Adaptive Enhanced Bloom Filters for Identifying Transactions of Interest in a Blockchain

Thomas Jay Rush
QuickBlocks.io
July 5, 2017

At their heart, blockchains store lists of transactions. While these transactions are, in every case, originated from and directed towards individual accounts, they are not ordered or indexed by account. Instead, the blockchain orders transactions in an append-only fashion based on the originating time of the transactions. This imposes the task of collating lists of transactions for a given account or collection of accounts on other parts of the system. Current implementations (i.e., web30.js) place this imposition on the application developer. QuickBlocks® intends to relieve the application developer of this burden, and in this paper, we describe optimizations we've developed that simultaneously allow for timely accumulation of per-account lists of transactions and minimize the resources needed to accomplish this task. It this attention to minimizing resource usage on the target machine that allows QuickBlocks® to realize a fully decentralized solution to this problem. **Keywords: blockchain, Ethereum, data analytics, blockchain auditing, blockchain accounting**

Introduction

Blockchains store lists of transactions. These transactions are included in a block on a time-oriented basis depending on many factors including a potentially arbitrary choice of the miner building the block. The account addresses[1] that initiate or receive transactions are, of course, stored on the blockchain, but not in an easily accessible or indexed manner. This implies that quickly building lists of transactions, given a particular address or a series of addresses, is time consuming and more difficult than might be expected.

This difficulty is exacerbated by the fact that a transaction's receiving address may be a "smart contract" which may further initiate transfers of value or what are commonly called "internal transactions." Obtaining a list of "internal transactions," particularly those received by an address (called herein an "incoming internal transaction"), is an onerous process.

One obvious method of obtaining per-account lists of transactions is to index the addresses. We believe, however, that this imposes too high a burden on the target machine. Our software, QuickBlocks® alleviates this problem while avoiding direct indexing of accounts. In this way, we are able to build a fully decentralized solution for the problem of accumulating lists of transactions of interest per account.

[1] We use the terms 'account' and 'address' interchangeably in this paper.

In a recent paper [3], we describe a method of obtaining off-chain, decentralized lists of transactions per-account from blockchains. In that paper, we describe a method of visiting each block in a local cache searching for transactions of interest. That process, while able to build an accurate list of transactions, is slow, requiring traversing every transaction (both internal and external) in every block.

In that previous paper, we furthermore present an optimization to per-block traversal called "Enhanced Bloom Filters." Enhanced Bloom Filters allows us to minimize the search time of our algorithm.

The current paper discusses an improvement to Enhanced Bloom Filters that we've called "Adaptive Enhanced Bloom Filters." Adaptive Enhanced Bloom Filters significantly improves upon all previous methods of identifying per-address lists of transactions of interest that we are aware of.

The reader is advised to familiarize him or herself to the idea of a Bloom filters [5]. This will allow the reader to better understand the following discussion.

Bloom Filters

A bloom filter is a fixed-length array, initially empty, of m bits representing members in a set. An arbitrary set member is inserted into the filter by hashing its value and then using one of k selection functions to choose a bit to *twiddle* in the filter (where *twiddle* means set to true). This process is repeated for each member in the set.

At a later time, if one wishes to check for the existence of a particular item in the set, one first creates a "candidate" bloom filter from that item alone. If the item is in the set, the same k_i bits will be set in both the candidate bloom filter and bloom filter representing the set. Bloom filters may report false positives, as the same k_i bits may have been set by more than one inserted member, but a bloom filter will never report a false negative. If one of the k_i bits is not twiddled in the filter, the item cannot have been a member of the set. This allows one to eliminate from further consideration any items that are known not to be in the set.

The value of m and k are arbitrary depending on the desired characteristics of the bloom filter. Higher values of m will lead to a lower probability of false positives as will higher values for k. However, increasing both of these values comes at the cost of higher processing and memory costs. For our application, we've chosen values (following the Ethereum Yellow Paper [1]) of m = 4096 and $k = 3$.

Bloom Filters in Existing Blockchains

Existing blockchains utilize bloom filters for various reasons. In the Ethereum blockchain for example, bloom filters are used in support of a pub/sub model of delivering notifications to distributed applications of triggered events. Here, they are used in order to identify addresses and other data involved in (or created during) the production of "events" or "log entries" (see [1] and [5]). The Ethereum blockchain stores bloom filters at every transaction that produces logs. These transaction-level bloom filters are then "rolled-up" to the block level. We call these "node-generated bloom filters." Originally, we stored these blooms in our binary cache, which imposed a storage cost of XXX bytes for the first 4,000,000 blocks[2].

While this data is useful for supporting pub/sub models and other purposes, the existing bloom filters did not store the information we needed to build lists of transactions per account. In particular, they did not store set membership for all the initiating and receiving addresses for both external and internal transactions. We sought a method to create bloom filters that would provide this information, while at the same

[2] Measurements for this paper were made at block 4,0000,000 on or around July 14, 2017.

time eliminating the existing bloom filters from the QuickBlocks binary cache. We refer to this method as Enhanced Bloom Filters.

Enhanced Bloom Filters (Version 1)

In our first, naïve, implementation of Enhanced Bloom Filters, we stored individual bloom filters per block as detailed in [3]. While this method works well to quickly identify blocks that contain transactions of interest per account, it has some drawbacks. Chief among these drawbacks is the fact that the number of files needed to store the bloom filters was of the same order as the number of blocks processed, $O(n)$. We produce Enhanced Bloom Filters per block following the method described in the "The Algorithm" section below.

While Enhanced Bloom Filters do eliminate the processing of blocks known not to contain transactions of interest, and thereby lower the processing cost of finding such transactions, and while they furthermore eliminate the need to store per-transaction bloom filters in the binary cache, they, undesirably, require that we open a file from disc at each block.

Enhanced Bloom Filters (Version 2)

As presented in [3], we improve upon the naïve per-block Enhanced Bloom Filter method by preparing filters for each group of five blocks (an arbitrarily chosen value). This process produces a consistent number of bloom filters per block (twenty bloom filters per one hundred blocks). One desirable side effect of this method is that it provides an easily calculable number representing the storage requirements for the bloom filters. Another desirable effect is that the number of disc reads is reduced by a factor of five. This method improves on our naïve method by lessening the number of files stored on disc by a factor of five, as well.

However, the method has a significant shortcoming: in many cases, these five-block bloom filters are either underutilized or oversaturated. The number of transactions per block ranges widely from zero to hundreds of transactions. Furthermore, the number of internal transactions per external transactions may range widely as well, from zero internal transactions for a simple value exchange to many hundreds of internal transactions for some smart contracts.

The present idea creates what we call Adaptive Enhanced Bloom Filter. Adaptive enhanced bloom filters "fill up" to a consistent level before being written to storage. This creates lower number of more-fully-utilized bloom filters during low-traffic periods while creating (possibly more) less-fully-saturated bloom filters during high-traffic times.

Given the nature of the blockchain data, where the number of transactions per block (and therefore the number of addresses per block) varies significantly from block to block, this both lessens storage and increases effectiveness of each bloom filter.

Adaptive Enhanced Bloom Filters

In order to address the higher-than-necessary storage costs and the less-than-effective nature of using a fixed number of blocks per bloom filter, we've implemented Adaptive Enhanced Bloom Filters. In this method, we store a bloom filter when it "fills-up" as opposed to after inserting a fixed number of blocks or elements. This both lessens the storage requirements and increases the effectiveness of each filter.

In order to understand this idea, we need to define a simple measure, called $bits_{twiddled}$, that, while perhaps a bit whimsically named, has a simple and precise definition:

$$bits_{twiddled} = \sum_{i=0}^{nbits} bit_i$$

That is, the number of true bits in the filter. We further define $pct_{twiddled}$, which is defined as

$$pct_{twiddled} = \frac{bits_{twiddled}}{m}$$

where m is the number of bits in the filter (in our case, 4096).

It is now a simple matter to describe the algorithm for storing a less imposing amount of data on the target machine while simultaneously lowering each filter's rate of false positives thereby improving the overall performance of building decentralized per-account lists of transactions of interest. Following this we present supporting data arguing for the effectiveness of our solution.

The Algorithm

Each block in a blockchain stores zero or more transactions. Each transaction stores, among other things, an initiating address (i.e. *to*) and a receiving address (i.e. *from*). A transaction's receipt may optionally store *contractAddress* which, if present, represents a newly created smart contract. Each of these addresses is "of interest" if one wishes to build a list of transactions per account as we do.

If the receiving address is a smart contract, there may also be one or more *traces* present in the transaction which include *to* and *from* fields of their own. These addresses are "of interest" as well. (Traces may be arbitrarily deep as smart contracts may call into other smart contracts which may further call into still more smart contracts up to the block's gas limit.)

There are many other data fields in a transaction where addresses may appear. For example, they may be embedded as a parameter in the input data field or as part of a trace. Other possible appearances of addresses are in the receipt's *logs* data structure.

We choose not to store these addresses in the Adaptive Enhanced Bloom Filter under the argument that they are redundant and therefore are not of interest.

To understand why these other addresses are redundant, consider a smart contract which stores all data sent to it. Perhaps the data is a list of addresses. Because we are interested only in addresses that participated in a transaction, these data are not of interest. The fact that addresses are being stored is of no consequence.

We therefore distinguish 'addresses of interest' (as listed above) and other types of addresses that may appear in transactions.

Only addresses that are directly involved by either initiating or receiving external or internal transactions are inserted into the bloom filters. Addresses stored as data are only interesting to the extent they are later participate in a transaction.

After identifying an address of interest, we insert it into the bloom filter. We do this by first hashing the address thus:

$$hash = sha3(address)$$

We then apply, in turn, each of the k selection functions which extract the lower 11 bits of i^{th} byte in hash, thereby identifying the bits that need to be twiddled in the current set-level filter.

$$bit\_index_i = byte_i(hash) \ \% \ 4096$$

The algorithm, which stores a minimum number of bloom filters with a maximum effectiveness as determined by the parameterized value of pct_{max} (for which we used an experimentally chosen value of 4.8%.) is presented in Table 1, and has been modified for clarity.

Results

In the following section, we present four measures of effectiveness for each of the three methods of searching for transactions of interest. These methods are (1) a straight-forward traversal of each transaction's data,

Table 1

"Build Adaptive Bloom Filters"

```
bloom_current = 0
for each block, b, in the chain:
  for each tx, t, in block b:
    let l = addresses of interest
        found in t (as above)

    for each address, a, in list l:
       bloom_candidate = to_bloom(a)
       bloom_current |= bloom_candidate

    if (pct_twiddled >= pct_max):
      write bloom_current to disc
        at block_number
         bloom_current = 0

    if bloom_current ≠ 0
      write bloom_current to disc
        at block_number
```

at each block, naively looking for transactions of interest; (2) a search using enhanced, but non-adaptive, bloom filters, and (3) a search using adaptive, enhanced bloom filters.

For each method, we've presented the storage requirements necessary to store the bloom filters. Following that we've presented (A) average $pct_{twiddled}$ in groups of 10,000 blocks as well as charts representing $pct_{twiddled}$ per block across the 4.0 million blocks, (B) performance timing for two different collections of addresses for each of the three methods, and (C) the percentage of false positive results from the bloom filters.

The performance measurements are shown under two scenarios. First, using a highly-activity and long-lived smart contract system called *singularDTV*, and second, using a randomly selected list of non-smart contract addresses.

Storage Requirements

The binary blocks stored in the QuickBlocks™ binary cache contain bloom filters at each transaction that emitted one or more log entry. These bloom filters are "rolled up" to the block level where the node uses them to filter for pub/sub listeners.

For our purposes, these "node-generated" bloom filters are inadequate for two reasons: (1) they store only addresses involved in the generation of log entries, and (2) they store event "topics" which do not help us in identifying addresses of interest.

Because we are, in effect, re-creating these node-generated bloom filters and enhancing them by including addresses of interest, we remove the native blooms from our binary cache. This significantly lessens the amount of data we need to store in our cache, as well as improves overall performance of the system due to the less time it takes to read each binary block. This consideration is reflected below. We generated the following data in late July of 2017 and chose to stop our analysis at 4,000,000 blocks.

Table 2 - Number of Blooms

Method	# of Blocks	Bloom Files
Node-generated bloom	4.0 mil	11,481,569
Non-adaptive enhanced bloom*	4.0 mil	768,462
Adaptive enhanced bloom	4.0 mil	446,192
** five blocks per bloom*		

For each of the three methods, we visited 4.0 million blocks. "Bloom Files" represents the number of separate files needed to store the bloom filters under each scenario.

Because the node-generated blooms are store internally to the cached binary data, we've represented the number of blocks and transactions that have non-zero bloom filters under the node-generated row.

The non-adaptive enhanced bloom filter method stores a bloom filter for every five blocks regardless of the activity level of the blocks, therefore one would expect to store approximately 800,000 files. Because many blocks are empty, there is slightly less bloos than expected.

The number of bloom files stored under the adaptive enhanced method is nearly half of that for the non-adaptive method as is to be expected. This reflects the fact that the adaptive method more fully utilizes the blooms.

Below we present the bytes required to store the above-mentioned number of files. As can be seen from the table below, the storage requirement needed to store enhanced bloom filters decreased nearly 50% over the node-generated method (which are inadequate for our purposes anyway).

The adaptive enhanced bloom filter method shows another nearly 41% decrease in data storage requirement over the non-adaptive method for a total decrease in storage of 70% over the node-generated method.

Table 3 – Bytes Required to Store Blooms

Method	MB Total
Node-generated	406,355,661
Non-adaptive enhanced	206,587,456
Adaptive enhanced	121,364,224

We next consider the fluctuation of the $pct_{twiddled}$ value both as an overall percentage of all blooms and on a per-block basis.

Variation in $pct_{twiddled}$

Besides lowering storage requirements, one of the main motivations of the Adaptive Enhanced Bloom Filter method was to ensure a more consistent, lower, expected false positive rate per filter than the other methods.

Using the non-adaptive method (where we store exactly five blocks per bloom) many of the blooms were highly under-utilized. For example, all of the first 46,146 blocks were empty. Notwithstanding this face, the non-adaptive method stored 9,228 bloom filters for those blocks.

On the other hand, if a series of five blocks each contained 100s of transactions, each with potentially 100s of internal transactions (as happened during the cleanup period pursuant to the spurious dragon hard fork) the bloom filter was overwhelmed. This resulted in false positives reports for every address checked.

The following table presents the average percentage of bits twiddled for each method. Following that, we present the same data broken into groups of 250,000 blocks so as to reveal the effect of increased blockchain usage over time.

Table 4 - Percent of Bits Twiddled

Blocks	Node Generated	Enhanced Non Adaptive	Enhanced Adaptive
0 to 4,000,000	5.70%	4.20%	7.99%

Next, we present the same data separated by groups of 250,000 blocks as we wish to analyze the effect of growing network usage on our work.

Table 5 - Percent of Bits Twiddled per 250,000

Blocks	Node Generated	Enhanced Non Adaptive	Enhanced Adaptive
to 250,000	2.89%	0.57%	5.92%
to 500,000	3.77%	0.74%	5.71%
to 750,000	4.12%	1.00%	5.46%
to 1,000,000	4.21%	1.69%	5.48%
to 1,250,000	4.61%	2.34%	5.92%
to 1,500,000	4.38%	2.81%	6.09%
to 1,750,000	5.14%	3.45%	6.36%
to 2,000,000	5.04%	3.63%	6.15%
to 2,250,000	5.19%	3.10%	6.37%
to 2,500,000	5.92%	2.95%	5.92%
to 2,750,000	5.41%	2.96%	6.29%
to 3,000,000	5.42%	2.97%	6.04%
to 3,250,000	5.66%	3.70%	6.28%
to 3,500,000	5.80%	5.61%	6.74%
to 3,750,000	6.11%	11.49%	7.68%
to 4,000,000	6.19%	16.84%	8.43%

The remainder of our analysis is presented in the charts in the appendices.

Conclusion

A common experience when one first enters the blockchain space is astonishment at the beautiful idea of a shared global ledger. Many people's first reaction, as was ours, is,

"Wow. I can finally figure out what's going on with my money."

A enthusiastic new entrant into the space may believe the claims that the data is accessible to everyone But, what one quickly realizes once one starts the process of looking at the data is that while the blockchain data is stored perfectly *for a blockchain*, it is stored poorly for accounting and auditing uses.

This is due to a complete lack of a node-generated per-account index. QuickBlocks™ attempts to alleviate this discrepancy between what a new user might expect of the data and what one actually finds the deeper one digs.

We intend to continue to explore ways to deliver fast, accurate, per-account lists of transactions in a fully decentralized manner. Stay tuned. ⊡

REFERENCES

[1] Wood, Gavin, "Ethereum: A Secured Decentralized Generalized Transaction Ledger; EIP-150 Revision (8bb760b - 2017-01-19)" accessed 2017-01-28 at https://ethereum. github.io/yellowpaper /paper.pdf.

[2] Buterin, Vitalik, "A Next-Generation Smart Contract and Decentralized Application Platform" accessed 2017-01-28 at https://github.com/ethereum/wiki/wiki /White-Paper

[3] Rush, Thomas Jay, "Decentralized, Off-Chain, per-Block Accounting, Monitoring, and Reconciliation for Blockchains"; July 2017.

[4] Rush, Thomas Jay, "Faster, Richer, Fully Customizable Data from Programmable Blockchains"; January 2017.

[5] Wikipedia. n.d. Bloom filter. Accessed 07 01, 2017. https://en.wikipedia.org/wiki/ Bloom_filter.