Title: Implement Matrix Multiplication with Hadoop/PySpark MapReduce

Problem Statement:

Implement matrix multiplication using the MapReduce paradigm within the PySpark framework. Given two matrices A and B, compute their product C, where C = A * B.

Theory:

1. Introduction:

Matrix multiplication is a fundamental operation in linear algebra, with applications in various fields like computer graphics, machine learning, and scientific computing. In a distributed computing environment like Hadoop or PySpark, we aim to perform this operation efficiently by leveraging parallelism. The MapReduce paradigm allows us to break down the multiplication into smaller, independent tasks that can be executed concurrently.

2. Problem Statement:

Given matrices A and B, we need to compute their product C. The dimensions of the matrices must be compatible for multiplication (i.e., the number of columns in A must equal the number of rows in B). The output should be the resulting matrix C, with each element C(i, j) representing the sum of the products of the corresponding elements from row i of A and column j of B.

3. Matrix Representation in RDD:

In PySpark, we represent matrices as Resilient Distributed Datasets (RDDs). Each element in the RDD represents a non-zero entry in the matrix, stored as a tuple (row, column, value). This representation is efficient for sparse matrices, as it only stores the non-zero elements.

For example, matrix A:

11 8 7

 8 0 5

would be represented as:

(0, 0, 11), (0, 1, 8), (0, 2, 7), (1, 0, 8), (1, 1, 0), (1, 2, 5)

4. MapReduce Workflow:

The MapReduce workflow for matrix multiplication involves the following steps:

- Map Phase:

    - Transform each matrix entry into a key-value pair.
    - For matrix A, the key is the column index, and the value is (row index, value).
    - For matrix B, the key is the row index, and the value is (column index, value).
- Join Phase:
    - Join the mapped RDDs based on the common key (column index of A and row index of B).
- Partial Product Computation:
    - Multiply the corresponding values from the joined pairs.
    - The key becomes (row index of A, column index of B), and the value is the partial product.
- Reduce Phase:
    - Sum the partial products for each (row index of A, column index of B) key.
    - The result is the element C(i, j) of the product matrix.

5. Explanation of Code (Matrix Multiplication Logic):

- RDD Creation: The input matrices are converted into RDDs of tuples (row, column, value).
- Map Phase: The `map` transformation is used to create key-value pairs. Matrix A is keyed by its column index, and matrix B is keyed by its row index.
- Join Phase: The `join` transformation combines the RDDs based on the common key (column of A and row of B).
- Partial Product Computation: Another `map` transformation calculates the partial products by multiplying the corresponding values.
- Reduce Phase: The `reduceByKey` transformation sums the partial products for each (row, column) position, producing the final result.
- Output: The result is collected and printed in a sorted manner.

6. Performance Considerations:

- Data Partitioning: Proper data partitioning is crucial for efficient parallel processing.
- Sparse Matrices: This implementation is optimized for sparse matrices.
- Shuffle Operations: Shuffle operations (like `join` and `reduceByKey`) can be expensive. Optimizing these operations is essential for performance.
- Memory Management: Large matrices can consume significant memory. Efficient memory management is vital.

7. Conclusion:

This assignment demonstrates how to implement matrix multiplication using the MapReduce paradigm in PySpark. By breaking down the problem into map, join, and reduce phases, we can efficiently perform matrix multiplication on large datasets. The use of RDDs and transformations allows for parallel processing, leading to significant performance gains.

INPUT

```
from pyspark.sql import SparkSession


# Initialize Spark session

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

sc = spark.sparkContext

matrix_A = [

    (0, 0, 11), (0, 1, 8), (0, 2, 7),

    (1, 0, 8), (1, 1, 0), (1, 2, 5)

]

matrix_B = [

    (0, 0, 0), (0, 1, 15),

    (1, 0, 9), (1, 1, 10),

    (2, 0, 17), (2, 1, 12)

]

# Convert matrices into RDDs

rdd_A = sc.parallelize(matrix_A)  # (row, col, value)

rdd_B = sc.parallelize(matrix_B)  # (row, col, value)

# Map phase: Convert matrix entries into (key, value) pairs

mapped_A = rdd_A.map(lambda x: (x[1], (x[0], x[2])))  # Keyed by column of
A

mapped_B = rdd_B.map(lambda x: (x[0], (x[1], x[2])))  # Keyed by row of B

# Join on common key (column index of A and row index of B)

joined = mapped_A.join(mapped_B)  # (col, ((row_A, val_A), (col_B,
val_B)))
```

```
# Compute partial products

partial_products = joined.map(lambda x: ((x[1][0][0], x[1][1][0]),
x[1][0][1] * x[1][1][1]))

# Reduce phase: Sum partial products for each (row, col) position

result = partial_products.reduceByKey(lambda x, y: x + y)

# Collect and print results

output = result.collect()

for ((row, col), value) in sorted(output):

    print(f"({row}, {col}) -> {value}")

# Stop Spark session

spark.stop()
```

OUTPUT

```
(0, 0) -> 191
(0, 1) -> 329
(1, 0) -> 85
(1, 1) -> 180
```

```python
from pyspark.sql import SparkSession

# Initialize Spark session
spark = SparkSession.builder.appName("MatrixMultiplication").getOrCreate()
sc = spark.sparkContext

# Example matrices
matrix_A = [
    (0, 0, 4), (0, 1, 6), (0, 2, 8),
    (1, 0, 5), (1, 1, 5), (1, 2, 4)
]

matrix_B = [
    (0, 0, 7), (0, 1, 8),
    (1, 0, 9), (1, 1, 10),
    (2, 0, 11), (2, 1, 12)
]

# Convert matrices into RDDs
rdd_A = sc.parallelize(matrix_A)  # (row, col, value)
rdd_B = sc.parallelize(matrix_B)  # (row, col, value)

# Map phase: Convert matrix entries into (key, value) pairs
mapped_A = rdd_A.map(lambda x: (x[1], (x[0], x[2])))
mapped_B = rdd_B.map(lambda x: (x[0], (x[1], x[2])))


joined = mapped_A.join(mapped_B)

# Compute partial products
partial_products = joined.map(lambda x: ((x[1][0][0], x[1][1][0]), x[1][0][1] * x[1][1][1]))

# Reduce phase: Sum partial products for each (row, col) position
result = partial_products.reduceByKey(lambda x, y: x + y)

# Collect and print results
output = result.collect()
for ((row, col), value) in sorted(output):
    print(f"({row}, {col}) -> {value}")

# Stop Spark session
spark.stop()
```

```
(0, 0) -> 170
(0, 1) -> 188
(1, 0) -> 124
(1, 1) -> 138
```