ILLINOIS DATA SCIENCE INITIATIVE

BUILDING WRITABLE DATA TYPES FOR HADOOP

Version: 0.0.1

Author(s): Professor Brunner, Nishil Shah

April 11, 2017

Building Writable Data Types for Hadoop

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

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

Compiled April 11, 2017

Writable objects in Hadoop can be serialized, allowing for efficient, secure transmission across a system. Let's learn how to construct our own Writable data types.

https://github.com/lcdm-uiuc

INTRODUCTION

Hadoop's MapReduce requires the usage of "writable" data types as input and output. These data types internally handle serialization and deserialization, enabling persistent data storage and retrieval. There are a limited number of provided writable classes, including support for all Java primitives, arrays, lists, maps, and strings (Text). In this report, we will create our own writable data types by implementing the Writable and WritableComparable interfaces.

THE DATA

In technical report "Processing Custom Input with Hadoop" we read and parse entire binary Block files from the bitcoin blockchain. Each file's byte contents are read by the RecordReader and sent to the map function in the form of BytesWritable. The map function constructs Block and Transaction objects using a third-party library (section 7 of the aforementioned report describes how to use external libraries with Hadoop). To finally transmit these objects to the reducer we instead build and use BlockWritable and TransactionWritable classes.

THE WRITABLE INTERFACE

Any data type holding the role of value during MapReduce needs to implement Writable. For now, in our TransactionWritable class, we want to store the unique transaction hash, bitcoin denomination, exchange rate, and time it was initiated. We can declare these variables in TransactionWritable as follows:

public class TransactionWritable implements Writable private String hash; private long amount; private long exchangeRate; private String date;

Implementations are required to override the write() and readFields() functions which take DataOutput and DataInput as parameters, respectively. This pair of functions is necessary for the serialization and deserialization of data when transferred between nodes in a distributed system. For TransactionWritable, they look like this:

public void write(DataOutput out) throws IOException out.writeChars(hash); out.writeLong(amount); out.writeLong(exchangeRate); out.writeChars(date);

public void readFields(DataInput in) throws IOException hash = in.readUTF(); amount = in.readLong(); exchangeRate = in.readLong(); date = in.readUTF();

If we were to store a writable data type within our implementation, we would call its own write() and readFields() functions. For example, if hash was an instance of Text rather than String, we would write with hash.write(out) and read with hash.readFields(in). To initialize and retrieve values from TransactionWritable it is useful to write constructors and setter/getter methods.

THE WRITABLECOMPARABLE INTERFACE

Similarly, any data type intended to be used as a key must implement WritableComparable, an extension of Java's comparable as well as Writable. Therefore, implementing WritableComparable allows use as both a key and a value.

We include write() and readFields() as shown previously to store a block's height, the time it was solved, and its SHA256 hash. Additionally, to satisfy the Comparable property we override compareTo():

public int compareTo(BlockWritable o) int thisHeight = height; int otherHeight = o.height; return (thisHeight < otherHeight ? -1 : 1);

It is essential for keys to be comparable because Hadoop sorts keys prior to reduction. Lastly, it is important (but not required) for our implementation to override the hashCode() function, which is called to handle equals comparisons. All values outputted with the same key after map() are transmitted to the same reducer. Therefore, one object should not output different hash codes on two separate calls to the function. Rather than relying on Java's default hashCode() function for objects, we can supply our our own unique hash code. In this case, no two blocks can share the same height in the Blockchain, so we can do the following:

public int hashCode() return height;

PUTTING IT ALL TOGETHER

To use our newly created BlockWritable and TransactionWritable classes we first package them with the line package datatypes; and move them to a new directory datatypes within the main project folder.

The last thing we need to do is explicitly declare the output types of both the map and reducer phases.

```
Job job = Job.getInstance(conf, "format blockchain"); //...other configurations
job.setMapOutputKeyClass(BlockWritable.class);
job.setMapOutputValueClass(TransactionWritable.class);
//reducer output job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);
```

Now we can use BlockWritable and TransactionWritable in MapReduce.

CONCLUSION

We have seen that it is easy to create Writable data types for Hadoop. By implementing the Writable and WritableComparable interfaces, we can use more complex data types as inputs to our map and reduce function.