

ICP 8

 ICP 8 

File Edit View Insert Runtime Tools Help

+ Code + Text



Reconnect Gemini

```
from pyspark import SparkContext, SparkConf

# Check if a SparkContext already exists
if 'sc' not in globals():
    # Initialize SparkContext only if one doesn't exist
    conf = SparkConf().setAppName("BigDataICP").setMaster("local")
    sc = SparkContext(conf=conf)
else:
    # Reuse the existing SparkContext
    print("Using existing SparkContext")

[ ] rdd1 = sc.parallelize(range(1, 16))
    print("Task 1:", rdd1.collect())
```

Task 1: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

 ICP 8 

File Edit View Insert Runtime Tools Help [All changes saved](#)

+ Code + Text

Reconnect Gemini

```
[ ] print("Elements in RDD:", rdd1.collect())
    print("Number of Partitions:", rdd.getNumPartitions())

[ ] print("First Element in RDD:", rdd1.first())

[ ] even_rdd = rdd1.filter(lambda x: x % 2 == 0)
    print("Even Elements:", even_rdd.collect())



[ ] squared_rdd = rdd1.map(lambda x: x**2)
    print("Squared Elements:", squared_rdd.collect())
```

Elements in RDD: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
Number of Partitions: 2

First Element in RDD: 1

Even Elements: [2, 4, 6, 8, 10, 12, 14]

Squared Elements: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225]

 ICP 8 

File Edit View Insert Runtime Tools Help [All changes saved](#)

+ Code + Text

Reconnect Gemini

```
[ ] sum_elements = rdd1.reduce(lambda x, y: x + y)
    print("Sum of Elements:", sum_elements)

[ ] rdd1.saveAsTextFile("rdd_output")

[ ] rdd2 = sc.parallelize(range(16, 21))
    rdd_union = rdd1.union(rdd2)
    print("Task 8 - Union of RDDs:", rdd_union.collect())

[ ] rdd_cartesian = rdd1.cartesian(rdd2)
    print("Task 9 - Cartesian product:", rdd_cartesian.collect())
```

Sum of Elements: 120

Task 8 - Union of RDDs: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

Task 9 - Cartesian product: [(1, 16), (1, 17), (2, 16), (3, 16), (2, 17), (3, 17), (4, 16), (5, 16), (6, 16), (7, 16), (4, 17), (5, 17), (6, 17), (7,

```
[ ] dict_data = {'a': 1, 'b': 2, 'c': 3}
rdd_dict = sc.parallelize(dict_data.items())
print("Task 10 - RDD from dictionary:", rdd_dict.collect())

Task 10 - RDD from dictionary: [('a', 1), ('b', 2), ('c', 3)]

[ ] rdd_with_duplicates = sc.parallelize([1, 2, 2, 3, 3, 3])
unique_counts = rdd_with_duplicates.countByValue()
print("Task 11 - Unique value counts:", unique_counts)

Task 11 - Unique value counts: defaultdict(<class 'int'>, {1: 1, 2: 2, 3: 3})

[ ] rdd_text_files = sc.textFile("/content/rdd_output/content/squared_rdd_output")
print("Task 12 - Combined text files RDD:", rdd_text_files.collect())

Task 12 - Combined text files RDD: ['8', '9', '10', '11', '12', '13', '14', '15', '1', '2', '3', '4', '5', '6', '7', '64', '81', '100', '121', '144',

[ ] print("First 5 lines of RDD:", rdd_text_files.take(5))

First 5 lines of RDD: ['8', '9', '10', '11', '12']
```

```
[ ] print("First 5 lines of RDD:", rdd_text_files.take(5))

First 5 lines of RDD: ['8', '9', '10', '11', '12']

[ ] from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)

/usr/local/lib/python3.10/dist-packages/pyspark/sql/context.py:113: FutureWarning: Deprecated in 3.0.0. Use SparkSession.builder.getOrCreate() instead
warnings.warn(

data = [("veda", 1), ("agastya", 2), ("darshana", 3)]
df = sqlContext.createDataFrame(data, ["Name", "ID"])
df.show()

+-----+
| Name | ID |
+-----+
| veda | 1 |
| agastya | 2 |
| darshana | 3 |
+-----+
```

RDDs are the most basic option, with no schema and minimal optimizations, making them slower and requiring more memory. They're fault-tolerant and type-safe, catching errors early, but lack SQL support and are generally harder to use. **DataFrames**, on the other hand, are structured like tables with schemas, which allows them to be optimized for faster performance. They support SQL queries and are high-level, making them easier to work with, though they detect errors at runtime. **Datasets** combine the best of both, offering the structure and SQL support of DataFrames, with added type safety and the highest performance due to advanced optimizations. They're also the most memory-efficient, making them ideal for complex application

GitHub Link: <https://github.com/w8162583/bda.git>