Ex. No. 3	**Basic RDD Transformations and Actions in Spark**
Youtube Link	**https://youtu.be/1Rfg2ssIwQk**
Date of Exercise	13.10.25

AIM

To understand and implement basic **RDD Transformations** (map, filter, flatMap, reduceByKey) and a variety of **RDD Actions** (retrieval and aggregation) in Apache Spark using PySpark.

Procedure:

1. **Start Spark Environment**
   - Install and configure Apache Spark and Python (PySpark).
   - Set environment variables SPARK_HOME and update PATH.
   - Open **PySpark shell** or a **Jupyter Notebook** with PySpark support.
2. **Initialize SparkContext**
   - Import SparkContext from pyspark.
   - Create a SparkContext object with a descriptive application name.
3. **Create an RDD**
   - Use parallelize() to create an RDD from a Python list.
   - Alternatively, load from an external text file using textFile().
4. **Apply Transformations**
   - **flatMap()** → Split sentences into words.
   - **map()** → Create key-value pairs (word, 1).
   - **reduceByKey()** → Count word occurrences.
   - **filter()** → Select words with frequency above a threshold.
5. **Perform Actions**
   - **Retrieval Actions:** collect(), take(), takeSample(), takeOrdered(), top(), first(), isEmpty(), foreach().

- **Aggregation Actions:** count(), countByValue(), reduce(), fold(), aggregate().

6. **Display and Compare Results**
   - Print outputs for each action.
   - Observe differences between **data retrieval** and **aggregation** actions.
7. **Stop SparkContext**
   - Call sc.stop() to release resources.

Program:

```
from pyspark import SparkContext

# Step 1: Initialize SparkContext

sc = SparkContext("local", "Basic RDD Transformations and Actions")

# Step 2: Create an RDD

data = ["Spark is fast", "Spark is powerful", "Spark is easy to use"]

rdd = sc.parallelize(data)

# Step 3: Transformations

words = rdd.flatMap(lambda line: line.split(" "))

word_pairs = words.map(lambda word: (word, 1))

word_count = word_pairs.reduceByKey(lambda a, b: a + b)

filtered_words = word_count.filter(lambda x: x[1] > 1)

# Step 4: Retrieval Actions

print("\n--- Retrieval Actions ---")

print("Collect:", words.collect())

print("Take(5):", words.take(5))

print("TakeSample (no replacement, 4):", words.takeSample(False, 4))
print("TakeOrdered (alphabetical, 5):", words.takeOrdered(5))

print("Top(5):", words.top(5))

print("First element:", words.first())

print("IsEmpty?:", words.isEmpty())

# foreach action (prints each word; order not guaranteed)

print("\nForeach output:")

words.foreach(lambda w: print("Word:", w))

# Step 5: Aggregation Actions

print("\n--- Aggregation Actions ---")

print("Count:", words.count())

print("CountByValue:", dict(words.countByValue()))

print("Reduce (total word count):", words.map(lambda x: 1).reduce(lambda a, b: a + b))

print("Fold (total word count):", words.map(lambda x: 1).fold(0, lambda a, b: a + b))

# Aggregate: Calculate average word length

agg_result = words.aggregate(

    (0, 0),  # (total length, word count)

    lambda acc, value: (acc[0] + len(value), acc[1] + 1),  # seqOp

    lambda acc1, acc2: (acc1[0] + acc2[0], acc1[1] + acc2[1])  # combOp

)

avg_word_length = agg_result[0] / agg_result[1]

print("Aggregate (avg word length):", avg_word_length)


# Step 6: Stop SparkContext

sc.stop()
```

Output:

```
--- Retrieval Actions ---

Collect: ['Spark', 'is', 'fast', 'Spark', 'is', 'powerful', 'Spark', 'is', 'easy', 'to', 'use']

Take(5): ['Spark', 'is', 'fast', 'Spark', 'is']

TakeSample (no replacement, 4): ['is', 'to', 'easy', 'Spark']

TakeOrdered (alphabetical, 5): ['Spark', 'Spark', 'Spark', 'easy', 'fast']

Top(5): ['use', 'to', 'powerful', 'is', 'is']

First element: Spark

IsEmpty?: False


Foreach output:

Word: Spark

Word: is

Word: fast

Word: Spark

Word: is

Word: powerful

Word: Spark

Word: is

Word: easy

Word: to

Word: use


--- Aggregation Actions ---

Count: 11

CountByValue: {'Spark': 3, 'is': 3, 'fast': 1, 'powerful': 1, 'easy': 1, 'to': 1, 'use': 1}
```

```
Reduce (total word count): 11

Fold (total word count): 11

Aggregate (avg word length): 3.8181818181818183
```

Result :

The Program was executed Succesfully.