Ong Yeow Hwee, Glen

Singapore University of Social Science

ICT337: Big Data Computing in the Cloud

Prof. Hu He

October 24, 2023

**Question 1**

**(a)**

**Premise**

Both PySpark DataFrame and PySpark Resilient Distributed Datasets (RDD) are abstractions offered by Apache Spark for distributed data processing. They act as APIs that give Spark framework users the ability to interact with and manipulate data. These two fundamental components of Spark provide diverse methods for distributed data processing, each with unique advantages and applications. In this project, I will discuss the concepts of PySpark RDD and PySpark DataFrame by exploring their respective characteristics, transformations, actions and use cases.

***Resilient Distributed Datasets (RDD)***

**Definition and Characteristics.** The main data structure used by Apache Spark is RDD. It is a set of immutable, fault-tolerant components that can be handled concurrently by a cluster of machines. RDDs make the physical properties of the data, such as partitions, accessible to users, allowing them to understand how the data is processed and distributed in a distributed computing environment. RDDs also use lineage graph to keep track of the data transformations performed, ensuring resiliency and the capacity to recompute missing or damaged partitions in the event of node failures.

**Transformations and Actions.** When performing transformations, a new RDD is produced by applying a function to the elements of an existing RDD. Common transformations such as "map" and "filter" follow the lazy evaluation paradigm, which means they postpone execution and create a chain of transformations. This strategy enables Spark to effectively optimize the execution plan. The crucial points where Spark executes the computations are represented by actions. Actions like "collect" and "reduce" trigger the previously defined lineage of transformations to be evaluated, which computes the desired result.

**Use Case.** RDDs are best suited for situations in which precise, low-level control over data processing is essential. They excel in scenarios involving semi-structed or unstructured data sources. In situations requiring specialized data manipulation and the execution of challenging, non-standard data processing workflows, RDDs truly shine. They are a versatile option for developers taking on complex data processing challenges due to their flexibility and capacity to handle complex data operations.

***DataFrames***

**Definition and Characteristics.** As a higher-level abstraction built on top of RDDs, DataFrames in PySpark handles structured data effectively. A DataFrame is essentially a distributed collection of named columns and rows that is immutable and distributed across a cluster of machines. The ability of Spark to automatically understand the data's structure and optimize query execution for better performance is one of the key benefits of DataFrames. DataFrames’ user-friendly and structured approach to data manipulation is especially well-suited for a variety of data processing tasks, especially those involving structured data sources.

**Transformations and Actions.** DataFrames provide a more extensive array of high-level operations, enhancing their usability. Transformation operations such as “select” and “groupBy” enable users to manipulate data with ease while adhering to a lazy evaluation paradigm, which postpones execution and creates a chain of transformations. On the other hand, action operations like “count” and “show” serve as triggers, initiating the execution of transformations and returning valuable insights. This combination of rich high-level operations and efficient execution mechanisms makes DataFrames an appealing choice for various data processing tasks, particularly those involving structured data sources.

**Use Case.** When it comes to processing structured data, DataFrames becomes the favored solution for a variety of data processing tasks. They offer streamlined operations for data exploration and wrangling in scenarios involving structured data formats like CSV, Parquet, or Avro, where their effectiveness really shines. The unique feature of DataFrames is their built-in compatibility with SQL-like queries, which gives users a simple way to query their data. Additionally, the Catalyst optimizer's inclusion guarantees automatic optimization of query execution plans, improving performance and making DataFrames particularly advantageous for tasks that require both structured data processing and query optimization.

Abstraction level Low-level, more control over the pre-processing part High-level, rich semantics

**(b)**

|  |  |  |
| --- | --- | --- |
| **Characteristic** | **Resilient Distributed Datasets (RDD)** | **DataFrames** |
| Abstraction level | Low-level, more control over pre-processing | High-level, rich semantics |
| Data structure | Unstructured | Structured (tabular) |
| Schema | No schema | Schema with named columns |
| Typing | No type enforcement | Strong typed |
| Catalyst optimizer | Not applicable | Optimizes query execution plans |
| Serialization | User-defined | Built-in |
| Fault-tolerance | Basic (requires lineage) | Built-in |
| Spark SQL | No | Yes |
| Transformations | ‘map’, ‘reduce’, custom transformations | ‘select’, ‘filter’, SQL queries, joins |
| Ease of use | More complex | More user-friendly |

*Table 1.1. Differences between PySpark RDD and PySpark Dataframes.*

**Question 2**

**(a)**

***Premise***

My local machine operates in windows, it is advantageous to create an isolated environment for running Apache Spark. The VMware Workstation 17 Player is downloaded, installed with the ubuntu iso, and launched (see Figure 1 and Figure 2). Subsequently, the following processes are applied within the VMware environment.

A screenshot of a computer

Description automatically generated

*Figure 1. Creating a Virtual machine in VMware with Ubuntu iso image.*

A screenshot of a computer

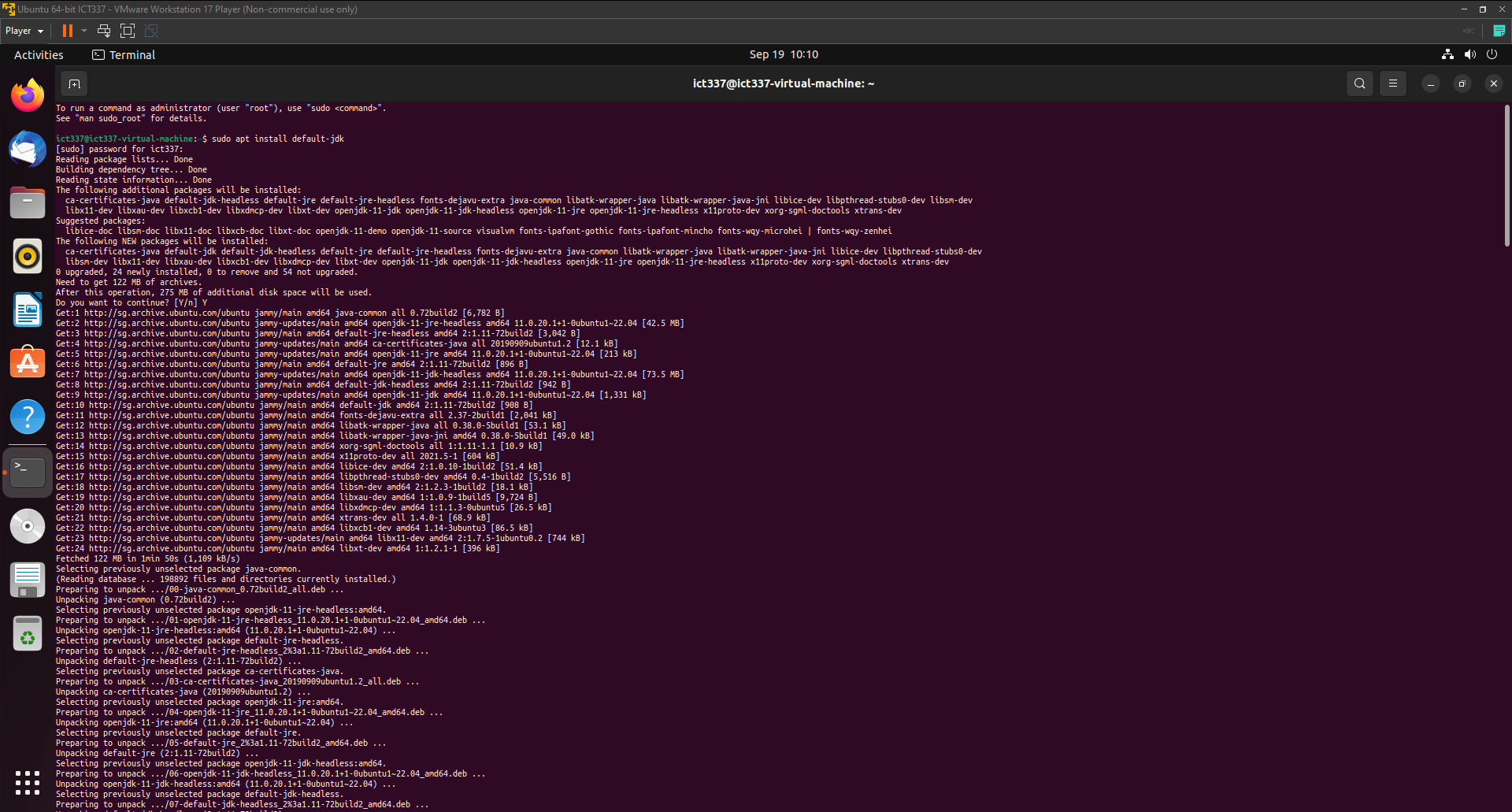
Description automatically generated

*Figure 2. Virtual machine configuration in VMware.*

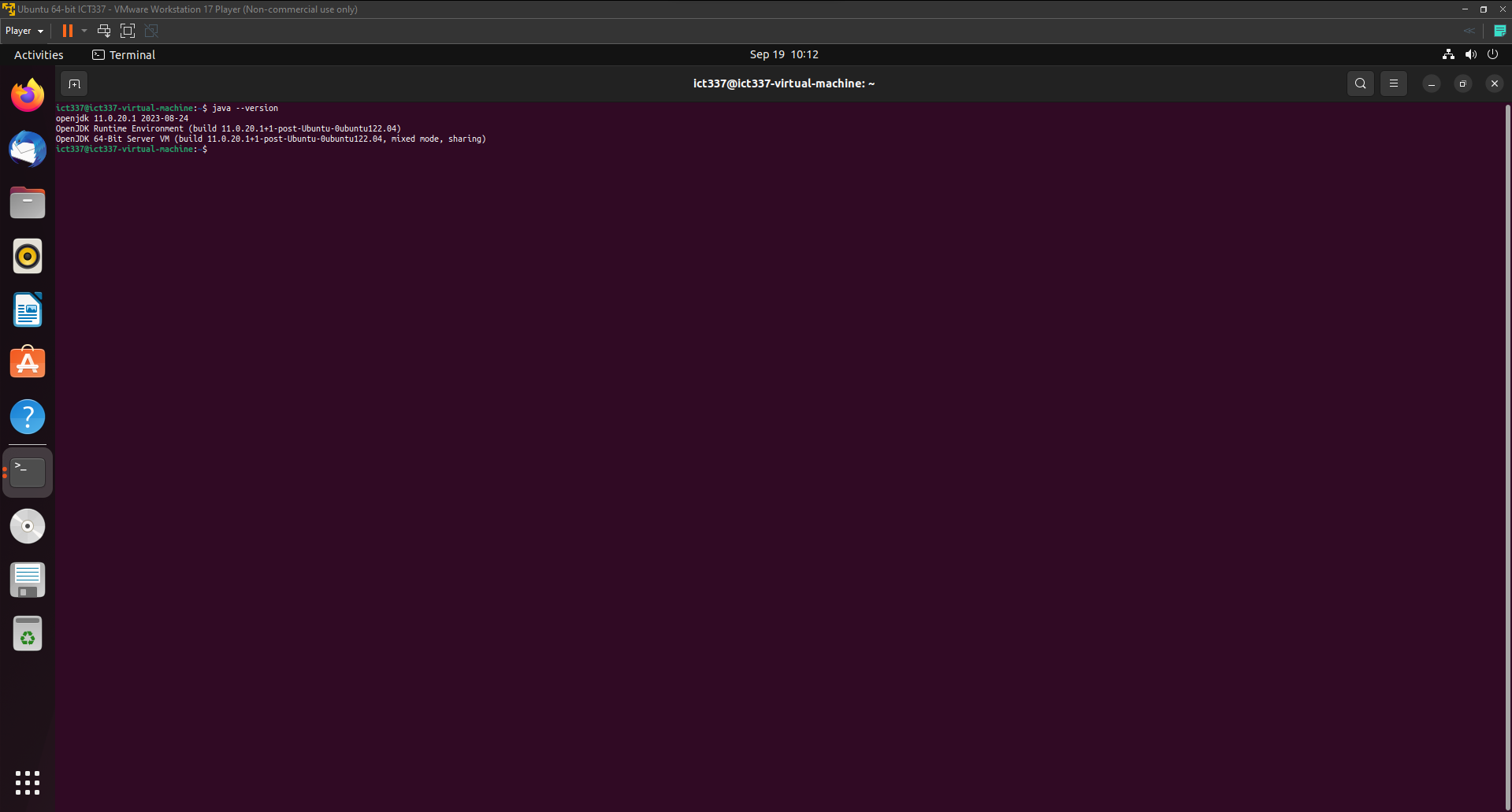
***Prerequisites***

Apache Spark requires Java Development Kit (1) and Python (2). In my case, I chose to download Apache Spark version 3.4.1 and the pre-built for Apache Hadoop package type from Apache Sparks official website (3).

Firstly, ‘sudo apt install default-jdk’ command is used in the terminal to download the Java Development Kit prerequisite and the ‘java –version’ command is used to verify if the Java has been installed accordingly (see Figure 3 and Figure 4).

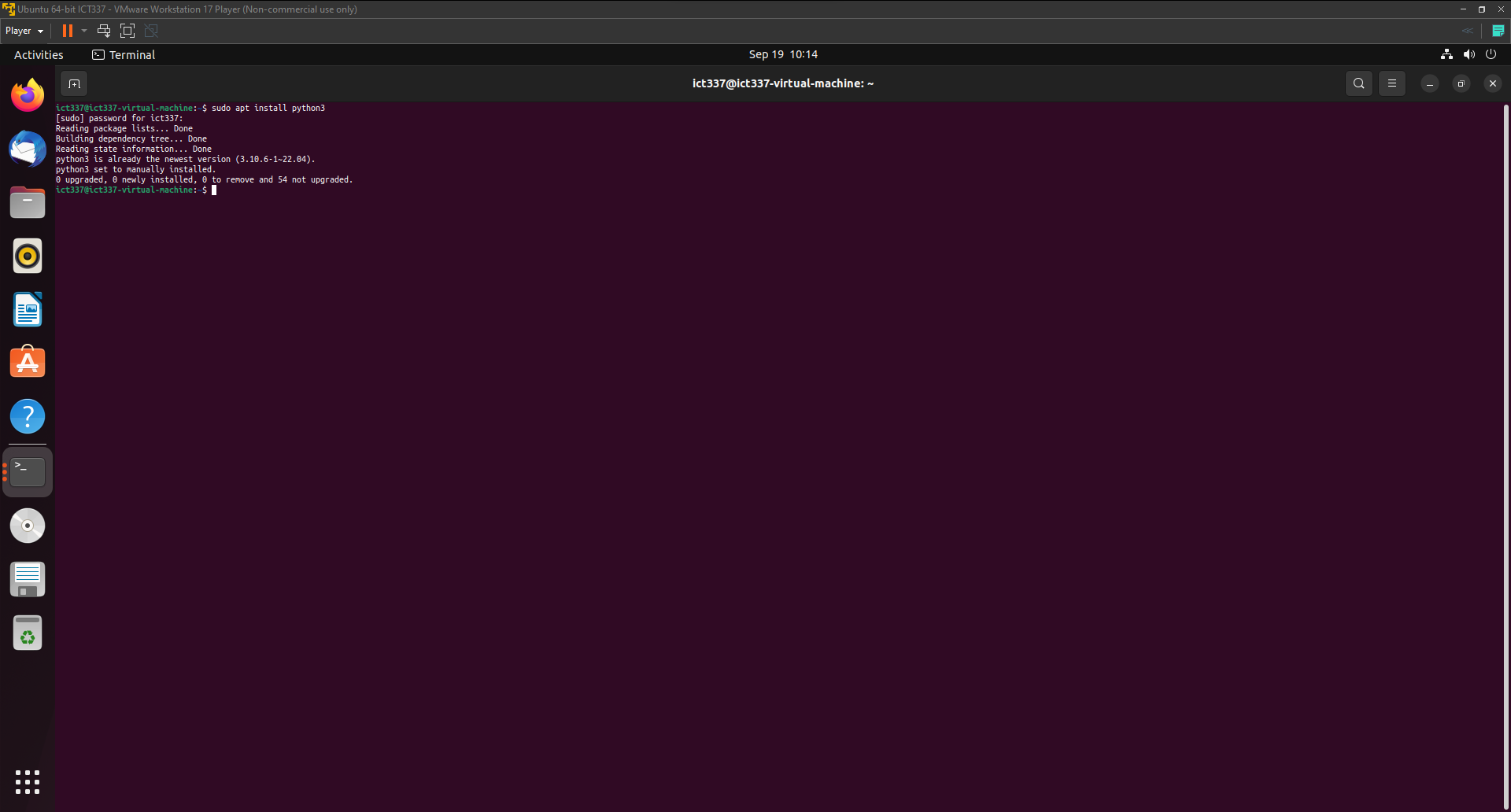


*Figure 3. Sample Java Development Kit installation in Ubuntu terminal.*



*Figure 4. Java Development Kit installation verification in Ubuntu terminal.*

Secondly, ‘sudo apt install python3’ command is used in the terminal to download the Python prerequisite and the ‘python3 –version’ command is used to verify if the Python has been installed accordingly (see Figure 5 and Figure 6).



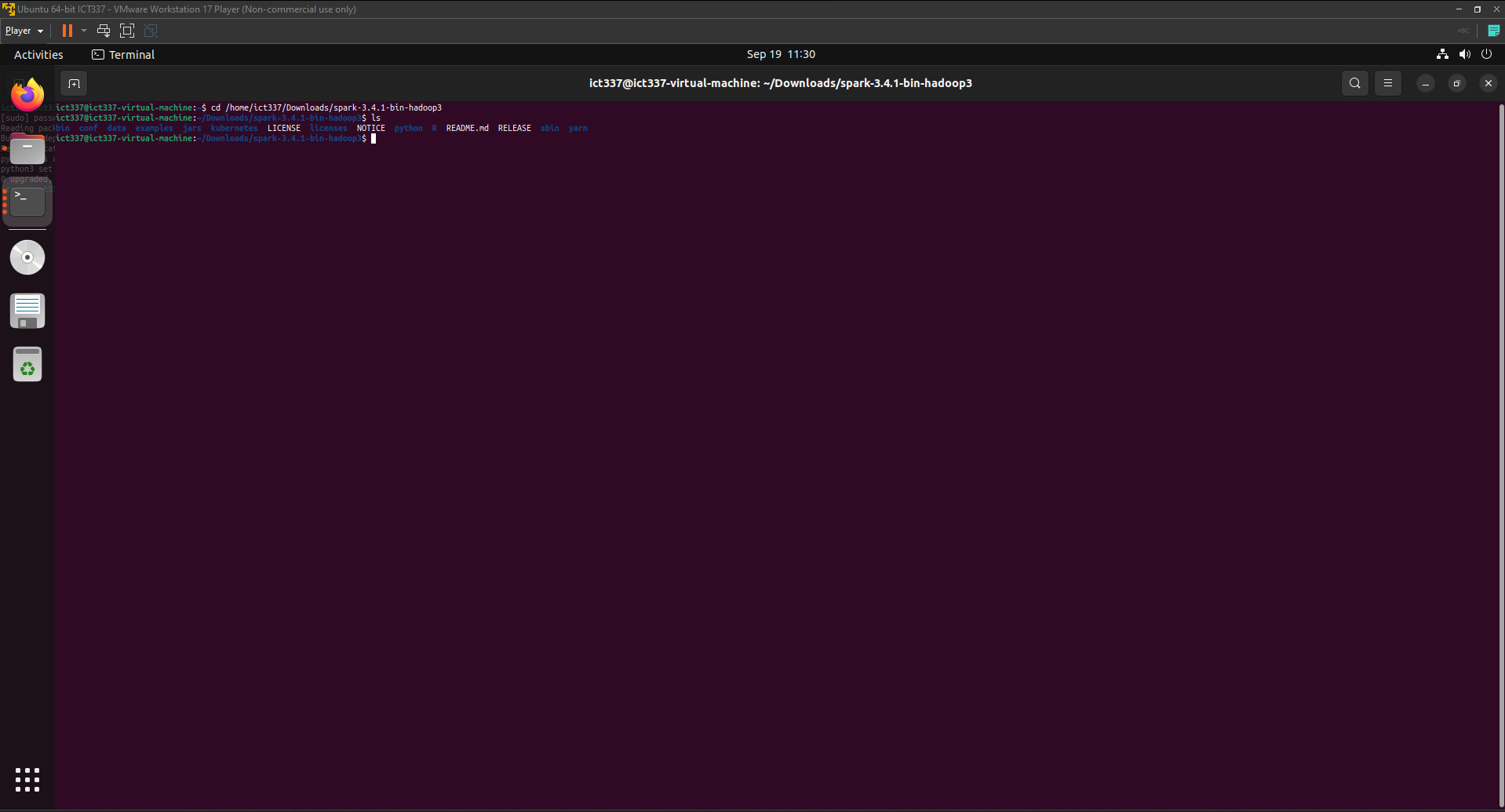
*Figure 5. Python3 installation in Ubuntu terminal.*

A screenshot of a computer

Description automatically generated

*Figure 6. Python3 installation verification in Ubuntu terminal.*

Thirdly, the Spark 3.4.1 release with pre-built for Apache Hadoop 3.3 and later is acquired from the Apache Spark official website. After obtaining the release, the.tgz file contents are extracted. In the terminal, ‘cd /home/ict337/Downloads/spark-3.4.1-bin-hadoop3’ command is used to navigate into the Spark Hadoop package directory and the ‘ls’ command is used to inspect the Hadoop folder (See Figure 7). To ensure the Spark Hadoop package is in place and functioning correctly, the ‘spark-shell’ is executed and Spark Web User Interfaced is launched. To accomplish this, the ‘cd bin’ command is used to navigate into the bin folder, followed by the execution of the ‘./spark-shell’ command (See Figure 8 and Figure 9).



*Figure 7. Changing directory to the downloaded and extracted .tgz file in Ubuntu terminal.*

A screenshot of a computer

Description automatically generated

*Figure 8. Execution of spark-shell in Ubuntu terminal*

A screenshot of a computer

Description automatically generated

*Figure 9. Spark Web User Interface*

Lastly, to demonstrate the non-interactive execution of built-in ‘pi.py’ example program, a new terminal window is opened and navigated into the into the Spark Hadoop package directory with the ‘cd /home/ict337/Downloads/spark-3.4.1-bin-hadoop3’ (See Figure 10). The ‘pi.py’ example program provided leverages Monte Carlo method to estimate pi. Notably, the number of partitions employed for the estimation is defaulted to 2 when no partition count is supplied in the command-line argument. This default behavior is governed by ‘partitions = int(sys.argv[1]) if len(sys.argv) > 1 else 2’ line of code in the pi.py example program. Therefore, the ‘./bin/spark-submit ./examples/src/main/python/pi.py’ command would suffice, returning a pi estimation of 3.135800 (see Figure 11).

A screenshot of a computer

Description automatically generated

*Figure 10. Changing directory to the Spark Hadoop package file in Ubuntu terminal.*

A screenshot of a computer screen

Description automatically generated

*Figure 11. Execution of ‘pi.py’ program without partitions supplied*

**(b)**

***Dependencies***

It is likely that the the 'pi.py' example program is written in Python 2 and to ensure compatibility with Python 2 and Python 3, the 'from \_future\_ import print\_function' statement is used to leverage the Python 3-style 'print' function such as '%f' formatter in Python 2. The 'import sys' line of code imports the 'sys' module to provide access to system-specific parameters. In the 'pi.py' example program, it uses command line arguments such as the 'sys.argv[]' and 'len(sys.argv)'.

Additionally, the 'from random import random' line of code imports the random function from the random module and the 'from operator import add' line of code imports the add function from the operator module. These functions are leveraged in 'pi.py' example program for randomizing the x and y axis and counting.

Furthermore, the 'from pyspark.sql import SparkSession' line of code imports the SparkSession class from the pyspark.sql module. This is used in the 'pi.py' example program as an API or entry point for configuring, creating, and interacting with Spark functionalities.

***Initialization***

Firstly, the 'if \_name\_ = "\_\_main\_\_"' line of code checks whether the script is executed as the main program to prevent unintended execution of the 'pi.py' example program. Additionally, the docstring of 'Usage: pi [partitions]' encased in the 3 double quotation marks provides the usage instructions for how to execute the 'pi.py' example program, indicating that it expects a 'partitions' argument to specify the number of partitions.

Secondly, inside the 'if \_name\_ = "\_\_main\_\_":' block of code, the 'SparkSession' class and '.builder' method is called to create and configure a spark instance respectively. Additionally, the '.appName("PythonPi")' method gives the application name of the spark job an identifier of "PythonPi". Furthermore, the '.getOrCreate()' method ensures a SparkSession is available for the 'pi.py' example program by checking whether there is an existing SparkSession and creates one if there is none.

***Main Application Logic***

Prior to the declaration of the ‘f’ function, some initial configuration is performed such as the number of partitions (1) which will be divided for parallel processing and determining the total number of data points across the specified number of partitions (2). Firstly, the line of code ‘partitions = int(sys.argv[1]) if len(sys.argv) > 1 else 2’ retrieves the ‘partitions’ value from the command-line. This operation identifies arguments specified by the user in the command-line argument. For example, in the command ‘./bin/spark-submit ./examples/src/main/python/pi.py 7’, it contains more than 1 system argument, satisfying the condition of ‘if len(sys.argv) > 1’. Hence the number 7 identified by ‘sys.argv[1]’ would be converted to an integer and assigns it to the ‘partitions’ variable. However, in the case where no arguments are specified in the command-line argument, the value of ‘len(sys.argv)’ would remain at 1. Therefore, the number of partitions to default at 2, as it does not satisfy the condition of ‘if len(sys.argv) > 1’. Secondly, the line of code ‘n = 100000 \* partitions’ multiplies 100000 to the number of partitions specified to be later distributed across the partitions. For example, in the command ‘./bin/spark-submit ./examples/src/main/python/pi.py 7’, spark would distribute 700000 data points across 7 partitions, with each partition handling a portion of the data points for parallel processing.

In the ‘f’ function, the x-coordinate, y-coordinate (1) are defined and determines whether the data point falls within a circle with a radius of 1 (2). Firstly, the lines of code ‘x = random() \* 2 – 1’ and ‘y = random() \* 2 – 1’ generate a floating number from 0 to 1. It subsequently multiplies the random floating number by 2 and subtracts 1 from it to ensure that the coordinates may fall within the range of -1 to 1. This process assigns random values within the range from -1 to 1 to the x-coordinate and y-coordinate. Secondly, the line of code ‘return 1 if x \*\* 2 + y \*\* 2 <= 1 else 0’ calculates the squared Euclidean distance from the given random x-coordinate and y-coordinate to origin. Subsequently, it categorizes the random data points, ‘1’ indicating the data point is within the range of -1 to 1 and ‘0’ indicating the data point is outside of this specified range.

In the line of code ‘count = spark.sparkContext.Parallelize(range(1, n + 1), partitions).map(f).reduce(add)’, it creates a Resilient Distributed Datasets (RDD) (1), uses spark to parallelize the generation of random data points (2), apply the ‘f’ function to each element in the RDD (3), and apply the ‘add’ function to add up all the ‘1’s and ‘0’s (4). Firstly, ‘spark.sparkContext’ is an entry point for the spark session to access spark operations such as creating a RDD and perform distributed data processing tasks. Secondly, the ‘.Parallelize(range(1, n + 1), partitions)’ method creates a RDD ‘n’ elements with a sequence of 1 to ‘n’, where ‘n’ represents the total number of data points. This process subsequently distributes these elements across the specified number of partitions. Thirdly, the ‘.map(f)’ transformation applies the ‘f’ function to the RDD and transforms each element in the RDD into ‘1’s or ‘0’s. Lastly, the ‘reduce(add)’ action executes the transformation in the RDD and aggregates the elements across all the partitions. Therefore, this process effectively counts the number of data points that fall within a circle with a radius of 1.

In the line of code ‘print("Pi is roughly %f" % (4.0 \* count/n))’, the ‘pi.py’ example program outputs ‘Pi is roughly %f’, where ‘%f’ is a placeholder for floating value computed by ‘%(4.0 \* count/n)’. Therefore, it computes the estimate of pi with ‘%(4.0 \* count/n)’ and outputs "Pi is roughly %f" in a single print statement. Finally, the ‘spark.stop()’ method is used to end the spark session to ensure a clean shutdown of spark instance in the ‘pi.py’ example program.

**Question 3**

import logging

import os

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, sum, avg, min, max, when

# Constants

SCRIPTS\_DIR = os.path.abspath(os.path.dirname(\_\_file\_\_))

DATA\_DIR = os.path.join(SCRIPTS\_DIR, "..", "data")

FLIGHTS\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "flights\_data\_v2.csv")

PLANES\_DATA\_FILE\_PATH = os.path.join(DATA\_DIR, "planes\_data\_v2.csv")

LOGGING\_LEVEL = logging.INFO

LOAD\_DATA\_ERROR\_MESSAGE = "An error occurred while loading data: {}"

FILE\_NOT\_FOUND\_MESSAGE = "The specified file does not exist: {}"

def configure\_logging():

    """Configure logging settings and return a logger object.

    Returns

    -------

    logger : object

        Logger object for logging messages.

    Notes

    -----

    This function initializes the logging settings, including the logging level,

    and returns a logger object that can be used for logging messages within the application.

    The default logging level is set to the value of the constant LOGGING\_LEVEL.

    """

    logging.basicConfig(*level*=LOGGING\_LEVEL)

    return logging.getLogger(\_\_name\_\_)

def create\_spark\_session(*app\_name*="TMA\_Data\_Analysis"):

    """

    Create and return a Spark session.

    Parameters

    ----------

    app\_name : str, optional

        The name of the Spark application, by default "TMA\_Data\_Analysis".

    Returns

    -------

    SparkSession

        The Spark session object.

    Notes

    -----

    This function initializes a Spark session, which is the entry point for working with Spark functionality.

    """

    return SparkSession.builder.appName(app\_name)\

        .config("spark.some.config.option", "some-value")\

        .getOrCreate()

def show\_dataframe(*df*, *max\_rows*=100, *show\_rows*=20):

    """

    Show rows of a DataFrame with the option to limit the number of rows displayed.

    Parameters

    ----------

    df : DataFrame

        The DataFrame to be displayed.

    max\_rows : int, optional

        The maximum number of rows to display. Default is 20.

    Returns

    -------

    None

    Notes

    -----

    This function displays rows of the input DataFrame. If the DataFrame contains more

    rows than the specified `max\_rows`, it will limit the display to the first `max\_rows`

    rows. If the DataFrame has fewer rows than `max\_rows`, it will display all available

    rows without truncation.

    """

    if df.count() > max\_rows:

        df.show(show\_rows)

    else:

        df.show(df.count(), *truncate*=False)

def load\_data(*spark*, *logger*, *file\_path*):

    """

    Load data from CSV file into a Spark DataFrame.

    Parameters

    ----------

    spark : SparkSession

        The Spark session.

    logger : Logger

        Logger object for logging messages.

    file\_path : str, optional

        The path to the CSV file to load.

    Returns

    -------

    DataFrame

        DataFrame containing the data.

    Raises

    ------

    FileNotFoundError

        If the specified file does not exist.

    Notes

    -----

    This function reads data from a CSV file and loads it into a Spark DataFrame.

    Note: The default logging level is set to the value of the constant LOGGING\_LEVEL.

    """

    try:

        # Load data from csv

        df = spark.read.option("inferSchema", "true").option(

            "header", "true").csv(file\_path)

        return df

    except *Exception* as e:

        if "Path does not exist" in *str*(e):

            logger.error(FILE\_NOT\_FOUND\_MESSAGE.format(file\_path))

            raise *FileNotFoundError*(f"File not found: {file\_path}")

        logger.error(LOAD\_DATA\_ERROR\_MESSAGE.format(*str*(e)))

        raise e

def process\_missing\_data(*df*, *logger*):

    """Process and analyze the loaded data.

    Parameters

    ----------

    df : DataFrame

        DataFrame containing the data.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame containing the data.

    Raises

    ------

    Exception

        If an error occurs during data processing.

    Notes

    -----

    This function checks for missing values in the columns of the input DataFrame, identifies and displays

    rows with missing values, and removes those rows from the DataFrame. It provides information about

    the number of missing values, the resulting cleaned DataFrame, and any errors encountered during

    the process.

    """

    try:

        # Check for missing values in columns

        columns\_to\_check = df.columns

        filter\_condition = None

        for column\_name in columns\_to\_check:

            if filter\_condition is None:

                filter\_condition = col(column\_name).isNull()

            else:

                filter\_condition = filter\_condition | col(column\_name).isNull()

        # Find and display rows with missing values

        missing\_data\_df = df.filter(filter\_condition)

        logger.info("Sample rows in the df DataFrame with Missing Value:")

        show\_dataframe(missing\_data\_df)

        missing\_occurrence = missing\_data\_df.count()

        logger.info(

            f"There are {missing\_occurrence} rows with missing values in df.\n")

        # Remove rows with missing values

        clean\_data\_df = df.filter(~filter\_condition)

        return clean\_data\_df

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def count\_by(*df*, *grouped\_columns*, *logger*):

    """

    Count occurrences of rows by grouping columns.

    Parameters

    ----------

    df : DataFrame

        DataFrame containing the data.

    grouped\_columns : list

        List of columns to group by.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame with counts, sorted in descending order.

    Raises

    ------

    Exception

        If an error occurs during counting.

    Notes

    -----

    This function groups the data in the input DataFrame by the specified columns and counts the

    occurrences of rows within each group. The result is a DataFrame containing counts, sorted in

    descending order based on the count values.

    """

    try:

        count\_by\_col = df.groupby(\*grouped\_columns).count()

        sorted\_counts\_df = count\_by\_col.orderBy("count", *ascending*=False)

        return sorted\_counts\_df

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def percentage\_by(*df*, *grouped\_columns*, *logger*):

    """

    Calculate the percentage of occurrences by grouping columns.

    Parameters

    ----------

    df : DataFrame

        DataFrame containing the data.

    grouped\_columns : list

        List of columns to group by.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame with counts and percentages, sorted in descending order.

    Raises

    ------

    Exception

        If an error occurs during percentage calculation.

    Notes

    -----

    This function groups the data in the input DataFrame by the specified columns and calculates

    the number and percentage of occurrences within each group relative to the total number of rows in the

    DataFrame. The result is a DataFrame containing both counts and percentages, sorted in

    descending order based on the percentage values.

    """

    try:

        total\_flights = df.count()

        total\_flights\_by = df.groupby(\*grouped\_columns).count()

        col\_percentage = total\_flights\_by.withColumn(

            "percentage", (total\_flights\_by["count"] / total\_flights) \* 100).orderBy("percentage", *ascending*=False)

        return col\_percentage

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def top\_n\_cat\_by(*df*, *grouped\_columns*, *n*, *logger*):

    """

    Find the top n category.

    Parameters

    ----------

    df : DataFrame

        DataFrame containing the data.

    grouped\_columns : list

        List of columns to group by.

    n : int

        Number of top category to retrieve.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame with the top category, sorted by count in descending order.

    Raises

    ------

    Exception

        If an error occurs during counting.

    Notes

    -----

    This function groups the data in the specified DataFrame by the specified columns and counts

    the occurrences of each category. It returns a DataFrame containing the top n categories

    with the highest counts, sorted in descending order based on the count values.

    """

    try:

        top\_cat = df.groupBy(\*grouped\_columns).count().orderBy(

            "count", *ascending*=False).limit(n)

        return top\_cat

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def bottom\_n\_cat\_by(*df*, *grouped\_columns*, *n*, *logger*):

    """

    Find the bottom n category.

    Parameters

    ----------

    df : DataFrame

        DataFrame containing the data.

    grouped\_columns : list

        List of columns to group by.

    n : int

        Number of bottom category to retrieve.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame with the bottom category, sorted by count in ascending order.

    Raises

    ------

    Exception

        If an error occurs during counting.

    Notes

    -----

    This function groups the data in the specified DataFrame by the specified columns and counts

    the occurrences of each category. It returns a DataFrame containing the bottom n categories

    with the highest counts, sorted in ascending order based on the count values.

    """

    try:

        bottom\_cat = df.groupBy(\*grouped\_columns).count().orderBy(

            "count", *ascending*=True).limit(n)

        return bottom\_cat

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def sql\_top\_n\_cat\_by(*df*, *grouped\_columns*, *n*, *spark*, *logger*):

    """

    Find the top n category.

    Parameters

    ----------

    df : DataFrame

        DataFrame containing the data.

    grouped\_columns : list

        List of columns to group by.

    n : int

        Number of top category to retrieve.

    spark : SparkSession

        The Spark session.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame with the top category, sorted by count in descending order.

    Raises

    ------

    Exception

        If an error occurs during counting.

    Notes

    -----

    This function groups the data in the specified DataFrame by the specified columns and counts

    the occurrences of each category. It returns a DataFrame containing the top n categories

    with the highest counts, sorted in descending order based on the count values.

    """

    try:

        df.createOrReplaceTempView("top\_flights\_planes")

        columns\_str = ", ".join(grouped\_columns)

        query = f"""

        SELECT {columns\_str}, COUNT(\*) AS count

        FROM top\_flights\_planes

        GROUP BY {columns\_str}

        ORDER BY count DESC

        LIMIT {n}

        """

        sql\_top\_n\_cat = spark.sql(query)

        return sql\_top\_n\_cat

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def sql\_bottom\_n\_cat\_by(*df*, *grouped\_columns*, *n*, *spark*, *logger*):

    """

    Find the bottom n category.

    Parameters

    ----------

    df : DataFrame

        DataFrame containing the data.

    grouped\_columns : list

        List of columns to group by.

    n : int

        Number of bottom category to retrieve.

    spark : SparkSession

        The Spark session.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame with the bottom category, sorted by count in ascending order.

    Raises

    ------

    Exception

        If an error occurs during counting.

    Notes

    -----

    This function groups the data in the specified DataFrame by the specified columns and counts

    the occurrences of each category. It returns a DataFrame containing the bottom n categories

    with the highest counts, sorted in ascending order based on the count values.

    """

    try:

        df.createOrReplaceTempView("bottom\_flights\_planes")

        columns\_str = ", ".join(grouped\_columns)

        query = f"""

        SELECT {columns\_str}, COUNT(\*) AS count

        FROM bottom\_flights\_planes

        GROUP BY {columns\_str}

        ORDER BY count ASC

        LIMIT {n}

        """

        sql\_bottom\_n\_cat = spark.sql(query)

        return sql\_bottom\_n\_cat

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def analyze\_average\_delay(*df*, *column*, *delay\_column*, *logger*):

    """

    Analyze average departure/arrival delay.

    Parameters

    ----------

    df : DataFrame

        The DataFrame containing the data.

    column : str

        The column by which to group the data for analysis.

    delay\_column : str

        The column representing departure/arrival delay.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame containing the average departure/arrival delay.

    Raises

    ------

    Exception

        If an error occurs during the computation.

    Notes

    -----

    This function calculates the average departure/arrival delay for a specified column,

    groups the data by another column, and orders the results in descending order based on the average delay.

    """

    try:

        suffix = delay\_column.split('\_')[0]

        new\_column\_name = f"average\_{suffix}\_delay"

        avg\_departure\_delay\_by\_column = df.groupBy(column).agg(avg(col(delay\_column)).alias(

            new\_column\_name)).orderBy(new\_column\_name, *ascending*=False)

        return avg\_departure\_delay\_by\_column

    except *Exception* as e:

        logger.error(f"An error occurred during data analysis: {*str*(e)}")

        raise e

def analyze\_positive\_delay(*df*, *column*, *delay\_column*, *logger*):

    """

    Analyze positive departure/arrival delay.

    Parameters

    ----------

    df : DataFrame

        The DataFrame containing the data.

    column : str

        The column by which to group the data for analysis.

    delay\_column : str

        The column representing departure/arrival delay.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame containing the analysis of positive departure/arrival delay.

    Raises

    ------

    Exception

        If an error occurs during the computation.

    Notes

    -----

    This function calculates the average positive departure/arrival delay for a specified column,

    groups the data by another column, and orders the results in descending order based on the average positive delay.

    """

    try:

        suffix = delay\_column.split('\_')[0]

        new\_column\_name = f"average\_positive\_{suffix}\_delay"

        avg\_positive\_delay\_by\_column = df.groupBy(column).agg(avg(when(col(delay\_column) > 0, col(

            delay\_column))).alias(new\_column\_name)).orderBy(new\_column\_name, *ascending*=False)

        return avg\_positive\_delay\_by\_column

    except *Exception* as e:

        logger.error(f"An error occurred during data analysis: {*str*(e)}")

        raise e

def analyze\_negative\_delay(*df*, *column*, *delay\_column*, *logger*):

    """

    Analyze negative departure/arrival delay.

    Parameters

    ----------

    df : DataFrame

        The DataFrame containing the data.

    column : str

        The column by which to group the data for analysis.

    delay\_column : str

        The column representing departure/arrival delay.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame containing the analysis of negative departure/arrival delay.

    Raises

    ------

    Exception

        If an error occurs during the computation.

    Notes

    -----

    This function calculates the average negative departure/arrival delay for a specified column,

    groups the data by another column, and orders the results in descending order based on the average negative delay.

    """

    try:

        suffix = delay\_column.split('\_')[0]

        new\_column\_name = f"average\_negative\_{suffix}\_delay"

        avg\_negative\_delay\_by\_column = df.groupBy(column).agg(avg(when(col(delay\_column) < 0, col(

            delay\_column))).alias(new\_column\_name)).orderBy(new\_column\_name, *ascending*=False)

        return avg\_negative\_delay\_by\_column

    except *Exception* as e:

        logger.error(f"An error occurred during data analysis: {*str*(e)}")

        raise e

def numeric\_stats(*df*, *group\_by\_column*, *numeric\_column*, *logger*):

    """

    Compute statistics for a numeric column grouped by another column.

    Parameters

    ----------

    df : DataFrame

        The input DataFrame containing the data.

    group\_by\_column : str

        The name of the column to group by.

    numeric\_column : str

        The name of the numeric column to compute statistics for.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        A DataFrame containing statistics (average, minimum, and maximum) for the numeric column

        grouped by the specified column.

    Raises

    ------

    Exception

        If an error occurs during the computation.

    Notes

    -----

    1. This function calculates statistics (average, minimum, maximum) for a specified numeric column in the DataFrame.

    2. The statistics are computed based on groups formed by the values in the specified 'group\_by\_column.'

    3. The resulting DataFrame is ordered in descending order of the average of the numeric column.

    """

    try:

        col\_stats = df.groupBy(group\_by\_column).agg(

            avg(numeric\_column).alias(f"average\_{numeric\_column}"),

            min(numeric\_column).alias(f"min\_{numeric\_column}"),

            max(numeric\_column).alias(f"max\_{numeric\_column}")

        ).orderBy(f"average\_{numeric\_column}", *ascending*=False)

        return col\_stats

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def compute\_flight\_speed(*df*, *distance*, *air\_time*, *logger*):

    """

    Calculate flight speed in miles per hour and add it as a new column.

    Parameters

    ----------

    df : DataFrame

        The input DataFrame containing flight data.

    distance : str

        The name of the column representing flight distance in miles.

    air\_time : str

        The name of the column representing flight air time in minutes.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        A DataFrame with an additional column, "flight\_speed (miles per hour)," representing the calculated

        flight speed for each record.

    Raises

    ------

    Exception

        If an error occurs during the computation.

    Notes

    -----

    This function calculates the flight speed (in miles per hour) by dividing the flight distance (in miles)

    by the flight air time (in minutes) and adds it as a new column to the input DataFrame.

    Note: Flight air time is converted to hours by dividing by 60 to obtain the speed in miles per hour.

    """

    try:

        df\_add\_speed = df.withColumn(

            "flight\_speed (miles per hour)", (col(distance) / (col(air\_time) / 60)))

        return df\_add\_speed

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def shortest\_n\_flight\_from\_origin(*df*, *origin\_column*, *origin\_name*, *measurement*, *n*, *logger*):

    """

    Find the shortest 'n' flights from a specific origin based on a measurement.

    Parameters

    ----------

    df : DataFrame

        The DataFrame containing the flight data.

    origin\_column : str

        The name of the column representing the flight origin.

    origin\_name : str

        The name of the origin for which to find the shortest flights.

    measurement : str

        The column name representing the measurement by which to find the shortest flights.

    n : int

        The number of shortest flights to retrieve.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame containing the 'n' shortest flights from the specified origin based on the given measurement.

    Raises

    ------

    Exception

        If an error occurs during the computation.

    Notes

    -----

    This function filters the DataFrame to select flights originating from a specific location (origin\_name).

    It then sorts these flights by the provided measurement column in ascending order and retrieves the top 'n' shortest flights.

    """

    try:

        origin = df.filter(df[origin\_column] == origin\_name)

        shortest\_flight = origin.select(

            origin\_column, measurement).orderBy(measurement, *ascending*=True).limit(n)

        return shortest\_flight

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def longest\_n\_flight\_from\_origin(*df*, *origin\_column*, *origin\_name*, *measurement*, *n*, *logger*):

    """

    Find the longest 'n' flights from a specific origin based on a measurement.

    Parameters

    ----------

    df : DataFrame

        The DataFrame containing the flight data.

    origin\_column : str

        The name of the column representing the flight origin.

    origin\_name : str

        The name of the origin for which to find the longest flights.

    measurement : str

        The column name representing the measurement by which to find the longest flights.

    n : int

        The number of longest flights to retrieve.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        DataFrame containing the 'n' longest flights from the specified origin based on the given measurement.

    Raises

    ------

    Exception

        If an error occurs during the computation.

    Notes

    -----

    This function filters the DataFrame to select flights originating from a specific location (origin\_name).

    It then sorts these flights by the provided measurement column in ascending order and retrieves the top 'n' longest flights.

    """

    try:

        origin = df.filter(df[origin\_column] == origin\_name)

        longest\_flight = origin.select(

            origin\_column, measurement).orderBy(measurement, *ascending*=False).limit(n)

        return longest\_flight

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def average\_duration(*df*, *carrier\_column*, *carrier\_name*, *origin\_column*, *origin\_name*, *measurement*, *logger*):

    """

    Calculate the average flight duration for a specific carrier and origin.

    Parameters

    ----------

    df : DataFrame

        The input DataFrame containing flight data.

    carrier\_column : str

        The name of the column representing the carrier.

    carrier\_name : str

        The name of the carrier for which to calculate the average duration.

    origin\_column : str

        The name of the column representing the origin airport.

    origin\_name : str

        The name of the origin airport for which to calculate the average duration.

    measurement : str

        The name of the column representing the flight duration measurement in minutes.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        A DataFrame with the average flight duration for the specified carrier and origin.

    Raises

    ------

    Exception

        If an error occurs during the computation.

    Notes

    -----

    This function filters the input DataFrame to select flights operated by a specific carrier and originating from a

    specific airport. It then calculates the average flight duration (in minutes) for these flights.

    """

    try:

        carrier = df.filter(df[carrier\_column] == carrier\_name)

        origin = carrier.filter(df[origin\_column] == origin\_name)

        average\_duration = origin.agg(

            avg(measurement).alias(f"average\_{measurement} (mins)"))

        return average\_duration

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def total\_duration(*df*, *carrier\_column*, *carrier\_name*, *origin\_column*, *origin\_name*, *measurement*, *logger*):

    """

    Calculate the total flight duration for a specific carrier and origin.

    Parameters

    ----------

    df : DataFrame

        The input DataFrame containing flight data.

    carrier\_column : str

        The name of the column representing the carrier.

    carrier\_name : str

        The name of the carrier for which to calculate the total duration.

    origin\_column : str

        The name of the column representing the origin airport.

    origin\_name : str

        The name of the origin airport for which to calculate the total duration.

    measurement : str

        The name of the column representing the flight duration measurement in minutes.

    logger : object

        Logger object for logging messages.

    Returns

    -------

    DataFrame

        A DataFrame with the total flight duration for the specified carrier and origin.

    Raises

    ------

    Exception

        If an error occurs during the computation.

    Notes

    -----

    This function filters the input DataFrame to select flights operated by a specific carrier and originating from a

    specific airport. It then calculates the total flight duration (in hours) for these flights.

    Note: Flight air time is converted to hours by dividing by 60 to obtain the total flight duration in hours.

    """

    try:

        carrier = df.filter(df[carrier\_column] == carrier\_name)

        origin = carrier.filter(df[origin\_column] == origin\_name)

        total\_duration\_hours = origin.agg(

            (sum(measurement) / 60).alias(f"total\_{measurement} (hours)"))

        return total\_duration\_hours

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

def main():

    """Entry point of the script.

    Parameters

    ----------

    None

    Returns

    -------

    None

    Notes

    -----

    This function serves as the entry point of the script for processing flight data. It performs the following steps:

    1. Configures the logging settings and initializes a logger.

    2. Creates a Spark session for data processing.

    3. Loads flight data from a CSV file and processes it to handle missing values.

    4. Performs various data analyses.

    5. Displays and logs the analysis results.

    6. Stops the Spark session when processing is complete.

    """

    logger = configure\_logging()

    spark = create\_spark\_session()

    try:

        flights\_data\_frame = load\_data(spark, logger, FLIGHTS\_DATA\_FILE\_PATH)

        logger.info("Sample rows in the df DataFrame:")

        show\_dataframe(flights\_data\_frame)

        occurrence = flights\_data\_frame.count()

        logger.info(f"There are {occurrence} number of df.\n")

        logger.info(flights\_data\_frame.schema)

        clean\_flights\_data\_df = process\_missing\_data(

            flights\_data\_frame, logger)

        logger.info("Sample rows in the cleaned df DataFrame:")

        show\_dataframe(clean\_flights\_data\_df)

        clean\_occurrence = clean\_flights\_data\_df.count()

        logger.info(

            f"{clean\_occurrence} rows remained after removing the rows with missing values.\n")

        flight\_by\_year\_month = count\_by(clean\_flights\_data\_df, [

            "year", "month"], logger)

        show\_dataframe(flight\_by\_year\_month)

        flight\_by\_day = count\_by(clean\_flights\_data\_df, ["day"], logger)

        show\_dataframe(flight\_by\_day)

        percentage\_flight\_by\_carrier = percentage\_by(

            clean\_flights\_data\_df, ["carrier"], logger)

        show\_dataframe(percentage\_flight\_by\_carrier)

        flights\_by\_origin = count\_by(clean\_flights\_data\_df, ["origin"], logger)

        show\_dataframe(flights\_by\_origin)

        flights\_by\_dest = count\_by(clean\_flights\_data\_df, ["dest"], logger)

        show\_dataframe(flights\_by\_dest)

        top\_10\_planes = top\_n\_cat\_by(

            clean\_flights\_data\_df, ["tailnum"], 10, logger)

        show\_dataframe(top\_10\_planes)

        flights\_by\_hour = count\_by(clean\_flights\_data\_df, ["hour"], logger)

        show\_dataframe(flights\_by\_hour)

        avg\_pos\_dep\_delay\_by\_carrier = analyze\_positive\_delay(

            clean\_flights\_data\_df, "carrier", "dep\_delay", logger)

        show\_dataframe(avg\_pos\_dep\_delay\_by\_carrier)

        avg\_dep\_delay\_by\_carrier = analyze\_average\_delay(

            clean\_flights\_data\_df, "carrier", "dep\_delay", logger)

        show\_dataframe(avg\_dep\_delay\_by\_carrier)

        avg\_dep\_delay\_by\_month = analyze\_average\_delay(

            clean\_flights\_data\_df, "month", "dep\_delay", logger)

        show\_dataframe(avg\_dep\_delay\_by\_month)

        avg\_dep\_delay\_by\_hour = analyze\_average\_delay(

            clean\_flights\_data\_df, "hour", "dep\_delay", logger)

        show\_dataframe(avg\_dep\_delay\_by\_hour)

        avg\_neg\_dep\_delay\_by\_carrier = analyze\_negative\_delay(

            clean\_flights\_data\_df, "carrier", "dep\_delay", logger)

        show\_dataframe(avg\_neg\_dep\_delay\_by\_carrier)

        avg\_neg\_dep\_delay\_by\_month = analyze\_negative\_delay(

            clean\_flights\_data\_df, "month", "dep\_delay", logger)

        show\_dataframe(avg\_neg\_dep\_delay\_by\_month)

        avg\_neg\_dep\_delay\_by\_hour = analyze\_negative\_delay(

            clean\_flights\_data\_df, "hour", "dep\_delay", logger)

        show\_dataframe(avg\_neg\_dep\_delay\_by\_hour)

        distance\_stats = numeric\_stats(

            clean\_flights\_data\_df, "carrier", "distance", logger)

        show\_dataframe(distance\_stats)

        transformed\_01\_df = compute\_flight\_speed(

            clean\_flights\_data\_df, "distance", "air\_time", logger)

        show\_dataframe(transformed\_01\_df)

        speed\_stats = numeric\_stats(

            transformed\_01\_df, "carrier", "flight\_speed (miles per hour)", logger)

        show\_dataframe(speed\_stats)

        shortest\_flight\_distance\_PDX = shortest\_n\_flight\_from\_origin(

            transformed\_01\_df, "origin", "PDX", "distance", 1, logger)

        show\_dataframe(shortest\_flight\_distance\_PDX)

        longest\_flight\_distance\_SEA = longest\_n\_flight\_from\_origin(

            transformed\_01\_df, "origin", "SEA", "distance", 1, logger)

        show\_dataframe(longest\_flight\_distance\_SEA)

        average\_duration\_UA\_SEA = average\_duration(

            transformed\_01\_df, "carrier", "UA", "origin", "SEA", "air\_time", logger)

        show\_dataframe(average\_duration\_UA\_SEA)

        total\_duration\_UA\_SEA = total\_duration(

            transformed\_01\_df, "carrier", "UA", "origin", "SEA", "air\_time", logger)

        show\_dataframe(total\_duration\_UA\_SEA)

        planes\_data\_frame = load\_data(spark, logger, PLANES\_DATA\_FILE\_PATH)

        clean\_planes\_data\_df = planes\_data\_frame.drop("speed")

        clean\_planes\_data\_df = clean\_planes\_data\_df.withColumnRenamed(

            "year", "plane\_year")

        show\_dataframe(clean\_planes\_data\_df)

        flights\_planes\_df = transformed\_01\_df.join(

            clean\_planes\_data\_df, *on*=["tailnum"], *how*="inner")

        show\_dataframe(flights\_planes\_df)

        logger.info(flights\_planes\_df.count())

        clean\_flights\_planes\_data\_df = process\_missing\_data(

            flights\_planes\_df, logger)

        top\_20\_flights\_planes = top\_n\_cat\_by(clean\_flights\_planes\_data\_df, [

            "carrier", "model", "plane\_year"], 20, logger)

        show\_dataframe(top\_20\_flights\_planes)

        bottom\_20\_flights\_planes = bottom\_n\_cat\_by(clean\_flights\_planes\_data\_df, [

            "carrier", "model", "plane\_year"], 20, logger)

        show\_dataframe(bottom\_20\_flights\_planes)

        sql\_top\_20\_flights\_planes = sql\_top\_n\_cat\_by(clean\_flights\_planes\_data\_df, [

            "carrier", "model", "plane\_year"], 20, spark, logger)

        show\_dataframe(sql\_top\_20\_flights\_planes)

        sql\_bottom\_20\_flights\_planes = sql\_bottom\_n\_cat\_by(clean\_flights\_planes\_data\_df, [

            "carrier", "model", "plane\_year"], 20, spark, logger)

        show\_dataframe(sql\_bottom\_20\_flights\_planes)

    except *Exception* as e:

        logger.error(f"An error occurred: {*str*(e)}")

        raise e

    finally:

        if spark is not None:

            spark.stop()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**(a)**

**(b)**

**(c)**

**(d)**

**(e)**

**Question 4**

(a)

(b)

(c)

(d)

(e)

(f)