Full Bitcoin Blockchain Data Made Easy

Jules Azad Emery, Matthieu Latapy

Full Bitcoin Blockchain Data Made Easy

Jules Azad Emery and Matthieu Latapy

Sorbonne Université, CNRS, LIP6, F-75005 Paris, France

bitcoin@complexnetworks.fr

Abstract

Despite the fact that it is publicly available, collecting and processing the full bitcoin blockchain data is not trivial. Its mere size, history, and other features indeed raise quite specific challenges, that we address in this paper. The strengths of our approach are the following: it relies on very basic and standard tools, which makes the procedure reliable and easily reproducible; it is a purely lossless procedure ensuring that we catch and preserve all existing data; it provides additional indexing that makes it easy to further process the whole data and select appropriate subsets of it. We present our procedure in details and illustrate its added value on large-scale use cases, like address clustering. We provide an implementation online, as well as the obtained dataset.

Introduction

Bitcoin [25] is the first and most widely used crypto-currency. Introduced in 2009, its market capitalization was above 1 trillion U.S. dollars in february 2021. With around 300,000 transactions daily, it is used worldwide in a variety of situations.

All bitcoin transactions are recorded in a public registry, called the blockchain. It is a sequence of blocks issued every ten minutes, each listing the transactions successfully processed since the previous block. This gives a unique opportunity to study real-world, large-scale financial activity. Therefore, bitcoin is at the core of an intense research activity.

However, accessing and processing bitcoin data raises serious challenges. First, the blockchain is ruled by complex protocols that evolved over time, making it necessary to have more than a basic understanding of how it works. In addition, precisely because the blockchain is public, its users use a wealth of obfuscation techniques to preserve their privacy and make transactions anonymous. Last but not least, the sheer size of the data makes it difficult to collect, and even more difficult to analyze; even trivial questions like counting the number of transactions involving a given user are difficult, at such a scale.

As a consequence, most bitcoin studies rely on partial views of the blockchain, on aggregated data, or on privately owned datasets; there is currently no publicly available and easy-to-use dataset containing all bitcoin information stored in the blockchain since its beginning.

Our goal with this paper is to provide such a dataset. We want to provide *all* existing data, without any assumption on what the final user will need. In addition, we want to pre-process these data to add the information needed to use it efficiently and easily. In order to ensure that our procedures are reliable, efficient, and reproducible, we want to rely uniquely on basic, standard command-line tools.

Our approach consists in a sequence of key steps, that we detail in the following sections of this paper:

I. **Collection.** We first perform the raw data extraction from the bitcoin blockchain itself, by setting up a bitcoin node and querying it for each block.

II. **Indexing.** We add to the raw data an integer index for each basic item, like transactions or addresses, without removing any of the original data.

III. **Distillation.** We extract specific subsets of the data that make it easy, fast, and compact to obtain various higher-level information.

IV. **Application.** In order to illustrate our contribution, we detail an advanced use of our dataset: the analysis of a classical address clustering heuristic.

We discuss **related work** in Section 5, and **perspectives** in Section 6.

All the code presented in this paper and all obtained data are documented and available online [11]. We give approximate execution times of each step, but they strongly depend on available bandwiudth, memory space, computation power, disk speed, implementation language, and other parameters; they should therefore be considered as indicative only.

1 Collection

In order to have a local copy of the blockchain, **we first set up our own bitcoin node**. To this end, we install and run the open-source *bitcoin-core* software [16]. It contacts a set of DNS nodes hard-coded in its source code, from which it obtains a list of running bitcoin nodes. Our node then downloads the blockchain from these nodes, thus obtaining its local copy.

We launched this procedure on March 13, 2021. It downloaded the blockchain available at that date in less than 6 hours, leading to a use of 360 GB of disk space.

In principle, one may then read and decode the local binary files used by the node to store its copy of blockchain data. This would be the fastest solution, but it is complex and prone to errors. Indeed these file formats are poorly documented, they changed over years, and they may change again in a close future. In addition, this approach only collects targeted parts of the data, in general. See for instance [1], and Section 5 for more details.

We therefore adopted a slower but simpler and safer approach. It is based on RPC (*Remote Procedure Call)*, the protocol provided to monitor bitcoin nodes [15]. It implements in particular a primitive that returns the block identifier of the i-th block in the chain, for any given i; and a primitive that returns a JSON object containing all the data available in a block of given identifier. Figure 1 presents a simplified version of such a JSON object, that we will detail below.

Thanks to these primitives, **we ran an RPC-based bitcoin blockchain data collection under the form of a JSON object for each block**. Instead of running both primitives for each block, we took benefit of the fact that each block contains the identifier of the previous block in the chain. We therefore collected the latest block (its number is provided by a query to any bitcoin node) and then iteratively collected the previous block until the beginning of the blockchain is reached.

We launched this data collection until block 674 000, dated March 10th, 2021, which took 58 hours. It produced a 2.1 TB (548 GB once compressed with `gzip`) text file in which each line is the JSON object describing a block. Notice however that obtained blocks are in reverse chronological order (the latest one first), which is not convenient for further analysis. We therefore reversed the initial file using the classical `tac` tool. It needs a non-compressed file as input, but it has the advantage of not storing the whole file into main memory, which is crucial here. It performed the reversing of the 674,001 lines of the 2.1 TB file in 28 hours.

```
{"height": 2, "previousblockhash": "h1",
 "hash" :"h2", "time": "t", "nextblockhash": "h3",
 "tx" : [
  {"txid": "B",
   "vin": [{"txid": "C", "vout": 0},
           {"txid": "D", "vout": 0}],
   "vout": [{"value": 2, "n": 0, "scriptPubKey":
             {"type": "pubkeyhash", "asm": "?",
              "addresses": ["a","c"]}},
            {"value": 5, "n": 1, "scriptPubKey":
             {"type": "pubkey", "asm": "c"}}]
  },
  {"txid": "E",
   "vin": [{"txid": "B", "vout": 1}],
   "vout": [{"value": 3, "n": 0, "scriptPubKey":
             {"type": "pubkey","asm": "f"}},
            {"value": 2, "n": 1, "scriptPubKey":
             {"type": "pubkey","asm": "e"}}]
  }]}
```

Figure 1: **Simplified example of a block description in JSON format.** This is block number `2`, recorded at time `t`, and it contains transactions, `B` and `E`. Transaction `B` has two inputs: the output `0` of transaction `C`, and the output `0` of transaction `D`. It has two outputs: its output `0` goes to addresses a and c; its output `1` goes to address `c`. Transaction `E` has one input: the output `1` of transaction `B`; and it has two outputs: its output `0` to address `f` and its output `1` to address `e`. The `value` fields indicate that transaction `B` sends `2` satoshis to addresses `a` and `c` and `5` satoshis to address `c`, taken from outputs `0` of transactions `C` and `D`; and that transaction `E` sends `3` satoshis to address `f` and `2` to address `e`, taken from output `1` of transaction `B`. See Figure 2 for a graphical representation.

As illustrated in Figure 1, each JSON description of a block has several fields. It begins by fields characterizing the block itself, including its rank in the blockchain (`height` field), its identifier (a hash code) and the one of its previous and next block, as well as a timestamp. It has another crucial field, named `tx`, that gives the list of transactions recorded in this block.

Each transaction in this list is itself described by a JSON object, with main fields `txid`, `vin` and `vout`. The `txid` field is the transaction identifier, that we will represent by a capital letter. Fields `vin` and `vout` are the lists of this transaction inputs and outputs (TIOs for short), respectively.

Each transaction output is described by a rank field `n` and has a `value` field giving the amount, in satoshis (a satoshi is 10^{-8} bitcoin)[1] sent to this output. Addresses may be

[1]Amounts are actually stored as an integer number of satoshis in the blockchain, but returned as a decimal number of bitcoins by JSON RPC calls. We convert them into satoshi units to avoid rounding errors.

associated to outputs, described by a JSON object in field `scriptPubKey`. We will represent addresses with lower case letters here.

Each transaction input is the output of a previous transaction; it is described by the `txid` of this transaction and the rank of its output under concern, given in a field named `vout` too.

Addresses require specific attention, as their format is not uniform. As already said, they are given in the `scriptPubKey` JSON object associated to transaction outputs. This JSON object has a `type` field, and if its value is `nulldata` or `nonstandard`, then this output has no directly available address. This is rare, though; in most case, either this JSON object has field named `addresses` that gives the addresses under concern, or the `type` field has value `pubkey`. In this last case, the address is the first word (with space separator) in another field, named `asm`. In order to make the data easier to parse, we then add an `addresses` field to each `scriptPubKey` JSON object, with the addresses we found in it.

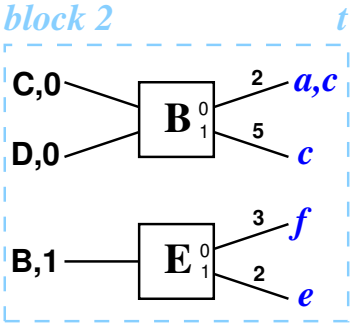

Figure 2: **Graphical representation of a bitcoin block and its transactions.** Their JSON description is in Figure 1. Transactions are represented by square boxes, with their capital letter identifier and output ranks. Addresses are in blue, lower case letters. Amounts appear on output edges of transactions, and the inputs give the corresponding transaction identifier and its output rank.

We graphically represent the key information contained in a block like in Figure 2. In this figure, we represent the block of Figure 1.

Let us insist on the fact that we do not detail *all* available fields above, and we do not represent them all on the picture. There is a wide variety of transaction types and other specific features, that vary over time. All these data are present in our dataset, but they are not our focus here.

With these data, **we already obtain some basic but interesting statistics**. In particular, the dataset contains 674,001 blocks and 623,483,734 transactions with 1,673,052,718 inputs and outputs.

As an illustration, we present in Figure 3 (left) the number of transactions over time, that displays the classical slow start until 2013, followed by a rapid growth that nowadays becomes linear. Figure 3 (right) presents the correlations between transaction number of inputs and outputs. These numbers span orders of magnitude, but large number of inputs are for transactions with only few outputs, and conversely. This indicates specific kinds of transactions, forged for obfuscation, as we will detail in Section 4. Instead, most transactions have only few inputs and outputs, and then the values are not significantly correlated (see the inset).

We also present in Figure 4 (left) the transaction amount distribution, in bitcoins. These amounts span 11 orders of magnitude, which is huge. Notice however that only 1 transaction over 6 has an amount of more than 1 bitcoin. One may guess that large amounts are for old transactions, when bitcoin value was very low, and that more recent transactions have much lower amounts. Figure 4 (right) shows that this is not true: although the average amount significantly decreases, there are still many recent transactions with large amounts, and instead very small amounts tend to disappear. This certainly is a consequence of the generalization of

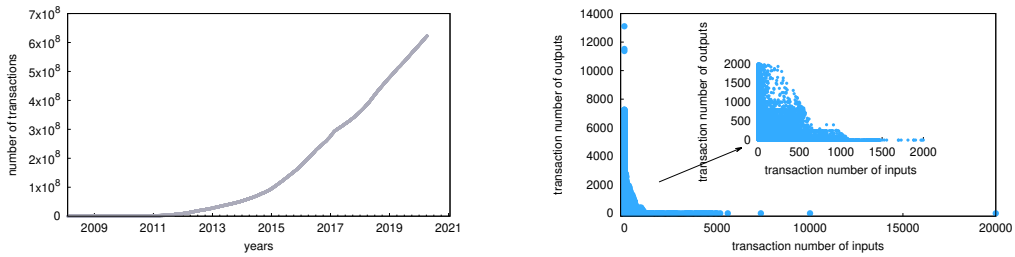

Figure 3: **Left: number of transactions over time**, since the beginning of the bitcoin blockchain. **Right: correlations between transaction number of inputs and outputs.** For each transaction with x inputs and y outputs, we display a dot at coordinates (x, y). The inset provides a zoom on smallest values.

platforms that group user-level transactions into large blockchain transactions for obfuscation and optimization reasons.

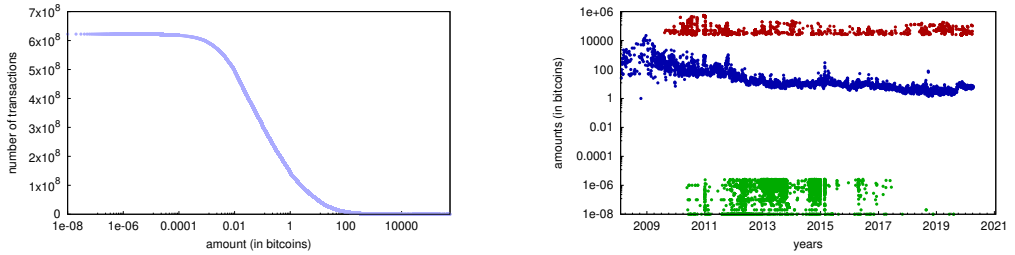

Figure 4: **Left: inverse cumulative distribution of transaction amounts.** For each amount x on the horizontal axis, we display the number of transactions with amount at least x bitcoins (obtained as the sum of their output `value` fields), in log-lin scales. **Right: transaction amounts over time.** For each of the $10,000$ transactions with largest amounts (in red, top dots), and for each of the $10,000$ ones with smallest amounts (in green, bottom dots), we display a dot at its time occurrence (horizontal axis) and amount in bitcoins (vertical axis, in log scale). In between, we plot (in blue, middle dots) the daily average transaction amount.

These plots provide good illustrations of what is readily feasible with the collected dataset. However, other key computations like counting the number of distinct addresses, or the number of occurrences of each address, are more subtle. They require a parsing of the whole data and storing the set of already seen addresses. Likewise, information on transaction inputs, like their associated addresses and satoshi amounts, is not readily available; finding them requires to store and query past transaction outputs. **The next section is devoted to making such operations easy.**

2 Indexing

Handling data at bitcoin blockchain scale requires appropriate indexing and pre-processing. **Indexes from 0 to $n-1$, where n is the number of items of a given kind, are particularly appealing**: one may then store information related to item of index i in an array of size n and access it very efficiently (in both space and time).

For instance, with addresses indexed this way, it is easy to count their occurrences: from an array initially filled with 0s, simply parse the data and increment the i-th cell each time address of index i occurs. The time cost of this counting is dominated by the data parsing, since it has a small $O(1)$ cost per address occurrence. Its space cost is limited to an array of n integers.

Another desirable feature is to have **indexes consistent with occurrence order**: the first occurring item has index 0, the next has index 1, and so on. With such indexings, counting address occurrences for the n' first addresses, for $n' \leq n$, requires an array of n' integers only. Otherwise, an array of size n is needed, like in the base case. More generally, any prefix of the data has indexes from 0 to $n' - 1$, where $n' \leq n$ is the number of items occurring in the prefix. This ensures that prefixes are themselves datasets encoded in the same way. We therefore say that **such indexes are *prefix-consistent***. There is a unique such indexing for a given dataset.

Here, items of interest are blocks, transactions, their inputs and outputs (TIOs), and addresses. As explained above, the `height` field of blocks already is its prefix-consistent index. Transactions have a `txid` field, that gives their native identifier (a 32 byte hash code). Similarly, addresses are alphanumeric character strings of variable length, representing public keys. The case of TIOs is more subtle: a TIO is uniquely identified by the `txid` of the transaction that created it (as an output), together with its rank in the output list of this transaction. Therefore, we consider the pair composed of this `txid` and this rank as the native identifier for this TIO. For instance, the first and second outputs of a transaction X have native identifier $X, 0$ and $X, 1$, respectively. They may later appear as inputs of other transactions.

In order to prepare the collected data for more advanced analysis (Section 4), **our goal here is to associate its prefix-consistent index to each occurrence of transaction, TIO, and address native identifier** in our dataset. Aiming at preserving its integrity, we however keep all the initial data and just add indexes within its JSON objects, in dedicated fields. For instance, we add to each JSON object having a `txid` field a new field, named `index` if this object is a transaction and named `txid_index` if it is a transaction input, with value the prefix-consistent index of this transaction.

The **classical approach** for prefix-consistent indexing is to store index tables in main memory, and then to perform indexing on-the-fly, while parsing the original data. However, this has prohibitive space and time costs.

Indeed, under reasonable assumptions, this approach requires to store at least the native identifiers in main memory. With the numbers of items above this leads to at least 60 GB of memory needs.

In addition, there are more than 27.6 billion item occurrences in total, and so we have to perform this number of searches in the indexes. The only data structure that fits the above space requirements are (sorted) arrays of native identifiers; but then searching has a prohibitive $\Theta(\log(n))$ time cost. Hash tables make index operations very fast, but have space requirements significantly larger than necessary: in addition to native identifiers, they must store indexes, and some spare space. Moreover, these approaches make un-ordered accesses to huge memory spaces, which has a strong impact on speed in practice.

Therefore, **we propose an approach that avoids storing indexes in central memory and still parses the data a very limited number of times.** This approach relies on standard command-line tools, in particular `sort`. It reads the raw data and adds index fields on-the-fly. To achieve this, we will build a file containing on its i-th line the pair `'i-1`

`index'`, meaning that `'index'` is the index of the $(i-1)$-th native identifier occurrence. Then, on-the-fly indexing is easy: it suffices to jointly parse the original data and this file. We detail the successive steps in the following, and illustrate intermediate results in the case of Figure 5.

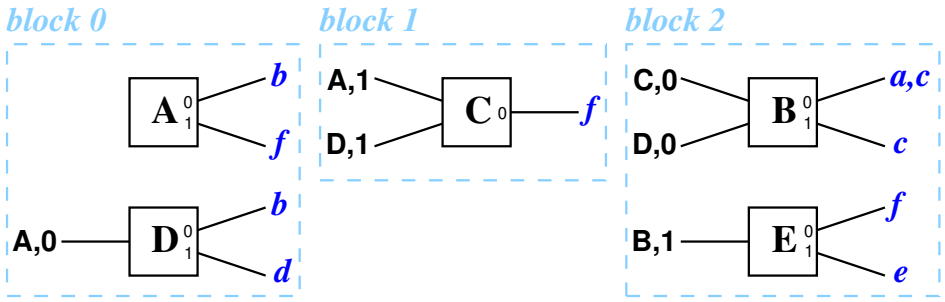

Figure 5: Example data for indexing. We consider three blocks, numbered 0, 1 and 2. Each contains transactions, named A, B, C, D, and E. Their input and output information are displayed as they are available in the dataset. We do not display amounts, timestamps, and other information that plays no role here.

Our first step consists in listing all occurrences of items under concern, together with their occurrence rank. In some cases, their index is readily available: when we parse the i-th transaction, we encounter its `txid` for the first time, and so its index is $i-1$; each TIO is created as a transaction output, therefore counting the number of transaction outputs encountered so far gives current TIO index. In these cases, we output the triplet `'occurrence-rank native-identifier index'`. In other cases, when the index of current item is not directly available, we only output the pair `'occurrence-rank native-identifier'`.

In the case of Figure 5, this leads to the following: `'0 A 0' '1 A,0 0' '2 b' '3 A,1 1' '4 f'` for transaction A, `'5 D 1' '6 A' '7 A,0' '8 D,0 2' '9 b' '10 D,1 3' '11 d'` for transaction D, `'12 C 2' '13 A' '14 A,1' '15 D' '16 D,1' '17 C,0 4' '18 f'` for transaction C, `'19 B 3' '20 C' '21 C,0' '22 D' '23 D,0' '24 B,0 5' '25 a' '26 c' '27 B,1 6' '28 c'` for transaction B, and `'29 E 4' '30 B' '31 B,1' '32 E,0 7' '33 f' '34 E,1 8' '35 e'` for transaction E.

In the obtained list, the native identifier of each item appears exactly once in a triplet with their index, except the ones of addresses. **Our second step** aims at building similar triplets for addresses. To do so, we parse the original data again and list all address occurrences together with their rank. We then sort this list with respect to addresses and we keep only the first tuple for each address. This gives the list of all distinct addresses, each with its first occurrence rank. We sort again with respect to this field, in order to obtain the list of all distinct addresses, ordered by rank. The index of any address is nothing but its rank in this list, and we finally output a triplet `'-1 address index'` for each address.

In the case of Figure 5, we first obtain `'b 0' 'f 1' 'b 2' 'd 3' 'f 4' 'a 5' 'c 6' 'c 7' 'f 8'` and `'e 9'`. We then obtain `'a 5' 'b 0' 'c 6' 'd 3' 'e 9'` and `'f 1'`, then `'b 0' 'f 1' 'd 3' 'a 5' 'c 6'` and `'e 9'`. Finally we obtain `'-1 b 0' '-1 f 1' '-1 d 2' '-1 a 3' '-1 c 4'` and `'-1 e 5'`.

These triplets are in a format similar to the ones from our first step. In our **third step** we put these two sets of tuples together, and sort them according to their second field, which is a native identifier, as well as their first field, numerically. This ensures that all tuples mentioning a given native identifier are grouped together. The first of them necessarily is a

triplet, and its third field is the index of the native identifier under concern (second field).

In our example, we obtain `‘-1 a 3’ ‘25 a’ ‘0 A 0’ ‘6 A’ ‘13 A’ ‘1 A,0 0’ ‘7 A,0’ ‘3 A,1 1’ ‘14 A,1’ ‘-1 b 0’ ‘2 b’ ‘9 b’ ‘19 B 3’ ‘30 B’ ‘24 B,0 5’ ‘27 B,1 6’ ‘31 B,1’ ‘-1 c 4’ ‘26 c’ ‘28 c’ ‘12 C 2’ ‘20 C’ ‘17 C,0 4’ ‘21 C,0’ ‘-1 d 2’ ‘11 d’ ‘5 D 1’ ‘15 D’ ‘22 D’ ‘8 D,0 2’ ‘23 D,0’ ‘10 D,1 3’ ‘16 D,1’ ‘-1 e 5’ ‘35 e’ ‘29 E 4’ ‘32 E,0 7’ ‘34 E,1 8’ ‘-1 f 1’ ‘4 f’ ‘18 f’ ‘33 f’`.

This leads to the **fourth and last step** of our indexing procedure. For each native identifier, we take its index on the first tuple mentioning it, and for each tuple mentioning it (except the ones starting with `‘-1’`), we output a pair `‘i index’`, where i is the first field of the tuple. Such a pair says that the i-th item occurrence in the dataset corresponds to index `‘index’`. We therefore sort these pairs according to their first field, and obtain the wanted file.

In our example, this leads to `‘0 0’ ‘1 0’ ‘2 0’ ‘3 1’ ‘4 1’ ‘5 1’ ‘6 0’ ‘7 0’ ‘8 2’ ‘9 0’ ‘10 3’ ‘11 2’ ‘12 2’ ‘13 0’ ‘14 1’ ‘15 1’ ‘16 3’ ‘17 4’ ‘18 1’ ‘19 3’ ‘20 2’ ‘21 4’ ‘22 1’ ‘23 2’ ‘24 5’ ‘25 3’ ‘26 4’ ‘27 6’ ‘28 4’ ‘29 4’ ‘30 3’ ‘31 6’ ‘32 7’ ‘33 1’ ‘34 8’ ‘35 5’`.

As explained above, **we finally perform the on-the-fly index addition** by parsing the original data and the obtained list of pairs jointly. In the whole procedure, no index is stored in central memory, and we never had to sort the original JSON file. Instead, we sort the list of all item occurrences and the index by taking benefit from the highly optimized `sort` tool, that uses external memory to handle huge tasks and only uses a user-specified amount of main memory. **In our settings, the full indexing took approximately** 98 **hours (close to** 4 **days)**, which is very reasonable given the fact that the initialization of the bitcoin node took 6 hours, and the RPC JSON data collection itself took 58 hours, approximately.

Notice that using an advanced sorting tool as above is not mandatory. If only a routine sorting function is available, then one may proceed as follows. First divide the total number of item occurrences M into parts of size N such that sorting N items fits in central memory. These N occurrences involve $N' \leq N$ distinct native identifiers. The procedure above is able to index them in the whole data. Running this $\frac{M}{N}$ makes the whole indexing, at the cost of repeated parsing of the original data. In this way, the whole procedure may be turned into a standalone program.

3 Distillation

All operations above preserve the original data; they only add address and index fields in order to make it easier to use. However, in most practical situations, one needs specific parts of the data only. Then, systematically resorting to the global dataset is an overkill: the file is huge and it requires JSON parsing which, in addition to a waste in computation time, requires a JSON parsing library.

Instead, users need to easily and quickly run their computations, with controlled space needs, for instance using low-level languages like C. This requires a filtering and pre-processing of the dataset, called *distillation*. **It extracts the needed information and puts it in a convenient format for easy, fast, and compact processing.** We illustrate such distillations in this section, as well as their practical uses.

First notice that, although relevant data depends on the targeted use, many need similar data. In the case of bitcoin, one is typically interested in the **list of transactions with their**

timestamp, input addresses, and output addresses. For this reason, we show how to distillate our dataset into a sequence of one line per transaction, each with the following space-separated fields: `'block timestamp tx nb-in nb-out first-in ... last-in first-out ... last-out'`. Here, `tx` stands for the index of the transaction under concern; `block` is the index of the block that contains this transaction, and `timestamp` is its timestamp; `nb-in` and `nb-out` give for the number of addresses involved in this transaction inputs and outputs, respectively; and the two sequences `first-in ... last-in` and `first-out ... last-out` give the indexes of these `nb-in` and `nb-out` addresses, respectively.

In the case of our guiding example (Figure 5), the lines of distilled data are: `'0 t0 0 0 2 0 1'` `'0 t0 1 1 2 0 0 2'` `'1 t1 2 2 1 1 2 1'` `'2 t2 3 2 2 0 1 3 4'` and `'2 t2 4 1 2 4 1 5'` where `t0`, `t1` and `t2` stand for the timestamps of blocks 0, 1, and 2, respectively.

We perform this distillation as follows. We parse the indexed dataset block by block, and store the current block index and timestamp. We parse transactions in current block in their order within the block, and consider current transaction index. This gives the three first fields of the output line for current transaction. We then parse its inputs and outputs and build the corresponding sets of addresses. Output addresses are directly available within the current transaction JSON object. Instead, input addresses are not readily available. But transaction inputs are always outputs of *former* transactions. We therefore store the addresses of each encountered transaction output, in an array indexed by TIO indexes. When a TIO is encountered as another transaction input, we query this array and obtain the corresponding addresses, which gives us all needed information.

Notice that transaction outputs are used only once as other transaction inputs. Therefore, keeping the addresses of already encountered transaction inputs is unnecessary. We provide an implementation of this procedure with this space optimization (we drop transaction output addresses once used as input) [11].

Our implementation of this procedure performed the whole distillation in 25 hours, leading to a 12 GB file containing the key information listed above (the mere parsing of the JSON file already takes 15 hours). During this process, like in most distillation tasks, **the indexes produced in previous section play a crucial role**. They make it possible to handle data in a very fast and compact way, with arrays queried by item indexes.

The distilled dataset is easy to parse, even in a low-level language like C or with shell scripts. Indexes make it easy, fast and compact to perform many operations. For instance, we obtain that 3.5 % of all addresses are used only once, 87.6 % only twice, and almost 1.2 % are used as input and output of a same transaction (often more than once). This reflects two well-known but rarely quantified bitcoin facts: many users collect transaction change on an address they already use; and many users avoid re-using addresses, for privacy concerns.

We also obtain that the most frequent address in transaction inputs and outputs appear there $3,324,680$ and $3,525,298$ times, respectively. More generally, we display in Figure 6 (left) the distribution of the total number of occurrences of each address, as well as their numbers of occurrences as transaction inputs and outputs. All these distributions are very similar, and span more than 6 orders of magnitude, showing a huge heterogeneity between addresses.

We display in Figure 6 (right) the timeline of transactions involving the two most frequent addresses in our dataset. The first one appears as soon as 2011, and it is intensively used for one year. Its uses then severely slows down, and becomes sporadic after 2015, but, surprisingly enough, it is still used by the end of 2018. Its number of occurrences as input and output are so close to each other that we cannot distinguish them in the plot. The behaviour of the

other address is quite different. It appears much later, just before 2017, but it is used only a few times until 2018. Then, its use is intensive for a few months. It slows down in 2019, and suddenly stops being used in mid-2020.

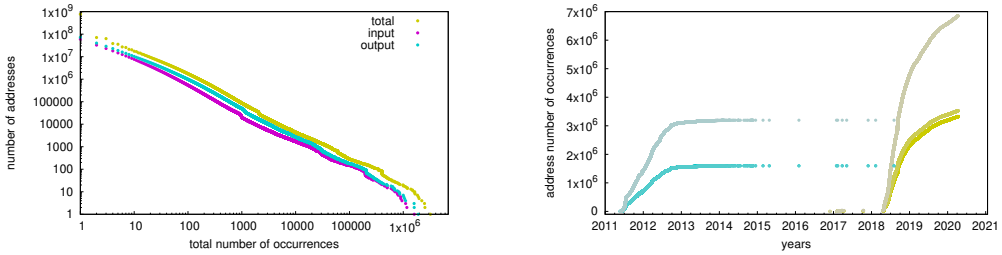

Figure 6: **Left: address reuse**, displayed as the inverse cumulative distribution of the number of occurrences of each address in the dataset, in total, as an input address, or as an output address. **Right: occurrences of the two most used addresses over time.** For the most used address in the dataset (rightmost plots, green colors) and the second most used one (leftmost plots, blue colors), we display its number of occurrences as input, output, and its total number of occurrences since the beginning of the blockchain.

Many other statistics are easy to obtain from various distilled versions of our dataset.

For instance, replacing addresses by amounts, gives transaction fees (the difference between input and output amounts), displayed in Figure 7 (left). Since the beginning of the blockchain, fees range from 0 to more than 100 bitcoins, but they are nowadays much more uniform: they range from 10^{-6} to 0.02 bitcoins for the last million of transactions, with the vast majority very close to 0.0002 bitcoins.

Replacing addresses by TIOs gives information on bitcoin flows, like the delay between the receiving and spending of bitcoins, displayed in Figure 7 (right). Many bitcoins are spent at a fast pace (just a few blocks), and the delay distribution is very close to a power-law for 4 decades. However, it has a cut-off due to the limited number of blocks currently in the blockchain. Also, specific spending delays are over-represented, like the ones around 1000 blocks.

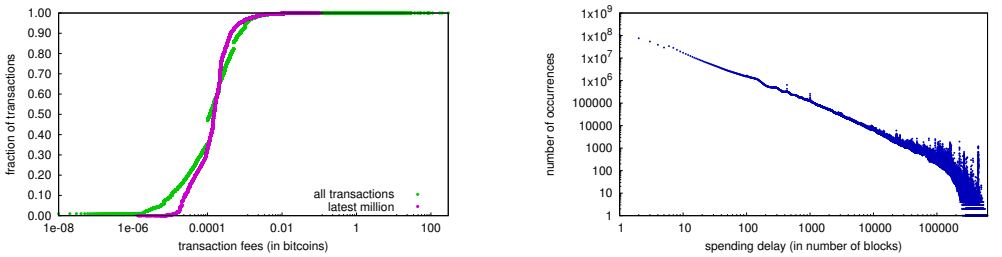

Figure 7: **Left: transaction fees**, *i.e.* difference between transaction output and input amounts (in bitcoins) for the whole dataset and for the latest million transactions. We display the cumulative distributions: for each amount x on the horizontal axis, we display a point indicating the fraction of transactions with fees (vertical axis) lower than this value. **Right: spending delay** distribution (log-log scales). The delay is the number of blocks issued between the time at which bitcoins are received as a transaction output, and the time at which they are spend as a transaction input.

Such statistics are easy to obtain with our dataset and methodology. All computations above take a few dozens of minutes from the distilled dataset and need a few GB of central memory only. We present a more advanced application in next section.

4 Application: address clustering

Transactions discussed above link input addresses to output addresses, but it often makes more sense to study bitcoin flows between individuals, companies, institutions, and other social entities. To do so, one has to **build clusters of addresses** that arguably belong to the same entity, and then observe transactions between these address clusters.

Many heuristics exist to build such clusters [14]. The most classical one is the *input-based heuristic*; it assumes that all input addresses of a transaction belong to the same cluster. Indeed, this transaction accesses the bitcoins associated to these addresses, and so it is reasonable to assume that a same entity owns them. We focus on this heuristic here for illustration purpose, but all the following applies the other heuristics.

Address clustering heuristics are mitigated by the use of various obfuscation and optimization methods [18]. Several of them, like for instance mixing, induce transactions with many and unrelated input addresses. Such transactions not only mistaken the input-based heuristic by erroneously indicating that their input addresses are related; they merge the clusters legitimately containing these addresses and so may have a dramatic impact on obtained clusters.

In order to illustrate the use of our dataset and the power brought by its pre-processing, we show here how to explore the relevance of this heuristic, and the impact of such transactions on its results.

To do so, we consider two key metrics: the number of obtained clusters and the size of the largest one. We compute these metrics when one considers only transactions with at most K_{in} inputs and K_{out} outputs. Observing how they vary with K_{in} and K_{out} shows the impact of suspicious transactions (the ones with large numbers of inputs and/or outputs) on obtained clusters. Indeed, unwanted transactions merge clusters of addresses, and so they increase their size and decrease their number.

Cluster computations usually rely on a graph in which nodes are addresses and two addresses are linked if they appear as input of a same transaction. Then, the wanted clusters are the connected components of this graph. However, not all links are needed: it is sufficient to ensure that all input addresses of each transaction are reachable from each other in the graph. For instance, linking the first input address of a transaction to all its other input addresses is sufficient; linking the first to the second, the second to the third, and so on, also works. Choosing between all appropriate solutions may have a marginal impact on the speed of cluster computations, but in all cases $k_{in} - 1$ links are sufficient, for any transaction with k_{in} input addresses.

Building this graph may be done by parsing the distilled data and writing as output the wanted links between address identifiers, $k_{in} - 1$ per transaction with k_{in} outputs. Then, these links are sorted to remove duplicates, and the graph is loaded into central memory to compute its connected component, in time and space linear with its number of links.

However, **a union-find approach** [17] is faster, more compact, and simpler. It uses the same list of links, but it does not need to sort it, and it only stores in central memory an array of all input address identifiers. It then needs a constant amortized time per link in the list

to *find* the components of its nodes and perform their *union* if needed. The overall time cost therefore is linear in the number of links in the list, which itself is linear in the total number of transaction inputs. Our address indexes are numbers from 0 to $n-1$ where n is the total number of addresses. Therefore, this method only needs to store n integers (of value at most n) in central memory.

Notice however that we investigate here the impact on obtained clusters of the considered maximal numbers K_{in} and K_{out} of input and output addresses. We therefore need to compute clusters for various values of these parameters. In order to reduce the redundancy of these computations, we perform them for all values of K_{in} in a row: we sort the link list according to the number of input addresses of the corresponding transaction, and output obtained information when the number of input addresses grows. We repeat this for all meaningful value K_{out} of the maximal number of output addresses.

In summary, we proceed as follows: we parse the distilled data and consider each transaction; for its k_{in} inputs, we list the $k_{in}-1$ links between the index of the first input address and all the others,together with k_{in} and its number of outputs k_{out}; we sort this list according to the k_{in} field in order to have all links in increasing transaction input number; then, for any given maximal number of output addresses K_{out} we perform union-find based on this sorted list by skipping lines with $k_{out} > K_{out}$ and, whenever the k_{in} field grows, we output k_{in}, the current number of clusters, and the current maximal cluster size. This gives the wanted metrics as a function of the maximal number of input addresses K_{in}, for the considered maximal number of output addresses K_{out}. Our implementation [11] obtains the results in a few dozens of minutes only.

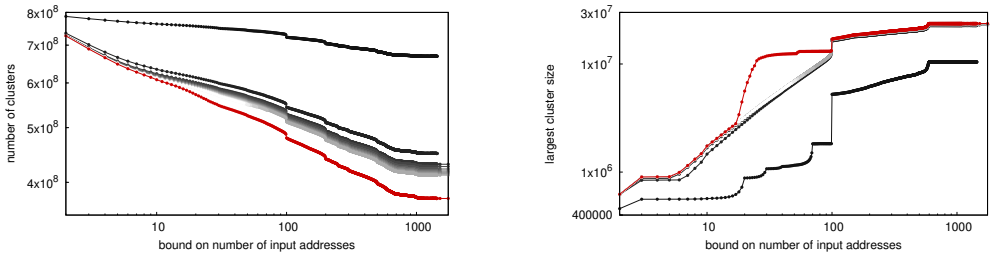

Figure 8: **Cluster metrics:** left: number of clusters; right: maximal cluster size. Each metric is displayed in log scale on the vertical axis, as a function of the maximal number K_{in} of input addresses of considered transactions, on the horizontal axis (log scale). We display the plots obtained for each value of K_{out} from 1 to 15, as well as the one obtained with no limitation on the number of outputs. The number of clusters (left) decreases with the number of considered transactions, therefore the bottommost plot (in red) is the one for any output number. Conversely, the maximal cluster size (right) increases with the number of considered transactions, therefore the topmost plot (in red) is the one for any output number.

Figure 8 displays the obtained plots. As expected, they show that using transactions with larger numbers of inputs and/or outputs has a strong impact on detected clusters (both their size and number). There is a clear difference between the partitions into clusters obtained when the maximal number of inputs K_{in} if below 10 or 20, and the ones obtained for larger values of K_{in}. Likewise, although we display plots for values of K_{out} up to 15, they are still very different from the ones obtained with no limit on the number of outputs.

Deepening these observations is out of the scope of this paper, but we can already conclude

that transactions with more than a dozen inputs or outputs have a strong impact on the input-based heuristic; they certainly play a specific role and should be handled with care. To go further, one may observe cluster size distributions in the same way, investigate other heuristics, and refine them. For instance one may consider weighted graphs between addresses, where the weight may represent the number of times addresses co-occur as input addresses, or other measures of heuristics reliability. This would not significantly change the procedure above.

Up to our knowledge, this is the first time that such plots are observed. **Certainly, the lack of available and indexed data was the limiting factor for such investigations.** As illustrated above, our dataset makes them much easier and faster to obtain than before.

5 Related work

Bitcoin is at the core of an intense research activity. A comprehensive review being out of reach, we focus here on data collection and management, with an emphasis on a few key contributions.

Many companies store their own copy of the bitcoin blockchain and provide partial access to it through dedicated web forms, APIs and/or CSV files. Their data is stored in local databases, and they provide additional information like charts, top users, address clusters, mixer lists, etc. Queries based on `txid`s and addresses are in general possible, as well as browsing between transactions.

For instance, some focus on user-friendly browsing [2, 4], with *BlockStream* providing open-source tools. The *Sphere 10 Software* company proposes a SQL-based approach [10]. *WalletExplorer* and *Chainalysis* emphasize identification of services, mostly based on manual investigation [5, 12]. *Blockchair* handles a wide variety of crypto-currencies [3], *OXT* has a particularly ideological policy [9], while *Kaiko* targets business applications [8].

Such data providers have two important strengths: a user-friendly access to bitcoin data, and additional information to help analysis. However, they provide very partial views: they focus on some aspects of the data, and large-scale computations are not efficient through such interfaces. Therefore, **researchers often conduct their own data collection, and provide the obtained datasets.**

The most advanced such dataset [22] probably is the one provided by Kondor et al [20,21]. The authors patch the bitcoin software in order to record transaction information. Then, they compute identifiers similar to ours, and perform address clustering. They provide data collection and analysis tool source code, as well as the data up to February 2020.

Other authors design and provide data collection and management frameworks for bitcoin blockchain.

An important contribution is the *Abe* free blockchain browser [1], that decodes binary blockchain files, stores data into a database, and provides query services. Others use Neo4j graph databases, like [6] that collects data by parsing the binary files, or [7] that uses parallel RPC calls, like us. Likewise, [13] proposes an open source Scala library using a local database [13] for efficient high-level analytics. Some frameworks target specific challenges, like graph analytics in [23,24], that use a cluster for data collection (custom C++ bitcoin data decoding) and analysis (Neo4j database). More recently, [26] target user and transaction profiling, using a Spark-based tool.

These frameworks based on general-purpose databases are outperformed by dedicated, highly optimized in-memory approaches. In particular, *BlockSci* [19] provides a very efficient

infrastructure to handle bitcoin (and other) blockchain data. It stores them in central memory and collects it using binary file decoding and RPC. It is used for instance in [27]. *DataChain* [28] is another example; it provides a lightweight, flexible and interoperable framework for high-level queries to blockchains.

Most of these works provide source code of their collection and analysis tools, and/or the datasets they obtain. It must be clear however that, because of the continuous evolution of bitcoin technologies, maintaining up-to-date tools and datasets is challenging; **many of them are outdated**. This is particularly true for methods that decode the bitcoin binary files.

In addition, **these tools have non-trivial requirements**: some need to compile complex tools, many rely on local database systems that require terabytes of disk space, others have huge central memory needs, etc.

Last but not least, to the best of our knowledge, **no available dataset provides the full information available in bitcoin blockchain**: they focus on parts of the data and only decode these parts; and/or they assume that some information is irrelevant and discard it. Even the Kondor et al dataset [22], which seems to be the most complete one, does not contain all blockchain data. In addition, such datasets often result from quite complex and poorly documented pre-processing steps, like data cleaning or clustering, and only the results of these pre-processings are provided. This may help for some dataset uses, but this may be a limitation for others; in all cases, this raises reproducibility and interpretation concerns.

6 Conclusion and perspectives

Our work provides a comprehensive view of bitcoin blockchain by capturing absolutely all data it contains. In contrast with previous works, it relies only on the simplest and most standard tools. It only requires a few hundred GB of disk space, including indexing. It is then easy to obtain very compact extractions of interest, and the indexes make them very convenient for a wide variety of studies. We give a thorough description of our procedure, with fully detailed application cases and documented code [11].

Our collection procedure may be improved in several ways. The following are particularly appealing: a dataset updating function, ideally in real-time; a parallel RPC procedure to reduce data collection time; querying pre-existing bitcoin nodes to avoid setting up our own node; as well as more advanced decoding of addresses and further analysis of ill-formed/non-standard transactions.

The **data pre-processing may also be improved**. One may split the dataset into several (compressed) files, and store the position of blocks or transactions (or other entities of interest) in these files. Browsing the dataset in non-sequential ways would then be easy, while keeping the representation compact. Further improvements may use database structures like B-trees, or even database libraries. This would bring our contribution closer to database-oriented solutions, though, with their advantages and drawbacks.

Acknowledgements. *This work is funded in part by the ANR (French National Agency of Research) under the FiT LabCom grant.*

References

[1] *Abe* free blockchain browser. `https://github.com/bitcoin-abe/bitcoin-abe`.

[2] *Blockchain company*. https://www.blockchain.com/explorer.

[3] *Blockchair*. https://blockchair.com.

[4] *BlockStream*. https://blockstream.info/.

[5] *Chainalysis*. https://www.chainalysis.com/.

[6] How to import the bitcoin blockchain into Neo4j. https://neo4j.com/blog/import-bitcoin-blockchain-neo4j.

[7] How to load bitcoin into neo4j in one day. https://medium.com/tokenanalyst.

[8] *Kaiko*. https://www.kaiko.com.

[9] *OXT (Other/Open eXploration Tool)*. https://oxt.me.

[10] *Sphere 10 Software*. http://blockchainsql.io.

[11] Supplementary material. http://bitcoin.complexnetworks.fr.

[12] *WalletExplorer*. https://www.walletexplorer.com.

[13] Massimo Bartoletti, Stefano Lande, Livio Pompianu, and Andrea Bracciali. A general framework for blockchain analytics. In *First Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, 2017.

[14] Rémy Cazabet, Rym Baccour, and Matthieu Latapy. Tracking bitcoin users activity using community detection on a network of weak signals. In *6th Conference on Complex Networks and Their Applications*, 2017.

[15] Bitcoin Community. bitcoin-core getblock rpc. https://bitcoincore.org/en/doc/0.21.0/rpc/blockchain/getblock.

[16] Bitcoin Community. bitcoin-core-0.21.0. 2020. https://bitcoincore.org/bin/bitcoin-core-0.21.0.

[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press.

[18] Younggee Hong, Hyunsoo Kwon, Jihwan Lee, and Junbeom Hur. A practical de-mixing algorithm for bitcoin mixing services. In *2nd ACM Workshop on Blockchains, Cryptocurrencies, and Contracts (BCC)*, 2018.

[19] Harry Kalodner, Malte Möser, Kevin Lee, Steven Goldfeder, Martin Plattner, Alishah Chator, and Arvind Narayanan. Blocksci: Design and applications of a blockchain analysis platform. In *29th USENIX Security Symposium*, 2020.

[20] Dániel Kondor, István Csabai, János Szüle, Márton Pósfai, and Gábor Vattay. Inferring the interplay between network structure and market effects in bitcoin. *New Journal of Physics*, 2014.

[21] Dániel Kondor, Márton Pósfai, István Csabai, and Gábor Vattay. Do the rich get richer? an empirical analysis of the bitcoin transaction network. *PLoS ONE 9(2)*, 2014.

[22] Dániel Kondor, Márton Pósfai, István Csabai, and Gábor Vattay. *Bitcoin Transaction Network*, dryad, dataset. `https://doi.org/10.5061/dryad.qz612jmcf`, 2021.

[23] Dan McGinn, Douglas McIlwraith, and Yike Guo. Data from: Towards open data blockchain analytics: a bitcoin perspective, dryad, dataset. `https://doi.org/10.5061/dryad.h9r0p65`, 2018.

[24] Dan McGinn, Douglas McIlwraith, and Yike Guo. Towards open data blockchain analytics: a bitcoin perspective. *Royal Society Open Science*, 2018.

[25] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008. `https://bitcoin.org/bitcoin.pdf`.

[26] Raj Sanjay Shah, Ashutosh Bhatia, Atith Gandhi, and Shray Mathur. Bitcoin data analytics: Scalable techniques for transaction clustering and embedding generation. In *13th IEEE COMSNETS*, 2021.

[27] Aman Sharma and Ashutosh Bhatia. Bitcoin's blockchain data analytics: A graph theoretic perspective, 2020.

[28] Demetris Trihinas. Datachain: A query framework for blockchains. In *11th ACM International Conference on Management of Digital EcoSystems*, 2019.