Week 04

Big Data Analytics

Topic:

MapReduce Basics codes and Components

Instructor: Hassan Jahangir

Introduction to Map-Reduce

- Map-Reduce: A programming model designed for parallel processing of data.
- Divided into two phases: Map Phase and Reduce Phase.
- Data is processed in parallel across various machines (nodes).
- Key components: Mapper, Reducer, Input, Input Splits, Record Reader, Map, Intermediate Output Disk.

Mapper in Map-Reduce

- Mapper: User-defined program operating on input-splits.
- Extends Mapper class, specifying input and output types.
- Each input-split processed by a separate Mapper.
- Produces intermediate output stored on local disk.
- Acts as input for Reducer for further processing.

Components of Mapper

- **Input:** Data or records for analysis.
- **Input Splits:** Logical division of input data.
- **Record Reader:** Converts input data to key-value pairs.
- **Map:** Processes key-value pairs, generates intermediate output.
- **Intermediate Output Disk:** Stores intermediate results on local disk.

Input:

- Input is records or the datasets that are used for analysis purposes. This Input data is set out with the help of **InputFormat**. It helps in identifying the location of the Input data which is stored in HDFS(Hadoop Distributed File System).

Input-Splits:

- These are responsible for converting the physical input data to some logical form so that Hadoop Mapper can easily handle it. Input-Splits are generated with the help of **InputFormat**. A large data set is divided into many input-splits which depend on the size of the input dataset. There will be a separate Mapper assigned for each input-splits. Input-Splits are only referencing to the input data, these are not the actual data. **DataBlocks** are not the only factor that decides the number of input-splits in a Map-Reduce. we can manually configure the size of input-splits in ***mapred.max.split.size*** property while the job is executing. All of these input-splits are utilized by each of the data blocks. The size of input splits is measured in bytes. Each input-split is stored at some memory location (Hostname Strings). Map-Reduce places map tasks near the location of the split as close as it is possible. The input-split with the larger size executed first so that the job-runtime can be minimized.

Record-Reader:

- Record-Reader is the process which deals with the output obtained from the input-splits and generates it’s own output as key-value pairs until the file ends. Each line present in a file will be assigned with the Byte-Offset with the help of Record-Reader. By-default Record-Reader uses **TextInputFormat** for converting the data obtained from the input-splits to the key-value pairs because Mapper can only handle key-value pairs.

Map:

- The key-value pair obtained from Record-Reader is then feed to the Map which generates a set of pairs of intermediate key-value pairs.

Intermediate output disk:

- Finally, the intermediate key-value pair output will be stored on the local disk as intermediate output. There is no need to store the data on HDFS as it is an intermediate output. If we store this data onto HDFS then the writing cost will be more because of it’s replication feature. It also increases its execution time. If somehow the executing job is terminated then, in that case, cleaning up this intermediate output available on HDFS is also difficult. The intermediate output is always stored on local disk which will be cleaned up once the job completes its execution. On local disk, this Mapper output is first stored in a buffer whose default size is 100MB which can be configured with ***io.sort.mb property***. The output of the mapper can be written to HDFS if and only if the job is Map job only, In that case, there will be no Reducer task so the intermediate output is our final output which can be written on HDFS. The number of Reducer tasks can be made zero manually with job.setNumReduceTasks(0). This Mapper output is of no use for the end-user as it is a temporary output useful for Reducer only.

Is it possible to calculate the number of Mappers In Hadoop?

Yes or No?

If Yen then how?

No then Why?

Yes

- *The number of blocks of input file defines the number of map-task in the Hadoop Map-phase, which can be calculated with the help of the below formula.*
- *Mapper = (total data size)/ (input split size)*

For Example:

- For a file of size 10TB(Data Size) where the size of each data block is 128 MB(input split size) the number of Mappers will be around 81920.

Driver Class

- The major component in a MapReduce job is a **Driver Class.** It is responsible for setting up a MapReduce Job to run-in Hadoop. We specify the names of **Mapper** and **Reducer** Classes long with data types and their respective job names.

Driver Class

```
public class ExampleDriver{   ...
 public static void main(String[] args)
throws Exception   {         // Create a Configuration object that is used to set other options
 Configuration conf = new Configuration() ;        // Create the object representing the job
 Job job = new Job(conf, "ExampleJob") ; // Set the name of the main class in the job jar file
 job.setJarByClass(ExampleDriver.class) ;      // Set the mapper class
 job.setMapperClass(ExampleMapper.class) ;      // Set the reducer class
 job.setReducerClass(ExampleReducer.class) ; // Set the types for the final output key and value
job.setOutputKeyClass(Text.class) ;
 job.setOutputValueClass(IntWritable.class) ;      // Set input and output file paths
FileInputFormat.addInputPath(job, new Path(args[0])) ;
 FileOutputFormat.setOutputPath(job, new Path(args[1]))        // Execute the job and wait for it to complete
 System.exit(job.waitForCompletion(true) ? 0 : 1);      }}
```

Mapper Class:

- A mapper’s main work is to produce a list of key value pairs to be processed later.
- A mapper receives a key value pair as parameters, and produce a list of new key value pairs.
-
- **For Example**:
- From each input to the mapper, the generated list of key value pairs is the key, combined with each of the values separated by comma.
- **Input**: (aaa,bbb,ccc,ddd))
- **Output**: List(aaa 1, bbb 1, ccc 1,aaa 1)

Mapper Class:

```
public static class MapClass extends 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 itr = new StringTokenizer(line);
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        output.collect(word, one);
      }
    }
  }
```

```
Extend the Mapper class with specified input and output key-value types
public static class MapClass extends Mapper<LongWritable, Text, Text, IntWritable> {

    // Initialize a constant IntWritable with value 1
    private final static IntWritable one = new IntWritable(1);

    // Initialize a Text object to store words
    private Text word = new Text();

    // Override the map method to process input key-value pairs
    public void map(LongWritable key, Text value,
                    OutputCollector<Text, IntWritable> output,
                    Reporter reporter) throws IOException {
        // Convert input value to string
        String line = value.toString();
        // Tokenize the input string
        StringTokenizer itr = new StringTokenizer(line);
        // Iterate through each token
        while (itr.hasMoreTokens()) {
            // Set the current token as the word
            word.set(itr.nextToken());
            // Emit the word with count 1
            output.collect(word, one);}}}
```

Shuffler Class:

- After the mapper and before the reducer, the shuffler and combining phases take place. The shuffler phase assures that every key value pair with the same key goes to the same reducer, the combining part converts all the key value pairs of the same key to group and form key,list(values) this is what the reducer ultimately receives.

Reducer Class:

- Reducer’s job is to take the key list(values) pair, operate on the grouped values, and store it somewhere. It takes the key list(values) pair, loop through the values concatenating them to a pipe-separated string, and send the new key value pair to the output.
-
- **For Example:**
- **Input**: [(aaa,List(1,1)),(bbb,List(1)),(ccc,List(1))]
- **Output**: [(aaa,2),(bbb,1),(ccc,1)]

Reducer Class:

```
public static class Reduce extends 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));
    }
  }
```

```
// Extend the Reducer class with specified input and output key-value types
public static class Reduce extends Reducer<Text, IntWritable, Text,
IntWritable> {

    // Override the reduce method to process input key and list of values
    public void reduce(Text key, Iterator<IntWritable> values,
                       OutputCollector<Text, IntWritable> output,
                       Reporter reporter) throws IOException {
        // Initialize a variable to store the sum
        int sum = 0;
        // Iterate through the list of values
        while (values.hasNext()) {
            // Add each value to the sum
            sum += values.next().get();
        }
        // Emit the key along with the sum as the value
        output.collect(key, new IntWritable(sum));
    }
}
```

Main Method:

- Create a instance for the Job Class and set the Mapper and Reducer class in the Main() method and execute the program.

Main Method:

```
public static void main(String[] args) throws Exception
{

            String arguments[] = new String[2];
            //For remote cluster set remote host_name:port instead of localhost:9000
            arguments[0] = "hdfs://localhost:9000/Data/WarPeace.txt"; // Input HDFS File
            arguments[1] = "hdfs://localhost:9000/OutPut"; // Output directory
            Configuration conf = new Configuration();
            Job job = new (conf, "WordCount");
            FileInputFormat.addInputPath(job, new Path(arguments[0]));
            FileOutputFormat.setOutputPath(job, new Path(arguments[1]));
            job.setJarByClass(WordCount.class);
            job.waitForCompletion(true);
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(IntWritable.class);
        job.setMapperClass(MapClass.class);
            job.setReducerClass(Reduce.class);
            job.setInputFormatClass(TextInputFormat.class);
            job.setOutputFormatClass(TextOutputFormat.class);
 }
```

```
public static void main(String[] args) throws Exception {
   // Create an array to hold input arguments
   String arguments[] = new String[2];
   // Set input file path (change if using a remote cluster)
   arguments[0] = "hdfs://localhost:9000/Data/WarPeace.txt"; // Input HDFS File
   // Set output directory path (change if using a remote cluster)
   arguments[1] = "hdfs://localhost:9000/OutPut"; // Output directory

   // Create a new configuration
   Configuration conf = new Configuration();

   // Create a new job with the configuration and a name
   Job job = new Job(conf, "WordCount");

   // Set the input and output paths for the job
   FileInputFormat.addInputPath(job, new Path(arguments[0]));
   FileOutputFormat.setOutputPath(job, new Path(arguments[1]));
// Set the main class for the job (WordCount in this case)
   job.setJarByClass(WordCount.class);

   // Wait for the job to complete
   job.waitForCompletion(true);

   // Set the output key and value classes
   job.setOutputKeyClass(Text.class);
   job.setOutputValueClass(IntWritable.class);

   // Set the Mapper and Reducer classes for the job
   job.setMapperClass(MapClass.class);
   job.setReducerClass(Reduce.class);

   // Set the input and output format classes
   job.setInputFormatClass(TextInputFormat.class);
   job.setOutputFormatClass(TextOutputFormat.class);
}
```