Spark vs. MapReduce

1. Overview

Both **Apache Spark** and **Hadoop MapReduce** are distributed data processing frameworks. They are designed to process large datasets across clusters of computers but differ significantly in terms of architecture, speed, flexibility, and ease of use.

- **MapReduce** is part of the Hadoop ecosystem and follows a disk-based, batch-processing model.
- **Spark** provides in-memory data processing, which makes it faster and more flexible than MapReduce.

2. Key Differences

Feature	Spark	MapReduce
Processing Model	In-memory (fast)	Disk-based (slower)
Ease of Use	Simple API (RDDs, DataFrames, SQL)	Complex API (low-level map and reduce)
Fault Tolerance	RDD lineage, DAG	Rewrites intermediate data to disk
Latency	Low latency	High latency
Language Support	Scala, Java, Python, R	Java, Python
Iterative Processing	Excellent (machine learning, graph processing)	Poor (needs multiple jobs)
Framework Integration	Seamless integration (MLlib, GraphX, etc.)	Limited

3. Architecture

- **MapReduce**: Breaks the job into two phases: `Map` and `Reduce`. Each phase writes intermediate data to disk, leading to high latency.
- **Spark**: Uses **Resilient Distributed Datasets (RDDs)**, and processes data in memory. The intermediate data is retained in memory, reducing disk I/O.

4. Performance Comparison

- **MapReduce**: As every iteration writes data to disk, it's slower, especially for iterative tasks.
- **Spark**: Leverages in-memory computations, making it significantly faster for iterative processes such as machine learning algorithms.

5. Code Example

5.1. Word Count in Hadoop MapReduce

Here is a simple example of word count using MapReduce:

Mapper Code (Java):

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

public class WordCountMapper extends Mapper<Object, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
        String[] words = value.toString().split("\\s+");
        for (String w : words) {
            word.set(w);
            context.write(word, one);
        }
    }
}
```

Reducer Code (Java):

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
```

```
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
    }
}
```

Driver Code (Java):

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCountDriver {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(WordCountDriver.class);
        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

5.2. Word Count in Apache Spark (Python – PySpark)

Now, let's look at the same word count example in Spark using PySpark:

PySpark Code:

```
from pyspark import SparkConf, SparkContext

# Initialize Spark
conf = SparkConf().setAppName("WordCount")
sc = SparkContext(conf=conf)

# Read input file
input_file = sc.textFile("hdfs://input.txt")

# Word count logic
words = input_file.flatMap(lambda line: line.split(" "))
word_counts = words.map(lambda word: (word, 1)).reduceByKey(lambda a, b: a + b)

# Save output to HDFS
word_counts.saveAsTextFile("hdfs://output")

# Stop Spark
sc.stop()
```

6. Performance Analysis

- **MapReduce**: Writes intermediate data to disk between the map and reduce stages. It processes data sequentially and is not optimized for iterative tasks.
- **Spark**: Uses in-memory computations, making it far more efficient for tasks with iterative operations. It processes data up to 100x faster than MapReduce in certain cases, especially for iterative machine learning algorithms.

7. Use Cases

Use Case	Spark	MapReduce
Batch Processing	Suitable but overkill for basic batch jobs	Very effective for batch processing
Real-Time Processing	Excellent (with Spark Streaming)	Not designed for real-time
Iterative Processing (ML/AI)	Perfect for iterative tasks (MLlib, GraphX)	Inefficient due to disk I/O between iterations
ETL (Extract, Transform, Load)	Fast for ETL with DataFrames	Suitable for basic ETL tasks

8. Fault Tolerance

Both frameworks handle fault tolerance but in different ways:

- **MapReduce**: Saves intermediate data to disk. If a node fails, it re-executes the job based on the saved data.
- **Spark**: Uses lineage to recompute only the lost partitions of RDDs, which is faster and more efficient.

9. Conclusion

Spark has gained popularity over MapReduce due to its speed, simplicity, and flexibility, especially for real-time and iterative processing. However, Hadoop MapReduce is still a reliable solution for batch jobs with high fault tolerance.

When to Use Spark	When to Use MapReduce
– Real-time data processing	– Large-scale batch jobs
– Machine learning tasks	– Simple ETL operations
– Graph processing	– When disk-based processing is fine