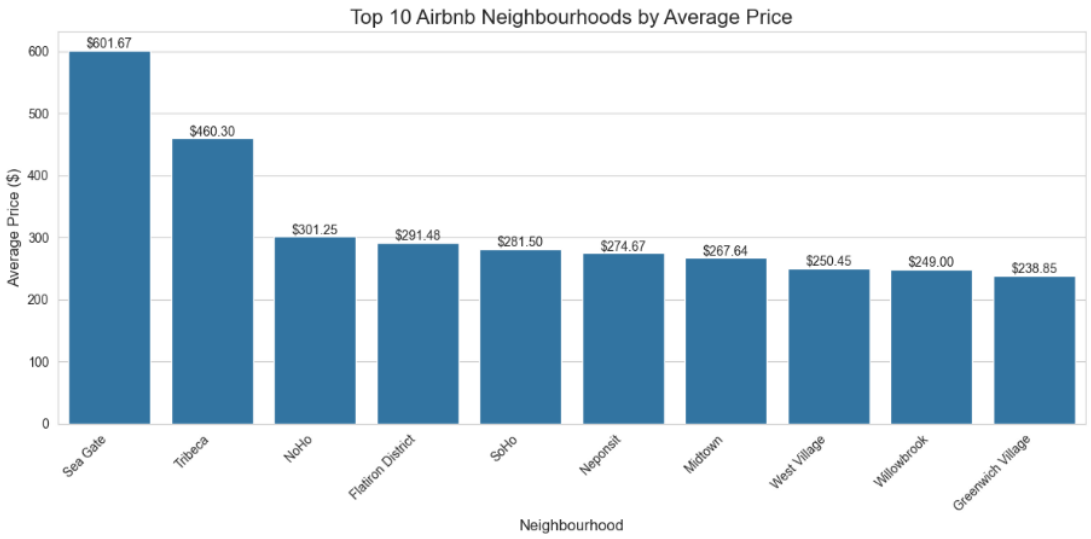
```
""" Adjust the layout of the plot """
plt.tight_layout()

""" Display the plot. """
plt.show()

""" Group the data by neighbourhood and Calculate the average price for each neighbourhood

Order the results by average price in descending order

Limit the output to the top 10 results """
neighbourhood_average_price_PySpark_DF = airbnb_data_cleaned_PySpark_DF.groupBy("neighbourhood") \
    .agg(round(avg("price"), 2).alias("average_price")) \
    .orderBy(desc("average_price")) \
    .limit(10)

print("Show the top 10 Airbnb's neighbourhood sorted from highest to lowest average price:\n")

# Show the results
neighbourhood_average_price_PySpark_DF.show()

""" Convert the Spark DataFrame to a Pandas DataFrame using toPandas(). """
neighbourhood_average_price_Pandas_DF = neighbourhood_average_price_PySpark_DF.toPandas()

# Display the Pandas DataFrame
print(f"Display the Pandas DataFrame:\n\n{neighbourhood_average_price_Pandas_DF}")

""" Set up the plot size. """
plt.figure(figsize=(12, 6))

""" Set up the plot style. """
sns.set_style("whitegrid")

""" Create a bar plot using seaborn's barplot function. """
```

```
ax = sns.barplot(x='neighbourhood', y='average_price', data=neighbourhood_average_price_Pandas_DF.head(10))


""" Customize the plot with a title. """
plt.title('Top 10 Airbnb Neighbourhoods by Average Price ($)', fontsize=16)


""" Customize the plot with labels. """
plt.xlabel('Neighbourhood', fontsize=12)


plt.ylabel('Average Price ($)', fontsize=12)


""" Customize the plot with rotated x-axis labels for better readability. """
plt.xticks(rotation=45, ha='right')


""" Add value labels on top of each bar for precise price information. """
for i, v in enumerate(neighbourhood_average_price_Pandas_DF['average_price'][:10]):
    ax.text(i, v, f'${v:.2f}', ha='center', va='bottom')


""" Adjust the layout of the plot """
plt.tight_layout()


""" Display the plot. """
plt.show()
```

Output:

```
Question 2(b) (6 marks)

Show the top 10 Airbnb's neighbourhood_group sorted from highest to lowest average price:

+-------------------+-------------+
|neighbourhood_group|average_price|
+-------------------+-------------+
|          Manhattan|       180.19|
|           Brooklyn|       121.49|
|             Queens|        95.83|
|      Staten Island|        90.42|
|              Bronx|        79.75|
+-------------------+-------------+

Display the Pandas DataFrame:

  neighbourhood_group  average_price
0           Manhattan         180.19
1            Brooklyn         121.49
2              Queens          95.83
3       Staten Island          90.42
4               Bronx          79.75
```


```
Show the top 10 Airbnb's neighbourhood sorted from highest to lowest average price:

+----------------+-------------+
|    neighbourhood|average_price|
+----------------+-------------+
|         Sea Gate|       601.67|
|          Tribeca|        460.3|
|             NoHo|       301.25|
|Flatiron District|       291.48|
|            SoHo|        281.5|
|         Neponsit|       274.67|
|          Midtown|       267.64|
|     West Village|       250.45|
|      Willowbrook|        249.0|
|Greenwich Village|       238.85|
+----------------+-------------+

Display the Pandas DataFrame:

       neighbourhood  average_price
0           Sea Gate         601.67
1            Tribeca         460.30
2               NoHo         301.25
3  Flatiron District         291.48
4               SoHo         281.50
...
6           Midtown         267.64
7      West Village         250.45
8       Willowbrook         249.00
9  Greenwich Village         238.85
```



Top 10 Airbnb Neighbourhood Groups by Average Price Bar Graph Insights:

1st Insight:

The 1st insight is that the Manhatan neighourhood group is the most expensive neighbourhood group because it has the highest average price of $180.19.

This insight also may mean that the Manhatan neighourhood group is the more luxurious option that caters to wealthier customers or business customers.

2nd Insight:

The 2nd insight is that the Bronx neighourhood group is the least expensive neighbourhood group because it has the lowest average price of $79.75.

This insight also may mean that the Bronx neighourhood group is the most affordable option that caters to budget-conscious customers or customers who cannot afford the more expensive alternatives.

Top 10 Airbnb Neighbourhoods by Average Price Bar Graph Insights:

1st Insight:

The 1st insight is that the most expensive neighbourhood is Sea Gate which has an average price of $601.67.

The 2nd most expensive neighbourhood is Tribeca which has an average price of $460.30

The least expensive neighbourhood is Greenwhich Village which has an average price of $238.85.

This insight means that there is a large variation in average prices across the various neighbourhoods.

2nd Insight:

The 2nd insight is the neighbourhoods like NoHo, Flatiron District, SoHo, Neponsit, Midtown, West Village, Willowbrook and Greenwich Village have average prices ranging from $301.25 to $238.85 which suggest a small average price variation between them.

This insight also means that there are more afforadble neighbourhood options than expensive neighbourhood options.

Question 2(c) (6 marks)

ANS:

```
""" Question 2(c) (6 marks) """
print("\nQuestion 2(c) (6 marks)\n")


# Show a few rows of the cleaned DataFrame
print("Show a sample of the cleaned DataFrame:\n")


airbnb_data_cleaned_PySpark_DF.show(10)


""" Creates a new DataFrame with unique host_ids and their total number of reviews. """
airbnb_data_cleaned_unique_hosts_total_reviews_PySpark_DF = airbnb_data_cleaned_PySpark_DF\
    .groupBy("host_id") \
        .agg(spark_sum("number_of_reviews")\
            .alias("total_reviews_across_each_hosts"))


print("Updated DataFrame with unique hosts and their total reviews:\n")


# Show the results
airbnb_data_cleaned_unique_hosts_total_reviews_PySpark_DF.show(10)
```

```
""" Shows the top 10 unique hosts by total reviews  """
airbnb_data_cleaned_unique_hosts_total_reviews_PySpark_DF =
airbnb_data_cleaned_unique_hosts_total_reviews_PySpark_DF.orderBy(
    col("total_reviews_across_each_hosts").desc()
)


# Show the updated DataFrame
print("Updated DataFrame with the top 10 unique hosts by total reviews:\n")


airbnb_data_cleaned_unique_hosts_total_reviews_PySpark_DF.show(10)


""" Calculates the new total number of reviews across all unique hosts """
total_Num_Of_Reviews_Across_All_Hosts = airbnb_data_cleaned_unique_hosts_total_reviews_PySpark_DF\
    .agg(spark_sum("total_reviews_across_each_hosts"))\
        .collect()[0][0]


print(f"New total number of reviews across all unique hosts: {total_Num_Of_Reviews_Across_All_Hosts}\n")


""" Calculate the popularity index for each host by adding a new column "popularity_index" to the DataFrame.


Which is calculated as (total_reviews_across_each_hosts / new_total_reviews) * 100.  """
airbnb_data_with_popularity_index_PySpark_DF =
airbnb_data_cleaned_unique_hosts_total_reviews_PySpark_DF.withColumn(
    "popularity_index_(%)",
    (col("total_reviews_across_each_hosts") / total_Num_Of_Reviews_Across_All_Hosts) * 100
)


print("Show each unique host, their total reviews and popularity index (%):\n")


""" Show the 1st 10 rows of the relevant columns to verify the new column has been added correctly. """
airbnb_data_with_popularity_index_PySpark_DF.show(10)


# Join the DataFrames
```

```
""" Joins the original DataFrame with the popularity index DataFrame on the "host_id" column."""
combined_PySpark_DF = airbnb_data_cleaned_PySpark_DF.join(airbnb_data_with_popularity_index_PySpark_DF, "host_id")


print("Show the combined DataFrame:\n")


combined_PySpark_DF.show(10)


""" This will display the hosts with the highest popularity index first,
giving you a quick view of the most popular hosts based on their share of total reviews. """
host_by_popularity_PySpark_DF = airbnb_data_with_popularity_index_PySpark_DF.select(
        "host_id",
        format_number("popularity_index_(%)", 3)\
            .alias("popularity_index_(%)")
    ) \
    .orderBy(col("popularity_index_(%)").desc())


print("Top Ten (10) most popular host with the structure (i.e., [host_id, popularity_index]):\n")


# Show the results
host_by_popularity_PySpark_DF.show(10, truncate=False)


""" Creates a new DataFrame by selecting two specific columns from the combined_df DataFrame:


    "neighbourhood": This column contains the names of different neighborhoods.


    "popularity_index_(%)": This column contains the popularity index for each neighborhood. """
neighborhood_by_popularity_PySpark_DF = combined_PySpark_DF.select("neighbourhood", "popularity_index_(%)")


print("Show the most popular neighbourhood by popularity DataFrame:\n")


neighborhood_by_popularity_PySpark_DF.show(10, truncate=False)


print("Show the neighbourhoods by average popularity DataFrame:\n")
```

```
""" Group the data by neighborhood and calculate the average popularity index for each neighborhood. """
neighborhood_by_average_popularity_PySpark_DF = neighborhood_by_popularity_PySpark_DF.groupBy("neighbourhood") \
    .agg(avg("popularity_index_(%)").alias("avg_popularity_index_(%)"))

neighborhood_by_average_popularity_PySpark_DF.show(10, truncate=False)

""" Sort the neighborhoods by their average popularity index in descending order and select the top 10 rows. """
sorted_neighborhood_by_average_popularity_PySpark_DF = neighborhood_by_average_popularity_PySpark_DF \
    .withColumn("avg_popularity_index_(%)", round(col("avg_popularity_index_(%)"), 3)) \
    .orderBy(col("avg_popularity_index_(%)").desc()) \
    .limit(10)

# Show the results
print("Top 10 Most Popular Neighborhoods based on Host's Popularity Index:")

sorted_neighborhood_by_average_popularity_PySpark_DF.show(10, truncate=False)

""" Converts the Spark DataFrame to a Pandas DataFrame for easier plotting """
sorted_neighborhood_by_average_popularity_Pandas_DF = sorted_neighborhood_by_average_popularity_PySpark_DF.toPandas()

# Display the Pandas DataFrame
print(f"Display the Pandas DataFrame:\n\n{sorted_neighborhood_by_average_popularity_Pandas_DF}")

# Set up the plot size
plt.figure(figsize=(12, 6))

# Set up the plot style
sns.set_style("whitegrid")

# Create a bar plot
ax = sns.barplot(x='neighbourhood', y='avg_popularity_index_(%)', data=sorted_neighborhood_by_average_popularity_Pandas_DF)
```

```
""" Customize the plot with a title. """
plt.title('Top 10 Most Popular Neighborhoods by Average Host Popularity Index', fontsize=16)

""" Customize the plot with labels. """
plt.xlabel('Neighborhood', fontsize=12)

plt.ylabel('Average Popularity Index (%)', fontsize=12)

""" Customize the plot with rotated x-axis labels for better readability. """
plt.xticks(rotation=45, ha='right')

""" Adds value labels on top of each bar for precise percentage information. """
for i, v in enumerate(sorted_neighborhood_by_average_popularity_Pandas_DF['avg_popularity_index_(%)']):
    ax.text(i, v, f'{v:.3f}%', ha='center', va='bottom')

# Adjust the layout
plt.tight_layout()

# Display the plot
plt.show()
```

Output:

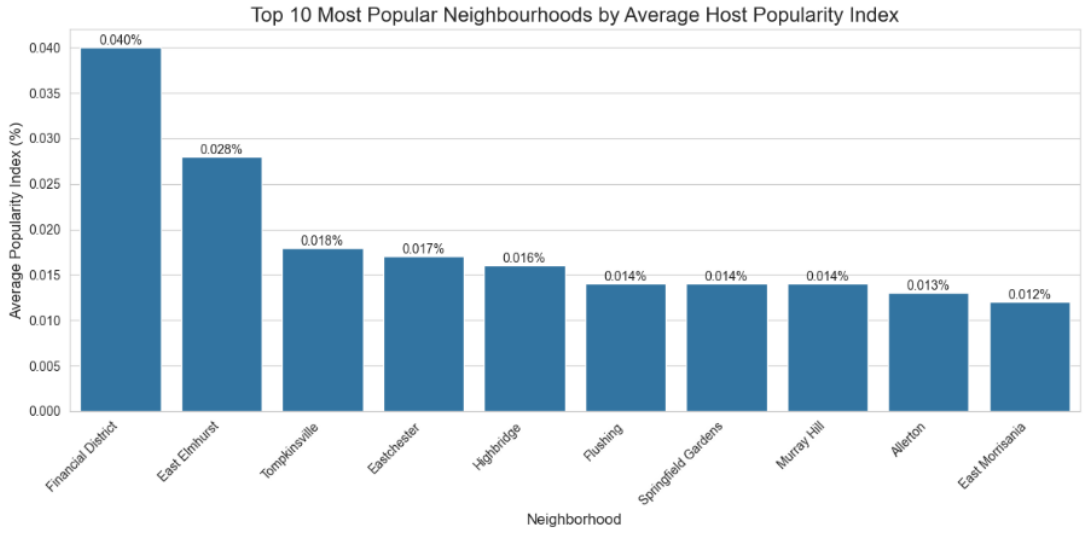
```
Question 2(c) (6 marks)

Show a sample of the cleaned DataFrame:

+----+--------------------+-------+-----------+-------------------+-----------------+--------+---------+---------------+-----+--------------+-----------------+-----------+------------------+
|  id|                name|host_id|  host_name|neighbourhood_group|     neighbourhood|latitude|longitude|      room_type|price|minimum_nights|number_of_reviews|last_review|reviews_per_month|
+----+--------------------+-------+-----------+-------------------+-----------------+--------+---------+---------------+-----+--------------+-----------------+-----------+------------------+
|2539|Clean & quiet apt...|   2787|       John|           Brooklyn|        Kensington|40.64749|-73.97237|   private room|149.0|             1|                9| 2018-10-19|              0.21|
|2595|Skylit Midtown Ca...|   2845|   Jennifer|          Manhattan|          Midtown|40.75362|-73.98377|entire home/apt|225.0|             1|               45| 2019-05-21|              0.38|
|3831|Cozy Entire Floor...|   4869|LisaRoxanne|           Brooklyn|      Clinton Hill|40.68514|-73.95976|entire home/apt| 89.0|             1|              270| 2019-07-05|              4.64|
|5022|Entire Apt: Spaci...|   7192|      Laura|          Manhattan|       East Harlem|40.79851|-73.94399|entire home/apt| 80.0|            10|                9| 2018-11-19|               0.1|
|5099|Large Cozy 1 BR A...|   7322|      Chris|          Manhattan|       Murray Hill|40.74767|  -73.975|entire home/apt|200.0|             3|               74| 2019-06-22|              0.59|
|5121|     BlissArtsSpace!|   7356|      Garon|           Brooklyn|Bedford-Stuyvesant|40.68688|-73.95596|   private room| 60.0|            45|               49| 2017-10-05|               0.4|
|5178|Large Furnished R...|   8967|   Shunichi|          Manhattan|    Hell's Kitchen|40.76489|-73.98493|   private room| 79.0|             2|              430| 2019-06-24|              3.47|
|5203|Cozy Clean Guest ...|   7490|  MaryEllen|          Manhattan|  Upper West Side|40.80178|-73.96723|   private room| 79.0|             2|              118| 2017-07-21|              0.99|
|5238|Cute & Cozy Lower...|   7549|        Ben|          Manhattan|         Chinatown|40.71344|-73.99037|entire home/apt|150.0|             1|              160| 2019-06-09|              1.33|
|5295|Beautiful 1br on ...|   7702|       Lena|          Manhattan|  Upper West Side|40.80316|-73.96545|entire home/apt|135.0|             5|               53| 2019-06-22|              0.43|
+----+--------------------+-------+-----------+-------------------+-----------------+--------+---------+---------------+-----+--------------+-----------------+-----------+------------------+
only showing top 10 rows

Updated DataFrame with unique hosts and their total reviews:

+-------+-----------------------------------+
|host_id|total_reviews_across_each_hosts|
+-------+-----------------------------------+
| 291112|                                 35|
|1384111|                                103|
|1597481|                                 13|
|2108853|                                 18|
|2429432|                                 27|
|2530670|                                134|
|3432742|                                  2|
|1360296|                                 13|
|2124690|                                  1|
|6414252|                                  1|
+-------+-----------------------------------+
only showing top 10 rows

Updated DataFrame with the top 10 unique hosts by total reviews:

+---------+-----------------------------------+
|  host_id|total_reviews_across_each_hosts|
+---------+-----------------------------------+
| 37312959|                               2273|
|   344035|                               2205|
| 26432133|                               2017|
| 35524316|                               1971|
| 40176101|                               1818|
|  4734398|                               1798|
| 16677326|                               1355|
|  6885157|                               1346|
|219517861|                               1281|
| 23591164|                               1269|
+---------+-----------------------------------+
only showing top 10 rows

New total number of reviews across all unique hosts: 1131950

Show each unique host, their total reviews and popularity index (%):

+---------+-----------------------------------+--------------------+
|  host_id|total_reviews_across_each_hosts|popularity_index_(%)|
+---------+-----------------------------------+--------------------+
| 37312959|                               2273|  0.2008039224347365|
|   344035|                               2205| 0.19479658995538673|
| 26432133|                               2017| 0.17818808251247847|
| 35524316|                               1971| 0.17412429877644772|
| 40176101|                               1818|  0.1606078006979107|
|  4734398|                               1798| 0.15884093820398426|
| 16677326|                               1355| 0.11970493396351428|
|  6885157|                               1346| 0.11890984584124739|
|219517861|                               1281| 0.11316754273598657|
| 23591164|                               1269| 0.11210742523963073|
+---------+-----------------------------------+--------------------+
only showing top 10 rows
```

Show the combined DataFrame:

```
+-------+----+--------------------+-----------+-------------------+-----------------+--------+---------------+-----+--------------+-----------------+-----------+------------------+
|host_id|  id|                name|  host_name|neighbourhood_group|     neighbourhood|latitude|      longitude|      room_type|price|minimum_nights|number_of_reviews|last_review|reviews_per_month|
+-------+----+--------------------+-----------+-------------------+-----------------+--------+---------------+-----+--------------+-----------------+-----------+------------------+
|   2787|2539|Clean & quiet apt...|       John|           Brooklyn|        Kensington|40.64749|-73.97237|   private room|149.0|             1|                9| 2018-10-19|              0.21|
|   2845|2595|Skylit Midtown Ca...|   Jennifer|          Manhattan|          Midtown|40.75362|-73.98377|entire home/apt|225.0|             1|               45| 2019-05-21|              0.38|
|   4869|3831|Cozy Entire Floor...|LisaRoxanne|           Brooklyn|      Clinton Hill|40.68514|-73.95976|entire home/apt| 89.0|             1|              270| 2019-07-05|              4.64|
|   7192|5022|Entire Apt: Spaci...|      Laura|          Manhattan|       East Harlem|40.79851|-73.94399|entire home/apt| 80.0|            10|                9| 2018-11-19|               0.1|
|   7322|5099|Large Cozy 1 BR A...|      Chris|          Manhattan|       Murray Hill|40.74767|  -73.975|entire home/apt|200.0|             3|               74| 2019-06-22|              0.59|
|   7356|5121|     BlissArtsSpace!|      Garon|           Brooklyn|Bedford-Stuyvesant|40.68688|-73.95596|   private room| 60.0|            45|               49| 2017-10-05|               0.4|
|   8967|5178|Large Furnished R...|   Shunichi|          Manhattan|    Hell's Kitchen|40.76489|-73.98493|   private room| 79.0|             2|              430| 2019-06-24|              3.47|
|   7490|5203|Cozy Clean Guest ...|  MaryEllen|          Manhattan|   Upper West Side|40.80178|-73.96723|   private room| 79.0|             2|              118| 2017-07-21|              0.99|
|   7549|5238|Cute & Cozy Lower...|        Ben|          Manhattan|         Chinatown|40.71344|-73.99037|entire home/apt|150.0|             1|              160| 2019-06-09|              1.33|
|   7702|5295|Beautiful 1br on ...|       Lena|          Manhattan|   Upper West Side|40.80316|-73.96545|entire home/apt|135.0|             5|               53| 2019-06-22|              0.43|
+-------+----+--------------------+-----------+-------------------+-----------------+--------+---------------+-----+--------------+-----------------+-----------+------------------+
only showing top 10 rows

Top Ten (10) most popular host with the structure (i.e., [host_id, popularity_index]):

+---------+---------------------+
|host_id  |popularity_index_(%)|
+---------+---------------------+
|37312959 |0.201                |
|344035   |0.195                |
|26432133 |0.178                |
|35524316 |0.174                |
|40176101 |0.161                |
|4734398  |0.159                |
|16677326 |0.120                |
|6885157  |0.119                |
|219517861|0.113                |
|23591164 |0.112                |
+---------+---------------------+
only showing top 10 rows
```

```
Show the most popular neighbourhood by popularity DataFrame:

+------------------+---------------------+
|neighbourhood     |popularity_index_(%) |
+------------------+---------------------+
|Chelsea           |0.030892089364371218 |
|Sunnyside         |0.009099341843721012 |
|Sunnyside         |0.009099341843721012 |
|Bedford-Stuyvesant|0.0011484606210521667|
|Hell's Kitchen    |0.0015901762445337693|
|Park Slope        |0.0023852643668800654 |
|Boerum Hill       |0.011837978709306949 |
|West Village      |1.7668624939264104E-4|
|East New York     |0.0011484606210521667|
|Bedford-Stuyvesant|8.834312469632052E-5 |
+------------------+---------------------+
only showing top 10 rows

Show the neighbourhoods by average popularity DataFrame:

+--------------+--------------------------+
|neighbourhood|avg_popularity_index_(%)|
+--------------+--------------------------+
|Corona        |0.006444478630864347     |
|Prince's Bay |0.010159459340076866     |
|Richmondtown |0.00697910685100932      |
|Westerleigh   |7.950881222668846E-4     |
|Mill Basin    |0.0015239189010115288    |
|Civic Center |0.0030743407394319545    |
|Douglaston    |0.0031424911499119 72    |
|Mount Hope    |0.0035337249878528193    |
|Marble Hill   |0.0015901762445337693    |
|Rego Park     |0.0043670951308214 48    |
+--------------+--------------------------+
only showing top 10 rows
```

```
Top 10 Most Popular Neighborhoods based on Host's Popularity Index:
+-------------------+--------------------------+
|neighbourhood      |avg_popularity_index_(%)|
+-------------------+--------------------------+
|Financial District |0.04                      |
|East Elmhurst      |0.028                     |
|Tompkinsville      |0.018                     |
|Eastchester        |0.017                     |
|Highbridge         |0.016                     |
|Flushing           |0.014                     |
|Springfield Gardens|0.014                     |
|Murray Hill        |0.014                     |
|Allerton           |0.013                     |
|East Morrisania    |0.012                     |
+-------------------+--------------------------+

Display the Pandas DataFrame:

       neighbourhood  avg_popularity_index_(%)
0   Financial District                     0.040
1        East Elmhurst                     0.028
2        Tompkinsville                     0.018
3          Eastchester                     0.017
4           Highbridge                     0.016
5             Flushing                     0.014
6  Springfield Gardens                     0.014
7          Murray Hill                     0.014
8             Allerton                     0.013
9      East Morrisania                     0.012
```



Top 10 Most Popular Neighbourhoods by Average Host Popularity Index Bar Graph Insights:

1st Insight:

The 1st insight is that the Financial District is the most popular neighbourhood with the highest average host popularity index of about 0.040% while East Morrisania is the least popular neighbourhood with the lowest average host popularity index of about 0.012%.

2nd Insight:

The 2nd insight is that the popularity of the neighbourhoods began to decrease gradually from Tompkinsville to East Morrisania with the average host popularity index ranging from about 0.018% to 0.012%.

Question 2(d) (6 marks)

ANS:

```
""" Question 2(d) (6 marks) """

print("\nQuestion 2(d) (6 marks)\n")


print("Show the combined DataFrame:\n")


combined_PySpark_DF.show(10)
```

```
""" Select the "room_type" column from the combined DataFrame.

Use the distinct() function to get unique values. """
distinct_room_types_PySpark_DF = combined_PySpark_DF\
    .select("room_type")\
        .distinct()

# Show the results
print("Available room types:\n")

distinct_room_types_PySpark_DF.show()

""" Get a list of distinct room types from the previously created DataFrame. """

""" It creates an empty list room_types  """
room_types = []

""" Then iterates through each row in the DataFrame,

appending the 'room_type' value from each row to the list.  """
for room_type_row in distinct_room_types_PySpark_DF.collect():
    room_types.append(room_type_row['room_type'])

print(f"Distinct room types: {room_types}\n")

""" Create a new DataFrame that calculates the average price for each combination of neighbourhood and room type.
"""
average_price_by_neighbourhood_room_type_PySpark_DF = combined_PySpark_DF\
    .groupBy("neighbourhood", "room_type")\
    .agg(
        round(avg("price"), 2).alias("avg_price")
    )

print("DataFrame with average price for each neighbourhood and room type:\n")
```

```
average_price_by_neighbourhood_room_type_PySpark_DF.show(10)


""" Pivots the DataFrame to create columns for each room type. """
average_price_by_neighbourhood_room_type_pivoted_PySpark_DF =
average_price_by_neighbourhood_room_type_PySpark_DF\
    .groupBy("neighbourhood")\
    .pivot("room_type")\
    .agg(
        round(avg("avg_price"), 2)
    ).na.fill(0)


print("Pivot the DataFrame to create columns for each room type:\n")


average_price_by_neighbourhood_room_type_pivoted_PySpark_DF.show(10)


""" Sort the results by neighbourhood name in ascending order """
sorted_average_price_by_neighbourhood_room_type_pivoted_PySpark_DF =
average_price_by_neighbourhood_room_type_pivoted_PySpark_DF.orderBy("neighbourhood")


# Show the results
print("Average price for each neighbourhood and room type sorted by neighbourhood in acscending order:\n")


sorted_average_price_by_neighbourhood_room_type_pivoted_PySpark_DF.show()


""" Get the top 20 neighborhoods by total average price. """
top_neighborhoods_by_neighborhoods_and_room_types_PySpark_DF =
sorted_average_price_by_neighbourhood_room_type_pivoted_PySpark_DF\
    .select(
        "neighbourhood",

        (col("Entire home/apt") + col("Private room") + col("Shared room")).alias("total_avg_price")


    )\
        .orderBy("total_avg_price", ascending=False)\
```

```
        .limit(20)


print("Show the ranking of the top 20 neighborhoods based on their total average price across all room types\n")


top_neighborhoods_by_neighborhoods_and_room_types_PySpark_DF.show()


""" Filters and joins DataFrames to create a new DataFrame containing detailed information about the top 20 neighborhoods. """
top_20_neighborhoods_data_PySpark_DF = sorted_average_price_by_neighbourhood_room_type_pivoted_PySpark_DF.join(top_neighborhoods_by_neighborhoods_and_room_types_PySpark_DF, "neighbourhood")


print("Filter the original DataFrame to include only the top 20 neighborhoods:\n")


top_20_neighborhoods_data_PySpark_DF.show(20, truncate=False)


""" Convert to Pandas DataFrame for easier plotting """
top_20_neighborhoods_data_Pandas_DF = top_20_neighborhoods_data_PySpark_DF.toPandas()


print("Check the actual column names in your Pandas DataFrame:\n")


print(top_20_neighborhoods_data_Pandas_DF.columns)


""" Use the pandas melt() function to reshape the DataFrame from wide to long format. """
top_20_neighborhoods_data_Melted_Pandas_DF = pd.melt(
    top_20_neighborhoods_data_Pandas_DF,
    id_vars=['neighbourhood'],
    value_vars=['entire home/apt', 'private room', 'shared room'],
    var_name='room_type',
    value_name='avg_price'
)


print("\nMelt the DataFrame to long format\n")


print(top_20_neighborhoods_data_Melted_Pandas_DF)
```

```
# Create the plot


# Set up the plot size
plt.figure(figsize=(15, 10))


""" Create a bar plot using the seaborn library. """
sns.barplot(x='neighbourhood', y='avg_price', hue='room_type',
data=top_20_neighborhoods_data_Melted_Pandas_DF)


# Customize the plot


""" Customize the plot with a title. """
plt.title('Average Price by Room Type for Top 20 Neighbourhoods', fontsize=16)


""" Customize the plot with labels. """
plt.xlabel('Neighbourhood', fontsize=12)


plt.ylabel('Average Price ($)', fontsize=12)


""" Customize the plot with rotated x-axis labels for better readability. """
plt.xticks(rotation=45, ha='right')


""" Customize the plot with a legend. """
plt.legend(title='Room Type')


# Adjust the layout
plt.tight_layout()


# Display the plot
plt.show()
```

Output:

```
Question 2(d) (6 marks)

Show the combined DataFrame:

+-------+----+--------------------+-----------+---------------+------------------+--------+---------+---------------+-----+--------
|host_id|  id|                name|  host_name|neighbourhood_group|     neighbourhood|latitude|longitude|      room_type|price|minimum_
+-------+----+--------------------+-----------+---------------+------------------+--------+---------+---------------+-----+--------
|   2787|2539|Clean & quiet apt...|       John|       Brooklyn|        Kensington|40.64749|-73.97237|   private room|149.0|
|   2845|2595|Skylit Midtown Ca...|   Jennifer|      Manhattan|           Midtown|40.75362|-73.98377|entire home/apt|225.0|
|   4869|3831|Cozy Entire Floor...|LisaRoxanne|       Brooklyn|      Clinton Hill|40.68514|-73.95976|entire home/apt| 89.0|
|   7192|5022|Entire Apt: Spaci...|      Laura|      Manhattan|       East Harlem|40.79851|-73.94399|entire home/apt| 80.0|
|   7322|5099|Large Cozy 1 BR A...|      Chris|      Manhattan|       Murray Hill|40.74767|  -73.975|entire home/apt|200.0|
|   7356|5121|     BlissArtsSpace!|      Garon|       Brooklyn|Bedford-Stuyvesant|40.68688|-73.95596|   private room| 60.0|
|   8967|5178|Large Furnished R...|   Shunichi|      Manhattan|    Hell's Kitchen|40.76489|-73.98493|   private room| 79.0|
|   7490|5203|Cozy Clean Guest ...|  MaryEllen|      Manhattan|   Upper West Side|40.80178|-73.96723|   private room| 79.0|
|   7549|5238|Cute & Cozy Lower...|        Ben|      Manhattan|         Chinatown|40.71344|-73.99037|entire home/apt|150.0|
|   7702|5295|Beautiful 1br on ...|       Lena|      Manhattan|   Upper West Side|40.80316|-73.96545|entire home/apt|135.0|
+-------+----+--------------------+-----------+---------------+------------------+--------+---------+---------------+-----+--------
only showing top 10 rows

Available room types:

+---------------+
|      room_type|
+---------------+
|entire home/apt|
|   private room|
|    shared room|
+---------------+

Distinct room types: ['entire home/apt', 'private room', 'shared room']

DataFrame with average price for each neighbourhood and room type:

+-----------------+---------------+---------+
|    neighbourhood|      room_type|avg_price|
+-----------------+---------------+---------+
|     Far Rockaway|entire home/apt|   122.29|
|     West Village|entire home/apt|   262.39|
|      Fort Greene|   private room|     84.9|
|      Kew Gardens|   private room|     56.5|
|      Port Morris|entire home/apt|    123.0|
|     Belle Harbor|   private room|   178.33|
|         Flatbush|   private room|    68.34|
|    Port Richmond|entire home/apt|    250.0|
|Concourse Village|entire home/apt|   107.43|
|  Stuyvesant Town|   private room|    94.87|
+-----------------+---------------+---------+
only showing top 10 rows

Pivot the DataFrame to create columns for each room type:

+-------------+---------------+------------+-----------+
|neighbourhood|entire home/apt|private room|shared room|
+-------------+---------------+------------+-----------+
|       Corona|         120.75|       49.38|      31.73|
| Richmondtown|          78.0|         0.0|        0.0|
| Prince's Bay|          135.0|         0.0|        0.0|
|   Mill Basin|         179.75|         0.0|        0.0|
|  Westerleigh|          103.0|        40.0|        0.0|
| Civic Center|         206.21|       107.0|        0.0|
|   Douglaston|          134.0|       53.75|        0.0|
|   Mount Hope|          107.9|       45.75|        0.0|
|  Marble Hill|          92.25|       95.17|        0.0|
|    Rego Park|         120.51|       55.29|      45.71|
+-------------+---------------+------------+-----------+
only showing top 10 rows
```

Average price for each neighbourhood and room type sorted by neighbourhood in acscending order:

neighbourhood	entire home/apt	private room	shared room
Allerton	124.6	67.41	0.0
Arden Heights	76.0	41.0	0.0
Arrochar	189.7	46.8	0.0
Arverne	199.8	88.8	0.0
Astoria	136.7	97.29	166.53
Bath Beach	118.0	55.63	0.0
Battery Park City	226.57	107.75	55.0
Bay Ridge	137.4	77.13	63.8
Bay Terrace	186.5	94.5	32.0
Bay Terrace, Stat...	102.5	0.0	0.0
Baychester	90.33	65.67	0.0
Bayside	380.25	50.47	0.0
Bayswater	151.5	58.43	0.0
Bedford-Stuyvesant	151.96	68.1	39.52
Belle Harbor	148.5	178.33	0.0
Bellerose	0.0	80.5	0.0
Belmont	228.67	53.38	49.0
Bensonhurst	103.32	58.51	79.0
Bergen Beach	108.71	85.0	0.0
Boerum Hill	189.25	83.97	0.0

only showing top 20 rows

Show the ranking of the top 20 neighborhoods based on their total average price across all room types:

neighbourhood	total_avg_price
Riverdale	1025.5
Sea Gate	951.0
Tribeca	690.78
SoHo	632.62
West Village	622.14
Midtown	610.7800000000001
Theater District	555.68
Murray Hill	547.76
Vinegar Hill	541.84
Nolita	493.62
Financial District	489.2
Flatiron District	479.05
Greenwich Village	454.57
Chelsea	451.2300000000001
Hell's Kitchen	441.86
NoHo	441.0
Bayside	430.72
Gramercy	419.66999999999996
Kips Bay	418.03999999999996
Upper West Side	414.04

Filter the original DataFrame to include only the top 20 neighborhoods:

neighbourhood	entire home/apt	private room	shared room	total_avg_price
Financial District	246.04	133.91	109.25	489.2
Midtown	289.99	222.19	98.6	610.7800000000001
Bayside	380.25	50.47	0.0	430.72
Hell's Kitchen	233.38	126.74	81.74	441.86
Greenwich Village	279.89	118.68	56.0	454.57
Riverdale	168.5	57.0	800.0	1025.5
Tribeca	556.06	134.72	0.0	690.78
Vinegar Hill	211.71	80.13	250.0	541.84
Flatiron District	323.23	155.82	0.0	479.05
Gramercy	241.04	115.38	63.25	419.66999999999996
NoHo	317.4	123.6	0.0	441.0
Sea Gate	854.0	97.0	0.0	951.0
Upper West Side	235.82	103.48	74.74	414.04
West Village	262.39	179.75	180.0	622.14
Kips Bay	211.06	109.58	97.4	418.03999999999996
Theater District	302.84	173.84	79.0	555.68
SoHo	358.15	126.97	147.5	632.62
Murray Hill	239.33	125.72	182.71	547.76
Chelsea	259.47	120.09	71.67	451.2300000000001
Nolita	302.48	116.14	75.0	493.62

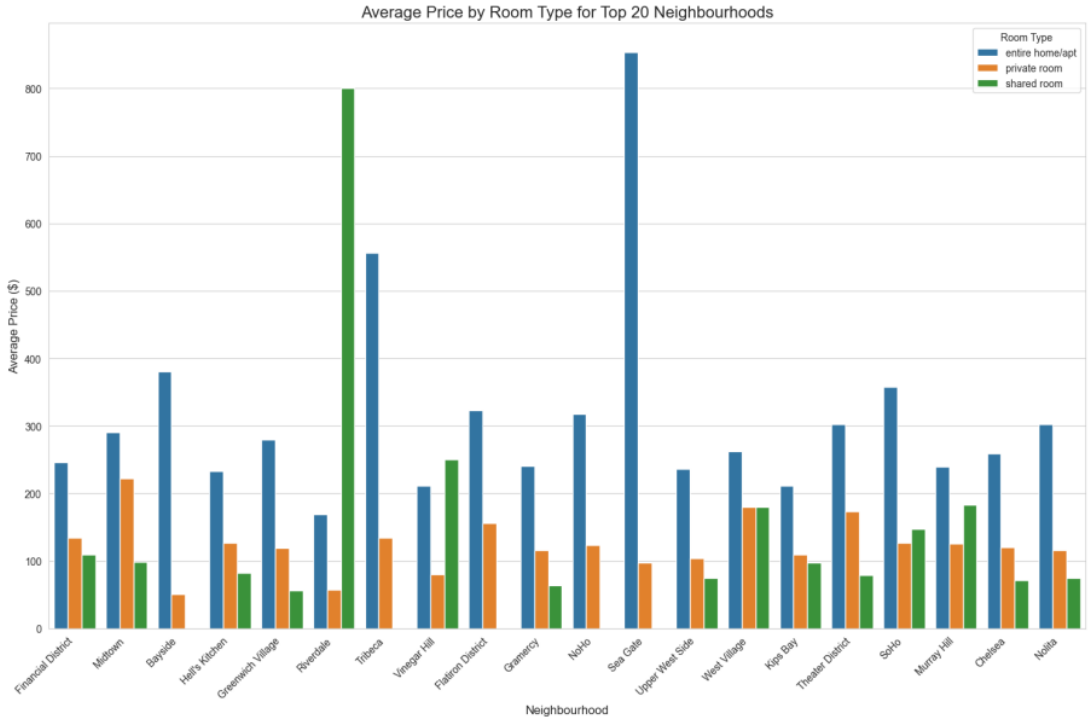
Check the actual column names in your Pandas DataFrame:

```
Index(['neighbourhood', 'entire home/apt', 'private room', 'shared room',
       'total_avg_price'],
      dtype='object')
```

Melt the DataFrame to long format

	neighbourhood	room_type	avg_price
0	Financial District	entire home/apt	246.04
1	Midtown	entire home/apt	289.99
2	Bayside	entire home/apt	380.25
3	Hell's Kitchen	entire home/apt	233.38
4	Greenwich Village	entire home/apt	279.89
5	Riverdale	entire home/apt	168.50
6	Tribeca	entire home/apt	556.06
7	Vinegar Hill	entire home/apt	211.71
8	Flatiron District	entire home/apt	323.23
9	Gramercy	entire home/apt	241.04
10	NoHo	entire home/apt	317.40
11	Sea Gate	entire home/apt	854.00
12	Upper West Side	entire home/apt	235.82
13	West Village	entire home/apt	262.39
14	Kips Bay	entire home/apt	211.06
15	Theater District	entire home/apt	302.84
16	SoHo	entire home/apt	358.15
17	Murray Hill	entire home/apt	239.33
18	Chelsea	entire home/apt	259.47
19	Nolita	entire home/apt	302.48
20	Financial District	private room	133.91

21	Midtown	private room	222.19
22	Bayside	private room	50.47
23	Hell's Kitchen	private room	126.74
24	Greenwich Village	private room	118.68
25	Riverdale	private room	57.00
26	Tribeca	private room	134.72
27	Vinegar Hill	private room	80.13
28	Flatiron District	private room	155.82
29	Gramercy	private room	115.38
30	NoHo	private room	123.60
31	Sea Gate	private room	97.00
32	Upper West Side	private room	103.48
33	West Village	private room	179.75
34	Kips Bay	private room	109.58
35	Theater District	private room	173.84
36	SoHo	private room	126.97
37	Murray Hill	private room	125.72
38	Chelsea	private room	120.09
39	Nolita	private room	116.14
40	Financial District	shared room	109.25
41	Midtown	shared room	98.60
42	Bayside	shared room	0.00
43	Hell's Kitchen	shared room	81.74
44	Greenwich Village	shared room	56.00
45	Riverdale	shared room	800.00
46	Tribeca	shared room	0.00
47	Vinegar Hill	shared room	250.00
48	Flatiron District	shared room	0.00
49	Gramercy	shared room	63.25
50	NoHo	shared room	0.00
51	Sea Gate	shared room	0.00
52	Upper West Side	shared room	74.74
53	West Village	shared room	180.00
54	Kips Bay	shared room	97.40
55	Theater District	shared room	79.00
56	SoHo	shared room	147.50
57	Murray Hill	shared room	182.71
58	Chelsea	shared room	71.67
59	Nolita	shared room	75.00



Average Price by Room Type for Top 20 Neighbourhoods Side-By-Side Bar Chart Insights:

1st Insight:

The 1st insight is room type entire home/apartment has consistently the highest average prices which makes it the most expensive room type across the neighbourhoods.

2nd Insight:

The 2nd insight is that the neighbourhood Sea Gate has the highest average price for the the room type entire home/apartment and the Sea Gate average price for the the room type entire home/apartment is much higher compared to other neighbourhoods.

Question 2(e) (16 marks)

ANS:

```
""" Question 2(e) (16 marks) """
print("\nQuestion 2(e) (16 marks)\n")


print("Show the combined DataFrame:\n")


combined_PySpark_DF.show(10)


""" Calculate the average price for each combination of neighbourhood group and room type. """
sorted_average_price_by_neighbourhood_group_and_room_type_PySpark_DF = combined_PySpark_DF\
    .groupBy("neighbourhood_group", "room_type") \
    .agg(
        round(avg("price"), 2)\
            .alias("average_price")
    ) \
    .orderBy(
        col("average_price").desc()
    )


# Show the results
```

```
print("Average price for each neighbourhood_group and room type, sorted from highest to lowest:\n")


sorted_average_price_by_neighbourhood_group_and_room_type_PySpark_DF.show()


""" Convert to Pandas DataFrame for easier plotting """
sorted_average_price_by_neighbourhood_group_and_room_type_Pandas_DF = sorted_average_price_by_neighbourhood_group_and_room_type_PySpark_DF.toPandas()


# Display the Pandas DataFrame
print(f"Display the Pandas DataFrame:\n\n{sorted_average_price_by_neighbourhood_group_and_room_type_Pandas_DF}")


# Set up the plot
plt.figure(figsize=(12, 8))


""" Create a bar plot using the seaborn library. """
ax = sns.barplot(x='neighbourhood_group', y='average_price', hue='room_type', data=sorted_average_price_by_neighbourhood_group_and_room_type_Pandas_DF)


# Customize the plot


""" Customize the plot with a title. """
plt.title('Average Price by Neighbourhood Group and Room Type', fontsize=16)


""" Customize the plot with labels. """
plt.xlabel('Neighbourhood Group', fontsize=12)


plt.ylabel('Average Price ($)', fontsize=12)


""" Customize the plot with rotated x-axis labels for better readability. """
plt.xticks(rotation=45)


""" Customize the plot with a legend. """
plt.legend(title='Room Type')
```

```
""" Add value labels on top of each bar. """
for container in ax.containers:
    ax.bar_label(container, fmt='${:.2f}', label_type='edge')


""" Adjusts the y-axis limit. """
plt.ylim(0, plt.ylim()[1] * 1.1)


# Adjust the layout
plt.tight_layout()


# Display the plot
plt.show()



""" Count the number of listings for each neighbourhood group. """
host_listings_by_neighbourhood_group_PySpark_DF = combined_PySpark_DF\
    .groupBy("neighbourhood_group") \
    .agg(
        count("id")
        .alias("total_listings")
    ) \
    .orderBy(
        col("total_listings")\
        .desc()
    )


# Show the results
print("Total number of host listings in each neighbourhood group, sorted from highest to lowest:\n")


host_listings_by_neighbourhood_group_PySpark_DF.show()


""" Convert to Pandas DataFrame for easier manipulation if needed """
host_listings_by_neighbourhood_group_Pandas_DF                                                                    =
host_listings_by_neighbourhood_group_PySpark_DF.toPandas()
```

```
# Display the Pandas DataFrame
print(f"Display the Pandas DataFrame:\n\n{host_listings_by_neighbourhood_group_Pandas_DF}")

# Set up the plot
plt.figure(figsize=(10, 6))

# Create the bar plot
sns.barplot(x='neighbourhood_group', y='total_listings', data=host_listings_by_neighbourhood_group_Pandas_DF)

# Customize the plot

""" Customize the plot with a title. """
plt.title('Total Host Listings by Neighbourhood Group', fontsize=16)

""" Customize the plot with labels. """
plt.xlabel('Neighbourhood Group', fontsize=12)

plt.ylabel('Total Listings', fontsize=12)

""" Customize the plot with rotated x-axis labels for better readability. """
plt.xticks(rotation=45)

""" Add value labels on top of each bar. """
for i, v in enumerate(host_listings_by_neighbourhood_group_Pandas_DF['total_listings']):
    plt.text(i, v, str(v), ha='center', va='bottom')

# Adjust the layout
plt.tight_layout()

# Display the plot
plt.show()
```

```
""" Group by host_name and count the number of listings. """
host_names_by_listings_count_PySpark_DF = combined_PySpark_DF\
    .groupBy("host_name") \
    .agg(
        count("id")\
        .alias("listing_count")
    ) \
    .orderBy(
        col("listing_count")\
        .desc()
    ) \
    .limit(10)

# Show the results
print("Top 10 hosts with the highest number of listings:\n")

host_names_by_listings_count_PySpark_DF.show()

""" Convert to Pandas DataFrame for easier manipulation if needed """
host_names_by_listings_count_Pandas_DF = host_names_by_listings_count_PySpark_DF.toPandas()

# Display the Pandas DataFrame
print(f"Display the Pandas DataFrame:\n\n{host_names_by_listings_count_Pandas_DF}")

# Set up the plot
plt.figure(figsize=(12, 6))

# Create the bar plot
sns.barplot(x='host_name', y='listing_count', data=host_names_by_listings_count_Pandas_DF)

# Customize the plot
```

```
""" Customize the plot with a title. """
plt.title('Top 10 Hosts with Highest Number of Listings', fontsize=16)

""" Customize the plot with labels. """
plt.xlabel('Host Name', fontsize=12)

plt.ylabel('Number of Listings', fontsize=12)

""" Customize the plot with rotated x-axis labels for better readability. """
plt.xticks(rotation=45, ha='right')

""" Add value labels on top of each bar. """
for i, v in enumerate(host_names_by_listings_count_Pandas_DF['listing_count']):
    plt.text(i, v, str(v), ha='center', va='bottom')

# Adjust the layout
plt.tight_layout()

# Display the plot
plt.show()



""" Group by host_name and room_type, calculate average number of reviews. """
host_names_and_room_type_by_average_reviews_PySpark_DF = combined_PySpark_DF\
    .groupBy("host_name", "room_type") \
    .agg(
        round(avg("number_of_reviews"), 2)\
        .alias("avg_reviews")
    ) \
    .orderBy(
        col("avg_reviews")\
        .desc()
```

```
    ) \
    .limit(20)


# Show the results
print("Top 20 hosts with the highest average number of reviews by room type:\n")


""" Use truncate=False to show the full content of each column. """
host_names_and_room_type_by_average_reviews_PySpark_DF.show(20, truncate=False)


""" Convert to Pandas DataFrame for easier manipulation if needed """
host_names_and_room_type_by_average_reviews_Pandas_DF = host_names_and_room_type_by_average_reviews_PySpark_DF.toPandas()


# Display the Pandas DataFrame
print(f"Display the Pandas DataFrame:\n\n{host_names_and_room_type_by_average_reviews_Pandas_DF}")


# Set up the plot
plt.figure(figsize=(15, 10))


""" Create a bar plot using the seaborn library. """
sns.barplot(x='host_name', y='avg_reviews', hue='room_type', data=host_names_and_room_type_by_average_reviews_Pandas_DF)


# Customize the plot


""" Customize the plot with a title. """
plt.title('Top 20 Hosts with Highest Average Number of Reviews by Room Type', fontsize=16)


""" Customize the plot with labels. """
plt.xlabel('Host Name', fontsize=12)


plt.ylabel('Average Number of Reviews', fontsize=12)


""" Customize the plot with rotated x-axis labels for better readability. """
```

```
plt.xticks(rotation=90)


""" Customize the plot with a legend. """
plt.legend(title='Room Type')


""" Add value labels on top of each bar.

The value labels on top of each bar provide the


exact average number of reviews for each host and room type combination.c"""
for i, v in enumerate(host_names_and_room_type_by_average_reviews_Pandas_DF['avg_reviews']):
    plt.text(i, v, f'{v:.2f}', ha='center', va='bottom', fontsize=8)


# Adjust the layout
plt.tight_layout()


# Display the plot
plt.show()



spark.stop()
```

Output:

Question 2(e) (16 marks)

Show the combined DataFrame:

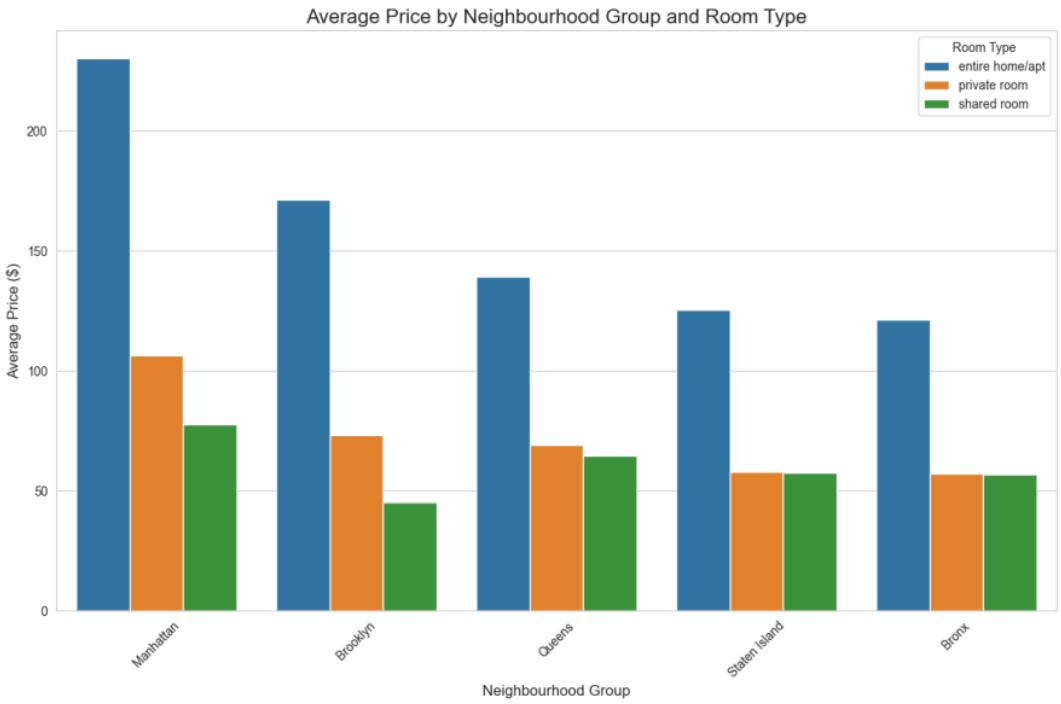
```
+-------+----+--------------------+-----------+-------------------+------------------+--------+---------+---------------+-----+--------
|host_id|  id|                name|  host_name|neighbourhood_group|     neighbourhood|latitude|longitude|      room_type|price|minimum_
+-------+----+--------------------+-----------+-------------------+------------------+--------+---------+---------------+-----+--------
|   2787|2539|Clean & quiet apt...|       John|           Brooklyn|        Kensington|40.64749|-73.97237|   private room|149.0|
|   2845|2595|Skylit Midtown Ca...|   Jennifer|          Manhattan|           Midtown|40.75362|-73.98377|entire home/apt|225.0|
|   4869|3831|Cozy Entire Floor...|LisaRoxanne|           Brooklyn|      Clinton Hill|40.68514|-73.95976|entire home/apt| 89.0|
|   7192|5022|Entire Apt: Spaci...|      Laura|          Manhattan|       East Harlem|40.79851|-73.94399|entire home/apt| 80.0|
|   7322|5099|Large Cozy 1 BR A...|      Chris|          Manhattan|       Murray Hill|40.74767|  -73.975|entire home/apt|200.0|
|   7356|5121|     BlissArtsSpace!|      Garon|           Brooklyn|Bedford-Stuyvesant|40.68688|-73.95596|   private room| 60.0|
|   8967|5178|Large Furnished R...|   Shunichi|          Manhattan|    Hell's Kitchen|40.76489|-73.98493|   private room| 79.0|
|   7490|5203|Cozy Clean Guest ...|  MaryEllen|          Manhattan|   Upper West Side|40.80178|-73.96723|   private room| 79.0|
|   7549|5238|Cute & Cozy Lower...|        Ben|          Manhattan|         Chinatown|40.71344|-73.99037|entire home/apt|150.0|
|   7702|5295|Beautiful 1br on ...|       Lena|          Manhattan|   Upper West Side|40.80316|-73.96545|entire home/apt|135.0|
+-------+----+--------------------+-----------+-------------------+------------------+--------+---------+---------------+-----+--------
only showing top 10 rows
```

Average price for each neighbourhood_group and room type, sorted from highest to lowest:

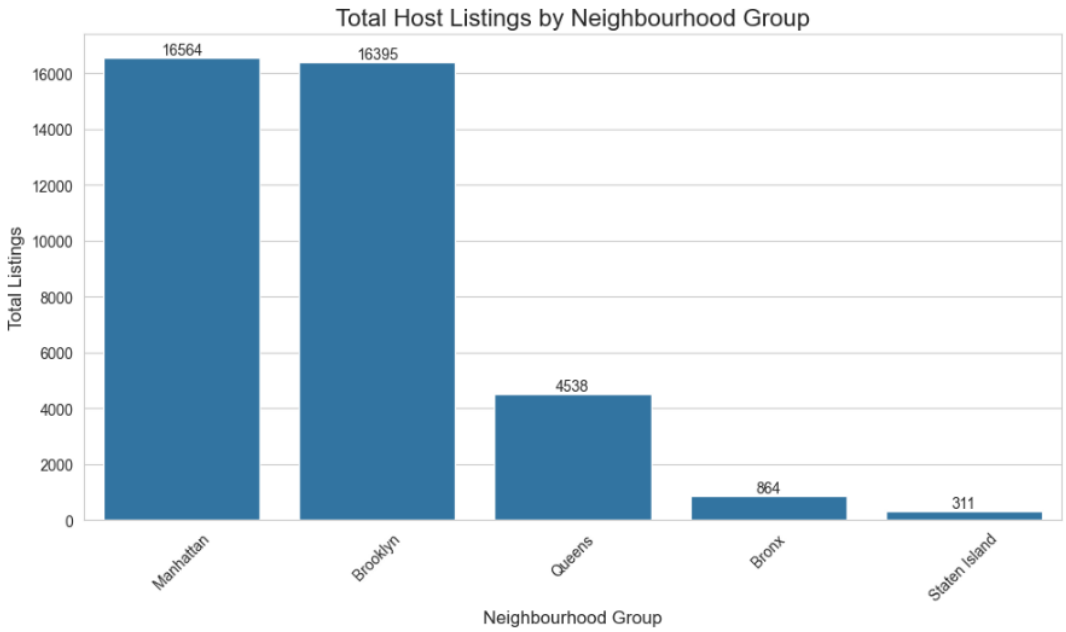
```
+-------------------+---------------+-------------+
|neighbourhood_group|      room_type|average_price|
+-------------------+---------------+-------------+
|          Manhattan|entire home/apt|       230.26|
|           Brooklyn|entire home/apt|       171.31|
|             Queens|entire home/apt|       139.31|
|      Staten Island|entire home/apt|       125.37|
|              Bronx|entire home/apt|       121.33|
|          Manhattan|   private room|       106.63|
|          Manhattan|    shared room|        77.81|
|           Brooklyn|   private room|        73.36|
|             Queens|   private room|        69.27|
|             Queens|    shared room|        64.72|
|      Staten Island|   private room|        57.88|
|      Staten Island|    shared room|         57.6|
|              Bronx|   private room|        57.07|
|              Bronx|    shared room|        56.93|
|           Brooklyn|    shared room|        45.46|
+-------------------+---------------+-------------+
```

Display the Pandas DataFrame:

```
   neighbourhood_group        room_type  average_price
0            Manhattan  entire home/apt         230.26
1             Brooklyn  entire home/apt         171.31
2               Queens  entire home/apt         139.31
3        Staten Island  entire home/apt         125.37
4                Bronx  entire home/apt         121.33
5            Manhattan     private room         106.63
6            Manhattan      shared room          77.81
7             Brooklyn     private room          73.36
8               Queens     private room          69.27
9               Queens      shared room          64.72
10       Staten Island     private room          57.88
11       Staten Island      shared room          57.60
12               Bronx     private room          57.07
13               Bronx      shared room          56.93
14            Brooklyn      shared room          45.46
```

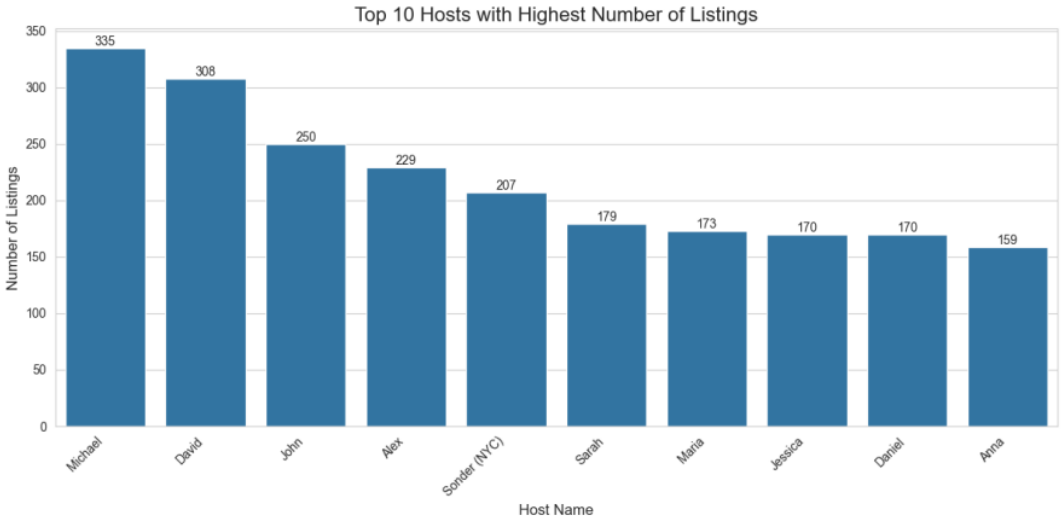



```
Total number of host listings in each neighbourhood group, sorted from highest to lowest:

+-------------------+--------------+
|neighbourhood_group|total_listings|
+-------------------+--------------+
|          Manhattan|         16564|
|           Brooklyn|         16395|
|             Queens|          4538|
|              Bronx|           864|
|      Staten Island|           311|
+-------------------+--------------+

Display the Pandas DataFrame:

  neighbourhood_group  total_listings
0           Manhattan           16564
1            Brooklyn           16395
2              Queens            4538
3               Bronx             864
4       Staten Island             311
```




```
Top 10 hosts with the highest number of listings:

+------------+-------------+
|   host_name|listing_count|
+------------+-------------+
|     Michael|          335|
|       David|          308|
|        John|          250|
|        Alex|          229|
|Sonder (NYC)|          207|
|       Sarah|          179|
|       Maria|          173|
|     Jessica|          170|
|      Daniel|          170|
|        Anna|          159|
+------------+-------------+
```

```
Display the Pandas DataFrame:

     host_name  listing_count
0      Michael            335
1        David            308
2         John            250
3         Alex            229
4  Sonder (NYC)            207
5        Sarah            179
6        Maria            173
7      Jessica            170
8       Daniel            170
9         Anna            159
```

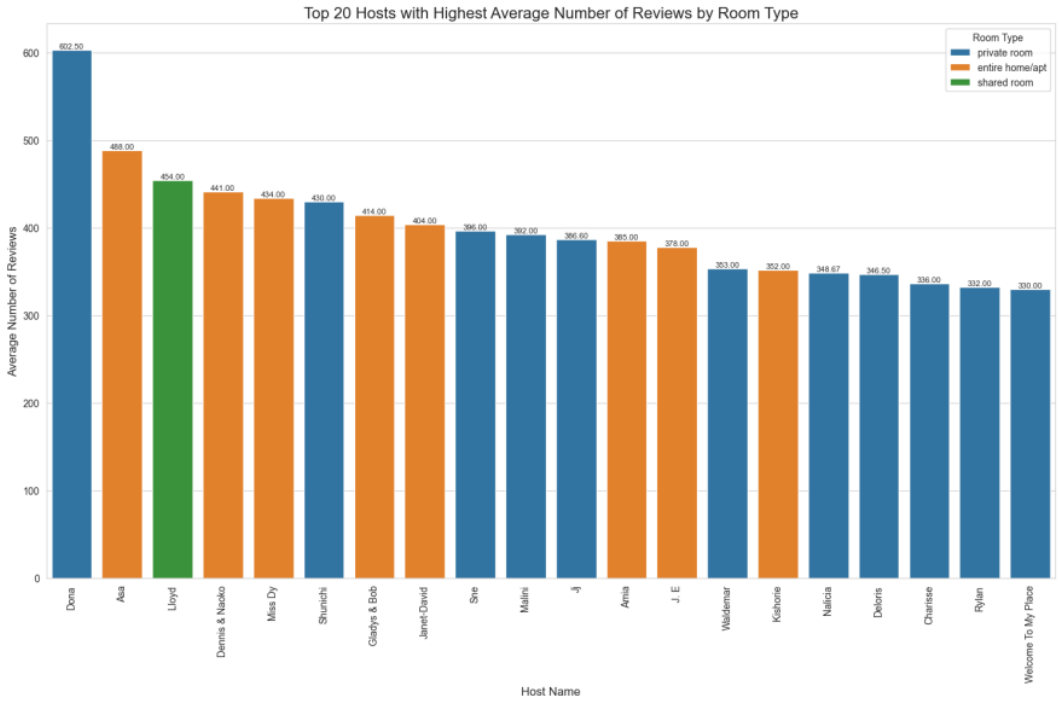

Top 10 Hosts with Highest Number of Listings
Number of Listings
Host Name
0
50
100
150
200
250
300
350
335
308
250
229
207
179
173
170
170
159
Michael
David
John
Alex
Sonder (NYC)
Sarah
Maria
Jessica
Daniel
Anna

Top 20 hosts with the highest average number of reviews by room type:

```
+------------------+---------------+-----------+
|host_name         |room_type      |avg_reviews|
+------------------+---------------+-----------+
|Dona              |private room   |602.5      |
|Asa               |entire home/apt|488.0      |
|Lloyd             |shared room    |454.0      |
|Dennis & Naoko    |entire home/apt|441.0      |
|Miss Dy           |entire home/apt|434.0      |
|Shunichi          |private room   |430.0      |
|Gladys & Bob      |entire home/apt|414.0      |
|Janet-David       |entire home/apt|404.0      |
|Sne               |private room   |396.0      |
|Malini            |private room   |392.0      |
|Jj                |private room   |386.6      |
|Amia              |entire home/apt|385.0      |
|J. E              |entire home/apt|378.0      |
|Waldemar          |private room   |353.0      |
|Kishorie          |entire home/apt|352.0      |
|Nalicia           |private room   |348.67     |
|Deloris           |private room   |346.5      |
|Charisse          |private room   |336.0      |
|Rylan             |private room   |332.0      |
|Welcome To My Place|private room   |330.0      |
+------------------+---------------+-----------+
```

Display the Pandas DataFrame:

	host_name	room_type	avg_reviews
0	Dona	private room	602.50
1	Asa	entire home/apt	488.00
2	Lloyd	shared room	454.00
3	Dennis & Naoko	entire home/apt	441.00
4	Miss Dy	entire home/apt	434.00
5	Shunichi	private room	430.00
6	Gladys & Bob	entire home/apt	414.00
7	Janet-David	entire home/apt	404.00
8	Sne	private room	396.00
9	Malini	private room	392.00
10	Jj	private room	386.60
11	Amia	entire home/apt	385.00
12	J. E	entire home/apt	378.00
13	Waldemar	private room	353.00
14	Kishorie	entire home/apt	352.00
15	Nalicia	private room	348.67
16	Deloris	private room	346.50
17	Charisse	private room	336.00
18	Rylan	private room	332.00
19	Welcome To My Place	private room	330.00



Average Price by Neighbourhood Group and Room Type Side-By-Side Bar Chart Insights:

1st Insight:

The 1st insight is the room type entire home/apartment consistently had the highest average price across all the neighbourhood groups which means that the room type entire home/apartment was the most expensive room type which suggest a higher premiums for the room type entire home/apartment.

2nd Insight:

The 2nd insight is the room type shared rooms consistently had the lowest average price across all the neighbourhood groups which means that the room type shared rooms was the most affordable room type.

Total Host Listings by Neighbourhood Group Bar Chart Insights:

1st Insight:

The 1st insight is the neighbourhood groups Manhatttan and Brooklyn have the most number of Host Listings with 16564 and 16395 listings compared to the other neighbourhood groups which means that Manhattan and Brooklyn have the largest Airbnb Market share.

2nd Insight:

The 2nd insight is the distribution of Airbnb Host Listings data is very unenven across the various neighbourhood groups because Manhattan and Brookyln account for majority of the Airbnb Host Listings while the other neighbourhood groups such as Queens, Bronx and Staten Island only account for a small share of the Airbnb Host Listings.

Top 10 Hosts with Highest Number of Listings Bar Chart Insights:

1st Insight:

The 1st insight is the Michael has the highest number of Airbnb Listings at 335 which means that he is the top host and Michael is followed by David who has the 2nd highest number of Airbnb Listings at 308 which means that he is the 2nd most popular host while Anna has the least number of Airbnb Listings at 159 making her the least popular host.

2nd Insight:

The 2nd insight is the 2 hosts Jessica and Daniel both have the same number of Airbnb Listings at 170 and the data distribution of Airbnb Hosts Listings gradually increases from Anna to Michael.

Top 20 Hosts with Highest Average Number of Reviews by Room Type Bar Chart Insights:

1st Insight:

The 1st insight is the Host Dora has the highest average number of reviews at 602.50 for the Room Type private room while the Host Welcome To My Place has the least average number of reviews at 330.00 for the Room Type private room.

2nd Insight:

The 2nd insight is that 19 out of 20 of the top Hosts with the Highest Average Number of Reviews which means a mass majority are reviewed by the Room Type's private room and home/aprtments with only 1 host Lloyed is in the top 20 reviewed by the Room Type shared room.

Question 3

Question 3(a) (3 marks)

ANS:

```
from pyspark.sql import SparkSession

from pyspark.sql import functions as f

from pyspark.sql.functions import *

from functools import reduce

from builtins import max as py_max

import matplotlib.pyplot as plt

import seaborn as sns

import pandas as pd

import numpy as np

import sys

""" Set the SPARK_LOCAL_IP environment variable: Before running your script, set this environment variable: """
import os
os.environ['SPARK_LOCAL_IP'] = 'localhost'

# Start spark session

""" Set the spark.driver.bindAddress: Add the following configuration to your SparkSession builder:

Use a specific port: If the issue persists, try specifying a port explicitly: """
spark = SparkSession \
```

```
    .builder \
    .appName("ICT337 ECA July 2024 Semester Question 3") \
    .config("spark.driver.bindAddress", "localhost") \
    .config("spark.driver.port", "4043") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()


""" Get the SparkContext from the SparkSession


This line of code retrieves the SparkContext object from a SparkSession:


The SparkContext (sc) is the main entry point for Spark functionality,


allowing you to create RDDs and perform lower-level operations. """
sc = spark.sparkContext




""" Question 3(a) (3 marks) """
print("\nQuestion 3(a) (3 marks)\n")


""" Read the 5-node-graph.txt file and store the content using Spark RDDs.  """
five_node_graph_Spark_RDD = sc.textFile("/Users/shawnyang/Downloads/ICT337 ECA July 2024 Semester/ECA Datasets/5-node-graph.txt")


""" Defines a function to process each line of the input file  """
def parse_line_from_node_graph_file(line):
    """ The line is split into parts using line.split() """
    line_Parts = line.split()


    """ The first part of each line is converted to an integer and assigned as the node_id. """
    line_Node_ID = int( line_Parts[0] )


    """ The second part of each line is converted to an integer and assigned as the distance. """
```

```
line_Distance = int( line_Parts[1] )

""" An empty list neighbors is initialized. """
node_neighbours_List = []

""" Use an if condition to check if there are more than two parts in the line (i.e., if there are neighbors):  """
if len(line_Parts) > 2:

    """ Checks if there are more than 3 parts, loop through each neighbor to split the third part by ':'

    to separate different neighbors. """
    for node_Neighbor in line_Parts[2].split(':'):

        """ Checks if the neighbor string is not empty. """
        if node_Neighbor:

            """ For each neighbor, it splits by ',' to get the neighbor's ID and weight. """
            node_Neighbor_Node_ID, node_Neighbor_Weight  = node_Neighbor.split(',')

            """ The neighbor's ID and weight are converted to integers and

            appended as a tuple to the neighbors list. """
            node_neighbours_List.append(
                ( int(node_Neighbor_Node_ID), int(node_Neighbor_Weight) )
            )

""" A path list is created. If the node_id is 1, it contains [1], otherwise it's empty. """

""" An empty list called path is initialized. """
path_List = []

""" Checks if the node_id is equal to 1. """
if line_Node_ID == 1:
    """ If the condition is true, it appends the node_id to the path list. """
```

```
        path_List.append(line_Node_ID)


    """ Finally, it returns a tuple containing the node_id and another tuple with distance, neighbors, and path. """
    return ( line_Node_ID, (line_Distance, node_neighbours_List, path_List) )


""" Uses map() to apply the parse_line() function to each line, creating the required RDD structure.

Apply the parse_line function to each line in five_node_graph_rdd. """
node_graph_RDD_Five = five_node_graph_Spark_RDD.map(parse_line_from_node_graph_file)


print("5-node-graph.txt RDD Content with the structure (node_ID,(distance, list of neighbors with associated weight, path)):\n")


""" Show the 5-node-graph.txt input file RDD content


Collect all elements from the distributed graph_rdd into the driver program as a list and starts iterating over them. """
for node_graph_RDD_Five_Contents in node_graph_RDD_Five.collect():


    """ For each item in the collected list, it prints the item to the console. """
    print(node_graph_RDD_Five_Contents)
```

Output:

```
Question 3(a) (3 marks)

5-node-graph.txt RDD Content with the structure (node_ID,(distance, list of neighbors with associated weight, path)):

(1, (0, [(2, 10), (3, 5)], [1]))
(2, (10000, [(3, 2), (4, 1)], []))
(3, (10000, [(2, 3), (4, 9), (5, 2)], []))
(4, (10000, [(5, 4)], []))
(5, (10000, [(1, 7), (4, 6)], []))
```

Question 3(b) (8 marks)

ANS:

```
""" Question 3(b) (8 marks) """
print("\nQuestion 3(b) (8 marks)\n")

""" This function defines a custom minimum finder to replace Python's built-in min() function,

which may not be available in certain Spark environments. """
def custom_Minimum_Finder(iterable, key=None):

    """ Checks if the iterable is empty. If so, it raises a ValueError. """
    if not iterable:
        raise ValueError("iterable is empty")



    """ We define a new function called identity that simply returns its input. """
    def identity(x):
        return x

    """ If no key function is provided, it uses the identity function to key (returns the item itself). """
    if key is None:
        key = identity

    """ Initializes the minimum item and its value with the first item in the iterable. """
    mininmum_Item = iterable[0]

    minimum_Value = key(mininmum_Item)

    """ Iterates through the remaining items, updating min_item and min_value if a smaller value is found. """
    for item in iterable[1:]:
        value = key(item)


        if value < minimum_Value:
```

```
            mininmum_Item = item
            minimum_Value = value

    """ Returns the item with the minimum value. """
    return mininmum_Item

""" This is the main function implementing Dijkstra's algorithm. It takes a graph RDD as input. """
def dijkstra_Algorithm(node_graph_RDD_Five):

    """ This inner function updates the distances for neighboring nodes. """
    def update_Distances_For_Neighbouring_Nodes(node):
        """ It unpacks the node information. """
        node_ID, (neighbour_Distance, node_Neighbours, node_Path) = node

        """ It creates an empty list to store the updated neighbor information. """
        updated_Neighbours = []

        """ It iterates through each neighbor and its weight. """
        for node_neighbour, node_weight in node_Neighbours:

            """ For each neighbor, it calculates the new distance and creates a new path. """
            new_Neighbour_Distance = neighbour_Distance + node_weight

            new_Neighbour_Path = node_Path + [node_neighbour]

            """ It appends the updated information for each neighbor to the list. """
            updated_Neighbours.append((node_neighbour, (new_Neighbour_Distance, new_Neighbour_Path)))

        """  Finally, it returns the list of updated neighbors. """
        return updated_Neighbours

    """ This inner function chooses the shortest path between two options.

    This function compares the first element (index 0) of two tuples a and b,
```

```
which represent the distances of two paths.  """
def select_Shortest_Path(a, b):
    if a[0] < b[0]:
        """  It returns the tuple with the smaller distance, effectively selecting the shorter path. """
        return a
    else:
        return b

""" Initializes the visited set with the starting node (1)

and the result list with the starting node's information. """
visited_set_with_Starting_Node = set([1])

shortest_Distance = {1: 0}

shortest_Path = {1: [1]}

""" This is the main loop of the algorithm, continuing until all nodes have been visited. """
while len(visited_set_with_Starting_Node) < node_graph_RDD_Five.count():

    """ We define a new function is_node_unvisited that takes a node tuple

    and returns True if the node has not been visited yet. """
    def nodes_yet_to_be_Reached(unreached_Node_Tuple):
        unreached_Node_ID = unreached_Node_Tuple[0]

        return unreached_Node_ID not in visited_set_with_Starting_Node

    """ This part finds candidate nodes to visit next, filtering out already visited nodes

    and choosing the shortest path to each unvisited node. """
    next_Candidate_Reachable_Nodes = node_graph_RDD_Five \
        .flatMap(update_Distances_For_Neighbouring_Nodes) \
```

```
            .filter(nodes_yet_to_be_Reached) \
                .reduceByKey(select_Shortest_Path) \
                    .collect()


    """ Finally, if there are no more candidate reachable nodes, the loop breaks. """
    if not next_Candidate_Reachable_Nodes:
        break


    """ We define a new function get_node_distance that takes a node tuple and returns the distance value

    which is the first element (index 0) of the second element (index 1) of the tuple. """
    def get_Next_Candidate_Reachable_Node_Distance(candidate_Node_Tuple):
        return candidate_Node_Tuple[1][0]



    """ Replace the min() function with the custom minimum finder


    Selects the next node to visit based on the shortest distance. """
    next_Reachable_Node, (neighbour_Distance, node_Path) = custom_Minimum_Finder(next_Candidate_Reachable_Nodes,
key=get_Next_Candidate_Reachable_Node_Distance)



    """ We add the next reachable node to the visited set"""
    visited_set_with_Starting_Node.add(next_Reachable_Node)


    """ Update the visited set shortest distance and path in the respective dictionaries.  """
    shortest_Distance[next_Reachable_Node] = neighbour_Distance


    shortest_Path[next_Reachable_Node] = node_Path


    """ Update the graph RDD with the new distance and path information


    We define a separate function called update_node_info. This function takes a node as input and returns the
updated node information."""
```

```
    def update_Shortest_Path_Node_Info(node):

        """ Extract the node_id and its current information (distance, neighbors, and path). """
        node_ID = node[0]

        current_Shortest_Distance, neighbor_Node, current_Shortest_Path = node[1]

        """ Check if we have a new shortest distance for this node in the shortest_Distance dictionary. """
        if node_ID in shortest_Distance:
            """ If we do, we use the new distance and path; otherwise, we keep the current ones. """
            new_Shortest_distance = shortest_Distance[node_ID]

            new_Shortest_Path = shortest_Path[node_ID]

        else:
            new_Shortest_distance = current_Shortest_Distance

            new_Shortest_Path = current_Shortest_Path

        """ Return the updated node information. """
        return (node_ID, (new_Shortest_distance, neighbor_Node, new_Shortest_Path))

    """ We then use this update_node_info function with the map operation on our RDD to update all nodes. """
    node_graph_RDD_Five = node_graph_RDD_Five.map(update_Shortest_Path_Node_Info)

""" Initialize an empty result list """
dijkstra_Algorithm_Results_List = []

"""  We iterate through the sorted keys of shortest_Distance """
for node_ID in sorted(shortest_Distance.keys()):

    """ Create a tuple for each node with its shortest distance and path, and append it to the result list. """
    dijkstra_Algorithm_Results_List.append((node_ID, (shortest_Distance[node_ID], shortest_Path[node_ID])))
```

```
    return dijkstra_Algorithm_Results_List


# Run the algorithm

shortest_Paths_List = dijkstra_Algorithm(node_graph_RDD_Five)


# Format and print the results

print("Show the iterative Dijkstra's algorithm RDD Content with the structure (node_ID, (shortest path distance, path traversal)):\n")


""" Prints the final result, showing the shortest distance and path for each node. """

for node_ID, (shortest_Distance, shortest_Path) in shortest_Paths_List:


    path_string = '→'.join(map(str, shortest_Path))


    print(f"Node {node_ID}: (Shortest distance: {shortest_Distance}, Path: {path_string})")
```

Output:

```
Question 3(b) (8 marks)

Show the iterative Dijkstra's algorithm RDD Content with the structure (node_ID, (shortest path distance, path traversal)):

Node 1: (Shortest distance: 0, Path: 1)
Node 2: (Shortest distance: 8, Path: 1→3→2)
Node 3: (Shortest distance: 5, Path: 1→3)
Node 4: (Shortest distance: 9, Path: 1→3→2→4)
Node 5: (Shortest distance: 7, Path: 1→3→5)
```

Question 3(c) (8 marks)

.**ANS:**

Based on the Question 3(a) and Question 3(b) source codes and outputs, this is how the shortest path computations are performed using each iteration step:

Iteration 1:

Begin at Node 1 which is the Source Node.

The path distance from Node 1 is 0 and the path is just itself which is [1].

The Dijkstra algorithm now checks the path distances between node 1 and its unvisited neighbours which are Node 2 and Node 3

- Node 2: path distance = 0 + 10 = 10, path = [1, 2]
  This means that the path distance to Node 2 is 10.

  So the new path distance from Node 1 to Node 2 is 0 + 10 = 10.

  The **path = [1, 2]** means that the path begins at Node 1 and ends at Node 2.

- Node 3: path distance = 0 + 5 = 5, path [1, 3]
  This means that the path distance to Node 3 is 5.

  So the new path distance from Node 1 to Node 3 is 0 + 5 = 5.

  The **path = [1, 3]** means that the path begins at Node 1 and ends at Node 3.

The path distance from Node 1 to its neighbours Node 2 is 10 and to Node 3 is 5.

Select Node 3 as the next node to visit with the path [1, 3] because it has the shortest distance of 5.

Iteration 2:

From Node 3,

The Dijkstra algorithm now checks the path distances between Node 3 and its unvisited neighbours which are Node 2, Node 4 and Node 5.

- **Node 2: path distance = (min(10, 5 + 3) = 8), path = [1, 2]**
  The existing path distance to Node 2 is 10 (1st iteration)

  The new path distance is 5 (path distance to Node 3) + 3 (weight from Node 2 to Node 3) = 8

  The algorithm opts for the minimum distance between either 10 or 8 and 8 is the shortest distance.

- **Node 4: path distance = 5 + 9 = 14, path = [1, 3, 4]**
  This means that the path distance to Node 3 is 5 and the weight from Node 3 to Node 4 is 9.

  So the new path distance from Node 3 to Node 4 is 5 + 9 = 14.

  The **path = [1, 3, 4]** means that the path begins at Node 1, moves to Node 3 and ends at Node 4.

- **Node 5: path distance = 5 + 2 = 7, path = [1, 3, 5]**

This means that the path distance to Node 3 is 5 and the weight from Node 3 to Node 5 is 2.

So the new path distance from Node 3 to Node 2 is 5 + 2 = 7.

The **path = [1, 3, 5]** means that the path begins at Node 1, moves to Node 3 and ends at Node 5.

Select Node 5 as the next node to visit with the path [1, 3, 5] because it has the shortest distance of 7.

Iteration 3:

From Node 5,

The Dijkstra algorithm now checks the path distances between Node 5 and its unvisited neighbours which are Node 2 and Node 4.

- **Node 4: path distance = (min(14, 7 + 6) = 13), path = [1, 3, 4]**
  The existing path distance to Node 4 is 14 (2nd iteration)

  The new path distance is 7 (path distance to Node 5) + 6 (weight from Node 5 to Node 4) = 13

  The algorithm opts for the minimum distance between either 14 or 13 and 13 is the shortest distance.

Select Node 2 as the next node to visit with the path [1, 2] because it is the closest unvisited node with a path distance of 8.

Iteration 4:

From Node 2,

The Dijkstra algorithm now checks the path distances between Node 2 and its unvisited neighbours which is Node 4.

- **Node 4: path distance = (min(13, 8 + 1) = 9), path = [1, 3, 4]**
  The existing path distance to Node 4 is 13 (3rd iteration)

  The new path distance is 8 (path distance to Node 2) + 1 (weight from Node 2 to Node 4) = 9

  The algorithm opts for the minimum distance between either 13 or 9 and 9 is the shortest distance.

Select Node 4 as the next node to visit with the path [1, 3, 4] because it is the last unvisited node.

The Dijkstra algorithm stops running after all nodes have been visited.

The condition to break from the infinite while loop executes when there are 0 unvisited nodes.

Question 3(d) (4 marks)

ANS:

```
""" Question 3(d) (4 marks) """
print("\nQuestion 3(d) (4 marks)\n")

""" This is the main function implementing Dijkstra's algorithm. It takes a graph RDD as input. """
def dijkstra_Algorithm(node_graph_RDD_Five):

    """ This inner function updates the distances for neighboring nodes. """
    def update_Distances_For_Neighbouring_Nodes(node):
        """ It unpacks the node information. """
        node_ID, (neighbour_Distance, node_Neighbours, node_Path) = node

        """ It creates an empty list to store the updated neighbor information. """
        updated_Neighbours = []

        """ It iterates through each neighbor and its weight. """
        for node_neighbour, node_weight in node_Neighbours:

            """ For each neighbor, it calculates the new distance and creates a new path. """
            new_Neighbour_Distance = neighbour_Distance + node_weight

            new_Neighbour_Path = node_Path + [node_neighbour]

            """ It appends the updated information for each neighbor to the list. """
            updated_Neighbours.append((node_neighbour, (new_Neighbour_Distance, new_Neighbour_Path)))
```

```
    """  Finally, it returns the list of updated neighbors. """
    return updated_Neighbours

""" This inner function chooses the shortest path between two options.

This function compares the first element (index 0) of two tuples a and b,

which represent the distances of two paths.  """
def select_Shortest_Path(a, b):
    if a[0] < b[0]:
        """  It returns the tuple with the smaller distance, effectively selecting the shorter path. """
        return a
    else:
        return b

""" Initializes the visited set with the starting node (1)

and the result list with the starting node's information. """
visited_set_with_Starting_Node = set([1])

shortest_Distance = {1: 0}

shortest_Path = {1: [1]}

number_of_Iterations = 0

""" This is the main loop of the algorithm, continuing until all nodes have been visited. """
while len(visited_set_with_Starting_Node) < node_graph_RDD_Five.count():

    number_of_Iterations = number_of_Iterations + 1

    """ We define a new function is_node_unvisited that takes a node tuple

    and returns True if the node has not been visited yet. """
```

```
    def nodes_yet_to_be_Reached(unreached_Node_Tuple):
        unreached_Node_ID = unreached_Node_Tuple[0]


        return unreached_Node_ID not in visited_set_with_Starting_Node


    """ This part finds candidate nodes to visit next, filtering out already visited nodes


    and choosing the shortest path to each unvisited node. """
    next_Candidate_Reachable_Nodes = node_graph_RDD_Five \
        .flatMap(update_Distances_For_Neighbouring_Nodes) \
            .filter(nodes_yet_to_be_Reached) \
                .reduceByKey(select_Shortest_Path) \
                    .collect()


    """ Finally, if there are no more candidate reachable nodes, the loop breaks. """
    if not next_Candidate_Reachable_Nodes:
        break


    """ We define a new function get_node_distance that takes a node tuple and returns the distance value


    which is the first element (index 0) of the second element (index 1) of the tuple. """
    def get_Next_Candidate_Reachable_Node_Distance(candidate_Node_Tuple):
        return candidate_Node_Tuple[1][0]



    """ Replace the min() function with the custom minimum finder


    Selects the next node to visit based on the shortest distance. """
    next_Reachable_Node, (neighbour_Distance, node_Path) = custom_Minimum_Finder(next_Candidate_Reachable_Nodes,
key=get_Next_Candidate_Reachable_Node_Distance)



    """ We add the next reachable node to the visited set"""
```

```
        visited_set_with_Starting_Node.add(next_Reachable_Node)


        """ Update the visited set shortest distance and path in the respective dictionaries.  """
        shortest_Distance[next_Reachable_Node] = neighbour_Distance


        shortest_Path[next_Reachable_Node] = node_Path



        """ Update the graph RDD with the new distance and path information

    We define a separate function called update_node_info. This function takes a node as input and returns the updated node information."""
        def update_Shortest_Path_Node_Info(node):


            """ Extract the node_id and its current information (distance, neighbors, and path). """
            node_ID = node[0]


            current_Shortest_Distance, neighbor_Node, current_Shortest_Path = node[1]


            """ Check if we have a new shortest distance for this node in the shortest_Distance dictionary. """
            if node_ID in shortest_Distance:
                """ If we do, we use the new distance and path; otherwise, we keep the current ones. """
                new_Shortest_distance = shortest_Distance[node_ID]


                new_Shortest_Path = shortest_Path[node_ID]


            else:
                new_Shortest_distance = current_Shortest_Distance


                new_Shortest_Path = current_Shortest_Path


            """ Return the updated node information. """
            return (node_ID, (new_Shortest_distance, neighbor_Node, new_Shortest_Path))
```

```
        """ We then use this update_node_info function with the map operation on our RDD to update all nodes. """
        node_graph_RDD_Five = node_graph_RDD_Five.map(update_Shortest_Path_Node_Info)

    """ Initialize an empty result list """
    dijkstra_Algorithm_Results_List = []

    """  We iterate through the sorted keys of shortest_Distance """
    for node_ID in sorted(shortest_Distance.keys()):

        """ Create a tuple for each node with its shortest distance and path, and append it to the result list. """
        dijkstra_Algorithm_Results_List.append((node_ID, (shortest_Distance[node_ID], shortest_Path[node_ID])))

    return dijkstra_Algorithm_Results_List, number_of_Iterations

""" This line runs the Dijkstra's algorithm on the input graph RDD

and returns the shortest paths and the number of iterations it took to complete."""
shortest_Paths_List, num_of_Shortest_Path_Iterations = dijkstra_Algorithm(node_graph_RDD_Five)

print(f"Number of iterations to complete the shortest path computation for 5-node-graph.txt: {num_of_Shortest_Path_Iterations}")

print("\nFinal output (node_ID, (shortest path distance, path traversal)), sorted by ascending node_ID:\n")

""" The for loop iterates through the shortest_paths result:

This unpacks each result into node ID, distance, and path. """
for node_ID, (shortest_Path_Distance, shortest_Path) in shortest_Paths_List:

    """ This prints the formatted output for each node including, node ID,

    shortest distance to that node from the start node and path to reach that node,

    formatted as a string with arrows (→) between node numbers  """
```

```
    print(f"(Node {node_ID}: (Shortest Path Distance: {shortest_Path_Distance}, Path Traversal: {'→'.join(map(str, shortest_Path))}))")
```

Output:

```
Question 3(d) (4 marks)

Number of iterations to complete the shortest path computation for 5-node-graph.txt: 4

Final output (node_ID, (shortest path distance, path traversal)), sorted by ascending node_ID:

(Node 1: (Shortest Path Distance: 0, Path Traversal: 1))
(Node 2: (Shortest Path Distance: 8, Path Traversal: 1→3→2))
(Node 3: (Shortest Path Distance: 5, Path Traversal: 1→3))
(Node 4: (Shortest Path Distance: 9, Path Traversal: 1→3→2→4))
(Node 5: (Shortest Path Distance: 7, Path Traversal: 1→3→5))
```

Question 3(e) (9 marks)

ANS:

```
""" Question 3(e) (9 marks) """
print("\nQuestion 3(e) (9 marks)\n")


""" This line runs the Dijkstra's algorithm on the input graph RDD


and returns the shortest paths and the number of iterations it took to complete."""
shortest_Paths_List, num_of_Shortest_Path_Iterations = dijkstra_Algorithm(node_graph_RDD_Five)


""" We define a new function get_node_distance that takes a node tuple and returns the distance value


which is the first element (index 0) of the second element (index 1) of the tuple. """
def get_Candidate_Reachable_Node_Distance(candidate_Node_Tuple):
    return candidate_Node_Tuple[1][0]


""" Sort the paths by distance in descending order


We pass this function as the key argument to sorted.


This tells the sorting function to use the distance value when comparing paths.
```

```
The reverse=True argument ensures that the paths are sorted in descending order of distance. """
sorted_shortest_Paths_List = sorted(shortest_Paths_List, key=get_Candidate_Reachable_Node_Distance, reverse=True)

""" Get the top 3 furthermost nodes """
top_3_Furthermost_Shortest_Paths_Nodes = sorted_shortest_Paths_List[:3]

print("Top Three (3) furthermost nodes, their paths & distances (sorted by descending distance):\n")

for node_ID, (shortest_Path_Distance, shortest_Path) in top_3_Furthermost_Shortest_Paths_Nodes:
    print(f"Node {node_ID}: (Distance: {shortest_Path_Distance}, Path: {'→'.join(map(str, shortest_Path))})")



""" This line runs the Dijkstra's algorithm on the input graph RDD

and returns the shortest paths and the number of iterations it took to complete."""
shortest_Paths_List, num_of_Shortest_Path_Iterations = dijkstra_Algorithm(node_graph_RDD_Five)

""" We define a new function that takes a path tuple and returns the length of the path.

The function unpacks the tuple to access the path, then returns its length. """
def get_Shortest_Paths_Length(path_tuple):
    _, (_, path) = path_tuple

    return len(path)

""" Find the path with the most hops

We use the py_max() function to apply get_path_length to each item in shortest_Paths_List.

The py_max() function is then used to find the maximum length among all paths.

Using py_max() instead of max() to find the maximum number of hops. """
maximum_Num_of_Shortest_Paths_Hops = py_max(map(get_Shortest_Paths_Length, shortest_Paths_List))
```

```
""" We initialize an empty list nodes_with_max_hops to store the results. """
shortest_Paths_Nodes_With_Maximum_Hops_List = []

""" Find all nodes with paths equal to max_hops

We iterate through each item in shortest_Paths_List, unpacking it into node, distance, and path. """
for node_ID, (shortest_Path_Distance, shortest_Path) in shortest_Paths_List:

    """ For each item, we check if the length of the path is equal to maximum_Num_of_Shortest_Paths_Hops. """
    if len(shortest_Path) == maximum_Num_of_Shortest_Paths_Hops:
        """ If the condition is true, we append a tuple containing the node, distance,

        and path to nodes_with_max_hops. """
        shortest_Paths_Nodes_With_Maximum_Hops_List.append((node_ID, (shortest_Path_Distance, shortest_Path)))

print("\nDestination node(s) with the most number of traversal hops in the path:\n")

for node_ID, (shortest_Path_Distance, shortest_Path) in shortest_Paths_Nodes_With_Maximum_Hops_List:
    print(f"Node {node_ID}: (Distance: {shortest_Path_Distance}, Path: {'→'.join(map(str, shortest_Path))}, Hops: {len(shortest_Path) - 1})")



""" This line runs the Dijkstra's algorithm on the input graph RDD

and returns the shortest paths and the number of iterations it took to complete."""
shortest_Paths_List, num_of_Shortest_Path_Iterations = dijkstra_Algorithm(node_graph_RDD_Five)

""" Get all node IDs from the graph

It gets all node IDs from the graph using graph_rdd.keys().collect(). """
get_Set_of_Node_ID_From_All_Graph_Nodes = set(node_graph_RDD_Five.keys().collect())

""" We initialize an empty set get_Set_of_Reachable_Graph_Nodes to store the reachable nodes. """
```

```
get_Set_of_Reachable_Graph_Nodes_Set = set()


""" Get the set of reachable nodes

It creates a set of reachable nodes by checking which nodes have a finite distance in shortest_paths.

We iterate through each item in shortest_Paths_List, unpacking it into node, distance, and path. """
for node_ID, (shortest_Path_Distance, shortest_Path) in shortest_Paths_List:

    """ For each item, we check if the distance is not equal to infinity (float('inf')). """
    if shortest_Path_Distance != float('inf'):
        """ If the condition is true, we add the node to the get_Set_of_Reachable_Graph_Nodes set. """
        get_Set_of_Reachable_Graph_Nodes_Set.add(node_ID)


""" Find unreachable nodes

It finds unreachable nodes by subtracting the set of reachable nodes from the set of all nodes. """
unreachable_Graph_Nodes = get_Set_of_Node_ID_From_All_Graph_Nodes -
get_Set_of_Reachable_Graph_Nodes_Set


""" Sort unreachable nodes by ascending node_ID

It sorts the unreachable nodes in ascending order. """
sorted_by_Ascending_Unreachable_Graph_Nodes_List = sorted(unreachable_Graph_Nodes)


print("\nNodes that are not reachable from source node (node_ID=1), sorted by ascending node_ID:\n")


""" It then checks if there are any unreachable nodes in sorted_by_Ascending_Unreachable_Graph_Nodes.

If there are unreachable nodes: """
if sorted_by_Ascending_Unreachable_Graph_Nodes_List:

    """ It iterates through each node in sorted_by_Ascending_Unreachable_Graph_Nodes """
    for unreachable_Nodes_From_Source_Node in sorted_by_Ascending_Unreachable_Graph_Nodes_List:
```

```
        """ For each unreachable node, it prints "Node {node}" """
        print(f"Node {unreachable_Nodes_From_Source_Node}")


else:
    """ If there are no unreachable nodes:

    It prints the statement """
    print("All nodes are reachable from the Node 1.")
```

Output:

```
Question 3(e) (9 marks)

Top Three (3) furthermost nodes, their paths & distances (sorted by descending distance):

Node 4: (Distance: 9, Path: 1→3→2→4)
Node 2: (Distance: 8, Path: 1→3→2)
Node 5: (Distance: 7, Path: 1→3→5)

Destination node(s) with the most number of traversal hops in the path:

Node 4: (Distance: 9, Path: 1→3→2→4, Hops: 3)

Nodes that are not reachable from source node (node_ID=1), sorted by ascending node_ID:

All nodes are reachable from the Node 1.
```

Question 3(f) (8 marks)

ANS:

```
""" Question 3(f) (8 marks) """
print("\nQuestion 3(f) (8 marks)\n")


print("Computations and Reslts for 20-node-graph.txt:\n")


""" Question 3(a) Section: """


""" Read the 20-node-graph.txt file and store the content using Spark RDDs. """
twenty_node_graph_Spark_RDD = sc.textFile("/Users/shawnyang/Downloads/ICT337 ECA July 2024 Semester/ECA Datasets/20-node-graph.txt")
```

```
""" Uses map() to apply the parse_line() function to each line, creating the required RDD structure. """
node_graph_RDD_Twenty = twenty_node_graph_Spark_RDD.map(parse_line_from_node_graph_file)


print("20-node-graph.txt RDD Content with the structure (node_ID,(distance, list of neighbors with associated weight, path)):\n")


""" Show the 5-node-graph.txt input file RDD content


Prints the content of the RDD using collect() and a for loop. """
for node_graph_RDD_Twenty_Contents in node_graph_RDD_Twenty.collect():
    print(node_graph_RDD_Twenty_Contents)


""" Question 3(d) Section: """


# Run Dijkstra's algorithm
twenty_Node_Graph_RDD_Shortest_Paths_Lists, twenty_Node_Graph_RDD_num_of_Shortest_Path_Iterations = dijkstra_Algorithm(node_graph_RDD_Twenty)


print(f"\nNumber of iterations to complete the shortest path computation for 20-node-graph.txt: {twenty_Node_Graph_RDD_num_of_Shortest_Path_Iterations}")


print("\nFinal output (node_ID, (shortest path distance, path traversal)), sorted by ascending node_ID:\n")


for node_ID, (shortest_Path_Distance, shortest_Path) in twenty_Node_Graph_RDD_Shortest_Paths_Lists:
    print(f"Node {node_ID}: (Shortest distance: {shortest_Path_Distance}, Path: {'→'.join(map(str, shortest_Path))})")


""" Question 3(e) Section: """


""" We define a new function get_node_distance that takes a node tuple and returns the distance value


which is the first element (index 0) of the second element (index 1) of the tuple. """
def get_Candidate_Reachable_Node_Distance(candidate_Node_Tuple):
    return candidate_Node_Tuple[1][0]


""" Sort the paths by distance in descending order
```

We pass this function as the key argument to sorted.

This tells the sorting function to use the distance value when comparing paths.

The reverse=True argument ensures that the paths are sorted in descending order of distance. """

twenty_Sorted_Shortest_Paths_List = sorted(twenty_Node_Graph_RDD_Shortest_Paths_Lists, key=get_Candidate_Reachable_Node_Distance, reverse=True)

""" Get the top 3 furthermost nodes """

twenty_Node_Top_3_Furthermost_Shortest_Paths_Nodes = twenty_Sorted_Shortest_Paths_List[:3]

print("\n20-node-graph.txt Top Three (3) furthermost nodes, their paths & distances (sorted by descending distance):\n")

for node_ID, (shortest_Path_Distance, shortest_Path) in twenty_Node_Top_3_Furthermost_Shortest_Paths_Nodes:

    print(f"Node {node_ID}: (Distance: {shortest_Path_Distance}, Path: {'→'.join(map(str, shortest_Path))})")

""" Find the path with the most hops

We use the py_max() function to apply get_path_length to each item in shortest_Paths_List.

The py_max() function is then used to find the maximum length among all paths.

Using py_max() instead of max() to find the maximum number of hops. """

maximum_Num_of_Shortest_Paths_Hops = py_max(map(get_Shortest_Paths_Length, twenty_Sorted_Shortest_Paths_List))

""" We initialize an empty list nodes_with_max_hops to store the results. """

twenty_Node_Shortest_Paths_With_Maximum_Hops_List = []

""" Find all nodes with paths equal to max_hops

We iterate through each item in shortest_Paths_List, unpacking it into node, distance, and path. """

```
for node_ID, (shortest_Path_Distance, shortest_Path) in twenty_Sorted_Shortest_Paths_List:


    """ For each item, we check if the length of the path is equal to maximum_Num_of_Shortest_Paths_Hops. """
    if len(shortest_Path) == maximum_Num_of_Shortest_Paths_Hops:
        """ If the condition is true, we append a tuple containing the node, distance,


        and path to nodes_with_max_hops. """
        twenty_Node_Shortest_Paths_With_Maximum_Hops_List.append((node_ID, (shortest_Path_Distance,
shortest_Path)))


print("\n20-node-graph.txt Destination node(s) with the most number of traversal hops in the path:\n")


for node_ID, (shortest_Path_Distance, shortest_Path) in twenty_Node_Shortest_Paths_With_Maximum_Hops_List:
    print(f"Node {node_ID}: (Distance: {shortest_Path_Distance}, Path: {'→'.join(map(str, shortest_Path))}, Hops:
{len(shortest_Path) - 1})")



""" Get all node IDs from the graph


It gets all node IDs from the graph using graph_rdd.keys().collect(). """
get_Set_of_Node_ID_From_All_Twenty_Nodes = set(node_graph_RDD_Twenty.keys().collect())


""" We initialize an empty set get_Set_of_Reachable_Graph_Nodes to store the reachable nodes. """
get_Set_of_Reachable_Twenty_Nodes_Set = set()


""" Get the set of reachable nodes


It creates a set of reachable nodes by checking which nodes have a finite distance in shortest_paths.


We iterate through each item in shortest_Paths_List, unpacking it into node, distance, and path. """
for node_ID, (shortest_Path_Distance, shortest_Path) in twenty_Node_Graph_RDD_Shortest_Paths_Lists:


    """ For each item, we check if the distance is not equal to infinity (float('inf')). """
    if shortest_Path_Distance != float('inf'):
```

```
        """ If the condition is true, we add the node to the get_Set_of_Reachable_Graph_Nodes set. """
        get_Set_of_Reachable_Twenty_Nodes_Set.add(node_ID)


""" Find unreachable nodes

It finds unreachable nodes by subtracting the set of reachable nodes from the set of all nodes. """
twenty_Node_Unreachable_Nodes_List = sorted(set( get_Set_of_Node_ID_From_All_Twenty_Nodes - get_Set_of_Reachable_Twenty_Nodes_Set ))


""" Sort unreachable nodes by ascending node_ID

It sorts the unreachable nodes in ascending order. """
sorted_by_Ascending_Twenty_Node_Unreachable_Nodes_List = sorted(twenty_Node_Unreachable_Nodes_List)


print("\nNodes that are not reachable from source node (node_ID=1), sorted by ascending node_ID:\n")


""" It then checks if there are any unreachable nodes in sorted_by_Ascending_Unreachable_Graph_Nodes.

If there are unreachable nodes: """
if sorted_by_Ascending_Twenty_Node_Unreachable_Nodes_List:


    """ It iterates through each node in sorted_by_Ascending_Unreachable_Graph_Nodes """
    for unreachable_Twenty_Nodes_From_Source_Node in sorted_by_Ascending_Twenty_Node_Unreachable_Nodes_List:
        """ For each unreachable node, it prints "Node {node}" """
        print(f"Node {unreachable_Twenty_Nodes_From_Source_Node}")


else:
    """ If there are no unreachable nodes:


    It prints the statement """
    print("All nodes are reachable from the Node 1.")
```

```
print("\n\n\nComputations and Results for 40-node-graph.txt:\n")


""" Question 3(a) Section: """


""" Read the 40-node-graph.txt file and store the content using Spark RDDs. """

fourty_node_graph_Spark_RDD = sc.textFile("/Users/shawnyang/Downloads/ICT337 ECA July 2024 Semester/ECA Datasets/40-node-graph.txt")


""" Uses map() to apply the parse_line() function to each line, creating the required RDD structure. """

node_graph_RDD_Fourty = fourty_node_graph_Spark_RDD.map(parse_line_from_node_graph_file)


print("40-node-graph.txt RDD Content with the structure (node_ID,(distance, list of neighbors with associated weight, path)):\n")


""" Show the 40-node-graph.txt input file RDD content


Prints the content of the RDD using collect() and a for loop. """

for item in node_graph_RDD_Fourty.collect():

    print(item)


""" Question 3(d) Section: """


# Run Dijkstra's algorithm

fourty_Node_Graph_RDD_Shortest_Paths_Lists, fourty_Node_Graph_RDD_num_of_Shortest_Path_Iterations = dijkstra_Algorithm(node_graph_RDD_Fourty)


print(f"\nNumber of iterations to complete the shortest path computation for 20-node-graph.txt: {fourty_Node_Graph_RDD_num_of_Shortest_Path_Iterations}")


print("\nFinal output (node_ID, (shortest path distance, path traversal)), sorted by ascending node_ID:\n")


for node_ID, (shortest_Path_Distance, shortest_Path) in fourty_Node_Graph_RDD_Shortest_Paths_Lists:

    print(f"Node {node_ID}: (Shortest distance: {shortest_Path_Distance}, Path: {'→'.join(map(str, shortest_Path))})")


""" Question 3(e) Section: """
```

```
""" We define a new function get_node_distance that takes a node tuple and returns the distance value

which is the first element (index 0) of the second element (index 1) of the tuple. """
def get_Candidate_Reachable_Node_Distance(candidate_Node_Tuple):
    return candidate_Node_Tuple[1][0]

""" Sort the paths by distance in descending order

We pass this function as the key argument to sorted.

This tells the sorting function to use the distance value when comparing paths.

The reverse=True argument ensures that the paths are sorted in descending order of distance. """
fourty_Sorted_Shortest_Paths_List = sorted(fourty_Node_Graph_RDD_Shortest_Paths_Lists, key=get_Candidate_Reachable_Node_Distance, reverse=True)

""" Get the top 3 furthermost nodes """
fourty_Node_Top_3_Furthermost_Shortest_Paths_Nodes = fourty_Sorted_Shortest_Paths_List[:3]

print("\n20-node-graph.txt Top Three (3) furthermost nodes, their paths & distances (sorted by descending distance):\n")

for node_ID, (shortest_Path_Distance, shortest_Path) in fourty_Node_Top_3_Furthermost_Shortest_Paths_Nodes:
    print(f"Node {node_ID}: (Distance: {shortest_Path_Distance}, Path: {'→'.join(map(str, shortest_Path))})")



""" Find the path with the most hops

We use the py_max() function to apply get_path_length to each item in shortest_Paths_List.

The py_max() function is then used to find the maximum length among all paths.

Using py_max() instead of max() to find the maximum number of hops. """
```

```
maximum_Num_of_Shortest_Paths_Hops = py_max(map(get_Shortest_Paths_Length, fourty_Sorted_Shortest_Paths_List))


""" We initialize an empty list nodes_with_max_hops to store the results. """

fourty_Node_Shortest_Paths_With_Maximum_Hops_List = []


""" Find all nodes with paths equal to max_hops


We iterate through each item in shortest_Paths_List, unpacking it into node, distance, and path. """

for node_ID, (shortest_Path_Distance, shortest_Path) in fourty_Sorted_Shortest_Paths_List:


    """ For each item, we check if the length of the path is equal to maximum_Num_of_Shortest_Paths_Hops. """

    if len(shortest_Path) == maximum_Num_of_Shortest_Paths_Hops:

        """ If the condition is true, we append a tuple containing the node, distance,


        and path to nodes_with_max_hops. """

        fourty_Node_Shortest_Paths_With_Maximum_Hops_List.append((node_ID, (shortest_Path_Distance, shortest_Path)))


print("\n20-node-graph.txt Destination node(s) with the most number of traversal hops in the path:\n")


for node_ID, (shortest_Path_Distance, shortest_Path) in fourty_Node_Shortest_Paths_With_Maximum_Hops_List:

    print(f"Node {node_ID}: (Distance: {shortest_Path_Distance}, Path: {'→'.join(map(str, shortest_Path))}, Hops: {len(shortest_Path) - 1})")




""" Get all node IDs from the graph


It gets all node IDs from the graph using graph_rdd.keys().collect(). """

get_Set_of_Node_ID_From_All_Fourty_Nodes = set(node_graph_RDD_Fourty.keys().collect())


""" We initialize an empty set get_Set_of_Reachable_Graph_Nodes to store the reachable nodes. """

get_Set_of_Reachable_Fourty_Nodes_Set = set()


""" Get the set of reachable nodes
```

```
It creates a set of reachable nodes by checking which nodes have a finite distance in shortest_paths.


We iterate through each item in shortest_Paths_List, unpacking it into node, distance, and path. """
for node_ID, (shortest_Path_Distance, shortest_Path) in fourty_Node_Graph_RDD_Shortest_Paths_Lists:


    """ For each item, we check if the distance is not equal to infinity (float('inf')). """
    if shortest_Path_Distance != float('inf'):
        """ If the condition is true, we add the node to the get_Set_of_Reachable_Graph_Nodes set. """
        get_Set_of_Reachable_Fourty_Nodes_Set.add(node_ID)


""" Find unreachable nodes

It finds unreachable nodes by subtracting the set of reachable nodes from the set of all nodes. """
fourty_Node_Unreachable_Nodes_List = sorted(set( get_Set_of_Node_ID_From_All_Fourty_Nodes - get_Set_of_Reachable_Fourty_Nodes_Set ))


""" Sort unreachable nodes by ascending node_ID

It sorts the unreachable nodes in ascending order. """
sorted_by_Ascending_Fourty_Node_Unreachable_Nodes_List = sorted(fourty_Node_Unreachable_Nodes_List)


print("\nNodes that are not reachable from source node (node_ID=1), sorted by ascending node_ID:\n")


""" It then checks if there are any unreachable nodes in sorted_by_Ascending_Unreachable_Graph_Nodes.

If there are unreachable nodes: """
if sorted_by_Ascending_Fourty_Node_Unreachable_Nodes_List:


    """ It iterates through each node in sorted_by_Ascending_Unreachable_Graph_Nodes """
    for unreachable_Fourty_Nodes_From_Source_Node in sorted_by_Ascending_Fourty_Node_Unreachable_Nodes_List:
        """ For each unreachable node, it prints "Node {node}" """
        print(f"Node {unreachable_Fourty_Nodes_From_Source_Node}")
```

```
else:
    """ If there are no unreachable nodes:

    It prints the statement """
    print("All nodes are reachable from the Node 1.")
```

Output:

```
Question 3(f) (8 marks)

Computations and Reslts for 20-node-graph.txt:

20-node-graph.txt RDD Content with the structure (node_ID,(distance, list of neighbors with associated weight, path)):

(1, (0, [(14, 18), (11, 15), (10, 13)], [1]))
(2, (10000, [(10, 6)], []))
(3, (10000, [(9, 7)], []))
(4, (10000, [(10, 20), (9, 3)], []))
(5, (10000, [(3, 15), (16, 20), (8, 16), (13, 11)], []))
(6, (10000, [(19, 13)], []))
(7, (10000, [(11, 5), (18, 5)], []))
(8, (10000, [(7, 17), (9, 18)], []))
(9, (10000, [(18, 19), (3, 6)], []))
(10, (10000, [(11, 8), (14, 2)], []))
(11, (10000, [(14, 20), (3, 16), (7, 14)], []))
(12, (10000, [(1, 19), (9, 18), (2, 2)], []))
(13, (10000, [(14, 15), (5, 15)], []))
(14, (10000, [(15, 19)], []))
(15, (10000, [(9, 9)], []))
(16, (10000, [(4, 6), (13, 10)], []))
(17, (10000, [(18, 5), (15, 16)], []))
(18, (10000, [(1, 8), (10, 6), (17, 20), (19, 3)], []))
(19, (10000, [(3, 13)], []))
(20, (10000, [(12, 17), (7, 6), (1, 12)], []))

Number of iterations to complete the shortest path computation for 20-node-graph.txt: 18
```

```
Final output (node_ID, (shortest path distance, path traversal)), sorted by ascending node_ID:

Node 1: (Shortest distance: 0, Path: 1)
Node 2: (Shortest distance: 10002, Path: 2)
Node 3: (Shortest distance: 31, Path: 1→11→3)
Node 4: (Shortest distance: 10006, Path: 4)
Node 5: (Shortest distance: 10025, Path: 13→5)
Node 7: (Shortest distance: 29, Path: 1→11→7)
Node 8: (Shortest distance: 10016, Path: 8)
Node 9: (Shortest distance: 38, Path: 1→11→3→9)
Node 10: (Shortest distance: 13, Path: 1→10)
Node 11: (Shortest distance: 15, Path: 1→11)
Node 12: (Shortest distance: 10017, Path: 12)
Node 13: (Shortest distance: 10010, Path: 13)
Node 14: (Shortest distance: 15, Path: 1→10→14)
Node 15: (Shortest distance: 34, Path: 1→10→14→15)
Node 16: (Shortest distance: 10020, Path: 16)
Node 17: (Shortest distance: 54, Path: 1→11→7→18→17)
Node 18: (Shortest distance: 34, Path: 1→11→7→18)
Node 19: (Shortest distance: 37, Path: 1→11→7→18→19)

20-node-graph.txt Top Three (3) furthermost nodes, their paths & distances (sorted by descending distance):

Node 5: (Distance: 10025, Path: 13→5)
Node 16: (Distance: 10020, Path: 16)
Node 12: (Distance: 10017, Path: 12)

20-node-graph.txt Destination node(s) with the most number of traversal hops in the path:

Node 17: (Distance: 54, Path: 1→11→7→18→17, Hops: 4)
Node 19: (Distance: 37, Path: 1→11→7→18→19, Hops: 4)

Nodes that are not reachable from source node (node_ID=1), sorted by ascending node_ID:

Node 6
Node 20
```

```
Computations and Reslts for 40-node-graph.txt:

40-node-graph.txt RDD Content with the structure (node_ID,(distance, list of neighbors with associated weight, path)):

(1, (0, [(13, 5)], [1]))
(2, (10000, [(5, 3), (20, 5), (37, 13)], []))
(3, (10000, [(24, 19)], []))
(4, (10000, [(5, 2), (2, 6)], []))
(5, (10000, [(2, 10), (13, 9), (33, 7)], []))
(6, (10000, [(20, 9), (22, 15)], []))
(7, (10000, [(9, 5)], []))
(8, (10000, [(27, 12), (4, 13), (30, 11), (29, 20)], []))
(9, (10000, [(14, 5), (24, 9)], []))
(10, (10000, [(36, 14)], []))
(11, (10000, [(21, 11), (39, 16)], []))
(12, (10000, [(10, 12), (18, 17), (28, 14), (2, 13)], []))
(13, (10000, [(36, 17), (8, 7), (17, 7)], []))
(14, (10000, [(31, 12), (36, 3), (38, 12), (39, 8)], []))
(15, (10000, [(7, 11), (25, 9), (30, 16)], []))
(16, (10000, [(11, 4), (4, 9), (23, 19), (34, 20)], []))
(17, (10000, [(10, 14), (24, 10), (14, 16), (37, 17)], []))
(18, (10000, [(9, 17)], []))
(19, (10000, [(25, 20), (3, 5)], []))
(20, (10000, [(18, 19), (7, 4)], []))
(21, (10000, [(20, 1), (13, 4), (12, 14)], []))
(22, (10000, [(27, 8)], []))
(23, (10000, [(20, 4)], []))
(24, (10000, [(23, 7), (21, 13), (1, 19)], []))
(25, (10000, [(39, 1), (36, 8)], []))
(26, (10000, [(38, 8), (29, 17), (23, 1)], []))
(27, (10000, [(7, 18), (15, 11)], []))
(28, (10000, [(33, 13), (27, 14), (6, 7)], []))
(29, (10000, [(21, 13), (16, 3), (40, 2)], []))
(30, (10000, [(20, 3), (30, 3)], []))
(31, (10000, [(33, 2), (27, 18)], []))
(32, (10000, [(18, 10), (6, 18), (13, 18), (30, 15)], []))
(33, (10000, [(18, 19), (3, 12), (5, 18)], []))
(34, (10000, [(2, 5)], []))
(35, (10000, [(38, 16)], []))
(36, (10000, [(15, 12)], []))
(37, (10000, [(18, 13), (26, 8), (5, 17)], []))
(38, (10000, [(23, 9), (8, 17), (38, 2), (18, 6)], []))
(39, (10000, [(22, 17)], []))
(40, (10000, [(34, 4), (28, 1), (3, 1), (27, 8)], []))

Number of iterations to complete the shortest path computation for 40-node-graph.txt: 37
```

```
Final output (node_ID, (shortest path distance, path traversal)), sorted by ascending node_ID:

Node 1: (Shortest distance: 0, Path: 1)
Node 2: (Shortest distance: 31, Path: 1→13→8→4→2)
Node 3: (Shortest distance: 35, Path: 1→13→8→29→40→3)
Node 4: (Shortest distance: 25, Path: 1→13→8→4)
Node 5: (Shortest distance: 27, Path: 1→13→8→4→5)
Node 6: (Shortest distance: 42, Path: 1→13→8→29→40→28→6)
Node 7: (Shortest distance: 30, Path: 1→13→8→30→20→7)
Node 8: (Shortest distance: 12, Path: 1→13→8)
Node 9: (Shortest distance: 35, Path: 1→13→8→30→20→7→9)
Node 10: (Shortest distance: 26, Path: 1→13→17→10)
Node 11: (Shortest distance: 39, Path: 1→13→8→29→16→11)
Node 12: (Shortest distance: 49, Path: 1→13→17→24→21→12)
Node 13: (Shortest distance: 5, Path: 1→13)
Node 14: (Shortest distance: 28, Path: 1→13→17→14)
Node 15: (Shortest distance: 34, Path: 1→13→36→15)
Node 16: (Shortest distance: 35, Path: 1→13→8→29→16)
Node 17: (Shortest distance: 12, Path: 1→13→17)
Node 18: (Shortest distance: 42, Path: 1→13→17→37→18)
Node 20: (Shortest distance: 26, Path: 1→13→8→30→20)
Node 21: (Shortest distance: 35, Path: 1→13→17→24→21)
Node 22: (Shortest distance: 53, Path: 1→13→17→14→39→22)
Node 23: (Shortest distance: 29, Path: 1→13→17→24→23)
Node 24: (Shortest distance: 22, Path: 1→13→17→24)
Node 25: (Shortest distance: 43, Path: 1→13→36→15→25)
Node 26: (Shortest distance: 37, Path: 1→13→17→37→26)
Node 27: (Shortest distance: 24, Path: 1→13→8→27)
Node 28: (Shortest distance: 35, Path: 1→13→8→29→40→28)
Node 29: (Shortest distance: 32, Path: 1→13→8→29)
Node 30: (Shortest distance: 23, Path: 1→13→8→30)
Node 31: (Shortest distance: 40, Path: 1→13→17→14→31)
Node 33: (Shortest distance: 34, Path: 1→13→8→4→5→33)
Node 34: (Shortest distance: 38, Path: 1→13→8→29→40→34)
Node 36: (Shortest distance: 22, Path: 1→13→36)
Node 37: (Shortest distance: 29, Path: 1→13→17→37)
Node 38: (Shortest distance: 40, Path: 1→13→17→14→38)
Node 39: (Shortest distance: 36, Path: 1→13→17→14→39)
Node 40: (Shortest distance: 34, Path: 1→13→8→29→40)
```

```
40-node-graph.txt Top Three (3) furthermost nodes, their paths & distances (sorted by descending distance):

Node 22: (Distance: 53, Path: 1→13→17→14→39→22)
Node 12: (Distance: 49, Path: 1→13→17→24→21→12)
Node 25: (Distance: 43, Path: 1→13→36→15→25)

40-node-graph.txt Destination node(s) with the most number of traversal hops in the path:

Node 6: (Distance: 42, Path: 1→13→8→29→40→28→6, Hops: 6)
Node 9: (Distance: 35, Path: 1→13→8→30→20→7→9, Hops: 6)

Nodes that are not reachable from source node (node_ID=1), sorted by ascending node_ID:

Node 19
Node 32
Node 35
```

----- END OF ECA PAPER -----

ICT337_ECA_B2110802_YANGXIANWEISHAWN_30102024.do...

ORIGINALITY REPORT

9%	5%	3%	6%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to Sim University Student Paper	1%
2	Submitted to University of Hertfordshire Student Paper	1%
3	Submitted to University of North Texas Student Paper	<1%
4	Submitted to Coventry University Student Paper	<1%
5	victoromondi1997.github.io Internet Source	<1%
6	Submitted to Queen's University of Belfast Student Paper	<1%
7	Balogh, András. "Model Transformation-Based Design of Dependable Systems", Budapest University of Technology and Economics (Hungary), 2024 Publication	<1%
8	Submitted to City University Student Paper	<1%

9	www.differencebetween.com Internet Source	<1%
10	Submitted to Kaplan College Student Paper	<1%
11	Submitted to Victoria University Student Paper	<1%
12	Hang T. Lau. "A Java Library of Graph Algorithms and Optimization", Chapman and Hall/CRC, 2019 Publication	<1%
13	Submitted to Instituto de Empress S.L. Student Paper	<1%
14	Submitted to University of Greenwich Student Paper	<1%
15	docs.openiap.io Internet Source	<1%
16	www.analyticsvidhya.com Internet Source	<1%
17	Wenzhen Lin, Fan Yang. "The Price of Short-Term Housing: A Study of Airbnb on 26 Regions in the United States", Journal of Housing Economics, 2024 Publication	<1%
18	Moshe Sniedovich. "Dynamic Programming - Foundations and Principles, Second Edition",	<1%

CRC Press, 2019
Publication

19	Submitted to ESC Rennes Student Paper	<1 %
20	Submitted to University of Auckland Student Paper	<1 %
21	community.databricks.com Internet Source	<1 %
22	simonegli.com Internet Source	<1 %
23	blog.csdn.net Internet Source	<1 %
24	Submitted to Old Dominion University Student Paper	<1 %
25	gitter.im Internet Source	<1 %
26	www.coursehero.com Internet Source	<1 %
27	Submitted to Lincoln University Student Paper	<1 %
28	Lawrence, John Charles. "TARDISS, Exploring the Potential for a Research Surveillance System in Secondary Medical Data Research", The Ohio State University, 2024 Publication	<1 %

29	Submitted to Glasgow Caledonian University Student Paper	<1 %
30	Submitted to Hult International Business School, Inc. Student Paper	<1 %
31	ir.library.knu.ua Internet Source	<1 %
32	harvard-iacs.github.io Internet Source	<1 %
33	www.skillshare.com Internet Source	<1 %
34	vineetkarandikar.medium.com Internet Source	<1 %
35	git.odin.cse.buffalo.edu Internet Source	<1 %
36	Deep Medhi, Karthik Ramasamy. "Routing Protocols: Framework and Principles", Elsevier BV, 2018 Publication	<1 %
37	Submitted to FFHS Fernfachhochschule Schweiz Student Paper	<1 %
38	Submitted to University of Edinburgh Student Paper	<1 %

39	Submitted to University of Technology, Sydney Student Paper	<1%
40	datascience.stackexchange.com Internet Source	<1%
41	notebook.community Internet Source	<1%
42	usermanual.wiki Internet Source	<1%
43	www.geeksforgeeks.org Internet Source	<1%

Exclude quotes	Off	Exclude matches	Off
Exclude bibliography	Off		