RDD

1. **Immutability**: Once an RDD is created, it cannot be modified. Any transformation applied to an RDD results in a new RDD.
2. **Distributed**: RDDs are distributed across the nodes of a cluster, enabling parallel processing.
3. **Fault Tolerance**: RDDs are designed to recover from failures using lineage information (i.e., the sequence of transformations that created them).
4. **Lazy Evaluation**: Transformations on RDDs are not executed immediately. Instead, they are recorded as a lineage of transformations which Spark's execution engine optimizes and executes only when an action is called.

Creating RDDs

RDDs can be created from existing data in a storage system (e.g., HDFS, S3, or local file system) or by parallelizing an existing collection in your driver program.

From Parallelized Collections

```
from pyspark import SparkContext

# Initialize SparkContext
sc = SparkContext("local", "RDD Example")

# Create RDD from a Python list
data = [1, 2, 3, 4, 5]
rdd = sc.parallelize(data)

# Show RDD content
print(rdd.collect())
```

From External Resource

```
# Create RDD from an external text file
rdd = sc.textFile("path/to/textfile.txt")

# Show first few elements
print(rdd.take(5))
```

RDD Operations

RDD operations are divided into two types: Transformations and Actions.

Transformations

Transformations are operations on RDDs that return a new RDD. They are lazy and are not executed immediately. Examples include `map()`, `filter()`, `flatMap()`, `reduceByKey()`, and `join()`.

- **map()**: Applies a function to each element of the RDD and returns a new RDD.

```
# Double each element in the RDD
rdd2 = rdd.map(lambda x: x * 2)
print(rdd2.collect())
```

- **filter()**: Returns a new RDD containing only the elements that satisfy a predicate.

```
# Filter elements greater than 3
rdd3 = rdd.filter(lambda x: x > 3)
print(rdd3.collect())
```

- **flatMap()**: Similar to `map`, but each input item can be mapped to zero or more output items (flattening the results).

```
# Split lines into words
rdd_flat = rdd.flatMap(lambda line: line.split(" "))
print(rdd_flat.collect())
```

Actions

Actions trigger the execution of transformations and return a result to the driver program or write data to an external storage. Examples include `collect()`, `take()`, `count()`, `saveAsTextFile()`, and `reduce()`.

- **collect()**: Returns all the elements of the RDD to the driver.

```
# Collect the RDD elements
result = rdd.collect()
print(result)
```

- **count()**: Returns the number of elements in the RDD.

```
# Count the elements in the RDD
count = rdd.count()
print(count)
```

- **reduce()**: Aggregates the elements of the RDD using a specified function.

```
# Sum all elements in the RDD
sum = rdd.reduce(lambda x, y: x + y)
print(sum)
```

RDD Persistence

RDDs can be cached or persisted in memory for efficient reuse. This is useful when an RDD is used multiple times in a program.

```
# Persist the RDD in memory
rdd.persist()

# Or using a specific storage level
from pyspark import StorageLevel
rdd.persist(StorageLevel.MEMORY_AND_DISK)
```

Example Workflow

Let’s consider an example workflow that reads a text file, processes the data, and outputs the results.

```
# Initialize SparkContext
sc = SparkContext("local", "RDD Workflow Example")

# Read text file into RDD
lines = sc.textFile("path/to/textfile.txt")

# Transformations: Split lines into words, filter words, and map each word to a (word, 1) pair
words = lines.flatMap(lambda line: line.split(" "))
filtered_words = words.filter(lambda word: len(word) > 3)
word_pairs = filtered_words.map(lambda word: (word, 1))

# ReduceByKey: Sum the counts for each word
word_counts = word_pairs.reduceByKey(lambda a, b: a + b)

# Action: Collect the result and print
result = word_counts.collect()
for word, count in result:
    print(f"{word}: {count}")

# Stop SparkContext
sc.stop()
```