Data Engineering

How MapReduce Works

Shivnath Babu
Lifecycle of a MapReduce Job

Map function

```java
public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);
    while (tokenizer.hasMoreTokens()) {
        word.set(tokenizer.nextToken());
        output.collect(word, one);
    }
}
```

Reduce function

```java
public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
    int sum = 0;
    while (values.hasNext()) { sum += values.next().get(); }
    output.collect(key, new IntWritable(sum));
}
```

Run this program as a MapReduce job

```java
public static class WordCount {
    public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
        public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
            String line = value.toString();
            StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken());
                output.collect(word, one);
            }
        }
    }
    public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {
        public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
            int sum = 0;
            while (values.hasNext()) { sum += values.next().get(); }
            output.collect(key, new IntWritable(sum));
        }
    }
    public static void main(String[] args) throws Exception {
        JobConf conf = new JobConf(WordCount.class);
        conf.setJobName("wordcount");
        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);
        conf.setMapperClass(Map.class);
        conf.setCombinerClass(Reduce.class);
        conf.setReducerClass(Reduce.class);
        conf.setInputFormat(TextInputFormat.class);
        conf.setOutputFormat(TextOutputFormat.class);
        conf.setInputPath(FileInputFormat.makePath("input"));
        conf.setOutputPath(FileOutputFormat.makePath("output"));
        JobClient.runJob(conf);
    }
}
```
Lifecycle of a MapReduce Job

```java
public class WordCount {
    public static class Map extends MapReduceBase implements 
    Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> 
        output, Reporter reporter) throws IOException {
            String line = value.toString();
            StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken());
                output.collect(word, one);
            }
        }
    }

    public static class Reduce extends MapReduceBase implements 
    Reducer<Text, IntWritable, Text, IntWritable> {
        public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> 
        output, Reporter reporter) throws IOException {
            int sum = 0;
            while (values.hasNext()) { sum += values.next().get(); }
            output.collect(key, new IntWritable(sum));
        }
    }

    public static void main(String[] args) throws Exception {
        JobConf conf = new JobConf(WordCount.class);
        conf.setJobName("wordcount");
        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);
        conf.setMapperClass(Map.class);
        conf.setCombinerClass(Reduce.class);
        conf.setReducerClass(Reduce.class);
        conf.setInputFormat(TextInputFormat.class);
        conf.setOutputFormat(TextOutputFormat.class);
        TextInputFormat.setInputPaths(conf, new Path(args[0]));
        TextOutputFormat.setOutputPath(conf, new Path(args[1]));
        JobClient.runJob(conf);
    }
}
```

Map function
Reduce function
Run this program as a MapReduce job
Lifecycle of a MapReduce Job

Input Splits  Map Wave 1  Map Wave 2  Reduce Wave 1  Reduce Wave 2
Job Submission

1. Copy Input Files
2. Submit Job
3. Get Input Files’ info
4. Create Splits
5. Upload job information
6. Submit Job

DFS
Input Files
Job.xml, Job.jar, splits
Client
User
JobTracker
Initialization
Scheduling
Execution

JobTracker → Task Tracker

Assign task for Execution

Task Tracker

Upto MAX_MAP_SLOTS
Map Task JVMs Concurrency

Upto MAX_REDUCE_SLOTS
Reduce Task JVMs Concurrency

Read into Local Disk

Job.xml, Job.jar

DFS
Reduce Tasks
Hadoop Distributed File-System (HDFS)
How are the number of splits, number of map and reduce tasks, memory allocation to tasks, etc., determined?
What if the number of reduces increased to 9?
Spark Programming Model

Driver Program

sc=new **SparkContext**

rDD=sc.textfile("hdfs:/...")
rDD.filter(...)
rDD.Cache
rDD.Count
rDD.map

User (Developer)

Writes

Cluster Manager

SparkContext

Worker Node

Executor

Cache

Task

Task

Worker Node

Executor

Cache

Task

Task

Datanode

HDFS

Taken from http://www.slideshare.net/fabiofumarola1/11-from-hadoop-to-spark-12
Spark Programming Model

**Driver Program**

```
sc = new SparkContext
rDD = sc.textfile("hdfs:/...")
rDD.filter(...)
rDD.Cache
rDD.Count
rDD.map
```

**RDD (Resilient Distributed Dataset)**

- Immutable Data structure
- In-memory (explicitly)
- Fault Tolerant
- Parallel Data Structure
- Controlled partitioning to optimize data placement
- Can be manipulated using a rich set of operators

**User (Developer)**

Write operations:

- First writes to an RDD
- Then uses operations like `filter`, `Cache`, `Count`, etc.
**RDD**

- Programming Interface: Programmer can perform 3 types of operations

### Transformations
- Create a new dataset from an existing one.
- Lazy in nature. They are executed only when some action is performed.
- Example:  
  - Map(func)  
  - Filter(func)  
  - Distinct()

### Actions
- Returns to the driver program a value or exports data to a storage system after performing a computation.
- Example:  
  - Count()  
  - Reduce(func)  
  - Collect  
  - Take()

### Persistence
- For caching datasets in-memory for future operations.
- Option to store on disk or RAM or mixed (Storage Level).
- Example:  
  - Persist()  
  - Cache()
How Spark works

• RDD: Parallel collection with partitions
• User application can create RDDs, transform them, and run actions.
• This results in a DAG (Directed Acyclic Graph) of operators.
• DAG is compiled into stages
• Each stage is executed as a collection of Tasks (one Task for each Partition).
Summary of Components

• Task: The fundamental unit of execution in Spark

• Stage: Collection of Tasks that run in parallel

• DAG: Logical Graph of RDD operations

• RDD: Parallel dataset of objects with partitions
Example

```
sc.textFile("/wiki/pagecounts")
```

RDD[String]
Example

```scala
sc.textFile("/wiki/pagecounts")
  .map(line => line.split("\t"))
```

```
RDD[String]
RDD[List[String]]
```
Example

```scala
sc.textFile("/wiki/pagecounts")
  .map(line => line.split("\t"))
  .map(R => (R[0], int(R[1])))
```

RDD[String]
RDD[List[String]]
RDD[(String, Int)]
Example

```
sc.textFile("/wiki/pagecounts")
  .map(line => line.split("\t"))
  .map(R => (R[0], int(R[1])))
  .reduceByKey(_+_)
```

RDD[String]
RDD[List[String]]
RDD[(String, Int)]
RDD[(String, Int)]
Example

```scala
sc.textFile("/wiki/pagecounts")
  .map(line => line.split("\t"))
  .map(R => (R[0], int(R[1])))
  .reduceByKey(_+_, 3)
  .collect()
```

**RDD[String]**
**RDD[List[String]]**
**RDD[(String, Int)]**
**RDD[(String, Int)]**
**Array[(String, Int)]**

**collect**

**reduceByKey**
Execution Plan

Stages are sequences of RDDs, that don’t have a Shuffle in between
Execution Plan

Stage 1
- textFile
- map
- map

Stage 2
- reduceByKey
- collect

1. Read HDFS split
2. Apply both the maps
3. Start partial reduce
4. Write shuffle data

Stage 1
- read shuffle data
- final reduce
- send result to driver program

Stage 2
Stage Execution

- Create a task for each Partition in the input RDD
- Serialize the Task
- Schedule and ship Tasks to Slaves

And all this happens internally (you don’t have to do anything)
Spark Executor (Slave)