

ILLINOIS DATA SCIENCE INITIATIVE

PROCESSING CUSTOM INPUT WITH HADOOP

Version: 0.0.1

Author(s): Professor Brunner, Nishil Shah

April 9, 2017

Processing Custom Input With Hadoop

PROFESSOR BRUNNER^{1,2,3} AND NISHIL SHAH^{1,2,3}

¹National Center For Supercomputing Applications (NCSA)

²Laboratory for Computation, Data, and Machine Learning

³University of Illinois

Compiled April 9, 2017

By implementing our own `FileInputFormat` and `RecordReader` for MapReduce, we can gain control over how Hadoop reads files, constructs input splits, and builds key-value pairs for the mapper.

<https://github.com/lcdm-uiuc>

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 to avoid reading incomplete data. Also, we may want to read files in obscure formats. We can accomplish this by writing our own `FileInputFormat` and `RecordReader` classes. Specifically, in this report we will read entire files at once.

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".

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.

FILEINPUTFORMAT

The `FileInputFormat` class is primarily responsible for creating input splits from files. 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

`BytesWritable`, respectively. `NullWritable` reads/writes no bytes; we use it as a placeholder for the key. `BytesWritable`, as the name suggests, stores a sequence of bytes. To prevent splitting the input file, we simply override `isSplittable()` as shown below.

```
language=Java @Override protected boolean isSplittable(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 is all we need to implement in our custom `FileInputFormat` class.

RECORDREADER

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

```
private InputSplit inputSplit; private Configuration conf; private boolean processedBlockFile = false;
```

```
private NullWritable key = NullWritable.get(); private BytesWritable value = new BytesWritable();
```

In general, the `nextKeyValue()` function is responsible for reading the `InputSplit` (the entire file in our case) to set the class' key and value. As previously mentioned, our key is type `NullWritable` and value is type `BytesWritable`. In `nextKeyValue()`, we read in the bytes of the `InputSplit` and set the value to those bytes.

We could do further processing on those bytes in this function to extract specific data if we wanted to. Then, we would need to construct custom `Writable` objects to pass to the mapper. This is explained in detail in "Building Writable Data Types for Hadoop". Instead, we leave this work for the mapper.

Our function looks like this:

```
public boolean nextKeyValue() throws IOException if(!processedBlock) //setup FileSplit fileSplit = (FileSplit)inputSplit; int splitLength = (int)fileSplit.getLength(); byte[] blockBytes = new byte[splitLength]; //get file
```

```
Path filePath = fileSplit.getPath(); FileSystem fileSystem =
filePath.getFileSystem(conf); // read bytes FSDDataInputStream
in = null; try in = fileSystem.open(filePath); IOUtils.readFully(in,
blockBytes, 0, blockBytes.length); value.set(blockBytes, 0, block-
Bytes.length); finally IOUtils.closeStream(in); return
processedBlock = true; 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 `FileSplit` once, we can leave `close()` empty.

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, "for-
mat blockchain"); //...other configurations
job.setInputFormatClass(BlockFileInputFormat.class);
```

Our mapper will now receive a `BytesWritable` instance which contains the entire contents of a block file. The map function will then parse the bytes using an external library, *bitcoinj*. From that, we construct `BlockWritable` and `TransactionWritable` instances which can be sent to the reducer for final processing. More information on this project can be found in Technical Report *xxxx*.

USING EXTERNAL LIBRARIES WITH HADOOP

Luckily, there exists an open source package called *bitcoinj* which we can use to parse the byte contents of each block file. The *bitcoinj* library includes functionality to generate a `Block` object from this sequence of bytes. From this `Block`, we can extract just the information we need into custom data types, `BlockWritable` and `TransactionWritable` for use in MapReduce. Refer to Technical Report "Building Writable Data Types for Hadoop" to learn about creating custom key and value types for Hadoop MapReduce.

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.

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

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.