1	a) Create 2 text files. Read the contents in a single RDD. b) Create 2 CSV files. Read the contents in a single RDD.	**20**
2	Create two dataframes one for employee and other for dept. Perform a) Left outer join b) Full outer join c) Inner join	**20**
4	Viva	**5**
5	Journal	**5**

1

```
# Step 1: Install PySpark and set up the Spark Session
!pip install pyspark
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("PracticalExam_Q1").getOrCreate()
sc = spark.sparkContext

print("--- Spark Session Created ---\n")

# Step 2: Create the necessary files for the question
with open("file1.txt", "w") as f:
  f.write("this is the first text file\n")
  f.write("it has two lines\n")

with open("file2.txt", "w") as f:
  f.write("this is the second text file\n")

with open("data1.csv", "w") as f:
  f.write("id,value\n")
  f.write("1,a\n")
  f.write("2,b\n")

with open("data2.csv", "w") as f:
  f.write("id,value\n")
  f.write("3,c\n")
  f.write("4,d\n")

print("--- Sample files for Question 1 created successfully ---\n")

# --- SOLUTION ---

# a) Read 2 text files into a single RDD
print("\nReading contents from two text files into a single RDD:")
text_rdd = sc.textFile("file1.txt,file2.txt")
print("Result:", text_rdd.collect())

# a) Read 2 CSV files into a single RDD
print("\nReading contents from two CSV files into a single RDD:")
```

```
csv_df = spark.read.csv(["data1.csv", "data2.csv"], header=True)
csv_rdd = csv_df.rdd
print("Result:", csv_rdd.collect())
```

2.

```
# Assuming the SparkSession 'spark' is already created from the previous question.

# --- SETUP ---
# Create the employee and department DataFrames
emp_data = [(1, "Smith", 10), (2, "Rose", 20), (3, "Williams", 10), (4, "Jones", 30)]
dept_data = [("Finance", 10), ("Marketing", 20), ("Sales", 30), ("IT", 40)]

emp_df = spark.createDataFrame(emp_data, ["emp_id", "name", "dept_id"])
dept_df = spark.createDataFrame(dept_data, ["dept_name", "dept_id"])

print("Employee DataFrame:")
emp_df.show()
print("Department DataFrame:")
dept_df.show()

# --- SOLUTION ---

# a) Perform Left outer join
print("\na) Left Outer Join Result:")
emp_df.join(dept_df, on="dept_id", how="left_outer").show()

# b) Perform Full outer join
print("\nb) Full Outer Join Result:")
emp_df.join(dept_df, on="dept_id", how="full_outer").show()

# c) Perform Inner join
print("\nc) Inner Join Result:")
emp_df.join(dept_df, on="dept_id", how="inner").show()
```

1	For the following data and schema create a dataframe and perform the given operations Data: Seq(Row(Row("James;","","Smith"),"36636","M","20000"), Row(Row("Michael","Rose",""),"40288","M","40000"), Row(Row("Robert","","Williams"),"42114","M","10000"), Row(Row("Maria","Anne","Jones"),"39192","F","45000"), Row(Row("Jen","Mary","Brown"),"","F","-1")) Schema should have the columns as: firstname, middlename, lastname, dob, gender, expenses All columns will be of type String Perform the following operations: a) Change the data type of expenses to Integer b) Rename dob to DateOfBirth c) Create a column that has value expense*5	**20**
2	Create a data frame with a nested array column. Perform the following operations: a) Flatten nested array b) Explode nested array c) Convert array of string to string column.	**20**
3	Viva	**5**
4	Journal	**5**

1.

```
# Step 1: Install PySpark and set up the Spark Session
!pip install pyspark
from pyspark.sql import SparkSession, Row
from pyspark.sql.types import StructType, StructField, StringType, IntegerType
from pyspark.sql.functions import col

spark = SparkSession.builder.appName("PracticalExam_Slip2_Q1").getOrCreate()
print("--- Spark Session Created ---\n")

# --- SETUP ---
# Define the data and schema
data = [
   Row(name=Row(firstname="James", middlename="", lastname="Smith"), dob="36636", gender="M", expenses="20000"),
   Row(name=Row(firstname="Michael", middlename="Rose", lastname=""), dob="40288", gender="M", expenses="40000"),
   Row(name=Row(firstname="Robert", middlename="", lastname="Williams"), dob="42114", gender="M", expenses="10000"),
   Row(name=Row(firstname="Maria", middlename="Anne", lastname="Jones"), dob="39192", gender="F", expenses="45000"),
   Row(name=Row(firstname="Jen", middlename="Mary", lastname="Brown"), dob="", gender="F", expenses="-1")
]
```

```
schema = StructType([
    StructField("name", StructType([
        StructField("firstname", StringType(), True),
        StructField("middlename", StringType(), True),
        StructField("lastname", StringType(), True)
    ])),
    StructField("dob", StringType(), True),
    StructField("gender", StringType(), True),
    StructField("expenses", StringType(), True)
])

# Create the DataFrame
df = spark.createDataFrame(data, schema)
print("Original DataFrame:")
df.show(truncate=False)
df.printSchema()

# --- SOLUTION ---

# a) Change the data type of expenses to Integer
print("\na) Changing 'expenses' to Integer type:")
df_a = df.withColumn("expenses", col("expenses").cast(IntegerType()))
df_a.printSchema()
df_a.show()

# b) Rename dob to DateOfBirth
print("\nb) Renaming 'dob' to 'DateOfBirth':")
df_b = df_a.withColumnRenamed("dob", "DateOfBirth")
df_b.show()

# c) Create a column that has value expense*5
print("\nc) Creating a 'bonus' column with 'expenses * 5':")
df_c = df_b.withColumn("bonus", col("expenses") * 5)
df_c.show()
```

2.

```
# Assuming the SparkSession 'spark' is already created from the previous question.
from pyspark.sql.functions import col, flatten, explode, concat_ws
from pyspark.sql.types import StructType, StructField, StringType, ArrayType

# --- SETUP ---
# Create the DataFrame with a nested array
data = [
    ("James", [["Java", "Scala", "C++"], ["Spark", "Java"]]),
    ("Michael", [["Spark", "Java", "C++"], ["Spark", "Java"]]),
    ("Robert", [["CSharp", "VB"], ["Spark", "Python"]])
]
schema = StructType([
    StructField("name", StringType(), True),
    StructField("subjects", ArrayType(ArrayType(StringType())), True)
])
df = spark.createDataFrame(data, schema)
print("Original DataFrame with nested array:")
df.show(truncate=False)

# --- SOLUTION ---

# a) Flatten nested array
print("\na) Flattened nested array:")
df_a = df.withColumn("subjects_flat", flatten(col("subjects")))
df_a.show(truncate=False)

# b) Explode nested array
print("\nb) Exploded nested array:")
# Note: Exploding a nested array directly creates rows with the inner arrays.
# To explode to individual elements, you must flatten first.
df_b = df_a.withColumn("subject", explode(col("subjects_flat")))
df_b.show(truncate=False)

# c) Convert array of string to string column
print("\nc) Converted array to a single string column:")
df_c = df_a.withColumn("subjects_string", concat_ws(", ", col("subjects_flat")))
df_c.show(truncate=False)
```

1	a) Create a data frame with today’s date and timestamp b) Display the hours, minutes and seconds from the timestamp	**20**
2	For the following employee data showing name, dept and salary, perform the given operations: Data: ("James", "Sales", 3000), ("Michael", "Sales", 4600), ("Robert", "Sales", 4100), ("Maria", "Finance", 3000), ("James", "Sales", 3000), ("Scott", "Finance", 3300), ("Jen", "Finance", 3900), ("Jeff", "Marketing", 3000), ("Kumar", "Marketing", 2000), ("Saif", "Sales", 4100), (Jason", "Sales", 9000), ("Alice", "Finance", 3700), ("Jenniffer", "Finance", 8900), ("Jenson", "Marketing", 9000) a) Create a data frame for the above data b) Display average salary c) Display number of unique departments d) Display number of employees with unique salary	**20**
3	Viva	**5**
4	Journal	**5**

1

```
# Step 1: Install PySpark and set up the Spark Session
!pip install pyspark

from pyspark.sql import SparkSession
from pyspark.sql.functions import current_date, current_timestamp, hour, minute, second, col

spark = SparkSession.builder.appName("PracticalExam_Slip3_Q1").getOrCreate()
print("--- Spark Session Created ---\n")

# --- SOLUTION ---

# a) Create a data frame with today's date and timestamp
# We start with a dummy DataFrame with one row to add columns to.
df = spark.range(1)
df_with_time = df.withColumn("today_date", current_date()) \
            .withColumn("current_ts", current_timestamp())

print("a) DataFrame with current date and timestamp:")
df_with_time.show(truncate=False)

# b) Display the hours, minutes and seconds from the timestamp
```

```
time_parts_df = df_with_time.withColumn("hour", hour(col("current_ts"))) \
                         .withColumn("minute", minute(col("current_ts"))) \
                         .withColumn("second", second(col("current_ts")))

print("\nb) Timestamp parts extracted:")
time_parts_df.select("current_ts", "hour", "minute", "second").show(truncate=False)
```

2.

```
# Assuming the SparkSession 'spark' is already created from the previous question.
from pyspark.sql.functions import avg, countDistinct

# --- SETUP ---
# The provided data has a typo `(Jason"`, which has been corrected to `("Jason"`.
employee_data = [
    ("James", "Sales", 3000), ("Michael", "Sales", 4600),
    ("Robert", "Sales", 4100), ("Maria", "Finance", 3000),
    ("James", "Sales", 3000), ("Scott", "Finance", 3300),
    ("Jen", "Finance", 3900), ("Jeff", "Marketing", 3000),
    ("Kumar", "Marketing", 2000), ("Saif", "Sales", 4100),
    ("Jason", "Sales", 9000), ("Alice", "Finance", 3700),
    ("Jenniffer", "Finance", 8900), ("Jenson", "Marketing", 9000)
]
columns = ["name", "department", "salary"]

# --- SOLUTION ---

# a) Create a data frame for the above data
emp_df = spark.createDataFrame(employee_data, columns)
print("a) Employee DataFrame:")
emp_df.show()

# b) Display average salary
avg_salary_df = emp_df.select(avg("salary").alias("average_salary"))
print("\nb) Average salary:")
avg_salary_df.show()

# c) Display number of unique departments
unique_dept_df = emp_df.select(countDistinct("department").alias("unique_departments"))
print("\nc) Number of unique departments:")
unique_dept_df.show()

# d) Display number of employees with unique salary
# This is interpreted as the count of distinct salary values.
unique_salary_df = emp_df.select(countDistinct("salary").alias("unique_salary_count"))
print("\nd) Number of unique salary values:")
unique_salary_df.show()
```

1	a) Create a data frame containing today’s date, date 2022-01-31, date 2021-03-22, date 2024-01-31, date 2023-11-11. b) Store the date in the format MM-DD-YYYY. c) Display the dates in the format DD/MM/YYYY d) Find the number of months between each of the dates and today’s date	**20**
2	a) Create data frame with a column that contains JSON string. b) Convert the JSON string into Struct type or Map type. c) Extract the Data from JSON and create them as new columns. d) Convert MapType or Struct type to JSON string	**20**
3	Viva	**5**
4	Journal	**5**

1.

```
# Step 1: Install PySpark and set up the Spark Session
!pip install pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, current_date, to_date, date_format, lit, months_between
from pyspark.sql.types import StructType, StructField, StringType

spark = SparkSession.builder.appName("PracticalExam_Slip4_Q1").getOrCreate()
print("--- Spark Session Created ---\n")

# --- SOLUTION ---

# a) and b) Create a DataFrame with dates stored in MM-DD-YYYY format
date_data = [
  ("01-31-2022",),
  ("03-22-2021",),
  ("01-31-2024",),
  ("11-11-2023",)
]
# Add today's date to the list
todays_date_str = spark.range(1).select(date_format(current_date(), "MM-dd-yyyy")).first()[0]
date_data.append((todays_date_str,))

date_df = spark.createDataFrame(date_data, ["date_str_mmddyyyy"])
print("a) and b) DataFrame with dates in MM-DD-YYYY format:")
date_df.show()

# To perform date operations, first convert the strings to a proper DateType
df_with_dates = date_df.withColumn("date_obj", to_date(col("date_str_mmddyyyy"), "MM-dd-yyyy"))

# c) Display the dates in the format DD/MM/YYYY
print("\nc) Dates displayed in DD/MM/YYYY format:")
df_formatted = df_with_dates.withColumn("date_str_ddmmyyyy", date_format(col("date_obj"), "dd/MM/yyyy"))
df_formatted.select("date_str_mmddyyyy", "date_str_ddmmyyyy").show()

# d) Find the number of months between each date and today's date
print("\nd) Number of months between each date and today:")
df_months_between = df_with_dates.withColumn("months_from_today", months_between(current_date(),
col("date_obj")))
df_months_between.select("date_str_mmddyyyy", "months_from_today").show()
```

2.

```
# Assuming the SparkSession 'spark' is already created from the previous question.
from pyspark.sql.functions import from_json, to_json, col
from pyspark.sql.types import StructType, StructField, StringType

# --- SOLUTION ---

# a) Create DataFrame with a JSON string column
json_data = [
   (1, '{"name":"Alice", "city":"New York"}'),
   (2, '{"name":"Bob", "city":"Los Angeles"}')
]
json_df = spark.createDataFrame(json_data, ["id", "json_str"])
print("a) DataFrame with a JSON string column:")
json_df.show(truncate=False)

# b) Convert the JSON string into Struct type
# First, define the schema that matches the JSON structure
json_schema = StructType([
   StructField("name", StringType(), True),
   StructField("city", StringType(), True)
])
df_with_struct = json_df.withColumn("parsed_struct", from_json(col("json_str"), json_schema))
print("\nb) DataFrame with JSON converted to a StructType column:")
df_with_struct.printSchema()
df_with_struct.show(truncate=False)

# c) Extract the Data from JSON and create them as new columns
df_extracted = df_with_struct.withColumn("name", col("parsed_struct.name")) \
                         .withColumn("city", col("parsed_struct.city"))
print("\nc) DataFrame with JSON data extracted into new columns:")
df_extracted.select("id", "name", "city").show()

# d) Convert Struct type to JSON string
df_converted_back = df_extracted.withColumn("new_json_str", to_json(struct("name", "city")))
print("\nd) DataFrame with columns converted back to a JSON string:")
df_converted_back.select("id", "new_json_str").show(truncate=False)
```

1	**Create a data frame containing today’s date, date 2022-01-31, date 2021-03-22, date 2024-01-31** **Add 5 days to each date and display the result.** **Display the new dates after subtracting 10 days from each date.** **For each date, display year, month, dayofweek, dayofmonth, dayofyear, next_day,weekofyear**	**20**
2	**Refer to the employee.json file. Perform the following operations:** **Print the names of employees above 25 years of age.** **Print the number of employees of different ages.**	**20**
3	**Viva**	**5**
4	**Journal**	**5**

1.

```
# Step 1: Install PySpark and set up the Spark Session
!pip install pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, lit, to_date, date_add, date_sub, year, month, dayofweek, dayofmonth, dayofyear, next_day, weekofyear, current_date

spark = SparkSession.builder.appName("PracticalExam_Slip5_Q1").getOrCreate()
print("--- Spark Session Created ---\n")

# --- SOLUTION ---

# a) Create a DataFrame with dates
# We create the DataFrame from string literals and convert them to DateType
date_data = ["2022-01-31", "2021-03-22", "2024-01-31"]
df_dates = spark.createDataFrame(date_data, "string").withColumnRenamed("value", "date_str")
df_dates = df_dates.union(spark.range(1).select(current_date().cast("string").alias("date_str"))) # Add today's date
df = df_dates.withColumn("date", to_date(col("date_str")))
print("a) Original DataFrame with dates:")
df.show()

# b) Add 5 days to each date
print("\nb) Dates after adding 5 days:")
df_plus_5 = df.withColumn("date_plus_5", date_add(col("date"), 5))
df_plus_5.show()

# c) Display the new dates after subtracting 10 days from each date
```

```
print("\nc) Dates after subtracting 10 days:")
df_minus_10 = df.withColumn("date_minus_10", date_sub(col("date"), 10))
df_minus_10.show()

# d) Display various parts for each date
print("\nd) Various date parts:")
df_parts = df.withColumn("year", year(col("date"))) \
            .withColumn("month", month(col("date"))) \
            .withColumn("dayofweek", dayofweek(col("date"))) \
            .withColumn("dayofmonth", dayofmonth(col("date"))) \
            .withColumn("dayofyear", dayofyear(col("date"))) \
            .withColumn("next_day", next_day(col("date"), "Sunday")) \
            .withColumn("weekofyear", weekofyear(col("date")))
df_parts.show()
```

2.

```
# Assuming the SparkSession 'spark' is already created from the previous question.
from pyspark.sql.functions import col

# --- SETUP ---
# Create the employee.json file in the Colab environment
json_content = """
{"name": "Michael", "age": 30}
{"name": "Andy", "age": 24}
{"name": "Justin", "age": 28}
{"name": "Berta", "age": 35}
{"name": "David", "age": 28}
"""
with open("employee.json", "w") as f:
   f.write(json_content)
print("--- employee.json file created successfully ---\n")

# --- SOLUTION ---
# Read the JSON file into a DataFrame
emp_df = spark.read.json("employee.json")
```

```
print("Original Employee DataFrame:")
emp_df.show()

# a) Print the names of employees above 25 years of age
print("\na) Names of employees older than 25:")
emp_df.filter(col("age") > 25).select("name").show()

# b) Print the number of employees of different ages
print("\nb) Number of employees for each age:")
emp_df.groupBy("age").count().show()
```

1	Create two dataframes one for employee and other for dept. Perform a) Left anti join b) Self join c) Left semi join	**20**
2	a) Create two case classes – Student and Address b) Create schema from these case classes	**20**
3	Viva	**5**
4	Journal	**5**

1.

```
# Step 1: Install PySpark and set up the Spark Session
!pip install pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

spark = SparkSession.builder.appName("PracticalExam_Slip6_Q1").getOrCreate()
print("--- Spark Session Created ---\n")

# --- SETUP ---
# Create the two DataFrames
emp_data = [(1, "Smith", 10), (2, "Rose", 20), (3, "Williams", 10), (4, "Jones", 30), (5, "Brown", 50)]
dept_data = [("Finance", 10), ("Marketing", 20), ("Sales", 30), ("IT", 40)]

emp_df = spark.createDataFrame(emp_data, ["emp_id", "name", "dept_id"])
dept_df = spark.createDataFrame(dept_data, ["dept_name", "dept_id"])

print("Employee DataFrame:")
emp_df.show()
print("Department DataFrame:")
dept_df.show()

# --- SOLUTION ---

# a) Left Anti Join
# This join returns only the rows from the left DataFrame that do not have a match in the right DataFrame.
print("\na) Left Anti Join (Employees in departments not in the dept table):")
emp_df.join(dept_df, on="dept_id", how="left_anti").show()

# b) Self Join
# This is joining a DataFrame to itself. You must use aliases to distinguish them.
# Example: Find employees who have the same department ID.
print("\nb) Self Join (Find pairs of employees in the same department):")
df1 = emp_df.alias("df1")
df2 = emp_df.alias("df2")
# We add df1.emp_id < df2.emp_id to avoid duplicate pairs and self-joins.
self_join_df = df1.join(df2, on="dept_id") \
            .where(col("df1.emp_id") < col("df2.emp_id")) \
            .select(col("df1.name").alias("emp1"), col("df2.name").alias("emp2"), "dept_id")
self_join_df.show()

# c) Left Semi Join
# This join is similar to an inner join, but it only returns the columns from the left DataFrame.
print("\nc) Left Semi Join (Employees in departments that exist in the dept table):")
emp_df.join(dept_df, on="dept_id", how="left_semi").show()
```

2.

```
# Assuming the SparkSession 'spark' is already created from the previous question.
from pyspark.sql import Row
from pyspark.sql.types import StructType, StructField, StringType, IntegerType

# --- EXPLANATION ---
# "Case classes" are a feature of the Scala language. They provide a concise way to define
# classes that are primarily used for holding data. Spark can automatically infer a schema from them in Scala.
#
# The direct equivalent in Python is to use a standard Python class or, more commonly,
# to define the schema explicitly using StructType and StructField.
# This practical demonstrates the PySpark way of achieving the same goal.

# --- SOLUTION ---

# a) Equivalent of creating "case classes"
# In Python, we can represent the data structure using standard classes or dictionaries.
# Here, we'll represent the data as nested Row objects, which is a common pattern.
student_data = [
    Row(name="John", age=20, address=Row(city="New York", zip_code="10001")),
    Row(name="Jane", age=22, address=Row(city="Los Angeles", zip_code="90001"))
]

# b) Create schema from the data structure
# We explicitly define the schema to match our data structure.
address_schema = StructType([
    StructField("city", StringType(), True),
    StructField("zip_code", StringType(), True)
])

student_schema = StructType([
    StructField("name", StringType(), True),
    StructField("age", IntegerType(), True),
    StructField("address", address_schema, True)
])

print("b) Explicitly created schema:")
student_schema.prettyJson()

# Now, create a DataFrame using this data and schema
student_df = spark.createDataFrame(student_data, student_schema)

print("\nDataFrame created from the schema:")
student_df.show(truncate=False)
student_df.printSchema()
```

1	Create a data frame with data that follows the below given schema emp_id, dept, properties (a structure containing salary and location) Return the map keys from spark SQL for this data frame	**20**
2	For the following employee data showing name, dept and salary, perform the given operations: Data: ("James", "Sales", 3000), ("Michael", "Sales", 4600), ("Robert", "Sales", 4100), ("Maria", "Finance", 3000), ("James", "Sales", 3000), ("Scott", "Finance", 3300), ("Jen", "Finance", 3900), ("Jeff", "Marketing", 3000), ("Kumar", "Marketing", 2000), ("Saif", "Sales", 4100), (Jason", "Sales", 9000), ("Alice", "Finance", 3700), ("Jenniffer", "Finance", 8900), ("Jenson", "Marketing", 9000) a) Create a data frame for the above data b) Find the highest salary value c) Find the lowest salary value d) Find the standard deviation for the salary	**20**
3	Viva	**5**
4	Journal	**5**

1.

```
# Step 1: Install PySpark and set up the Spark Session
!pip install pyspark
from pyspark.sql import SparkSession, Row
from pyspark.sql.functions import col

spark = SparkSession.builder.appName("PracticalExam_Slip7_Q1").getOrCreate()
print("--- Spark Session Created ---\n")

# --- EXPLANATION ---
# The question asks to "Return the map keys", but the `properties` column is a struct, not a map.
# A struct has fixed fields (like keys), while a map can have arbitrary key-value pairs.
# The correct interpretation is to access the fields of the struct, not "map keys".

# --- SETUP ---
# Create a DataFrame with a struct column
data = [
   (1, "Finance", Row(salary=90000, location="New York")),
   (2, "Marketing", Row(salary=80000, location="Chicago")),
   (3, "Sales", Row(salary=120000, location="New York"))
]
df = spark.createDataFrame(data, ["emp_id", "dept", "properties"])
print("Original DataFrame with Struct column:")
df.show(truncate=False)
df.printSchema()
```

```
# --- SOLUTION ---
# Access the fields of the 'properties' struct
print("\nAccessing the fields ('keys') of the 'properties' struct:")
df.select(
    col("emp_id"),
    col("properties.salary"),
    col("properties.location")
).show()
```

2.

```
# Assuming the SparkSession 'spark' is already created from the previous question.
from pyspark.sql.functions import max, min, stddev

# --- SETUP ---
# The provided data has a typo `(Jason"`, which has been corrected to `("Jason"`.
employee_data = [
    ("James", "Sales", 3000), ("Michael", "Sales", 4600),
    ("Robert", "Sales", 4100), ("Maria", "Finance", 3000),
    ("James", "Sales", 3000), ("Scott", "Finance", 3300),
    ("Jen", "Finance", 3900), ("Jeff", "Marketing", 3000),
    ("Kumar", "Marketing", 2000), ("Saif", "Sales", 4100),
    ("Jason", "Sales", 9000), ("Alice", "Finance", 3700),
    ("Jenniffer", "Finance", 8900), ("Jenson", "Marketing", 9000)
]
columns = ["name", "department", "salary"]

# --- SOLUTION ---

# a) Create a data frame for the above data
emp_df = spark.createDataFrame(employee_data, columns)
print("a) Employee DataFrame:")
emp_df.show()

# b) Find the highest salary value
print("\nb) Highest Salary:")
emp_df.select(max("salary").alias("highest_salary")).show()

# c) Find the lowest salary value
print("\nc) Lowest Salary:")
emp_df.select(min("salary").alias("lowest_salary")).show()

# d) Find the standard deviation for the salary
print("\nd) Standard Deviation of Salary:")
emp_df.select(stddev("salary").alias("stddev_salary")).show()
```

1	Create a data frame with a nested array column. Perform the following operations: a) Flatten nested array b) Explode nested array c) Convert array of string to string column.	**20**
2	a) Create data frame with a column that contains JSON string. b) Convert the JSON string into Struct type or Map type. c) Extract the Data from JSON and create them as new columns. d) Convert MapType or Struct type to JSON string	**20**
3	Viva	**5**
4	Journal	**5**

1.

```
# Step 1: Install PySpark and set up the Spark Session
!pip install pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, flatten, explode, concat_ws
from pyspark.sql.types import StructType, StructField, StringType, ArrayType

spark = SparkSession.builder.appName("PracticalExam_Slip8_Q1").getOrCreate()
print("--- Spark Session Created ---\n")

# --- SETUP ---
# Create the DataFrame with a nested array
data = [
  ("James", [["Java", "Scala", "C++"], ["Spark", "Java"]]),
  ("Michael", [["Spark", "Java", "C++"], ["Spark", "Java"]]),
  ("Robert", [["CSharp", "VB"], ["Spark", "Python"]])
]
schema = StructType([
  StructField("name", StringType(), True),
  StructField("subjects", ArrayType(ArrayType(StringType())), True)
])
df = spark.createDataFrame(data, schema)
print("Original DataFrame with nested array:")
df.show(truncate=False)

# --- SOLUTION ---
# a) Flatten nested array
print("\na) Flattened nested array:")
df_a = df.withColumn("subjects_flat", flatten(col("subjects")))
df_a.show(truncate=False)

# b) Explode nested array
print("\nb) Exploded nested array to rows:")
# To get individual elements, you must flatten first.
df_b = df_a.withColumn("subject", explode(col("subjects_flat")))
df_b.show(truncate=False)

# c) Convert array of string to string column
print("\nc) Converted array to a single string column:")
df_c = df_a.withColumn("subjects_string", concat_ws(", ", col("subjects_flat")))
df_c.show(truncate=False)
```

2.

```
# Assuming the SparkSession 'spark' is already created from the previous question.
from pyspark.sql.functions import from_json, to_json, col, struct
from pyspark.sql.types import StructType, StructField, StringType

# --- SOLUTION ---

# a) Create DataFrame with a JSON string column
json_data = [
    (1, '{"name":"Alice", "city":"New York"}'),
    (2, '{"name":"Bob", "city":"Los Angeles"}')
]
json_df = spark.createDataFrame(json_data, ["id", "json_str"])
print("a) DataFrame with a JSON string column:")
json_df.show(truncate=False)

# b) Convert the JSON string into Struct type
# Define the schema that matches the JSON structure
json_schema = StructType([
   StructField("name", StringType(), True),
   StructField("city", StringType(), True)
])
df_with_struct = json_df.withColumn("parsed_struct", from_json(col("json_str"), json_schema))
print("\nb) DataFrame with JSON converted to a StructType column:")
df_with_struct.printSchema()
df_with_struct.show(truncate=False)

# c) Extract the Data from JSON and create them as new columns
df_extracted = df_with_struct.withColumn("name", col("parsed_struct.name")) \
                  .withColumn("city", col("parsed_struct.city"))
print("\nc) DataFrame with JSON data extracted into new columns:")
df_extracted.select("id", "name", "city").show()

# d) Convert Struct type to JSON string
df_converted_back = df_extracted.withColumn("new_json_str", to_json(struct("name", "city")))
print("\nd) DataFrame with columns converted back to a JSON string:")
df_converted_back.select("id", "new_json_str").show(truncate=False)
```

1	Create a Spark RDD using 5 different Functions	**20**
2	Write example for following Spark RDD Actions: a. aggregate b. treeAggregate c. fold d. reduce e. collect	**20**
3	Viva	**5**
4	Journal	**5**

1.

```
# Step 1: Install PySpark and set up the Spark Session
!pip install pyspark
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("PracticalExam_Slip9_Q1").getOrCreate()
sc = spark.sparkContext
print("--- Spark Session Created ---\n")

# --- SETUP ---
# Create a sample text file
with open("sample_rdd.txt", "w") as f:
   f.write("line one\n")
   f.write("line two\n")
   f.write("line three\n")
print("--- Sample file for RDD creation created ---\n")


# --- SOLUTION ---
print("--- Creating RDDs using 5 different methods ---\n")

# 1. Using sc.parallelize() on a list
list_data = [1, 2, 3, 4, 5]
rdd1 = sc.parallelize(list_data)
print("1. RDD from a Python list (parallelize):")
print(rdd1.collect())

# 2. Using sc.textFile() to read a text file
```

```
rdd2 = sc.textFile("sample_rdd.txt")
print("\n2. RDD from a text file (textFile):")
print(rdd2.collect())

# 3. Using sc.range()
rdd3 = sc.range(1, 6) # Creates an RDD with elements 1, 2, 3, 4, 5
print("\n3. RDD from a range (range):")
print(rdd3.collect())

# 4. By transforming an existing RDD (e.g., using map)
rdd4 = rdd1.map(lambda x: x * x)
print("\n4. RDD by transforming another RDD (map):")
print(rdd4.collect())

# 5. From a DataFrame
df = spark.createDataFrame([("a", 1), ("b", 2)], ["letter", "number"])
rdd5 = df.rdd
print("\n5. RDD from a DataFrame (.rdd):")
print(rdd5.collect())
```

2.

```
# Assuming the SparkContext 'sc' is already created from the previous question.

# --- SETUP ---
rdd = sc.parallelize([1, 2, 3, 4, 5])
print("Using the following RDD for actions:", rdd.collect())

# --- SOLUTION ---

# a) aggregate
# Action: Sums elements and adds an initial value to each partition and then to the final result.
# (sum_of_elements + initial_value * num_partitions) + initial_value
# Here: (1+2+3+4+5) -> 15. Let's assume 2 partitions.
# Partition 1: 1+2 -> 3. Partition 2: 3+4+5 -> 12.
# Add initial value 10 to each partition sum: (3+10) + (12+10) = 45
seqOp = (lambda x, y: x + y)
combOp = (lambda x, y: x + y)
agg_result = rdd.aggregate(0, seqOp, combOp) # Using 0 as initial value is same as reduce
print("\na) aggregate (sum):", agg_result)

# b) treeAggregate
# Similar to aggregate but performs aggregation in a tree-like pattern, which is more efficient for large datasets.
tree_agg_result = rdd.treeAggregate(0, seqOp, combOp)
print("\nb) treeAggregate (sum):", tree_agg_result)

# c) fold
# Similar to reduce but takes a "zero value" to be used for the initial call in each partition.
fold_result = rdd.fold(0, lambda x, y: x + y)
print("\nc) fold (sum):", fold_result)

# d) reduce
# Aggregates the elements of the RDD using a specified commutative and associative binary operator.
reduce_result = rdd.reduce(lambda x, y: x + y)
print("\nd) reduce (sum):", reduce_result)

# e) collect
# Returns all the elements of the RDD as a list to the driver program.
collect_result = rdd.collect()
print("\ne) collect:", collect_result)
```

1	Write example for following Spark RDD Actions: a. count b. countApproxDistinct c. first d. top e. Min	**20**
2	Write Spark Pair RDD Functions.	**20**
3	Viva	**5**
4	Journal	**5**

1.

```
# Step 1: Install PySpark and set up the Spark Session
!pip install pyspark
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("PracticalExam_Slip10_Q1").getOrCreate()
sc = spark.sparkContext
print("--- Spark Session Created ---\n")

# --- SETUP ---
rdd = sc.parallelize([2, 5, 1, 3, 4, 2, 5])
print("Using the following RDD for actions:", rdd.collect())

# --- SOLUTION ---

# a) count
# Returns the number of elements in the RDD.
count_result = rdd.count()
print("\na) count:", count_result)

# b) countApproxDistinct
# Returns the approximate number of distinct elements. Useful for large datasets.
approx_distinct_result = rdd.countApproxDistinct()
print("\nb) countApproxDistinct:", approx_distinct_result)
# Note: The exact distinct count is 5 (1, 2, 3, 4, 5)

# c) first
# Returns the first element of the RDD.
first_result = rdd.first()
print("\nc) first:", first_result)

# d) top
# Returns the top n elements from an RDD, ordered in descending order.
top_3_result = rdd.top(3)
print("\nd) top(3):", top_3_result)

# e) min
# Returns the minimum element of the RDD.
min_result = rdd.min()
print("\ne) min:", min_result)
```

2.

```
# Assuming the SparkContext 'sc' is already created from the previous question.

# --- SETUP ---
# A Pair RDD is an RDD where each element is a key-value tuple.
# Let's create a sample Pair RDD.
data = [("apple", 1), ("banana", 2), ("apple", 3), ("orange", 4), ("banana", 5)]
pair_rdd = sc.parallelize(data)
print("Using the following Pair RDD:", pair_rdd.collect())

# --- SOLUTION ---
# Here are examples of common Pair RDD functions.

# 1. reduceByKey()
# Merges the values for each key using an associative and commutative reduce function.
print("\n1. reduceByKey (sum of values for each key):")
reduced_rdd = pair_rdd.reduceByKey(lambda a, b: a + b)
print(reduced_rdd.collect())

# 2. groupByKey()
# Groups the values for each key in the RDD into a single sequence.
print("\n2. groupByKey:")
grouped_rdd = pair_rdd.groupByKey()
# The result contains an iterable object, so we map it to a list for printing.
print(grouped_rdd.mapValues(list).collect())

# 3. sortByKey()
# Sorts the RDD by key.
print("\n3. sortByKey (ascending):")
sorted_rdd = pair_rdd.sortByKey()
print(sorted_rdd.collect())

# 4. keys() and values()
# Return an RDD of just the keys or just the values.
print("\n4. keys() and values():")
keys_rdd = pair_rdd.keys()
values_rdd = pair_rdd.values()
print("Keys:", keys_rdd.collect())
print("Values:", values_rdd.collect())

# 5. join()
# Joins two Pair RDDs based on their keys.
other_data = [("apple", "red"), ("orange", "orange"), ("grape", "purple")]
other_pair_rdd = sc.parallelize(other_data)
print("\n5. join:")
joined_rdd = pair_rdd.join(other_pair_rdd)
print(joined_rdd.collect())
```

1	Get new dates by adding 4 days, and subtracting 7 days in below dates "2020-01-02","2023-01-15","2025-01-30"	**20**
2	Use the Operation Read CSV file on RDD with Scala operation	**20**
3	Viva	**5**
4	Journal	**5**

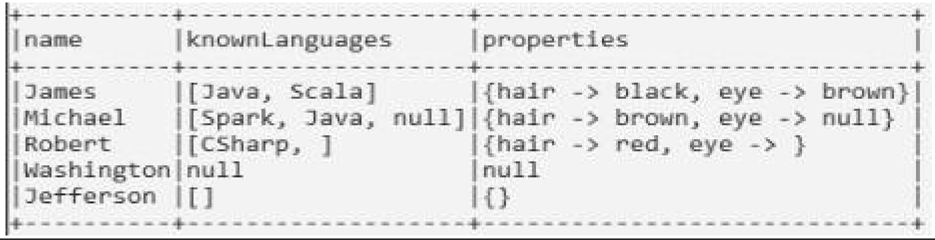
1.

```
# Step 1: Install PySpark and set up the Spark Session
!pip install pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, to_date, date_add, date_sub

spark = SparkSession.builder.appName("PracticalExam_Slip11_Q1").getOrCreate()
print("--- Spark Session Created ---\n")

# --- SETUP ---
date_data = ["2020-01-02", "2023-01-15", "2025-01-30"]
df = spark.createDataFrame(date_data, "string").withColumnRenamed("value", "date_str")
# Convert strings to DateType
df = df.withColumn("date", to_date(col("date_str")))
print("Original DataFrame with dates:")
df.show()

# --- SOLUTION ---

# Add 4 days to each date
df_plus_4 = df.withColumn("date_plus_4_days", date_add(col("date"), 4))
print("\nDates after adding 4 days:")
df_plus_4.show()

# Subtract 7 days from each date
df_minus_7 = df.withColumn("date_minus_7_days", date_sub(col("date"), 7))
print("\nDates after subtracting 7 days:")
df_minus_7.show()
```

2.

```
# Assuming the SparkContext 'sc' and SparkSession 'spark' are already created.

# --- EXPLANATION ---
# The question asks to use a "Scala operation". In PySpark, we use Python operations (like lambda functions).
# The most common way to read a CSV is with a DataFrame, but to fulfill the "on RDD" requirement,
# we will read the file as a text RDD and then parse it.

# --- SETUP ---
# Create a sample CSV file
with open("student_data.csv", "w") as f:
   f.write("id,name,score\n")
   f.write("1,Alice,85\n")
   f.write("2,Bob,90\n")
   f.write("3,Cathy,78\n")
print("--- student_data.csv created successfully ---\n")
```

```
# --- SOLUTION ---

# 1. Read the CSV file into a text RDD
text_rdd = sc.textFile("student_data.csv")
print("RDD as raw text lines:")
print(text_rdd.collect())

# 2. Get the header and filter it out
header = text_rdd.first()
data_rdd = text_rdd.filter(lambda line: line != header)
print("\nRDD after removing the header:")
print(data_rdd.collect())


# 3. Use an RDD operation (map) to parse the data
# This is the equivalent of a "Scala operation" in PySpark.
parsed_rdd = data_rdd.map(lambda line: line.split(","))
# Convert score to an integer
parsed_rdd = parsed_rdd.map(lambda parts: (int(parts[0]), parts[1], int(parts[2])))

print("\nRDD after parsing and transforming with a map operation:")
print(parsed_rdd.collect())

# Example: Filter for scores above 80
high_scores_rdd = parsed_rdd.filter(lambda x: x[2] > 80)
print("\nResult of another operation (filter for score > 80):")
print(high_scores_rdd.collect())
```

<table>
<tr><td>1</td><td>Create table as follows containing array and map operations

<pre>
+----------+-------------------+-----------------------------+
|name |knownLanguages |properties |
+----------+-------------------+-----------------------------+
James	[Java, Scala]	{hair -> black, eye -> brown}
Michael	[Spark, Java, null]	{hair -> brown, eye -> null}
Robert	[CSharp,]	{hair -> red, eye -> }
Washington	null	null
Jefferson	[]	{}
+----------+-------------------+-----------------------------+
</pre></td><td><b>20</b></td></tr>
<tr><td>2</td><td>Find current timestamp and hour, Minute, second separately for today's date</td><td><b>20</b></td></tr>
<tr><td>3</td><td>Viva</td><td><b>5</b></td></tr>
<tr><td>4</td><td>Journal</td><td><b>5</b></td></tr>
</table>

1.

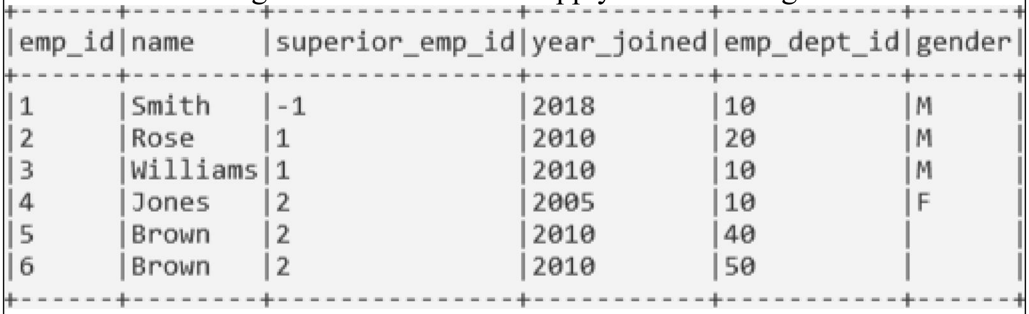
```
# Step 1: Install PySpark and set up the Spark Session
!pip install pyspark
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType, ArrayType, MapType

spark = SparkSession.builder.appName("PracticalExam_Slip12_Q1").getOrCreate()
print("--- Spark Session Created ---\n")

# --- EXPLANATION ---
# The table in the question is poorly formatted but describes a DataFrame with three columns:
# 'name' (String), 'knownLanguages' (Array of Strings), and 'properties' (Map of String to String).
# We will create the data based on this interpretation.

# --- SOLUTION ---

# Define the data, handling nulls and empty values from the question
data = [
    ("James", ["Java", "Scala"], {"hair": "black", "eye": "brown"}),
    ("Michael", ["Spark", "Java", None], {"hair": "brown", "eye": None}),
    ("Robert", ["CSharp", None], {"hair": "red", "eye": None})
]

# Define the schema for the DataFrame
schema = StructType([
    StructField("name", StringType(), True),
    StructField("knownLanguages", ArrayType(StringType()), True),
    StructField("properties", MapType(StringType(), StringType()), True)
])

# Create the DataFrame
df = spark.createDataFrame(data, schema)

print("Created DataFrame:")
df.show(truncate=False)
df.printSchema()
```

2.

```
# Assuming the SparkSession 'spark' is already created from the previous question.
from pyspark.sql.functions import current_timestamp, hour, minute, second, col

# --- SOLUTION ---

# Create a dummy DataFrame with one row to demonstrate the functions
df = spark.range(1)

# Add a column with the current timestamp
df_with_ts = df.withColumn("current_timestamp", current_timestamp())
print("DataFrame with current timestamp:")
df_with_ts.show(truncate=False)


# Extract hour, minute, and second into separate columns
time_parts_df = df_with_ts.withColumn("hour", hour(col("current_timestamp"))) \
                .withColumn("minute", minute(col("current_timestamp"))) \
                .withColumn("second", second(col("current_timestamp")))

print("\nTimestamp with hour, minute, and second extracted:")
time_parts_df.select("current_timestamp", "hour", "minute", "second").show(truncate=False)
```

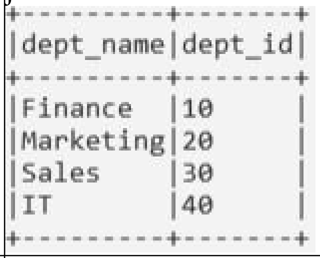
<table>
<tr><td>1</td><td>Write a Maven dependencies for writing and Reading Avro Data File</td><td><b>20</b></td></tr>
<tr><td>2</td><td>

Create the following two data frames and apply Inner and Right Outer



```
+------+--------+---------------+-----------+-----------+------+
|emp_id|name    |superior_emp_id|year_joined|emp_dept_id|gender|
+------+--------+---------------+-----------+-----------+------+
|1     |Smith   |-1             |2018       |10         |M     |
|2     |Rose    |1              |2010       |20         |M     |
|3     |Williams|1              |2010       |10         |M     |
|4     |Jones   |2              |2005       |10         |F     |
|5     |Brown   |2              |2010       |40         |      |
|6     |Brown   |2              |2010       |50         |      |
+------+--------+---------------+-----------+-----------+------+
```

join.   20M



```
+---------+-------+
|dept_name|dept_id|
+---------+-------+
|Finance  |10     |
|Marketing|20     |
|Sales    |30     |
|IT       |40     |
+---------+-------+
```

</td><td><b>20</b></td></tr>
<tr><td>3</td><td>Viva</td><td><b>5</b></td></tr>
<tr><td>4</td><td>Journal</td><td><b>5</b></td></tr>
</table>

1.

1. **To launch pyspark from the command line**

   Bash:

   ```
   pyspark --packages org.apache.spark:spark-avro_2.12:3.5.1
   ```

2. **To configure it within a Colab notebook or Python script:**

   ```
   from pyspark.sql import SparkSession

   spark = SparkSession.builder \
     .appName("AvroExample") \
     .config("spark.jars.packages", "org.apache.spark:spark-avro_2.12:3.5.1") \
     .getOrCreate()

   print("SparkSession created with Avro package.")
   ```

2.

```
# Step 1: Install PySpark and set up the Spark Session
!pip install pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

spark = SparkSession.builder.appName("PracticalExam_Slip13_Q2").getOrCreate()
print("--- Spark Session Created ---\n")

# --- SETUP ---
# Create the two DataFrames based on the provided tables.
emp_data = [
  (1, "Smith", None, 2018, 10, "M"),
```

```
   (2, "Rose", 1, 2010, 20, "M"),
   (3, "Williams", 1, 2010, 10, "M"),
   (4, "Jones", 2, 2005, 10, "F"),
   (5, "Brown", 2, 2010, 40, None),
   (6, "Brown", 2, 2010, 50, None) # Corrected from the original slip's typo
]
emp_columns = ["emp_id", "name", "superior_emp_id", "year_joined", "emp_dept_id", "gender"]
emp_df = spark.createDataFrame(emp_data, emp_columns)

dept_data = [("Finance", 10), ("Marketing", 20), ("Sales", 30), ("IT", 40)]
dept_columns = ["dept_name", "dept_id"]
dept_df = spark.createDataFrame(dept_data, dept_columns)

print("Employee DataFrame:")
emp_df.show()
print("Department DataFrame:")
dept_df.show()

# --- SOLUTION ---
# We need to join on the department ID. The columns have different names,
# so we need to specify the join condition explicitly.
join_condition = emp_df.emp_dept_id == dept_df.dept_id

# a) Inner Join
print("\na) Inner Join Result:")
emp_df.join(dept_df, join_condition, "inner").show()

# b) Right Outer Join
print("\nb) Right Outer Join Result:")
emp_df.join(dept_df, join_condition, "right_outer").show()
```