ILLINOIS DATA SCIENCE INITIATIVE

TECHNICAL REPORTS

Processing Custom Input Formats with Hadoop

Author:
Nishil Shah

February 23, 2017

Processing Custom Input Formats With Hadoop

NISHIL SHAH[1,3] **AND PROFESSOR ROBERT J. BRUNNER**[2,3]

[1] *National Center For Supercomputing Applications (NCSA)*
[2] *Laboratory for Computation, Data, and Machine Learning*
[3] *Illinois Data Science Initiative*

Compiled February 28, 2017

I don't know what goes here

https://github.com/lcdm-uiuc

1. INTRODUCTION

Hadoop includes a collection of input formats (DBInputFormat, KeyValueTextInputFormat, NLineInputFormat, etc...) used to read data files in those respective forms. Typically, Hadoop MapReduce jobs split input data into individual sections, operating on each separately. In some cases, it becomes necessary to customize input splits and/or read files in obscure formats. Specifically, in this report we will write custom FileInputFormat and RecordReader classes to read entire files at once.

2. ASSUMPTIONS

This technical report assumes:

- We have the Blockchain downloaded, as shown in "placeholder_report_name".
- The data is stored on HDFS, as explained in "placeholder_report_name".

3. THE DATA

At the time of writing, the Blockchain is stored in a compressed format of over 450,000 binary files. Each file represents one block of the chain. Furthermore, each file contains block metadata as well as a sequence of confirmed transactions. Directly extracting data from a large number of files in this format would be unnecessary difficult and computationally expensive. The files also contain a lot of extraneous information we would like to ignore. Therefore, we will apply MapReduce to transform the Blockchain into a more friendly form.

4. FILEINPUTFORMAT

The FileInputFormat class is primarily responsible for creating input splits from raw data. It also creates a RecordReader which builds key-value pairs from each InputSplit.

To create a custom BlockFileInputFormat we extend FileInputFormat<K,V> where K and V are types of the output key and value pairs. For our purposes, we will use NullWritable and BlockWritable, respectively. NullWritable reads/writes no bytes; we use it as a placeholder for the key. BlockWritable will be introduced later and extended from an external library. To prevent splitting, we simply override `isSplitable()` as shown below.

```
@Override
protected boolean isSplitable(JobContext context, Path
    filename) {
   return false;
}
```

We will need to construct our own RecordReader. So, we add the following:

```
@Override
public RecordReader<NullWritable, BlockWritable>
    createRecordReader(Input split, TaskAttemptContext
    context) {
   return new BlockFileRecordReader();
}
```

That's all we need to implement in our custom FileInputformat class.

5. RECORDREADER

As explained previously, RecordReader<K,V> builds key-value pairs from the input and passes them to the mapper. Let's create our own BlockFileRecordReader. First, we instantiate class variables to hold our input data, generated key-value pair, and a flag:

```
private FileSplit fileSplit;
private Configuration conf;
private boolean processedBlockFile = false;

private NullWritable key = NullWritable.get();
private BlockWritable value = BlockWritable.get();
```

In general, the `nextKeyValue()` function is responsible for reading the FileSplit (the entire file in our case) to set the key and value. Luckily, there exists an open source package called *bitcoinj* which we can use to parse the binary block files. Section 6 discusses how to compile and run Hadoop jobs involving external jar files. The *bitcoinj* library includes a BlockFileLoader which loads blocks by file path and returns an instance of the Block class. From this Block, we can extract just the information we need into BlockWritable, our custom value type. Refer to Technical Report *xxxx* to learn about creating custom key and value types for Hadoop MapReduce.

Our function looks like this:

```
public boolean nextKeyValue() throws IOException {
    if(!processedBlockFile) {
        String filePath = fileSplit.getPath().toString();
        List<File> blockFiles = new ArrayList<>();
        blockFiles.add(new File(filePath));

        BlockFileLoader blockFileLoader = new
             BlockFileLoader(MainNetParams.get(),
             blockFiles);
        if(blockFileLoader.hasNext()) {
            Block block = blockFileLoader.next();
            value = new BlockWritable(block);
        }
        processedBlockFile = true;
        return processedBlockFile;
    }
    return false;
}
```

To successfully extend RecordReader, we must also override the functions `getCurrentKey()`, `getCurrentValue()`, `getProgress()`, and `close()`. `getProgress()` returns how much of the input the RecordReader has processed (0.0 - 1.0). Since we only use each InputSplit once, we can leave `close()` empty.

6. PUTTING IT ALL TOGETHER

Since our custom FileInputFormat and RecordReader are in separate Java files, we package them by adding `package blockparser;` at the top of each file. To assure the classes are found at runtime, we move them inside a new directory `blockparser` in the project directory.

Finally, to put our custom FileInputFormat to use in our main MapReduce driver, we `import blockparser.BlockFileInputFormat;` and configure the Job as follows:

```
Job job = Job.getInstance(conf, "format blockchain");
//...other configurations
job.setInputFormatClass(BlockFileInputFormat.class);
```

Our mapper will now receive a BlockWritable instance which contains block metadata and transaction data. To consolidate all transactions into n output files, we output a number from 1 to n as the key with a TransactionWritable instance as the value. The reducer receives all the transactions with they same key. The reducer then writes all received transactions to a text file. In this new format it is quicker and easier to perform computations. More information on this project can be found in Technical Report *xxxx*.

7. USING EXTERNAL LIBRARIES WITH HADOOP

In this task, we needed to use an open source library *bitcoinj* to handle processing of the binary block files. To compile and run MapReduce using an external jar, add the jar to the classpath. By running `echo $HADOOP_CLASSPATH` on the command-line, you'll see a list of directories that belong to Hadoop's classpath on your system. Moving the jar (*bitcoinj.jar* in our case) to any of those directories does the trick. In addition, the following configuration should be added to your job in the main driver to ensure the jar is located during runtime.

```
job.addFileToClassPath("path/to/bitcoinj.jar");
```

8. CONCLUSION

Hadoop makes it easy to process data in various formats. By writing custom FileInputFormat and RecordReader classes, we can customize the way Hadoop splits and reads input for use in MapReduce.