Ong Yeow Hwee, Glen

Singapore University of Social Science

ICT337: Big Data Computing in the Cloud

Prof. Hu He

October 28, 2022

**Question 1**

**(a)**

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

Resilient Distributed Datasets (RDD) and DataFrames are APIs that Spark offers to serve as the primary abstraction used for distributed data processing in the PySpark library.

The high-level APIs, DataFrames and Datasets. The low-level API, RDD.

RDD is Apache Sparks primary data structure. It is an immutable group of objects that computes on several cluster nodes. Furthermore, with the use of RDD lineage graph, the system is resilient of fault-tolerant and is therefore able to recompute missing or damaged partitions because of node failure.

DataFrame in PySpark is a distributed grouping of rows with name columns. It is equivalent to an excel sheet with column headers or a table in a relational database. The DataFrame is partitioned across servers in data centers.

Immutable: we can only build a RDD or DataFrame once without being able to edit it.

***DataFrames***

**(b)**

|  |  |  |
| --- | --- | --- |
| **Characteristic** | **Resilient Distributed Datasets (RDD)** | **DataFrames** |
| Abstraction level | Low-level | High-level |
| Catalyst |  |  |
| Data structure | Unstructured | Structured (tabular) |
| Schema | No schema | Schema with named columns |
| Typing | No type enforcement | Strong typed |
| 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

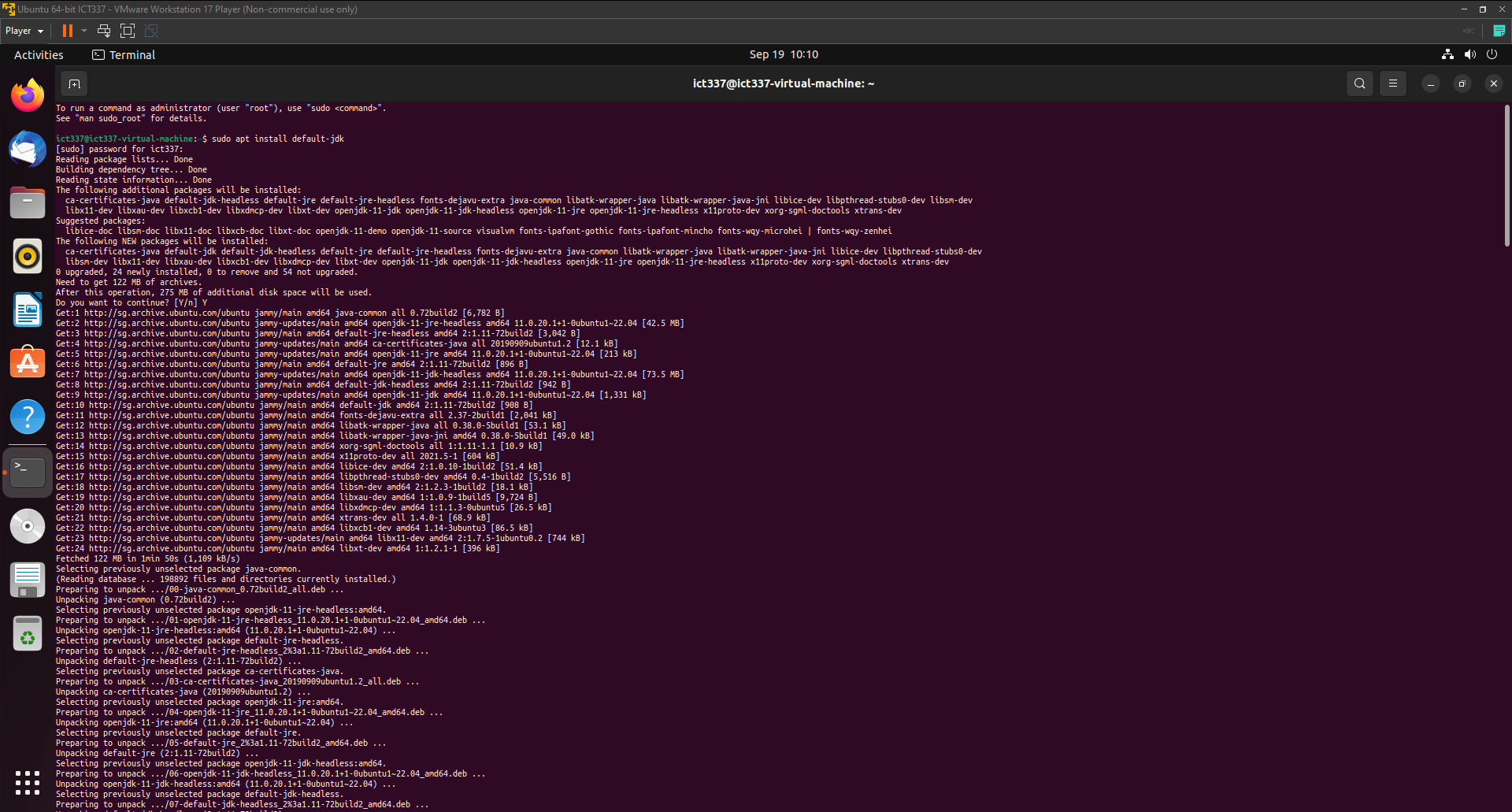
Description automatically generated

*Figure 2. Virtual machine configuration in VMware.*

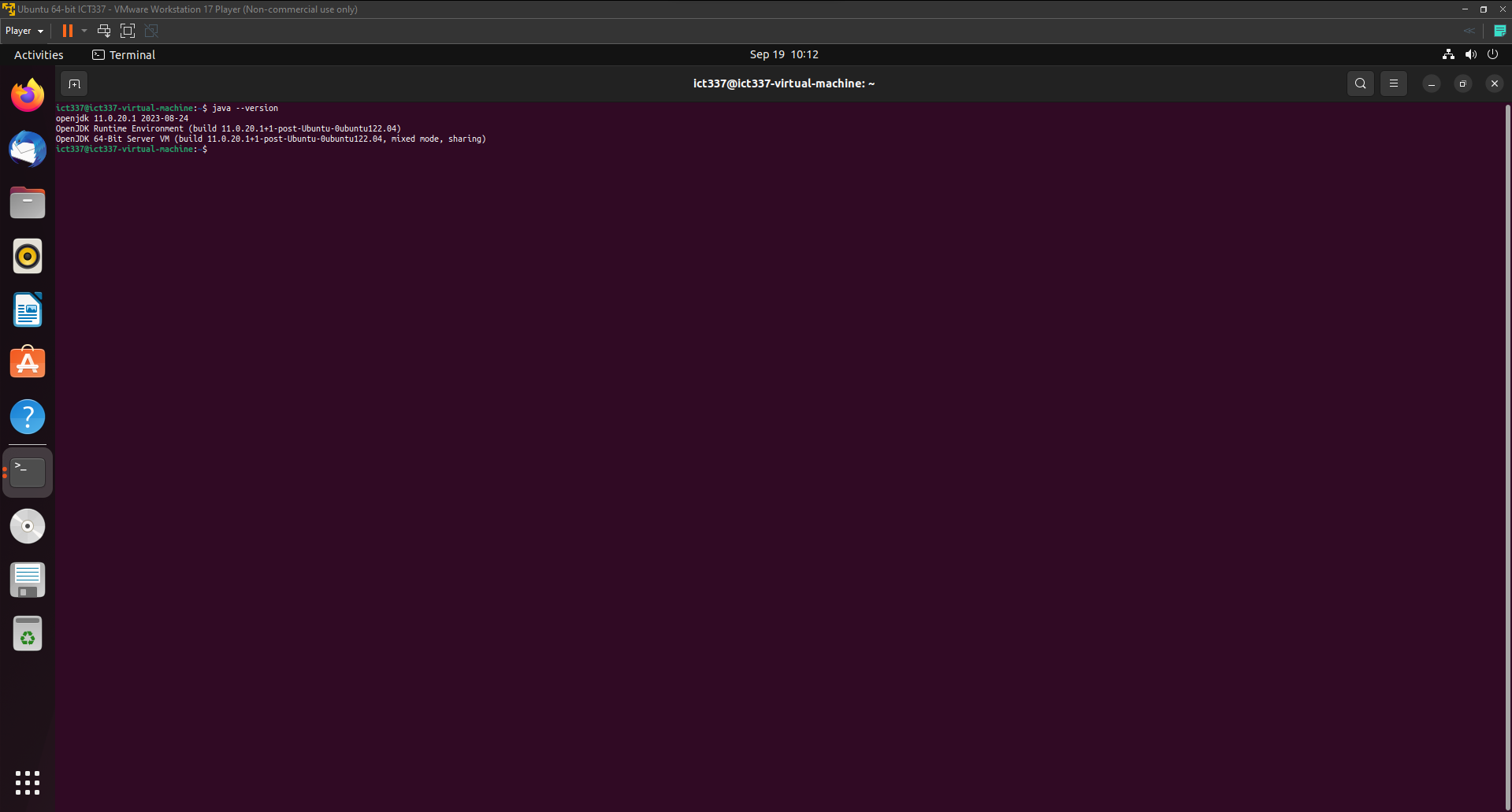
***Prerequisites***

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).

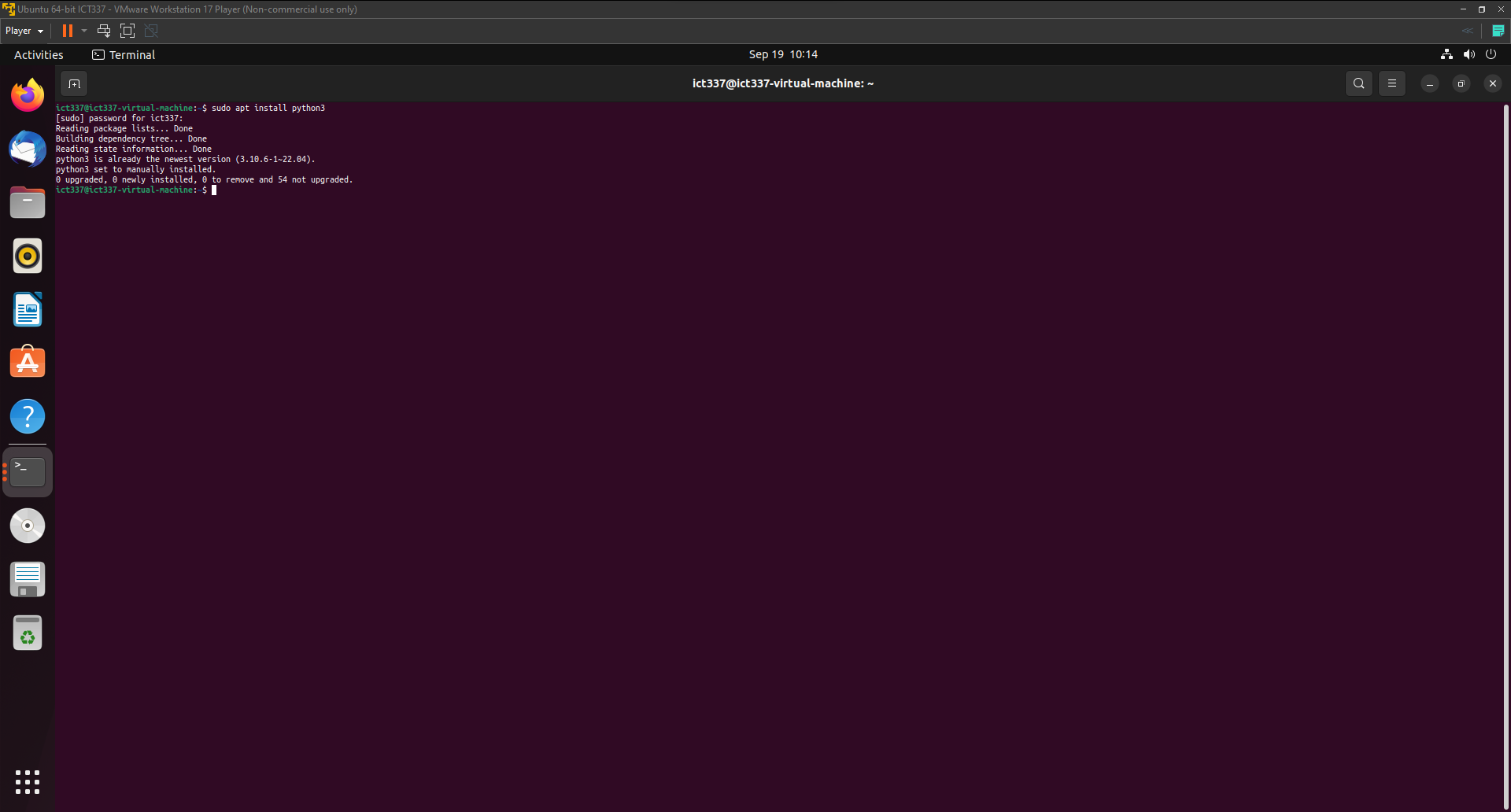


*Figure 3. Sample Java Development Kit installation in Ubuntu terminal.*



*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).



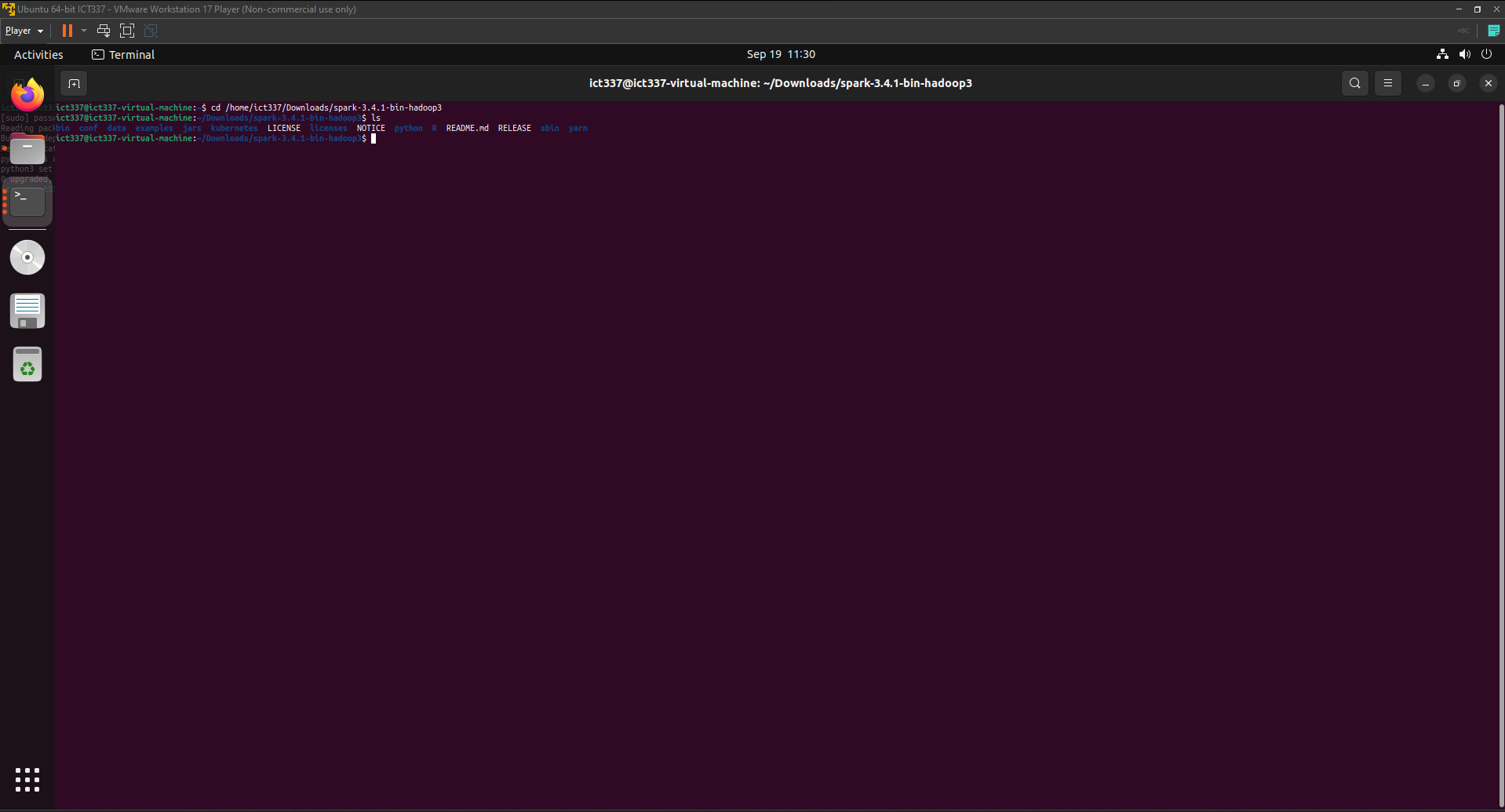
*Figure 5. Python3 installation in Ubuntu terminal.*

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**

**Question 4**