A general framework for blockchain analytics

Massimo Bartoletti[1], Andrea Bracciali[2], Stefano Lande[1], and
Livio Pompianu[1]

[1] University of Cagliari, Italy
{bart,lande,livio.pompianu}@unica.it
[2] University of Stirling, UK
abb@cs.stir.ac.uk

Abstract. Modern cryptocurrencies exploit decentralised blockchains to record a public and unalterable history of transactions. Besides transactions, further information is stored for different, and often undisclosed, purposes, making the blockchains a rich and increasingly growing source of valuable information, in part of difficult interpretation. Many data analytics have been developed, mostly based on specifically designed and ad-hoc engineered approaches. We propose a general-purpose framework, seamlessly supporting data analytics on both Bitcoin and Ethereum — currently the two most prominent cryptocurrencies. Such a framework allows us to integrate relevant blockchain data with data from other sources, and to organise them in a database, either SQL or NoSQL. Our framework is released as an open-source Scala library. We illustrate the distinguishing features of our approach on a set of significant use cases, which allow us to empirically compare ours to other competing proposals, and evaluate the impact of the database choice on scalability.

1 Introduction

The last few years have witnessed a steady growth in interest on blockchains, driven by the success of Bitcoin and, more recently, of Ethereum. This has fostered the research on several aspects of blockchain technologies, from their theoretical foundations — both cryptographic [5,9] and economic [17,30] — to their security and privacy [1,6,10,13,20].

Among the research topics emerging from blockchain technologies, one that has received major interest is the analysis of the data stored in blockchains. Indeed, the two main blockchains contain several gigabytes of data (∼130GB for Bitcoin, ∼300GB for Ethereum), that only in part are related to currency transfers. Developing analytics on these data allows us to obtain several insights, as well as economic indicators that help to predict market trends.

Many works on data analytics have been recently published, addressing anonymity issues, e.g. by de-anonymising users [19,20,26,28], clustering transactions [11,31], or evaluating anonymising services [23]. Other analyses have addressed criminal activities, e.g. by studying denial-of-service attacks [2,33], ransomware [15], and various financial frauds [21,24,32]. Many statistics on Bitcoin and Ethereum exist, measuring e.g. economic indicators [16,29], transaction fees [22], the usage of metadata [3], *etc.*

Available from ACM at https://doi.org/10.1145/3152824.3152831

Analysis goal	Gathered data	Sources
Anonymity	Transactions graph	bitcoind [19,20,23,28,31], forum.bitcoin.org [28]
	OP_RETURN metadata	bitcoind [23]
	IP addresses	bitcoin faucet [28], blockchain.info [23]
	address tags	blockchain.info [19,20,31], bitcointalk.org [19,20,31]
	address tags	bitcoin-otc.com [31], bitfunder.org [31]
Market analytics	Transactions graph	bitcoind [16], blockexplorer.com [29]
	IP addresses	blockchain.info, ipinfo.io [16]
	address tags	blockchain.info [16]
	trade data	bitcoincharts.com [16]
Cyber-crime	Transactions graph	bitcoind [2,32,33], blockchain.info [15,21], Bitcore [4]
	mempool	bitcoind [2]
	unconfirmed transactions	bitcoind [2]
	no longer online services	archive.org [32,33]
	list of DDoS attacks	bitcointalk.org [33]
	mining pools	blockchain.info, bitcoin wiki [33]
	trades on assets/services	bitcoin wiki [33]
	list of fraudulent services	bitcointalk.org [15,32], badbitcoin.org [32], cryptohyips.com [32]
	address tags	blockchain.info [32]
	exchange rate	bitcoincharts.com [15,32], quandl.com [15]
Metadata	OP_RETURN transactions	bitcoind [3]
	OP_RETURN identifiers	kaiko.com, opreturn.org, bitcoin wiki [3]
Transaction fees	Transactions graph	bitcoind [22]
	exchange rate	coindesk.com [22]
	mining pools	blockchain.info [22]

Table 1. Data gathered by various blockchain analyses.

A common trait of these works is that they create *views* of the blockchain which contain all the data needed for the goals of the analysis. In many cases, this requires to combine data *within* the blockchain with data from the *outside*. These data are retrieved from a variety of sources, e.g. blockchain explorers, wikis, discussion forums, and dedicated sites (see Table 1 for a brief survey). Despite such studies share several common operations, e.g., scanning all the blocks and the transactions in the blockchain, converting the value of a transaction from bitcoins to *USD*, *etc.*, researchers so far tended to implement ad-hoc tools for their analyses, rather than reusing standard libraries. Further, most of the few available tools have limitations, e.g. they feature a fixed set of analytics, or they do not allow to combine blockchain data with external data, or they are not amenable to be updated. The consequence is that the same functionalities have been implemented again and again as new analytics have been developed, as witnessed by Table 1.

In this context, we believe that the introduction of an efficient, modular and general-purpose abstraction layer to manage internal and external information is key for blockchain data analytics, along the lines of the software engineering best practices of *reuse*.

Contributions. The main contribution of this paper is a framework to create general-purpose analytics on the blockchains of Bitcoin and Ethereum. The design of our tool is based on an exhaustive survey of the literature on the analysis of blockchains. The results of our survey, summarized in Table 1, highlight the need to process external data besides those already present on the blockchain. To this purpose, the workflow supported by our tool consists of two steps: (i) we construct a *view* of the blockchain, also containing the needed external data, and we save it in a database; (ii) we analyse the view by using the query language of the DBMS. The first step is supported by a new Scala library. Distinguishably, we allow views to be organised either as a MySQL database, or a MongoDB collection. Our library supports the most commonly used external data, e.g. exchange rates, address tags, protocol identifiers, and can be easily extended by linking the relevant data sources. We evaluate the effectiveness of our framework by means of a set of paradigmatic use cases, which we distribute, together with the source code of our library, under an open source license[1]. We exploit our use cases to evaluate the performance of SQL *vs.* NoSQL databases for storing and querying blockchain views. As a byproduct of our study, we provide a qualitative comparison of the other tools for general-purpose blockchain analytics.

Structure of the paper. In Section 3 we will illustrate our framework through a series of use cases. We will perform experiments (using consumer hardware) which analyse blockchain metadata, exchange rates transactions fees, and address tags. In Section 4 we will discuss some implementation details of our framework, and we will evaluate its effectiveness, and the choice between SQL or NoSQL. In Section 5 we will compare the existing general-purpose blockchain parsers with ours, and finally in Section 6 we will draw some conclusions.

2 Background on Bitcoin

Bitcoin is a decentralized cryptocurrency [25,5], that has recently reached a market capitalization of 100 *USD* billions[2]. Bitcoin can be seen as a huge ledger of *transactions*, which represent transfers of bitcoins (*BTC*). This ledger — usually called *blockchain* — is maintained by a peer-to-peer network of nodes, and a consensus protocol ensures that it can only be updated consistently (e.g., one cannot tamper with or remove an already-published transaction).

To give the intuition on how Bitcoin works, we consider two transactions t_0 and t_1, which we graphically represent as follows:

t_0
in: $\cdots$
in-script: $\cdots$
out-script(x): F_0
value: v_0

t_1
in: $hash(\mathrm{t}_0)$
in-script: σ_1
out-script($\cdots$): $\cdots$
value: v_1

The transaction t_0 contains v_0 bitcoins, which can be *redeemed* by putting on the blockchain a transaction (e.g., t_1), whose in field is the cryptographic hash of the whole t_0. To redeem t_0, the in-script of t_1 must contain a value σ_1 which makes the out-script of t_0 evaluate to true. In its general form, the out-script is a program in a (not Turing-complete) scripting language, featuring a limited set

[1] https://github.com/bitbart/blockchain-analytics-api

[2] Source: crypto-currency market capitalizations http://coinmarketcap.com

of logic, arithmetic, and cryptographic operators. Typically, the out-script is just a signature verification.

Now, assume that the blockchain contains t_0, not yet redeemed, when someone tries to append t_1. To validate this operation, the nodes of the Bitcoin network check that $v_1 \leq v_0$, and then they evaluate the out-script F_0, by instantiating its formal parameter x to the value σ_1. If, after the substitution, F_0 evaluates to true, then t_1 redeems t_0, meaning that the value of t_0 is transferred to the new transaction t_1, and t_0 becomes no longer redeemable. A new transaction can now redeem t_1 by satisfying its out-script.

Bitcoin transactions may be more general than the ones illustrated by the previous example, in that there can be multiple inputs and outputs. Each output has an associated out-script and value, and can be redeemed independently from others. Consequently, in fields must specify which output they are redeeming. Similarly, a transaction with multiple inputs associates an in-script to each of them. To be valid, the sum of the values of all the inputs must be greater or equal to the sum of the values of all outputs.

The Bitcoin network is populated by a large set nodes, called *miners*, which collect transactions from clients, and are in charge of appending the valid ones to the blockchain. To this purpose, each miner keeps a local copy of the blockchain, and a set of unconfirmed transactions received by clients, which it groups into *blocks*. The goal of miners is to add these blocks to the blockchain, in order to get a revenue. Appending a new block B_i to the blockchain requires miners to solve a cryptographic puzzle, which involves the hash $h(B_{i-1})$ of block B_{i-1}, a sequence of unconfirmed transactions $\langle T_i \rangle_i$, and some salt R. The goal of miners is to win the "lottery" for publishing the next block, i.e. to solve the cryptopuzzle before the others; when this happens, the miner receives a reward in newly generated bitcoins, and a *fee* for each transaction included in the mined block (the fee of a transaction is the difference between the values of its inputs and outputs). If a miner claims the solution of the current cryptopuzzle, the others discard their attempts, update their local copies of the blockchain with the new block B_i, and start mining a new block on top of B_i. In addition, miners are asked to verify the validity of the transactions in B_i by executing the associated scripts.

3 Creating blockchain analytics

We illustrate our framework through some case studies, which, for uniformity, have been developed for the Bitcoin case. We refer to our github repository[1] for some Ethereum examples. Our library APIs provide the following Scala classes to represent the primitive entities of the blockchain:

- `BlockchainLib`: main library class. It provides the `getBlockchain` method, to iterate over `Block` objects.
- `Block`: contains a list of transactions, and some block-related attributes (e.g., block hash and creation time).
- `Transaction`: contains various related attributes (e.g., transaction hash and size).

```
1  object MyBlockchain {
2      def main(args: Array[String]): Unit = {
3  
4      val blockchain = BlockchainLib.getBitcoinBlockchain(
5          new BitcoinSettings("user", "password", "8332", MainNet))
6      val mongo = new DatabaseSettings("myDatabase", MongoDB, "user", "password
           ")
7      val myBlockchain = new Collection("myBlockchain", mongo)
8  
9      blockchain.end(473100).foreach(block => {
10         block.bitcoinTxs.foreach(tx => {
11             myBlockchain.append(List(
12                 ("txHash", tx.hash),
13                 ("blockHash", block.hash),
14                 ("date", block.date),
15                 ("inputs", tx.inputs),
16                 ("outputs", tx.outputs)
17             ))
18         })
19     })
20     }
21 }
```

Fig. 1. A basic view of the blockchain.

The library constructs the above-mentioned Scala objects by scanning a local copy of the blockchain. It uses the client, either Bitcoin Core or Parity, to have a direct access to the blocks, exploiting the provided indices. For Bitcoin, it uses the BitcoinJ library as a basis to represent the various kinds of objects, while for Ethereum it uses suitable Scala representations. The APIs allow constructed objects to be exported as MongoDB documents or MySQL records. In MongoDB (a widespread non-relational DBMS) a database is a set of *collections*, each of them containing *documents*. Documents are lists of pairs `(k,v)`, where `k` is a string (called *field name*), and `v` is either a value or a MongoDB document. Conversely, MySQL implements the relational model, and represents an objects as a record in a table. In Sections 3.1 to 3.5 we develop a series of analytics on Bitcoin. Full Scala code which builds the needed blockchain views, queries, and analysis results can be found in the GitHub repository of the project[1].

3.1 A basic view of the Bitcoin blockchain

Since all the analyses shown in Table 1 explore the transaction graph (e.g. they investigate output values, timestamps, metadata, *etc.*), our first case study focusses on a basic view of the Bitcoin blockchain containing no external data. The documents in the resulting collection represent transactions, and they include: (i) the transaction hash; (ii) the hash of the enclosing block; (iii) the date in which the block was appended to the blockchain; (iv) the list of transaction inputs and outputs.

We show in Figure 1 how to use our APIs to construct this collection. Lines 1-2 are standard Scala instructions to define the `main` function. The object `blockchain` constructed at line 4 is a handle to the Bitcoin blockchain. At line 5 we setup the connection to Bitcoin Core, by providing the needed parameters

```
db.myBlockchain.aggregate([
    { $group : {
        _id: {
            year : { $year : "$date" },
            month : { $month : "$date" },
            day : { $dayOfMonth : "$date" },
        },
        avgIn: { $avg: {$size : "$inputs"} },
        avgOut: { $avg: {$size : "$outputs"} }
    }},
    { $sort : { _id : 1}}
]);
```

Fig. 2. A query to estimate the average number of inputs and outputs by date.

(user, password, and port), and by indicating that we want to use the main network (alternatively, the parameter `TestNet` allows to use the test network). At line 6 we setup the connection to MongoDB (alternatively, the parameter `MySQL` allows to use MySQL). Since lines 1-6 are similar for all our case studies, for the sake of brevity we will omit them in the subsequent listings. We declare the target collection `myBlockchain` at line 7. At this point, we start navigating the blockchain (from the origin block up to block number 473100) to populate the collection. To do that we iterate over the blocks (line 9) (note that `b => {...}` is an anonymous function, where `b` is a parameter, and `{...}` is its body), and for each block we iterate over its transactions (at line 10). For each transaction we append a new document to `myBlockchain` (lines 11-16). This document is a set of fields of the form `(k,v)`, where `k` is the field name, and `v` is the associated value. For instance, at line 12 we stipulate that the field `txHash` will contain the hash of the transaction, represented by `tx.hash`. This value is obtained by the API `BitcoinTransaction`.

Running this piece of code results in a view, which we can process to obtain several standard statistics, like e.g. the number of daily transactions, their average value, the largest recent transactions, *etc.*[3] Hereafter we consider another kind of analysis, i.e. the evolution over the years of the number of transaction inputs and outputs. To this purpose, we run the MongoDB query shown in Figure 2. The query first groups the documents with the same date. Then, for each group, it computes the average number of inputs and outputs. Finally, the results are sorted in ascending order. The results of the query are graphically rendered in Figure 3, which shows the average number of inputs/outputs by date. We see that, after an initial phase, the average number of inputs and outputs has stabilised between 2 and 3. This is mainly due to the fact that most transactions are published through standard wallets, which try to minimise the number of inputs; a typical transaction has two outputs, one to perform the payment and the other for the change. We also observe a few peaks in the number of inputs and outputs, which are probably related to experimentation of new services, like e.g. CoinJoin.

[3] Note that one could also perform these queries during the construction of the view. However, this would not be convenient in general, since — as we will see also in the following case studies — many relevant queries can be performed on the same view.

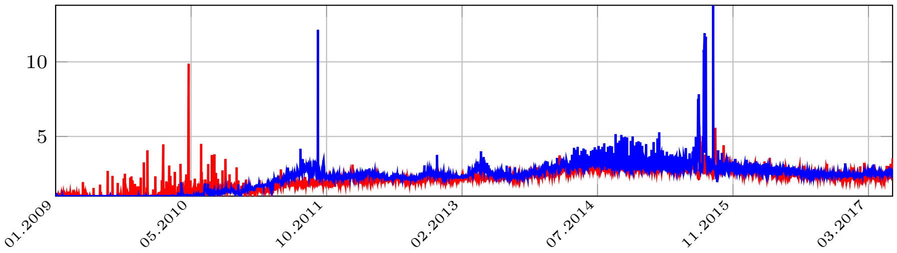

Fig. 3. Average number of inputs (red line) and outputs (blue line) by date.

```
1  val opReturnOutputs = new Collection("opReturn", mongo)
2  
3  blockchain.start(290000).end(473100).foreach(block => {
4      block.bitcoinTxs.foreach(tx => {
5          tx.outputs.foreach(out => {
6          if(out.isOpreturn()) {
7            opReturnOutputs.append(List(
8            ("txHash", tx.hash),
9            ("date", block.date),
10           ("protocol", OpReturn.getApplication(out.outScript.toString)),
11           ("metadata", out.getMetadata())
12           ))
13         }
14         })
15     })
16 })
```

Fig. 4. Exposing OP_RETURN metadata.

3.2 Analysing OP_RETURN metadata

Besides being used as a cryptocurrency, Bitcoin allows for appending a few bytes of metadata to transaction outputs. This is done preeminently through the OP_RETURN operator. Several protocols exploit this feature to implement blockchain-based applications, like e.g. digital assets and notarization services [3].

We now construct a view of the blockchain which exposes the protocol metadata. More specifically, the entries of the view represent transaction outputs, and are composed of: (i) the hash of the transaction containing the output; (ii) the date in which the transaction has been appended to the blockchain; (iii) the name of the protocol that produced the transaction; (iv) the metadata contained in the OP_RETURN script. Figure 4 shows the Scala code to construct this collection (we omit the declaration of the `main` method, already shown in Figure 1). At line 3 we scan the blockchain, starting from block 290,000 since OP_RETURN transactions were only relayed as standard transactions after the release 0.9.0 of Bitcoin Core. We then iterate through transactions at line 4, and through their outputs at line 5. We append a new document to our collection (lines 7-11) whenever the output of the corresponding transaction is an OP_RETURN (line 6). The method `OpReturn.getApplication` of our APIs takes as input a piece of metadata, and returns the name of the associated protocol. This is inferred by the results of the analysis in [3].

The obtained view can be used to perform various analyses. For instance, we show in Figure 5 the number of transactions associated with the most used protocols (only those with at least 1000 transactions). The protocol with the highest number of transactions is Colu, which is used to certify and transfer the ownership of physical assets. The second most used protocol is Omni, followed by Blockstore, a key-value store upon which other protocols are based.

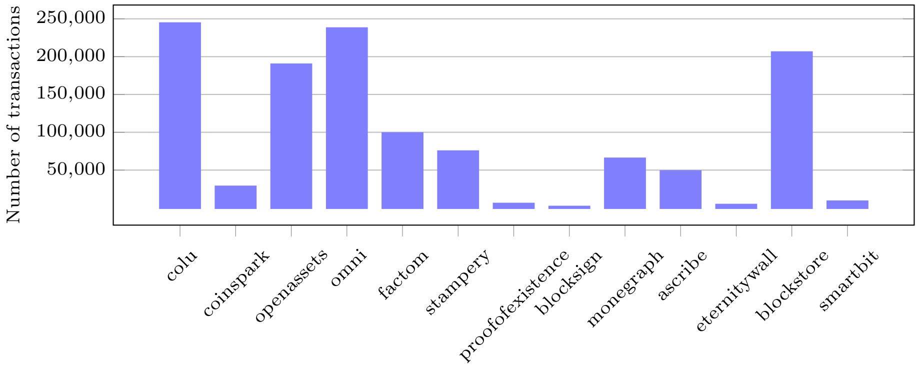

Fig. 5. Number of transactions per protocol (only protocols with > 1000 transactions).

3.3 Exchange rates

Several analyses in Table 1 use exchange rates for quantifying the economic impact of various phenomena (e.g. cyber-crime attacks, transaction fees, business activities). In this section we analyse how the value transferred in transactions is affected by the exchange rate between *USD* and *BTC* over the years. Since exchange rates are not stored in the Bitcoin blockchain, we need to obtain these data from an external source, e.g. the Coindesk APIs. Using these data, we construct a blockchain view where each transaction is associated with the exchange rate at the time it has been appended to the blockchain. More specifically, we construct a MongoDB collection whose documents represent transactions containing: (i) the transaction hash; (ii) the date in which the transaction has been appended to the blockchain; (iii) the sum of its output values (in *BTC*); (iv) the exchange rate between *BTC* and *USD* in such date.

Figure 6 shows the Scala code which builds this collection, using our APIs. At line 1 we declare the collection that we are going to build, `txWithRates`. At lines 3-4 we iterate over all the transactions in the Bitcoin blockchain. For each one, at lines 5-9 we add a new document to `txWithRates`. The total amount of *BTC* sent by the current transaction is stored in the field `outputSum` (line 8). The exchange rate is obtained by invoking the method `Exchange` of our APIs (line 9). This method takes a date and retrieves from Coindesk the exchange rate *BTC*/*USD* in that date.

```
val txWithRates = new Collection("txWithRates", mongo)

blockchain.end(473100).foreach(block => {
    block.bitcoinTxs.foreach(tx => {
        txWithRates.append(List(
            ("txHash", tx.hash),
            ("date", block.date),
            ("outputSum", tx.getOutputsSum()),
            ("rate", Exchange.getRate(block.date))
        ))
    })
})
```

Fig. 6. Exposing exchange rates.

We can analyse the obtained collection in many ways, in order to study how exchange rates are related to the movements of currency in Bitcoin. For instance, one can obtain statistics about the daily transaction volume in *USD*, the market capitalization, the list of richest addresses, *etc.* Hereafter, we measure the average value of outputs (in *BTC*) of the transactions in intervals of exchange rates. The diagram in Figure 7 shows the results of this analysis, where we have split exchange rates in 7 intervals of equal size. In the first five intervals we observe the expected behaviour, i.e. the value of outputs decreases as the exchange rate increases. Perhaps surprisingly, the last two intervals show an increase in the value of outputs when the value *BTC* has surpassed 1500 *USD*. This may be explained by speculative operations on Bitcoin.

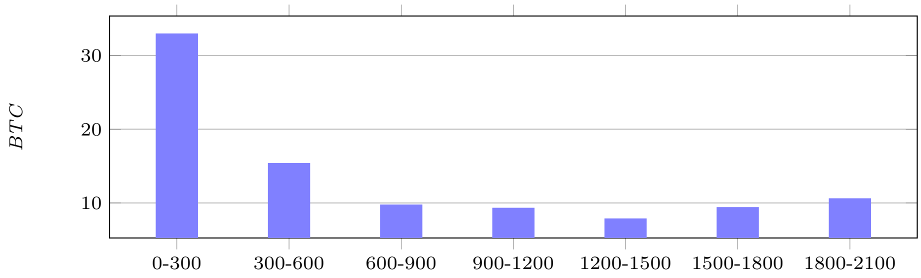

Fig. 7. Average value of outputs (in *BTC*) by exchange rate.

3.4 Transaction fees

In this section we study *transaction fees*, which are earned by miners when they append a new block to the blockchain. Each transaction in the block pays a fee, which in Bitcoin is defined as the difference between its input and output values. While the values of outputs are stored explicitly in the transaction, those of inputs are not: to obtain them, one must retrieve from a past block the transaction that is redeemed by the input. This can be obtained through a "deep" scan of the blockchain, which is featured by our library. We show in Figure 8

```
1  val blockchain = BlockchainLib.getBitcoinBlockchain(new BitcoinSettings("user","
       password","8332",MainNet,true))
2  val mongo = new DatabaseSettings("myDatabase", MongoDB, "user", "password")
3  val txWithFees = new Collection("txWithFees", mongo)
4
5  blockchain.end(473100).foreach(block => {
6      block.bitcoinTxs.foreach(tx => {
7          txWithFees.append(List(
8              ("blockHash", block.hash),
9              ("txHash", tx.hash),
10             ("fee", tx.getInputsSum() - tx.getOutputsSum()),
11             ("date", block.date),
12             ("rate", Exchange.getRate(block.date))
13         ))
14     })
15 })
```

Fig. 8. Exposing transaction fees.

how to construct a collection which contains, for each transaction: (i) the hash of the enclosing block; (ii) the transaction hash; (iii) the fee; (iv) the date in which the transaction was appended to the blockchain; (v) the exchange rate between *BTC* and *USD* in such date.

The extra parameter `true` in the `BitcoinSettings` constructor (missing in the previous example), triggers the "deep" scan. When scanning the blockchain in this way, the library maintains a map which associates transaction outputs to their values, and inspects this map to obtain the value of inputs[4]. The methods `getInputsSum` (resp., `getOutputsSum`) at line 10 returns the sum of the values of the inputs (resp., the outputs) of a transaction.

The obtained collection can be used to perform several standard statistics, e.g. the daily total transaction fees, the average fee, the percentage earned by miners from transaction fees, *etc.* Here we analyse the so-called *whale transactions* [14], which pay a unusually high fee to miners. To obtain the whale transactions, we first compute the average $\bar{x}$ and standard deviation σ of the fees in all transactions: in *USD*, we have $\bar{x} = 0.41$, $\sigma = 12.09$. Then, we define whale transactions as those which pay a fee greater than $\bar{x} + 2\sigma = 24.58$ *USD*. Overall we collect $242,839$ whale transactions; those with biggest fee are displayed in Figure 9.

Fee (USD)	Date	Transaction hash
136243.37	2016-04-26 14:15:22	cc455ae816e6cdafdb58d54e35d4f46d860047458eacf1c7405dc634631c570d
56493.50	2017-01-04 20:01:28	d38bd67153d774a7dab80a055cb52571aa85f6cac8f35f936c4349ca308e6380
39502.15	2017-05-31 14:28:51	cb95ab3aef378c14bc59d0db682d96202b981c1f8fad7d66e23e0be06f2a00c4
25095.71	2017-05-31 14:28:51	8e12a1aba87e4657f5fabec1121ed57f706805ad6d4ffe88c6fce78596bd9b75
23518.00	2013-08-28 10:45:17	4ed20e0768124bc67dc684d57941be1482ccdaa45dadb64be12afba8c8554537

Fig. 9. The five biggest whale transactions.

[4] Since inputs can only redeemed transactions on past blocks, the map always contains the required output. Although coinbase inputs do not have a value in the map, we calculate their value using the total fees of the current block and the block height (reward is halved each 210,000 blocks).

```
1  val mySQL = new DatabaseSettings("outwithtags", MySQL, "user", "password")
2  val tags=new Tag("src/main/scala/tcs/custom/input.txt")
3  val outTable = new Table(sql"""
4      create table if not exists tagsoutputs(
5        id serial not null primary key,
6        transactionHash varchar(256) not null,
7        txdate TIMESTAMP not null,
8        outvalue bigint unsigned,
9        address varchar(256),
10       tag varchar(256)
11   )""", mySQL)
12
13 blockchain.end(473100).foreach(block => {
14   block.bitcoinTxs.foreach(tx => {
15     tx.outputs.foreach(out => {
16       out.getAddress(MainNet) match {
17         case Some(add) =>
18           tags.getValue(add) match {
19             case Some(tag) => {
20               outTable.insert(sql"insert into tagsoutputs (transactionHash,
                     txdate, outvalue, address, tag) values (${tx.hash.toString
                     }, ${block.date}, ${out.value}, ${add.toString}, ${tags.
                     getValue(add)})")}
21             case None => }
22         case None =>
23       }
24     })
25   })
26 })
```

Fig. 10. Associating transaction outputs with tags (SQL version).

3.5 Address tags

The webpage `blockchain.info/tags` hosts a list of associations between Bitcoin addresses and *tags* which briefly describe their usage[5]. Table 1 shows that address tags are widely adopted, e.g. analytics for cyber-crime usually retrieve addresses tagged as scam or ransomware on forums; market analyses exploit tags for recognising addresses of business services; anonymity studies tag the addresses that seem to belong to the same entity. In this section we construct a blockchain view where outputs are associated with the tags of the address which can redeem them (we discard the outputs with untagged addresses). More specifically, we construct an SQL table whose columns represent transaction outputs containing: (i) hash of the enclosing transaction; (ii) the date in which the transaction has been appended to the blockchain; (iii) the output value (in *BTC*); (iv) the address receiving the payment; (v) the tag associated to the address.

Figure 10 shows the Scala script which builds this table. At line 1, we connect to the MySQL database. We retrieve tags from an external source, the `blockchain.info` website. While in the previous case studies we have retrieved external data by querying the source (e.g. the Coindesk APIs), in this case we query a local file in which we have stored the data fetched from `blockchain.info`. At line 2, given the file containing tags, the `Tag` class builds a `Map` which asso-

[5] For instance, address 1PQCrkzWweCw4huVLcDXttAZbSrrLbJ92L is associated to tag *Linux Mint Donations* `http://www.linuxmint.com/donors.php`

ciate each address to the correspondent tag. At lines 4-11 we create a new table. At lines 13-15 we iterate over all the transaction outputs. At line 16 we try to extract the address which can redeem the current output. If we find it (line 17), then we search the map for the associated tag (line 18); if a tag is found (line 19) we insert a new row into the `tagsoutputs` table (line 20).

Using the obtained view, one can aggregate transactions on different business levels [16] to obtain statistics about the total number of transactions, the amount of *BTC* exchanged, the geographical distributions of tagged service, *etc.* In particular, we aggregate all addresses whose tag starts with *SatoshiDICE*, and then we measure the number of daily transactions which send *BTC* to one of these addresses. The diagram in Figure 11 shows the results of this analysis. The fall in the number of transactions at the start of 2015 may be due to the fact that SatoshiDICE is using untagged addresses.

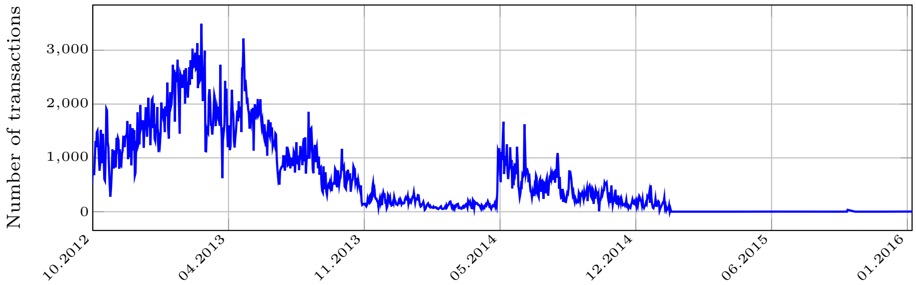

Fig. 11. Number of daily transactions to addresses tagged with *SatoshiDICE**.

4 Implementation and validation

We implement the Ethereum-side of our library by exploiting Parity, queried by means of the web3j library. Bitcoin data is provided by both BitcoinJ and the RPC interface of Bitcoin Core. While BitcoinJ APIs only allow the programmer to retrieve a block by its hash, Bitcoin Core's interface exposes calls to do so by its height on the chain. Furthermore, BitcoinJ's "block objects" do not carry information about block height and the hash of the next block (they only have backward pointers, as defined in the blockchain), which can be fetched by using Bitcoin Core. Our APIs allow to navigate blockchains. Particularly, in the Bitcoin case, we do this by iterating over these steps: (i) get the hash h of the block of height i, by using Bitcoin Core; (ii) get the block with hash h, by using BitcoinJ; (iii) increment i. By default, the loop starts from 0 and stops at the last block. The methods `blockchain.start(i)`, and `blockchain.end(j)` allow to scan an interval of blockchains, as shown in Section 3.2. We write the SQL queries exploiting ScalikeJDBC, a SQL-based DB access library for Scala. ScalikeJDBC provides also a DSL for writing SQL queries.

Case study	MongoDB			MySQL		
	Create	Query	Size	Create	Query	Size
Basic view	9 h	2860 s	300 GB	9 h	3.5 h	266 GB
OP_RETURN metadata	2 h	0.5 s	0.5 GB	1.4 h	2.5 s	0.5 GB
Exchange rates	5 h	477 s	34 GB	4.5 h	243 s	27 GB
Transaction fees	9 h	448 s	51 GB	8.5 h	614 s	43.5 GB
Address tags	4 h	1.8 s	0.8 GB	2.3 h	2.7 s	0.6 GB

Table 2. Performance evaluation of our framework.

We carry out our experiments using consumer hardware, i.e. a PC with a quad-core Intel Core i5-4440 CPU @ 3.10GHz, equipped with 32GB of RAM and 2TB of hard disk storage. All the experiments scan the Bitcoin blockchain from the origin block up to block number 473100 (added on 2017/06/27). Table 2 displays a comparison of the size of each view, and the time required to create and query it.

Note that the size of the blockchain view constructed in `Basic` (Section 3.1) is more than twice than the current Bitcoin blockchain. This is because, while Bitcoin stores scripts in binary format, our library writes them as strings, so to allow for constructing indices and performing queries on scripts. Moreover, the SQL query in `Basic` is particularly slow because of the join operations it performs. Note instead that the query times in SQL and MongoDB are quite similar in all the other cases, where no join operation is required.

5 Comparison with related tools

We now compare other general-purpose blockchain analysis tools with ours. Table 3 summarises the comparison, focussing on the target blockchain, the DBMS used, the support for creating a custom schema, and for embedding external data. The rightmost column indicates the date of the most recent commit in the repository. Note that all the tools which support Bitcoin also work on Bitcoin-based *altcoins*.

The projects `blockparser` and `rusty-blockparser` allow one to perform full scans of the blockchain, and to define custom listeners which are called each time a new block or transaction is read. Unlike our library, these tools offer limited built-in support for combining blockchain and external data. The website `blockchainsql.io` has a GUI through which one can write and execute SQL queries on the Bitcoin blockchain. This is the only tool, among those mentioned in Table 3, that does not need to store a local copy of the blockchain. A drawback is that the database schema is fixed, hence it is not possible to use it for analytics which require external data. While the other tools store results on secondary memory, `blockparser` and `BlockSci` keep all the data in RAM. Although this speeds up the execution, it demands for "big memory servers", since the size of the blockchains of both Bitcoin and Ethereum has largely surpassed the amount of RAM available on consumer hardware. Note instead that the disk-based tools also work on consumer hardware. Some low-level optimizations, combined with

Tool	Blockchain	Database	Schema	Ext. data	Updated
blockparser	BTC	RAM-only	Custom	Custom	2015-12
rusty-blockparser	BTC	SQL, CSV	Fixed	Custom	2017-09
blockchainsql.io	BTC	SQL	Fixed	None	N/A
BlockSci	BTC	RAM-only	Custom	Custom	2017-09
python-parser	BTC	None	None	Custom	2017-05
Our framework	**BTC, ETH**	**MySQL, MongoDB**	**Custom**	**Custom**	**2017-09**

Table 3. General-purpose blockchain analytics frameworks.

an in-memory DBMS, help [12] to overwhelm the performance of the disk-based tools. Unlike the other tools, [12] provides also data about transactions broadcast on the peer-to-peer network.

Remarkably, as far as we know none of the analyses mentioned in Table 1 uses the general-purpose tools in Table 3. Instead, several of them acquire blockchain raw data by using Bitcoin Core[6] (the reference Bitcoin client), and encapsulate them into Java objects with the BitcoinJ APIs before processing. However, neither Bitcoin Core nor BitcoinJ are natural tools to analyse the blockchain: the intended use of BitcoinJ is to support the development of wallets, and so it only gives direct access to blocks and transactions from their *hash*, but it does not allow to perform forward scans of the blockchain. On the other hand, Bitcoin Core would provide the means to scan the blockchain, but this requires expertise on its low-level RPC interface, and even doing so would result in raw pieces of JSON data, without any abstraction layer.

A precise comparison of the performance of these tools against ours is beyond the goals of this paper. The performance analysis in Table 2 is a first step towards the definition of a suite of benchmarks for evaluating blockchain parsers.

6 Conclusions and future work

We have presented a framework for developing general-purpose analytics on the blockchains of Bitcoin and Ethereum. Its main component is a Scala library which can be used to construct views of the blockchain, possibly integrating blockchain data with data retrieved from external sources. Blockchain views can be stored as SQL or NoSQL databases, and can be analysed by using their query languages. Our experiments confirmed the effectiveness and generality of our approach, which uniformly comprises in a single framework several use cases addressed by various ad-hoc approaches in literature. Indeed, the expressiveness of our framework overcomes that of the closer proposals in the built-in support for external data, and the support of different kinds of databases and blockchains. Importantly, coming in the form of an open source library for a mainstream language, our framework is amenable of being validated and extended by a community effort, following reuse best practices.

[6] https://bitcoin.org/en/bitcoin-core. Another popular tool for accessing the blockchain was Bitcointools (https://github.com/gavinandresen/bitcointools), but it seems no longer available.

On the comparison of SQL vs NoSQL, our experiments did not highlight significant differences in the complexity of writing and executing queries in the two languages. Instead, we observed that the schema-less nature of NoSQL databases simplifies the Scala scripts. From Table 2 we see that both creation and query time are comparable as order of magnitude. As already discussed in Section 4, the difference in the execution time of queries is due to join operations in SQL. A more accurate analysis, carried over a larger benchmark, is scope for future work. Anyway, it is worth recalling that the goal of our proposal is provide to the final user the flexibility to choose the preferred database, rather than ascertain an idea of best-fit-for-all in the choice.

Although our framework is general enough to cover most of the analyses in Table 1, it has some limitations that can be overcome with future extensions. In particular, some analyses addressing e.g. information propagation, forks and attacks [7,8,18,27] require to gather data from the underlying peer-to-peer network. To support this kind of analyses one has to run a customized node (either of Bitcoin or Ethereum). Such an extension would also be helpful to obtain on-the-fly updates of the analyses.

Acknowledgments. This work is partially supported by Aut. Reg. of Sardinia P.I.A. 2013 "NOMAD". This paper is based upon work from COST Action IC1406 cHiPSET, supported by COST (European Cooperation in Science and Technology).

References

1. Androulaki, E., Karame, G., Roeschlin, M., Scherer, T., Capkun, S.: Evaluating user privacy in Bitcoin. In: Financial Cryptography and Data Security. LNCS, vol. 7859, pp. 34–51. Springer (2013)
2. Baqer, K., Huang, D.Y., McCoy, D., Weaver, N.: Stressing out: Bitcoin "stress testing". In: Financial Cryptography Workshops. LNCS, vol. 9604, pp. 3–18. Springer (2016)
3. Bartoletti, M., Pompianu, L.: An analysis of Bitcoin OP_RETURN metadata. In: Financial Cryptography Workshops. LNCS, vol. 10323. Springer (2017)
4. Bistarelli, S., Santini, F.: Go with the -Bitcoin- flow, with visual analytics. In: ARES. pp. 38:1–38:6 (2017)
5. Bonneau, J., Miller, A., Clark, J., Narayanan, A., Kroll, J.A., Felten, E.W.: SoK: Research perspectives and challenges for Bitcoin and cryptocurrencies. In: IEEE S & P. pp. 104–121 (2015)
6. Bonneau, J., Narayanan, A., Miller, A., Clark, J., Kroll, J.A., Felten, E.W.: Mixcoin: Anonymity for Bitcoin with accountable mixes. In: Financial Cryptography and Data Security. LNCS, vol. 8437, pp. 486–504. Springer (2014)
7. Decker, C., Wattenhofer, R.: Information propagation in the Bitcoin network. In: P2P. pp. 1–10. IEEE (2013)
8. Donet, J.A.D., Pérez-Solà, C., Herrera-Joancomartí, J.: The Bitcoin P2P network. In: Financial Cryptography Workshops. LNCS, vol. 8438, pp. 87–102. Springer (2014)
9. Garay, J.A., Kiayias, A., Leonardos, N.: The Bitcoin backbone protocol: Analysis and applications. In: EUROCRYPT. LNCS, vol. 9057, pp. 281–310. Springer (2015)

10. Gervais, A., Karame, G.O., Wüst, K., Glykantzis, V., Ritzdorf, H., Capkun, S.: On the security and performance of proof of work blockchains. In: ACM SIGSAC Conference on Computer and Communications Security. pp. 3–16. ACM (2016)
11. Harrigan, M., Fretter, C.: The unreasonable effectiveness of address clustering. In: UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld. pp. 368–373. IEEE (2016)
12. Kalodner, H., Goldfeder, S., Chator, A., Möser, M., Narayanan, A.: Blocksci: Design and applications of a blockchain analysis platform. arXiv preprint arXiv:1709.02489 (2017)
13. Karame, G.O., Androulaki, E., Roeschlin, M., Gervais, A., Capkun, S.: Misbehavior in Bitcoin: A study of double-spending and accountability. ACM Trans. Inf. Syst. Secur. 18(1), 2 (2015), http://doi.acm.org/10.1145/2732196
14. Liao, K., Katz, J.: Incentivizing blockchain forks via whale transactions. In: Financial Cryptography Workshops. LNCS, vol. 10323. Springer (2017)
15. Liao, K., Zhao, Z., Doupé, A., Ahn, G.: Behind closed doors: measurement and analysis of CryptoLocker ransoms in Bitcoin. In: APWG Symp. on Electronic Crime Research (eCrime). pp. 1–13. IEEE (2016)
16. Lischke, M., Fabian, B.: Analyzing the Bitcoin network: The first four years. Future Internet 8(1), 7 (2016)
17. Luu, L., Saha, R., Parameshwaran, I., Saxena, P., Hobor, A.: On power splitting games in distributed computation: The case of Bitcoin pooled mining. In: IEEE Computer Security Foundations Symposium. pp. 397–411. IEEE (2015)
18. McCorry, P., Shahandashti, S.F., Hao, F.: Refund attacks on Bitcoin's payment protocol. In: Financial Cryptography and Data Security. LNCS, vol. 9603, pp. 581–599. Springer (2016)
19. Meiklejohn, S., Pomarole, M., Jordan, G., Levchenko, K., McCoy, D., Voelker, G.M., Savage, S.: A fistful of bitcoins: characterizing payments among men with no names. In: Internet Measurement Conference. pp. 127–140. ACM (2013)
20. Meiklejohn, S., Pomarole, M., Jordan, G., Levchenko, K., McCoy, D., Voelker, G.M., Savage, S.: A fistful of Bitcoins: characterizing payments among men with no names. Commun. ACM 59(4), 86–93 (2016)
21. Möser, M., Böhme, R., Breuker, D.: An inquiry into money laundering tools in the Bitcoin ecosystem. In: APWG Symp. on Electronic Crime Research (eCrime). pp. 1–14. IEEE (2013)
22. Möser, M., Böhme, R.: Trends, tips, tolls: A longitudinal study of Bitcoin transaction fees. In: Financial Cryptography Workshops. LNCS, vol. 8976, pp. 19–33. Springer (2015)
23. Möser, M., Böhme, R.: Anonymous alone? measuring bitcoin's second-generation anonymization techniques. In: EuroS&P Workshops. pp. 32–41 (2017)
24. Möser, M., Böhme, R., Breuker, D.: Towards risk scoring of Bitcoin transactions. In: Financial Cryptography Workshops. LNCS, vol. 8438, pp. 16–32. Springer (2014)
25. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf (2008)
26. Ober, M., Katzenbeisser, S., Hamacher, K.: Structure and anonymity of the Bitcoin transaction graph. Future Internet 5(2), 237–250 (2013)
27. Pappalardo, G., di Matteo, T., Caldarelli, G., Aste, T.: Blockchain inefficiency in the Bitcoin peers network. CoRR abs/1704.01414 (2017), http://arxiv.org/abs/1704.01414
28. Reid, F., Harrigan, M.: An analysis of anonymity in the Bitcoin system. In: Security and privacy in social networks, pp. 197–223. Springer (2013)

29. Ron, D., Shamir, A.: Quantitative analysis of the full Bitcoin transaction graph. In: Financial Cryptography and Data Security. LNCS, vol. 7859, pp. 6–24. Springer (2013)
30. Schrijvers, O., Bonneau, J., Boneh, D., Roughgarden, T.: Incentive compatibility of Bitcoin mining pool reward functions. In: Financial Cryptography and Data Security. LNCS, vol. 9603, pp. 477–498. Springer (2016)
31. Spagnuolo, M., Maggi, F., Zanero, S.: Bitiodine: Extracting intelligence from the Bitcoin network. In: Financial Cryptography and Data Security. LNCS, vol. 8437, pp. 457–468. Springer (2014)
32. Vasek, M., Moore, T.: There's no free lunch, even using Bitcoin: Tracking the popularity and profits of virtual currency scams. In: Financial Cryptography and Data Security. LNCS, vol. 8975, pp. 44–61. Springer (2015)
33. Vasek, M., Thornton, M., Moore, T.: Empirical analysis of Denial-of-Service attacks in the Bitcoin ecosystem. In: Financial Cryptography Workshops. LNCS, vol. 8438, pp. 57–71. Springer (2014)