Paper 3153-2019

SAS®-Coin: Understanding Blockchains Through Simple Data Step Examples

Seth Hoffman, GEICO

ABSTRACT

Have you ever wanted to start your own small economy, but had no idea how to handle the recordkeeping? In that case, it is time to learn about what "blockchain" technology can do. Using Base SAS® code examples, this paper will illustrate the principles behind creating a blockchain system to provide an unalterable record of validated transactions for your project.

INTRODUCTION

A blockchain is a method of ensuring that data, accumulated and recorded serially over time, remains unaltered. The method for ensuring this immutability is public broadcast. This can be through any medium, such as emails or newspaper advertising[1]. If there is only one copy of a blockchain and someone edited it, there would be no way know anything happened. When many people have copies of a blockchain, that one edited version will not match with the all the other copies and will be ignored.

The amount and type of information in each block is a free choice for the designer of a blockchain. Blocks can store plain text, complex data schemes, or even software code. These types of blocks can then be used to create simple document archives, Bitcoins, or Ethereum. Besides the information contained in the body of a block, there is also a part called the header. The header section contains information needed to fully summarize a block and connect it to the previous and future block.

To make a "chain", each block must link to the one that preceded it. In a fully trusted environment, a simple numerically ordered naming convention would suffice. When the ability to create block becomes decentralized, more than just trust is needed. To ensure one block is building on the last, each new block should contain information about the previous one. Including the summary of the previous block into the current one provides this assurance.

One method of creating a reasonably sized summary of data is using a hash function (not to be confused with the DATA STEP Hash Object).

HASHING

A hash function is an algorithm which translates an input message of any length into an output message of a fixed length. In order to be useful for cryptographic applications, a hash algorithm should be deterministic, always producing the same output given the same input. The function should be fast to compute. Also, the output should be irreversible.

Irreversibility comes in part from the data reduction aspect of the algorithm. While there are an infinite number of input strings, there are only 2^{256} output strings. So, for any given output it would be impossible to know which of the potentially infinite input messages created it. Another important property is that a small change to the input should produce a large difference in the output.

As of release 9.4M1, Base SAS includes the 256 bit version of commonly used "Secure Hash Algorithm 2" cryptographic hash function[2] :

```
DATA Secure_Hash_Algorithm_256bits;
   FORMAT hash_hex $hex64. hash_char $64.;
   hash_hex = sha256("Unscramble this!");
   hash_char = INPUT(PUT(hash_hex, $hex64.), $64.);
RUN;
```

The algorithm produces a 256 bit binary number. To make the output slightly more readable, hexadecimal is often used instead ($2^{256} = 16^{64}$). If directly saved as a character string, every 8 bits (2 hexadecimal digits) will be displayed as its respective character on the ASCII table. This is problematic as there are a number of non-printing control characters.

Both variables will contain the value:

```
595F0F6347B202F49131E4C80B1EE540736C958418ACEE4EE2482EC8256BE9BD
```

Without the FORMAT statement, ASCII values of the number would be:

```
Y_ cG²ô‘1äÈ-å@sl•„ ¬îNâH.È%ké½
```

According to the ASCII table, "59" (the 1st two digits of the hexadecimal number) correspond to the text value "Y". The next two hex digits, "59", translate to "_". Looking at an ASCII table, it is possible to verify the twenty-eight other characters and find where the two non-printing characters are supposed to be.

SUMMARIZING DATA

To validate that no data has been changed, it is important that all data is used in the generation of the hash. The actual method or order of collecting the data is arbitrary. All that matters is that the method is consistent and replicable by anyone trying to audit the blockchain.

```
DATA block2;
  x = "A"; y = 3.0; z = "01JAN2019"d; OUTPUT;
  x = "B"; y = 4.1; z = "15MAR1994"d; OUTPUT;
RUN;

%MACRO data_summary(dataset, last_summary);
   DATA _NULL_;
      SET &dataset END=eof;
      FORMAT file_summary $hex64.;
      RETAIN file_summary;
      row_summary = SHA256(CATS(x, y, z));
      IF _N_ EQ 1 THEN file_sumamry = row_summary;
      file_summary = SHA256(CATS(row_summary, file_summary));
      IF eof THEN DO;
         file_summary = SHA256(CATS(file_summary, last_summary));
         CALL SYMPUTX("file_summary",
                      INPUT(PUT(file_summary,$hex64.),$64.));
   RUN;
   %PUT &file_summary;
%MEND;

%data_summary(block2, block1_summary);
```

The above code shows one possible method of summarizing all the data in a block. All the variables in a row are concatenated and that concatenation is then hashed. The hash of each row is then combined with that of the next row, and that combination is hashed again. This summarization. The previous block's summary is merged in at the end and included in the final hash.

PROOF OF WORK

Imagine that there is a mature blockchain system with thousands of blocks. A malicious actor modifies the records in block number 23. Because it is a blockchain, they not only have to recalculate the hash of their new block 23, but they have to pass that to block 24 so that block's hash can be recalculated. To have a valid blockchain, where each block follows the implementation rules, they continue this till all existing blocks have been recalculated. At this point they can start to argue that they have the "real" blockchain and everybody should follow their lead.

Hash functions are quick to calculate, so this type of problem could cause real headaches for a system where anybody can create new blocks. One solution is known as including a "proof of work"[3] requirement to create a block. One simple method is to require the hash value for a block be less than a certain number. As the data shouldn't change, a "nonce" value is added to the hash, and incremented each time until a hash value below the target threshold is generated:

```
DATA proof_of_work;
   block_data = "SAS Global Forum 2019";
   nonce = 0;
   difficulty_target = "000000";
   DO UNTIL (SUBSTR(INPUT(PUT(SHA256(CATS(block_data, nonce)), $hex64.),
                          $64.), 1, 6) EQ difficulty_target);
      nonce = nonce + 1;
   END;
RUN;

NOTE: DATA statement used (Total process time):
      real time           6.06 seconds
      cpu time            6.02 seconds
```

The more 0's a fixed-length formatted number starts with, the smaller it is. The above example is looking for a hash value that starts with at least six 0's. Such a value was found in 6 seconds, after 2,866,473 tries. If this is an average amount of time to recalculate a block, than even a chain with thousands of blocks could be altered and recalculated in a few hours. Making the target a little bit harder, finding a hash starting with seven 0's, greatly increases the required number of tries (479,860,822) and the time to find it.

```
NOTE: DATA statement used (Total process time):
      real time           17:11.06
      cpu time            17:11.28
```

Beyond helping to prevent malicious edits, blockchains like Bitcoin use proof of work as a contest to see who gets to mine the next block. Every miner tries to be the first to find a winning hash value. Including a timestamp in the hash input allows a miner to prove they were the first one to create a new block. If many miners are active and creating new blocks too often, the difficulty target is increased. Conversely, if less miners are active and it takes longer for blocks to be mined, the difficulty target will be decreased.

ANONYMITY AND PROOF OF OWNERSHIP

The ubiquitous method of proving our identities to a computer system is the password. Only you, and whoever looks at the post-it notes on your desk, will be able to provide it when challenged. Public distributed systems are not good places to store a password that should be kept secret. It would be like storing everyone's money in a big pile and saying "please take only what is yours". Fortunately, the power of math provides a solution in the form of public key cryptography.

With a public key system, everyone has a public key and a private key. A message encoded with a private key can only be decoded with its corresponding public key. Ownership of a message can be proved by sending a public key, a plain-text message, and the encrypted version of the message. Only the holder of the private key would have been able to encrypt the message such that it would be decrypted by that public key.

The first non-classified public key system[4] was based on prime numbers and modular arithmetic. The following code shows the simple math for encoding, also known as "digitally signing", and decoding a message:

```
DATA public_key_example;
   modulus = 3131;     /* = 31 x 101. Small primes are not secure */
   private_key = 1663; /* Derived by math from [31,101] */
   public_key = 727;   /* Derived by more math from [31, 101] */
   plain_text = 2723;  /* The message! */

   cypher_text = plain_text;
   DO i = 1 TO (private_key - 1); /* Modular arithmetic to encrypt */
      cypher_text = MOD((plain_text * cypher_text), modulus);
   END;
   PUT cypher_text;    /* "2909", what could this really mean ??? */

   recovered_text = cypher_text;
   DO i = 1 to (public_key - 1); /* Modular arithmetic to decrypt */
      recovered_text = MOD((cypher_text * recovered_text), modulus);
   END;
   PUT recovered_text; /* "2723" matches the original message*/
RUN;
```

Bitcoin uses the public key as the unique identifier for a group of coins that can be spent. Whoever has the private key, and no one else, can then create and digitally sign a message saying they are spending those coins. Besides the number of coins spent, the message includes a new public key belonging to the new owner of the coins. Without permanent identifiers, transactions can remain anonymous.

Computers have grown so powerful that the above example would require prime numbers that are 4,096 bits long. Because these long numbers are so unwieldy, public key systems now rely on interesting properties of elliptic curves. While the math to encrypt and decrypt messages is easy to implement, understanding how it works is a much harder endeavor.

STARTING A MINER[5]

By making sure that every block in chain follows the rules, miners are how a distributed blockchain is implemented. For an example blockchain currency, miners would have to go through the entire chain to derive a current list of what coins are spendable. Granting a little bit of currency to whoever creates a new block is one way to entice people to run miners for your blockchain.

Once a data format and block structure has been defined, you should reward yourself by creating the first block. The following example sketches out how a very simple blockchain could be organized:

```
DATA blockchain;
   INPUT block_data $;
   DATALINES;
1c-3131-727-650-50
1h-164427464-468548638-1000-248008-650
2i1-3131-727-50-2773-50
2o1-5141-5-20
2o2-7373-3571-30
2c-4141-369-235-50
2h-650-628352786-1000-1960319-235
;
RUN;
```

The above dataset contains rows for two blocks of data. The first row, starting with "1c", defines who gets the coin rewards for that block. The next row, starting with "1h", is the header row containing information about the data in that block. The next rows represent transactions, the rewards, and the header for the second block.

Datasets whose rows have different numbers of fields are easily handled by reading each row in as a single string. Processing a single row of data at a time might be very constraining for datasets with multiple types of rows, especially if they need to interact. To handle out of order row processing, the Hash Object (not to be confused with hash function) makes life very easy.

The following example shows code for reading in the header row. Coin rewards, input, and output transactions would each be read in and have their own hash object with similar code. An additional Hash object would be needed to maintain the list of currently unspent coins:

```
DATA _NULL_;
   If _N_ = 1 THEN DO; /*Define the Hash Object for the header*/
      LENGTH h_index h_prevHash h_merkle h_difficulty h_nonce h_blockId
             i_index i_address i_message i_signature i_amount
             o_index o_address o_message
             u_address u_message $20.;
      DECLARE HASH h(ordered:"a");
      DECLARE HITER hi("h");
      rc = h.defineKey("h_index");
      rc = h.defineData("h_prevHash", "h_merkle", "h_difficulty",
                        "h_nonce", "h_blockId", "h_index");
      CALL MISSING(h_index, h_prevHash, h_merkle, h_difficulty,
                   h_nonce, h_blockId);
   END;
   SET blockchain END=eof; /*Read in blockchain*/
   re = PRXPARSE('/h(\d)-(\d+)-(\d+)-1000-(\d+)-(\d+)/');
   IF PRXMATCH(re, block_data) THEN
      h.add(key:PRXPOSN(re,1,block_data), data:PRXPOSN(re, 2, block_data),
      data:PRXPOSN(re, 3, block_data), data:"1000",
      data:PRXPOSN(re, 4, block_data), data:PRXPOSN(re, 5, block_data),
      data:PRXPOSN(re, 1, block_data));
```

Once the data is all read in, the miner starts its work of checking all the transactions. The first block is usually hardcoded as there can't be transactions till at least one person has something to spend. In this example, once "`3131-727`" has their 50 coins, they spend them with the coins now going to "`5141-5`" and "`7373-3571`". Besides making sure that the

header is valid for each block, a miner should also validate the transactions. To be valid, the input's digital signature, "2773" would decrypt to the message: "50". Without further blocks, the two output transactions and the miner of the second block would be the list of unspent coins.

```
IF eof;
   rc = hi.first(); /*start with the first block*/
   rc = c.find(key:i);
   rc = u.add(key:c_address, data:c_message);
   DO x = 2 TO /* however many blocks */;
     rc = hi.next();
     /*Example only has 1 input per block, so no loop for inputs*/
     rc = i.find(key:COMPRESS(x || "1")); /*load the input*/
     rc = u.find(key:i_address); /*find it's unspent transaction*/
     unspent_coins = i_message;
     DO y - 1 TO 2; /*example  has 2 outputs*/
        rc = o.find(key:COMPRESS(x || y); /*load the output*/
        IF (rc EQ 0) THEN DO; /*if the output exists*/
           unspent_coins = unspent_coins - o.message;
           rc = u.add(key:o_address, data:o_message);
        END;
     END;
   rc = u.remove(key:i_address); /*coins are now spent*/
   rc = c.find(key:x);
   miner_reward = PUT(INPUT(CATS(c_message + unspent_coins),4.),$20.));
   rc = u.add(key:c_address, data:miner_reward);
END;
rc = u.output(dataset:"unspent_coin_transactions");
RUN;
```

CONCLUSION

How a blockchain is implemented is highly dependent on the type of problem it is looking to solve. The number of users, the amount of trust, the need for secrecy, the level of decentralization, and other factors go into determining what will make a useful and robust blockchain. With only a few centralized trusted users, a blockchain might be implemented just for the sake of keeping up with buzzwords and trends. On the opposite end of the spectrum, a blockchain utilizing the built-in trust created by the math of hash functions and public key encryption could be an elegant solution. Regardless of the ultimate complexity of a final implementation, it will have started with the few high-level concepts presented here.

REFERENCES

1 Oberhaus, Daniel. "The World's Oldest Blockchain Has Been Hiding in the New York Times Since 1995." Motherboard. August 27, 3028. Available at https://motherboard.vice.com/en_us/article/j5nzx4/what-was-the-first-blockchain.

2 Hemedinger, Chris. "A fresh helping of hash: the SHA256 function in SAS 9.4m1." SAS. Jan 18 2014. Available at https://blogs.sas.com/content/sasdummy/2014/01/18/sha256-function-sas94/

3 Dwork, Cynthia and Naor, Moni. 1993. "Pricing via Processing, Or, Combatting Junk Mail, Advances in Cryptology". *CRYPTO'92: Lecture Notes in Computer Science No. 740*. Springer: 139–147. Available at http://www.wisdom.weizmann.ac.il/~naor/PAPERS/pvp.ps

4 Rivest, R.L, Shamir, A., and Adelman L. Feb 1978. "A method for obtaining digital signatures and public-key cryptosystems". Communications of the ACM, volume 21 Issue 2.:120-126. Available at https://people.csail.mit.edu/rivest/Rsapaper.pdf

5 Hoffman, Seth Oct 2018. "Mining Bitcoins: A Step-by-Data Step Simulation" SESUG 2018. Available at https://www.lexjansen.com/sesug/2018/SESUG2018_Paper-107_Final_PDF.pdf

RECOMMENDED READING

- "Bitcoin: A Peer-to-Peer Electronic Cash System". Available at https://bitcoin.org/bitcoin.pdf

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Seth W Hoffman
GEICO
301-986-3072
shoffman@geico.com