BTCSpark: Scalable Analysis of the Bitcoin Blockchain using Spark

Jeremy Rubin
jr@mit.edu

December 16, 2015

1 Introduction

There is a large demand in the Bitcoin research ecosystem for high quality, scalable analytic software. Analysis can help developers quantify the risks and benefits of modifications to the Bitcoin protocol, as well as monitor for errant behavior. Historians might use Blockchain analysis to understand how various events impacted on chain activity. Corporations can use analysis to understand their customers better[1]. To quote Madars Virza, Co-Inventor of Zerocash and researcher at MIT CSAIL, "I need to quickly prototype ideas for my research, but parsing the Blockchain for each project is an arduous task, so I'm forced to speculate. The research world is in great need for programmable Blockchain analysis tools."

In order to serve this need, I have developed BTCSpark for my Large Scale Systems (6.S897) final project under Matei Zaharia. BTCSpark is a layer on top of Apache Spark for analyzing the Bitcoin Blockchain. It provides an easy to use, flexible, and good performance environment for researchers and developers to query the Blockchain and to build Blockchain analysis tools. BTCSpark is open source software, in contrast to almost all other user-friendly Blockchain analysis tools available today.

BTCSpark can be said to be a forward thinking project. Currently, the Bitcoin Blockchain is 50 GB – certainly feasible to fit it all into memory. However, as the Blockchain grows, and as users of the Blockchain call for it to grow at a faster rate, there will be more and more data. Not only that, but the Blockchain data is in a minimal data format. Fully expanding all of the connections and richness of the data set can require much more memory, even when the underlying set is only 50GB (considering each transaction as a node with potentially

[1] Sadly, Bitcoin lacks strong anonymity guarantees

With any luck, BTCSpark will serve as the de facto way for researchers to perform and share reproducible Blockchain analysis.

Before reading this paper, I recommend reviewing Appendix A if you are not highly familiar with Bitcoin, it provides some bare bones information about the protocol and data available.

2 Previous Work

A lot of the work on this topic is either not being done or closed source and secret as API based companies want to sell their analysis capabilities and miners gain competitive advantages by being able to execute these rules more efficiently. However, there are several related categories of prior art.

2.1 Block Explorers

Generally, there is a class of websites which provide tools to analyze the blockchain called block explorers. There are numerous examples [1] [2] [3] [4] [5] [6].

Many block explorers are just open source frontends and proprietary backends. It is unclear how performant the backend service is. Worryingly some of them have been faulted for improper behavior [11]. Academics doing research in the space have been critiqued for their reliance on block explorers for this reason [14]. There is some variety in offered features, but generally the API is somewhat consistent. Furthermore, the APIs do not allow for highly customized queries as may be needed.

2.2 Blockparser

Blockparser [9] is a single machine architecture for parsing and querying the blockchain. They provide the following runtimes for various operations on a generic ubuntu server with 8GB of RAM, performed in June 2012. The Blockchain was around 1GB at this point, now it is 44GB, so operations are likely much more than 40 times slower, as the entire chain can no longer fit into memory trivially. Figure 1 gives some expected run times showing what amount of time it might take to run, if it scales perfectly linearly or with an additional factor of 2 for GC overhead. Furthermore, such a solution is not robust for the future. The Bitcoin blockchain continues to grow, and infrastructure needs to be ready and available for when it does.

1 GB Input Time (seconds)	40GB Input No Overhead Time (minutes)	40GB Input ×2 Overhead Time (minutes)	Description
1	0.66	1.33	Compute simple blockchain stats, full chain parse
20	13.33	26.66	Extract all transactions for a popular address
20	13.33	26.66	Compute the closure of an address, that is the list of addresses that provably belong to the same person
30	20	40	Compute and print the balance for all keys ever used in a TX since the beginning of time

Figure 1: Expected bounds on BlockParser performance

2.3 Bitcoin Node Software

There are various implementations of Bitcoin node software which may be relevant to this project. All nodes provide some API by which blocks, blockheaders, and pending transactions can be queried. There are no great performance comparisons available among implementations, but Bitcoin-Core is likely the most performant presently.

2.3.1 ACINQ

ACINQ is a Scala Bitcoin library [7]. ACINQ is in Beta and not ready for full use.

2.3.2 Bitcoinj

Bitcoinj is a Java implementation of the Bitcoin protocol [8]. There are some concerns over the correctness of some features in the code, but the core functionality seems fine.

2.3.3 btcd

Btcd is a golang implementation of the bitcoin protocol [10]. It aims to be bug-for-bug compliant with Bitcoin core.

2.3.4 Bitcoin-Core

Bitcoin-Core is the main implementation of Bitcoin.

```
cdef class LazyTransactionInput:
    cdef bytes a
    cdef size_t offset
    cdef uint32_t i
    def __cinit__(self, bytes a, size_t o, uint32_t i):
        self.a=a
        self.offset = o
        self.i = i
    def __call__(self):
        return TransactionInput.of_buffer(self.a,
                                &self.offset, self.i)
    def __reduce__(self):
        return (self.__class__, (self.a,
                                self.offset, self.i))
```

Figure 2: An example of a lazy object, which only stores the offset and a pointer to the buffer needed

2.3.5 Relay Only Node

Matt Corallo has a relay only node network he built which is not fully verifying but provides much faster block and transaction propagation than that of alternatives. By connecting BlockSpark to the Relay Only network, a performance advantage could be gained if it is to be used in a streaming context. According to Corallo, "It is in use in one way or another by the majority of major miners" [12].

2.4 Academic Analysis

Fergal et al have a great analysis paper, but it has no performance information [13]. It seems the authors have gone on to found QuantaBytes to make their tools available.

Adi Shamir has a bitcoin analysis paper, but it also lacks performance information [15]. His analysis face harsh criticism for using unreliable block explorers [14]. Their code is not available.

3 Design

BTCSpark will, naturally, be using Apache Spark as a basis of it's design. The general idea will be to develop a Bitcoin Blockchain parsing library that integrates well with spark, and then run it on an Amazon AWS Spark cluster. The Bitcoin Blockchain data will be manually loaded from a user owned Bitcoin Node – this is important so that the data is correct, to address Garzik's complaint. The library will take advantage of two key concepts to gain performance.

```
def TOAD(self):
    self.fetch_chain().map(unlazy)\
        .flatMap(lambda b: b.txns)\
        .map(unlazy).flatMap(lambda txn:
                        map(lambda txo:
                        (txo.value & ~0x3FFF, 1),
                        txn.tx_outs.map(unlazy)))\
        .reduceByKey(lambda x,y: x+y)\
        .saveAsTextFile(sb.output_to("TOAD"))
```

Figure 3: The TOAD Benchmark

3.1 Laziness with Eager Free

The code will not create objects until it is absolutely necessary. Even though the data, being stored in serial without indexes, often requires a full parse in order to get the offsets, by not allocating objects performance gains can be realized. Furthermore, lazy allocation will eagerly free its objects (not memoizing the computation), as once offsets are computed it is easy to recompute the objects needed, and freeing the memory for other computations will be good for performance. Figure 2 shows what one of the Lazy objects looks like – the real object, and it's sub objects only get allocated on "__call__".

3.2 Cython

Cython is a tool for Python which can compile a dialect of Python to C, and compile that C to a shared object which can be imported as a normal Python Library. This Cythonized version is typically much faster, even with minimal modifications as it removes interpretation overhead. With more modification, the use of native C types can allow the compiler to make even greater optimizations. BTCSpark uses Cython to improve performance at the library level.

4 Performance

4.1 Benchmarks

4.1.1 Transaction Output Amount Distribution

In Bitcoin, each transaction output specifies a an amount of bitcoin to associate with itself subsubsection A.2.5. The Transaction Output Amount Distribution Benchmark, or TOAD for short, buckets the transactions by value, ignoring the bottom 14 bits (about 10 cents USD as of December 16, 2015).

This is a great benchmark for figuring out the performance of this system because it looks at a lot of the data (all the transaction outputs) and is useful for showing that laziness does not have high overheads.

The output of the TOAD analysis is shown in Figure 4.

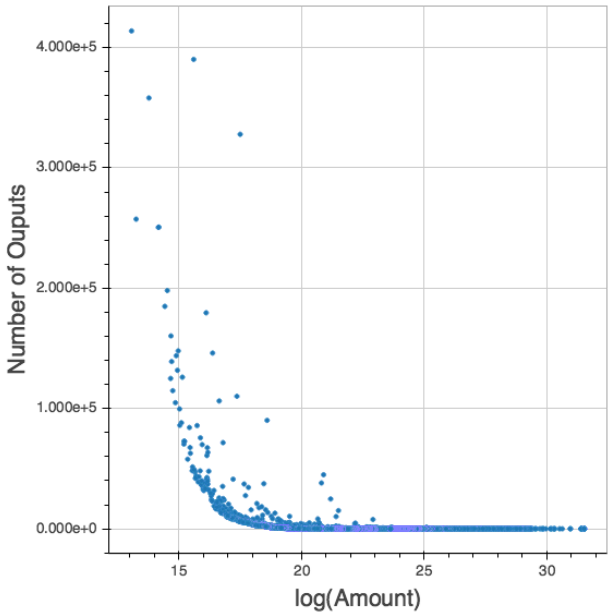


Figure 4: Output of the TOAD function

4.1.2 BIP100

In Bitcoin, protocol changes are made via the Bitcoin Improvement Protocol (BIP), and are assigned numbers. BIP 100 had miners voting for protocol changes by putting the text "BIP100" into the script signature subsubsection A.2.3 of the coinbase transaction [2].

In order to find all of the BIP100 blocks, the BIP100 BenchmarkFigure 5 can be used. This script fetches the chain from hadoop, and then looks at the first transaction in every block.

This is an excellent measure for emphasizing the effectiveness of lazy evalu-

[2]https://github.com/jgarzik/bip100/blob/master/bip-0100.mediawiki

```
def BIP100(self):
    self.fetch_chain().map(unlazy)\
        .map(lambda b:
             b.txns[0].tx_ins[0].signature_script)\
        .filter(lambda f: "BIP100" in f)\
        .saveAsTextFile(self.output_to("BIP100_Blocks"))
```

Figure 5: The BIP100 Benchmark

ation because only a very small amount of data (one transaction) is needed per block.

4.2 Eager Implementation

The TOAD and BIP100 benchmarks were run using the eager implementation. The performance, shown in Figure 6, while not bad, is not fantastic.

Cores	Machines	TOAD	BIP100
10	5	21	19

Figure 6: Running the TOAD and BIP100 Benchmarks on a 5 slave cluster with eager evaluation. The cluster had the blockchain files available with a replication factor of 1.

4.3 Laziness

The Lazy implementation brings considerable performance gains to the BIP100 Benchmark. This is because it parses significantly less data than the eager implementation, which will have to parse the entire block. However, the lazy implementation does not fair worse than the eager implementation on TOAD, which suggests that despite having to parse all the offsets twice[3], the overhead is not significant. This confirms the motivation for using laziness in a way that only saves allocations and keeps memory free is worthwhile.

Cores	Machines	TOAD	BIP100
10	5	21	6.7

Figure 7: Running the TOAD and BIP100 Benchmarks on a 5 slave cluster with lazy evaluation. The cluster had the blockchain files available with a replication factor of 1.

4.4 Cythonized

The Cythonized implementation brings further gains in performance as shown in Figure 8. While the BIP100 Benchmark runs in about 75% of the time, the TOAD Benchmark takes 40% of the compute time. This is because BIP100 is mostly IO bound, whereas TOAD has a much more intense computational load.

Cores	Machines	TOAD	BIP100
10	5	8.4	5.0

Figure 8: Running the TOAD and BIP100 Benchmarks using the cythonized code on a 5 slave cluster. The cluster had the blockchain files available with a replication factor of 1.

[3]due to the serialized format

4.5 Scalability

To evaluate the scaling performance the lazy cythonized code was run in varying cluster configurations as shown in Figure 9. This shows that as the number of machines increases, the speed of the computation decreases exponentially. This is a fantastic result, and should give confidence that BTCSpark will be a scalable analysis platform as Bitcoin grows. For reference, scaling analysis was also done on the same cluster as the earlier section in Figure 10.

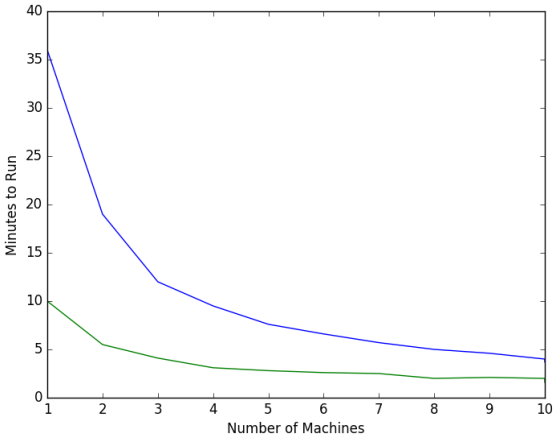


Figure 9: Running the TOAD and BIP100 Benchmarks on varying numbers of Cores and machines, not including the driver. On all runs there was access to the entire 11 node (master and slaves) hadoop cluster which had 3 replicas of each blockchain file. See the raw in data Figure 12.

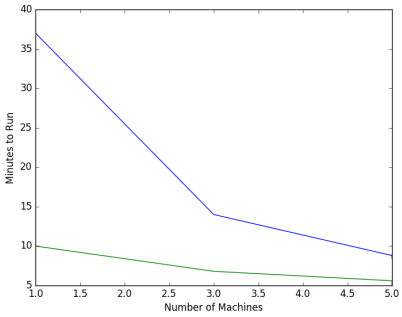


Figure 10: Running the TOAD and BIP100 Benchmarks on varying numbers of Cores and machines, not including the driver. On all runs there was access to the entire 6 node (master and slaves) hadoop cluster which had 1 replica of each blockchain file. Graph is less smooth than Figure 9 as the number of samples points was smaller. See the raw in data Figure 13.

```
def big_tx(bound):
    t = txns(block_objs)\
        .flatMap(lambda tx:
                  [(lambda txo: (tx.tx_id,
                                 txo.index,
                                 txo.value,
                                 txo.pk_script)
                               )(unlazy(lazy_txo))
                   for lazy_txo in tx.tx_outs])
    fields = [StructField(field_name, _type(),False)
                for field_name, _type in
                    [("txid", StringType),
                    ("index", IntegerType),
                    ("value", LongType),
                    ("script", StringType)]]
    schema = StructType(fields)
    txo_table = sb.sql.createDataFrame(t, schema)
    txo_table.registerTempTable("txos")
    big_txs = sb.sql.sql("SELECT * FROM txos where\
                         value > %s"%bound).collect()
```

Figure 11: Using Spark SQL to make a table of all transaction outputs, and then collecting all transactions above a bound

5 Future Work

5.1 Compare Cython to Scala

It would be interesting to compare performance of these two languages.

5.2 Spark SQL

Preliminarily, BTCSpark is integrated & working with Spark SQL. Figure 11 demonstrates building a table with Spark SQL and filtering for large-valued output transactions. Performance wise, these queries run with very reasonable time (Figure 11 ran in 5 minutes on the 10 slave cluster).

5.3 Spark Streaming

Earlier versions of BTCSpark (eagerly evaluating) were compatible with spark streaming. However, there are a couple difficulties in using spark streaming with respect to the way that a Bitcoin node adds new blocks. It should be possible to get this operational with not too much fanfare.

5.4 Query Library

Now that the core infrastructure of BTCSpark is operational, it should be feasible to develop a large library of queries. Over IAP, I may run a small hackathon to get some friends to write queries for it.

5.5 Visualizations

I began experimenting with Bokeh, a python library for visualization. It's a terrific library – visualization is tough to standardize without knowing what kind of data is incoming, but perhaps I can build some common components for types of Bitcoin data.

6 Reflections

In reflection, I have high hopes for BTCSpark to serve as a useful basis for the Bitcoin community. The tool has reasonable performance, and a cluster can quickly be spun up and running in under an hour. Furthermore, as Bitcoin continues to grow, BTCSpark will scale with it – as the results have shown, the system is quite scalable. Hopefully, with more community interest the platform will develop a mature set of libraries and queries.

With respect to building a distributed system, I am also concurrently taking Parallel and Numerical computing this semester. My project in that class ended up being more about a building a fault tolerant system from complete scratch, and this project more about analysis using existing tools – so it goes. One thing I noticed is the differences in frustrations: with this project, I found that using a tool like spark got me 90% of the way to a working product, however, as it always is, the last 10% takes 90% of the time. Thus I felt like I made little progress this semester, as lots of time was spent polishing the code, improving performance, and dealing with digging down into Spark to get the cluster launching facilities operating properly for certain vagaries. In contrast, in my other project, I was working from scratch. I made leaps in bounds of progress, implementing Paxos, a distributed file system, and a mapreduce cluster executor very quickly. However, once I hit the 90% mark, I then ran into the same final 10% that takes so much time – various bits of the design were falling apart, performance was not tolerable, and other issues. Irrespective of the above, I'm certainly happy with my progress on both projects this semester, just notes on how different pathways yield different types of results at different rates.

References

[1] `https://bitcoinchain.com`.

[2] `https://blockchain.info`.

[3] `http://btc.blockr.io/`.

[4] https://www.blocktrail.com/BTC.

[5] https://live.blockcypher.com.

[6] http://explorer.chain.com.

[7] Acinq. https://github.com/ACINQ/bitcoin-lib.

[8] Bitcoinj. https://github.com/bitcoinj/bitcoinj.

[9] Blockparser. https://github.com/znort987/blockparser.

[10] btcd. https://github.com/btcsuite/btcd.

[11] Satoshi's coins have not moved. blockchain.info is dead! – reddit. https://www.reddit.com/r/Bitcoin/comments/3frn1d/satoshis_coins_have_not_moved_blockchaininfo_is/.

[12] M. Corallo. http://bitcoinrelaynetwork.org.

[13] M. H. Fergal Reid. An analysis of anonymity in the bitcoin system. http://arxiv.org/abs/1107.4524.

[14] J. Garzik. Peer review of "quantitative analysis of the full bitcoin transaction graph". https://gist.github.com/jgarzik/3901921.

[15] D. Ron and A. Shamir. Quantitative analysis of the full bitcoin transaction graph. Cryptology ePrint Archive, Report 2012/584, 2012. http://eprint.iacr.org/.

A Bitcoin

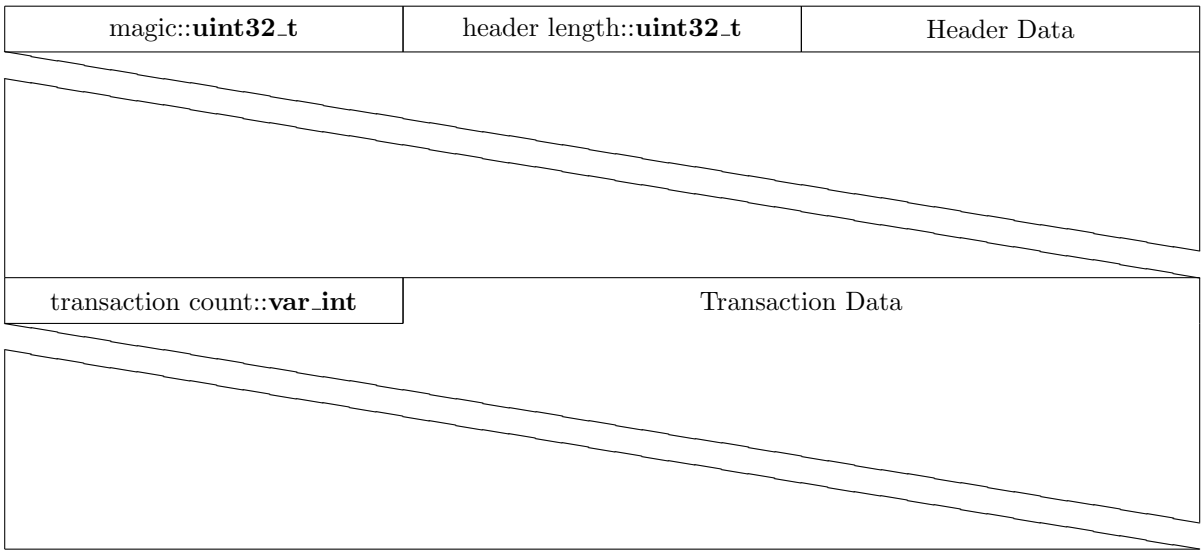
A.1 At a Glance

- Bitcoin is a decentralized byzantine fault tolerant e-cash scheme
- Market Cap: About 5 Billion in Bitcoin itself, Billions in Ecosystem Companies as well
- Data Rate: 1 MegaByte/10 Minutes, but the (some of) community would like to increase this to support more users
- Current Data Size: 400×126 MB Block Files

A.2 The Blockchain: Serialized Network Format

A brief overview of the network format follows for convenience.

A.2.1 Block



A.2.2 Header

version::**uint32_t**	previous block::**uint8_t[32]**	Merkle root::**uint8_t[32]**
time stamp::**uint32_t**	difficulty::**uint32_t**	nonce::**uint32_t**

A.2.3 Transaction Input

spends::OutPoint	transaction index::**uint32_t**	script length::**var_int**
script::**uint8_t**[script length]	sequence ::**uint32_t**	

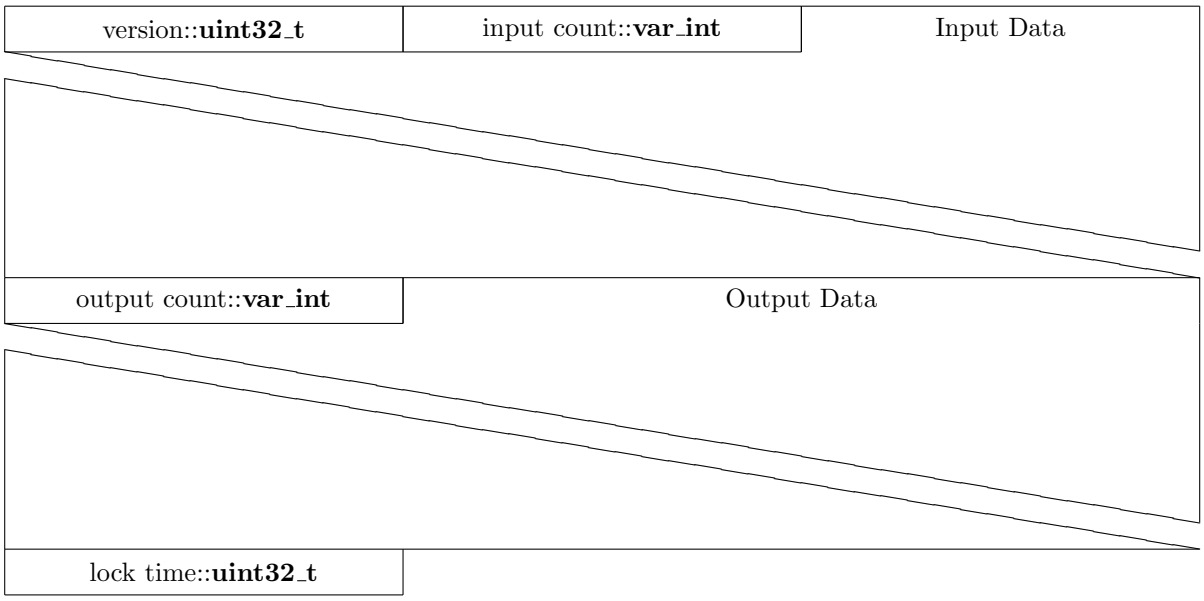
A.2.4 Transaction Outpoint

transaction id::**uint8_t[32]**	output index::**uint32_t**

A.2.5 Transaction Output

value ::**int32_t**	script length::**var_int**	script::**uint8_t**[script length]

A.2.6 Transaction



B Data

Cores	Machines	TOAD	BIP100
1	1	36	10.0
2	2	19	5.5
3	3	12	4.1
4	4	9.5	3.1
5	5	7.6	2.8
6	6	6.6	2.6
7	7	5.7	2.5
8	8	5.0	2.0
9	9	4.6	2.1
10	10	4.0	2.0
12	10	4.0	1.8
14	10	3.9	1.8
16	10	3.9	1.8
18	10	3.7	1.8
20	10	3.7	1.6

Figure 12: Data for Figure 9

Cores	Machines	TOAD	BIP100
10	5	8.4	5.0
8	5	8.8	5.3
6	5	8.9	5.5
5	5	8.8	5.6
3	3	14	6.8
1	1	37	10

Figure 13: Data for Figure 10