Big Data Processing Concepts II

Topics:

I. **Introduction to Hadoop Architecture: Namenode, datanode, jobtracker,tasktracker**
II. **Exercise: Map/Reduce function coding in WordCount application.**
III. **Compile and Run WordCount**

I. Introduction to Hadoop Architecture: namenode, datanode, jobtracker and tasktracker

The master nodes manage the distribution of the work and coordinate distribution of the data, the worker nodes are the machines that actually perform the work and store the data.
MapReduce is a master-slave architecture

A. Master: JobTracker
B. Slaves: TaskTrackers (100s or 1000s of tasktrackers)

Every datanode is running a tasktracker (see Fig.1).

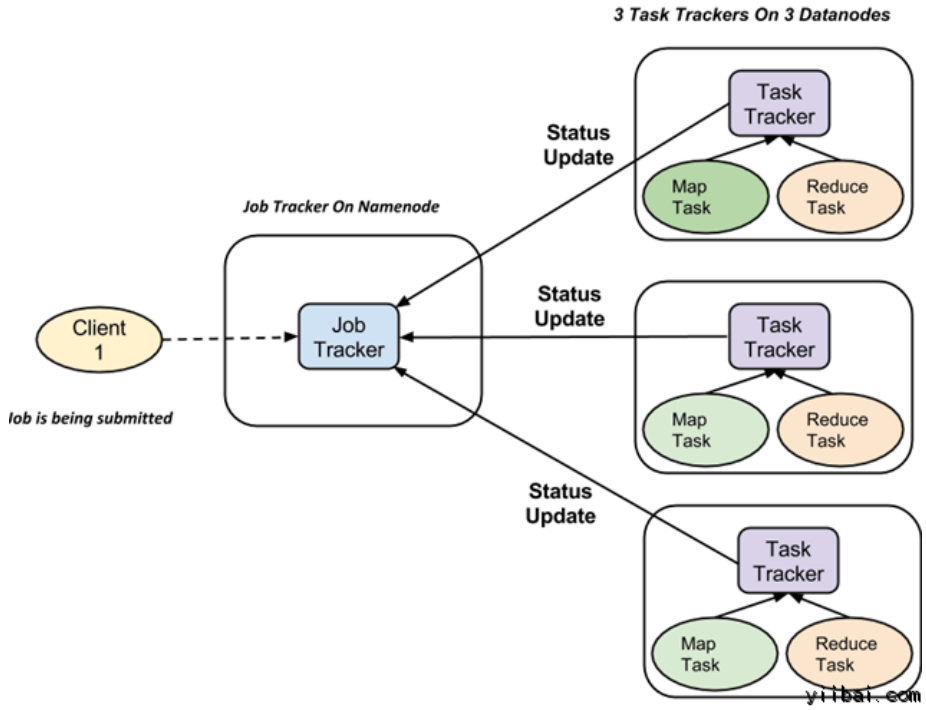

Fig. 1 Simple Hadoop Architecture

1) **NameNode:**
   - Master-slave architecture
   - 1x NameNode
     - Manages namespace, coordinates for clients
     - Directory lookups and changes
     - Block to DataNode mappings
   - Files are composed of blocks
     - Blocks are stored by DataNodes
   - Note: User data never comes to or from a NameNode.
     - The NameNode just coordinates

2) **DataNode:**

   C. Many DataNodes
     - One per node in the cluster. Represent the node to the NameNode
     - Manage storage attached to node
     - Handles read(), write() requests, etc. for clients
     - Store blocks as per NameNode

3) **MapReduce: Hadoop Execution Layer (see Fig.2) :**

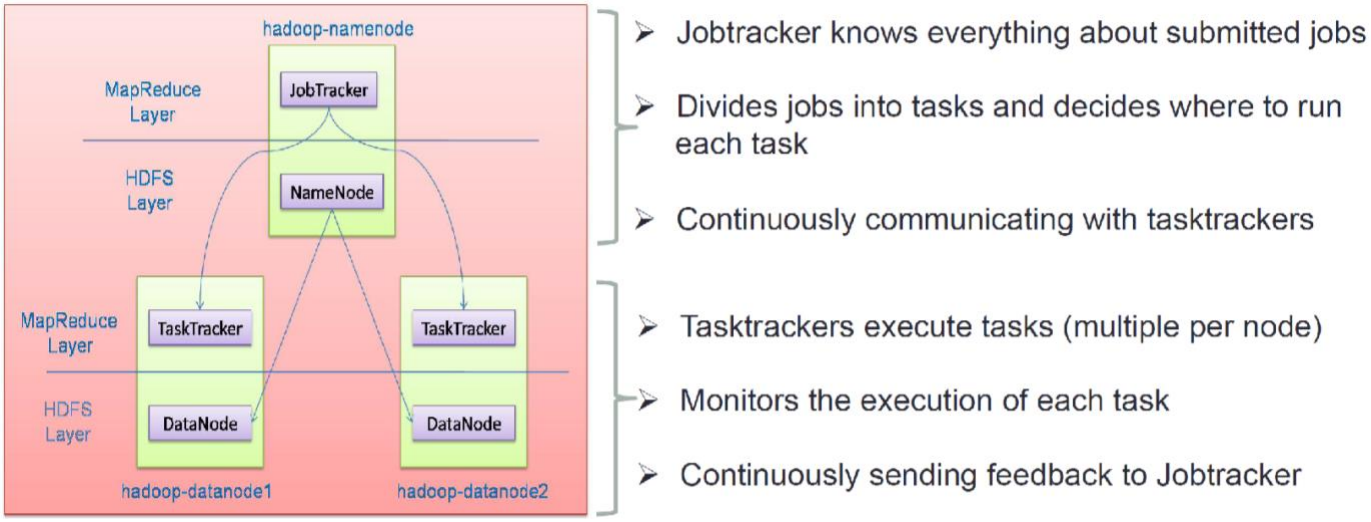

Fig. 2 Hadoop Execution Layer

II. Exercise： Map/Reduce function coding in WordCount application.

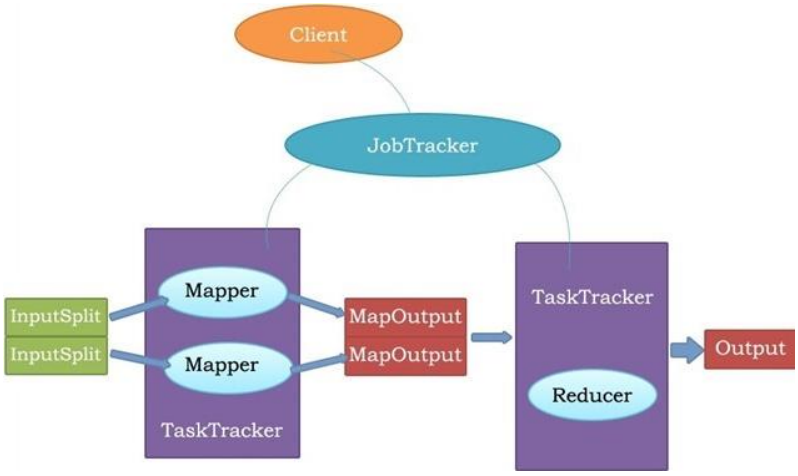

Fig.3 MapReduce execution sequence.

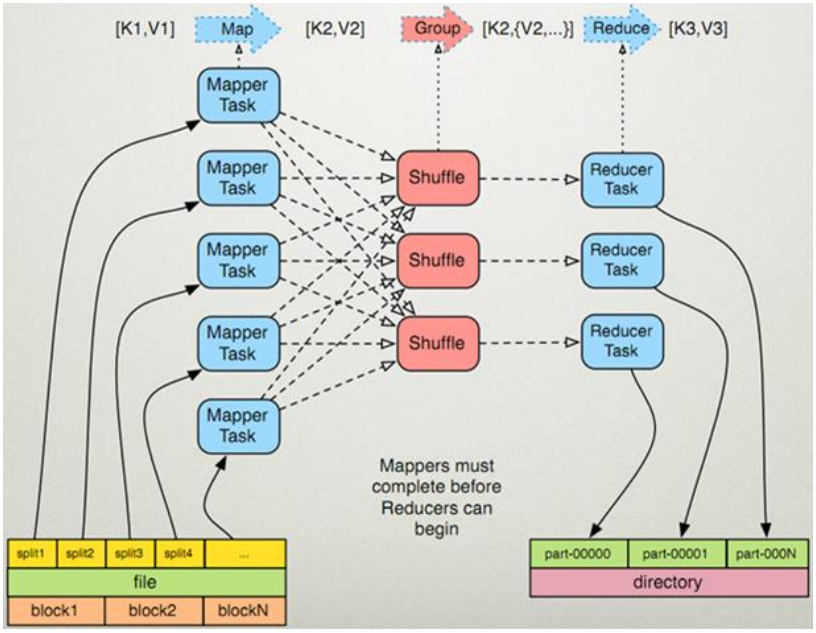

Fig. 4 MapReduce Data Flow.

Exercises: please read Fig.3 and Fig.4 to recall the sequence of MapReduce execution flow. Then comprehend below detailed Map and Reduce Phases and work on the coding exercise in a concrete example of *WordCount* application. In this exercise, our input dataset is *Hello.txt*, which only contains two lines as below.

```
Hello you
Hello me
```

A. Map Phase:

1) **Read:** Read files from HDFS, dataset file is divided into multiple smaller splits, each line is parsed into a key-value pair. Map function invokes per a key-value pair.

Output: Two <K, V> pairs: <0, hello you> and <10, hello me>

2) **Map:** The mapper processes each key-value pair as per the user-defined logic and further generates a key-value pair as its output. When all records of the split have been processed, the output is a list of key-value pairs where multiple key-value pairs can exist for the same key.

Output of Map: <Hello, 1>, <you, 1>, <Hello, 1>, <me, 1>

Exercise1: To get these outputs, implement your Map function here.

```
public static class TokenizerMapper 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 {
        StringTokenizer itr = new StringTokenizer(value.toString());
        while( itr.hasMoreTokens()) {
            word.set( itr.nextToken() );
            context.write( word, one );
        }
    }
}
```

3) **Partition**: The partitioning stage assigning the outputs to specific reducers. The partition function is the last stage of the map task. It returns the index of the reducer to which a particular partition should be sent.

B. Reduce Phase:

1) **Shuffle and Sort**: Output from all partitioners is copied across the network to the nodes running the reduce task. The MapReduce engine automatically groups and sorts the key-value pairs according to the keys. So that the output contains a sorted list of all input keys and their values with the same keys appearing together.

Output: <Hello, {1, 1}>, <me, {1}>, <you, {1}>

2) **Reduce:** The mapper processes each key-value pair as per the user-defined logic and further generates a key-value pair as its output. When all records of the split have been processed, the output is a list of key-value pairs where multiple key-value pairs can exist for the same key.

Output of Reduce:

Hello 2

Me 1

You 1

Exercise2: To get these outputs, implement your Reduce function here.

```
public static class IntSumReducer extends Reducer<Text, IntWritable, Text,
IntWritable>{
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values, Context
context ) throws IOException, InterruptedException {
        int sum = 0;

        for( IntWritable val : values){
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}
```

3) The output of the reducer, that is the key-value pairs, is then written out as a separate file—one file per reducer. The results are saved to the output location in HDFS.