ILLINOIS DATA SCIENCE INITIATIVE

TECHNICAL REPORTS

Processing the Blockchain with Hadoop and Spark

Author:
Nishil Shah

April 11, 2017

Processing the Blockchain with Hadoop and Spark

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 May 4, 2017

Using several big data technologies, we process over 100 gigabytes of Bitcoin transactions to yield statistics and trends on the usage of the most popular peer-to-peer electronic currency.

https://github.com/lcdm-uiuc

1. INTRODUCTION

A. Bitcoin

Bitcoin is a peer-to-peer virtual currency built to eliminate the need for intermediaries like businesses and banks who process transactions and handle disputes. Bitcoin accomplishes this with a decentralized system that allows for non-reversible transactions verified by a chain of computational proof, commonly known as the Blockchain. Bitcoin is decentralized in that the entire ledger of transactions is stored on every node in the network. Each node collects new transactions into blocks of a limited size. Nodes simultaneously work on a difficult proof-of-work that when found allows the solving node to broadcast the finalized block to other nodes on the network. All other nodes then must verify the block's transactions to ensure its contents are valid before appending it to their local copy of the chain. Anyone can start a Bitcoin node by installing the Bitcoin Core software.

There are no tangible "coins". Bitcoins are simply a series of digital messages sent from address to address signed with a private key of the sender. Coins can only be spent from an address if a user has its associated private key. Overall, the Bitcoin system is incredibly robust, allowing for quick and secure transactions without regulation of third-parties like financial institutions or governments. More information on Bitcoin can be found in the paper by Bitcoin's creator, Satoshi Nakamoto.

B. Purpose

At the time of writing, 1 Bitcoin is equivalent to over 1200 USD; all the Bitcoins in circulation have a market cap of almost 20 billion dollars. Clearly, the world sees value in Bitcoin as a medium of exchange, store of value, and/or unit of account. A market of such volume is certainly worthy of exploration. Uncovering trends and metrics on the usage of Bitcoins can be useful in predicting future market activity and possibly even help to identify limitations in its underlying system. Emerging big data technologies make it relatively simple to model, process, and analyze the immense amount of transactional data in the Bitcoin network.

Overall, the purpose of this technical report is to find patterns in Bitcoin transactions to better understand how people use Bitcoin and contribute to the growing community. Simultaneously, we aim to show the procedure involved in applying big data technologies to a real world dataset.

2. PREREQUISITES

A. Assumptions

This technical report assumes:

- We have Apache Ambari set up on Openstack as outlined in the technical report, Setting up Ambari on Openstack Nebula.
- Our Openstack cluster has Hadoop, YARN, and Spark installed.

B. Tools

The following technologies are used in our implementation:

- Bitcoin Core v. 0.14.0 [website] [download]
- bitcoinj (for parsing) [website] [download]
- Apache™ Hadoop® [website] [download]
- Apache Spark™ [website] [download]

All software was written in Java, Scala, and Python.

3. PROCESS

First, we downloaded Bitcoin's entire transaction history (as of April 11, 2017). The data includes over 7 full years of bitcoin transactions starting from Bitcoin's launch in January of 2009. This procedure involved retrieving and verifying every single one of the 467,890 blocks in the Bitcoin blockchain (over 212 million transactions). The blockchain's raw format is a sequence of binary files sized about 128 MiB each. Each file contains over 500 blocks. To simplify this data for further analysis, we transform the dataset into a reduced text format which includes only the information relevant to our analysis. After this step, we run calculations using Apache Spark™ to discover interesting statistics and trends about Bitcoin transactions. Finally, we build visualizations from the output data.

4. DOWNLOADING THE BLOCKCHAIN

A. Bitcoin Core

The Bitcoin Core software provides functionality to run a full node on any computer. This is the most secure way to download the blockchain, provided by https://bitcoin.org. Once the node is fully synced with the network, one can also use Bitcoin Core's local wallet to receiving and spending coins.

Since the blockchain is very large, we will download everything to a 7 terabyte volume attached to our cluster's master node:

```
$ cd /mnt/volume
$ sudo wget https://bitcoin.org/bin/bitcoin-core-0.14.0
    /bitcoin-0.14.0-x86_64-linux-gnu.tar.gz
$ tar xf bitcoin-0.14.0-x86_64-linux-gnu.tar.gz
```

To retrieve each block since Bitcoin's genesis block, all we need to do is start the included daemon. By default, it downloads all data to a new directory `/root/.bitcoin` and downloads all data to it. To use our volume, we can create a symbolic link from that directory to `/mnt/volume/bitcoin-0.14.0/.bitcoin`:

```
$ mkdir /mnt/volume/bitcoin-0.14.0/.bitcoin
$ ln -s /mnt/volume/bitcoin-0.14.0/.bitcoin
    /root/.bitcoin
```

Now, to start the daemon:

```
$ ./mnt/volume/bitcoin-0.14.0/bin/bitcoind -daemon
```

The daemon takes a few moments to get started. Slowly, block files will begin to appear in `.../.bitcoin/blocks/`. The blockchain's primary inherent property requires the daemon to download and verify each block independently before moving on to the next to ensure a valid chain. This process takes many hours. There exists an alternative which reduces the daemon's workload.

B. Indexed Blockchains

There are several resources online which offer indexed versions of the blockchain with either the raw block files or a `Bootstrap.dat` file. After downloading an indexed blockchain into the `.../.bitcoin/` directory, the daemon still must verify them and then continue its work from the last file in the indexed state. The daemon is started as shown above.

5. PARSING THE BLOCKCHAIN WITH HADOOP

A. The Data

As mentioned before, the blockchain is stored in a series of binary files:

```
$ ls /.bitcoin/blocks/
...blk00491.dat blk00492.dat blk00493.dat blk00494.dat
   blk00495.dat blk00496.dat blk00497.dat blk00498.dat...
```

Each `blk*.dat` file describes the contents of several blocks, including header information and transactions. The block header contains important metadata like the block's hash, the previous block's hash, and the time it was solved. Each transaction has its own data including (but not limited to) its hash, input and output addresses, and Bitcoin value sent.

This file format is not directly effective for performing large computations. It is difficult to group, sort, and search through transactions in this structure. Moreover, constantly parsing and extracting data from each file is expensive. The files also contain some excess information that we would rather overlook. We can solve this problem by using Hadoop MapReduce to transform the blockchain into a simplified text format.

B. Why MapReduce?

MapReduce is an adequate method for this task because the blockchain can easily be split into equally sized, independent chunks. The processing of one block does not rely on another and thus can be parallelized. We start by parsing each block's byte contents in the map phase to `Block` and `Transaction` objects. The mapper outputs all blocks and transactions as values by using a parent GenericWritable class. We use a date-based key to split the work among several reducers. Each reducer will receive a list of transactions and blocks that occurred in the same month and output them to separate files. The MapReduce output directory looks something like this:

```
 $ hdfs dfs -ls /shared/bitcoin/output
...blocks-2009-2 blocks-2009-3 blocks-2009-4
    blocks-2009-5 blocks-2009-6 blocks-2009-7...
...transactions-2013-2 transactions-2013-3
    transactions-2013-4 transactions-2013-5...
```

C. Writing a Custom Input Format

By default, Hadoop partitions input files into appropriately sized splits before the map phase. This is insufficient given the structure of our data. Obviously, Hadoop does not know how to identify the beginning and end of each block within the `blk*.dat` files, leading to the possibility of incomplete records. Therefore, we can prevent input splitting and read entire input files at once prior to the map phase. We do this by implementing two classes, `FileInputFormat` and `RecordReader`.

C.1. FileInputFormat

The `FileInputFormat` class is primarily responsible for creating input splits from files. It also creates an instance of `RecordReader` that builds key-value pairs from each input split.

To create a custom class `BlockFileInputFormat` we extend `FileInputFormat<K,V>` where `K` and `V` are types of the output key and value pairs (and in turn, input to the mapper). For our purposes, we will use Hadoop's `NullWritable` and `BytesWritable` classes, respectively. `NullWritable` reads/writes no bytes; we use it as a placeholder for the key. `BytesWritable` stores a sequence of bytes. To prevent splitting the input file we can override the `isSplitable()` function:

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

We will also need to override createRecordReader() to return a block file record reader that we will create next:

```
@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.

C.2. RecordReader

As mentioned previously, `RecordReader<K,V>` builds key-value pairs from an input split (the entire file in our case) and task context for the mapper. First, we instantiate class variables to hold input and the generated key-value pair:

```
class BlockFileRecordReader extends
     RecordReader<NullWritable, BytesWritable> {
    private NullWritable key = NullWritable.get();
    private BytesWritable value = new BytesWritable();

    private InputSplit inputSplit;
    private TaskAttemptContext taskAttemptContext;

    private byte[] fileBytes;
    private int fileIndex;
}
```

On initialization, `BlockFileRecordReader` reads the entire input file and populates `fileBytes`. `fileIndex` is initially set to 0 and tracks the current position in the byte array. Given an input split, it is trivial to read its associated file entirely:

```
private byte[] readFile() throws IOException {
    //set up
    Configuration conf =
         taskAttemptContext.getConfiguration();
    FileSplit fileSplit = (FileSplit)inputSplit;
    int splitLength = (int)fileSplit.getLength();
    byte[] blockFileBytes = new byte[splitLength];
    //get file
    Path filePath = fileSplit.getPath();
    FileSystem fileSystem = filePath.getFileSystem(conf);
    //read bytes
    FSDataInputStream in = null;
    try {
        in = fileSystem.open(filePath);
        IOUtils.readFully(in, blockFileBytes, 0,
             blockFileBytes.length);
    } finally {
        IOUtils.closeStream(in);
    }
    return blockFileBytes;
}
```

The `nextKeyValue()` function does the important work, setting the next key and value pair every time it is invoked. Since the type of key is `NullWritable`, we do not worry about setting it. The value is the byte contents of a single block. This function returns true on success and returns false when it is done processing the input split. The general logic follows:

```
public boolean nextKeyValue() throws IOException {
    byte[] blockBytes =
         BlockUtils.nextBlockBytes(fileBytes, fileIndex);

    if(blockBytes != null) {
        value.set(blockBytes, 0, blockBytes.length);
        fileIndex += (4 + blockBytes.length);
        return true;
    }
    return false;
}
```

The `BlockUtils.nextBlockBytes()` function retrieves the bytes corresponding to the next block in the file. This function uses a bit manipulation technique adopted from the bitcoinj library. The specifics of this function can be found here. We increment `fileIndex` by `(4 + blockBytes.length)` because each block is preceded with 4 bytes that specify the block's size in bytes.

To completely extend `RecordReader`, we also override the functions `getCurrentKey()`, `getCurrentValue()`, `getProgress()`, and `close()`. `getProgress()` returns a float representing how much of the input the `RecordReader` has processed (0.0 - 1.0). The other two get functions are used internally by Hadoop when invocating the mapper. Since we already close the input stream after reading all bytes, we leave `close()` empty.

D. Custom Writable Data Types

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.

Recall we use an external library `bitcoinj` to parse each block and its associated transactions. This library provides `Block` and `Transaction` classes. To send these objects from the mapper to the reducer as values, we must use them to create our own writable data types by implementing the `Writable` interface.

D.1. BlockWritable

In the BlockWritable class, we store the following information:

```
public class BlockWritable implements Writable {
    private String hash;
    private String prevHash;
    private String time;
    private long work;
    private int transactionCount;
}
```

To successfully implement `Writable`, our class just needs to override the `write()` and `readFields()` functions. We also write a constructor which takes an instance of `Block` and grabs relevant data from it. `write()` and `readFields()` are necessary to serialize and deserialize writable objects as they are sent across nodes in a distributed system.

```
@Override
public void write(DataOutput out) throws IOException {
    out.writeUTF(hash);
    out.writeUTF(prevHash);
    out.writeUTF(merkleRoot);
    out.writeUTF(time);
    out.writeLong(work);
    out.writeInt(transactionCount);
}

@Override
public void readFields(DataInput in) throws IOException {
    hash = in.readUTF();
    prevHash = in.readUTF();
    merkleRoot = in.readUTF();
    time = in.readUTF();
    work = in.readLong();
    transactionCount = in.readInt();
}
```

We also write a `toText()` function for writing to output. This function concatenates the data fields into a comma separated string of values and returns a Text object initialized with said string.

D.2. TransactionWritable

The process is similar for creating `TransactionWritable`, so we will not go into depth on the same procedure. We care about the following data for transactions:

```
public class TransactionWritable implements Writable {
    private String blockHash;
    private String hash;
    private long value;
    private int size;
    private boolean isCoinBase;
    private String inputs;
    private String outputs;
    private String outputValues;
}
```

The number of input and output addresses vary per transaction. We deal with this by joining each address in a string with the ":" symbol. `outputValues` holds the amount of Bitcoin (in satoshis) sent to each output address and follows the same protocol. Sometimes, it is impossible to decode the original input or output address, in which case we use the string "null" as a placeholder. Example values for transaction `ac8d7dde0667e1e8691bbed0b75742275acf617ff5531a86ffd2129747551b74` is:

```
inputs = "null"
outputs = "1B9R81wLa3b9aaCYaLtKGUcb59EVPcNKju:1PJnjo4n2Rt5jWTUr2wxe1jWAJnY"
outputValues = "4050000:5000000000"
```

Java's `DataOutputStream` enforces a 64KB limit on strings it will serialize. Some transactions had a large amount of inputs/outputs resulting in strings well over the limit. We got around this restriction by writing and reading the string's bytes directly as follows:

```
private void writeLongString(String string, DataOutput
     out) throws IOException {
    byte[] data = string.getBytes("UTF-8");
    out.writeInt(data.length);
    out.write(data);
}

private String readLongString(DataInput in) throws
     IOException {
    int length = in.readInt();
    byte[] data = new byte[length];
    in.readFully(data);
    return new String(data, "UTF-8");
}
```

The rest of the process is the same as `BlockWritable`.

D.3. MessageWritable (implementing GenericWritable)

As stated previously, our mapper outputs both blocks and transactions as values to the reducer. Hadoop, however, does not allow for different value types. To handle this, we implement a wrapper class called `GenericWritable` which allows us to wrap multiple writable types into it. Implementations are required to override the static `CLASSES` variable and `getTypes()`. Our implementation, `MessageWritable`, looks like this:

```
public class MessageWritable extends GenericWritable {
    private static Class[] CLASSES = {
        BlockWritable.class,
        TransactionWritable.class
    };

    public MessageWritable(Writable instance) {
        set(instance);
    }

    @Override
    protected Class[] getTypes() {
        return CLASSES;
    }
}
```

We show how to use this class in section 5.E.

E. MapReduce

We have shown how to construct all the prerequisite components needed to write the main MapReduce driver. The driver consists of implementations of `Mapper` and `Reducer` and the main function where the job's configurations are set.

E.1. Mapper

If you recall, our mapper receives a `NullWritable` key and `BytesWritable` value, the contents of a single block. The structure of our mapper class is:

```
public static class BlockMapper extends Mapper
  <NullWritable, BytesWritable, Text, MessageWritable> {
    //output
    private Text outKey;
    private MessageWritable outValue;

    public void map(NullWritable key, BytesWritable value,
         Context context) {
        //process bytes
    }
}
```

The map function parses the block into a bitcoinj `Block` object using `BlockUtils.parseBlock()`, a wrapper around a bitcoinj library call. Again, the source code of the utilities class can be found here. From a `Block`, we can retrieve all of its included transactions. The full map function follows:

```
//parse block
byte[] blockBytes = value.getBytes();
Block block = BlockUtils.parseBlock(blockBytes);

//write block
BlockWritable blockWritable = new BlockWritable(block);
outKey = BlockUtils.getKey(blockWritable);
outValue = new MessageWritable(blockWritable);
context(outKey, outValue);

//write transactions
String blockHash = blockWritable.getHash();
for(Transaction tx : nextBlock.getTransactions()) {
    TransactionWritable txWritable = new
         TransactionWritable(tx, blockHash);
    outValue = new MessageWritable(txWritable);
    context.write(outKey, outValue);
}
```

Next, we generate writable versions of the block and all its transactions. Since we cannot pass two different value types from the mapper to the reducer, we use them to instantiate `MessageWritable` objects to output with a key. The key is generated from the month and year in which the block was solved. Next, we implement the Reducer class.

E.2. Reducer

Each reducer will aggregate the blocks and transactions that confirmed in the same month. This makes it easy to do date-based computations such as calculating the number of transactions per month or Bitcoin value transacted per month. Mainly, the Reducer will output the data to their respective month-labeled files

with the use of `MultipleOutputs`. Our `Reducer` implementation looks like this:

```
public static class BlockReducer extends Reducer
  <Text, MessageWritable, Text, NullWritable> {
    //output
    private Text key = new TextWritable();
    private NullWritable value = NullWritable.get();
    private MultipleOutputs outputs;

    public void setup(Context context) {
        outputs = new MultipleOutputs(context);
    }

    public void reduce(Text key, Iterable<MessageWritable>
         values, Context context) {
        //process data
    }

    public void cleanup(Context context) {
        outputs.close();
    }
}
```

With `MultipleOutputs`, we can specify any file name to write to. We will write to different files depending on whether the value is a block or a transaction in the reduce function:

```
for(MessageWritable mWritable: values) {
    Writable message = mWritable.get();
    if(message instanceof BlockWritable) {
        key = ((BlockWritable)message).toText();
        outputs.write(key, value, "blocks");
    } else if(message instanceof TransactionWritable) {
        key = ((TransactionWritable)message).toText();
        outputs.write(key, value, "transactions");
    }
}
```

And that's it!

E.3. Using External Libraries With Hadoop

To compile and run MapReduce using an external jar, we add the jar to the classpath. Running `echo $HADOOP_CLASSPATH` on the command-line returns a list of directories that belong to Hadoop's classpath on the system. Moving the jar (*bitcoinj.jar* in our case) to any of those directories does the trick. In addition, we must add the following configuration to the job in the main driver to ensure the jar is located during runtime.

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

E.4. Putting It All Together

We then configure the MapReduce job in the main driver so it knows which classes and types to use. First, we set a configuration:

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    conf.setBoolean("mapreduce.map.out.compress", true);
}
```

Hadoop saves intermediary results between the map and reduce phases. By compressing the intermediary output, we save disk space. The rest of the main driver configures the job itself:

```
public static void main(String[] args) throws Exception {
    ...
    Job job = Job.getInstance(conf, "reduce_blockchain");
    //include jars in classpath
    job.setJar("rbc.jar");
    job.addFileToClassPath(new
         Path("/user/nishil/bitcoin/bitcoinj.jar"));
    job.setMapperClass(BlockMapper.class);
    job.setReducerClass(BlockReducer.class);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(MessageWritable.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(NullWritable.class);
    job.setInputFormatClass(BlockFileInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);
    job.setNumReduceTasks(8);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
```

We are required to set the mapper and reducer class and their respective output value types. In addition, we set the input format to the custom `BlockFileInputFormat` class we created in section 5.C.

The `BlockFileInputFormat`, `BlockFileRecordReader`, and `BlockUtils` classes are packaged using `package blockparser`. We also package the writable datatypes, `BlockWritable`, `TransactionWritable`, and `MessageWritable` into `package datatypes`. The main MapReduce job in `ReduceBlockchain.java` is in the main directory.

The following commands are used to compile the source code into a jar file and run the job:

```
$ hadoop com.sun.tools.javac.Main \
    *.java blockparser/*.java datatypes/*.java
$ jar cvf rbc.jar \
    *.class blockparser/*.class datatypes/*.class
$ yarn jar rbc.jar ReduceBlockchain \
    /shared/blockchain /shared/reduced_blockchain
```

6. ANALYZING THE BLOCKCHAIN WITH SPARK

A. Project Setup

As mentioned earlier in the report, our Openstack cluster already has Spark install. For this part of the project, however, we will be developing off-cluster in Intellij IDEA. Using the sbt-assembly plugin, we can package our source code and all dependencies into a fat JAR and easily run it on the cluster. This is great because we do not have to worry about dependency differences between the local machine and the cluster.

First, we create a new SBT project in Intellij IDEA, setting the Scala and SBT versions of our local machine. To add sbt-assembly to our project, we create an `assembly.sbt` file in the main project directory and add this line to it:

```
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.14.4")
```

Pressing the "Import Project" button on the top right corner of Intellij refreshes its dependencies. Next we set other properties of the project in the `build.sbt` file, including the scala version and spark libraries among other things. `build.sbt` looks like this:

```
name := "analyze_blockchain"
version := "1.0"

scalaVersion := "2.11.7"
val sparkVersion = "2.1.0"
```

```
libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % sparkVersion,
  "org.apache.spark" %% "spark-sql" % sparkVersion,
  "org.apache.spark" %% "spark-yarn" % sparkVersion
)
```

To make this work, we need to download the associated Spark libraries and import them into the project. We are ready to get started.

B. Determining Address Growth Per Month

One of the many metrics used to roughly gauge Bitcoin adoption is the number of new addresses used per unit of time. At first glance, it does not seem intuitive to solve this with Spark because it involves finding the first use of each address. It is actually rather easy to solve this problem with a series of maps and reduces in Spark.

First, we import all blocks and transactions:

```
import org.apache.spark.SparkContext

object NewAddresses {
    def run(inputPath: String, outputPath: String, sc:
         SparkContext) {
        //set full input paths
        val blocksPath = "hdfs://" + inputPath + "/blocks*"
        val transactionsPath = "hdfs://" + outputPath +
             "/transactions*"
        //import data
        val blocksRDD = sc.textFile(blockPath)
        val transactionsRDD = sc.textFile(transactionsPath)
    }
}
```

Remember that data in each line is comma delimited. For each block, we only need its hash and solve time, so we can map each line to just those two elements. In addition, the solve time needs to be parsed from a string into `DateTime` object that can be compared. So we can add the following lines:

```
val dateFormat = new SimpleDateFormat("EEE MMM dd
     HH:mm:ss Z yyyy")

//split data by commas
val blocks = blocksRDD.map(line => line.split(",")) \
    .map(arr => (arr(0), dateFormat.parse(arr(3))))
val transactions = transactionsRDD \
    .map(line => line.split(","))
```

Specifically, we are interested in output addresses here. Input addresses do not necessarily provide any insight, because users can create and add unlimited input addresses to a transaction. An increase in new output addresses per month can imply an increase in transactions to new or distinct users. To isolate output addresses:

```
val outputAddresses = transactions.flatMap(arr => arr(6) \
    .split(":").map(addr => (arr(0), addr))) \
    .filter(tx => tx._2 != "null")
```

Since addresses are delimited with colons in our reduced dataset, we split the string by ":" and map each output address to a separate tuple. We also remove addresses that are "null", which resulted from an inability to decode certain input/output scripts.

At this point, all records in `blocks` and `outputAddresses` use block hash as their key. Therefore, we can join both RDDs to get the times the associated transactions were confirmed in their respective blocks. After this, we can reduce the resulting RDD by key, continuously leaving the earliest date as the value. Lastly, we convert the date's format to just a year and month.

```
//join blocks and output addresses
val combined = blocks.join(outputAddresses).map(entry =>
     (entry._2._2, entry._2._1))
//get month each address was first used
val addressFirstSeen = combined.reduceByKey((a, b) =>
     if(a.before(b)) a else b) \
      .map(entry => (entry._1, newFormat.format(entry._2)))
```

To finally get the number of new addresses per month, we swap keys and values and count the number of values per key:

```
val outputFilePath = "hdfs://"+outputPath+"/new_addresses"

//number of new addresses per month
addressFirstSeen.map(entry => (entry._2, 1)) \
    .reduceByKey(_ + _).repartition(1) \
    .saveAsTextFile(outputFilePath)
```

The final RDD is stored in the specified `outputFilePath`. Results are shown in section 7.B.

C. Looking at Address Use & Reuse

It is interesting to see the way addresses are used. For instance, we can produce a distribution of the frequency and number of times addresses are used for transactions. One would suspect the addresses used the most are deposit addresses for online gambling sites or donation addresses.

Using the same method shown in the previous section, we import the blocks and transactions. From the transactions, we are able to extract both input and output addresses and combine the two RDDs as they share the same schema. Just like earlier, we join the resulting RDD with the blocks to retrieve solve times:

```
val combined = blocks.join(addresses)
```

In two lines we can compute the use count for addresses and a distribution of use count versus the number of addresses with that use count:

```
val addressUseCount = combined.map(entry => \
    (entry._2._2, 1)).reduceByKey(_ + _)
//key: use count, value: num addresses used count times
val addressUseTimes = addressUseCount.map(_.swap) \
    .map(entry => (entry._1, 1)).reduceByKey(_ + _)
```

After, we can calculate the average number of times an address is used with the expression `totalAddressCount / uniqueAddressCount`, where `totalAddressCount` is the total number of inputs and outputs in all transactions and `uniqueAddressCount` is `addressUseCount.count()`.

A similar strategy making use of the block solve times gives us the frequency of address use. Full code is shown here.

D. Printing To A File on HDFS

To save an RDD to HDFS, the `saveAsTextFile(/path)` comes in handy. However, Spark partitions the output file by default, which we want to avoid given the small size our job's output. To do this, we use the `repartition(1)` function with the parameter 1 to shift all the data to a single node prior to saving it to a text file.

In some cases, we also use the `PrintWriter` class to print out some custom calculations. We've created this helper function

which builds a `PrintWriter` from an input path:

```
def printWriter(outputPath: String): PrintWriter = {
    val path = new Path(outputPath)
    val conf = new Configuration()
    val fileSystem = FileSystem.get(conf)
    val output = fileSystem.create(path)
    return new PrintWriter(output)
}
```

The above code identifies the file system based on the Hadoop configuration, creates an output stream to it, and finally returns a `PrintWriter`. This method is useful for printing any type of data to HDFS.

E. Running The Job

Now we are ready to package the source code into a JAR to run on our cluster. To do this, we run the following commands in the base project directory:

```
$ sbt
[info] Loading project definiton from /path/to/project
[info] Set current project to analyze_blockchain
> assembly
...
```

After this, we see many merge conflicts between identical classes in different libraries. To alleviate this, we add merge strategies in the `build.sbt` file. Merge strategies specify what to do in the situation of a merge conflict. Here is a snippet of our merge resolution strategies:

```
assemblyMergeStrategy in assembly := {
  case PathList("org","aopalliance", xs @ _*) =>
       MergeStrategy.last
  case PathList("javax", "inject", xs @ _*) =>
       MergeStrategy.last
  case PathList("javax", "servlet", xs @ _*) =>
       MergeStrategy.last
  case x if x.endsWith(".html") => MergeStrategy.discard
  case x =>
    val oldStrategy = (assemblyMergeStrategy in
         assembly).value
    oldStrategy(x)
}
```

The merge strategies above are applied depending on the file path. For the first three cases, we take the last file. If the conflicting files ends with ".html", we discard them because HTML files aren't significant to the usage of the libraries. Lastly, there is a default case for all other files. The full set of merge strategies can be seen here.

After including sufficient merge strategies, running the two commands `sbt` and `assembly` properly assembles the FAT jar, which can be found at path `target/scala-{scala-version}/analyze_blockchain-assembly-1.0.jar`. Then we send the JAR to our cluster using either an FTP client or `scp` from the command-line.

Finally, we use `spark-submit` to run the Spark job. We include the JAR file, the main class and its input arguments. The main class runs the appropriate code for the specified task so that we don't need an individual jar for separate tasks (which would require multiple copies of dependencies).

```
$ spark-submit --class AnalyzeBlockchain \
   analyze_blockchain-assembly-1.0.jar <task> \
   /path/to/input /path/to/output
```

7. RESULTS

A. MapReduce

The MapReduce job ran in 1 hour and 24 minutes, reducing the original 111.3 GB of compressed, binary data to 73.7 GB of text.

Block data is outputted in the following format:

```
<hash>,<prevHash>,<merkleRoot>,<time>,<work>,<version>,
<transactionCount>
```

An example entry is:

```
000000000000000002eec3b07facdb475a338ddf00b5d771ae6875833\
19eae65,000000000000000000609b5f89c999e13e53577f45b333de3\
003dd6f764c44a0,2020b99ca4f1fe778d274ec12c95e69a0c9f0ff94\
2badeeaffb5d22782e139f8,Wed Oct 26 01:20:39 UTC 2016,\
940796024229891532,536870912,2497
```

Transaction data follows this format:

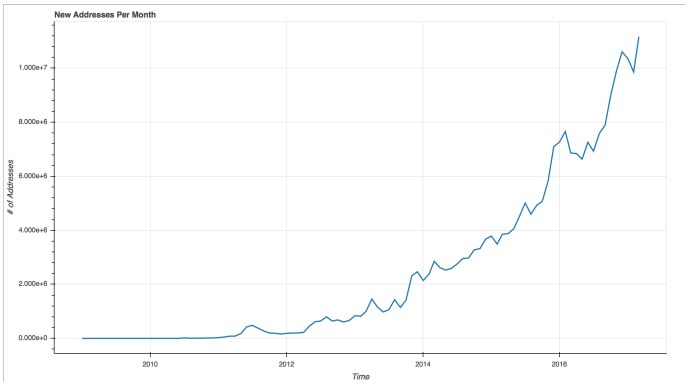
```
<blockHash>,<hash>,<value>,<size>,<isCoinBase>,<inputs>,<outputs>,
<outputValues>
```

Example:

```
00000000058bd2da17c28ab57f5119d67de2af90bd3209b7fb0318a46\
b7971ab,1572000000,4fdf94fc860dea8c725db7deba79cb9c8549d7\
d84bdd2615e866f462e219c1fd,258,false,19VdEX8QCwgMXfXWm9K7\
kmL8p121jRab9V,1CnU8X8u15hbyA3PPrxXNM2N2dXbyorGoV:18gBZns\
uSrhYLjvPUgwvDUJmksfREUGBTT,1571000000:1000000
```

B. Address Growth Per Month

The exact results from running this analysis can be seen here. In the following graph, we have omitted results from April 2017 due to incomplete data from that month.



There is an obvious rapid and increasing growth in the number of new addresses used per month.

C. Address Reuse

8. CONCLUSION

In this technical report, we downloaded and performed an analysis on the full Bitcoin blockchain from the currency's inception until April 11, 2017. We discovered an upward trend in new addresses used per month, supporting an increase in Bitcoin adoption. We also looked into the reuse of addresses, which provides many insights. We found that XX percent of addresses are only used once and that addresses are reused an average of X.X times. This gives an estimate of how long people hold onto Bitcoin before spending it.

We have also seen the procedures involved in processing a large real world dataset with big data technologies such as Hadoop and Spark. Overall, Bitcoin is the biggest and most popular virtual currency in the world today. Every day, hundreds of blocks are being added to the blockchain that we explored in this technical report. This technical report is just a glimpse into the amount of information stored in the blockchain and the vast potential for big data applications in this area.