PySpark Practical Exam Paper

Advanced Transformations and Actions

Duration: 90 minutes
Total Questions: 10
Difficulty: Intermediate to Advanced
Marks: 10 × 10 = 100 marks

QUESTION 1: Load and Transform Data (CSV Loading + Column Operations)

Marks: 10

Question

You are given a CSV file `employees.csv` with the following sample data:

```
emp_id,emp_name,department,salary,joining_date
1,Alice,IT,75000,2022-01-15
2,Bob,HR,65000,2021-03-20
3,Clara,IT,85000,2020-06-10
4,David,Finance,90000,2019-11-05
5,Eve,IT,78000,2023-02-28
```

Task:

1. Load the CSV file with header and infer schema.
2. Rename the column `emp_name` to `employee_name`.
3. Add a new column `bonus_amount` calculated as 10% of salary.
4. Add another column `department_upper` that converts the department name

to uppercase.
5. Select and display only: emp_id, employee_name, department_upper, salary, bonus_amount.

Answer

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, lit, upper

# Create SparkSession
spark = SparkSession.builder.appName("EmployeeTransform").getOrCreate()

# Load CSV with header and schema inference
df = spark.read.option("header", True).option("inferSchema", True).csv("path/to/employe

# Rename column
df = df.withColumnRenamed("emp_name", "employee_name")

# Add bonus column (10% of salary)
df = df.withColumn("bonus_amount", col("salary") * 0.10)

# Add department uppercase column
df = df.withColumn("department_upper", upper(col("department")))

# Select required columns and display
result_df = df.select("emp_id", "employee_name", "department_upper", "salary", "bonus_
result_df.show(truncate=False)
```

Explanation

- **spark.read.option("header", True):** Treats the first row as column names instead of data.
- **inferSchema, True:** Automatically detects data types (emp_id as integer, salary as double).
- **withColumnRenamed():** Changes column name from emp_name to employee_name for clarity.

- **col("salary") * 0.10:** References the salary column and multiplies by 0.10 (10%) to calculate bonus.
- **upper(col("department")):** Converts text to uppercase (IT, HR, FINANCE, etc.).
- **select():** Transformation that picks specific columns, reducing data footprint.
- **show():** Action that triggers computation and displays results.

Key Concepts Tested

- CSV file reading with options
- Column renaming
- Column creation with arithmetic operations
- String functions (upper)
- Column selection

QUESTION 2: Handling Null Values and Data Cleansing

Marks: 10

Question

You have a customer dataset with missing values:

```
customer_id,customer_name,email,phone,city,age
101,Alice,alice@email.com,9876543210,Mumbai,28
102,Bob,,9876543211,Delhi,NULL
103,Clara,clara@email.com,,Bangalore,35
104,,david@email.com,9876543213,Chennai,42
105,Eve,eve@email.com,9876543214,NULL,29
```

Task:

1. Load the data and show the null values.
2. Remove rows where customer_name is NULL.
3. Fill NULL values in the phone column with "Unknown".
4. Fill NULL values in the city column with "Not Specified".
5. Drop the email column entirely.
6. Display the cleaned data.

Answer

```python
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("CustomerCleansing").getOrCreate()

# Create DataFrame with sample data
data = [
    (101, "Alice", "alice@email.com", "9876543210", "Mumbai", 28),
    (102, "Bob", None, "9876543211", "Delhi", None),
    (103, "Clara", "clara@email.com", None, "Bangalore", 35),
    (104, None, "david@email.com", "9876543213", "Chennai", 42),
    (105, "Eve", "eve@email.com", "9876543214", None, 29)
]

schema = ["customer_id", "customer_name", "email", "phone", "city", "age"]
df = spark.createDataFrame(data, schema)

# Show null values (all data for inspection)
print("Original Data with Nulls:")
df.show(truncate=False)

# Remove rows where customer_name is NULL
df_cleaned = df.dropna(subset=["customer_name"])

# Fill NULL in phone with "Unknown"
df_cleaned = df_cleaned.fillna({"phone": "Unknown", "city": "Not Specified"})

# Drop email column
df_cleaned = df_cleaned.drop("email")

# Display cleaned data
print("\nCleaned Data:")
df_cleaned.show(truncate=False)
```

Explanation

- **dropna(subset=["customer_name"]):** Removes entire rows where customer_name is NULL, ensuring no invalid customer records remain.
- **fillna({...}):** Dictionary-based fill replaces NULL values in specific columns:

  - phone → "Unknown" (indicating missing contact)
  - city → "Not Specified" (placeholder for unknown location)
- **drop("email"):** Removes the email column entirely, freeing memory if not needed.
- **show():** Action that executes the transformation chain and displays final result.

Key Concepts Tested

- Null value detection
- Dropping rows with nulls in specific columns
- Filling nulls with default values
- Column dropping
- Data cleansing workflows

QUESTION 3: Total Purchases by Customer (GroupBy + Aggregation)

Marks: 10

Question

You have a sales transaction dataset:

```
transaction_id,customer_id,customer_name,product,amount,purchase_date
1,101,Alice,Laptop,50000,2025-01-10
2,102,Bob,Mobile,25000,2025-01-11
3,101,Alice,Keyboard,5000,2025-01-12
4,103,Clara,Mouse,1500,2025-01-13
5,101,Alice,Monitor,15000,2025-01-14
6,102,Bob,Keyboard,5000,2025-01-15
```

Task:

1. Load the transaction data.
2. Group by `customer_id` and `customer_name`.
3. Calculate:
   - Total purchase amount per customer
   - Number of transactions per customer
   - Average transaction amount per customer
4. Sort by total purchase amount in descending order.
5. Display the results.

Answer

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import sum, count, avg, col

spark = SparkSession.builder.appName("SalesAnalysis").getOrCreate()

# Create sample data
data = [
    (1, 101, "Alice", "Laptop", 50000, "2025-01-10"),
    (2, 102, "Bob", "Mobile", 25000, "2025-01-11"),
    (3, 101, "Alice", "Keyboard", 5000, "2025-01-12"),
    (4, 103, "Clara", "Mouse", 1500, "2025-01-13"),
    (5, 101, "Alice", "Monitor", 15000, "2025-01-14"),
    (6, 102, "Bob", "Keyboard", 5000, "2025-01-15")
]

schema = ["transaction_id", "customer_id", "customer_name", "product", "amount", "purc
df = spark.createDataFrame(data, schema)

# Group by customer and aggregate
customer_summary = df.groupBy("customer_id", "customer_name").agg(
    sum("amount").alias("total_purchase"),
    count("transaction_id").alias("transaction_count"),
    avg("amount").alias("avg_transaction_amount")
)

# Sort by total purchase descending
customer_summary = customer_summary.orderBy(col("total_purchase").desc())

# Display results
customer_summary.show(truncate=False)
```

Expected Output

```
customer_id | customer_name | total_purchase | transaction_count | avg_transaction_an
101         | Alice         | 70000          | 3                 | 23333.33
102         | Bob           | 30000          | 2                 | 15000.0
103         | Clara         | 1500           | 1                 | 1500.0
```

Explanation

- **groupBy("customer_id", "customer_name"):** Partitions data by customer; subsequent aggregations apply within each group.
- **sum("amount").alias("total_purchase"):** Adds all amounts for each customer; .alias() renames the column for readability.
- **count("transaction_id").alias("transaction_count"):** Counts rows (transactions) per customer group.
- **avg("amount").alias("avg_transaction_amount"):** Computes mean transaction value per customer.
- **orderBy(col("total_purchase").desc()):** Sorts descending (highest totals first) for easy identification of top customers.

Key Concepts Tested

- DataFrame creation and loading
- GroupBy operation (transformation)
- Multiple aggregations in a single agg() call
- Sorting results
- Aliasing columns for clarity

QUESTION 4: Running Payroll Calculation (Window Functions + When/Otherwise)

Marks: 10

Question

You have an employee payroll dataset:

```
emp_id,emp_name,department,salary,performance_rating
1,Alice,IT,75000,4.5
2,Bob,HR,65000,3.8
3,Clara,IT,85000,4.2
4,David,Finance,90000,4.8
5,Eve,IT,78000,4.0
```

Task:

1. Load the data.
2. Add a bonus column based on performance rating:
   - Rating ≥ 4.5: 15% bonus
   - Rating 4.0-4.49: 10% bonus
   - Rating < 4.0: 5% bonus
3. Calculate total compensation (salary + bonus).
4. Rank employees within each department by salary (descending).
5. Display: emp_id, emp_name, department, salary, bonus, total_compensation, dept_rank.

Answer

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, when, round, rank
from pyspark.sql.window import Window

spark = SparkSession.builder.appName("PayrollCalculation").getOrCreate()

# Sample payroll data
data = [
    (1, "Alice", "IT", 75000, 4.5),
    (2, "Bob", "HR", 65000, 3.8),
    (3, "Clara", "IT", 85000, 4.2),
    (4, "David", "Finance", 90000, 4.8),
    (5, "Eve", "IT", 78000, 4.0)
]

schema = ["emp_id", "emp_name", "department", "salary", "performance_rating"]
df = spark.createDataFrame(data, schema)

# Add bonus column with nested when/otherwise logic
df = df.withColumn(
    "bonus",
    when(col("performance_rating") >= 4.5, col("salary") * 0.15)
    .when((col("performance_rating") >= 4.0) & (col("performance_rating") < 4.5), col("sala
    .otherwise(col("salary") * 0.05)
)

# Calculate total compensation
df = df.withColumn("total_compensation", col("salary") + col("bonus"))

# Define window for department ranking
dept_window = Window.partitionBy("department").orderBy(col("salary").desc())

# Add department rank
df = df.withColumn("dept_rank", rank().over(dept_window))

# Select and display required columns
result = df.select("emp_id", "emp_name", "department", "salary", "bonus", "total_comper
result.show(truncate=False)
```

Expected Output

```
emp_id | emp_name | department | salary | bonus   | total_compensation | dept_rank
1      | Alice    | IT         | 75000  | 7500.0  | 82500.0            | 2
3      | Clara    | IT         | 85000  | 8500.0  | 93500.0            | 1
5      | Eve      | IT         | 78000  | 7800.0  | 85800.0            | 3
2      | Bob      | HR         | 65000  | 3250.0  | 68250.0            | 1
4      | David    | Finance    | 90000  | 13500.0 | 103500.0           | 1
```

Explanation

- **when().when().otherwise():** Nested conditional logic creates tiered bonuses:
  - Rating ≥ 4.5 → 15% bonus
  - Rating [4.0, 4.5) → 10% bonus
  - Rating < 4.0 → 5% bonus
- **col("salary") * 0.15:** Multiplies salary by percentage; bonus is calculated relative to base pay.
- **withColumn("total_compensation", ...):** Sums salary and bonus for gross compensation.
- **Window.partitionBy("department").orderBy(col("salary").desc()):** Creates ranking scope:
  - `partitionBy()` groups employees by department
  - `orderBy(col("salary").desc())` orders within group by salary (highest first)
- **rank().over(dept_window):** Assigns rank; employees with same salary get same rank, next rank skips.

Key Concepts Tested

- When/Otherwise conditional logic (nested)
- Arithmetic operations and column creation
- Window functions (partitionBy, orderBy)
- Ranking within groups
- Complex transformation chaining

QUESTION 5: Food and Beverage Sales Summary (GroupBy with Multiple Aggregations)

Marks: 10

Question

You have a product sales dataset:

```
product_id,product_name,category,quantity_sold,unit_price,stock_on_hand
1,Coffee Beans,Beverage,150,500,80
2,Tea Leaves,Beverage,200,300,120
3,Bread,Food,300,100,50
4,Milk,Beverage,250,150,40
5,Cheese,Food,100,800,30
6,Orange Juice,Beverage,180,200,60
```

Task:

1. Load the product sales data.
2. Calculate:
    - Total quantity sold per category
    - Total revenue per category (quantity_sold × unit_price)
    - Total stock on hand per category
    - Average unit price per category
3. Sort by total revenue in descending order.
4. Display results.

Answer

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import sum, avg, col, round

spark = SparkSession.builder.appName("SalesAnalysis").getOrCreate()

# Sample product data
data = [
    (1, "Coffee Beans", "Beverage", 150, 500, 80),
    (2, "Tea Leaves", "Beverage", 200, 300, 120),
    (3, "Bread", "Food", 300, 100, 50),
    (4, "Milk", "Beverage", 250, 150, 40),
    (5, "Cheese", "Food", 100, 800, 30),
    (6, "Orange Juice", "Beverage", 180, 200, 60)
]

schema = ["product_id", "product_name", "category", "quantity_sold", "unit_price", "stock
df = spark.createDataFrame(data, schema)

# Add revenue column
df = df.withColumn("revenue", col("quantity_sold") * col("unit_price"))

# Group by category and aggregate
category_summary = df.groupBy("category").agg(
    sum("quantity_sold").alias("total_quantity"),
    sum("revenue").alias("total_revenue"),
    sum("stock_on_hand").alias("total_stock"),
    round(avg("unit_price"), 2).alias("avg_unit_price")
)

# Sort by revenue descending
category_summary = category_summary.orderBy(col("total_revenue").desc())

# Display
category_summary.show(truncate=False)
```

Expected Output

```
category   | total_quantity | total_revenue | total_stock | avg_unit_price
Beverage   | 780            | 237000        | 320         | 287.5
Food       | 400            | 110000        | 80          | 450.0
```

Explanation

- **withColumn("revenue", col("quantity_sold") * col("unit_price")):** Creates derived column by multiplying quantity and unit price for each product.
- **groupBy("category"):** Groups all products by their category (Beverage, Food).
- **sum("quantity_sold").alias("total_quantity"):** Sums quantities within each category.
- **sum("revenue").alias("total_revenue"):** Calculates total sales revenue per category.
- **round(avg("unit_price"), 2):** Computes average price per category, rounded to 2 decimals.
- **orderBy(col("total_revenue").desc()):** Sorts categories by revenue (highest first).

Key Concepts Tested

- Derived column creation (revenue calculation)
- Multiple aggregations with aliases
- Rounding functions
- GroupBy with multiple functions
- Sorting aggregated results

QUESTION 6: Discount Calculation and Pricing (Conditional Logic + Arithmetic)

Marks: 10

Question

You have a product inventory:

```
product_id,product_name,category,original_price,quantity_in_stock
1,Laptop,Electronics,100000,15
2,Mouse,Electronics,2000,50
3,USB Cable,Electronics,500,100
4,Monitor,Electronics,30000,8
5,Keyboard,Electronics,8000,25
```

Task:

1. Load the inventory data.
2. Apply discount logic:
   - Electronics with price > 50000: 15% discount
   - Electronics with price 10000-50000: 10% discount
   - Electronics with price < 10000: 5% discount
3. Calculate discounted price.
4. Calculate total inventory value (discounted_price × quantity).
5. Display: product_id, product_name, original_price, discount_percent, discounted_price, total_inventory_value.

Answer

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, when, round

spark = SparkSession.builder.appName("DiscountCalculation").getOrCreate()

# Sample inventory data
data = [
    (1, "Laptop", "Electronics", 100000, 15),
    (2, "Mouse", "Electronics", 2000, 50),
    (3, "USB Cable", "Electronics", 500, 100),
    (4, "Monitor", "Electronics", 30000, 8),
```

```
    (4, "Monitor", "Electronics", 30000, 8),
    (5, "Keyboard", "Electronics", 8000, 25)
]

schema = ["product_id", "product_name", "category", "original_price", "quantity_in_stock
df = spark.createDataFrame(data, schema)

# Apply tiered discount logic
df = df.withColumn(
    "discount_percent",
    when(col("original_price") > 50000, 15)
    .when((col("original_price") >= 10000) & (col("original_price") <= 50000), 10)
    .otherwise(5)
)

# Calculate discounted price
df = df.withColumn(
    "discounted_price",
    col("original_price") * (1 - col("discount_percent") / 100)
)

# Calculate total inventory value
df = df.withColumn(
    "total_inventory_value",
    col("discounted_price") * col("quantity_in_stock")
)

# Select and display required columns
result = df.select(
    "product_id",
    "product_name",
    "original_price",
    "discount_percent",
    round("discounted_price", 2).alias("discounted_price"),
    round("total_inventory_value", 2).alias("total_inventory_value")
)

result.show(truncate=False)
```

Expected Output

```
product_id | product_name | original_price | discount_percent | discounted_price | total_i
1          | Laptop       | 100000         | 15               | 85000.0          | 1275000.0
2          | Mouse        | 2000           | 5                | 1900.0           | 95000.0
3          | USB Cable    | 500            | 5                | 475.0            | 47500.0
4          | Monitor      | 30000          | 10               | 27000.0          | 216000.0
5          | Keyboard     | 8000           | 5                | 7600.0           | 190000.0
```

Explanation

- **when(...).when(...).otherwise(...):** Tiered discount logic based on price ranges.
- **col("original_price") * (1 - col("discount_percent") / 100):** Formula to calculate discounted price:
  - Discount of 15% → multiply by 0.85
  - Discount of 10% → multiply by 0.90
  - Discount of 5% → multiply by 0.95
- **col("discounted_price") * col("quantity_in_stock"):** Inventory value based on discounted price.
- **round(..., 2):** Rounds to 2 decimals for currency display.

Key Concepts Tested

- Tiered conditional logic (when/otherwise)
- Percentage calculations
- Multiple derived columns
- Column chaining
- Rounding for financial data

QUESTION 7: Load and Flatten Nested JSON Data (JSON Reading + Array/Explode Functions)

Marks: 10

Question

You have a nested JSON file `employees.json`:

```json
[
  {
    "emp_id": 1,
    "emp_name": "Alice",
    "department": "IT",
    "skills": ["Python", "Spark", "SQL"]
  },
  {
    "emp_id": 2,
    "emp_name": "Bob",
    "department": "HR",
    "skills": ["Recruitment", "Payroll"]
  },
  {
    "emp_id": 3,
    "emp_name": "Clara",
    "department": "IT",
    "skills": ["Java", "Docker", "Kubernetes", "Python"]
  }
]
```

Task:

1. Load the multiline JSON file.
2. Display the raw nested structure.
3. Flatten the data by exploding the `skills` array into individual rows.
4. Display flattened data.
5. Count unique skills across all employees.

Answer

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode, col, countDistinct

spark = SparkSession.builder.appName("JSONFlattening").getOrCreate()

# Load multiline JSON file
df = spark.read.option("multiline", True).json("path/to/employees.json")

print("Original Nested Structure:")
df.show(truncate=False)
df.printSchema()

# Explode the skills array to flatten data
flattened_df = df.select(
    col("emp_id"),
    col("emp_name"),
    col("department"),
    explode(col("skills")).alias("skill")
)

print("\nFlattened Data:")
flattened_df.show(truncate=False)

# Count distinct skills
skill_count = flattened_df.select(countDistinct("skill").alias("unique_skills")).collect()
print(f"\nTotal Unique Skills: {skill_count[0][0]}")

# Alternative: Show all distinct skills
flattened_df.select("skill").distinct().show()
```

Expected Output

Original:

```
emp_id | emp_name | department | skills
1      | Alice    | IT         | [Python, Spark, SQL]
2      | Bob      | HR         | [Recruitment, Payroll]
3      | Clara    | IT         | [Java, Docker, Kubernetes, Python]
```

Flattened:

```
emp_id | emp_name | department | skill
1      | Alice    | IT         | Python
1      | Alice    | IT         | Spark
1      | Alice    | IT         | SQL
2      | Bob      | HR         | Recruitment
2      | Bob      | HR         | Payroll
3      | Clara    | IT         | Java
3      | Clara    | IT         | Docker
3      | Clara    | IT         | Kubernetes
3      | Clara    | IT         | Python
```

Explanation

- **spark.read.option("multiline", True).json():** Reads pretty-printed JSON where arrays/objects span multiple lines.
- **explode(col("skills")):** Expands array column into multiple rows:
  - 1 row with 3 skills → 3 rows
  - Preserves other columns (emp_id, emp_name, department)
- **.alias("skill"):** Renames the exploded column from "skills" to "skill" (singular).
- **countDistinct("skill"):** Counts unique skill values across all rows.
- **distinct():** Returns unique skill values.

Key Concepts Tested

- JSON file reading with multiline option
- Schema inspection (printSchema)
- Array explosion (row expansion)
- Column aliasing
- Distinct value counting

QUESTION 8: Employee Earnings Above Department Average (Window Functions + Self-Referencing)

Marks: 10

Question

You have an employee salary dataset:

```
emp_id,emp_name,department,salary
1,Alice,IT,75000
2,Bob,IT,80000
3,Clara,IT,70000
4,David,Finance,90000
5,Eve,Finance,85000
6,Frank,HR,60000
```

Task:

1. Load the employee data.
2. Calculate the average salary per department using a window function.
3. Identify employees whose salary is above their department's average.
4. Display: emp_id, emp_name, department, salary, dept_avg_salary, is_above_avg (Yes/No).
5. Sort by department and salary descending.

Answer

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, avg, when, round
from pyspark.sql.window import Window

spark = SparkSession.builder.appName("EmployeeAnalysis").getOrCreate()

# Sample employee data
data = [
    (1, "Alice", "IT", 75000),
    (2, "Bob", "IT", 80000),
    (3, "Clara", "IT", 70000),
    (4, "David", "Finance", 90000),
    (5, "Eve", "Finance", 85000),
    (6, "Frank", "HR", 60000)
]

schema = ["emp_id", "emp_name", "department", "salary"]
df = spark.createDataFrame(data, schema)

# Define window specification for department average
dept_window = Window.partitionBy("department")

# Add department average salary column using window function
df = df.withColumn(
    "dept_avg_salary",
    round(avg(col("salary")).over(dept_window), 2)
)

# Determine if employee salary is above department average
df = df.withColumn(
    "is_above_avg",
    when(col("salary") > col("dept_avg_salary"), "Yes").otherwise("No")
)

# Sort by department and salary (descending)
result = df.orderBy(col("department"), col("salary").desc())

# Display required columns
result.select("emp_id", "emp_name", "department", "salary", "dept_avg_salary", "is_abov
```

Expected Output

```
emp_id | emp_name | department | salary | dept_avg_salary | is_above_avg
2      | Bob      | Finance    | 90000  | 87500.0         | Yes
5      | Eve      | Finance    | 85000  | 87500.0         | No
6      | Frank    | HR         | 60000  | 60000.0         | No
2      | Bob      | IT         | 80000  | 75000.0         | Yes
1      | Alice    | IT         | 75000  | 75000.0         | No
3      | Clara    | IT         | 70000  | 75000.0         | No
```

Explanation

- **Window.partitionBy("department"):** Creates window scope per department. All employees in IT see only IT average; Finance employees see Finance average.
- **avg(col("salary")).over(dept_window):** Computes average salary within each partition without grouping. All rows from a department retain their original records plus the department average.
- **when(col("salary") > col("dept_avg_salary"), "Yes").otherwise("No"):** Compares individual salary to department average:
  - If salary > avg → "Yes"
  - Else → "No"
- **orderBy(col("department"), col("salary").desc()):** Sorts by department first, then salary descending within each department.

Key Concepts Tested

- Window functions with partitionBy
- Aggregate functions within windows (avg)
- Conditional comparisons
- Multi-column sorting
- Self-referencing columns in window context

QUESTION 9: Remove Duplicates Using Window Functions

Marks: 10

Question

You have a customer dataset with duplicates:

```
customer_id,customer_name,email,city,signup_date
101,Alice,alice@email.com,Mumbai,2023-01-15
102,Bob,bob@email.com,Delhi,2023-02-20
101,Alice,alice@email.com,Mumbai,2023-01-15
103,Clara,clara@email.com,Bangalore,2023-03-10
102,Bob,bob@email.com,Delhi,2023-02-20
104,David,david@email.com,Chennai,2023-04-05
```

Task:

1. Load the customer data with duplicates.
2. Show the original data with duplicate rows.
3. Use a window function to assign a row number within each customer group (ordered by signup_date).
4. Keep only the first occurrence of each customer (row_number = 1).
5. Display the deduplicated data.

Answer

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import row_number
from pyspark.sql.window import Window

spark = SparkSession.builder.appName("DeduplicationWindow").getOrCreate()

# Sample data with duplicates
data = [
    (101, "Alice", "alice@email.com", "Mumbai", "2023-01-15"),
    (102, "Bob", "bob@email.com", "Delhi", "2023-02-20"),
    (101, "Alice", "alice@email.com", "Mumbai", "2023-01-15"),
    (103, "Clara", "clara@email.com", "Bangalore", "2023-03-10"),
    (102, "Bob", "bob@email.com", "Delhi", "2023-02-20"),
    (104, "David", "david@email.com", "Chennai", "2023-04-05")
]

schema = ["customer_id", "customer_name", "email", "city", "signup_date"]
df = spark.createDataFrame(data, schema)

print("Original Data with Duplicates:")
df.show(truncate=False)
print(f"Total Records: {df.count()}")

# Define window to assign row number per customer (ordered by signup_date)
window_spec = Window.partitionBy("customer_id").orderBy("signup_date")

# Add row number column
df_with_rn = df.withColumn("row_num", row_number().over(window_spec))

# Keep only row_number = 1 (first occurrence)
df_deduplicated = df_with_rn.filter("row_num = 1").drop("row_num")

print("\nDeduplicated Data:")
df_deduplicated.show(truncate=False)
print(f"Total Unique Records: {df_deduplicated.count()}")
```

Expected Output

Before:

```
customer_id | customer_name | email             | city       | signup_date
101         | Alice         | alice@email.com    | Mumbai     | 2023-01-15
102         | Bob           | bob@email.com      | Delhi      | 2023-02-20
101         | Alice         | alice@email.com    | Mumbai     | 2023-01-15
103         | Clara         | clara@email.com    | Bangalore  | 2023-03-10
102         | Bob           | bob@email.com      | Delhi      | 2023-02-20
104         | David         | david@email.com    | Chennai    | 2023-04-05
Total Records: 6
```

After:

```
customer_id | customer_name | email             | city       | signup_date
101         | Alice         | alice@email.com    | Mumbai     | 2023-01-15
102         | Bob           | bob@email.com      | Delhi      | 2023-02-20
103         | Clara         | clara@email.com    | Bangalore  | 2023-03-10
104         | David         | david@email.com    | Chennai    | 2023-04-05
Total Unique Records: 4
```

Explanation

- **Window.partitionBy("customer_id").orderBy("signup_date"):** Creates a window for each customer, ordered by signup date (earliest first).
- **row_number().over(window_spec):** Assigns sequential numbers (1, 2, 3...) within each customer group:
  - First occurrence of customer 101 → row_num = 1
  - Duplicate of customer 101 → row_num = 2
- **filter("row_num = 1"):** Keeps only the first occurrence of each customer.
- **drop("row_num"):** Removes the temporary row_number column.

Key Concepts Tested

- Window functions for deduplication

- Row numbering within partitions
- Filtering based on row numbers
- Preserving first occurrences
- Distinct counting

QUESTION 10: Word Count Program in PySpark (RDD Transformations + Actions)

Marks: 10

Question

You have a text file `sample.txt` containing:

```
Hello world hello spark
Spark is big data Spark
Hello hello Spark world
```

Task:

1. Load the text file using RDD.
2. Split each line into words.
3. Convert all words to lowercase.
4. Create (word, 1) pairs for each word.
5. Aggregate counts for each word.
6. Sort by count descending.
7. Display top 10 words and their counts.

Answer

```python
from pyspark.sql import SparkSession

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

# Load text file as RDD
text_rdd = sc.textFile("path/to/sample.txt")

# Perform word count transformations
word_counts = (text_rdd
    .flatMap(lambda line: line.split())          # Split each line into words
    .map(lambda word: word.lower())              # Convert to lowercase
    .map(lambda word: (word, 1))                 # Create (word, 1) pairs
    .reduceByKey(lambda a, b: a + b)             # Sum counts for each word
    .sortBy(lambda x: x[1], ascending=False)     # Sort by count descending
)

# Action: Collect and display top 10
top_words = word_counts.take(10)

print("Word Count Results (Top 10):")
for word, count in top_words:
    print(f"{word}: {count}")

# Alternative: Save to file
word_counts.saveAsTextFile("path/to/output/wordcount")

# Show total unique words
total_unique = word_counts.count()
print(f"\nTotal Unique Words: {total_unique}")
```

Expected Output

```
Word Count Results (Top 10):
hello: 4
spark: 4
world: 2
is: 1
big: 1
data: 1

Total Unique Words: 6
```

Explanation

- **sc.textFile():** Reads text file and returns RDD where each element is a line.
- **flatMap(lambda line: line.split()):** Transformation that:
    - Splits each line by whitespace into words
    - Flattens results (flatMap returns single RDD of words, not nested)
    - Example: ["Hello world", "Spark"] → ["Hello", "world", "Spark"]
- **map(lambda word: word.lower()):** Transformation converting each word to lowercase.
- **map(lambda word: (word, 1)):** Transformation creating (word, 1) tuples for counting.
- **reduceByKey(lambda a, b: a + b):** Transformation that:
    - Groups by key (word)
    - Aggregates values using lambda function (a + b adds counts)
    - Result: (word, total_count) pairs
- **sortBy(lambda x: x[1], ascending=False):** Transformation sorting by count (index 1) descending.
- **take(10):** Action that returns first 10 elements from RDD.
- **saveAsTextFile():** Action that writes results to HDFS/file system.
- **count():** Action that returns total number of elements.

Key Concepts Tested

- RDD creation from text files

- flatMap transformation
- map transformation (multiple chained)
- reduceByKey aggregation
- Sorting RDD data
- RDD actions (take, count, saveAsTextFile)
- Lambda functions in RDD operations

SUMMARY OF KEY CONCEPTS COVERED

Concept	Questions
DataFrame Creation & Loading	Q1, Q2, Q3, Q4, Q5
CSV/JSON File Reading	Q1, Q7
Column Operations (select, rename, add, drop)	Q1, Q6, Q8
Filtering & Conditions	Q2, Q8
When/Otherwise Logic	Q4, Q6
GroupBy & Aggregations	Q3, Q5
Window Functions	Q4, Q8, Q9
Arithmetic Operations	Q1, Q6
String Functions	Q1, Q7
Array/Explode Functions	Q7
Null Handling	Q2
Sorting/Ordering	Q3, Q4, Q8, Q10
Joins	- (Covered in extended questions)
RDD Operations	Q10

Concept	Questions
Actions (show, count, take, collect)	All questions

SCORING RUBRIC

Each question (10 marks) is evaluated as follows:

- **Code Correctness (5 marks):** Code runs without errors; transformations are applied correctly.
- **Output Accuracy (3 marks):** Output matches expected results; logic is sound.
- **Explanation (2 marks):** Clear explanation of logic, transformations, and concepts.

Pass: ≥ 60 marks (6/10 questions correct)
Merit: ≥ 80 marks (8/10 questions correct)
Distinction: ≥ 90 marks (9/10 questions correct)

INSTRUCTIONS FOR STUDENTS

1. **Environment Setup:**
    - Ensure PySpark is installed and SparkSession is initialized.
    - All sample data is provided in questions; use it as-is.
2. **Execution:**
    - Write and test code in a Jupyter notebook or PySpark shell.
    - Verify output matches expected results.
    - Document your understanding of transformations and actions.
3. **Answer Format:**
    - Submit Python code for each question.
    - Include brief explanations of each transformation step.

    - Show actual output/results.

4. **Time Management:**
    - ~9 minutes per question on average.
    - Start with easier questions (Q1, Q2, Q3) if time-pressed.
    - Advanced questions (Q7, Q8, Q9, Q10) can take more time.

Good Luck! Practice, Understand, Excel!